Chapitre 11 - Exploration exhaustive

Que faire quand on ne peut pas appliquer diviser pour régner/ gloutons / programmation dynamique ?

- recherche par force brute
- backtracking

I. Recherche par force brute

1. Types de problèmes

Problèmes de décision :

Soit V un ensemble et P une propriété. On cherche $x \in V$ tel que x vérifie P

Exemples :

- V est l'ensemble des chaînes de caractères. $P(x)$ vérifie si x est le mot de passe recherché.
- V est l'ensemble des indices d'un tableau. $P(x)$ vérifie si x est l'indice de l'élément recherché.
- V est l'ensemble des remplissages possibles d'un sudoku. $P(x)$ vérifie si la grille x respecte les règles.

Plus généralement, tous les problèmes de type "puzzle, casse-tête" correspondent à des problèmes de décision.

Eventuellement, les algorithmes de type recherche exhaustive peuvent aussi permettre de résoudre :

- problème de combinatoire
- problème d'optimisation

2. Principe d'une recherche par force brute

Définition : recherche par force brute

énumération de l'ensemble des solutions potentielles de V en testant la propriété P sur chaque valeur rencontrée.

Algorithme :

```
Pour chaque valeur x dans V :
        Si P(x) est vérifiée :
                arrête la recherche avec x comme solution
fin de la recherche , aucune solution
```

Construction de l'ensemble V :

- parfois c'est immédiat (exemple : ensemble des indices d'un tableau)
- parfois on peut construire V en entier avant la recherche (si V est assez réduit)

Exemple : cryptarithme -> trouver le chiffre (entre 0 et 9) correspondant à chaque lettre

Peu de lettres et seulement 10 chiffres donc on peut construire V facilement.

- Dans le cas général :

Construire une première solution potentielle puis en déduire les suivantes au fur et à mesure.

Exemple : sudoku

On part avec la solution potentielle contenant que des 1 dans les cases vides, puis on construit la suivante en remplaçant le premier 1 par un 2, puis etc.

Complexité :

- Dépend du nombre de solution potentielle $|V|$ nombre d'itérations en $\mathcal{O}(|V|)$
- Vérification de P
- construction de la solution potentielle suivante
- ordonner les solutions potentielles afin d'avoir plus de chances de trouver la bonne assez tôt dans les itérations. (= date de balayage)

3. Complexité des algorithmes de recherche par force brute

La complexité inclut : la construction de l'ensemble des solutions potentielles, le parcours de cet ensemble, et le test déterminant s'il s'agit d'une solution.

Optimisation possible : ordonner les solutions potentielles (exemple : droite de balayage).

II. Backtracking

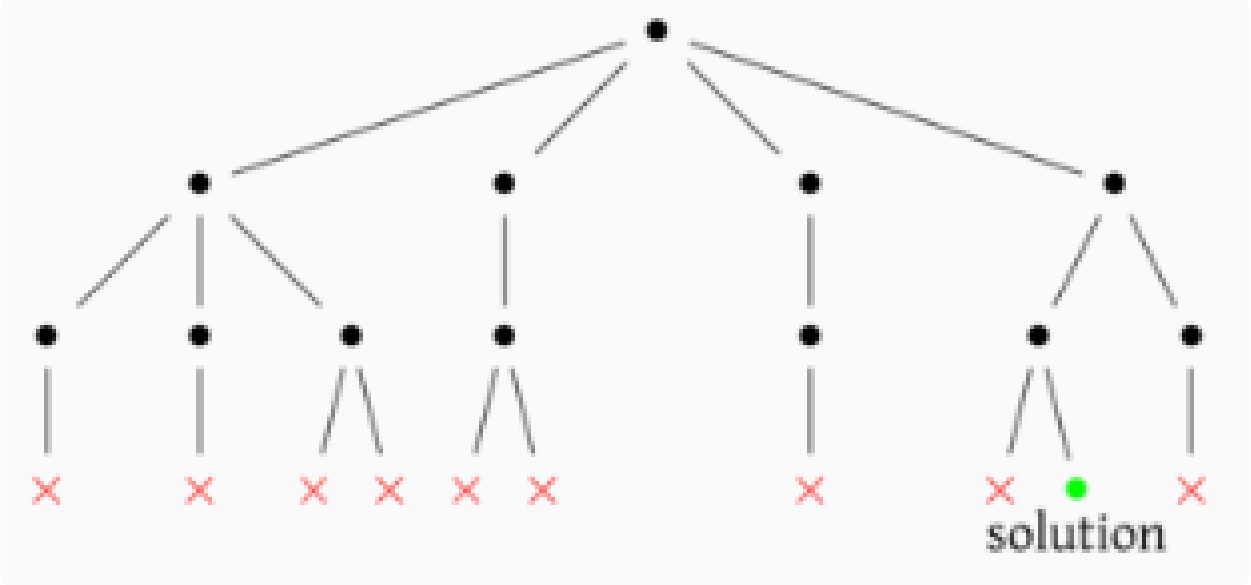
Correspond à la dernière manière de construire l'ensemble des solutions potentielles

1. Principe des algorithmes de type « retour sur trace »

Construction d'une solution potentielle petit à petit, en revenant en arrière si elle n'aboutit pas pour construire le reste des solutions potentielles.

Exemple : grille de sudoku avec n cases vides

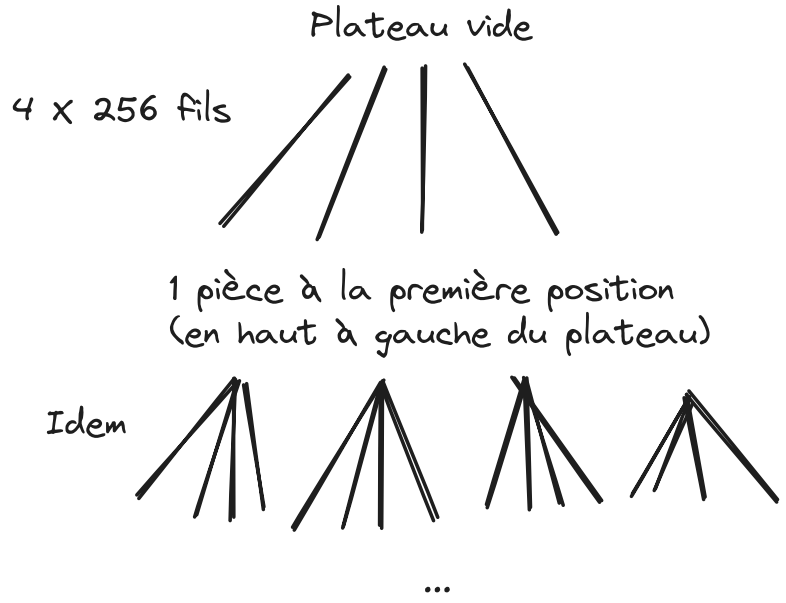
On peut représenter n'importe quel algorithme de backtracking par un arbre :



La racine représente le problème d'origine et chaque nœud interne de l'arbre correspond à la construction en cours d'une solution partielle tandis que les feuilles sont les solutions potentielles.

Exemple : puzzle Eternity

Objectif : remplir le plateau avec les 256 pièces du puzzle tel que 2 pièces adjacentes ont le même bord en commun.



Algorithme de backtracking :

- construction initiale : $X \rightarrow$ construction partielle vide = racine de l'arbre
- constructions_suivantes : construction partielle $c \rightarrow$ liste des constructions suivants c (fils de c)
- est_solution_potentielle : construction partielle $c \rightarrow$ booléen indiquant si c est une solution potentielle (si c est une feuille)
- est_solution : solution potentielle $\rightarrow$ booléen indiquant si c'est la solution au problème

Algorithme de backtracking :

```
Fonction solution():
        renvoyer backtracking(construction_initiale())

Fonction backtracking(construction):
        si est_solution_potentielle(construction) : //feuille
                renvoyer est_solution(construction)
        sinon :
                Pour chaque fils de
construction_suivantes(construction):
                        Si backtracking(fils) est VRAI :
                                renvoyer Vrai
                renvoyer Faux
```

→ Pour l'instant, le backtracking a la même complexité que la recherche par force brute car on explore toutes les solutions potentielles (dans le pire des cas) sans se soucier du

fait que certaines constructions n'ont aucune chance d'être une solution.

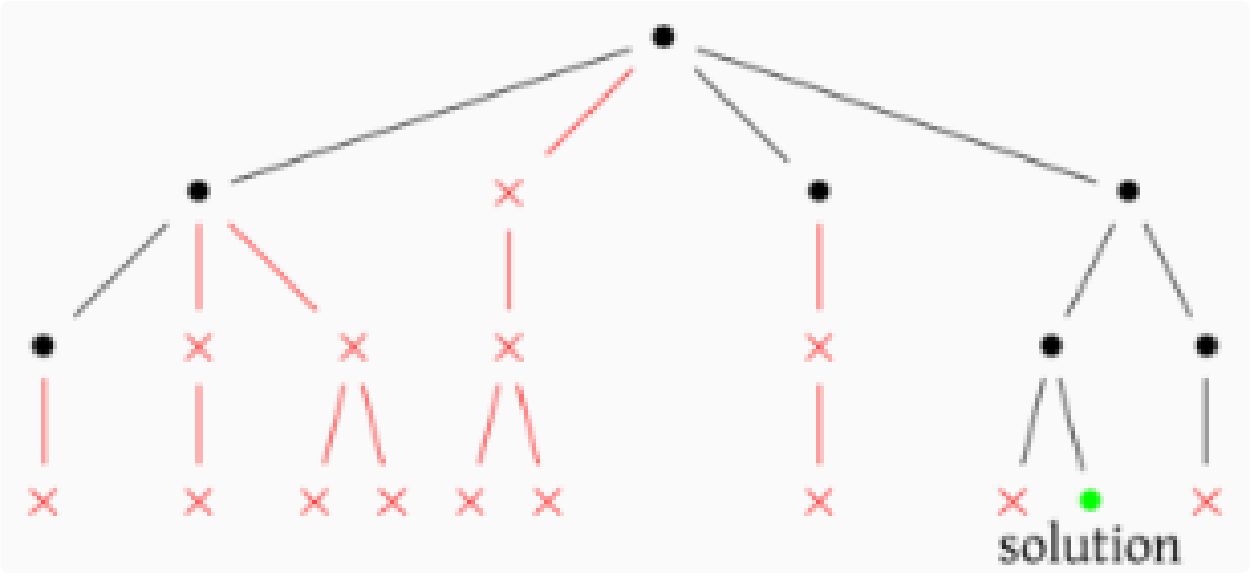
2. Optimisations

Il existe plusieurs manières d'améliorer la complexité temporelle d'un algorithme de backtracking.

Evaluation partielle :

A chaque nœud interne, on s'assure que la construction partielle a des chances d'être une solution = évaluation partielle de la construction.

Arbre du backtracking :



Elagage de l'arbre :

→ certains sous-arbres ne sont jamais explorés si l'évaluation partielle de la racine de l'arbre a échoué.

Dans l'algorithme, on ne parcourt pas les fils d'un nœud interne si l'évaluation partielle échoue

Avantage : une grande (généralement très grande) partie de l'arbre n'est pas explorée

Inconvénient : l'évaluation partielle peut parfois avoir un coût important, redondances des constructions partielles

Dans le cas où deux chemins distincts depuis la racine mènent à deux nœuds internes différents mais représentant la même configuration

Exemple : jeu du solitaire

Mémoïsation de la fonction backtracking pour ne pas explorer plusieurs fois une configuration redondante

Heuristique :

Objectif : parcourir les branches de l'arbre qui ont le plus de chance d'aboutir à une solution avant les autres

Exemple : sudoku

Avec le backtracking, on va explorer dans cet ordre :

- le sous-arbre où on a ajouté un 1 dans la première case.
- le sous-arbre où on a ajouté un 2 dans la première case.
- etc

Pour améliorer le parcours pour le sudoku :

on regarde pour chaque case vide laquelle a le moins de chiffres potentiels et on remplit d'abord celle-ci

L'heuristique h est une fonction qui à une construction partielle c associe un réel tel que plus $h(c)$ est grande, plus c a des chances d'aboutir à la solution. Puis on parcourt les fils dans l'ordre des valeurs donnés par l'heuristique