MASTER EN SCIENCES INFORMATIQUES

Recherche heuristique encapsulée

Amezouj, Karim

Award date:
2009

Facultés Universitaires
Notre-Dame de la Paix, Namur
Institut d'Informatique.
Année académique 2008-2009

RECHERCHE HEURISTIQUE ENCAPSULÉE

KARIM AMEZOUJ

Mémoire présenté en vue de l'obtention du grade de licencié en informatique.

RÉSUMÉ

Dans ce document, nous présentons une étude pour réaliser la modélisation de problèmes d'optimisation sous contraintes et leur résolution soit par des opérateurs déclaratifs pour la recherche heuristique, soit par des opérateurs hybrides utilisant à la fois la programmation sous contraintes pour réduire l'espace de recherche et les opérateurs déclaratifs. Le prototypage sera développé en Curry, langage de programmation fonctionnelle logique. Le but poursuivi étant de montrer s'il y a indépendance de l'expression déclarative du problème de ses techniques de résolution ainsi que la possibilité de réutilisation des opérateurs ou de la modélisation d'un problème.

Mots-clés : contrainte, CSP, heuristique, modélisation, Curry, programmation fonctionnelle, programmation logique.

ABSTRACT

In this document we present a study about constraint solving and optimization problem modeling and their resolutions either by declarative operators for the heuristics research, either by hybrids operators using simultaneously the constraint programming to reduce the scope of search and the declarative operators. The prototype will be written in Curry, a functional logical programming language. The objective is to look for a possible independence between the formulation of a declarative problem and the resolution techniques used and also the possibility to reuse the operators or the modeling of a problem.

Keywords : constraint, CSP, heuristic, modeling, Curry, functional programming, logical programming.

Je tiens à remercier Pierre-Yves Schobbens, promoteur de ce mémoire, pour sa compréhension, son aide, ses conseils et son temps.

Ma reconnaissance s'adresse aussi à Isabelle Linden et Jean Paul Leclercq pour ces années passées en leur compagnie.

Pour finir, un merci à mes parents, mes deux enfants, Estelle et Florian, ainsi qu'à toutes les personnes qui m'ont poussé lorsque j'en avais besoin.

TABLE DES MATIÈRES

TABLE DES FIGURES

INTRODUCTION

A l'heure actuelle ; le monde cherche à satisfaire trois grands principes :

- la simplicité ;
- la réutilisation ;
- la rapidité.

Dans le cadre de cette étude, nous allons essayer de rencontrer ces trois principes dans le monde de la résolution de problèmes complexes.

La simplicité, premier pilier, est lié au paradigme de programmation choisi, nous allons tenter d'utiliser un langage de type FLP qui a comme caractéristique l'approche déclarative, ce qui permet de décrire les actions et la logique dont nous pensons qu'il serait un chemin vers une solution, en lieu et place d'une suite de commandes qui est une approche impérative. Le langage retenu pour cette étude est Curry. Nous introduirons certaines de ses caractéristiques dans le chapitre 2.

La réutilisation sera le second pilier. Nous poursuivrons un but d'indépendance entre la modélisation d'un problème et la manière dont nous chercherons à le solutionner. Ceci nous obligera à travailler sur deux sous-études qui seront, d'un côté, la modélisation la plus générique possible d'un problème et, de l'autre, la création d'opérateurs ayant suffisamment de paramètres en entrée que pour satisfaire le plus grand nombre d'utilisations potentielles.

La rapidité, troisième pilier, sera rencontrée par les méthodes choisies et testées. Nous allons passer en revue une série de métaheuristiques. Ce passage en revue n'a pas pour but l'étude complète de la métaheuristique mais la compréhension de son fonctionnement afin de pouvoir décrire celui-ci dans un mode déclaratif. Nous nous essayerons aussi à quelques études de métaheuristiques hybrides.

La structure de l'étude qui suit est de complexité croissante. Les chapitres 1 à 4 sont des rappels ou des introductions à des concepts et théories.

- Le chapitre 1 nous permettra d'avoir un vocable commun, ce qui apportera une vision commune sur ce qu'est un problème complexe et comment nous pourrions approcher une solution.
- Le chapitre 2 nous fera rentrer de plein pied dans le monde FLP et dans certaines de ses caractéristiques natives.
- Les chapitres 3 et 4 approcheront de manière précise des méthodes de résolution. Retenons la notion de filtrage pour le 3 et les métaheuristiques et leur possible convergence vers des programmes à mémoire adaptative pour le 4.

Les chapitres 5 et 6 sont les deux sous-études. Ils sont basés sur les chapitres précédents et remontent dans leur conclusion respective les points d'attention rencontrés.

Remarque : Afin de cerner plus facilement les idées principales d'un chapitre, elles seront reprises dans une carte heuristique en tête de chapitre. La carte reprendra généralement uniquement les points de structure mais, si un concept important existe dans le chapitre, il y apparaîtra aussi.

La méthode qui a été utilisée pour réaliser cette étude est analogue à celle utilisée pour structurer ce document. En premier lieu, un travail de compréhension de la demande, se traduisant par une série de concepts, eux-mêmes demandant des recherches, et par l'installation d'un environnement de test. Ensuite, fort de ces acquis, l'étape suivante a été de se poser la question de l'implémentation. Malgré l'évidence que la modélisation d'un problème est à réaliser avant de pouvoir résoudre ce dernier, la demande d'indépendance a permis de travailler sur les deux sujets en parallèle. Cette approche parallèle s'est vite transformée en boucle de validation entre les deux sous-études. La validation étant, d'un côté, l'indépendance mais, de l'autre, les dépendances des deux sous-études.

CHAPITRE 1 : CONCEPTS

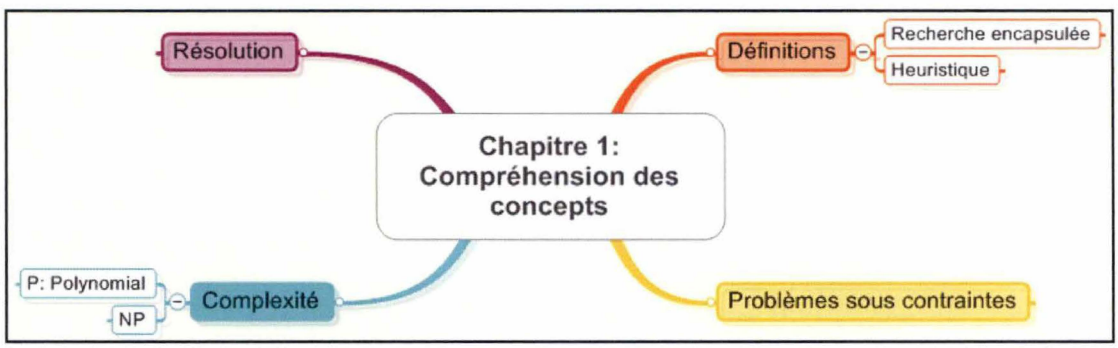


Figure 1: Carte heuristique - Compréhension des concepts

Dans ce premier chapitre nous allons définir et introduire le cadre de notre étude. Les points qui seront développés sont les heuristiques et la recherche encapsulée. Le langage Curry qui sera utilisé pour l'implémentation sera présenté dans le chapitre 2.

1. COMMENÇONS PAR UN PEU DE VOCABULAIRE

Définition d'heuristique

petit Robert :

> *1° Adj. Qui sert à la découverte.*
>
> *2° N.f. Partie de la science qui a pour objet la découverte des faits.*

Recherche encapsulée :

Cette étude est basée sur le langage Curry[Chapitre 2] qui est un langage de programmation fonctionnel logique. Ce type de langage possède deux caractéristiques principales qui les différencient des autres langages de programmation :

1. effectuer des calculs avec une partie des données
2. rechercher les solutions d'un objectif (goal)

Le fait d'avoir la possibilité de réaliser cette recherche peut être considéré dans beaucoup de cas comme pratique mais elle n'est pas la panacée universelle.

Dans des cas pratiques, cette recherche native – qui est usuellement une recherche en profondeur d'abord au travers d'un backtracking global – peut ne pas être à la hauteur pour rechercher des solutions. Pour répondre à ces cas pratiques, et ne plus laisser la recherche native se dérouler sans contrôle, on dit que l'on « encapsule » la recherche.

Cette action d'encapsulation peut être comprise comme le fait que le programme va se dérouler dans une sorte d'environnement. Cet environnement contrôlera les choix faits par le programme et quand ils sont faits. Cet environnement protègera le reste de l'application des effets de ces choix.

La recherche encapsulée est importante pour :

1. la modularité : plusieurs programmes peuvent fonctionner en même temps avec les mêmes ressources sans interférer.
2. la composition : une recherche encapsulée peut elle-même contenir une recherche encapsulée.

Allons plus loin pour commencer à cerner ce qu'est une heuristiqu[HGH][Chapitre 4] :

Aujourd'hui, nous sommes de plus en plus confrontés à des problèmes dits NP-difficiles. Ces problèmes sont par exemple : le routage réseaux, la planification, la recherche d'un chemin optimal sur son GPS. Nous sommes en fait en face de problèmes d'optimisation combinatoire. Cette optimisation combinatoire occupe une place importante en mathématiques discrètes, en recherche opérationnelle et aussi en informatique. La place qu'elle occupe se justifie par deux raisons :

1. la difficulté des problèmes d'optimisation ;

2. la modélisation d'un grand nombre de problèmes pratiques sous la forme d'un problème d'optimisation combinatoire.

Nous pouvons constater que, s'il est facile de définir notre problème sous une forme d'optimisation combinatoire, il n'en est pas de même pour sa résolution.
Cette difficulté de résolution que nous pouvons traduire par la non existence (du moins pour le moment) d'une solution algorithmique efficace valable pour toutes les données peut se résumer par le qualificatif de NP-difficile.

Nous savons que nous avons affaire à une classe de problèmes que nous rencontrons régulièrement et donc de nombreuses méthodes de résolution ont été développées à la fois en recherche opérationnelle et à la fois en intelligence artificielle.
De prime abord, nous pouvons classer ces méthodes en deux grandes classes : les méthodes exactes et les méthodes approchées.
Les méthodes exactes sont complètes dans la recherche et donc elles garantissent la « complétude » de la résolution. Nous pourrions dire que la solution, si solution il y a, est l'optimum et on peut le prouver.
Les méthodes approchées sont à contrario incomplètes, nous perdons la preuve que la solution est l'optimum et nous ne sommes même pas certains d'avoir cet optimum mais nous nous rapprochons de lui avec une plus grande efficacité.

Dans une vision simplifiée du fonctionnement d'une méthode exacte, nous pouvons imaginer que la méthode énumère, souvent de manière implicite, l'ensemble des solutions de l'espace de recherche. Des améliorations existent pour rendre l'énumération des solutions plus performantes. Ces améliorations consistent par exemple à détecter le plus tôt possible les échecs ou à introduire des heuristiques permettant d'orienter les différents choix.
Exemple : les algorithmes de retour arrière (backward chaining) sont des méthodes exactes.
Le problème principal des méthodes exactes est leur dépendance à la taille du problème. Si nous avons à résoudre un problème que nous pouvons qualifier de taille raisonnable, nous pouvons utiliser avec succès une méthode exacte. Par contre, comme le temps de calcul nécessaire pour trouver une solution a tendance à augmenter exponentiellement avec la taille

du problème, les méthodes exactes ne peuvent plus être prises en considération dans les problèmes de taille importante.

Quelle approche pourrions-nous avoir face à ces problèmes qui demandent un temps de calcul si long ? Ce terme approche est on ne peut plus juste car ce sont les méthodes dites approchées qui peuvent représenter une alternative intéressante pour traiter les problèmes d'optimisation de grande taille. Toutefois, nous ne devons pas oublier le fait que ces méthodes sont incomplètes et donc que nous ne serons pas certains de trouver l'optimum mais que le fait de s'en rapprocher est suffisant.

Nous nous rendons compte que la recherche de solution pour un problème est fonction du problème, du temps que l'on est prêt à consacrer à sa résolution et de la qualité recherchée de la solution. En fonction de ces trois critères, nous pouvons passer d'une méthode exacte à une méthode approchée. Ces méthodes approchées portent le nom de métaheuristique.
Nous pouvons définir une métaheuristique comme étant un ensemble de concepts fondamentaux qui, ensemble, permettent la création de méthodes heuristiques capables de résoudre un problème d'optimisation. Ceci veut dire que les métaheuristiques sont paramétrables et de fait applicables à une large classe de problèmes.

Nous pouvons déjà introduire à ce niveau une première classification de ces métaheuristiques. Nous avons les méthodes dites de voisinage telles que le recuit simulé ou la recherche tabou et les algorithmes dits évolutifs tels que les algorithmes génétiques. En utilisant ces approches pour résoudre des problèmes, nous allons pouvoir faire face à des problèmes de plus grande taille.

Nous allons introduire dans la suite de ce chapitre un ensemble de concepts liés aux problèmes sous contraintes, à la complexité d'un problème et, enfin, une proposition de classification de méthode de recherche dont le nom est connu tel que A*, recherche tabou, algorithme génétique.

2. PROBLÈME D'AFFECTATION SOUS CONTRAINTES

Dans notre étude nous allons parler de contraintes, de problèmes sous contraintes, d'optimisation de problèmes sous contraintes. Nous allons introduire ce concept qui sera un des piliers de notre travail. Les problèmes sous contraintes peuvent être définis comme une classe de problèmes dont une solution peut être décrite explicitement par une affectation de valeurs à l'ensemble des variables du problème.

Nous pouvons proposer une classification des types de contraintes :

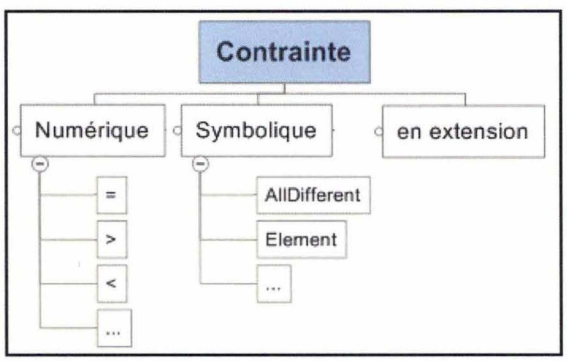


Figure 2 : Classification des contraintes

Les contraintes numériques sont de la forme d'équations ou d'inéquations entre les différentes variables.
Les contraintes symboliques sont une qualité associée à un ensemble de contraintes. Si nous prenons AllDifferent dans le cas de domaine numérique, nous pourrions l'exprimer par un ensemble d'inéquations.

Etant donné un ensemble fini de variables V = {V1,V2, .., Vn} et un ensemble de domaines finis associés D = {D1,D2, .. , Dn}, une solution potentielle du problème consiste à choisir pour chaque variable Vi $\in$ V une valeur v $\in$ Di. L'ensemble S des solutions potentielles est donc représenté par le produit cartésien D1'xD2'x..xDn' des domaines.

On dispose d'un ensemble de contraintes C = {C1,C2,..,Cp} chaque contrainte Cj ∈ C est une relation sur un sous-ensemble Vj' de V. Ce sous-ensemble spécifie quelles combinaisons de valeurs sont compatibles pour les variables de V'j.

Cette classe de problèmes où nous recherchons à satisfaire des contraintes porte le nom de CSP (constraint satisfaction problems). Nous avons aussi la possibilité d'avoir un élément à optimiser et cela donne naissance à un problème d'optimisation sous contraintes : CSOP (constraint satisfaction and optimization problem).[FrWa]

Etant donné un triplet <V,D,C> : <variable, domaine, contrainte>

- le problème de type CSP consiste à trouver une assignation qui satisfasse toutes les contraintes. Un sous-ensemble de problèmes CSP est la classe des problèmes MCSP qui, eux, recherchent à satisfaire un maximum de contraintes et non plus la totalité des contraintes. Cette classe permet de chercher une solution pour un problème dont certaines contraintes ne pourraient pas être satisfaites en même temps.
- Le problème de type CSOP reprend la même approche mais en fait nous avons affaire à un quadruplet de la forme <V,D,C,f> : <variables, domaines, contraintes, fonction> et la solution recherchée doit satisfaire l'ensemble des contraintes mais elle doit également optimiser la fonction f : S → R.

Ces problèmes de type CSP/CSOP ou dérivés se rencontrent très régulièrement, par exemple dans :

- l'affectation de ressources ;
- la planification ;
- l'ordonnancement ;
- nous pouvons aussi inclure des exemples plus didactiques comme le sudoku, la coloration de graphes, le problème de N queen.

3. Complexité

Nous avons introduit une différence basée sur la taille du problème pour dire qu'une méthode exacte est réaliste ou qu'il serait souhaitable de passer à une méthode approchée. Comme nous le montrerons dans la suite de cette étude, le choix et la manière de résoudre un problème sont laissés à la discrétion de la personne cherchant une solution mais sur quel critère peut-elle ou doit-elle se baser pour faire son choix ?

Nous allons essayer de cerner la complexité d'un problème par le temps nécessaire à sa résolution. Voici les ordres de grandeurs théoriques pour des algorithmes

- O(1) complexité constante (indépendante de la taille de la donnée)
- O(log(n)) complexité logarithmique
- O(n) complexité linéaire
- O(n.log(n)) complexité quasi-linéaire
- $O(n^2)$ complexité quadratique
- $O(n^3)$ complexité cubique
- **$O(n^p)$ complexité polynomiale**
- $O(n^{\log(n)})$ complexité quasi-polynomiale
- $O(2^n)$ complexité exponentielle
- O(n!) complexité factorielle

Nous avons aussi des familles de classes de complexité en temps et en espace. Le temps étant le temps estimé nécessaire pour la résolution et l'espace étant l'espace mémoire nécessaire lors de la résolution :

- TIME(t(n))
- NTIME(t(n))
- SPACE(s(n))
- NSPACE(s(n))

De ces classifications, nous arrivons à des classes de complexité :

- Classes L et NL

- **Classe P** : Ce sont les problèmes pouvant être résolus sur une machine déterministe en un temps polynomial par rapport à la taille de la donnée. On qualifie le problème de polynomial quand il s'agit d'un problème de complexité $O(n^k)$ pour un certain k
- **Classe NP et classe Co-NP (complémentaire de NP)** : Ce sont les problèmes non-déterministes polynomiaux, qui peuvent être résolus sur une machine non-déterministe en un temps polynomial. Ceci veut dire que cette classe englobe les problèmes qui, pour être résolus, demandent que l'on énumère l'ensemble des solutions possibles et que l'on teste ces solutions à l'aide d'un algorithme polynomial.
- Classe PSPACE
- Classe EXPTIME
- Classe NC (Nick's Class)

Les deux classes en gras (P et NP) sont les classes de problèmes qui nous intéresseront dans la suite de cette étude.

Nous allons donc nous adresser soit à des problèmes qui peuvent être résolus par un algorithme polynomial soit à des problèmes dont nous devrions énumérer l'ensemble des solutions et ensuite les valider par un algorithme polynomial. Pour ces derniers problèmes, il est évident que le temps de recherche peut être long et que donc nous passerons d'une recherche de la solution à une approximation de la solution par une méthode heuristique.

4. Méthode de résolution

La littérature nous renseigne un nombre important de méthodes de résolution pour nos problèmes d'optimisation combinatoire. Nous n'allons pas en rédiger une liste exhaustive mais en présenter certaines et surtout essayer de les classer. Ce classement sera fonction des points qui ont été présentés dans cette introduction. Par la suite, nous referons un classement qui sera basé sur d'autres arguments.

Nous allons établir une première classification en fonction de la méthode poursuivie par l'algorithme : exact ou approché.

La seconde classification sera effectuée sur l'approche proposée par la méthode :

- Construction
- Méthode de voisinage
- Évolution

Approche de construction : une méthode de construction construit pas à pas une solution.

Approche de voisinage : la méthode débute avec une configuration initiale et réalise un processus itératif qui consiste à remplacer la configuration courante par l'un de ses voisins en tenant compte de la fonction de coût. Ce processus s'arrête et retourne la meilleure configuration trouvée quand la condition d'arrêt est rencontrée.

Approche évolutive : le but est de faire évoluer une population pour tendre vers la solution.

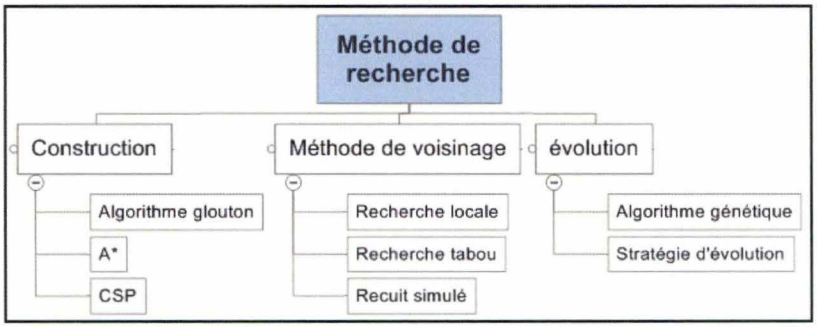


Figure 3 : Classification des méthodes de recherche

5. LE PROBLÈME QU'EST LA RECHERCHE DE SOLUTION DE PROBLÈMES

L'approche courante pour la résolution de problèmes est d'en cerner la complexité et de chercher sa Solution. Nous imaginons qu'à la sortie d'une application cherchant à résoudre un problème, nous trouverons une réponse. Or ici, aucun des opérateurs ne nous certifie trouver une solution, il ne peut que nous aider à avancer vers un embryon de solution, une solution ou alors nous montrer que notre problème n'a pas de solution. Ceci indique donc

clairement que d'un problème, nous arrivons à un nouveau problème soit plus simple, soit avec une spécification plus claire, soit enfin à un point d'arrêt défini tel que : découverte de la solution, l'absence de solution, le temps écoulé, la non évolution du processus, l'évolution trop lente, ...

Ces différents points vont être abordés et détaillés dans les chapitres suivants.

CHAPITRE 2 : CURRY, LANGAGE FONCTIONNEL LOGIQUE

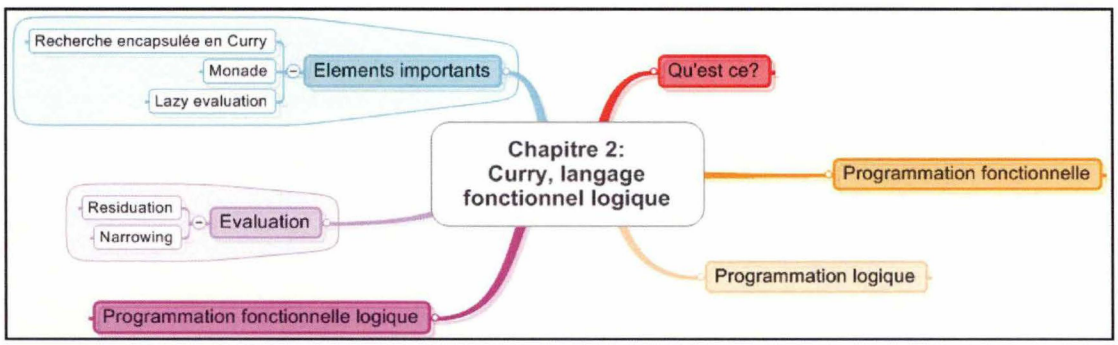


Figure 4 : Carte heuristique - Curry

1. QU'EST-CE ?

Curry[1] est un langage multi-paradigmes[2] fonctionnel, logique et concurrent. Il est donc un hybride entre la programmation fonctionnelle telle que Haskell et logique telle que Prolog.
De part ses origines, il peut être considéré comme une extension du langage fonctionnel Haskell. Un des buts qu'il poursuit est de proposer un standard pour les langages fonctionnels logiques. Plusieurs implémentations sont disponibles : The Münster Curry Compiler, PACKS, Curry2Prolog, FLVM, Sloth et Zinc. Cette liste provenant du site officiel[www.curry-language.org] reprend les implémentations les plus récentes.
Nous utiliserons pour nos prototypages l'implémentation PACKS (the Portland Aachen Kiel Curry System).

[1] Haskell Brooks Curry (né le 12 septembre 1900 et décédé le 1er septembre 1982) était un mathématicien et logicien américain. Ses travaux ont posé les bases de la programmation fonctionnelle. Curry est principalement connu pour son travail sur la logique combinatoire. Curry est également connu pour le paradoxe de Curry et pour la correspondance de Curry-Howard. Deux langages de programmation sont nommés en son hommage : Haskell et Curry.

[2] Un paradigme de programmation est un style fondamental de programmation informatique qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation (à comparer à la méthodologie, qui est une manière de résoudre des problèmes spécifiques de génie logiciel).

Comme nous l'avons indiqué, Curry provient de trois mondes et il propose d'intégrer les caractéristiques propres de ces trois mondes.

- Du monde de la programmation fonctionnelle
  - Nested expressions
  - Evaluation paresseuse
  - Fonctions d'ordre supérieur
- Du monde de la programmation logique
  - Les variables logiques
  - Les recherches built-in
- Du monde de la programmation concurrente
  - Evaluation concurrente de contraintes

Ceci donne un langage qui possède nativement des caractéristiques additionnelles en comparaison aux langages purs :

- Programmation fonctionnelle
  - Recherche
  - Calcul avec une information incomplète
- Programmation logique
  - Meilleure évaluation

Pour finir, Curry intègre également les principes les plus importants des langages fonctionnels logiques (FLP) la « residuation » et le « narrowing ». Ces deux points seront explicités, lorsque nous parlerons et détaillerons l'évaluation dans la programmation fonctionnelle logique[Chapitre 2.5.a et b]

2. Programmation fonctionnelle

La programmation fonctionnelle est un paradigme de programmation considérant le calcul en tant qu'évaluation de fonctions mathématiques.

Par contre, dans ce paradigme, la notion de changement d'état et de mutation de données n'existent pas.

La programmation fonctionnelle s'affranchit de façon radicale des effets secondaires en interdisant toute opération d'affectation.
Le premier langage fonctionnel fut Lisp qui donna naissance à Scheme ou Common Lisp. Ces premiers langages n'étaient pas ou peu typés au contraire de leurs successeurs comme ML ou Haskell qui sont quant à eux fortement typés.
Nous pouvons donner une caractéristique permettant de définir un langage fortement typé : la compilation ou l'exécution peuvent détecter des erreurs de typage.

Le paradigme fonctionnel n'utilisant pas la notion de changement d'état, il est logique que ce paradigme n'utilise pas de machine d'états pour décrire un programme.
Ce qui permet de décrire un programme est ce que nous pourrions décrire comme étant une succession de fonctions.
Chaque fonction est définie par :

- ses paramètres d'entrée
- un seul paramètre de sortie
- à un même n-uplet en entrée correspond une et une seule valeur de sortie

La conséquence de la correspondance n-uplet en entrée, valeur de sortie est que les fonctions n'introduisent pas des effets de bord.
Un des avantages des fonctions sans effet de bord est la facilité que l'on a à les tester de manière unitaire.

Un programme est donc une application, au sens mathématique, qui ne donne qu'un seul résultat pour chaque ensemble de valeurs en entrée.

Si nous pensons en paradigme impératif, cela est déstabilisant. Par contre, si nous faisons table rase de l'approche impérative, il est naturel de penser à un programme comme étant une succession de fonctions/fonctionnalités devant se dérouler les unes après les autres.

Nous allons dans la suite de cette étude proposer des pseudo-codes d'algorithmes connus qui sont sous une forme impérative. La question qui se pose est comment passer d'un mode impératif à un mode déclaratif.

En pratique, pour des raisons d'efficacité et du fait que certains algorithmes s'expriment aisément avec une machine d'états, certains langages fonctionnels autorisent la programmation impérative en permettant de spécifier que certaines variables soient assignables et donc cela peut introduire localement des effets de bord. Cette classe de langages fonctionnels est dite impure.

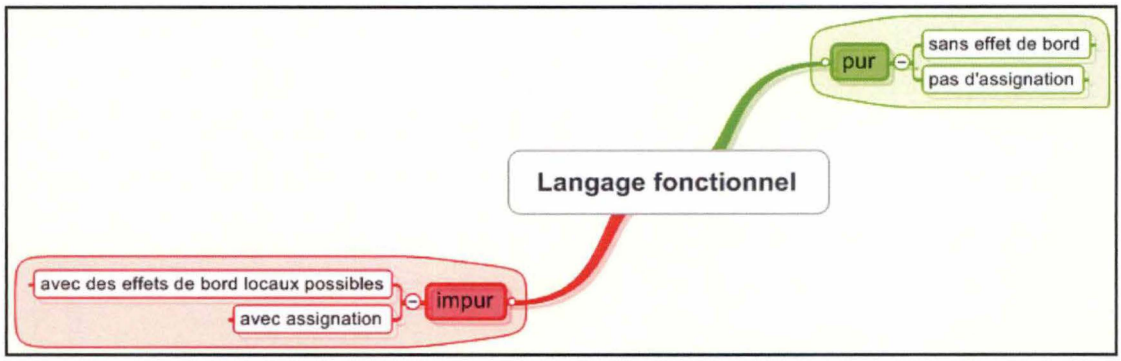


Figure 5 : Différence langage fonctionnel pur et impur

A contrario, les langages dits purement fonctionnels ne permettent pas la programmation impérative. Cela à pour conséquence que les effets de bord sont inexistants et qu'ils sont prêts à l'exécution concurrente de programmes.
Le typage est un point important pour les langages fonctionnels et ils possèdent des types et des structures de données de haut niveau : Int, Float, List, Function, …

La notion de boucle n'étant pas présente en programmation fonctionnelle, elle est remplacée par la notion de récursivité[Chapitre 6.b] (fait d'inclure l'appel d'une fonction dans sa propre définition). Pour arriver au même résultat après compilation qu'une boucle définie en programmation impérative, les langages fonctionnels utilisent une gestion de pile sophistiquée avec une récursivité dite terminale.

Une dernière caractéristique importante des langages fonctionnels est l'usage des fonctions d'ordre supérieur[Chapitre 5.6]. Une fonction est dite d'ordre supérieur lorsqu'elle peut prendre des fonctions comme argument et/ou retourner une fonction comme résultat.

3. Programmation logique

La programmation logique est un paradigme de programmation considérant le calcul en tant que processus de déduction ou de construction de preuves. Un programme logique consiste en un ensemble de faits élémentaires et des règles de logique associant des conséquences plus ou moins directes. Ces faits et ces règles sont exploités par un démonstrateur de théorème ou monteur d'inférence, en réaction à une question ou requête.

Cet ensemble faits et règles rend la programmation logique fondamentalement différente de la plupart des autres langages de programmation. La principale différence entre la programmation logique et les langages que nous pourrions considérer comme conventionnels est la nature déclarative de la logique du fait qu'elle s'attaque davantage au quoi qu'au comment. Un programme écrit en C par exemple, ne peut pas, en général, être compris sans prendre en compte des considérations opérationnelles. Ceci est dû au fait que ce programme C ne peut être compris sans connaître la manière dont il va être exécuté. En comparaison, la compréhension d'un programme logique peut être comprise sans penser à son exécution car les clauses sont lisibles et compréhensibles sans avoir à l'esprit leur exécution ou évaluation.

4. Programmation fonctionnelle logique

Après avoir défini la programmation fonctionnelle et la programmation logique, nous pouvons cerner ce qu'est la programmation fonctionnelle logique. Cette programmation tient des deux paradigmes, elle combine le meilleur des deux mondes pour nous proposer un nouveau paradigme de programmation :

- fonctions d'ordre supérieur ;
- I/O déclaratif ;
- contraintes concurrentes

Ce type de langage cherche à atteindre :

- exécution efficace des langages fonctionnels ;

- flexibilité de langages logiques ;
- éviter les caractéristiques de non déclaration de prolog

La création d'un langage fonctionnel logique peut se faire de deux manières :

- étendre un langage fonctionnel avec des caractéristiques propres aux langages logiques ;
- étendre un langage logique avec des caractéristiques propres aux langages fonctionnels.

Curry, langage utilisé dans notre étude est une extension du langage Haskell lui-même langage fonctionnel par des caractéristiques des langages logiques.

5. EVALUATION :

L'évaluation d'expressions contenant des variables logiques est un sujet important et délicat. Cette caractéristique est l'une des plus importantes des langages fonctionnels logiques. Il existe deux approches possibles pour traiter l'évaluation des expressions contenant des variables logiques : residuation et narrowing.

a. RESIDUATION

Supposons *e*, expression à évaluer, et v une variable faisant partie de l'expression *e*. Supposons que *e* ne peut pas être évaluée car la valeur de v est inconnue. La residuation met en attente l'évaluation de *e*. Si cela est possible, d'autres expressions f sont évaluées en espérant que ces évaluations vont à un moment ou un autre instancier v à une valeur. Si cela arrive et lorsque cela arrive, l'évaluation de *e* continue. Si l'expression f n'existe pas, l'évaluation de *e* se termine en erreur.

```
Prelude>    z==2+2 where z free
            Free variables in goal: z
            *** Goal suspended!
```

b. NARROWING

Une équation conditionnelle de la forme llc=r est éligible pour une réduction si sa condition c a été solutionnée. Une différentiation vis-à-vis des langages uniquement fonctionnels où les conditions sont uniquement évaluées en valeur booléenne, les langages fonctionnels logiques permettent la résolution de conditions en recherchant les valeurs pour les inconnues présentes dans la condition. Le concept de narrowing est utilisé pour résoudre ce genre de conditions.

Narrowing (étranglement) est un mécanisme où une variable est liée à une valeur choisie parmi l'ensemble des possibilités imposées par les contraintes. Chaque valeur potentielle est testée dans un certain ordre. Ce mécanisme est une extension de la programmation logique. Comme elle, le narrowing dans Curry permet de faire des recherches et de tester celles-ci mais elle permet en plus de générer des valeurs. De ce fait, nous pouvons considérer une fonction comme étant une relation. En effet sa valeur peut être calculée « dans les deux directions ».

Nous pouvons donc comparer le concept de narrowing à une réduction dans une gestion de graphe. Il existe une multitude de stratégies pour effectuer cette réduction. Le choix dans Curry a été l'implémentation de la stratégie « Needed narrowing » qui correspond à la stratégie « lazy strategy ».
En comparaison à la residuation, si e ne peut pas être évalué parce que la valeur de v est inconnue, le narrowing présume une valeur a pour v. Cette valeur présumée est sans contrainte si ce n'est que seules les valeurs qui rendent possible la continuation du calcul sont choisies. L'opération « = := », appelée constrained equality, est prédéfinie dans Curry. Cette opération est similaire à l'égalité booléenne avec deux grandes différences :

- la première différence est le type retourné par l'opération, la constrained egality, qui retourne le type Success. Ce type n'a pas de constructeur visible. Une expression de type Success peut soit réussir, soit rater. Il existe des opérations prédéfinies « success » et « failed » pour encoder les réussites ou échecs dans un programme.
- La seconde différence est que l'opération « = := » étrangle au lieu de « résider » donc :

```
Prelude>    z=:=2+2 where z free
            Free variables in goal: z
            Result: success
            Bindings:
            z=4 ?
```

6. Eléments importants utilisés :

a. Recherche encapsulée

La recherche globale, implémentée par backtracking, doit pouvoir être évitée dans certaines conditions (vouloir garder le contrôle de la recherche, calcul concurrent, ...). C'est pourquoi il est parfois nécessaire d'encapsuler la recherche, par exemple, les calculs non-déterministes dans des parties d'un large programme. Des calculs non-déterministes peuvent arriver lorsqu'une fonction doit être évaluée avec une variable libre à une position d'argument flexible. Dans ce cas, le calcul doit suivre différentes branches avec différentes contraintes appliquées à l'expression courante. Ceci à pour effet que l'expression est divisée en, au minimum, deux disjonctions. Pour donner le contrôle au programmeur sur les actions à prendre dans cette situation, Curry propose un opérateur primitif de recherche qui est la base pour implémenter des stratégies de recherche sophistiquées.

La recherche est utilisée pour trouver des solutions à des contraintes. La recherche est toujours initiée par une contrainte contenant une variable pour laquelle une solution devrait être calculée.
Dès lors que la variable peut être liée à différentes solutions dans différentes branches de disjonctions, elle doit être rendue abstraite.

De ce fait, un objectif de recherche a le type *a -> Success* où a est le type de la valeur que nous recherchons. En particulier, si *c* est une contrainte contenant une variable x et que nous sommes intéressés par une solution pour x, par exemple, les valeurs de x qui satisfont la

contrainte c, cela donne la forme suivante comme but de recherche correspondant (\x ->c). Toutefois, toute autre expression d'un type équivalent peut aussi être utilisée comme but de recherche.

Pour contrôler la recherche exécutée afin de trouver les solutions aux buts de recherche, Curry propose l'opérateur prédéfini try :: (a -> Success) -> [a -> Success] qui prend un objectif de recherche et produit une liste d'objectifs de recherche.
L'opérateur de recherche try essaye d'évaluer l'objectif de recherche jusqu'à ce que le calcul finisse ou effectue une scission non-déterministe. Dans ce dernier cas, le calcul est arrêté et les différents objectifs de recherche créés par cette scission sont retournés comme liste de résultat.
Donc, une expression de la forme try (\x -> c) peut avoir trois résultats possibles :

1. Une liste vide. Ceci indique que l'objectif de recherche (\x -> c) n'a pas de solution. Par exemple, l'expression try (\x -> 1 =:=2) se réduit à [].
2. Une liste contenant un simple élément. Dans ce cas, l'objectif de recherche (\x -> c) a une solution représentée par l'élément de la liste. Par exemple, l'expression try (\x -> [x] =:= [0]) se réduit en [\x -> x =:= 0]. En général, la liste de solutions de l'opérateur try composée d'un seul élément est de la forme [\x -> x =:= e] où e est complètement évalué. Cela veut dire que e ne contient pas de fonction définie. Si ce n'est pas le cas, l'objectif peut ne pas être solutionné du fait de la définition des égalités des contraintes.
3. Une liste contenant plus qu'un élément. Dans ce cas, l'évaluation de l'objectif de recherche (\x -> c) demande une étape de calcul non-déterministe. Les différentes alternatives possibles après cette étape non-déterministe sont représentées comme étant un élément de cette liste. Par exemple, si la fonction f est définie par

   f a = c

   f b = d

   l'expression try (\x -> f x =:= d) se réduit à une liste [\x->x=:=a & f a =:= d, \x->x=:=b & f b =:= d]. Cet exemple nous montre aussi pourquoi une variable de recherche doit être abstraite : les liaisons alternatives ne peuvent pas être actuellement réalisées (puisque une variable libre est uniquement liée au maximum à une valeur

dans chaque thread de calcul) mais elles sont représentées par une contrainte d'égalité dans l'objectif de recherche.

b. MONADE

Nous allons avoir besoin d'intégrer nos opérateurs dans le monde, c'est-à-dire qu'il se peut que nos opérateurs aient besoin d'informations supplémentaires de la part du demandeur. En nous basant sur les spécifications des langages fonctionnels logiques, cela semble complexe pour ne pas dire impossible. En effet, nous ne savons pas ce qui se passe et quand cela se passe, hors comment établir un dialogue. Curry propose une solution avec son module IO. Ce module permet d'écrire et de lire, en un mot d'échanger, des informations avec le monde extérieur au programme.

Nous allons introduire brièvement ce principe pour nous permettre de cerner ce qui est possible en Curry et d'élargir ainsi notre compréhension de ce langage.

Dans l'approche de monade I/O, le monde extérieur n'est pas directement accessible mais seulement au travers d'actions qui vont changer ce monde. Donc, le monde est encapsulé dans un datatype abstrait qui propose des actions pour interagir avec lui. Le type de ces actions est IO t qui est un raccourci pour World -> (t, World) où World définit le type de tous les états du monde extérieur.

Si une action de type IO t est appliquée à un monde particulier, elle produit une valeur de type t et un nouveau monde. Par exemple, getChar est une action appliquée au monde extérieur qui permet, lors de son exécution, la lecture d'un caractère depuis l'entrée standard. getChar a le type IO Char. Le point important est que les valeurs de type World ne sont pas accessibles aux programmeurs. Ceux-ci peuvent uniquement créer des actions sur le monde. La conséquence du point précédent est qu'un programme ayant des opérations d'entrée/sortie (I/O) a une succession d'actions comme résultat. Ces actions sont calculées et exécutées lorsque le programme est connecté au monde. Par exemple :

```
getChar :: IO Char
getLine :: IO String
```

sont des actions qui lisent le caractère ou la ligne suivante de l'entrée standard. Les fonctions

```
putChar ::Char -> IO()
```

putStr ::String -> IO()

putStrLn ::String -> IO()

prennent un caractère ou une ligne et produisent une action qui, lorsqu'elle est appliquée à un monde, présente ce caractère ou ligne à la sortie standard.

Dès lors qu'un programme interactif est composé d'une séquence d'opérations d'I/O, où l'ordre dans la séquence est important, il existe deux opérations pour composer des actions dans un ordre précis.

La fonction (>>) :: IO a -> IO b -> IO b prend deux actions comme entrée et produit une action comme sortie. L'action de sortie consiste en l'exécution de la première action suivie par la seconde action. Il faut tenir compte du fait que la valeur produite par la première action est ignorée par la seconde. Si la valeur de la première action doit être prise en compte avant l'exécution de la seconde action, la composition des deux actions doit être réalisée par la fonction (>>=) :: IO a -> (a -> IO b) -> IO b où le second argument est une fonction tenant compte de la valeur produite par la première action comme entrée et exécute une deuxième action. Par exemple, l'action getLine >>= putStrLn est de type IO () et copie, lors de son exécution, une ligne provenant de l'entrée standard vers la sortie standard. La fonction return retourne :: a -> IO a est dans certains cas utile pour terminer une séquence d'actions et pour retourner un résultat d'opération d'entrée/sortie.

Donc, return v est une action qui ne change pas le monde et qui retourne la valeur v.

Pour exécuter une action, il doit exister une expression principale dans le programme. Par exemple, les programmes interactifs ont le type IO (). Dès lors que le monde ne peut pas être copié (notons que le monde contient au minimum l'ensemble du file system), on doit exclure les relations entre non-déterminisme et les opérations d'entrée/sortie. Ce qui veut dire que l'action appliquée doit toujours être connue. >> et >>= suspendent leur exécution si leurs arguments sont des variables libres.

En conséquence, toutes les recherches possibles doivent être encapsulées entre des opérations d'entrée/sortie. Le compilateur pourra prévenir l'utilisateur sur les calculs non-déterministes qui peuvent se produire dans les actions d'entrée/sortie, ce qui permettra au programmeur de les encapsuler.

Pour notre étude sur des recherches encapsulées, les fonctions browse :: (a->Success) -> IO (), mais aussi browseList :: [a->Success] -> IO () peuvent être utiles pour nos tests.

c. LAZY EVALUATION

LazyEvaluation[Beff] devient pratique quand une expression est soit trop coûteuse, soit impossible à évaluer et n'a pas le besoin d'être évaluée à tout prix. C'est aussi un facilitateur pour la définition de structure data récursive infinie. En effet, comme chaque niveau de récursivité est évalué uniquement si cela est nécessaire, les données sont générées si elles sont utilisées et l'évaluation de la structure data peut être arrêtée lorsque son usage est complet.

LazyEvaluation demande que les données utilisées dans le calcul soient disponibles et significatives au moment de l'évaluation. Dans les langages fonctionnels logiques purs, cette condition est garantie par le fait que les changements d'état ne sont pas permis.

LazyEvaluation nous donne l'opportunité de postposer le calcul jusqu'au moment où les données sont disponibles.

Essayons de bien cerner cette évaluation paresseuse.
Prenons le code suivant :

```
magic :: Int -> Int -> [Int]
magic 0 _ = []
magic m n = m : (magic n (m+n))

getIt :: [Int] -> Int -> Int
getIt [] _ = 0
getIt (x:xs) 1 = x
getIt (x:xs) n = getIt xs (n-1)
```

Analysons ce code : nous constatons facilement que (magic 1 1) représente la suite de Fibonacci, c'est-à-dire [1,1,2,3,5,...]. Nous avons comme résultat une liste infinie.
Essayons maintenant d'évaluer getIt (magic 1 1) 3.

Dans ce contexte, getIt (magic 1 1) x est en fait simplement (magic 1 1) !! (x - 1).
N'oublions pas que magic 1 1 ne se termine jamais alors que nous constatons que l'évaluation de getIt (magic 1 1) 3, elle, se termine.
Nous pouvons nous poser la question : comment est-il possible que, alors qu'un paramètre d'une fonction est infini et ne peut donc pas être évalué, la fonction complète elle peut l'être ? La réponse est notre LazyEvaluation.
Nous allons utiliser Haskell (Curry) pour essayer de nous faire une idée de la manière dont la fonction complète est évaluée.
La première étape à réaliser est de calculer quel pattern de getIt concorde avec l'expression getIt (magic 1 1) 3.
Le premier pattern correspond uniquement si nous passons une liste vide comme argument et n'importe quel Int. Nous passons 3 comme Int, donc il nous reste à savoir si nous passons une liste vide. Pour pouvoir répondre à cette question, une première étape d'évaluation de (magic 1 1) est nécessaire.

getIt (1 : (magic 1 (1+1)) 3

Nous constatons que nous ne passons pas une liste vide puisque nous avons une liste (1 : (magic 1 (1+1)) avec une tête égale à 1.
Conclusion, nous n'avons pas de concordance avec le premier pattern de getIt.
Nous allons regarder le second pattern. Celui-ci s'attend à une liste non vide, ce que nous avons, mais également à un Int égal à 1, or nous avons 3.
Il ne nous reste plus que le troisième pattern qui, lui, concorde. Les concordances sont :

x avec 1
xs avec(magic 1 (1+1))
n avec 3

Cet ensemble nous donne comme résultat : getIt (magic 1 (1+1)) (3-1).
Nous recommençons les recherches de concordance et nous arrivons à la même conclusion avec comme résultat : getIt (magic (1+1)(1+(1+1)))) (2-1)
Nous devons à nouveau recommencer les recherches de concordance :
Evaluation de (1+1)

getIt (magic 2 (1+(1+1)))) (2-1)

Evaluation de (magic 2 (1+(1+1))))

getIt (2 : magic (1 +(1+1)) (2+(1+(1+1)))) (2-1)

Evaluation de (2-1)

getIt (2 : magic (1+(1+1)) (2+(1+(1+1)))) 1

Nous avons maintenant une concordance avec le second pattern : getIt (x:xs) 1 = x, ce pattern est un pattern terminal puisque lorsque nous avons une liste non vide et un Int égale à 1, le résultat est la tête de la liste.

Dans ce cas-ci, le résultat renvoyé sera 2. Nous avons donc trouvé un résultat alors qu'un des paramètres utilisés était une liste infinie.

CHAPITRE 3 : FILTRAGE

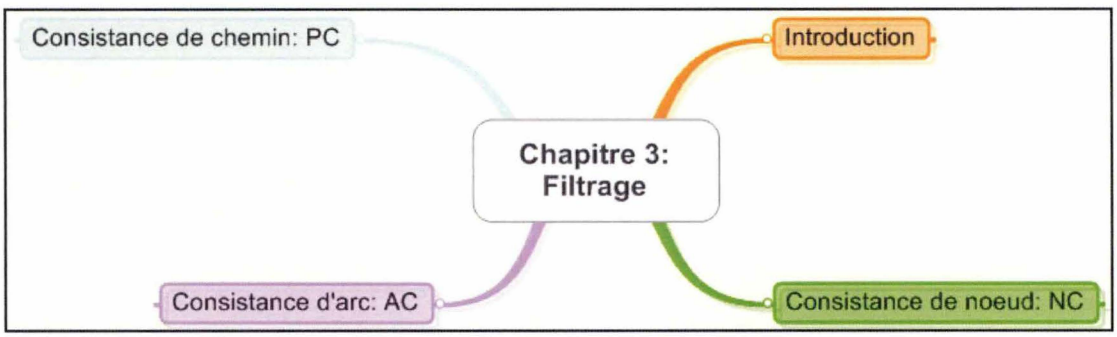


Figure 6 : Carte heuristique - Filtrage

1. INTRODUCTION

Les techniques de consistance ont été développées en premier pour améliorer la performance des programmes de reconnaissance d'images. La reconnaissance d'images impliquait jusqu'alors la qualification de toutes les lignes d'une image et ce d'une manière consistante.
Le nombre de combinaisons possibles pouvait être énorme alors qu'un petit nombre étaient consistantes.
Les techniques de consistance ont pour but d'exclure le plus tôt possible dans le processus les qualifications inconsistantes et donc d'écourter la recherche des qualifications consistantes.
Ces techniques dédicacées à un type de problèmes ont depuis lors montré leur intérêt dans une large variété de problèmes de recherche.
En se rappelant un des points du chapitre précédent parlant de la complexité[Chapitre 1.4], les techniques de consistance sont déterministes et sont à opposer à la recherche qui est non déterministe.
Ceci nous donne comme séquence : un calcul déterministe est exécuté en premier et ce aussi longtemps que possible. Ensuite, lorsque plus aucune propagation n'est encore possible, nous passons à une recherche non déterministe.

Les techniques de consistance sont rarement utilisées seules pour résoudre entièrement des problèmes sous contraintes. Toutefois, nous constatons que le résultat des techniques de consistance est une diminution de l'espace de recherche, ce qui peut être considéré comme une action de filtrage.

Dans la suite de ce chapitre, nous allons introduire les grands types de consistance basés sur l'arité.

- Arité = 1 (unaire) : une seule variable est concernée : $x \neq 2$
- Arité = 2 (binaire) : concerne un couple de variables : $x > y$
- Arité =k : concerne k variables

2. CONSISTANCE DE NŒUD : NC

Nous allons commencer par la plus simple des techniques de consistance qui est la validation d'une contrainte unaire. Cette technique porte le nom de consistance de nœud, en faisant référence à un graphe.

Dans un graphe de contraintes, un nœud représente une variable V.

Cette variable satisfait l'ensemble des contraintes si pour toute valeur x dans le domaine de V, chaque contrainte unaire impliquant V est satisfaite.

La définition d'un domaine pour une variable est l'ensemble de toutes les valeurs possibles qui peuvent être assignées à cette variable.

Ceci nous permet de dire que si le domaine D de la variable V contient une valeur x qui ne satisfait pas la contrainte unaire de V, alors l'instanciation de V avec la valeur x va toujours résulter en une erreur.

Donc l'inconsistance de nœud peut immédiatement éliminer cette valeur x du domaine D de chaque variable V qui ne satisfait pas la contrainte unaire de V.

Nous proposons un pseudo-code écrit de manière impérative (Algorithme NC), le problème du passage d'un paradigme impératif à un paradigme déclaratif sera l'un des problèmes important et récurent dans notre travail.

Algorithme NC

```
procedure NC
        for each V in nodes(G)
```

```
            for each X in the domain D of V
                  if any unary constraint on V is inconsistent with X
                  then
                        delete X from D;
                  endif
            endfor
      endfor
end NC
```

3. Consistance d'arc : AC

Si le graphe des contraintes est prouvé nœud consistant, nous pouvons retirer de nos contraintes les contraintes unaires puisqu'elles sont toutes satisfaites. En travaillant sur des problèmes sous contraintes binaires, nous devons maintenant prouver la consistance des contraintes binaires. Dans le graphe des contraintes, les contraintes binaires correspondent aux arcs, de là le nom de consistance d'arc.

La définition de la consistance d'arc pour un arc (Vi,Vj) est : si pour toute valeur x du domaine Di de Vi, il existe une valeur y du domaine Dy de Vy tel que Vi=x et Vj=y est permis par la contrainte binaire entre Vi et Vj.

Nous devons tenir compte également que le concept de consistance d'arc est directionnelle, ce qui implique que, si (Vi,Vj) est prouvé consistant, cela n'implique pas automatiquement que (Vj,Vi) est consistant.

Si nous rencontrons une valeur x pour laquelle (Vi,Vj) n'est pas consistant, nous pouvons rendre consistant (Vi,Vj) en retirant la valeur x du domaine Di de Vi.

Remarquons que le fait d'éliminer cette valeur n'élimine pas une solution du problème sous contraintes, du moins, dans le cadre d'un problème CSP. Néanmoins, il existe une classe de problèmes MCSP (Maximum constraint satisfaction problem) où cette remarque n'est pas valable, étant donné que, pour un problème de type MCSP, on cherche à satisfaire un maximum de contraintes et pas l'ensemble des contraintes. Nous n'aborderons pas ce type de problèmes pour le moment mais nous devons en tenir compte lors de la définition des problèmes acceptables pour nos opérateurs.

L'algorithme suivant propose de rendre un problème prouvé arc consistant en éliminant du domaine la valeur.

Algorithme REVISE

```
procedure REVISE(Vi,Vj)
        DELETE <- false;
        for each X in Di do
                if there is no such Y in Dj such that (X,Y) is consistent,
                        then
                                delete X from Di;
                                DELETE <- true;
                endif;
        endfor;
        return DELETE;
end REVISE
```

L'algorithme précédent rend un seul arc consistant mais ne perdons pas de vue que nous voulons rendre un graphe complet arc consistant. Ceci veut donc dire que lorsque nous retirons une valeur d'un domaine Di de Vi, nous réduisons le domaine Di, il se peut que cette même valeur soit utilisée pour rendre un autre arc (Vh,Vi) consistant. Nous devons donc propager le changement. Cette propagation peut s'effectuer en réexécutant notre premier algorithme sur l'ensemble des arcs, même ceux qui avaient déjà été prouvés arc consistant, jusqu'à ce que nous ne rencontrions plus d'inconsistance et donc que nous n'effectuions plus de réduction de domaine.

Ce concept de propagation est la base de l'algorithme AC-1.

Algorithm AC-1

```
procedure AC-1
        Q <- {(Vi,Vj) in arcs(G),i≠j};
        repeat
                CHANGE <- false;
                for each (Vi,Vj) in Q do
                        CHANGE <- REVISE(Vi,Vj) or CHANGE;
                endfor
        until not(CHANGE)
end AC-1
```

Nous pouvons directement critiquer cet algorithme pour sa performance. En effet, un seul changement dans un domaine force le fait de devoir prouver à nouveau l'ensemble des arcs même si la modification n'affecte qu'un sous-ensemble de l'ensemble des arcs.

Nous pourrions commencer à améliorer l'algorithme AC-1 en cherchant à travailler par sous-ensembles. Le sous-ensemble de travail serait construit par rapport à l'arc que l'on a à rendre consistant. Si nous prenons l'arc (Vk,Vm) et que nous devons réduire le domaine de Vk (n'oublions pas que (Vk,Vm) prouvé arc consistant n'implique pas (Vm,Vk) arc consistant), nous pouvons rendre inconsistant l'ensemble des arcs ayant Vk comme second nœud (_,Vk). Donc il n'est plus nécessaire de repasser en revue l'ensemble des arcs du graphe mais uniquement ce sous-ensemble.
L'algorithme prenant cette amélioration en compte est l'AC-3. Nous mettons également son pseudo-code pour montrer que nous avons besoin d'un effet mémoire temporaire(Q).

Algorithm AC-3

```
procedure AC-3
        Q <- {(Vi,Vj) in arcs(G),i≠j};
        while not Q empty
                select and delete any arc (Vk,Vm) from Q;
                if REVISE(Vk,Vm) then
                        Q <- Q union {(Vi,Vk) such that (Vi,Vk) in arcs(G),i≠k,i≠m}
                endif
        endwhile
end AC-3
```

Cette amélioration n'est pas la seule possible et il existe des algorithmes AC-4, AC-5, AC-7, … Le but ici n'est pas de détailler le meilleur algorithme mais d'expliciter les principes et de voir comment traduire cela dans un paradigme déclaratif[Chapitre 6].

4. Consistance de chemin : PC (K consistency)

Nous rappelons qu'il existe pour les problèmes ayant une arité > 2 la consistance de chemin.
Un graphe est prouvé K consistant si la condition suivante est rencontrée :
Instancions K-1 variables avec des valeurs prises dans leur domaine respectif et qui satisfont l'ensemble des contraintes qui existent entre ces K-1 variables et choisissons n'importe quelle $K^{éme}$ variable. Il doit exister une valeur pour cette $K^{éme}$ variable qui satisfasse l'ensemble des contraintes existant entre ces K variables.
Un graphe est dit fortement K consistant s'il est prouvé J consistant pour $\forall j \leq K$

CHAPITRE 4 : HEURISTIQUE

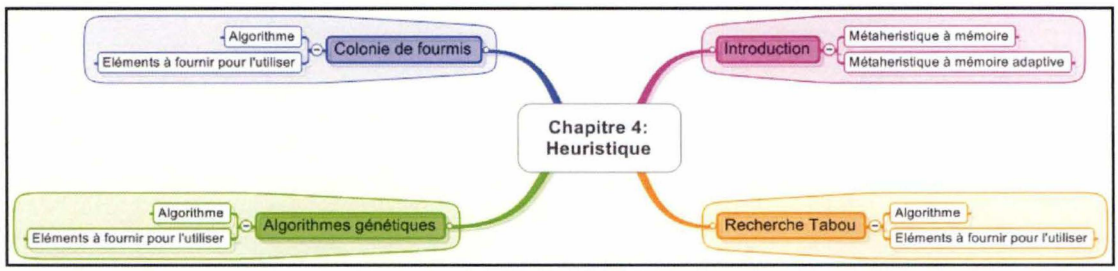


Figure 7 : Carte heuristique - Heuristique

1. INTRODUCTION

a. VISION UNIQUE DES HEURISTIQUES UNIQUES

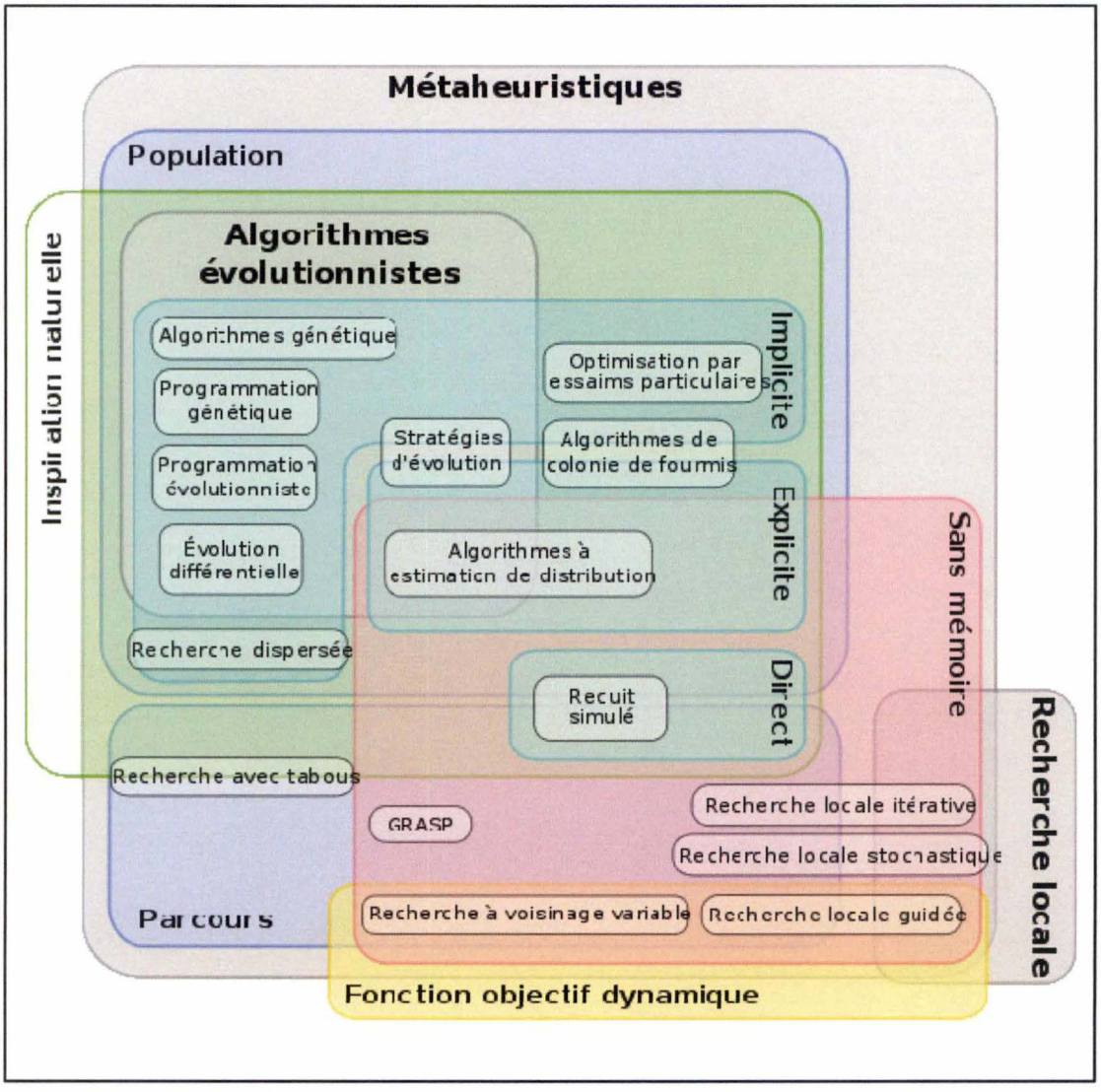
Dans le chapitre 1, nous avons introduit la notion d'heuristique dans le cadre de notre étude. En recherchant l'utilisation pratique que l'on fait de ces méthodes, nous pouvons nous rendre compte qu'un grand nombre de nos appareils les utilises, à commencer par ceux connectés à un réseau.

En ce qui nous concerne, les métaheuristiques qui ne sont que méthodes génériques et heuristiques d'optimisation ou méthodes générales de recherche locale, se développent de manière explosive à l'heure actuelle. Si nous allons sur le net à la recherche de métaheuristique, nous trouvons 114.000 réponses, dans ces réponses nous avons une grande partie qui est liée à la recherche opérationnelle, ensuite une partie liée à l'intelligence artificielle et le reste sont des articles, des algorithmes, …

Parmi ces articles, on peut citer ceux dont le sujet revient le plus souvent :

- Algorithme génétique
- Recuit simulé
- Recherche tabou

- Colonie de fourmis



[Dreo]

Figure 8 : Classification des heuristiques

Si on passe du temps à lire le contenu de ces articles, on se rend compte que l'intérêt que l'on porte aux métaheuristiques n'est pas dû uniquement à un effet de mode mais bien à des raisons comme la facilité d'implémentation, la capacité et la flexibilité qu'ils ont à prendre en considération des contraintes spécifiques liées à la nature même du problème qu'on leur soumet, la qualité de la solution et la rapidité pour y parvenir.

Pourtant d'un point de vue purement théorique, il est difficile de justifier l'utilisation de métaheuristiques, peu ont été prouvées convergentes par exemple. Nous pouvons donner à titre d'information, l'existence de théorèmes de convergence pour le recuit simulé ou la recherche tabou. Ces théorèmes démontrent que l'on a une probabilité élevée de trouver une solution globalement optimale si l'on alloue un temps de calcul extravagant. Ce temps est considéré comme supérieur au temps qui serait nécessaire pour permettre l'énumération complète de l'espace de solution.

Nous savons maintenant ce qu'en pense la théorie mais qu'en est-il de la pratique ? Il en est tout autre chose ! En pratique, les performances de ces techniques sont excellentes. Nous pouvons aussi lire que de plus en plus de nouvelles techniques sont basées sur l'hybridation de deux (voire plus) métaheuristiques. Cette approche hybride sera discuté plus loin dans notre étude[Chapitre 6.b/c].

Nous pouvons également constater que ces hybridations lissent les différences et que donc les caractéristiques d'une métaheuristique disparaissent ou sont diminuées par son croisement avec une autre métaheuristique.

Toutefois, ces méthodes hybrides partagent trois particularités :

- elles mémorisent des solutions ou des caractéristiques des solutions visitées durant le processus de recherche ;
- elles font usage d'une procédure créant une nouvelle solution à partir des informations mémorisées ;
- elles sont quasi toutes dotées d'une recherche locale tel qu'une recherche tabou ou un recuit simulé.

Si nous acceptons ces trois particularités, nous pouvons imaginer avoir une représentation unifiée pour ces méthodes.

Un nom a été proposé pour l'ensemble de ces méthodes : programmation à mémoire adaptative (PMA).[TGGP]

Derrière cet acronyme, se cache un algorithme en quatre étapes :

- mémorisation d'un jeu de solutions ou d'une structure de données rassemblant les particularités des solutions produites par la recherche.

- Répéter tant qu'un critère d'arrêt n'est pas satisfait :
  - Construction d'une solution provisoire sur la base des données mémorisées.
  - Amélioration de la solution par un algorithme de recherche locale.
  - Mémorisation de la nouvelle solution ou de la structure de données associée.

Nous utiliserons dans les chapitres 5 et 6 cette approche. Dans le chapitre 5 qui se focalisera sur la représentation d'un problème, nous tiendrons compte de cet aspect mémoire et de la nécessité de primitives de gestion, tandis que dans le chapitre 6 qui se concentrera sur la résolution de problèmes, nous essayerons de traduire de manière déclarative la succession de ces quatre étapes.

b. MÉTAHEURISTIQUE À MÉMOIRE

Avant d'aller plus loin dans la PMA, il nous semble important de comprendre la gestion « normale » de la mémoire dans le cadre de certaines métaheuristiques. Toutefois, le but n'est pas d'aller trop loin dans cette étude car elle sera approfondie dans ce même chapitre[Chapitre 4.2/3/4].

- Algorithmes génétiques : pour cet algorithme, nous considérons la population comme étant la mémoire. Nous devons pouvoir ajouter un nouvel individu, mais aussi pouvoir en retirer un. Un individu peut avoir une pondération pour aider au choix des deux individus allant devenir parents. Un enfant doit pouvoir subir une mutation et donc cela se traduit par une primitive de gestion d'un individu. En résumé, nous gérons la population d'individus mais également l'individu.
- Recherche tabou : dans ce cas-ci, nous avons une solution et un ensemble de modifications qui, après avoir été appliquées, sont considérées comme tabou. Nous devons garder en mémoire, la solution et une file de type FIFO (first in first out) comprenant les chemins, modifications tabous.

- Colonies de fourmis : nous avons à connaître l'ensemble des solutions admissibles, la fonction objectifs à optimiser et enfin les traces qui représenteront la mémoire adaptative.

c. MÉTAHEURISTIQUE À MÉMOIRE ADAPTIVE

Si nous reprenons les métaheuristiques décrites au point b, nous pouvons nous demander où se trouve leur point commun. Pour montrer la convergence de ces méthodes, nous devons nous référer à leurs implémentations les plus récentes et les plus optimales. Ces implémentations fonctionnent toutes suivant les mêmes principes. Nous sommes donc devant une unification à un point tel qu'il devient de plus en plus difficile de les distinguer.
Prenons par exemple la population d'un algorithme génétique et les traces d'une colonie de fourmis : les deux peuvent être considérées comme une mémoire spéciale d'une recherche tabou. Réciproquement, une recherche tabou construisant périodiquement une nouvelle solution en utilisant une mémoire peut être assimilée à une colonie de fourmis. Ce qui nous permet de dire que les méthodes généralistes les plus usitées de résolution de problèmes d'optimisation combinatoire présentent un ensemble de caractéristiques identiques :

- les solutions générées par la recherche sont mémorisées ;
- une solution provisoire est construite en consultant la mémoire ;
- la solution provisoire est améliorée à l'aide d'une recherche locale ;
- la nouvelle solution provisoire est utilisée pour mettre à jour la mémoire, pour l'adapter aux nouvelles connaissances que la solution apporte.

d. MÉTAHEURISTIQUE À MÉMOIRE ADAPTIVE PARALLÈLE

Il nous reste un dernier point à valider pour accepter la PMA, c'est l'utilisation en parallèle par plusieurs processus. C'est le cas pour la colonie de fourmis où nous avons plusieurs processus fonctionnant en parallèle et de manière indépendante. Chacun des processus doit pouvoir mettre à jour la mémoire avec l'information qu'il a recueillie : trace qui est une pondération sur le chemin qu'il a testé.

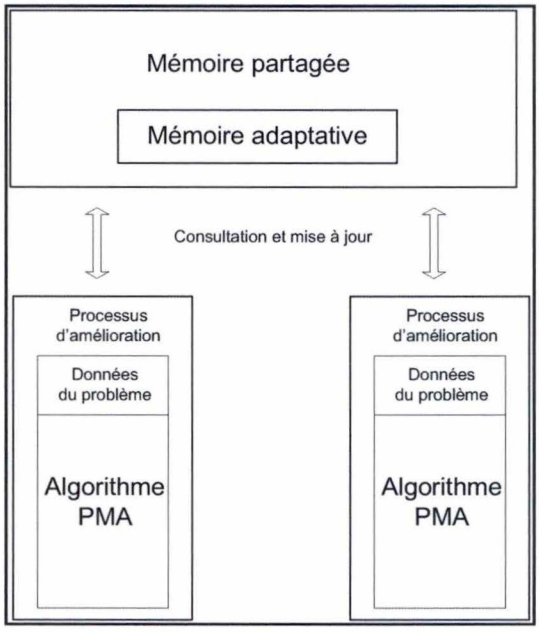


Figure 9 : Programme à mémoire adaptive parallèle

La proposition faite pour une PMA parallèle (figure 9) est l'exécution de la boucle principale par chacun des processus avec un seul moyen de communication inter processus qui serait la mémoire adaptive.

Nous émettons une réserve quant à cette approche pour des raisons de « lock » de mémoire lors des mises à jour mais surtout vis-à-vis de la consistance de cette mémoire. En effet, en généralisant le monde des métaheuristiques par la PMA et ensuite par la PMA parallèle, on quitte le monde du mono thread pour rejoindre celui du multi threading et de ses contraintes.

Une autre compréhension serait d'imaginer que le problème peut être divisé en sous-problèmes pouvant être traités simultanément mais surtout indépendamment.

Une autre approche est celle proposée par le langage de programmation utilisé pour implémenter l'algorithme. En effet, rappelons-nous, par exemple, que Curry langage multi paradigmes propose nativement l'exécution concurrente et cherche à résoudre les contraintes de manière concurrentielle.

2. Etude de cas : Recherche tabou

a. Introduction

L'idée de la recherche tabou consiste, à partir d'une position donnée, à en explorer le voisinage et à choisir la position dans ce voisinage qui minimise la fonction objectif.
Il est essentiel de noter que cette opération peut conduire à augmenter la valeur de la fonction (dans un problème de minimisation) : c'est le cas lorsque tous les points du voisinage ont une valeur plus élevée. C'est à partir de ce mécanisme que l'on sort d'un minimum local.
Le risque cependant est qu'à l'étape suivante, on retombe dans le minimum local auquel on vient d'échapper. C'est pourquoi il faut que l'heuristique ait de la mémoire : le mécanisme consiste à interdire (d'où le nom de tabou) de revenir sur les dernières positions explorées.
Les positions déjà explorées sont conservées dans une file FIFO (appelée souvent liste tabou) d'une taille donnée, qui est un paramètre ajustable de l'heuristique. Cette pile doit conserver des positions complètes, ce qui dans certains types de problèmes, peut nécessiter l'archivage d'une grande quantité d'informations. Cette difficulté peut être contournée en ne gardant en mémoire que les mouvements précédents, associés à la valeur de la fonction à minimiser.[FoLaLo] et [Gend]

b. Algorithme

Nous partons de l'hypothèse que nous devons minimiser une fonction f(S).
S : La solution actuelle
S* : la meilleure solution connue
f* : la valeur de S*
N(S) : les environs de S
N'(S) : un sous-ensemble de N(S) permis, c'est-à-dire en ayant retiré les mouvements tabous et qui est permis par la fonction d'aspiration.

START Recherche Tabou

Choisir ou construire une solution initiale S0

S <- S0

f* <- f(S0)

S* <- S0

T <- []

Tant que critère d'arrêt non rencontré faire

Choisir S dans argmin [f(S')] ; S' ∈ N'(S)

Si f(S) < f*

Alors

f* <- f(S)

S* <- S

END Si

Enregistrer dans T le mouvement courant comme tabou

Retirer les plus vieux mouvements si nécessaire.

END Tant que

END Recherche Tabou

c. Eléments à fournir pour l'utiliser

La fonction à optimiser

Le type d'optimisation

3. Etude de cas : Algorithmes génétiques

a. Introduction

Les algorithmes génétiques, afin de permettre la résolution de problèmes, se basent sur les différents principes décrits ci-dessus. De manière globale, on commence avec une population de base qui se compose le plus souvent de chaînes de caractères correspondant chacune à un chromosome. Nous reviendrons par la suite sur les différentes structures de données possibles (voir Codage) mais nous retiendrons pour le moment l'utilisation du codage binaire (ex. : 0100110).

Le contenu de cette population initiale est généré aléatoirement. On attribue à chacune des solutions une note qui correspond à son adaptation au problème. Ensuite, on effectue une sélection au sein de cette population.

Il existe plusieurs techniques de sélection. Voici les principales utilisées :

- sélection par rang : cette technique de sélection choisit toujours les individus possédant les meilleurs scores d'adaptation, le hasard n'entre donc pas dans ce mode de sélection.
- Probabilité de sélection proportionnelle à l'adaptation (appelée aussi roulette ou roue de la fortune) : pour chaque individu, la probabilité d'être sélectionné est proportionnelle à son adaptation au problème. Afin de sélectionner un individu, on utilise le principe de la roue de la fortune biaisée. Cette roue est une roue de la fortune classique sur laquelle chaque individu est représenté par une portion proportionnelle à son adaptation. On effectue ensuite un tirage au sort homogène sur cette roue.
- Sélection par tournoi : cette technique utilise la sélection proportionnelle sur des paires d'individus, puis choisit parmi ces paires l'individu qui a le meilleur score d'adaptation.
- Sélection uniforme : la sélection se fait aléatoirement, uniformément et sans intervention de la valeur d'adaptation. Chaque individu a donc une probabilité 1/P d'être sélectionné, où P est le nombre total d'individus dans la population.

Lorsque deux chromosomes ont été sélectionnés, on réalise un enjambement. On effectue ensuite des mutations sur une faible proportion d'individus, choisis aléatoirement. Ce processus nous fournit une nouvelle population. On réitère le processus un grand nombre de fois, de manière à imiter le principe d'évolution, qui ne prend son sens que sur un nombre important de générations. On peut arrêter le processus au bout d'un nombre arbitraire de générations ou lorsqu'une solution possède une note suffisamment satisfaisante.

Considérons par exemple les deux individus suivants dans une population où chaque individu correspond à une chaîne de 8 bits : A = 00110010 et B = 01010100. On ajuste la probabilité d'enjambement à 0.7.

Chromosome	Contenu
A	00:110010
B	01:010100
A'	00 010100
B'	01 110010

Supposons ici que l'enjambement ait lieu, on choisit alors aléatoirement la place de cet enjambement (toutes les places ayant la même probabilité d'être choisies). En supposant que l'enjambement ait lieu après le deuxième allèle, on obtient A' et B' (« : » marquant l'enjambement sur A et B). Ensuite, chacun des gènes des fils (ici, chacun des bits des chaînes) est sujet à la mutation. De la même manière que pour les combinaisons, on définit un taux de mutation (très bas, de l'ordre de 0,001 - ici on peut s'attendre à ce que A' et B' restent identiques).

En effectuant ces opérations (sélection de deux individus, enjambement, mutation) un nombre de fois correspondant à la taille de la population divisée par deux, on se retrouve alors avec une nouvelle population (la première génération) ayant la même taille que la population initiale, et qui contient globalement des solutions plus proches de l'optimum. Le principe des algorithmes génétiques est d'effectuer ces opérations un maximum de fois de façon à augmenter la justesse du résultat.

Il existe plusieurs techniques qui permettent éventuellement d'optimiser ces algorithmes. On trouve par exemple des techniques dans lesquelles on insère à chaque génération quelques individus non issus de la descendance de la génération précédente mais générés aléatoirement. Ainsi, on peut espérer éviter une convergence vers un optimum local.[Reev]

b. Algorithme

START Algorithme génétique

Choisir une population initiale de chromosomes

Tant que critère d'arrêt non rencontré faire

Répéter

Si la condition de croisement est satisfaite

Alors

Choisir les chromosomes parents

Choisir les paramètres de croisement

Exécuter le croisement

END Si

Si la condition de mutation est satisfaite

Alors

Choisir les points de mutation

Exécuter la mutation

END Si

Evaluer la pertinence de la descendance

Jusqu'à ce que suffisamment de descendants générés

Choisir une nouvelle population

END Tant que

END Algorithme génétique

c. ELÉMENTS À FOURNIR POUR L'UTILISER

La fonction à optimiser

Le type d'optimisation

Le codage du problème (binaire ou autre)

Les conditions de croisement et de mutation

4. ETUDE DE CAS : COLONIE DE FOURMIS

a. INTRODUCTION

Les algorithmes de colonies de fourmis sont des algorithmes inspirés du comportement des fourmis et qui constituent une famille de métaheuristiques d'optimisation.

b. ALGORITHME

START Colonie de fourmis

Initialisation des pistes de phéromones

Tant que critère d'arrêt non rencontré faire

construire les solutions composant par composant

utilisation (facultative) d'une heuristique

mise à jour des pistes de phéromones ;

END Tant que

END Colonie de fourmis

c. ELÉMENTS À FOURNIR POUR L'UTILISER

La fonction à optimiser

Le type d'optimisation

CHAPITRE 5 : MODÉLISATION D'UN PROBLÈME

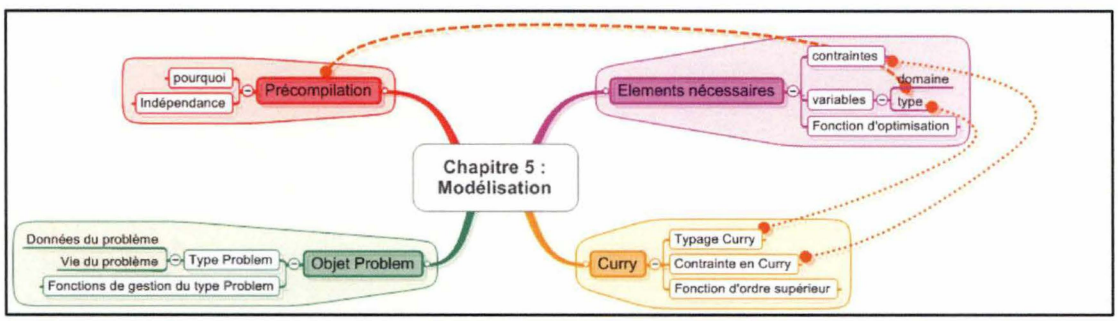


Figure 10 : Carte heuristique - Modélisation d'un problème

1. INTRODUCTION

Définition de problème (petit Robert) :

1° Question à résoudre qui prête à discussion dans une science. Question à résoudre, portant soit sur un résultat inconnu à trouver à partir de certaines données, soit par la détermination de la méthode à suivre pour obtenir un résultat supposé connu.

2° Difficulté qu'il faut résoudre pour obtenir un certain résultat ; situation instable ou dangereuse exigeant une décision.

Nous allons essayer dans ce chapitre de modéliser un problème d'une manière telle qu'il y aura indépendance entre notre modélisation et la manière de solutionner le problème.

Il faut toutefois se rappeler :

- qu'un problème n'a pas obligatoirement de solution
- qu'un problème peut générer un nouveau problème lorsque l'on essaye de le solutionner
- qu'un problème peut avoir un ensemble de solutions acceptables et une solution optimale.

Commençons par qualifier ce que nous entendons par problème et surtout quel type de problème va nous préoccuper. Nous avons donné une définition générale du mot problème et il est clair que nous ne comptons pas nous attaquer aux problèmes en général mais aux problèmes d'optimisation sous contraintes.

2. Eléments nécessaires

Quels sont les éléments nécessaires pour qualifier complètement un problème afin que nous puissions lui appliquer un opérateur sans devoir demander de nouvelles informations ? Arrivés à ce point, nous nous trouvons devant le dilemme de l'œuf ou de la poule. En effet, comment connaître les éléments nécessaires à donner aux opérateurs de résolution, et ce, peu importe l'opérateur choisi, et de l'autre, comment trouver une modélisation, en partant de rien, capable d'être comprise par n'importe lequel de nos opérateurs de solution ?
Pour ce faire nous avons, comme souvent, travaillé par « try and error ». Nous devons commencer par comprendre un problème et le traduire, le typer en Curry et définir les interfaces dont nous aurons besoin pour le faire vivre, pour enfin permettre l'application d'un ou plusieurs opérateurs de résolution.
Nous avons, dans la suite de ce chapitre, rappelé le typage en Curry. En effet, notre objet « Problème » sera du type « entier, réel, liste, ... » en fonction des éléments de description qui seront introduits pour décrire ce dernier.
Nous devons aussi nous conformer à l'approche de programmation fonctionnelle logique qui demande de créer un objet et ses fonctions de gestion telles que constructeur, ...
Nous pouvons nous demander si la fonction représentant l'opérateur de résolution fait partie des fonctions de gestion ou alors si nous devons la mettre dans un module reprenant l'ensemble des opérateurs.
Le choix qui a été fait est de différencier les fonctions « proches » de l'objet et les fonctions qui auront à utiliser un tel objet.

Qu'est ce qui définit un problème ?

- ses variables ;
- les valeurs que peuvent prendre ces variables ;

- les contraintes sur ces variables ;
- la fonction d'optimisation.

Doit-on mettre les valeurs des étapes intermédiaires de résolution dans l'objet ou est-ce une qualité propre à l'opérateur ?
Notre choix a été de laisser à chacun ses spécificités. C'est pourquoi notre objet Problème n'aura aucune idée de la manière dont on cherche à le résoudre et les fonctions de recherche de solutions auront à gérer un objet ayant deux parties principales : la zone de travail qui doit pouvoir être partagée entre plusieurs opérateurs de solutions et la sortie qui est un nouveau problème que l'on espère plus proche de la solution que le précédent.

Qu'est ce qui n'est pas le problème en tant que tel mais qui est à la fois lié au problème et à la manière dont on va essayer de le résoudre ?

- le type des variables
- le domaine des variables et la manière dont on le défini (continu, fini)

Ce sont les points qui viennent d'être donnés qui seront discutés dans la conclusion ce de chapitre.

3. TYPAGE EN CURRY

Build in types

- Boolean Values
- Constraints

  The type Success is the result type of expressions used in conditions of defining rules. Since conditions must be successfully evaluated in order to apply the rule, the type Success can also be interpreted as the type of successful evaluations. A function with result type Success is also called a constraint.

  Constraints are different from Boolean-valued functions: a Boolean expression reduces to True or False whereas a constraint is checked for satisfiability. A constraint is applied (i.e., solved) in a condition of a conditional equation. The equational

constraint e1=:=e2 is an elementary constraint which is solvable if the expressions e1 and e2 are evaluable to unifiable data terms. success denotes an always solvable constraint.

Constraints can be combined into a conjunction of the form c1 & c2 & . . . & cn by applying the concurrent conjunction operator &. In this case, all constraints in the conjunction are evaluated concurrently. Constraints can also be attached to arbitrary expressions by the operator &>, i.e., the constrained expression c &> e is evaluated by first solving constraint c and then evaluating e. This operator can also be used to enforce a sequential order for constraint evaluation, e.g., the combined constraint c1 &> c2 will be evaluated by first completely evaluating c1 and then c2.

Constraints can be passed as parameters or results of functions like any other data object. For instance, the following function takes a list of constraints as input and produces a single constraint, the concurrent conjunction of all constraints of the input list:

```
conj :: [Success] -> Success
conj [] = success
conj (c:cs) = c & conj cs
```

The trivial constraint success is usually not shown in answers to a constraint expression. For instance, the constraint x*x=:=y & x=:=2 is evaluated to the answer {x=2, y=4}

- Functions

The type t1 -> t2 is the type of a function which produces a value of type t2 for each argument of type t1. A function f is applied to an argument x by writing "f x". The type expression t1 -> t2 -> · · · -> tn+1 is an abbreviation for the type t1 -> (t2 -> (· · · -> tn+1))and denotes the type of a (currified) n-ary function, i.e., -> associates to the right. Similarly, the expression f e1 e2 . . . en is an abbreviation for the expression (...((f e1) e2) . . . en) and denotes the application of a function f to n arguments, i.e., the application associates to the left.

The prelude also defines a right-associative application operator "$" which is sometimes useful to avoid brackets. Since $ has low, right-associative binding precedence, the expression "f $ g $ 3+4" is equivalent to "f (g (3+4))".

Furthermore, the prelude also defines right-associative application operators that enforces the evaluation of the argument to a particular degree. For instance, the definition of "$!" is based on a predefined (infix) operator seq that evaluates the first argument and returns the value of the second argument:

```
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Thus, if inf is a non-terminating function (e.g., defined by "inf = inf") and f a constant function defined by "f _ = 0", then the evaluation of the expression "f $! inf" does not terminate whereas the expression "f $ inf" evaluates to 0. Similarly, the operator "$!!" completely evaluates its second argument (i.e., to a normal form), "$#" evaluates its second argument to a non-variable term (by ensureNotFree, see Section 5.4) and suspends, if necessary, and "$##" completely evaluates its second argument to a term without variables.

- Integers
- Floating point Numbers
- Lists
- Characters
- Strings
- Tuples
- Type System

Curry is a strongly typed language with a Hindley/Milner-like polymorphic type system [14].7 Each variable, constructor and operation has a unique type, where only the types of constructors have to be declared by datatype declarations (see Section 2.1). The types of functions can be declared (see Section 2.3) but can also be omitted. In the latter case they will be reconstructed by a type inference mechanism.

Note that Curry is an explicitly typed language, i.e., each function has a type. The type can only be omitted if the type inferencer is able to reconstruct it and to insert the missing type declaration. In particular, the type inferencer can reconstruct only those types which are visible in the module (cf. Section 6). Each type inferencer of a Curry implementation must be able to insert the types of the parameters and the free variables (cf. Section 2.5) for each rule. The automatic inference of the types of the

defined functions might require further restrictions depending on the type inference method. Therefore, the following definition of a well-typed Curry program assumes that the types of all defined functions are given (either by the programmer or by the type inferencer). A Curry implementation must accept a well-typed program if all types are explicitly provided but should also support the inference of function types according to [14].

A type expression is either a type variable, a basic type like Bool, Success, Int, Float, Char (or any other type constructor of arity 0), or a type constructor application of the form (T 1 . . . n)

- Record

  A record is a data structure for bundling several data of various types. It consists of typed data fields where each field is associated with a unique label. These labels can be used to construct, select or update fields in a record.

  Unlike labeled data fields in Haskell, records are not syntactic sugar but a real extension of the language. The basic concept is described in [20] but the current version does not yet provide all features mentioned there. The restrictions are explained in Section 3.3.7.

4. CONTRAINTES

The type Success is the result type of expressions used in conditions of defining rules. Since conditions must be successfully evaluated in order to apply the rule, the type Success can also be interpreted as the type of successful evaluations. A function with result type Success is also called a constraint.

Constraints are different from Boolean-valued functions: a Boolean expression reduces to True or False whereas a constraint is checked for satisfiability. A constraint is applied (i.e., solved) in a condition of a conditional equation. The equational constraint e1=:=e2 is an elementary constraint which is solvable if the expressions e1 and e2 are evaluable to unifiable data terms. success denotes an always solvable constraint.

Constraints can be combined into a conjunction of the form c1 & c2 & . . . & cn by applying the concurrent conjunction operator &. In this case, all constraints in the conjunction are evaluated concurrently. Constraints can also be attached to arbitrary expressions by the operator &>, i.e., the constrained expression c &> e is evaluated by first solving constraint c and then evaluating e. This operator can also be used to enforce a sequential order for constraint evaluation, e.g., the combined constraint c1 &> c2 will be evaluated by first completely evaluating c1 and then c2.

Constraints can be passed as parameters or results of functions like any other data object. For instance, the following function takes a list of constraints as input and produces a single constraint, the concurrent conjunction of all constraints of the input list:

conj :: [Success] -> Success

conj [] = success

conj (c:cs) = c & conj cs

The trivial constraint success is usually not shown in answers to a constraint expression. For instance, the constraint x*x=:=y & x=:=2 is evaluated to the answer {x=2, y=4}

5. Higher-order Functions

Warren [49] has shown for the case of logic programming that the higher-order features of functional programming can be implemented by providing a (first-order) definition of the application function.

Since Curry supports the higher-order features of current functional languages but excludes the guessing of higher-order objects by higher-order unification (as, e.g., in [25, 38, 43]), the operational semantics specified in Figure 2 can be simply extended to cover Curry's higher-order features by adding the following axiom (here we denote by the infix operator "@" the application of a function to an expression):

Eval[['(e1, . . . , em)@e]]) '(e1, . . . , em, e) if ' has arity n and m < n

Here are some "traditional" higher-order functions to show that the familiar functional programming techniques can be applied in Curry. Note that the functions map, foldr, and filter are predefined in Curry (see Appendix B).

```
-- Map a function on a list (predefined):
map :: (t1->t2) -> [t1] -> [t2]
map f [] = []
map f (x:xs) = f x:map f xs
-- Fold a list (predefined):
foldr :: (t1->t2->t2) -> t2 -> [t1] -> t2
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
-- Filter elements in a list (predefined):
filter :: (t -> Bool) -> [t] -> [t]
filter p [] = []
filter p (x:xs) = if p x then x:filter p xs
else filter p xs
-- Quicksort function:
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = quicksort (filter (<= x) xs)
++ [x]
++ quicksort (filter (> x) xs)
```

6. OBJET : PROBLEM

La première approche que nous avons essayé d'implémenter est la modélisation d'un problème par un ensemble de listes [[String]] :

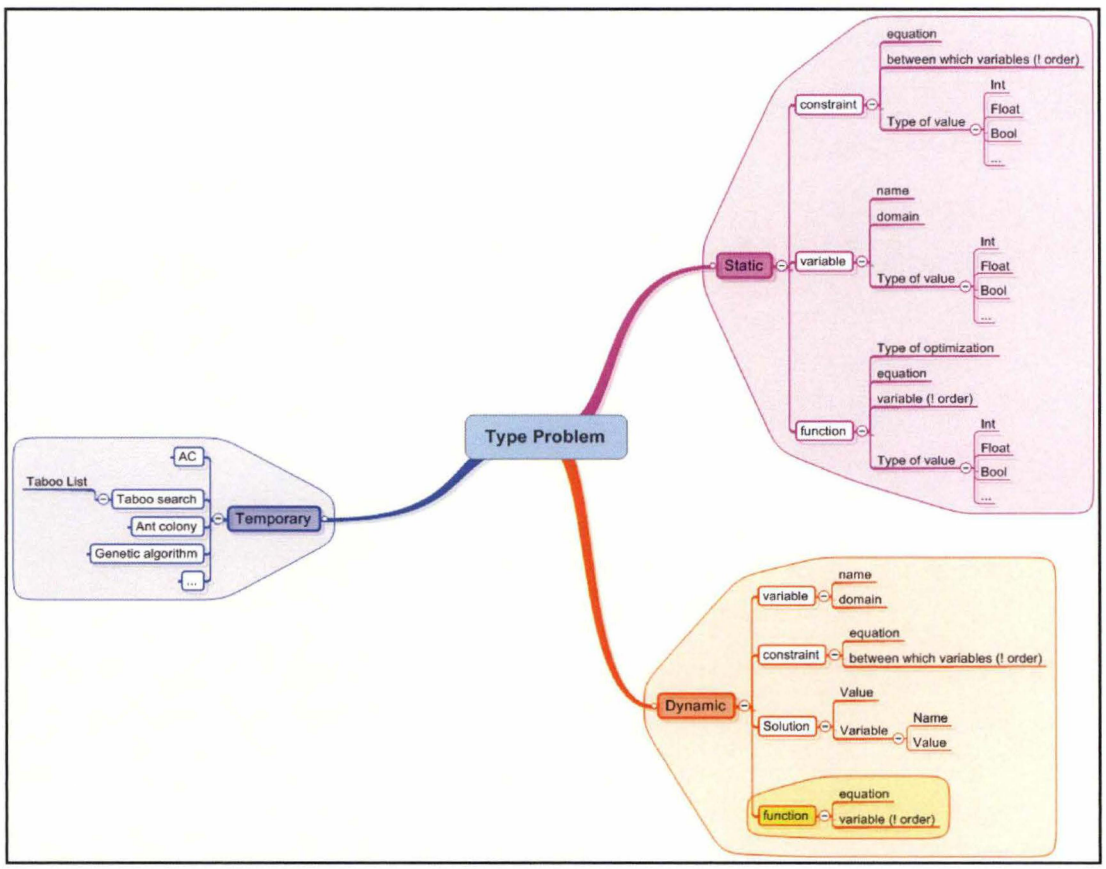
Ceci nous donne :

[[V1,V2,..,Vn],
[[D1],[D2], .., [Dn]],
[« C1 », « C2 », .., « Cm »],
[« fct à optimiser »]]

En partant sur cette modélisation, nous avons dû développer une fonction de parsing pour pouvoir retrouver l'ensemble des informations.

Les premiers tests nous ont permis de nous familiariser avec Curry mais aussi de nous rendre compte que notre problème n'était pas actif. Les contraintes et la fonction étaient une suite de caractères. Nous avions à notre disposition la possibilité de « recréer » les fonctions et de les rendre actives en utilisant la fonctionnalité de fonctions d'ordre supérieur.

Après quelques essais, la décision a été prise d'abandonner cette première implémentation de notre modèle. Il ne nous permet pas, par exemple, mélanger des structures différentes telles que [String] et [[String]].



[BKNHRS]

Figure 11 : Carte heuristique - Typage d'un problème

En parallèle, nous avons avancé sur la construction de notre premier opérateur et nous avons commencé à avoir une première idée des informations qui seraient nécessaires.
Nous avons résumé ces étapes dans une carte heuristique figure 11.

Nous avons par la suite essayé deux autres possibilités :

- les records ;
- un arbre.

Celles-ci n'ont pas apporté une solution à la demande de rendre le modèle vivant et utilisable mais ont permis de penser structure et dynamique.
Nous avons attaqué les sous-projets en parallèle et nous avons avancé sur les opérateurs (AC : arc concistency et TS : taboo search). Le but d'indépendance nous a permis de faire une liste d'informations propres au problème que nous devions pouvoir retrouver via des fonctions.

Fonctions liées à Problem :

- Afficher le problème
  printPob:: Problem -> IO()
- Afficher les variables
  printVar:: Problem-> IO()
- Afficher le domaine de validité de chaque variable
  printDom:: Problem-> IO()
- Afficher les contraintes
  printConstraint :: Problem-> IO()
- Afficher la fonction à optimiser
  printFct:: Problem-> IO()
- Tester la validité d'un problème
  checkProb:: Problem -> Success
- Tester un tuple vis-à-vis des contraintes
  checkTuple:: [Valeur] -> Problem -> Success
- Mise à jour du domain d'une variable
  updateDom:: Domain-> Variable -> Problem -> Problem

- Mise à jour de la solution
  updateSol:: [Valeur] -> Problem -> Problem – La solution étant le calcul de la fonction pour le tuple [Valeur]
- Mise à jour de la fonction
  updateFct:: Fonction -> Problem -> Problem
- Calcul de la valeur de la fonction en fonction d'un tuple
  valFct:: [Valeur] -> Problem -> Valeur
- Utiliser la fonction d'optimisation
  useOpt:: Problem -> (Valeur -> Valeur)
- ...

L'ensemble comprenant notre type problème et ses fonctions devient notre objet problème.
Nous avons essayé en parallèle la modélisation en arbre et en record. La modélisation en arbre a elle-même été testée sous forme d'arbre à branches multiples figure 12 et sous la forme d'un arbre binaire figure 13.

La recherche dans un arbre à branches multiples n'est pas des plus performantes car elle ne peut pas être facilement automatisée et donc cette solution a été rapidement rejetée. Par contre, l'approche sous forme d'arbre binaire permet un parcours récursif, une théorie solide et des algorithmes existent sur le sujet.
Nous avons aussi testé et validé le choix de la modélisation sous forme de record.

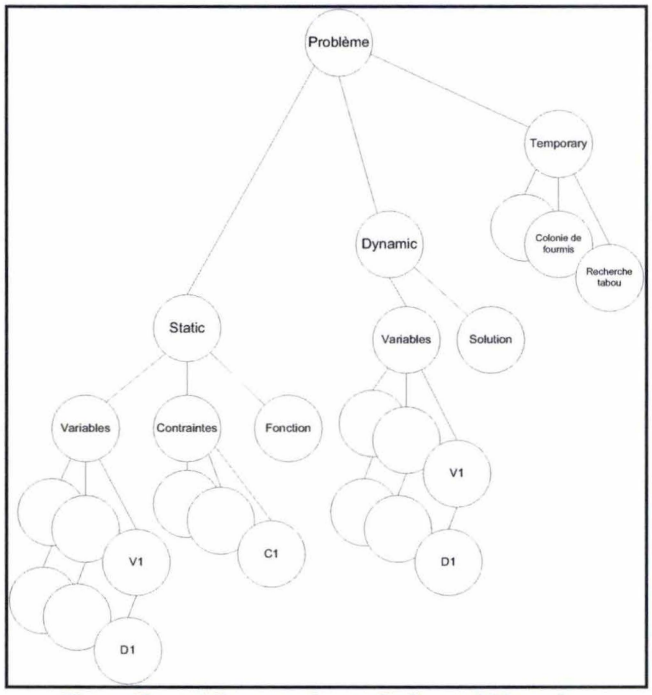


Figure 12 : problème sous forme d'arbre multi branches

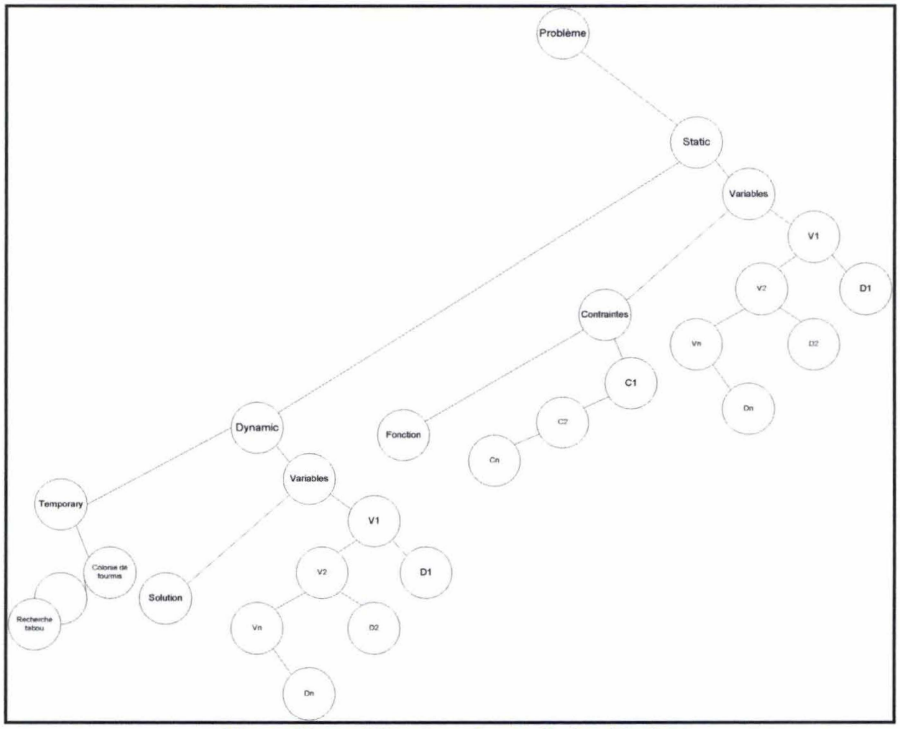


Figure 13 : problème sous forme d'arbre binaire

A ce stade, nous avons une modélisation sous forme de records ou sous forme d'arbre binaire dont chaque élément est de type String.

7. PRÉCOMPILATION

Problème du type CSOP a domaine fini
3 variables [x,y,z]
x dans le domaine fini [0..5]
y dans le domaine fini [0..10]
z dans le domaine fini [-3..4]
les contraintes sur les variables sont
x*y=x+y
x<z
z>=y
la fonction à optimiser
f(x,y,z) = ^2 + 2*x*y + y^2 - z

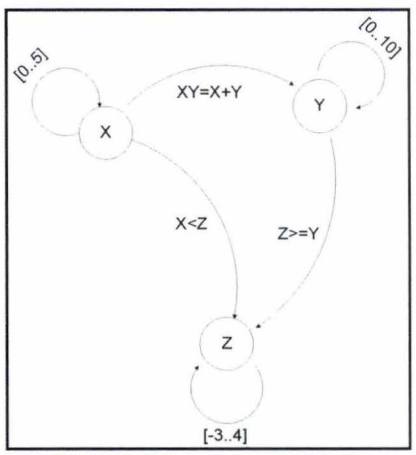


Figure 14 : Graphe du problème

Nous avons pour le moment une vue statique d'un problème, c'est-à-dire que nous avons défini une manière de présenter un problème dans son état initial. Cette représentation ne suit aucune règle d'aucun langage, ni aucun formatage permettant l'utilisation d'un opérateur ou autre pour permettre le traitement du problème.

Ceci nous posera un problème lorsque nous tenterons d'utiliser l'information contenue dans notre représentation. Comment utiliser les contraintes, comment utiliser les domaines de définition des variables si ce n'est en exécutant un parsing des données ?[Huttc]
Le parsing réalisé, nous avons devant nous des listes de string. Ces listes ne sont pas actives, elles représentent le problème, le décrivent mais cela s'arrête là.
Nous avons utilisé les possibilités de Curry telles que les listes de contraintes en lieu et place de listes de string ainsi que les fonctions d'ordre supérieur, ce qui nous a permis de passer une fonction comme variable.
Avec cet ensemble de listes, nous avons réussi à activer une partie de l'information disponible mais il nous est clairement apparu que cette partie était insuffisante.

Voici une fonction permettant de valider si une solution (tuple) répond à l'ensemble des contraintes du problème :

```
[["x","y"],
 ["[1,2,3]","[0..5]"],
 [["x*y=x+y"]],
 ["x+2*y-5"]]
```

```
42 check :: [Int] -> Success
43 check sol =
44   sol =:= [x,y] &
48   elem2 x [1,2,3] &
49   domain [y] 0 5 &
51   c1 x y &
52   x >=# y &
53   labeling [] sol
54   where x,y free
```

Nous pouvons voir ici ce que nous propose Curry pour la résolution de problèmes à domaine fini (cfr Annexe CLPFD), ensuite le codage d'un espace par ses bornes et enfin l'utilisation d'une fonction elem2 (elem2 :: a -> [a] -> Success) qui n'est que l'extension de la fonction native elem (type : elem :: a -> [a] -> Bool). « c1 » représente la contrainte : $x *\# y =\# x +\# y$.

A la vue des limitations rencontrées, du nombre de cas à imaginer et surtout de l'attente d'indépendance souhaitée, nous avons étudié une toute autre approche. Nous savons que notre objectif final est d'utiliser un ou des opérateurs dans un langage Curry pour solutionner un problème donné, donc pourquoi ne pas compiler notre problème pour qu'il puisse être codé en Curry et que surtout cela nous permette d'utiliser les librairies actuelles mais aussi futures ?

Nous allons donc avoir la représentation de notre problème et de l'ensemble des fonctions de gestion qui sont nécessaires et ensuite leur compilation en Curry.

Ceci permettra de reconnaître certains types du problème et d'importer, si nécessaire, la librairie CLPFD, CLPR, CLPB, ... tout en ne perdant pas de vue le fait que nous voulons implémenter une recherche encapsulée.

Il est à remarquer que le but n'est pas d'utiliser les fonctionnalités de recherche existantes en Curry mais bien de rendre vivantes les données de notre problème.

Pour réaliser cela nous sommes partis sur une précompilation de la définition du problème.
Nous partons d'une représentation simple et facilement gérable par l'utilisateur pour le traduire sous un format Curry. Ce passage nous permettra de rendre l'ensemble des informations actif.
Le choix entre une représentation sous forme d'arbre binaire et sous forme de record a été dicté par la nécessité de construction. En effet, une modélisation sous forme d'arbre demande la création d'un constructeur pour l'objet problème, tandis que la solution record demande uniquement une réécriture du problème en Curry. De plus, la compilation nous permet la validation « Curry » du problème modélisé.
Le type problem sera codé de la manière suivante :

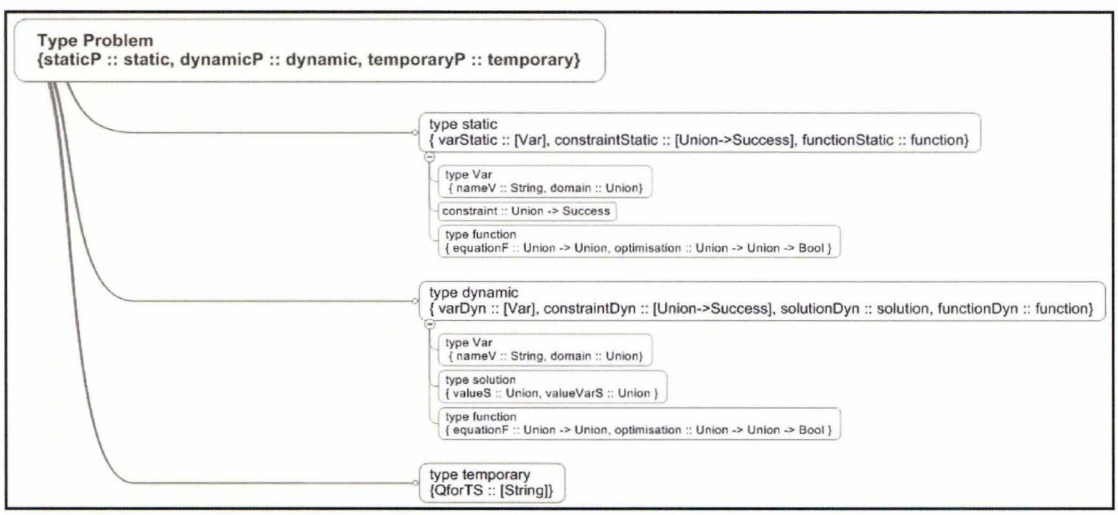


Figure 15 : problème sous forme de records

Nous avons avec cette modélisation la possibilité de modifier les domaines, les valeurs, la fonction d'optimisation.
Un point d'attention persiste : le typage des variables.[Hutta] Nous avons utilisé la caractéristique de polymorphisme de Curry qui permet d'avoir une modélisation plus simple, seulement nous devrons en tenir compte lors de la modélisation des domaines des variables. Dans un problème de type CSP, nous aurons la possibilité d'énumérer le domaine mais dans

le cas de problème à domaine continu, nous n'aurons que des contraintes pour définir le domaine (par exemple cVar1, cVar2, ... dans l'exemple qui suit)

```
1 data Union = Reel Float | Entier Int | Binaire Bool | Fini [Int]
2
3 type variable = { nameV :: String, domain :: [Union]}
4
5 -- constraint :: [Union] -> Success
6
7 type function = { equationF :: [Union] -> Union, optimisation :: Union -> Union -> Bool }
8
9 type solution = { valueS :: Union, valueVarS :: [Union] }
10
11 type static = { varStatic :: [variable], constraintStatic :: [[Union]->Success], functionStatic ::
function}
12 type dynamic = { varDyn :: [variable], constraintDyn :: [[Union]->Success], solutionDyn ::
solution, functionDyn :: function}
13 type temporary = {QforTS :: [String]}
14
15
16 type problem = {staticP :: static, dynamicP :: dynamic, temporaryP :: temporary}
```

Voici la modélisation du problème exposé à la figure 14.

```
18 ------------------------------------------------
19 -- Instanciation d'un probleme P1
20 -- Probleme du type CSOP a domaine fini
21 -- 3 variables [x,y,z]
22 -- x dans le domaine fini [0..5]
23 -- y dans le domaine fini [0..10]
24 -- z dans le domaine fini [-3..4]
25 -- les contraintes sur les variables sont
26 --    x*y=x+y
27 --    x<z
28 --    z>=y
29 -- la fonction a optimiser
30 -- f(x,y,z) = ^2 + 2*x*y + y^2 - z
31 ------------------------------------------------
32

40 var1 = {nameV = "x", domain = Fini[0..5]}
41 cVar1 (Fini [x,_,_]) = ( elem x [0..5] ) =:= True
42
43 var2 = {nameV = "y", domain = Fini[0..10]}
44 cVar2 (Fini [_,y,_]) = ( elem y [0..10]) =:= True
45
46 var3 = {nameV = "z", domain = Fini[-3..4]}
47 cVar3 (Fini [_,_,z]) = ( elem z [-3..4]) =:= True
48
49 c1 (Fini [x,y,_]) = x * y =:= x + y
50 c2 (Fini [x,_,z]) = (x < z) =:= True
51 c3 (Fini [_,y,z]) = (z >= y) =:= True
52
53 f (Fini [x,y,z]) = Entier (x * x + 2 * x * y + y * y - z)
54
55
56 functionP1 = {equationF = f, optimisation = (>) }
57 solutionP1 = {valueS = Entier 0, valueVarS = (Fini [0, 0, 0])}
58
```

```
59 staticP1 = { varStatic = [var1,var2,var3], constraintStatic = [cVar1,cVar2,cVar3,c1,c2,c3],
functionStatic = functionP1}
60 dynamicP1 = { varDyn = [var1,var2,var3], constraintDyn = [cVar1,cVar2,cVar3,c1,c2,c3], functionDyn
= functionP1, solutionDyn = solutionP1}
61 temporaryP1 = {QforTS = []}
62
63 problemP1 = { staticP = staticP1, dynamicP = dynamicP1, temporaryP = temporaryP1}
```

La zone mémoire temporaryP1 ne contient rien pour le moment. Elle devra être composée d'espaces formatés pour mémoriser les informations temporaires des métaheuristiques utilisées.

8. Indépendance de la représentation du problème de sa résolution

A ce stade, nous pouvons dire que nos données du problème n'ont aucune connaissance de la manière dont elles vont être utilisées et que, par la même occasion, la personne codant son problème n'est pas consciente de la suite qui va être donnée.
Nous pouvons toutefois nous poser les questions de la représentation et de l'utilisation dans Curry après la compilation. Nous nous souvenons qu'après chaque étape lors de la résolution d'un problème, nous avons un nouveau problème. Ce qui, pour rappel, signifie que la sortie de la compilation n'est que la représentation du problème dans son état initial.

Nous avons introduit une modélisation plus large du problème, valable lors de l'ensemble des itérations, visant à le solutionner. Une fois cette modélisation introduite, nous avons généralisé le modèle pour qu'il intègre à la fois les étapes de résolution et les étapes d'initialisation et de fin.

Un nouveau problème résultant de l'application d'un opérateur de solution est défini par trois grands types de données :

1. Les données statiques reprenant :
   - l'ensemble des contraintes,
   - le type de données à traiter,
   - le nombre de variables,
   - la fonction à optimaliser,
   - le type d'optimalisation : maximiser, minimiser, … .

2. Les données dynamiques :

   celles-ci englobent les domaines de validité des variables. Ces domaines ne sont pas statiques, l'un des opérateurs présentés a pour but de diminuer ceux-ci en respectant la consistance d'arc (par ex). A la sortie d'AC, nous devrions trouver le même problème avec les mêmes données statiques mais avec des domaines de validité des variables vérifiés.

   Nous y trouvons aussi le meilleur résultat du moment pour le problème.

3. Les données temporaires :

   elles contiendront l'ensemble de la « mémoire » nécessaire aux opérateurs lors de la résolution du problème. Il peut s'agir par exemple de la liste tabou. Nous espérons faire de ces données temporaires liées à la vie d'un problème, c'est-à-dire de sa création à l'arrêt fonction des conditions d'arrêt du problème, une zone de partage entre chaque sous-étape dans un opérateur comme entre opérateurs. Cette approche aura de l'importance lorsque nous parlerons d'opérateur hybride.

   D'un point de vue pratique, le développement du passage du fichier comprenant les données d'un problème à la création du fichier Curry a été réalisé par l'implémentation d'un simple script.

   Le fichier obtenu par l'étape de compilation sera la base pour mettre en place la résolution du problème qui est l'objectif du chapitre suivant.

CHAPITRE 6 : RÉSOLUTION D'UN PROBLÈME

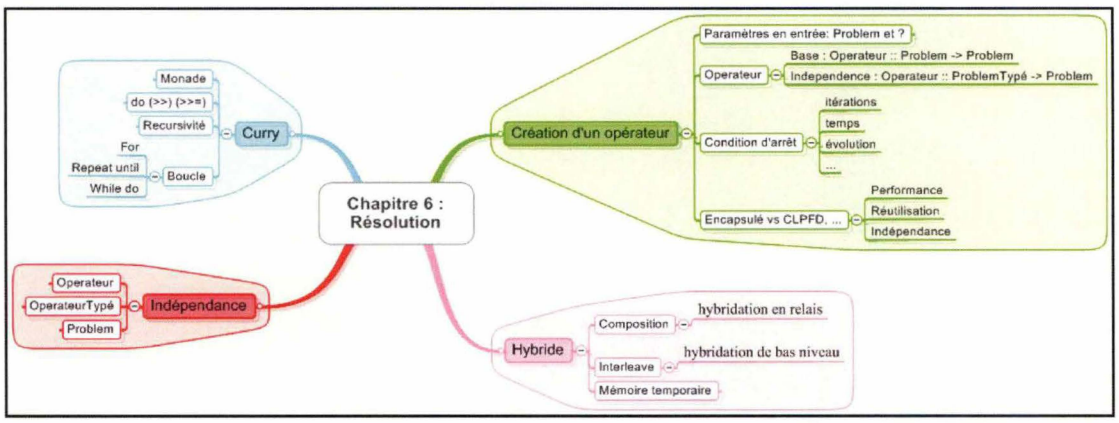


Figure 16 : Carte heuristique - Résolution d'un problème

1. INTRODUCTION

Le premier problème auquel nous avons dû faire face est la compréhension des heuristiques que nous voulions implémenter.

En effet, la plupart des algorithmes sont donnés sous forme de pseudo-code mais de manière impérative. Notre volonté étant d'utiliser un langage fonctionnel logique, nous devons donc en premier lieu ne plus tenir compte de ce mode devenu naturel qu'est l'impératif pour passer au déclaratif.

Le déclaratif est naturel, du moins pour les personnes sans notion de programmation et ayant un fort esprit logique. Mais, qui dit naturel sous-entend que l'on doit maîtriser son sujet pour pouvoir en parler de manière simple et structurée.

Parler d'un problème ne pose pas de problème .Chacun de nous a des problèmes et peut en parler mais combien savent parler de leurs problèmes de manière claire ?

Partons du principe que le problème est clairement connu.

Pour la phase suivante, qui est la résolution du problème, il est attendu de pouvoir décrire chaque étape de telle manière que cette description soit, en soi, la source de l'implémentation de la résolution du problème.

Une contrainte supplémentaire existe, puisque nous recherchons la création d'opérateurs qui doivent être indépendants de la manière dont le problème a été introduit. Cette contrainte est rencontrée par l'ensemble des primitives qui ont été créées avec l'objet Problem.

Dans la suite du chapitre, nous montrerons l'importance et la complexité de gérer la condition d'arrêt de la recherche.

Pour finir et avant de conclure ce chapitre, nous allons montrer les interactions possibles entre opérateurs[Annexe Taxinomie de l'hybridation] :

- la première, la plus simple : l'utilisation d'un et un seul opérateur qui va faire évoluer le problème vers sa solution.
- La deuxième, logique et attendue : nous utilisons un premier opérateur pour faire évoluer le problème et lorsque celui-ci semble avoir atteint sa condition d'arrêt, on repart du problème avec un autre opérateur en partant du nouveau problème généré par l'opérateur précédent. Nous pouvons parler de composition d'opérateur, cette approche ne s'arrête pas à 2 opérateurs.
- La troisième, la plus complexe : un opérateur de résolution fait appel à un autre opérateur pour résoudre une de ses étapes. Ceci signifie que nous avons un problème qui contient en son sein une série de sous-problèmes dont la nature demande une résolution par un autre opérateur. Toujours pour nous assurer d'avoir cette indépendance, nous utiliserons les fonctions d'ordre supérieur qui nous permettront de passer l'opérateur de résolution en fonction de la volonté du demandeur ou en fonction du nouveau type de problème que l'on vient de rencontrer.

Remarque préliminaire : il est à signaler que dans la littérature relative à la recherche encapsulée dans le cadre « functional logic language », on parle de l'utilisation de

l'opérateur try. Lors de nos recherches et essais, cet opérateur était bien défini dans le prélude de l'implémentation de Curry utilisé mais n'est pas implémenté. [CurryM]

memoire/GA> :type try
try :: (a -> Success) -> [a -> Success]

ERROR:
! 'Prelude.try not yet implemented!'

2. Création d'opérateurs

Nous avons besoin d'une bonne connaissance de l'élément à implémenter. Nous pouvons nous référer au chapitre 4 pour les métaheuristiques, au chapitre 2 pour le langage et enfin au chapitre 5 pour une proposition de modélisation d'un problème..
Nous avons besoin de techniques de base permettant la création de code optimisé. Ces techniques sont disponibles via des lectures de livres de références dans les différents paradigmes.
Nous allons avoir besoin de traduire la notion de repeat until, ceci se traduit grâce à l'utilisation de la caratéristique monade[Vanh][chapitre 2]

```
until :: (a -> IO Bool) -> (a -> IO a) -> a -> IO a
until p f x = do
  test <- p x
  if test
    then return x
    else f x >>= until p f
```

Nous avons besoin d'un environnement de test. Nous avons réalisé une installation sur un serveur linux. Nous avons utilisé une version de démonstration de Sicstus prolog 3.12 et la version 1.6 de Pakcs.

3. Condition d'arrêt

Temps, itérations, variations, ... Voici une liste non exhaustive de conditions d'arrêt. Nous pouvons remarquer qu'aucun algorithme pris dans sa généralité n'est certain de trouver une solution et s'il converge vers cette solution, on ne connaît pas le temps qui lui sera nécessaire pour y arriver. Le but des heuristiques étant de converger vers une approximation acceptable de la solution, nous nous devons de laisser à l'utilisateur de nos opérateurs un maximum de liberté quant aux conditions d'arrêt.

Toutefois, la condition nombre d'itérations est pauvre en sens car cela est fonction de la puissance de calcul mise à disposition. Les conditions indépendantes de l'environnement d'exécution sont à notre avis les plus pertinentes. Ces conditions sont toutes fonctions de l'algorithme qui sera codé. Le nom donné à la condition est aussi en relation avec l'historique de l'heuristique : [Tsan]

- énergie/température : Recuit simulé
- phéromones : Colonie de fourmis
- ...

Dans la plupart des lectures d'algorithmes réalisées dans le cadre de cette étude, nous avons pu constater que cette donnée de condition d'arrêt est un point qui n'est pas explicité. La question qui nous vient directement à l'esprit est : comment gérer cette condition ?

Deux possibilités nous sont offertes :

1. permettre la gestion de la condition par la personne utilisant l'opérateur. En effet, celle-ci est la plus à même de déterminer la valeur qu'elle espère obtenir, ayant introduit la fonction à optimiser (cfr chapitre précédent). Il faudra toutefois ajouter des garde-fous pour ne pas entrer dans un opérateur bouclant et ne rencontrant jamais sa condition d'arrêt.
2. Prendre en charge cette condition d'arrêt. Ceci implique de connaître les attentes de tout le monde vis-à-vis de l'ensemble de leurs problèmes.

Il est évident que nous allons partir vers la possibilité 1 avec un garde-fou. Nous essayerons de proposer à l'utilisateur une liste exhaustive de conditions à remplir pour utiliser l'opérateur de manière optimale. Nous essayerons de faire du pattern matching pour réaliser le bon choix en fonction des informations reçues.

La question qui peut être posée maintenant est ce que nous devons faire de ces compléments d'information. Ils sont nécessaires pour tendre vers une solution acceptable pour le demandeur mais ne font pas partie du problème.
A la vue de ceci, nous pourrions recommencer une discussion sur ce qu'est le problème. Est-ce le problème tel que présenté en introduction à ce document ou alors est-ce la résolution du problème qui est notre problème ou n'est-ce pas tout simplement les deux ?
La personne doit avoir une idée de la résolution si on la veut optimale, en contre partie on souhaite ne rien devoir savoir que pour utiliser un opérateur. Ceci n'est pas à considérer comme étant un dilemme mais comme étant antagoniste.
Nous allons prendre les algorithmes génétiques pour illustrer ce point.
Nous avons une fonction
GA :: population -> population
Population étant un record contenant comme information d'arrêt le nombre maximum de générations que l'on peut générer ou la pertinence à partir de laquelle nous ne créerons plus de nouvelle génération. Ces deux conditions seront reprises dans une fonction
isDone :: population -> Bool.
Ce qui va nous donner une boucle d'exécution GA = until isDone CréerNouvellePopulation.

4. HYBRIDE

Dans la littérature, nous trouvons de plus en plus de recherches montrant que l'on tend plus vite vers une solution si l'on mixe des méthodes de recherche. L'une des premières idées est d'utiliser un algorithme de filtrage (AC, NC, ...) et ensuite une recherche tabou.
Nous allons mettre en avant une utilisation possible de nos opérateurs, qui réponde à ces deux visions d'hybridation.[VaDu]

Pour une hybridation TS(AC(Problem)), cela correspond à la diminution de l'espace de solution en première étape suivi d'une recherche tabou. Dans le cadre de notre problème, l'exécution de AC sur un domaine de solution initial
-- x dans le domaine fini [0..5]
-- y dans le domaine fini [0..10]

-- z dans le domaine fini [-3..4]

donne comme résultat :

memoire/AC> valideDomainsProblem

Result: [[0,2],[0,2],[1,2,3,4]]

More solutions? [Y(es)/n(o)/a(ll)]

a. Partage de la mémoire

Ce point va introduire une proposition de structure de la zone temporaire dont nous avons montré l'existence dans le Chapitre 5.8.2.

Dans le cas d'une hybridation, quelle est l'information que l'on doit pouvoir partager entre les deux opérateurs ?

Comment formaliser, structurer cette zone d'échanges ?

Il est évident que les informations de définition du problème doivent être partagées. Ceci comporte les données statiques mais il en est de même pour les données dynamiques telles que le domaine de validité des variables, la meilleure solution du moment.

En effet, c'est en se basant sur ces données que le nouvel opérateur va se mettre à rechercher une amélioration. Le passé est toujours une source d'information et doit donc être traité comme tel. Ceci implique une structure pour conserver ces informations.

Si nous avons cette structure, il va avoir besoin de primitives de gestion pour continuer à prétendre à notre indépendance de codage.

Nous avons donc l'énoncé du problème, nous avons à un moment et sa meilleure solution et ses domaines de définition. Mais qu'en est-il de ce qui a permis d'arriver à la situation du temps t ? Nous sommes passés par une série d'étapes qui ont permis d'éliminer des solutions car elles étaient moins bonnes que l'une proposée par un de ses successeurs. De plus, suivant les heuristiques choisies, nous avons des informations à conserver entre deux runs, n'est-il pas utile de garder trace de ces informations et de les proposer à l'heuristique suivante ?

Existe-t-il une correspondance entre les informations de l'heuristique algorithme génétique et la recherche tabou ?

Nous n'avons pas pu nous pencher comme nous l'aurions voulu sur cette question mais nous sommes partis du principe qu'il doit exister un intérêt et un lien entre les heuristiques et les informations qu'elles brassent.
Ces idées et approche demandent l'écriture de primitives de passage d'un format vers un autre.
Nous pourrions imaginer une fonction :
Translate :: Operateur -> Operateur -> Problem -> Problem
Le but serait de transformer le maximum d'informations de la zone temporaire de l'opérateur 1 vers l'opérateur 2. Nous aurions comme résultat un problème prêt de part l'indépendance de notre modélisation[chapitre 5] et ayant déjà une série d'information provenant de l'exécution précédente.

b. SÉQUENTIELLE (COMPOSITION)

Cette utilisation d'un opérateur suivi d'un deuxième est naturelle, il s'agit d'une simple composition de deux fonctions : (f o g) x.
Quand utiliser cette composition ? Il est certain que ce sont les heuristiques qui vont tendre vers la solution et pas les solutions filtrage qui elles sont présentes pour diminuer le domaine des variables à des valeurs permises.
Ceci signifie que, si l'on filtre en premier et qu'ensuite on recherche un tuple optimal pour le problème, la recherche heuristique ne se perdra pas dans des valeurs non permises.

Le passage d'un opérateur de filtrage à une heuristique ne demande aucune réflexion pour les raisons suivantes :

- l'objectif poursuivi n'est pas le même et il en est de même pour les données modifiées.
- Opérateur :: Problem -> Problem. Nous avons comme résultat un problème et nous utilisons ce résultat comme entrée pour l'opérateur suivant.
- Nous passons l'ensemble des données relatives au problème à l'opérateur suivant, cet ensemble est, pour rappel, les données statiques, dynamiques et temporaires.

Le passage d'une heuristique à une autre heuristique demande l'étude de primitives de mutation pour les données temporaires. Dans le cadre de notre travail, nous n'allons passer que les données statiques et dynamiques. Si nous nous attardons sur les données dynamiques, les heuristiques ne vont pas modifier la définition des domaines de validités mais uniquement la solution au moment t qui est optimal pour le problème. Donc la seconde heuristique aura comme référence la solution optimale trouvée par l'heuristique précédente pour le problème.
Cette valeur est liée à un tuple qui sera aussi passé d'heuristique en heuristique. Nous pouvons utiliser, si besoin, cette information comme point de départ pour certaines heuristiques.

Une remarque doit être faite au regard de notre réflexion et de certaines heuristiques : nous avons établi que notre fonction à optimiser est une donnée statique, ce qui semble évident car le contraire impliquerait que nous changeons le problème à résoudre.

Malheureusement un point n'a pas été abordé, le codage du problème pour utiliser l'heuristique. Un exemple est l'algorithme génétique qui demande un codage de la fonction sous forme d'une suite de bits. C'est cette liste de bits qui est manipulée et qui fournit les valeurs des variables.
Est-ce que cela remet en cause notre modélisation d'un problème ?
Non, mais nous devons en tenir compte, même si la littérature montre que, par exemple, ce codage binaire pose des problèmes et que de nouvelles propositions de représentation des problèmes valides pour l'heuristique algorithme génétique arrivent.
Toutefois, il serait intéressant de se pencher sur cette problématique de la représentation de l'élément à optimiser. En tout cas, il est certain que ce codage propre à une heuristique va à l'encontre de nos attentes d'indépendance problème/résolution.

c. INTERLEAVE

Dans le point précédent, nous avons laissé à un opérateur tout loisir de manipuler notre problème pour nous fournir sa réponse. Cette réponse est un nouveau problème qui peut être réutilisé par un second opérateur moyennant la remarque sur les codages spécifiques à certaines heuristiques.[BaHa]

Dans cette vision que nous avons baptisée « interleave / tissé » , nous poursuivons le codage d'opérateur hybride représentant le courant actuel.

La littérature nous montre que, par expérience, on a constaté que pour certains problèmes, certaines classes d'heuristiques tendent très rapidement vers une première approximation de la solution avant d'arriver à un palier.

Ensuite, lors des itérations, on se rend compte que l'on a affaire à des sous- problèmes NP-complet pouvant être optimisés par une autre heuristique.

Ceci nous met face à deux nouveaux problèmes :

- classification des problèmes pour arriver à utiliser la bonne heuristique pour les résoudre ;
- utilisation d'heuristiques pour certaines étapes de choix dans une heuristique.

Notre premier problème n'en est pas un, nous pouvons en premier lieu estimer qu'il reste celui de la personne voulant le résoudre. Nous lui proposons un environnement de travail dans lequel un maximum d'éléments n'ont pas d'interaction, mais le choix de l'heuristique reste à sa discrétion. Nous ne lui proposons pas un opérateur : Solutionne :: Problem -> Solution -> Success

Par contre, le deuxième point fait partie de notre problématique. Nous avons fait un choix de précompilation pour passer de la description du problème à son codage en Curry pour rendre actif un maximum d'informations et utiliser les fonctionnalités, librairies de Curry.

La question du début devient, pouvons-nous imbriquer nos opérateurs dans un opérateur et passer l'ensemble des informations nécessaires à son utilisation ?

Contrairement à la composition, dans le tissage le problème que cherchent à résoudre les opérateurs imbriqués n'est pas le même que le problème de base.

Nous pouvons prendre le cas d'un algorithme génétique hybride qui utilise l'heuristique de recuit simulé mais aussi l'heuristique de colonie de fourmis utilisant une heuristique de recherche locale telle que la recherche tabou.

Dans le cadre de cette étude, nous ne prendrons pas parti sur l'intérêt et les effets de bord d'introduire des heuristiques dans une heuristique.

Les conditions et les types de problèmes ou d'étapes sont complexes et demanderaient à eux seuls une réflexion au cas par cas.

Nous allons donc nous limiter à essayer de proposer une solution ou du moins un embryon de solution permettant cette imbrication et nous laisserons le choix des heuristiques à la discrétion de l'utilisateur.

Pour illustrer l'hybridation, nous pouvons prendre l'algorithme génétique couplé à une recherche tabou.

La configuration standard d'un algorithme génétique est :

Type configGA = {cConfig ::ConfigChromosome, pConfig :: ConfigPopulation, newPopulation ::Fonction, maxFitness ::Int,} maxGeneration ::Int, gen ::Rand}

Avec dans notre zone mémoire Temporary la donnée : currentGeneration.

Lors de chaque itération, des parents sont choisis pour générer la nouvelle population. L'hybridation servira à améliorer les nouveaux individus en faisant une recherche tabou sur ces individus.

La recherche tabou est un exemple mais nous pourrions utiliser n'importe quelles heuristiques dites à recherche locale, recuit simulé en est une autre.

Nous devons fournir une condition d'arrêt pour la recherche tabou sous forme d'itérations maximum.

5. Meilleur Choix vs Classification

Ce chapitre survolera une problématique qui est régulièrement rencontrée dans l'utilisation d'heuristiques lors de la résolution de problème NP-complet.

Y a-t-il un moyen de classifier les problèmes ?

Y a-t-il un lien entre cette classification et l'utilisation de certaines heuristiques ?

Mais avant toutes choses, pouvons nous certifier que l'on utilise le bon opérateur ou peut être la première question que l'on devrait se poser : doit on utiliser une recherche heuristique pour solutionner, cerner, tendre vers la solution de ce problème ?[FVW]

Conditions d'éligibilité d'un problème :

- nombre de solutions important : en effet, l'accroissement des performances des heuristiques par rapport aux algorithmes classiques est plus marqué lorsque les

espaces de recherches sont importants. En effet, pour un espace dont la taille est faible, il peut être plus sûr de parcourir cet espace de manière exhaustive afin d'obtenir la solution optimale en un temps qui restera somme toute correct. Au contraire, utiliser un algorithme génétique engendrera le risque d'obtenir une solution non optimale en un temps qui restera sensiblement identique.

- Pas d'algorithme déterministe adapté et raisonnable.
- Lorsque l'on préfère avoir une solution relativement bonne rapidement plutôt que d'avoir la solution optimale en une durée indéfinie.

En considérant que le problème est éligible, quelle direction prendre pour tendre rapidement vers une approximation de la solution ? Ce point restera ouvert dans le cadre de cette étude mais il serait bon de se pencher sur cette question qui revient régulièrement. Nous pouvons comparer cette question au choix que fait un artisan devant ses outils lorsqu'il doit effectuer une tâche précise avec un outil dédicacé.

La rapidité lors de l'exécution, la finesse mais aussi la qualité du travail sont fonctions de l'outil et de son utilisation. Nous nous attelons à rendre cette utilisation la plus simple en cachant sa complexité par son codage. Par contre, la validation du choix de l'outil ne peut être réalisée que par la connaissance du problème, de la tâche à exécuter.

- La première étape serait la création d'une liste exhaustive des types de problèmes et leurs caractéristiques, cette suite d'éléments les différenciant, on peut espérer pouvoir classifier bon nombre de problèmes.
- La seconde étape serait de réaliser une modélisation de la relation entre le type d'un problème et la ou les heuristique(s) pour le solutionner. Cette relation ne devrait certainement pas être 1 <-> 1 mais devrait plutôt tendre vers une relation n <-> m. Une simplification de cette relation sera nécessaire.
- Dernier point, l'existence d'opérateurs hybrides. Pour ces opérateurs, nous pouvons, sans trop nous avancer, estimer que leur fonctionnement vis-à-vis du monde extérieur est équivalent aux caractéristiques de l'heuristique de base. L'influence des « sous heuristiques » ne devant pas être ressentie à l'extérieur si ce n'est par l'augmentation de la performance globale.

6. EXEMPLE : AC

```
-- Problem

-- Record Problem

-- AC
-- Mise a jour des domaines des variables.
-- Parametre qui est la taille de la liste tabou et le nombre maximum d'itérations
AC :: configAC -> Problem -> Problem

---------------------------------
-- valideDomainsProblem
-- Permet la validation d'un domain

valideDomainsProblem :: Problem -> [Union]

---------------------------------
-- valideValuesDomain
-- Permet la validation d'une variable d'un domain
-- On donne en entree la variable a valider et l'ensemble des autres domaines des autres variables

valideValuesDomain :: Problem -> [Union]

---------------------------------
-- check
-- Permet de valider un tuple comme consistante visa vis des contraintes
-- Fonction cree lors de la compilation du probleme
-- Utilise les librairies en fonction du type de probleme
-- comprend les domaines
--          les contraintes
--          la fonction d optimisation


check [x,y,z] =
  x >= 1 &&
  x <= 3 &&
  elem y [0..5] &&
  elem z [0..10] &&
  x * y == x + y &&
  x >= y &&
  z > (4 + (y * x))
```

7. INDÉPENDANCE DE LA RÉSOLUTION DU PROBLÈME

Dans ce chapitre nous avons essayé de créer des opérateurs basés sur des heuristiques ou des méthodologies de filtrage. A chaque étape, nous nous sommes rendu compte que nous devions régulièrement apporter des modifications à la structure de la partie temporaire de l'Objet Problem. Aucune modification des parties statiques ou dynamiques n'a été nécessaire. Toutefois, il est également clair qu'un certain nombre de primitives doivent exister que pour pouvoir réaliser notre objectif.

Nos opérateurs ont besoin d'informations qui ne peuvent provenir que de la personne responsable du problème :

- type de problème

- condition d'arrêt
- opérateur à utiliser
- filtrage ou non
- hybride ou pas et si oui quelle heuristique avec quelle autre heuristique

A la vue de cet ensemble de conditions, nous ne pouvons pas conclure que nous avons rempli l'ensemble de nos attentes.
Notre pouvons accepter que notre modélisation d'un problème est indépendante de sa résolution mais, en aucun cas, nous ne pourrons réutiliser telle quelle une partie de ce travail.
L'ensemble forme un tout qu'il sera difficile de scinder, ce tout a un nom, c'est l'objet Problem qui nous arrive avec ses informations, données mais surtout avec les méthodes pour les utiliser. N'oublions pas aussi que ces méthodes existent en grande partie après la compilation vers Curry.

Pour conclure ce chapitre, nous avons ouvert beaucoup de portes, soulevé un grand nombre de questions, nous n'avons en aucun cas cherché à gérer ces nouvelles réflexions mais il est certain qu'une grande partie de ces voies est déjà balisé et que la recherche avance à grands pas dans ces problématiques.

CONCLUSION

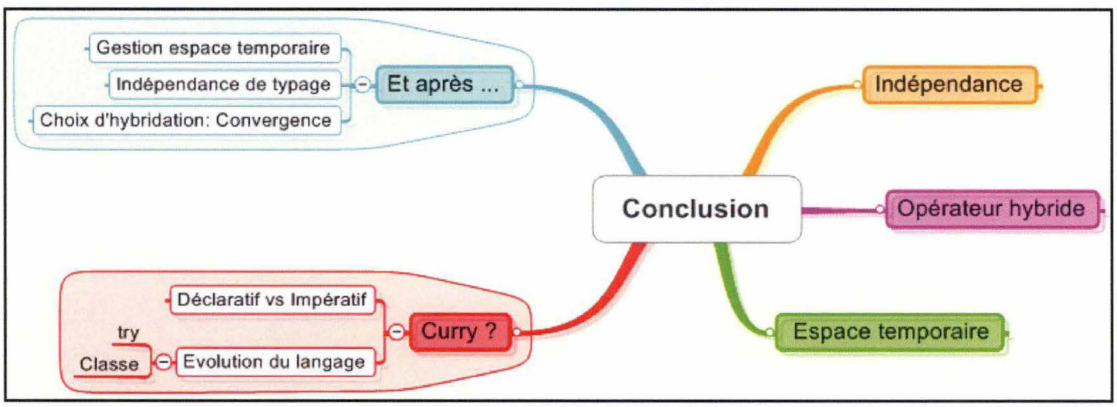


Figure 17 : Carte heuristique - Conclusion

Dans le cadre de cette étude, nous avons parcouru bon nombre de théories et de concepts, nous sommes partis à la recherche d'indépendance, nous nous sommes libérés de certaines contraintes dans le cadre de problèmes d'optimisation sous contraintes mais, toutefois, pouvons nous affirmer que nous avons rempli notre objectif ?

La vision de recherche encapsulée est d'actualité dans la programmation logique où la recherche est prise en compte par le langage, par la manière de programmer. De ce fait, elle devient incontrôlable, nous ne pouvons pas savoir ce qui se passe car c'est lors de la compilation et de son exécution que prend naissance une possible solution. Pour reprendre en main le cheminement de la recherche de solution, nous devons l'encapsuler.

Le mot « recherche » prend rapidement toute sa force en se basant sur les chapitres 3 et 4. Des problèmes ne pouvant pas être solutionnés endéans un temps acceptable peuvent être approchés en utilisant des logiques empiriques, provenant de l'étude et de l'observation de la nature. Une modélisation créée en premier lieu pour simuler, reproduire, étudier un

événement naturel donne naissance à une théorie, un algorithme et pour finir à une manière de penser.

Pouvons-nous dire qu'à la vue de nos résultats, nous avons démontré ou du moins nous avons avancé sur la voie de la recherche heuristique encapsulée ?

Notre réponse ne peut être que mitigée, ni le oui ni le non ne sont dominants. Intellectuellement, un travail de fond a été effectué mais il a été en partie purement théorique de par la jeunesse du langage et de l'implémentation choisie.

Il est certain que, comme écrit dans la conclusion du chapitre 6, nous avons devant nous encore plus de possibilités ou de questions demandant des réponses qu'avant de commencer cette étude.

Un premier point d'attention est l'antagonisme entre la nécessité de paramétriser les heuristiques en fonction du problème et l'envie, l'objectif de généraliser ces mêmes heuristiques pour rendre leur utilisation indépendante.
Nous pouvons requalifier « indépendance » comme étant l'indépendance de l'expression déclarative du problème, donc sur la manière et pas le contenu.
Seulement, dans le chapitre 5, nous avons conclu qu'il existe toujours un lien entre le contenu et la manière. Ce lien, nous avons essayé de le rendre le plus ténu possible, en passant par une approche de précompilation.

Un deuxième point d'attention est la création d'opérateurs hybrides de la forme « interleave ». Nous généralisons un mode de résolution par des opérateurs heuristiques utilisant à leur tour des opérateurs heuristiques pour certaines de leurs étapes. En poussant cette vision à l'extrême nous pouvons imaginer une approche récursive de l'appel à un opérateur, en sachant que les sous-problèmes ne sont pas des parties du problème de base mais un nouveau problème local dont néanmoins la solution permet d'avancer vers la solution globale.
N'oublions pas aussi que les opérateurs hybrides peuvent poser des problèmes potentiels de convergence car l'heuristique de base peut être convergente si on laisse son algorithme

inchangé mais peut ne plus l'être si l'on change ses routines de choix natif par des heuristiques pouvant introduire des « sauts » dans l'espace de solutions et donc des divergences.

Un troisième point d'attention est la gestion et surtout le passage entre heuristiques de l'espace temporaire, cet espace qui contient l'ensemble des informations utilisées et nécessaires à l'heuristique en cours pour arriver à la solution actuelle. Cet ensemble a un format et une signification propres tel que liste tabou, population, chemins et phéromones, température, ... qui ne peuvent être compris que par une seule heuristique ou dans le meilleur des cas par un type équivalent d'heuristiques. Etudier le passage, les fonctions primitives permettant de traduire, de transposer ces informations d'un type d'heuristique vers un autre serait un plus. La question que l'on peut se poser est : est-ce possible mais surtout est ce que cela a un intérêt ?

Le dernier point d'attention est le choix du paradigme de programmation. Est-ce un choix pertinent ? Rend-il les concepts plus simples à l'implémentation, à l'utilisation, aux deux ?
Globalement, le fait de vouloir implémenter une manière de résoudre un problème et pas de décrire ce dernier n'est pas en parfaite harmonie avec ce paradigme. Si l'on se réfère au chapitre sur la recherche encapsulée, il y a quelques propositions propres à Curry mais aussi la proposition d'utiliser des routines externes. N'oublions pas que nous voulons garder le contrôle durant l'exécution de l'opérateur et que d'un autre côté ce langage se veut « pur ».

En conclusion, ce mémoire ne se veut pas une réponse mais un premier pas dans la voie connue que sont les heuristiques implémentées dans un langage fonctionnel et logique.
Il demande une suite qui permettrait de répondre à une partie de questions ouvertes.

BIBLIOGRAPHIE

[BaHA] Barichard V., Hao J-K « Un algorithme hybride pour le problème de sac à dos multi-objectifs » Actes JNPC'02

[Beff] Une introduction à Haskell, Slides 51-65,20 septembre 2003

[BKNHRS] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross ans Sonia Schulenburg. « Hyper-heuristics : an Emerging Direction in modern Search Technology »Handbook of metaheuristics eBook ISBN 0-306-48056-5, p457-p474

[CurryM] Hanus M. « User Manual » Packs 1.8.1 version of June 7, 2007

[CurryR] Hanus M. « Curry An Integrated Functional Logic Language » Version 0.8.2 March 28,2006

[CurryT] Antoy S., Hanus M.. « Curry A tutorial Introduction »Draft of December 5,2007

[DoSt] Marco Dorigo and Thomas Stützle. « The Ant Colony Optimization Metaheuristic : Algorithms, Applications and Advances »Handbook of metaheuristics eBook ISBN 0-306-48056-5, p251-p286

[Dreo] Dréo J. « Different classifications of metaheuristics » http://fr.wikipedia.org/wiki/Fichier:Metaheuristics_classification_fr.svg

[FoLaLo] Filippo Focacci, François Laburthe And Andrea Lodi. « Local Search and Constraint Programming »Handbook of metaheuristics eBook ISBN 0-306-48056-5, p369-p404

[FrWa] Eugene C.Freuder and Mark Wallace. « Constraint Satisfaction »Handbook of metaheuristics eBook ISBN 0-306-48056-5, p405-p428

[FVW] Andreas Fink, Stefan VoB and David L. Woodruff. « Metaheuristic Class Libraries »Handbook of metaheuristics eBook ISBN 0-306-48056-5, p515-p537

[Gend] Gendreau M. « An Introduction to Tabu Search »Handbook of metaheuristics eBook ISBN 0-306-48056-5, p37-p54

[HGH] Hao j., Galinier P., Habib M. « Métaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes » Revues d'intelligence Artificielle 1999 p1-p39

[HoTs] Hoos H.H., Tsang E. « Local Search Methods. » Handbook of Constraint Programming P135-158

[Hutta] Hutton G. « Polymorphic types » Programming In Haskell p33-p34

[Huttb] Hutton G. « Recursive Functions » Programming In Haskell p57-p69

[Huttc] Hutton G. « Functional Parsers » Programming In Haskell p75-p87

[Reev] Reeves C. «Genetic Algorithms »Handbook of metaheuristics eBook ISBN 0-306-48056-5, p55-p82

[TGGP] Taillard E., Gambardelle L., Gendreau M., Potvin JY « La programmation à mémoire adaptative ou l'évolution des algorithmes évolutifs »

[Tsan] Tsang E. « Fundamental concepts in the CSP » « Problem reduction » Foundations of constraint Satisfaction p53-p63, p70-p76

[VaDu] Vasquez M, Dupont A. « Filtrage par arc-consistance et recherche tabou pour l'allocation de fréquences avec polarisation » Actes JNPC'02

[Vanh] Vanhoof W. « Haskell » Programmation fonctionnelle p1-p25

ANNEXES

1. CURRY LIBRARY CLPFD

Library for finite domain constraint solving.[CurryM]

The general structure of a specification of an FD problem is as follows: domain constraint & fd constraint & labeling where:

domain constraint specifies the possible range of the FD variables (see constraint domain) fd constraint specifies the constraint to be satisfied by a valid solution (see constraints #+, #-,allDifferent, etc below) labeling is a labeling function to search for a concrete solution.

Note: This library is based on the corresponding library of Sicstus-Prolog but does not implement the complete functionality of the Sicstus-Prolog library. However, using the PAKCS interface for external functions, it is relatively easy to provide the complete functionality.

Exported types:

data LabelingOption

This datatype contains all options to control the instantiated of FD variables with the enumeration constraint labeling.

Exported constructors:

LeftMost :: LabelingOption

LeftMost - The leftmost variable is selected for instantiation (default)

FirstFail :: LabelingOption

FirstFail - The leftmost variable with the smallest domain is selected (also known as firstfail principle)

FirstFailConstrained :: LabelingOption

FirstFailConstrained - The leftmost variable with the smallest domain and the most constraints on it is selected.

Min :: LabelingOption

Min - The leftmost variable with the smalled lower bound is selected.

Max :: LabelingOption

Max - The leftmost variable with the greatest upper bound is selected.

Step :: LabelingOption

Step - Make a binary choice between x=#b and x/=#b for the selected variable x where b is the lower or upper bound of x (default).

Enum :: LabelingOption

Enum - Make a multiple choice for the selected variable for all the values in its domain.

Bisect :: LabelingOption

Bisect - Make a binary choice between x<=#m and x>#m for the selected variable x where m is the midpoint of the domain x (also known as domain splitting).

Up :: LabelingOption

Up - The domain is explored for instantiation in ascending order (default).

Down :: LabelingOption

Down - The domain is explored for instantiation in descending order.

All :: LabelingOption

All - Enumerate all solutions by backtracking (default).

Minimize :: Int ! LabelingOption

Minimize v - Find a solution that minimizes the domain variable v (using a branch-andbound algorithm).

Maximize :: Int ! LabelingOption

Maximize v - Find a solution that maximizes the domain variable v (using a branch-andbound algorithm).

Assumptions :: Int ! LabelingOption

Assumptions x - The variable x is unified with the number of choices made by the selected enumeration strategy when a solution is found.

Exported functions:

domain :: [Int] ! Int ! Int ! Success

Constraint to specify the domain of all finite domain variables.

(+#) :: Int ! Int ! Int

Addition of FD variables.

(-#) :: Int ! Int ! Int

Subtraction of FD variables.

(*#) :: Int ! Int ! Int

Multiplication of FD variables.

(=#) :: Int ! Int ! Success

Equality of FD variables.

(/=#) :: Int ! Int ! Success

Disequality of FD variables.

(<#) :: Int ! Int ! Success

"Less than" constraint on FD variables.

(<=#) :: Int ! Int ! Success

"Less than or equal" constraint on FD variables.

(>#) :: Int ! Int ! Success

"Greater than" constraint on FD variables.

(>=#) :: Int ! Int ! Success

"Greater than or equal" constraint on FD variables.

sum :: [Int] ! (Int ! Int ! Success) ! Int ! Success

Relates the sum of FD variables with some integer of FD variable.

scalarProduct :: [Int] ! [Int] ! (Int ! Int ! Success) ! Int ! Success

(scalarProduct cs vs relop v) is satisfied if ((cs*vs) relop v) is satisfied. The first argument must be a list of integers. The other arguments are as in sum.

count :: Int ! [Int] ! (Int ! Int ! Success) ! Int ! Success

(count v vs relop c) is satisfied if (n relop c), where n is the number of elements in the list of FD variables vs that are equal to v, is satisfied. The first argument must be an integer. The other arguments are as in sum.

allDifferent :: [Int] ! Success

"All different" constraint on FD variables.

all different :: [Int] ! Success

For backward compatibility. Use allDifferent.

indomain :: Int ! Success

Instantiate a single FD variable to its values in the specified domain.

labeling :: [LabelingOption] ! [Int] ! Success

Instantiate FD variables to their values in the specified domain.

2. CURRY LIBRARY CLPR

Library for constraint programming with arithmetic constraints over reals.[CurryM]

Exported functions:

(+.) :: Float ! Float ! Float

Addition on floats in arithmetic constraints.

(-.) :: Float ! Float ! Float

Subtraction on floats in arithmetic constraints.

(*.) :: Float ! Float ! Float

Multiplication on floats in arithmetic constraints.

(/.) :: Float ! Float ! Float

Division on floats in arithmetic constraints.

(<.) :: Float ! Float ! Success

"Less than" constraint on floats.

(>.) :: Float ! Float ! Success

"Greater than" constraint on floats.

(<=.) :: Float ! Float ! Success

"Less than or equal" constraint on floats.

(>=.) :: Float ! Float ! Success

"Greater than or equal" constraint on floats.

i2f :: Int ! Float

Conversion function from integers to floats. Rigid in the first argument, i.e., suspends until the first argument is ground.

minimumFor :: (a ! Success) ! (a ! Float) ! a

Computes the minimum with respect to a given constraint. (minimumFor g f) evaluates to x if (g x) is satisfied and (f x) is minimal. The evaluation fails if such a minimal value does not exist. The evaluation suspends if it contains unbound non-local variables.

minimize :: (a ! Success) ! (a ! Float) ! a ! Success

Minimization constraint. (minimize g f x) is satisfied if (g x) is satisfied and (f x) is minimal. The evaluation suspends if it contains unbound non-local variables.

maximumFor :: (a ! Success) ! (a ! Float) ! a

Computes the maximum with respect to a given constraint. (maximumFor g f) evaluates to x if (g x) is satisfied and (f x) is maximal. The evaluation fails if such a maximal value does not exist. The evaluation suspends if it contains unbound non-local variables.

maximize :: (a ! Success) ! (a ! Float) ! a ! Success

Maximization constraint. (maximize g f x) is satisfied if (g x) is satisfied and (f x) is maximal. The evaluation suspends if it contains unbound non-local variables.

3. RECURSION

Defining recursive functions is like riding a bicycle: it looks easy when someone else is doing it, may seem impossible when you first try to do it yourself, but becomes simple and natural with practice. In this section we offer some advice for defining functions in general, and recursive functions in particular, using a five step process that we introduce by means of examples.[Huttb]

Example - product

As a simple first example, we show how the definition given earlier in this chapter for the library function that calculates the product of a list of numbers can be constructed in a stepwise manner.

Step 1: define the type

Thinking about types is very helpful when defining functions, so it is good practice to define the type of a function prior to starting to define the function itself. In this case, we begin with the type product :: [Int] → Int that states that product takes a list of integers and produces a single integer.

As in this example, it is often useful to begin with a simple type, which can be refined or generalised later on as appropriate.

Step 2: enumerate the cases

For most types of argument, there are a number of standard cases to consider. For lists, the standard cases are the empty list and non-empty lists, so we can write down to the following skeleton definition using pattern matching:

product [ ] =

product (n : ns) =

For non-negative integers, the standard cases are 0 and n+1, for logical values they are False and True, and so on. As with the type, we may need to refine the cases later on, but it is useful to begin with the standard cases.

Step 3: define the simple cases

By definition, the product of zero integers is one, because one is the identity for multiplication. Hence it is straightforward to define the empty list case:

product [ ] = 1

product (n : ns) =

As in this example, the simple cases often become base cases.

Step 4: define the other cases

How can we calculate the product of a non-empty list of integers? For this step, it is useful to first consider the ingredients that can be used, such as the function itself (product), the arguments (n and ns), and library functions of relevant types (+, −, ∗, and so on.) In this case, we simply multiply the first integer and the product of the remaining list of integers:

product [ ] = 1

product (n : ns) = n ∗ product ns

As in this example, the other cases often become recursive cases.

Step 5: generalise and simplify

Once a function has been defined using the above process, it often becomes clear that it can be generalised and simplified. For example, the function product does not depend on the precise kind of numbers to which it is applied, so its type can be generalised from integers to any numeric type:

product :: Num a ⇒ [a] → a

4. TAXINOMIE DE L'HYBRIDATION

On parle d'hybridation quand la métaheuristique considérée est composée de plusieurs méthodes se répartissant les tâches de recherche. La taxinomie des métaheuristiques hybrides se sépare en deux parties : une classification hiérarchique et une classification plate. La classification est applicable aux méthodes déterministes aussi bien qu'aux métaheuristiques.

La classification hierarchique (figure 18) se fonde sur le niveau (bas ou haut) de l'hybridation et sur son application (en relais ou concurrente). Dans une hybridation de bas niveau, une fonction donnée d'une métaheuristique (par exemple, la mutation dans un algorithme évolutionnaire) est remplacée par une autre métaheuristique (par exemple une recherche avec tabou). Dans le cas du haut niveau, le fonctionnement interne « normal » des métaheuristiques n'est pas modifié. Dans une hybridation en relais, les métaheuristiques sont lancées les unes après les autres, chacune prenant en entrée la sortie produite par la précédente. Dans la concurrence (ou co-évolution), chaque algorithme utilise une série d'agents coopérants ensembles.

Cette première classification dégage quatre classes générales :

- bas niveau & relais (abrégé LRH en anglais),
- bas niveau & co-évolution (abrégé LCH),
- haut niveau & relais (HRH),
- haut niveau & co-évolution (HCH).

La seconde partie dégage plusieurs critères, pouvant caractériser les hybridations :

- si l'hybridation se fait entre plusieurs instances d'une même métaheuristique, elle est homogène, sinon, elle est hétérogène ;
- si les méthodes recherchent dans tout l'espace de recherche, on parlera d'hybridation globale, si elles se limitent à des sous-parties de l'espace, d'hybridation partielle ;
- si les algorithmes mis en jeu travaillent tous à résoudre le même problème, on parlera d'approche générale, s'ils sont lancés sur des problèmes différents, d'hybridation spécialisée.

Ces différentes catégories peuvent être combinées, la classification hierarchique étant la plus générale.

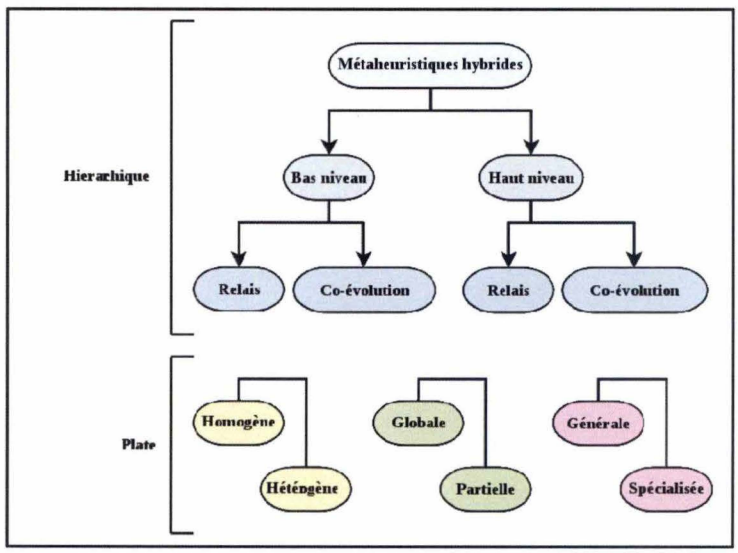


Figure 18 : Taxinomie de l'hybridation