Algorithmique Avancée 2022
Projets

A) Règles

Le choix des projets sera fait le mercredi 9/11/2021 à 9h00. Vous trouverez ci-dessous 52 propositions de sujets (incluant les 23.A,B,C,D) que vous devez vous répartir individuellement.

a) Quoi rendre et quand ?

Vous devrez rendre la version complète de votre projet.
Il faut fournir une version électronique.
Cette version comprend, a minima, un **rapport** sur votre travail (choix, faits, problèmes rencontrés, code d'illustration, mode d'emploi, exemple d'utilisation, discussion...), le **code** lui-même (un ou plusieurs fichiers y compris le Makefile correspondant).
Il est conseillé d'envoyer le plus régulièrement possible des versions intermédiaires.
Le projet doit fonctionner sur l'une au moins des machines du Bocal.
Vous douvez rendre votre travail :
Avant le jeudi 8 décembre, 12h, et le présenter au cours de la semaine suivante.

b) Améliorations

Il est possible d'améliorer votre travail pratiquement jusqu'à la présentation mais il faut envoyer les améliorations pour qu'elles soient validées.

c) Présentation de vos travaux

Les présentations devront être préparées de façon à ne pas dépasser dix minutes. Elles peuvent inclure des diapositives et une démonstration...

d) NB

Vous trouverez à la fin des énoncés des fonctions permettant de

- remplir une grande matrice inversible
- générer un graphe non pondéré
- générer un graphe pondéré.

B) Droites

Dans tous ces projets, il faut :

- Programmer un algorithme de référence, par exemple Bresenham'65.
- Programmer l'algorithme demandé.
- Comparer en temps et en utilisation mémoire les deux algorithmes.
- Améliorer significativement l'algorithme demandé.

Attention, il faut que votre travail permette de **vérifier** que les valeurs obtenues sont les bonnes et aussi de **mesurer** le temps, donc avec des données assez grandes et répétitives. Le temps d'affichage venant grêver ces valeurs, l'affichage sera omis lors du test de temps.

1. **Berstel**
2. **Dulucq-Bourdin**
3. **Boyer-Bourdin'2000**
4. **Boyer-Bourdin'1999**
5. **Pas-de-trois**
6. **Castle and Pitteway**
7. **Angle and Morrisson**

C) Graphes

Nous avons vu quatre structures pour stocker des graphes : matrice d'adjacence `Mat`, liste d'adjacence `Lis`, vecteur de triplets `Vec`, tableau de brins `Brin`. Pour les projets sur les graphes, vous devrez répondre au problème posé avec les structures demandées, et discuter des résultats en terme de temps de calcul et de taille de la mémoire utilisée.

8. **Composantes connexes 1.** Sur de grands graphes (plus de 10000 noeuds) remplis aléatoirement par une fonction *remplir_graphe*, mettre en place les algorithmes de recherche du plus court chemin et de composantes connexes avec `Mat` et `Brin`.

9. **Composantes connexes 2.** Sur de grands graphes (plus de 10000 noeuds) remplis aléatoirement par une fonction *remplir_graphe*, mettre en place les algorithmes de recherche du plus court chemin et de composantes connexes avec `Lis` et `Vec`.

10. **Composantes connexes 3.** Sur de grands graphes (plus de 10000 noeuds) remplis aléatoirement par une fonction *remplir_graphe*, mettre en place les algorithmes de recherche du plus court chemin et de composantes connexes avec `Lis` et `Brin`.

11. **Voyageur de commerce 1.** Le problème du voyageur de commerce est un problème d'optimisation qui consiste à déterminer, étant donné une liste de villes et les distances entre toutes les paires de villes, le plus court chemin passant par chaque ville une et une seule fois. Proposer une méthode de résolution aussi satisfaisante que possible du problème du voyageur de commerce avec les structures `Mat` et `Brin`.

12. **Voyageur de commerce 2.** Le problème du voyageur de commerce est un problème d'optimisation qui consiste à déterminer, étant donné une liste de villes et les distances entre toutes les paires de villes, le plus court chemin passant par chaque ville une et une seule fois. Proposer une méthode de résolution aussi satisfaisante que possible du problème du voyageur de commerce avec les structures `Lis` et `Vec`.

13. **Voyageur de commerce 3.** Le problème du voyageur de commerce est un problème d'optimisation qui consiste à déterminer, étant donné une liste de villes et les distances entre toutes les paires de villes, le plus court chemin passant par chaque ville une et une seule fois. Proposer une méthode de résolution aussi satisfaisante que possible du problème du voyageur de commerce avec les structures `Vec` et `Brin`.

14. **Gestion d'un réseau de transports 1.** On voudra gérer le réseau de transports en commun en métro de la ville de Tokyo, via un graphe implémenté avec les structures `Lis` et `Vec`. Résoudre alors le problème du plus court chemin afin de trouver le meilleur itinéraire d'une station vers une autre, par exemple avec Dijkstra.

15. **Gestion d'un réseau de transports 2.** On voudra gérer le réseau de transports en commun en tramway et en bus de la ville de Reims, via un graphe implémenté avec les structures `Lis` et `Brin`. Résoudre alors le problème du plus court chemin afin de trouver le meilleur itinéraire d'une station vers une autre, par exemple avec Dijkstra.

16. **Gestion d'un réseau de transports 3.** On voudra gérer le réseau de transports en commun en métro de la ville de Bruxelles, via un graphe implémenté avec les structures `Mat` et `Brin`. Résoudre alors le problème du plus court chemin afin de trouver le meilleur itinéraire d'une station vers une autre, par exemple avec Dijkstra.

17. **Cols des Alpes.** On voudra représenter les routes entre les cols de la chaîne de montagne des Alpes dans un graphe implémenté avec les structures `Lis` et `Vect`, et déterminer le chemin permettant de passer d'un col à un autre en minimisant le dénivelé. Les dénivelés seront représentés par des poids positifs lorsqu'il y a un dénivelé positif, négatif sinon. On pourra par exemple se servir de cette page et de Wikipedia.

18. **Cols des Pyrénées.** On voudra représenter les routes entre les cols de la chaîne de montagne des Pyrénées dans un graphe implémenté avec les structures `Mat` et `Brin`, et déterminer le chemin permettant de passer d'un col à un autre en minimisant le dénivelé. Les dénivelés seront représentés par des poids positifs lorsqu'il y a un dénivelé positif, négatif sinon. On pourra par exemple se servir de cette page et de Wikipedia.

19. **Cols des Vosges.** On voudra représenter les routes entre les cols de la chaîne de montagne des Vosges dans un graphe implémenté avec les structures `Vec` et `Brin`, et déterminer le chemin permettant de passer d'un col à un autre en minimisant le dénivelé. Les dénivelés seront représentés par des poids positifs lorsqu'il y a un dénivelé positif, négatif sinon. On pourra par exemple se servir de cette page et de Wikipedia.

20. **Analyse d'un programme 1.** Un programme peut être considéré comme un graphe orienté où chaque fonction est un nœud. Prendre un programme important (plusieurs dizaines de fichiers, plusieurs dizaines de milliers de lignes de code) et l'analyser sous la forme d'un graphe avec les structures `Mat` et `Lis`. On cherchera les composantes connexes et les cycles.

21. **Analyse d'un programme 2.** Un programme peut être considéré comme un graphe orienté où chaque fonction est un nœud. Prendre un programme important (plusieurs dizaines de fichiers, plusieurs dizaines de milliers de lignes de code) et l'analyser sous la forme d'un graphe avec

les structures `Mat` et `Vec`. On cherchera les composantes connexes et les cycles.

D) Compression d'images

Attention, les projets concernant la compression d'images doivent être faits en utilisant uniquement les bibliothèques standard plus OpenGL, GLUT et/ou SDL et GL4D. La base du programme sera celle donnée en cours, ou celle donnée dans le cours de Programmation Graphique. Il n'est donc absolument pas question de réécrire le programme lui-même, ce que vous avez à faire est simplement d'ajouter des fonctions à ce programme.

Dans la suite on tentera de compresser une image. Il faudra le faire, ainsi que la décompression associée, et valider la méthode en termes de :

- temps de calcul, compression et décompression,
- perte en qualité éventuelle,
- ratio de compression.

22. Compression par gestion de surfaces unies. En dessin, par exemple en BD, il est fréquent de trouver de grands à-plats de couleur. Plutôt que de compresser l'image elle-même, pixel par pixel, il faut ici détecter ces taches de couleur, et les traiter individuellement.
Trouver les taches consiste à trouver et numéroter les zones connexes ayant la même couleur.
Une bonne méthode consisterait à commencer par trouver tous les pixels connexes ayant une certaine couleur, ils forment une tache. Puis à réitérer ce processus pour tous les pixels.
On a donc toutes les taches, si elles ont été listées. On peut alors mémoriser l'image comme une liste de zones (données par leur contour et la couleur).
Mettre en place cet algorithme.
Mettre en place l'algorithme de décompression/affichage d'une telle image.
Étudier le gain, en temps et en espace.

23. Par gestion de graphe non orienté. Ce projet représente en fait quatre projets différents. Les projets A, B, C et D, voir plus bas.
Nous considérons ici qu'une image est un graphe où chaque pixel est un noeud lié au plus à ses quatre voisins. Nous rajoutons une contrainte : un pixel p est lié à son voisin v, si et seulement si p et v sont effectivement voisins (au sens des quatre voisins) et ont la "même" couleur.
Étudier ce graphe consiste à le gérer, avec son million de noeuds.
Mettez en place une méthode de recherche des parties connexes du graphe.
Votre méthode doit vous avoir fourni un numéro pour chaque partie connexe.
Il suffit maintenant de parcourir une partie connexe en cherchant les sommets qui n'ont pas quatre successeurs, ce sont les sommets de la frontière de la région. Ces sommets seront conservés dans une liste (ou un vecteur). Il est conseillé de se souvenir aussi des pixels liés ou des pixels non liés (au choix).
La liste de ces listes compose les pixels de bords des regions. Ils organisent donc une partition de l'image. Avec cette liste de listes et la couleur qui y est associée, vous créez une nouvelle façon de stocker l'image.
Il faut maintenant, mettre en place un système de sauvegarde et un système de restauration de l'image.

A) Utilisez `Mat`.
B) Utilisez `Lis`.
C) Utilisez `Vec`.
D) Utilisez `Brin`.

24. Compression d'une image couleur par l'algorithme des quadtrees. Mettre en place la méthode, bien connue pour les images en noir et blanc.
La tester en temps et en taille de fichier sur de nombreuses images.
Si les résultats ne vous semblent pas concluants, tentez d'améliorer la méthode.

25. RLE. Mettre en place et testez le run-length-encoding.
On devra en particulier :

- Tester le RLE à partir des trois plans R, puis G, puis B.
- Tester le RLE en ayant transformé l'image en mode HSV et en séparant les champs de H, de S et de V.
- NB : Écrire dans chaque cas la fonction qui code en mode RLE, la fonction qui fait la sauvegarde et la fonction qui permet d'afficher une image ainsi sauvegardée.

— NB : Essayer votre méthode sur un bon nombre d'images et comparer les résultats obtenus, en particulier par rapport à des méthodes concurrentes (LZW...).

26. **LZW** Programmer l'algorithme LZW, pour la compression et pour la décompression. L'appliquer aux images selon les modalités suivantes :

— Directement au niveau du fichier ppm.

— En séparant le fichier en quatre parties, entête, les octets de rouge, les octets de vert et les octets de bleu. Puis en appliquant LZW sur chacun de ces quatre fichiers.

— En transformant l'image pour passer en mode HSV. Puis en séparant le fichier en quatre parties, entête, les octets de hue, les octets de saturation et les octets de value. Puis en appliquant LZW sur chacun de ces quatre fichiers.

Comparer sur de nombreuses images, les résultats obtenus, en particulier face à des algorithmes concurrents (RLE...).

27. **Par fenêtres.** Le principe est, ici, brutal : il s'agit de parcourir l'image par fenêtres de tailles nxn et conservant uniquement un groupe de quatre pixels pour chaque fenêtre. Choisir, pour chaque fenêtre, efficacement, les couleurs de ces quatre pixels.
Dans ce cas le ratio est connu, c'est 4/(nxn). Par contre ce qui ne l'est pas, c'est la qualité de l'image produite.
Mettre en place un système de mesure de la qualité de l'image (on peut chercher sur le web).
Mettre en place une procédure subjective, questionnaire, de mesure de la qualité de l'image produite.
Discuter les résultats produits.

28. **Compression d'images par ondelettes.** Programmer et utiliser l'algorithme des ondelettes (cf. page Wikipedia) Pour faire de la compression d'images. Il faudra utiliser l'algorithme sur l'image en RGB et sur l'image en HSV.
Compression et décompression seront implémentées et il faudra tester les deux approches sur suffisamment d'images pour discuter les résultats produits.

29. **Par simplification.** Pour toute fenêtre (nxn) de l'image, on trouve une valeur minimale de chacune des couleurs. Puis pour chaque pixel de la fenêtre, la couleur est donnée par un écart sous la forme de quelques chiffres binaires avec la couleur minimale. Par exemple 3 bits pour le vert, 3 pour le rouge et 2 pour le bleu. On a donc, par fenêtre, une couleur puis uniquement un octet pour chaque pixel.

— Mettre en place cette méthode pour compresser l'image.

— Mettre en place cette méthode pour décompresser et afficher l'image.

— Comparer la qualité des images (cf. ci-dessus).

— Améliorer la méthode en donnant aux chiffres binaires utilisés des valeurs autres que directement +1, +2... +8, car l'écart peut être beaucoup plus important.

E) Compression d'images par "color quantization"

Dans la suite on tentera de compresser une image par réduction du nombre de couleurs utilisées. Il faut, évidemment, programmer la compression et la décompression. On pourra appliquer un algorithme de Dithering pour minimiser les erreurs faites en simplifiant le nombre de couleurs.
Il faudra valider la méthode en termes de :

— temps de calcul, compression et décompression,

— perte en qualité éventuelle,

— ratio de compression.

On commencera bien sûr par transformer l'image en passant par une table d'indirection de couleurs (C-LUT).

30. **Halftoning and Dithering.** Dans ce projet, on réduit à, par exemple, quelques niveaux pour chaque composante (R, G et B). Par exemple on pourra utiliser 64 couleurs avec des niveaux simples (0, 85, 170, 255 pour chaque composante). On utilise alors une CLUT de 64 entrées, sur 6 chiffres binaires.

— mettre en place une telle compression/décompression d'image.

— On se rend compte qu'en utilisant une telle méthode, une erreur est faite avec le choix d'une couleur simplifiée au lieu de la couleur sur 256 niveaux initiale. Pour corriger cette erreur, mettez en place une méthode de diffusion d'erreur vue en cours (dithering).

31. **Limiter le nombre de couleurs par la méthode des octrees.**

32. **Limiter le nombre de couleurs par les BSP.**

33. **Limiter le nombre de couleurs par la méthode de k-means.**

34. **Limiter le nombre de couleurs par une adaptation de l'algorithme de Voronoi.**

F) IA et Jeux

35. **IA pour le jeu d'Othello.** Implémenter un jeu d'Othello et une intelligence artificielle jouant selon un algorithme minimax, puis un élagage alpha-beta. Comparer l'efficacité de l'IA à différentes profondeurs de jeu.

36. **IA pour le jeu de Dames.** Implémenter un jeu de Dames et une intelligence artificielle jouant selon un algorithme minimax, puis un élagage alpha-beta. Comparer l'efficacité de l'IA à différentes profondeurs de jeu.

37. **IA pour le jeu de Backgammon.** Implémenter un jeu de Backgammon et une intelligence artificielle jouant selon un algorithme minimax, puis un élagage alpha-beta. Comparer l'efficacité de l'IA à différentes profondeurs de jeu.

38. **IA pour le jeu de Bataille Navale.** Implémenter un jeu de Bataille Navale et une intelligence artificielle jouant selon un algorithme minimax, puis un élagage alpha-beta. Comparer l'efficacité de l'IA à différentes profondeurs de jeu.

39. **IA pour le jeu de Breakthrough.** Implémenter un jeu de Breakthrough et une intelligence artificielle jouant selon un algorithme minimax, puis un élagage alpha-beta. Comparer l'efficacité de l'IA à différentes profondeurs de jeu.

40. **IA pour le jeu Ultimate TicTacToe.** Implémenter un jeu de Ultimate TicTacToe (ou morpion 9×9) et une intelligence artificielle jouant selon un algorithme minimax, puis un élagage alpha-beta. Comparer l'efficacité de l'IA à différentes profondeurs de jeu.

41. **IA pour le jeu d'Awalé.** Implémenter un jeu d'Awalé (ou un autre jeu de type *mancala*) et une intelligence artificielle jouant selon un algorithme minimax, puis un élagage alpha-beta. Comparer l'efficacité de l'IA à différentes profondeurs de jeu.

42. **IA pour le jeu de Nim.** Implémenter un jeu de Nim à plusieurs tas et une intelligence artificielle jouant selon un algorithme minimax, puis un élagage alpha-beta. Comparer l'efficacité de l'IA à différentes profondeurs de jeu.

43. **Résolution d'un Sudoku.** Créer un programme permettant de générer une grille de Sudoku aléatoire, puis un programme permettant de résoudre un Sudoku en parcourant tout l'arbre des possibilités de remplissage des cases vides. Proposer des améliorations de cet algorithme.

44. **Résolution du problème des n reines.** Le but du problème des 8 reines est de placer 8 reines d'un jeu d'échecs sur un échiquier classique sans que les dames puissent se menacer mutuellement, conformément aux règles du jeu d'échecs, c'est-à-dire que les reines ne doivent pas être sur la même ligne, colonne, ou diagonale. Il a été démontré qu'il y a 12 manières possible, à symétrie près, de résoudre ce problème. Créer un programme permettant de résoudre le problème des n reines visant à placer n reines sur un échiquier de taille $n \times n$, en affichant toutes les solutions sous la forme d'un vecteur de positions. Proposer des améliorations de cet algorithme.

45. **Résolution d'un jeu de solitaire.** Créer un programme permettant de résoudre le casse-tête du solitaire en parcourant tout l'arbre des possibilités de mouvement. On implémentera la résolution pour différentes formes et connexités du plateau (européen, anglais en croix, diamant, triangulaire). Proposer des améliorations de cet algorithme.

46. **Résolution d'un jeu de Sokoban.** Créer un programme permettant de jouer au jeu de Sokoban, et de résoudre un niveau en parcourant tout l'arbre des possibilités de mouvement. On représentera les murs par le symbole #, les caisses par des \$, une case finale par $\cdot$, le personnage par un @ et $*$ pour une caisse sur une case finale.

G) Nombres et combinatoire

47. **Nombres de Catalan.** En mathématiques, les nombres de Catalan $C(n)$ sont une suite de nombres entiers utilisés dans divers problèmes en

combinatoire. Ils permettent notamment de dénombrer le nombre d'arbres binaires possédant exactement n nœud internes, ou encore le nombre de manières de parenthéser correctement un mot de longueur n avec autant de parenthèses ouvrantes que fermantes, et une parenthèse ouvrante arrive toujours avant la parenthèse qui la referme. Implémenter plusieurs algorithmes permettant de calculer les nombres $C(n)$, et les comparer en temps d'exécution. Écrire un programme qui affiche toutes les manières de bien parenthéser un mot de longueur n, discuter de son efficacité.

48. **Nombres de Motzkin.** Les nombres de Motzkin $M(n)$ forment une suite de nombres entiers utilisés dans divers problèmes en combinatoire. Ils permettent par exemple de compter le nombre de chemins dans le plan qui vont de $(0, 0)$ vers $(n, 0)$ avec des pas horizontaux, et obliques (vers le nord-est ou le sud-est), appelés *chemins de Motzkin*. Implémenter plusieurs algorithmes permettant de calculer les nombres $M(n)$, et les comparer en temps d'exécution. Écrire un programme qui affiche tous les chemins de Motzkin de longueur n, discuter de son efficacité.

49. **Nombres de Bell.** Les nombres de Bell $B(n)$ forment une suite de nombres entiers correspondant au nombre de manières de partitionner un ensemble à n éléments distincts. Implémenter plusieurs algorithmes permettant de calculer les nombres $B(n)$, et les comparer en temps d'exécution. Écrire un programme qui affiche toutes les partitions d'un ensemble à n éléments, discuter de son efficacité.

H) Fonctions utiles

a) Remplir une grande matrice

Vous trouverez en 1 une fonction de remplissage d'une grande matrice.

b) Créer un graphe non pondéré

Vous trouverez dans la table 2 une fonction de création d'un graphe non pondéré.

c) Créer un graphe pondéré

De la même manière vous trouverez en 3 une fonction de génération d'un graphe pondéré.

```
#include <stdio.h>

#define MAXMAT 1000
typedef float matrice [MAXMAT] [MAXMAT];
struct matrfloat {
  int n;
  matrice m;
} ;
typedef struct matrfloat mat_t;

mat_t creer (int nb) {
  mat_t m;
  int i, j, k, l, z, zi;
  m.n = nb;
  for (i = 0, k=nb; i < nb; i++, k--) {
l = k;
m.m[i][i] = (float) l;
l >>= 1;
for (j = i+1, z=1, zi = 0; j < nb; j++) {
  if (z == zi) {
m.m[i][j] = (float) l;
m.m[j][i] = (float) l;
l >>= 1;
z <<= 1;
zi = 0;
  }
  else {
m.m[i][j] = 0.0;
m.m[j][i] = 0.0;
zi++;
  }
}
  }
  return m;
}
int main () {
  mat_t m;
  int nb;

  nb = 60;
  m = creer(nb);
}
```

TABLE 1 – Remplir une grande matrice

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
typedef short unsigned Shu;
struct graphe {
  int nbs;
  Shu * tab;
} ;
typedef struct graphe graphe;
graphe creegraphe (int nbs) {
  Shu i, j, max, num;
  float v, taux;
  graphe g;
  g.nbs = nbs;
  max = nbs * nbs;
  taux = 25.0;
  num = nbs / 10;
  while (num > 1) {
num /= 5;
taux /= 3.0;
  }
  taux /= 100.0;
  printf("taux %g\n", taux);
  g.tab = (Shu *) malloc (max * sizeof(Shu));
  memset(g.tab, 0, max);
  srand(time(NULL));
  for (num = 0, i = 0; i < nbs; i++)
for (j = 0; j < nbs; j++) {
  v = (float) rand () / RAND_MAX;
  g.tab[num++] = v < taux ? 1 : 0;
}
  return g;
}
```

Table 2 – Graphe non pondéré

```
graphe creegraphe (int nbs) {
  Shu i, j, max, num;
  float v, taux;
  graphe g;
  g.nbs = nbs;
  max = nbs * nbs;
  taux = 25.0;
  num = nbs / 10;
  while (num > 1) {
num /= 5;
taux /= 3.0;
  }
  taux /= 100.0;
  printf("taux %g\n", taux);
  g.tab = (Shu *) malloc (max * sizeof(Shu));
  memset(g.tab, 0, max);
  srand(time(NULL));
  for (num = 0, i = 0; i < nbs; i++)
for (j = 0; j < nbs; j++) {
  v = (float) rand () / RAND_MAX;
  g.tab[num++] = v < taux ? (Shu) (v * 1000.) : 0;
}
  return g;
}
```

Table 3 – Graphe pondéré