Projets d'Algorithmique.

Cahier des charges et éléments d'évaluation

Les projets d'algorithmique se feront par groupe (d'au plus 3 personnes). Chaque groupe choisira un des sujets présentés ci-dessous (plusieurs groupes peuvent choisir un même sujet).

Chaque projet vise à mettre en pratique les différentes compétences/connaissances que nous avons essayées de dispenser au cours du bloc "Algorithmique". L'idée est donc, outre le fait de programmer, que vous réflechissiez à la correction et aux performances des différents algorithmes que vous proposerez (et que vous compariez vos algorithmes). Idéalement (si possible), il serait intéressant que vous compariez différents algorithmes que différents groupes proposeraient pour un même projet.

Chaque projet essaie tout d'abord de vous guider en vous fournissant les structures de données que vous pourrez utiliser pour le mener à bien, puis en vous guidant vers quelques fonctions simples nécessaires au projet, enfin, plus ou moins rapidement, nous vous laissons autonomes dans vos choix. Pour aller plus loin, il faut faire comme tous les programmeurs : se rappeler que le web est votre ami (à condition de trouver les bons mots clé (à défaut, vous pouvez bien sûr nous contacter).

Le rendu des projets se fera sous forme d'un code Python **bien commenté ! ! !** (c'est très important de toujours écrire des commentaires dans le code, décrivant ce que représentent les variables, ce que sont sensées retourner les fonctions, comment exécuter votre code...) et d'un rapport (3-4 pages) décrivant votre travail. Dans le rapport, nous attendons que vous décriviez en quelques lignes (the least the best... si c'est compréhensible) chacun de vos algorithmes (y compris ceux donnés dans le sujet), leur correction et leur complexité.

Un code qui ne fonctionne pas ne sera généralement pas un mauvais point (n'exagérons rien, les fonctions les plus simples doivent être exécutables et renvoyer un résultat correct) dans la mesure où votre rapport décrira qu'est ce qui ne va pas (dans quels cas il ne donne pas la réponse attendue) et ce que vous avez essayé (même sans succès) pour y remédier.

Le rapport ainsi que le code python seront à envoyer à l'adresse suivante : *nicolas.nisse@inria.fr*.

Remarque : Nous vous invitons à lire tous les projets (quel que soit celui que vous choisirez) puisque des indications (aides en Python...) données dans certains projets peuvent être utiles dans d'autres.

1 Ensemble indépendant de poids maximum

Dans ce projet, nous nous intéressons au problème suivant :
Problème de l'ensemble indépendant de poids maximum dans les graphes.
Entrées : un graphe sommet-pondéré $G = (V, E, w)$ avec $w(u) \in \mathbb{R}^+$ pour tout sommet $u \in V$. (L'ensemble E représente les arêtes du graphe).
Sortie : un ensemble indépendant $S \subseteq V$ (ensemble de sommets deux à deux non adjacents) qui maximise la somme des poids $\sum_{u \in S} w(u)$.

Pour coder un graphe, vous pourrez utiliser des listes d'adjacences (chaque sommet a une liste contenant ses voisins). Plus précisément, vous pouvez utiliser un dictionnaire qui à chaque sommet (codé par un entier) associe sa liste de voisins.

```
sage: grapheVide = {}
sage: grapheVide
{}
sage: graphe = {0: [1,2,3],1: [0,4],2:[0,4],3:[0],4:[1,2]}
sage: graphe
{0: [1, 2, 3], 1: [0, 4], 2: [0, 4], 3: [0], 4: [1, 2]}
sage: graphe[3]
[0]
sage: graphe[4]
[1, 2]
sage: graphe[5]=[3]
sage: graphe[3].append(5)
sage: graphe
{0: [1, 2, 3], 1: [0, 4], 2: [0, 4], 3: [0, 5], 4: [1, 2], 5: [3]}
sage: graphe[6]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-68-c7393e9f4d30> in <module>()
----> 1 graphe[Integer(6)]

KeyError: 6
sage: graphe.has_key(6)
False
sage:
```

Dans l'exemple ci-dessus, à la ligne 4, on crée un graphe (appelé "graphe") avec 5 sommets $0, 1, \cdots, 4$. Il s'agit d'un "carré" formé par les sommets $0, 1, 4, 2$ (dans cet ordre) plus le sommet 3 adjacent à 0. Les lignes 11 et 12 ajoutent un sommet 5 adjacent à 3. Notons qu'il est important de vérifier qu'un dictionnaire contient bien une clé avant de l'utiliser.

Pour réaliser ce projet, certaines fonctions devront être utilisées : par exemple étant deux sommets, savoir s'ils sont voisins.

Un ensemble $I \subseteq V$ de sommets est un indépendent maximal si, pour tout somme $v \in V$, $I \cup \{v\}$ n'est pas indépendent. Un ensemble $I \subseteq V$ de sommets est un indépendent maximum si, pour indépendent $J \subseteq V$, $w(J) \leq w(I)$ (avec $w(I) = \sum_{v \in I} w(v)$).

- Décrivez et programmez deux algorithmes (de complexité polynomiale) différents qui, étant donné un graphe sommet-pondéré en entrée, calculent un ensemble indépendant. Un algorithme pourra être très basique et servir essentiellement pour les comparaisons. L'autre pourra être l'algorithme que vous pensez efficace pour notre problème (par exemple, un algorithme glouton calculant un indépendent maximal). Prouvez la correction de vos algorithmes et donnez leur complexité temporelle (ici, on comptera le nombre d'itérations des diverses boucles).
- Décrivez et programmez un algorithme qui calcule un ensemble indépendant de poids maximum (avec une complexité possiblement exponentielle). Prouvez la correction de votre algorithme et donnez sa complexité temporelle (on comptera encore le nombre d'itérations des diverses boucles).
- Un arbre est un graphe connexe (il existe un chemin de tout sommet vers tout autre sommet) sans cycle. Décrivez et programmez un algorithme qui calcule un ensemble indépendant maximum. On commencera par un arbre non pondéré (tout sommet est de poids 1), auquel cas un algorithme glouton est suffisant. Puis vous considérerez le cas pondéré, où un algorithme par programmation dynamique (nécessitant le calcul de 2 "sous-problèmes") est possible. Prouvez la correction de vos algorithmes et donnez leur complexité.
  Remarque : les arbres sont des graphes propices aux algorithmes par programmation dynamique pour de nombreux problèmes : on calcule récursivement des solutions pour les sous-arbres (sous-problèmes) et on combine les solutions.

— Programmez un algorithme qui permettra de générer des graphes aléatoires avec comme paramètres le nombre de sommets n et la probabilité p d'existence d'une arête entre deux sommets. (voir sur le web la notion de graphes Erdös-Renyi).
— Testez vos algorithmes pour différents graphes générés aléatoirement en faisant variés les paramètres n et p. Les comparaisons porteront sur le temps de calcul et la valeur de la solution (somme des poids de l'ensemble indépendant). Il sera opportun de faire plusieurs graphes aléatoires par jeu de paramètres pour avoir des observations qui se rapprochent de la moyenne.

2 Puissance 4

Programmez des algorithmes qui jouent (de manière efficace, si possible optimale) contre un humain (ou un autre programme).

Pour l'interface graphique, nous vous proposons la solution suivante (si vous voulez vous essayer à de "vraies/jolies" interfaces graphiques allez y, mais sachez que nous ne pourrons pas vous aider).

Le plateau de jeu sera représenté par un tableau de tableaux (ou une matrice si vous préférez) de caractères. Un 0 représentera une case vide, un $'J'$ sera un pion jaune et un $'R'$ un pion rouge. Voici un exemple :

```
sage: x = [0,0,0,'J','R',0]
sage: y = [0,'R','R','J','R',0]
sage: t = [x,y]
sage: for i in range(len(t)):
....:     for j in range(len(t[i])):
....:         print str(t[i][j]),
....:     print
....:
0 0 0 J R 0
0 R R J R 0
sage:
sage:
sage: def InitGrid(n,m):
....:     return [[0 for i in range(m)] for j in range(n)]
....:
sage: def printGrid(t):
....:     for i in range(len(t)):
....:         for j in range(len(t[i])):
....:             print str(t[i][j]),
....:         print
....:
sage: g = InitGrid(3,4)
sage: printGrid(g)
0 0 0 0
0 0 0 0
0 0 0 0
sage:
```

Testez ces fonctions. Testez la fonction *printGrid* sans la virgule à la fin de la ligne 6. sans le "*print*" à la dernière ligne.

Pour commencer, on peut définir des fonctions faciles qui seront clairement utiles.

— Donner une fonction qui prend en entrée une grille (un tableau de tableaux comme défini ci-dessus) et décide si elle est valide (on ne peut pas avoir un 0 "sous" un $'J'$ ou un $'R'$.
— Donner une fonction qui prend en entrée une grille et une couleur est dit si le joueur de cette couleur a gagné (4 pions alignés horizontalement, verticalement ou en diagonale).

— Donner une fonction qui prend en entrée une grille et une couleur et renvoie la grille obtenue après que le joueur de cette couleur a ajouté un pion dans la colonne i.

Ensuite, on pourra faire un programme qui pourra faire jouer 2 humains l'un contre l'autre. Pour cela on jouera avec des fonctions simples "entrées/sorties" de Python. On pourra utiliser la fonction "print" (par exemple : print "Joueur Rouge, tapez le numéro de la colonne dans laquelle vous voulez insérez un pion") suivie de la fonction "input" (voir `https://mathsp.tuxfamily.org/spip.php?article232`) qui permet facilement de récupérer des entrées tapées au clavier.

Maintenant, on en arrive à la partie intéressante. Programmer une "intélligence artificielle" (nous employons volontairement ce mot pompeux pour dire "un algorithme") pour jouer. Plusieurs algorithmes simples peuvent être utilisés au début.

— À chaque tour, le joueur ordinateur met un pion dans une colonne aléatoire. Le slide 30 de la présentation sur l'IA donne un exemple de génération de réel aléatoire en Python.
— À chaque tour, le joueur ordinateur essaie d'abord d'empêcher l'autre joueur de gagner (si l'autre joueur a un coup gagnant, l'ordinateur joue un coup qui "tue" le coup gagnant). Sinon, il joue un coup aléatoire.
— À chaque tour, le joueur ordinateur essaie d'abord de jouer un coup gagnant pour lui (s'il existe). Sinon, il essaye d'empêcher l'autre joueur de gagner (si l'autre joueur a un coup gagnant, l'ordinateur joue un coup qui "tue" le coup gagnant). Sinon, il joue un coup aléatoire.

Les 2 derniers exemples précédents sont des exemples d'algorithmes **gloutons**.

Il serait intéressant de faire jouer vos différents algorithmes les uns contre les autres et d'établir des statistiques sur le(s)quel(s) gagne(nt).

Pour le puissance 4, il existe un algorithme connu gagnant à tous les coups (s'il commence). Idéalement, programmez le.

3 Jeu de Hex

Même questions (et indications) que pour le puissance 4 mais avec le jeu de Hex : `https://fr.wikipedia.org/wiki/Hex`.

Remarque : De même que pour le puissance 4, on peut prouver qu'il existe un algorithme gagnant pour le premier joueur. Au contraire du puissance 4, dans le cas du jeu de Hex, on ne connaît pas cet algorithme!!! Étonnant, non ?

4 Étude théorique des algorithmes de tri et comparaison empirique

Dans ce projet, on veut étudier et comparer les complexités temporelles (en pire cas, en moyenne, en meilleur cas) et spatiale de différents algorithmes de tris.

La première étape consiste en l'implémentation de nombreux algorithmes de tris (dont beaucoup vus en cours) : nous attendons (au moins)

— tri par insertion
— tri à bulles
— tri-fusion
— tri-rapide
— tri par tas

— tri par dénombrement (décrit dans la remarque avant la section 4.5 dans les notes de cours)
— Radix sort (qu'on vous laisse découvrir par vous même)

Pour chacun de ces algorithmes, donner (et prouver) les complexités temporelles (en pire cas, en meilleur cas) et spatiale. Pour chacun d'eux donner un tableau qui atteint le pire cas et un tableau qui donne la meilleure complexité temporelle. Déterminer la complexité en moyenne du tri rapide (exercices 52 à 58 des notes de cours).

La seconde étape consiste à réaliser une batterie de simulations, pour chacun de ces tris, à l'aide de tableaux générés alátoirement. (Le slide 30 de la présentation sur l'IA donne un exemple de génération de réel aléatoire en Python.) Vous mesurerez les temps moyens d'exécution (cf `https://docs.python.org/fr/3/library/time.html`) pour différentes tailles de tableaux, présenterez les résultats obtenus sous formes de courbes (cf. `https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/4740942-maitrisez-les-possibilites-offertes-par-matplotlib`) et analyserez les résultats (comparant les algorithmes entre eux et vis-à-vis de l'étude théorique).

Pour aller plus loin, faîtes de même avec la complexité spatiale mesurant l'espace mémoire utilisé.

5 Jeu du solitaire (casse-tête)

Dans ce projet, nous nous intéressons au casse-tête du solitaire :
https ://fr.wikipedia.org/wiki/Solitaire_(casse-tête)

— Vous programmerez les algorithmes permettant de jouer. On pourra représenter le plateau par une matrice par exemple. Au départ il y a des billes sur chacune des cases. La toute première étape du jeu consiste à enlever une première bille. Les étapes suivantes consistent à déplacer une bille pour enlever une autre (principe du solitaire).
  Vous programmerez notamment une fonction qui, étant donné deux cases et vérifiant toutes les conditions requises, déplace la bille de la première case sur la deuxième et enlève la bille qui est entre les deux cases. Les conditions requises sont que la première case doit contenir une bille, la deuxième non, les deux cases doivent être sur une même ligne ou une même colonne, il doit y avoir une seule case entre les deux cases passées en entrée et cette case entre les deux doit contenir une bille. Autrement dit, ce sont les conditions pour faire un mouvement valide à ce jeu.
— Vous programmerez trois algorithmes qui permettent de déterminer une stratégie pour le jeu du Solitaire. Un de ces algorithmes sera optimal dans le sens qu'il déterminera une stratégie optimale (qui laisse le moins de billes à la fin). Vous évaluerez leurs complexités.
— Vous comparerez (en termes de qualité de la solution et du temps de calcul) vos algorithmes en faisant varier les plateaux (tailles, formes). Nous pourrons notamment commencer par des plateaux de forme rectangulaire et d'assez petites tailles.
— Pour conclure, pour toutes les dimensions de plateaux rectangulaires contenant au plus x billes, vous donnerez le plus petit nombre de billes qu'il est possible de laisser à la fin du jeu (nombre de billes après une stratégie optimale). Pour cela, vous pourrez utiliser votre algorithme optimal. Vous pourrez donner vos résultats sous forme de tableau. Enfin, vous calibrerez la valeur de x permettant de calculer ce tableau en un temps raisonnable.

6 Étude d'exemples classiques d'algorithme Diviser pour régner

Répondre formellement aux questions des sections 3.2.2 (multiplication de polynômes) et 3.3 (multiplication de matrices) des notes de cours (exercices 23 à 32). Pour chacun des 2 problèmes, les 2 algorithmes demandés (naïf et diviser pour régner) devront être implémentés et comparés par des simulations (voir la section 4 plus haut pour la marche à suivre pour les simulations et l'analyse des résultats). Vous finirez par une petite étude bibliographique sur, pour chacun des problèmes, quels sont les meilleurs algorithmes (en théorie et en pratique) existant et utilisés actuellement.

7 Problème des k plus proches voisins avec des quadtrees

Un point du plan est défini par le couple de ses coordonnées. On se donne un entier k, un ensemble S de points (représenté par un tableau de points) et un point P. Comme dans le cours, le but du problème est de déterminer les k points de S les plus proches de P.

Tout d'abord, écrire en Python une fonction qui calcule séquentiellement la distance entre P et chaque point de S, les trie (on écrira l'algorithme de tri) dans l'ordre croissant des distances et renvoie les k points les plus proches de P. Prouvez que votre algorithme est correct et donnez sa complexité en fonction du nombre de points (du cardinal de S). Pour déterminer la complexité, on comptera le nombre de comparaisons, d'opérations (opérations arithmétiques, racines...) et d'affectations effectuées.

Le but du problème est de répondre au même problème lorsque l'ensemble S est donné sous forme d'un quadtree (défini en cours) et de comparer les performances de l'algorithme utilisant un quadtree avec celles de l'algorithme trivial décrit plus haut.

La première étape est donc de comprendre comment peut être codé un quadtree en Python et comment transformer le tableau de points en entrée en quadtree. Pour vous aider à démarrer, nous vous proposons une façon d'implémenter un quadtree et quelques fonctions préliminaires pour les construire dans le fichier *quadtree.py* dans le Dropbox. Voyez également les commentaires de la section 1 pour l'utilisation basique de dictionnaires.

8 Code de Huffman

Pour ce projet, nous vous laissons presque sans indications. La compression de fichiers est un procédé important puisque cela permet de "gagner de la place en mémoire" (vous avez tous déjà "zipper" un fichier pour réduire sa taille afin que vous puissiez l'envoyer par mail). Ici, on considère la compression de texte (séquence de lettres). Le code de Huffman permet une telle compression. Nous vous conseillons fortement de voir cette vidéo : `https://www.youtube.com/watch?v=oqMx1cuw6mo` (même (et en fait, surtout) si vous ne choisissez pas ce projet) pour une excellente vulgarisation de ce concept.

Le but de ce projet est de concevoir deux algorithmes : un de compression et un second de décompression de texte en utilisant le code de Huffman. Bien sûr, il faudra expliquer vos algorithmes, prouver leur correction et déterminer leur complexité temporelle.

Discutez du gain en espace mémoire avant et après compression.

9 Algorithmes autour des labyrinthes

9.1 Attendus de base

— Créer un générateur de labyrinthes pour différents algorithmes : par exemple, regarder `https://www.jamisbuck.org/mazes/` pour une source d'inspiration. Dans le rapport, présenter les algorithmes, leur complexité, les avantages/inconvénients de chacun. Le but n'est pas d'être exhaustif, mais plutÃ´t d'en choisir quelques uns (3 à 5) ayant des caractéristiques remarquables (aspect, complexité, etc.)

— Présenter et implémenter des algorithmes (2-3) permettant de sortir du labyrinthe. Comme précédemment, il s'agit de choisir quelques algorithmes, de les implémenter, de les présenter et comparer leurs avantages/inconvénients.

— La visualisation peut se faire avec du texte ; il n'est pas indispensable d'utiliser une bibliothèque graphique. Dit autrement, l'aspect du résultat ne sera que peu valorisé dans l'évaluation. Inutile donc d'y passer trop de temps.

Vous trouverez en https ://gitlab.inria.fr/diu-algo/labyrinthe une ébauche de projet, avec un exemple de générateur.

9.2 Possibilités d'extensions

Si les points précédents sont couverts, et qu'il reste du temps, les sujets ci-dessous peuvent être traités. L'ordre n'est pas du tout indicatif, et il s'agit plutÃ´t de choisir celui qui vous intéresse le plus (dans le cadre de son exploitation pédagogique ultérieure).

— Considérer des labyrinthes en trois dimensions ;

— Ajouter des objets aléatoires (des coffres contenant des pièces d'or) et trouver un chemin qui maximise le gain (en considérant par exemple qu'un déplacement coÃ≫te -1).

— Ajouter des monstres et calculer un chemin avec la meilleure probabilité de sortir vivant (et le plus riche possible, cf. point précédent) du labyrinthe (avec ces règles `http://litteraction.fr/article/litterature-interactive/systeme-additif-simple`).