Recherche heuristique et méta-heuristique

Matthieu Amiguet

2008 – 2009

haute école arc ingénierie
neuchâtel berne jura

Introduction 3

- Un grand nombre de problèmes d'IA sont caractérisés par l'absence d'algorithmes permettant de construire directement une solution à partir de la donnée du problème
- On doit donc parcourir un ensemble de possibilités pour trouver
  - la meilleure...
  - ...ou une pas trop mauvaise !
- Souvent, l'ensemble des solutions à parcourir est très grand et ne peut pas être considéré en entier
  - Sinon ce ne serait sans doute pas de l'IA...
- On doit donc utiliser des *méthodes heuristiques*.

Heuristique ? 4

- Heuristique : du grec *heuristo*, "je trouve"
  - Penser aux fameux "Heureka" d'archimède !
- En informatique, une heuristique est une technique visant à accélérer la recherche d'une solution à un problème
  - Son but est d'aider à chercher dans la bonne direction
  - Fait souvent appel à des connaissances "expertes"
- Les techniques heuristiques permettent généralement de faire un compromis entre la rapidité de la recherche et la qualité de la solution trouvée
  - Trouver une solution optimale en un temps "pas trop long"
  - Trouver rapidement une solution "pas trop mauvaise"

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Recherche dans un espace d'états 6

- Caractérisation générale d'un problème de recherche dans un espace d'états :
  - Un ensemble d'*états* X, partagé en états *légaux* et en états *illégaux* ($X = L \cup I$, $L \cap I = \emptyset$)
  - Un *état initial* $i \in X$
  - Des *états finaux* $F \subset X$ (éventuellement un seul $F = \{f\}$)
  - Les états sont reliés par des *transitions* $T \subset X \times X$
  - Un ensemble O d'*opérateurs*. Chaque état possède un sous-ensemble d'opérateurs *applicables* $Op(x) \subset O$.
  - À chaque opérateur applicable correspond une *transition* vers un autre état.
- Le problème peut alors se poser de deux manières
  - Trouver un état final $f \in F$ atteignable depuis i
  - Trouver une suite d'opérateurs (un *chemin*) permettant de passer de i à un $f \in F$.

Remarques 7

- Les états finaux peuvent être donnés en *extension* (énumération de tous les états possibles) ou en *intention* (description des caractéristiques d'un état final)
- On ne s'intéresse pour l'instant qu'à des cas *déterministes* : dans un état donné, l'application d'un opérateur donné aura toujours le même résultat
- Les éléments ci-dessus définissent au fait un graphe appelé *graphe d'états*
  - Ce graphe n'est généralement pas représenté explicitement dans la mémoire (problème d'espace)...
  - ... mais construit au fur et à mesure des besoins !

Exemple 1 8
Le loup, la chèvre et le choux

État initial	Le loup, la chèvre, le choux et le bateau sur la rive gauche
État final	Le loup, la chèvre, le choux et le bateau sur la rive droite
Opérateurs	Transporter en bateau le loup, la chèvre ou le choux d'une rive à l'autre. Applicable si le bateau est sur la bonne rive
État légal	Ne pas avoir le loup et la chèvre ou la chèvre et le choux sur la même rive sans le bateau.

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Exemple 2
Conception d'horaires

État initial Un horaire vide
États finaux Tous les cours sont placés sans conflit
Opérateur Placer ou déplacer un cours.

- Dans ce cas, il est illusoire de parcourir l'ensemble des possibilités !

Ingrédients

- Nous aurons besoin des fonctions suivantes :
  - `legal(state)` : vrai si un état est légal, faux sinon
  - `final(state)` : vrai si un état est final, faux sinon
  - `op-applicables(state)` : renvoie la liste des opérateurs applicables dans l'état donné
  - `applique(op, state)` : renvoie l'état obtenu par après application de l'opérateur dans l'état donné
- Il faut aussi maintenir deux listes :
  - `frontiere` : contient les prochains états à explorer
    - Ceci suffirait dans le cas d'un arbre...
  - `histoire` : contient les états déjà explorés
    - Pour éviter de visiter un état déjà traité.

State


```
class State:
    def legal(self):
        ...
    def final(self):
        ...
    def applicableOperators(self):
        ...
    def apply(self, op):
        newState = ...
        # Si on veut se souvenir du chemin:
        newState.parent = self
        newState.op = op
        return newState
```


Search


```
def search(init):
    frontiere = [init]
    history = []
    while frontiere:
        etat = frontiere.pop()
        history.append(etat)
        if etat.final():
            return etat
        ops = etat.applicableOps()
        for op in ops:
            new = etat.apply(op)
            if (new not in frontiere) \
            and (new not in history) \
            and new.legal():
                insert(frontiere,new) # <-- !!!
    return "Pas de solution"
```

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Cas particuliers

- Suivant le type de stockage dans `frontière`, on obtient différents types de recherche

  Pile parcours en profondeur
  File parcours en largeur

- On peut aussi insérer les états dans `frontière` dans un ordre déterminé par une valeur : parcours par préférence
  - Si la valeur est le nombre d'opérateurs appliqués depuis l'état initial et qu'on considère les états dans l'ordre croissant de cette valeur, on retrouve le parcours en largeur
  - Le choix de cette valeur peut beaucoup influencer l'efficacité (moyenne) de la recherche.

L'idée : "suivre la vallée"

- Supposons que toute transition a un coût et qu'on cherche le chemin de coût minimal depuis un point de départ jusqu'à un état final
- Si on connaît tout le graphe, on peut considérer la fonction $f^*(x) = g^*(x) + h^*(x)$, où pour un état x
  - $g^*(x)$ est le coût minimal de l'état initial à x
  - $h^*(x)$ est le coût minimal de x à l'état final le plus proche

Propriété de f^*

Sur tout les noeuds d'un chemin optimal $x_0, x_1, \ldots, x_n$, $f^*(x_i)$ est constante et minimale

Quand on ne sait pas tout...

- On ne connaît généralement pas $g^*(x)$, mais seulement $g(x)$, le coût minimal *trouvé jusqu'à maintenant*. Pour tout noeud x :
  - $g(x) \geq g^*(x)$
  - $g(x)$ ne peux que diminuer en cours d'algorithme et converge vers $g^*(x)$
- On ne connaît pas $h^*(x)$, mais on peut essayer de l'estimer par $h(x)$
  - Une telle estimation, destinée à guider le parcours, est appelée une *heuristique*
- L'algorithme qui utilise la fonction $f(x) = g(x) + h(x)$ comme critère pour ordonner les états dans `frontière` s'appelle *Algorithme A*.

A^*

- Les heuristiques "optimistes" possèdent des caractéristiques intéressantes :
- Si $h(x) \leq h^*(x)$ pour tout noeud x, on a le cas particulier de *l'algorithme A^**
  - Une telle heuristique est qualifiée d'*admissible*

Théorème

A^* trouve toujours un chemin optimal s'il existe.

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Le problème des cycles 17

- Si on explore un *graphe* et non un *arbre*, on peut soudain trouver un meilleur chemin pour un noeud déjà rencontré
- Si le noeud déjà rencontré est encore dans `frontiere`, il suffit de mettre à jour son coût g et de le reclasser en fonction de la nouvelle somme $f+g$
  - On ne peut rien faire là-contre, mais ce n'est pas si grave... (ie pas si cher en calcul)
- Si le noeud est dans `histoire`, il ne suffit pas de mettre à jour son coût : il faudrait adapter le coût de tous ses descendants et les "remonter" dans `frontiere`...
  - Ceci peut être considérablement plus embêtant...

Révision des coûts dans `histoire` 18

Il existe deux manières de gérer ce problème

1. Renoncer à tenir à jour une `histoire`.
   - Marche à condition qu'un chemin existe ! (sinon, ne finit jamais !)
2. S'arranger pour ne traiter un noeud (et donc le mettre dans `histoire`) que lorsqu'on est sûr d'avoir trouvé le chemin le plus court jusqu'à ce noeud.
   - C'est le cas avec les heursitiques dites *consistantes*.

Heuristique consistante 19

- Une heuristique est dite *consistante* (ou *monotone*) si

  pour tout noeud n et
  pour tout successeur n' de n généré par une action a avec un coût $c(n,a,n')$, on a
  $h(n) \leq c(n,a,n') + h(n')$
- Ça ressemble à l'*inégalité triangulaire*
- En français : l'extension du chemin à un noeud voisin ne fait jamais diminuer l'estimation $f = g + h$ de la longueur totale du chemin.
- Une heuristique consistante est forcément admissible
  - Le contraire n'est pas forcément vrai, même s'il faut chercher un peu pour trouver un exemple !

A^* – propriétés 20

- S'il existe une solution, A^* la touvera toujours
- A^* est un algorithme très efficace pour trouver un chemin à coût minimal
  - L'efficacité dépend évidemment de l'heuristique...
  - ... mais pour une heuristique donnée, aucun algorithme ne parcourera moins de noeuds que A^* avant de trouver la solution
- Si C^* est le coût du chemin optimal, A^* développe tous les noeuds n avec $f(n) < C^*$
  - donc plus l'heuristique h a des valeurs *élevées*, plus elle sera utile pour trouver rapidement une solution...
  - ... mais h doit toujours rester *inférieure* à h^* !

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

A^* – usages et limites

- Usage à envisager chaque fois qu'on cherche un *chemin optimal* – géométrique ou conceptuel
  - Jeux (déplacement des personnages. . .)
  - Casse-tête (Taquin, Rubic's cube. . .)
  - Planification (transports, . . .)
  - . . .
- A^* donne un *chemin optimal*. Parfois, on peut se contenter d'un *chemin quelconque* ou on cherche seulement une *solution* (un état final). Dans ce cas, on peut par exemple utiliser A (heuristique non admissible).
- Comme A^* garde tous les noeuds en mémoire, il utilise une grande quantité de mémoire
  - Il existe diverses variantes qui font au fait des compromis temps-mémoire (RBFS, MA*, SMA*, . . .)

A^* – pour en savoir plus

- S. Russel, P. Norvig, "Intelligence Artificielle", Pearson France, 2006.
- `http://inst.eecs.berkeley.edu/~cs188/sp07/slides/SP07%20cs188%20lecture%204%20--%20a-star%20search.pdf`
- `http://www.policyalmanac.org/games/aStarTutorial.htm`
- `http://en.wikipedia.org/wiki/A-star_search_algorithm`
- `http://el-tramo.be/software/jsearchdemo/`

Méta-heuristiques ?

- Une des difficultés de la recherche heuristique est de trouver une heuristique appropriée
- L'heuristique dépend beaucoup du domaine d'application et son développement demande souvent des compétences avancées
- On peut donc s'intéresser à des méthodes heuristiques globales
  - proposant une méthode de recherche générique
  - dépendant aussi peu que possible du domaine d'application
- On parle alors de *méta-heuristique*
- Les *algorithmes génétiques* font partie de cette famille d'algorithmes.

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Algorithmes génétiques 25

- Développés par John Holland (Université du Michigan) à partir des années 1960
- Les algorithmes génétiques s'inspirent
  - de la théorie de l'évolution de Darwin
  - de la génétique

  pour proposer une méthode de recherche et/ou d'optimisation.
- Ce qu'on cherche ici est une *solution* à un problème et non un *chemin* vers cette solution

  (Même si la solution peut être un chemin !)

Darwin et la sélection naturelle 26

- Charles Robert Darwin (Grande-Bretagne, 1809–1882)
- Publie en 1859 "The origin of species" qui pose les bases de la théorie de la sélection naturelle
- Ses idées n'allaient à l'époque pas de soi :

  « La conclusion fondamentale à laquelle nous sommes arrivés dans cet ouvrage, à savoir que l'homme descend de quelque forme d'organisation inférieure, sera, je le regrette de le penser, fort désagréable à beaucoup de personnes. » (Charles Darwin, la "Descendance de l'homme et la sélection naturelle")

Sélection naturelle – idées principales 27

- Les individus possèdent un certains nombre de caractéristiques héréditaires
- Ces caractéristiques peuvent évoluer (aléatoirement, ou en tout cas de manière *non orientée*)
- Un environnement à ressources limitées provoque une sélection, qui favorise la survie des individus *les plus adaptés*
  - Sélection de survie
  - Sélection sexuelle
- Darwin n'a pas expliqué le mode de transmission de caractères héréditaires
  - Notamment, il n'a pas exclu l'hérédité des caractères acquis !

Mendel et la génétique 28

- Johann Gregor Mendel (Autriche, 1822–1884)
- Considéré comme le père de la génétique
- A vécu à la même époque que Darwin, mais il semble qu'il n'aient jamais correspondu !
- Sur la base d'expériences botaniques, publie ses fameuses "lois de Mendel" qui décrivent les règles de l'hérédité
- Il postule notamment que l'hérédité est gouvernée par des "doubles commandes" chez les deux parents et qu'une seule commande est transmise à l'enfant
- Au cours du XXe siècle, on découvrira progressivement les chromosomes et l'ADN, qui permettent d'expliciter le support de l'hérédité

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Un peu de terminologie 29

- Le support physique du matériel héréditaire est l'*acide désoxyribonucléique* (ADN), qui peut former des chaînes ("double hélice") de *nucléotides* (A,T,G,C)
- Une suite de nucléotides codant pour une protéine est appelée *gène*
- L'ensemble des gènes constitue le *génome*, dont le support physique est le *chromosome*
  - Certaines partie du génome sont *non-codantes* !

Le principe de base 30

- On choisit un code pour représenter les solutions d'un problème sous la forme de "chromosomes" (informatiques !)
- On génère (plus ou moins) au hasard une *population* représentant une partie du matériel génétique possible
- On fait ensuite évoluer cette population selon trois mécanismes :
  - La *sélection* qui a tendance à ne garder que les individus les plus adaptés (les meilleures solutions du problème)
  - Le *croisement*, qui combine les caractéristiques de deux chromosomes pour en donner un nouveau
  - La *mutation*, qui va modifier (plus ou moins) aléatoirement un ou plusieurs gènes d'un génome

Le codage 31

- Le codage est un élément important pour l'utilisation des algorithmes génétiques
- Il doit bien entendu permettre d'atteindre toutes les solutions !
- Il doit également se prêter aux opérations de croisement et de mutation
  - À noter que ces opérations n'ont pas besoin de correspondre à des changements "significatifs" du point de vue du problème...
  - ... tant que le gène obtenu code toujours une solution correcte !
- On en distingue habituellement trois types :
  - Le codage binaire
  - Le codage sur un alphabet
  - Le codage en arbre

Le codage binaire 32

- Codage utilisé à l'origine par John Holland
- Avantages
  - Maximise les possibilités de croisement et de mutation
  - Un bit donné ne représentant souvent rien de très significatif à nos yeux, l'algorithme est susceptible de trouver des solutions très originales !
  - Peut être traité de manière efficace et générique
- Désavantage majeur
  - Codage souvent peu naturel, parfois même difficile à mettre en place...

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Codage sur un alphabet 33

- Il s'agit d'un code tout à fait standard : on se donne un alphabet de base pour exprimer les solutions du problème
- Il faut parfois définir un *langage* pour délimiter les chaînes significatives
- Les mutations et croisements doivent alors rester dans le langage !
- Souvent plus naturel que le codage binaire

Codage en arbre 34

- Plutôt que de représenter une solution sous forme de chaîne "à plat", on la représente sous forme d'arbre
- Permet dans certains cas de simplifier le problème de rester dans le langage lors des croisements et mutations
- Peut permettre de mieux séparer des sous-problèmes dans des cas où la représentation est grande
  - "Si la première branche contient X, le contenu de la deuxième branche n'a pas d'importance. . . "
  - À l'extrême, pourrait même servir à représenter des solutions infinies. . .

La sélection 35

- Pour faire évoluer la population, il faut "faire de la place"
- Pour ce faire, on va à chaque pas de l'algorithme sélectionner une sous-partie de la population qui servira de base à la population suivante
- Il existe de nombreuses manière de le faire, dont nous allons survoler les plus courantes

Les types de sélection 36

Sélection uniforme On choisit au hasard n individus. Pas très efficace

Sélection par rang On choisit toujours les n meilleurs individus. Efficace, mais risque de provoquer une convergence trop rapide vers un optimum local

Sélection probabiliste Chaque individu a une probabilité d'être choisi proportionnelle à son degré d'adaptation

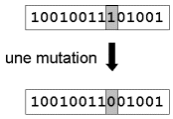
Sélection par tournois On choisit (uniformément ou non) des paires d'individus et on les fait "s'affronter" : le plus adapté sera choisi

- Chacun de ces types de sélection peut être enrichi d'un mécanisme *élitiste* : on garde toujours le meilleur individu (pour ne pas "régresser" dans notre recherche. . .)

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Croisement et mutations génériques (codage binaire) 37

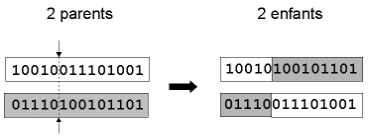
- Avec un codage binaire (ou tout codage d'alphabet A dont le langage est tout A^*), on peut définir des opérateurs de croisement et du mutation génériques
- Mutation :



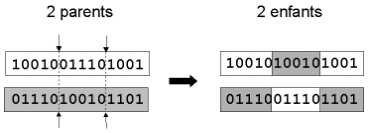
- Le croisement peut s'effectuer en 1 ou n points

Croisements en n points 38

- Croisement en un point :



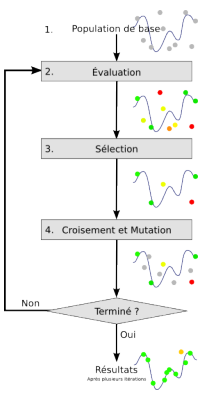
- Croisement en deux points



Croisements spécifiques 39

- Suivant le codage choisi, on peut être amené à développer des croisements et des mutations "ad hoc"
  - En particulier, le résultat du croisement doit toujours représenter une solution du problème !
- Exemple : le problème du voyageur de commerce
  - Un codage naturel est de représenter une solution par une suite de nombre représentant les villes dans l'ordre parcouru
  - Dans ce cas, les opérateurs de mutation et de croisement génériques *ne sont pas applicables* !
- Suivant les cas, on peut "optimiser" les opérateurs de croisement et/ou de mutation en ne les appliquant que s'ils amènent une amélioration (convergence plus rapide, mais exploration moins large...)

Vue d'ensemble 40



BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

L'arrêt 41

- Une question délicate est : quand considère-t-on qu'on a terminé ?
- Un algorithme génétique est un algorithme "Anytime" : il peut fournir une réponse en tout temps, mais la *qualité* de la réponse augmente avec le temps
- Critères d'arrêt possible :
  - On a trouvé une solution *assez bonne*
  - Budget (temps ou calcul) épuisé
  - Nombre de générations maximal atteint
  - Pas d'amélioration dans la qualité des meilleures solutions sur n générations
  - Observation humaine
  - ...

Quelques considérations sur l'efficacité 42

- L'idée de base des algorithmes génétiques est de se reposer sur une évolution parfaitement aléatoire pour explorer librement l'espace des solutions
- La convergence vers une bonne solution est alors amenée par le processus de sélection
- Cette idée, proche des théories de l'évolution, permet de trouver des solutions originales, mais peut être très lente...
- On peut encourager une recherche plus rapide avec des opérateurs de mutation et de croisement un peu plus "orientés"
  - Il faut cependant se méfier des optima locaux...

Quand utiliser les algorithmes génétiques 43

- Les algorithmes génétiques peuvent trouver une bonne solution de manière efficace, mais il ne constituent bien entendu pas une solution universelle !
- Il sont adaptés quand
  - L'espace des solutions est grand (sinon, autant faire une recherche exhaustive...)
  - Il n'existe pas d'algorithme déterministe suffisamment efficace
  - On préfère une *assez bonne* solution *assez rapidement* plutôt que la solution optimale en un temps indéterminé

Quelques exemples d'application 44

- Le problème du voyageur de commerce et autres problèmes de recherche
  - Conception d'horaires, ...
- Motorola semble avoir utilisé les algorithmes génétiques pour développer automatiquement des suites de tests logiciels
  - Individu : jeu d'entrée
  - Valeur d'adaptation : portions de code testées
- Pilotage d'un robot (NASA pour Pathfinder, Sony pour Aibo, ...)

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Les limites

- La principale difficulté des algorithmes génétiques est qu'ils sont très sensibles au choix des paramètres
  - Taille de la population
  - Taux de renouvellement
  - Taux de mutation
  - ...
- On est jamais sûr d'avoir trouvé la solution optimale !
- En particulier, on peut se retrouver bloqué dans un optimum local
  - Pour éviter cela, on peut essayer de toujours garder une certaine variété de population...

Proches parents

- La *programmation génétique* est basée sur les mêmes principes, mais on fait évoluer le code, pas les paramètres !
- Le *recuit simulé* utilise un seul individu qui parcourt l'espace des solutions, d'abord très librement puis de manière de plus en plus contrainte
- L'optimisation par colonie de fourmis (!) fait se promener des "fourmis" dans l'espace des solutions et utilise la diffusion de "phéromones" pour attirer d'autres fourmis dans les zones intéressantes
- ...

Pour en savoir plus...

- `http://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique`
- `http://magnin.plil.net/spip.php?rubrique8`
- `http://www.faqs.org/faqs/ai-faq/genetic/`
- ... Sans oublier une applet d'une redoutable efficacité pour résoudre le problème du voyageur de commerce : `http://www.mac.cie.uva.es/~arratia/cursos/UVA/GeneticTSP/JAVASimultn/TSP.html`

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch