Résolution de problèmes combinatoires (partie I)

Pierre-Édouard Portier

14 mars 2011

1 La résolution de problèmes

1.1 Plan de résolution

Un aspect essentiel et difficile de la résolution de problèmes tient à la construction d'un plan de résolution. C'est peut-être l'étape d'une résolution de problème qui demande le plus d'invention, de créativité. Prenons un exemple.

1.2 L'âge des trois filles de Watson

1.2.1 Énoncé

Watson rencontre son ami Holmes : «

— Aujourd'hui, Sherlock, mes trois filles célèbrent leurs anniversaires ! Mais pouvez-vous en déduire leurs âges ?

— Vous devez m'en dire un peu plus Watson...

— Alors, voici un indice : le produit des âges de mes filles est égal à 36.

— Ce n'est pas suffisant...

— La somme de leurs âges est égal au nombre de pipes dans votre collection.

— Voyons... Non Watson, j'ai encore besoin d'information.

— La plus âgée de mes filles ne supporte pas l'odeur d'opium qui règne dans votre appartement.

— Maintenant, Watson, je connais la solution ! »

1.2.2 Solution

1.3 Ce qui rend un problème difficile

- La taille de l'espace d'états où se déploie la recherche est *hénaurme*
- La modélisation du problème le transforme à un point tel qu'une solution est en pratique inutile
- Il est difficile de déterminer une *fonction d'évaluation* qui affecte un score à une solution potentielle. Par exemple : la fonction d'évaluation varie avec le temps.
- Les solutions sont contraintes à un point tel que la découverte d'une solution *possible* est déjà difficile. Ainsi, la découverte d'une solution optimale est d'autant plus difficile.
- Etc.

1.4 Deux problèmes modèles

1.4.1 SAT

Il s'agit du problème de satisfaction de contraintes booléennes : trouver une solution pour une équation booléenne dans sa forme normale conjonctive. Soit un exemple avec 100 variables :

$(x_{87} \vee x_2 \vee \neg x_{54}) \wedge (x_8 \vee x_{13}) \wedge \ldots$

Il est immédiat que la taille de l'espace d'état est $2^{100} \simeq 1.27 \times 10^{30}$. Pour avoir un ordre de grandeur, si nous pouvons tester 1000 solutions par secondes, dans 13,7 milliards d'années, nous n'aurons exploré que 1% des solutions possibles ! (Pourquoi 13,7 milliards d'années ?)

Par ailleurs, comment définir une fonction d'évaluation utile ? Est-il possible d'opérer une distinction qualitative entre deux états pour lesquels la fonction booléenne est évaluée *fausse* ?

1.4.2 TSP

Le problème du voyageur de commerce (*Traveling Salesman Problem*) qui doit visiter toutes les villes où résident ses clients et retourner à son point de départ en minimisant la distance parcourue.

Supposons qu'il y ait n villes et que le problème soit symétrique (c'est-à-dire que le coût pour aller de la ville A à la ville B est égal à celui pour aller de la ville B à la ville A). La taille de l'espace d'états est alors $(n-1)!/2$. Ce qui est rapidement beaucoup plus important que les tailles d'espaces d'états pour le problème SAT... En pratique *inconcevable* !

Par contre, une fonction d'évaluation naïve et potentiellement utile est immédiate : la somme des distances entre chaque paires de villes dans l'ordre d'une solution (par exemple, $F(V3, V1, V2) = d(V3, V1) + d(V1, V2)$).

1.5 Modélisation

1.5.1 Construction d'un espace d'états

Dans ce qui a été montré plus haut, il faut remarquer que la taille d'un espace d'état n'est pas liée directement au problème mais à son modèle. Ainsi, la modélisation, ou en ce qui nous concerne ici : le dessin d'un espace d'états, est une étape essentielle d'une résolution de problème.

Réfléchir par exemple à comment construire 4 triangles équilatéraux avec six allumettes...

1.5.2 Expression de l'objectif

Il faut être capable d'exprimer clairement l'objectif d'une recherche. Pour le cas de TSP, ce serait par exemple : $min \sum dist(x, y)$.

1.5.3 Construction d'une fonction d'évaluation

Souvent un problème est accompagné de contraintes qui peuvent guider la construction de la fonction d'évaluation. Autrement dit, l'espace d'état comprend souvent des solutions acceptables et d'autres inacceptables. Des problèmes d'optimisation (type TSP) peuvent se voir ajouter des contraintes. D'autres problèmes consistent uniquement à satisfaire un

ensemble de contraintes ; comme par exemple placer 8 reines sur un échiquier de telle sorte qu'aucune ne soit en prise.

1.6 Définition d'un problème de recherche dans un espace d'états

Étant donné un espace d'états $\mathcal{S}$ et le sous-ensemble des solutions acceptables $\mathcal{F} \subseteq \mathcal{S}$, trouver $x \in \mathcal{F}$ tel que $eval(x) \leq eval(y)$ pour tout $y \in \mathcal{F}$.

Remarquons que la formulation proposée convient à un problème où il s'agit de minimiser la fonction d'évaluation. Mais la formulation reste générique, puisqu'aucun sens n'est ici associé à la fonction d'évaluation.

Un x répondant au problème est dit être une solution globale. Trouver une telle solution peut souvent se révéler difficile... En fait, pour chacun des problèmes auxquels nous nous intéressons, il n'existe pas d'algorithme polynomial connu (cf. le cours de mathématiques discrètes de Céline Robardet pour une introduction à la notion de complexité d'un algorithme).

2 Les approches exactes

2.1 Méthodes par séparation et évaluation (Branch & Bound)

Les méthodes par séparation et évaluation permettent de trouver une solution exacte pour les problèmes qui possèdent des contraintes définissant un ensemble de solutions réalisables.

2.1.1 Séparation

La phase de séparation consiste à partitionner l'espace d'état en une arborescence. La racine contient toutes les solutions. L'ensemble des fils d'un nœud de l'arborescence forme une partition des états de ce nœud (l'union des sous-ensembles associés aux fils d'un sommet doit toujours être égal à l'ensemble associé à ce sommet).

2.1.2 Évaluation d'une borne supérieure et utilisation d'une borne inférieure

On possède une fonction d'évaluation qui à tout sommet de la partition arborescente de l'espace d'états permet d'associer un majorant (une borne supérieure) des valeurs objectifs des états de ce sommet. De plus, on est capable de calculer une première solution.

Ainsi, il ne sera pas nécessaire de séparer un sommet S si :

- L'évaluation de S donne une borne supérieure inférieure ou égale à la meilleure solution courante.
- S est vide (les contraintes du problème ne sont pas respectées)
- La meilleure des solutions contenues dans S est connue (S est une feuille, ou l'évaluation est exacte).

2.1.3 Type de parcours

L'arborescence qui partitionne l'espace d'états peut être construite en profondeur ou en largeur ou en privilégiant les sommets dont l'évaluation est maximale (stratégie "meilleur d'abord" ou "best first search"). On choisira souvent la stratégie "en profondeur d'abord" pour son utilisation efficace de l'espace mémoire, en effet les arbres de recherche pour les problèmes combinatoires sont souvent beaucoup plus larges que profonds.

2.1.4 Énoncé du problème du sac-à-dos

Il s'agit de remplir un sac-à-dos avec un sous-ensemble de N objets x_i avec $i \in [1, N]$. À chaque objet x_i sont associés un poids w_i et une utilité u_i. De plus, le sac-à-dos ne peut pas supporter un poids supérieur à W. Il s'agit de remplir le sac-à-dos en maximisant l'utilité tout en respectant la contrainte de poids.

Soient les données suivantes : $u_1 = 10$; $w_1 = 6$; $u_2 = 8$; $w_2 = 5$; $u_3 = 5$; $w_3 = 4$ et $W = 9$.

2.1.5 Résolution du problème du sac-à-dos

2.2 Programmation dynamique

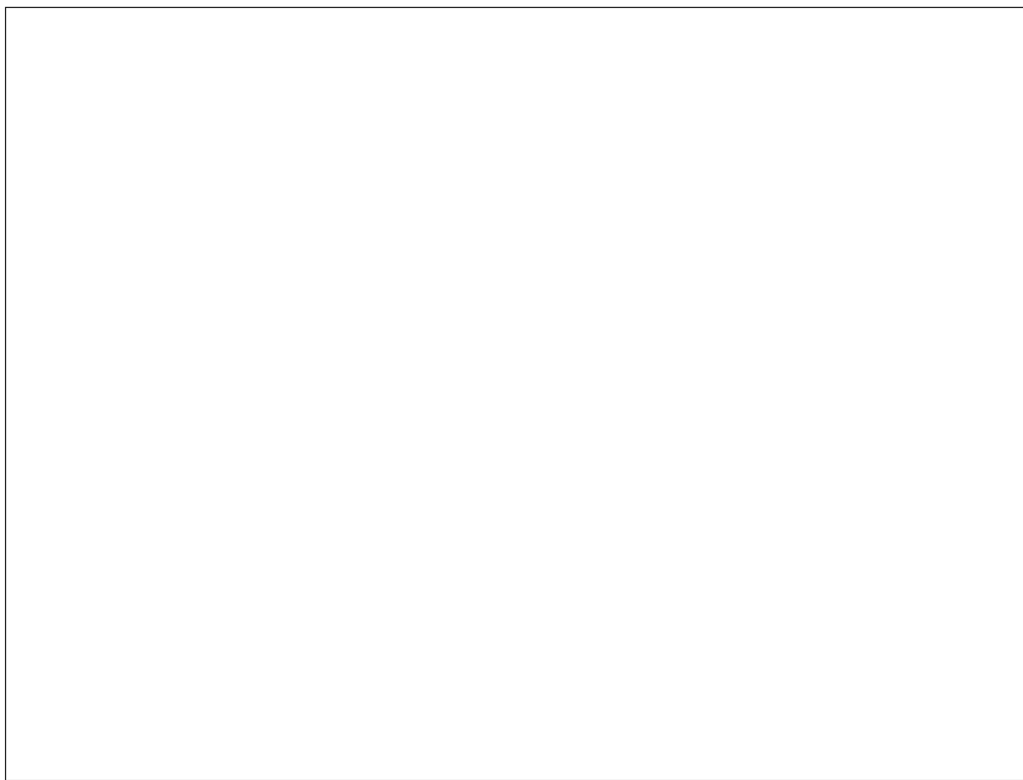
2.2.1 Approche bottom-up

La programmation dynamique est une approche *bottom-up* où il s'agit de débuter avec les sous-problèmes les plus simples puis de combiner leurs solutions pour répondre à des sous-problèmes plus gros, jusqu'à répondre au problème original.

2.2.2 Le principe d'optimalité

- Dans une séquence optimale de décisions, toute sous-séquence doit être optimale.
- Pour certains problèmes, une solution optimale peut-être trouvée en prenant les décisions l'une après l'autre et sans jamais revenir sur ses choix.
- Pour la majorité des problèmes il est nécessaire d'énumérer toutes les séquences de décisions possibles et de choisir la meilleure.
- La programmation dynamique réduit les calculs nécessaires en usant du *principe d'optimalité*.

2.2.3 Exemple sur le problème du sac-à-dos



3 Heuristiques classiques

3.1 Introduction

Pour de nombreux problèmes (TSP, SAT, etc.), les méthodes qui viennent d'être présentées et qui permettent de trouver une solution exacte sont en pratique inutilisables sauf pour des instances de petites tailles. Il faut donc souvent se contenter de solutions approchées. Les méthodes utilisées pour obtenir ces approximations sont souvent appelées *heuristiques*. De plus, lorsqu'elle sont suffisamment génériques pour pouvoir s'appliquer à de nombreux types de problèmes, elles sont alors appelées *méta-heuristiques*.

Cette section présente des heuristiques simples et très classiques. La prochaine partie portera l'accent sur des heuristiques un peu plus subtiles.

3.2 La notion de voisinage

La région de l'espace d'états *proche* d'un point de cet espace est appelée le voisinage de ce point. Toute la question tient au choix d'une notion de distance qui permette de définir précisément cette notion de proximité. Cette distance peut prendre le plus souvent deux formes :

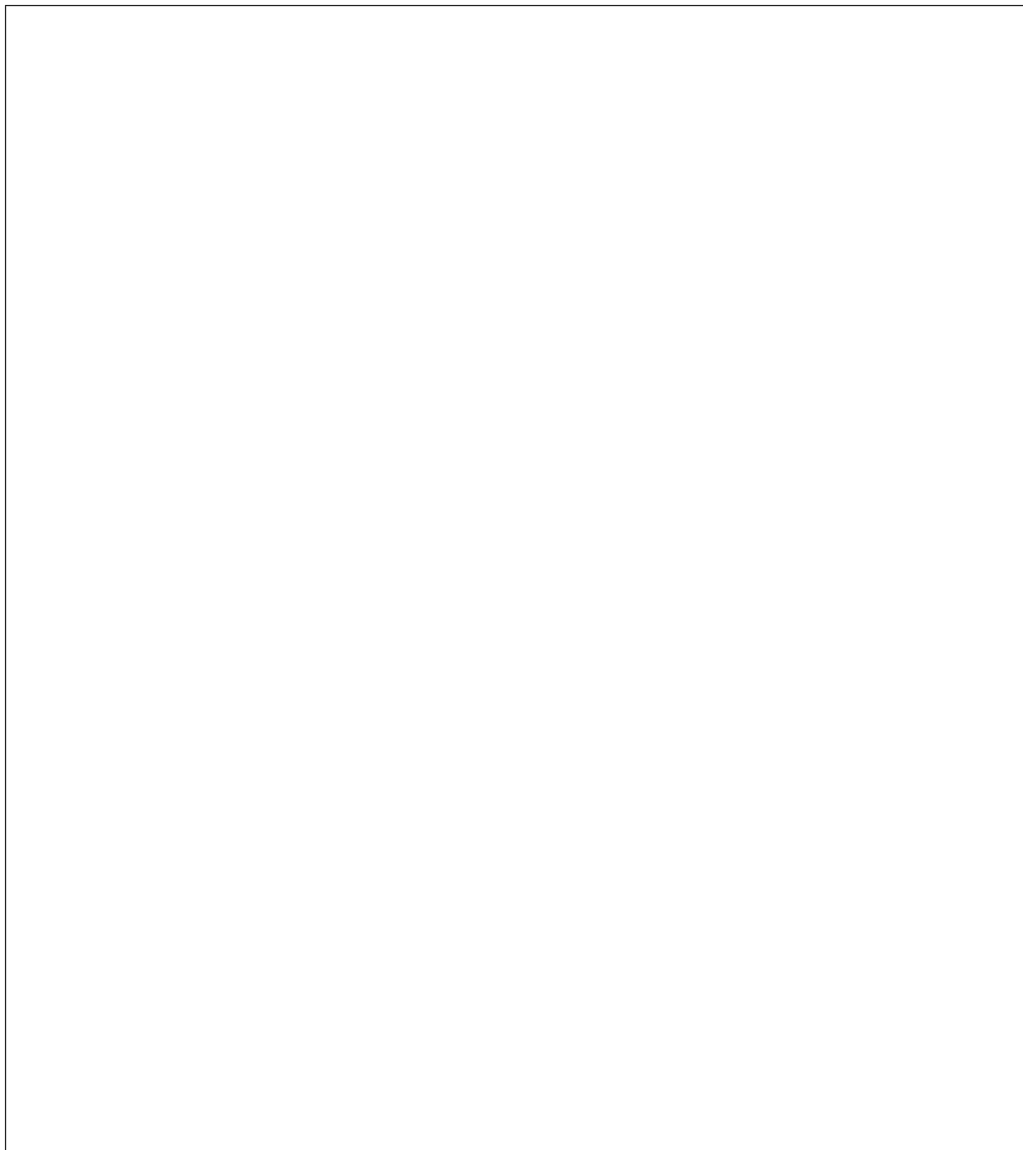
- $dist : S \times S \longrightarrow \mathbb{R}$ avec le voisinage $N(x)$ défini comme : $N(x) = \{y \in S : dist(x, y) \leq \epsilon\}$. C'est par exemple le cas de la distance de Hamming pour le problème SAT : le nombre de bits qui diffèrent entre deux affectations des variables de l'équation booléenne.
- $m : S \longrightarrow 2^S$ est une fonction de mapping qui définit directement un voisinage pour tout point de S. Dans le cas du problème TSP, on peut définir la fonction *2-swap* qui génère toutes les solutions obtenues en intervertissant exactement deux villes de la solution originale. Ainsi, lorsque le nombre de villes est égal à n, la fonction 2-swap associe à chaque solution un voisinage qui a pour taille $n(n-1)/2$.

Une fois une notion de distance définie, un état $x \in \mathcal{F}$ est un *optimum local* par rapport au voisinage N si et seulement si $eval(x) \leq eval(y)$ pour tout $y \in N(x)$.

Pour des voisinages "pas trop grands", il devient assez facile de déterminer un optimum local. Le problème tient maintenant au fait que les fonctions d'évaluation vont souvent associer à l'espace d'états une topographie vallonnée... D'où les méthodes d'amélioration itérative (ou *hill climbing*).

3.3 Méthodes d'amélioration itérative

3.3.1 Un algorithme de hill-climbing



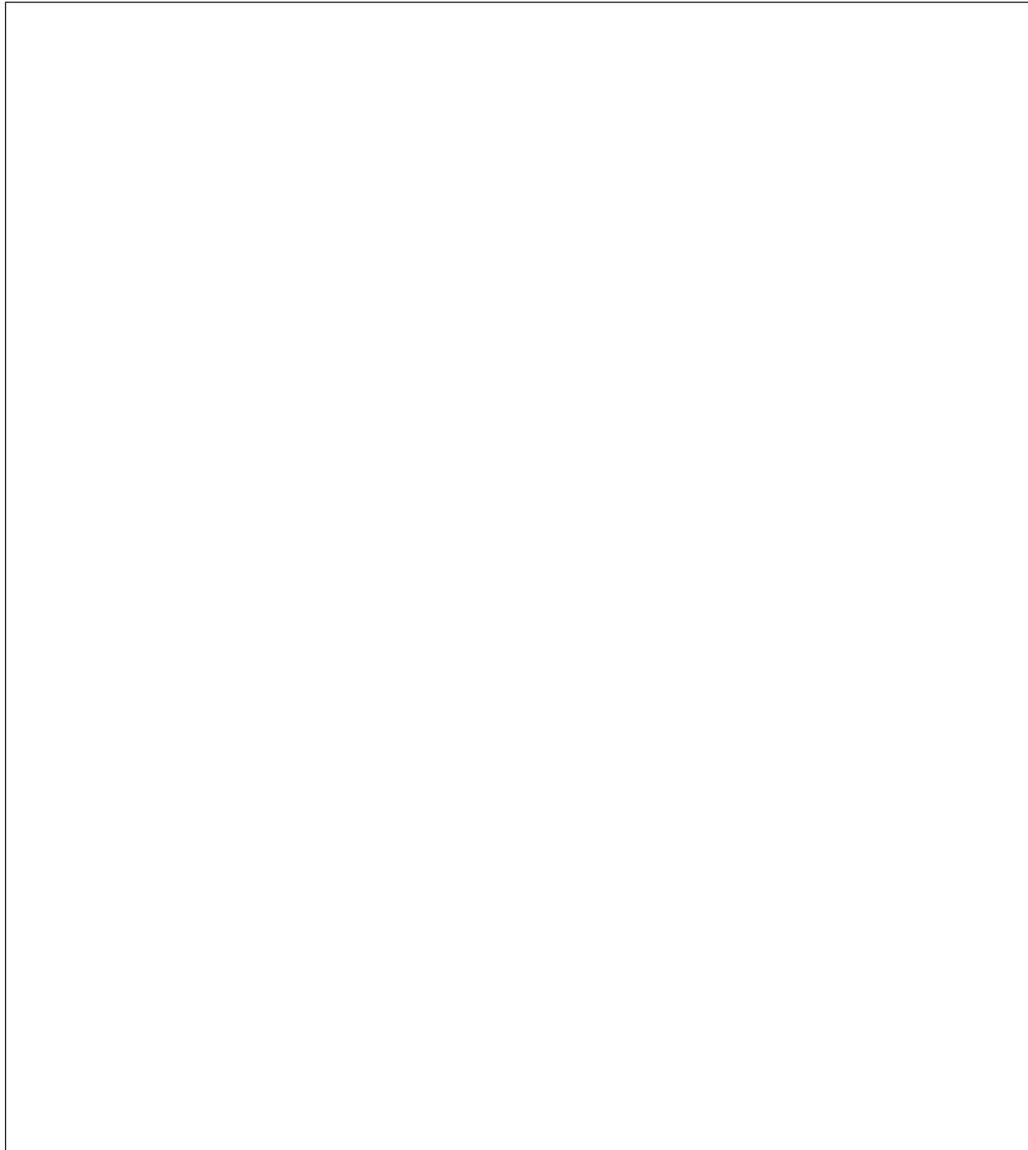
3.3.2 Algorithmes gloutons

Pour initialiser notre algorithme de hill-climbing, il faut être capable de trouver rapidement une première solution. Dans ce cas, les algorithmes dits *gloutons* sont efficaces. Nous en avons déjà rencontré un lors de l'étude de la technique *branch & bound* sur le problème du sac-à-dos. Un algorithme glouton est un algorithme pour lequel, à chaque itération, on fixe la valeur d'une des variables décrivant le problème, jusqu'à obtenir une solution complète, et sans jamais revenir sur un choix. Il faut donc être capable

d'évaluer une "meilleure décision" à chaque étape.

Deux exemples d'algorithmes gloutons pour le problème TSP :

- La ville la plus proche
- La plus petite distance entre deux villes
- Etc.



3.4 L'algorithme A*

Nous avons vu que les algorithmes gloutons qui cherchent toujours le prochain meilleur mouvement ne donnent pas toujours de bons résultats. Nous supposons ici que l'espace d'états peut être organisé sous la forme d'un arbre (nous reviendrons sur ce point en TD). Au lieu d'explorer l'arbre au moyen d'une recherche en profondeur, nous allons choisir le prochain nœud

à explorer grâce à une heuristique qui offre une évaluation du résultat final si ce nœud est choisi. On appelle cette méthode "Best First Search" (BFS).

3.4.1 Squelette de l'algorithme BFS

3.4.2 La notion de nœud disponible

Le concept clef est celui de *nœud disponible*. L'algorithme maintient deux listes de nœuds : la liste ouverte et la liste fermée. La première contient les nœuds libres, la seconde ceux qui ont déjà été traités. Une fois un nœud placé dans la liste fermée, tous ses fils sont placés dans la liste ouverte puis ils sont évalués au moyen d'une heuristique et le meilleur est sélectionné pour les prochains traitements.

Étant donnée cette description, pourquoi avoir demandé que l'espace d'état soit représenté sous la forme d'un arbre ?

La fonction d'évaluation peut s'écrire sous la forme : $eval(q) = c(q) + h(q)$, avec q une solution partielle, $c(q)$ une évaluation des décisions prises jusqu'à présent, $h(q)$ une évaluation de la qualité des décisions à venir.

Pour le problème TSP, que pourrait être $c(q)$?

Notons $h*$ l'heuristique parfaite qui retourne le coût de la meilleure solution possible étant donnée la solution partielle q. Pour un problème où il s'agit de minimiser les coûts, on dit qu'un algorithme BFS est admissible si : $h(q) \leq h*(q)$ pour tous les nœuds q. Un algorithme BFS admissible

est appelé A*. Ainsi, puisque h sous-estime le coût d'une solution, il est impossible de rater la meilleurs solution !

Notons que si l'on possède deux heuristiques h_1 et h_2 telles que pour tout q : $h_1(q) \leq h*(q)$ et $h_2(q) \leq h*(q)$ et $h_1(q) \leq h_2(q)$, on dit que h_2 est meilleure que h_1.

3.4.3 Un exemple pour récapituler