Projets

$\rightarrow$ 5 sujets

- Backtracking
  - 8 reines
  - Sudoku
- Recherche de motifs
  - Boyer-Moore
  - Rabin-Karp
- Compression LZW

Triés par ordre croissant en difficulté.

Normalement, 2 groupes par sujet

On y consacre les TPs, TDs, Khôlles (et éventuellement un cours).

Soutenances : 14 juin $10^h \rightarrow 12^h$ et $15^h \rightarrow 17^h$ (sur les heures de TIPE)

À rendre : présentation (au format PDF) et codes (C et/ou oCamL); au plus tard pour le 20 Juin.

Contenu de la présentation :

- Description des méthodes
- Exemple
- Implementations (et les choix qui ont été fait)
- Analyse de algorithme
- Résultats
- Conclusion et perspectives

L'oral durera au maximum 20-25 min (on aura la durée exacte plus tard).

I. n reines – backtracking

Groupe A. Iba, Antoine R., Bastien D., Arthur

Problème : placer le nombre maximal de reines sur un échiquier de $n \times n$.

Rappel : la reine attaque la ligne, la colonne et les deux diagonales.

Backtracking : expiration exhaustive ; on part de la racine, et on descend dans l'arbre. Si on ne trouve pas de solution, on remonte.

Algorithme : On procède ligne par ligne. Une structure est créée contenant la position des reines déjà posées et le nombre de reines à poser. On tente de poser un maximum de reines. Quand on ne peut plus poser de reines, on remonte dans l'arbre des possibilités et on essaie une nouvelle position pour la reine. La solution partielle est une liste $(r_1, \ldots, r_n)$ où r_i est la position de la i-ème reine.

Complexité : l'algorithme a une complexité en $\mathcal{O}(n!)$.

Exemples :

N	# Solutions
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724

Groupe B. Narada, Nicolas, Charlotte, Alan

Même sujet que le groupe précédent.

Conjecture : On peut placer une reine sur chaque ligne.

On n'essaie pas toutes les configurations car certaines sont impossibles ; on fixe les contraintes.

Différence avec le groupe A. : la structure contient tous les tableaux solutions. Le problème est que la complexité en mémoire est très importante et à partir de $n = 11$, il est impossible que le programme se termine sur `repl.it`.

Complexité temporelle : $\mathcal{O}(n!)$.

Complexité mémorielle : $\mathcal{O}$(beaucoup).

II. Boyer-Moore – Recherche d'un motif

Groupe A. Ruben, Noémie, Alex, Juliette, Antoine V.

Algorithme de recherche de motif dans un texte → un des algorithme les plus populaire et efficace pour la recherche de motif.

Approche naïve : on tente de placer le motif sur chaque lettre $\implies$ complexité importante

On commence par la fin du motif et s'il n'y a pas une lettre du motif, on fait un "saut" de la taille du motif moins la position dans la table de saut du caractère qui a bloqué le motif.

Il est possible que l'on rate la solution si l'on ne fait pas attention (exemple : rechercher "AAA" dans "BBAAAZZ").

Implémentation en OCaml plus adaptée que celle en C car utilisation de listes.

Groupe B. Khadim, Iwan, Émile, Tibério

Complexité de la méthode exhaustive : $\mathcal{O}(\ell \times k)$ où ℓ est la longueur de la chaîne, k est celle du motif.

Méthode avec décalage (Boyer-Moore) : meilleur des cas $\mathcal{O}\left(\frac{\ell}{k}\right)$ (approximatif).

Méthode optimisée : utilisation de liste chaînées et de plusieurs "curseurs" pour analyser le texte.

$\longrightarrow$ Pas de problème pour la recherche de "AAA" dans "BBAAAZZ"

Boyer-Moore II (hors programme) : utilisation de deux tables de saut ; si on trouve une lettre mais pas la suivante, on regarde dans le motif si l'on ne peut pas avoir trouvé une autre partie du motif et on fait le "saut" nécessaire.

III. Résolution de Sudoku – Backtracking

Groupe A. Bastien F., Thomas, Timothée S., Hugo

Bah, non. $\implies$ Complexité en $\mathcal{O}$(beaucoup).

Groupe B. Aubin, Samy, Kyriann

On choisit *arbitrairement* une solution.

Attention, ici n est la taille de la grille entière.

Utilisation du backtracking mais avec des contraintes au lieu de possibilités.

Choix à faire à chaque cases réduisant les possibilités des autres, si aucune possibilité pour une case, on change la case précédente.

Utilisation du récursif d'où l'utilisation du OCaml. Deux structures utilisées : `case_contrainte` et `case_grille`.

Utilisation d'une seule boucle pour faire à la fois la vérification ligne et colonne $\implies \mathcal{O}(n)$ où n est la taille *totale* de la grille.

Construction de la file de priorité en $\mathcal{O}(n^3)$.

Résolution du Sudoku en $\mathcal{O}\left(n^{n^2}\right)$.

Mettre à jour la liste des cases à traiter permettrais de remplir la 20 % de la grille.

IV. Compression Lempel-Ziv-Welch

Groupe A. Dorian, Lucine, Benoît, Kilian, Nathan R.

Des variantes de l'algorithme sont utilisées pour le format `GIF` et `TIFF`.

Utilisation des redondances dans le texte pour diminuer la taille → dictionnaire.

Au lieu d'utiliser 8 bits pour stocker un caractère, diminuer cette taille en fonction de la fréquence dans le texte.

Présence d'un caractère pour indiquer la fin du texte (`EOF`) → `\Z`.

Formation de séquences à partir des séquences déjà créées : "TON" utilise la séquence "TO".

Peut aller de 10 % à 50 % de compression. Pour 10 000 caractères de Harry Potter, on a 40 % de compression.

Spécificité du C : "méthode `append`"
avec des tableaux → trop coûteux
avec les liste chaînées

D'où l'utilisation de deux liste chaînées : pour le code et pour le dictionnaire.

Puis, transformation de la liste chaînée à un tableau pour la fin de la fonction principale.

Différence : ajout d'un champ `queue` à la liste chainée pour avoir une complexité d'ajout d'un élément en $\mathcal{O}(1)$.

(ajout de Khadim : on peut utiliser une fonction Hors-programme qui permettrai l'utilisation des tableaux au lieu des listes chaînées).

Utilisation du code `ASCII` pour représenter la séquence.

Autre possibilité : utilisation d'une fonction récursive ? $\implies$ OCaml

Pour le code en OCaml, utilisation d'un tableau pour représenter le dictionnaire.

Améliorations : deuxième passage pour détecter des motifs de motifs ($\rightsquigarrow$ mots communs) uniquement possible car on stocke le dictionnaire pour la décompression. Possibilité d'un troisième passage, etc...

Décompression en $\mathcal{O}(n)$.

Groupe B. Timothée N-L., Nathan S., Lilian, Loïc

Utilisation également dans le format `ZIP`.

Surtout efficace sur des textes longs car répétition de motifs.

Utilisation d'indices supérieurs à 255 $\implies$ plusieurs octets (contrairement au code `ASCII`).

Utilisation de tableaux pour représenter le dictionnaire (contrairement au groupe précédent).

$\rightsquigarrow$ retrouver le dictionnaire à partir du texte décodé.

Complexités : - compression en $\mathcal{O}(n \times p)$ où la taille du code est q et celle du dictionnaire est n ; - décompression en $\mathcal{O}(n^2 \times p)$.

Économie en mémoire allant jusqu'à 50 %. Bien plus efficace sur les textes longs ou sur les images.

Sans avoir accès au dictionnaire, on ne peut pas compresser deux fois (ou plus).