TP 02 – Trois mini-projets

Langage de programmation au choix (à valider avec votre chargé de TP).

Cette planche est composée de trois exercices (mini-projets) indépendants.

Votre rapport devra expliquer le fonctionnement de vos algorithmes, développer les analyses de complexité, et commenter les graphiques obtenus expérimentalement.

N'oubliez pas d'inclure avec votre code un fichier readme décrivant son organisation, ainsi que les commandes pour compiler (le cas échéant) et exécuter vos programmes.

Exercice 1. *Diviser-pour-régnier*

Le tri deux-tiers est un algorithme récursif qui fonctionne en trois temps :

- **(a)** trier les premiers $2/3$ du tableau,
- **(b)** trier les derniers $2/3$ du tableau,
- **(c)** trier les premiers $2/3$ du tableau.

Si le tableau est de taille inférieure ou égale à deux, alors on le trie directement.

1. Implémenter l'algorithme du tri deux-tiers.
2. Expliquer pourquoi cet algorithme trie correctement tout tableau.
3. Analyser sa complexité dans le pire cas, à l'aide du Master théorème.
4. Tester votre algorithme sur des tableaux remplis aléatoirement. Produire un graphique.

Exercice 2. *Sudoku*

Le Sudoku est un jeu de placement des chiffres de 1 à 9 dans une grille de 9 lignes et 9 colonnes, également découpée en 9 zones de taille 3×3. Les chiffres de 1 à 9 doivent être placés de façon à ce que chaque ligne, chaque colonne et chaque zone contienne tous les chiffres de 1 à 9. Étant donnés des chiffres initialement placés dans la grille, le problème de décision du Sudoku consiste à savoir si la grille peut être complétée ou non en respectant les règles ci-dessus.

1. Donner un exemple d'instance négative du problème de décision du Sudoku.
2. En notation asymptotique, quelle est la complexité d'un algorithmique de résolution du jeu Sudoku sur une grille de taille 9×9 ?

On généralise le Sudoku aux grilles de taille $n^2 \times n^2$, qui peuvent être découpées en n^2 zones de taille $n \times n$. Le jeu utilise alors n^2 symboles différents pour remplir les cases de la grille.

3. Expliquer comment modéliser une instance du problème Sudoku généralisé à la taille $n^2 \times n^2$, comme une formule propositionnelle qui est satisfaisable si et seulement si la grille a une solution. Compter son nombre de variables et de clauses en fonction de n.
4. Implémenter votre réduction vers SAT de la question 3, pour la taille $n = 3$ (le Sudoku classique), en suivant la structure suivante :
   - **(a)** Votre programme prendra en entrée une grille 9×9 comme une chaîne de 81 caractères sur l'alphabet $\{1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$, où `.` indique une case initialement vide (les lignes de la grille sont cacaténées, voir benchmarks en question 5), et produira une formule au format DIMACS (`https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`).

(b) Votre programme appelle `minisat` (`http://minisat.se/`) pour décider la satisfiabilité, et obenir un modèle si possible.

(c) Votre programme doit inclure une fonctionnalité d'affichage (dans le terminal, c'est suffisant) de la grille de départ et de sa solution (si celle-ci existe).

5. Mesurer expérimentalement le nombre de grilles que votre procédure peut résoudre en 1 seconde, pour chacun des deux jeux de données de grilles suivants :

— `https://pageperso.lis-lab.fr/kevin.perrot/benchmarks/benchmark_sudoku_1.txt`

— `https://pageperso.lis-lab.fr/kevin.perrot/benchmarks/benchmark_sudoku_2.txt`

Source : `https://github.com/t-dillon/tdoku/` benchmarks `17_clue` et `forum_hardest_1905_11+`.

Exercice 3. *Presque-stable*

Dans un graphe non-orienté sans boucle $G = (V, E)$, un *presque-stable* est un sous-ensemble de sommets qui ont au plus une arête entre eux, c'est-à-dire $X \subseteq V$ tel que $|E \cap X^2| \leq 1$.

1. Créer deux générateurs de graphes aléatoires :

(a) Le premier prend en entrée un entier n et un réel $0 \leq p \leq 1$, et construit un graphe à n sommets où chacune des $n(n-1)/2$ arêtes existe avec probabilité p.

(b) Le second prend en plus un entier k en entrée, commence avec le même principe qu'en **a**, puis post-traite le graphe pour qu'il contienne un presque-stable de taille k (sur un sous-ensemble de k sommets choisis aléatoirement).

Pour chacun des algorithmes suivants, vous présenterez son fonctionnement, une analyse de complexité, et une étude expérimentale à l'aide des deux générateurs de la question 1.

2. Implémenter un algorithme de vérification de presque-stable, prenant en entrée un graphe $G = (V, E)$ et un sous-ensemble $X \subseteq V$, et vérifiant si X est un presque-stable de G.

Pour un graphe donné, un presque-stable $X \subseteq V$ est dit *maximal* lorsqu'il n'existe aucun autre presque-stable $Y \subseteq V$ qui le contient strictement, c'est à dire avec $X \subsetneq Y$.

3. Implémenter un algorithme prenant en entrée un graphe $G = (V, E)$, et retournant un presque-stable maximal de G.

Pour un graphe donné, un presque-stable $X \subseteq V$ est dit *maximum* lorsqu'il n'existe aucun autre presque-stable $Y \subseteq V$ plus grand que lui, c'est-à-dire avec $|X| < |Y|$.

4. Implémenter un algorithme prenant en entrée un graphe $G = (V, E)$, et retournant un presque-stable maximum de G.

5. Implémenter un algorithme proposant un compromis entre la recherche d'un presque-stable maximal et maximum, en termes de temps de calcul et de qualité de la solution retournée. C'est-à-dire, essayer de faire mieux que maximal, en temps polynomial.

On considère maintenant le problème de décision qui consiste, étant donné un graphe G et un entier k, à décider si G possède un presque-stable de taille k. (Ce problème est NP-complet.)

6. Pour toute instance (G, k), donner une formule $\varphi_{G,k}$ telle que $\varphi_{G,k}$ est satisfaisable si et seulement si G possède un presque-stable de taille k.

7. Comparer expérimentalement l'efficacité en temps de cette réduction vers SAT suivie d'un appel à `minisat` (`http://minisat.se/`), à une adaptation de votre réponse à la question 4 (maximum) pour résoudre exactement ce même problème de décision.