Backtracking

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Backtracking IRL

Calvin & Hobbes — Bill Watterson

Backtracking : un exemple avec des ♘

Table des matières

Principe

- Technique de résolution d'un problème satisfaisant deux conditions :
  1. On connaît (un algorithme calculant) une formule logique $\varphi : \mathcal{C} \to \{V, F\}$ telle que résoudre le problème consiste à trouver $x \in \mathcal{C}$ tel que $\varphi(x) = V$.
  2. $\mathcal{C}$ est fini, et l'on connaît un algorithme permettant d'énumérer ses éléments.

Algorithme général

```
let exhaustive_search c phi =
    my_find_opt phi c
```

pour une certaine fonction de parcours `my_find_opt`…

- Dans le pire cas, le coût est donné par le coût (éventuellement amorti) de l'énumération de $\mathcal{C}$ $\times$ le coût de l'évaluation de φ.

Exemples

Problème du ♘

- On teste chacune des 63! permutations des cases (autres que celle de départ) possibles et teste si elles sont accessibles en séquence

Trier un tableau de N éléments

- On teste les $N!$ permutations possibles et teste si le résultat est trié

Une recherche exhaustive est rarement rapide…

Table des matières

Solutions partielles ; successeurs ; contraintes

Certains problèmes de recherche sont tels que l'on peut rejeter (efficacement) des solutions candidates même quand elles sont *partiellement* connues

Exemple : problème du ♘

Il ne sert à rien d'explorer les 62! circuits commençant par ♘$a_1 d_4$ puisque ce déplacement n'est pas valide…

Solutions partielles

Définition informelle

Une solution partielle d'un ensemble $\mathcal{C}$ est un élément de $\mathcal{C}$ dont certains « bouts » sont indéterminés

Exemples

Connaître seulement le début d'un mot, les premiers chiffres d'un nombre, les premières cases d'un tableau, les premiers éléments d'une liste, le « début » d'une permutation…

Représentation

Informellement : on remplace des « bouts » de la solution par des '?'

- En OCaml, une possibilité générique : passer par un `type 'a option`
  - Exemple : `(sol : int list)` devient `(psol : int option list)`, où `None` représente '?'
- En C : pas de méthode générique plaisante, mais souvent des approches *ad hoc* raisonnables

Ordre partiel pour les solutions partielles

On veut pouvoir *étendre* des solutions partielle, ce qui nécessite de leur définir des *successeurs*

Approche générale

- Pour $x \in \mathcal{C}$, on peut (généralement) faire correspondre plusieurs $y \neq x$ de $\mathcal{C}'$ qui lui sont identiques « là où ils sont définis »
- Notation possible (non standard) : $x \subset y$ (et $x \subseteq y$; $y \supset x$; $y \supseteq x$)

Exemple

- $\mathcal{C} = \{0,1\}^n, \mathcal{C}' = \{0,1,?\}^n$ et pour tout $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n \in \mathcal{C}'$, $x \subseteq y$ ssi. :

$$\bigwedge_{i=1}^{n} x_i = y_i \vee x_i = ?$$

Successeurs d'une solution partielle ; arbre de solutions partielles

Fonction successeur

On définit pour chaque solution (partielle) x un ensemble (éventuellement vide) $\mathrm{succ}(x)$ satisfaisant les deux conditions (et plus…) :

1. $y \in \mathrm{succ}(x) \to y \supset x$
2. succ induit une structure d'arbre sur $\mathcal{C}'$, dont les feuilles sont les éléments originels de $\mathcal{C}$

- Un successeur « complète » une solution partielle en définissant des valeurs qui ne l'étaient pas, et l'on peut énumérer tous les éléments de $\mathcal{C}$ une unique fois en parcourant un arbre défini par les successeurs d'une solution « vide »

Exemple

- Soit $\mathcal{C} = \{0,1\}^2, \mathcal{C}' = \{0,1,?\}^2$ on peut prendre :
  - $\text{succ}(??) = \{0?, 1?\}$
  - $\text{succ}(0?) = \{00, 01\}$
  - $\text{succ}(1?) = \{10, 11\}$
  - $\text{succ}(00) = \text{succ}(01) = \text{succ}(10) = \text{succ}(11) = \varnothing$
- Ou $\text{succ}(??) = \{00, 01, 10, 11\}$, mais manque d'intérêt

Pour que la structure de solution partielle soit intéressante, l'arbre construit par succ doit être de hauteur > 1

Test d'une solution partielle

On aimerait pouvoir définir φ' t.q. $\varphi'(x) = V$ ssi. $\exists y \supseteq x$ t.q. $\varphi(y) = V$, mais ce n'est en général pas facile

Intérêt : utiliser φ' pour :

Conditions sur φ'

Objectif

- Étant donné φ, on veut φ' t.q. si y satisfait φ, alors *toute solution partielle* $x \subseteq y$ *doit satisfaire* φ'
- Plus formellement :

$$\varphi(y) \to x \subseteq y \to \varphi'(x)$$

  - Par contraposée :

$$\neg(x \subseteq y \to \varphi'(x)) \to \neg\varphi(y)$$

  - d'où

$$(x \subseteq y) \wedge \neg\varphi'(x) \to \neg\varphi(y)$$

  - ou encore

$$\neg\varphi'(x) \wedge (y \supseteq x) \to \neg\varphi(y)$$

  - si une solution partielle est rejetée, alors elle ne *peut pas* être complétée en solution valide

Construction de φ'

Pas de méthode générale (pas possible ?) ; on esquisse juste une :

Approche *gloutonne*

- « φ' vérifie tout ce qu'il est possible de vérifier avec φ étant donné ce qui est connu de son argument »
- Affirmation : ça marche
  - On répond F seulement si l'on peut conclure et V sinon : pas de *faux négatifs* possibles

Exemple : problème du ♘

On rejette une solution partielle ssi. plus aucune case non déjà visitée n'est accessible par le ♘

Exemple : problème des trois ♖

- Objectif : placer trois ♖sur un échiquier 3×3 t.q.' aucune capture possible
- Domaine $\mathcal{C} = \{(x_i, y_i)_{1 \leq i \leq 3}\}$ des coordonnées des trois tours et :

$$\varphi((x_1, y_1), (x_2, y_2), (x_3, y_3)) =$$

$$(x_1 \neq x_2) \wedge (x_1 \neq x_3) \wedge (x_2 \neq x_3) \wedge (y_1 \neq y_2) \wedge (y_1 \neq y_3) \wedge (y_2 \neq y_3)$$

- On a alors par exemple (pour φ' glouton) :
  - $\varphi((1,1),(2,2),(3,3)) = V$
  - $\varphi'((?,?),(?,?),(?,?)) = V$
  - $\varphi'((?,1),(2,2),(3,3)) = V$
  - $\varphi'((?,1),(?,1),(3,3)) = F$
  - Sur un échiquier de taille 2×2, $\varphi'((?,?),(1,1),(2,2)) = V$, bien que cette solution partielle ne puisse pas être étendue en une solution complète

Table des matières

Principe

On résout un problème de recherche exhaustive en partant d'une solution partielle vide que l'on essaye de compléter petit à petit ; si cela devient impossible on annule le dernier choix que l'on remplace par un autre, en remontant si nécessaire

Parcours d'arbre

- C'est en fait un parcours (en profondeur) de l'arbre défini par succ et φ'
  - L'arbre n'est pas « construit » explicitement
  - On espère ne pas avoir à visiter toutes les feuilles
  - Le choix de succ et φ' impacte l'efficacité de l'approche

Algorithme de parcours d'arbre du backtracking, version générique OCaml

```
let rec _backtrack (psol : 'a) (succs : 'a -> 'a list) phi' phi
    : 'a option =
    let next = List.filter phi' (succs psol) in
    let rec loop
        = function
        | [] -> None
        | x::xs -> if phi x then Some x else
                   match _backtrack x succs phi' phi with
                   | None -> loop xs
                   | sol -> sol
    in
    loop next
```

En pratique, on ne construit pas explicitement la liste des successeurs mais la génère plutôt *on-the-fly*

Backtracking : subtilités impératives

- Un algorithme de backtracking peut s'employer indifféremment dans un contexte fonctionnel ou impératif
- Dans les cas impératifs, il est souvent nécessaire de *nettoyer* les solutions partielles infructueuses lors du retour des appels récursifs, par exemple :

```
// ...
int old = g[i];
g[i] = new_promising_value;
if (!hurray_got_a_solution(g))
{
  g[i] = old;
  return false;
}
// ...
```

- Peut éventuellement s'éviter en utilisant des structures de données (semi-)persistantes

Table des matières

Backtracking : pourquoi et où ?

Intérêt du backtracking

- Souvent une bonne alternative à la recherche exhaustive naïve… quand le problème s'y prête
  - Besoin de solutions partielles etc.
- Moins naïf, mais toujours une recherche exhaustive
  - On peut devoir explorer toutes les feuilles
  - Pas de gain de complexité en général
  - Performances en pratique parfois dures à analyser

Dans la nature

- Principe de *base* utilisé dans les *SAT solvers*
- Utilisé dans le langage de programmation par contraintes *Prolog*

Pour aller plus loin

Toutes les solutions

- Souvent facile de modifier un algorithme de backtracking pour renvoyer *toutes* les solutions du problème
  - Il suffit de traiter chaque solution et de continuer le parcours sans s'arrêter

Toutes les solutions, à la demande

- Aussi possible de *générer* toutes les solutions : un générateur produit une nouvelle solution quand on l'appelle
  - Pratique s'il y a beaucoup de solutions
  - Faisable en OCaml avec des continuations  (peu de modifications par rapport à la recherche de base, mais conceptuellement subtiles)
  - Parfois faisable de façon *ad hoc* (*cf.* TP)

Pour aller plus loin *bis*

Heuristiques, backjumping, symétries

- Ajout d'heuristique : explorer les branches prometteuses en premier
  - *Cf.* problème du ♘
- Backjumping : revenir plus d'un nœud en arrière
  - Exemple : CDCL dans les SAT solvers
- Exploiter les symétries du problème
  - Aussi valable en recherche exhaustive sans backtracking