Algorithmique Avancée 2024-2025
Projets

A) Règles

Le choix des projets sera fait le mardi 22/10/2024 à 15h00. Vous trouverez ci-dessous 36 propositions de sujets que vous devez vous répartir individuellement.

a) Quoi rendre et quand ?

Vous devrez rendre par mail la version complète de votre projet.
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 devez rendre votre travail :
Avant le mardi 3 décembre, 15h, et le présenter 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

- générer un graphe non pondéré orienté (cf.Table 1),
- générer un graphe non pondéré non orienté (cf.Table 2),
- générer un graphe pondéré (cf.Table 3).

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 données sont les bonnes et aussi de **mesurer** le temps passé, donc avec des données suffisament grandes. 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. **Angel and Morrisson**

C) Graphes

Nous avons vu six structures pour stocker des graphes :

- matrice d'adjacence `Mat`,
- matrice compacte par triplets `Mat3`,
- matrice compacte par vecteur de vecteurs `Matvec`,
- liste d'adjacence `Lis` pour chaque noeud,
- vecteur d'adjacence `Vec` pour chaque noeud,
- 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 adaptée de *creer_graphe*, cf. Table 1, mettre en place les algorithmes de recherche du plus court chemin et de composantes connexes avec `Mat3` et `Brin`.

9. **Composantes connexes 2.** Sur de grands graphes (plus de 10000 noeuds) remplis aléatoirement par une fonction adaptée de *creer_graphe*, cf. Table 1, mettre en place les algorithmes de recherche du plus court chemin et de composantes connexes avec `Matvec` et `Vec`.

10. **Composantes connexes 3.** Sur de grands graphes (plus de 10000 noeuds) remplis aléatoirement par une fonction adaptée de *creer_graphe*, cf. Table 1, 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 des 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 `Mat3` et `Brin`. On considérera une partie connexe du graphe fourni par une fonction adaptée de *creer_graphe*, cf. Table 3,

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 des 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`. On considérera une partie connexe du graphe fourni par une fonction adaptée de *creer_graphe*, cf. Table 3.

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 des 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`. On considérera une partie connexe du graphe fourni par une fonction adaptée de *creer_graphe*, cf. Table 3,

14. **Gestion d'un réseau de transports 1.** On voudra gérer le réseau de transports en commun en métro de la ville de Shenzhen, via un graphe implémenté avec les structures `Lis` et `Matvec`. 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 3.** On voudra gérer le réseau de transports en commun en métro de la ville de Madrid, via un graphe implémenté avec les structures `Mat3` 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.

16. **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 `Matvec` et `Lis`. On cherchera les composantes connexes et les cycles.

17. **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 `Mat3` 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 faut donc :

- écrire une fonction qui compresse l'image et permet de l'enregistrer dans un fichier,
- écrire une fonction qui permet de lire un tel fichier,
- écrire une fonction qui permet de décompresser l'image et de l'afficher sur écran.

Il faut valider la méthode en termes de :

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

Gestion de graphe non orienté. Ces projets utilisent la même base.
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.

18. **Utilisez** `Mat3`.

19. **Utilisez** `Lis`.

20. **Utilisez** `Vec`.

21. **Utilisez** `Brin`.

22. **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...).

23. **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...).

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 faut :

- écrire une fonction qui construit la Table d'Indirection de Couleurs (Color LUT),
- écrire une fonction qui compresse l'image et permet de l'enregistrer dans un fichier,
- écrire une fonction qui permet de lire un tel fichier,
- écrire une fonction qui permet de décompresser l'image et de l'afficher sur écran.

Il faudra valider la méthode en termes de :

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

– ratio de compression.

24. **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).

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

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

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

F) IA et Jeux

28. **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.

29. **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.

30. **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.

31. **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.

32. **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.

33. **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.

G) Nombres et combinatoire

34. **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.

35. **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.

36. **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) Créer un graphe non pondéré

Vous trouverez dans la table 1 une fonction de création d'un graphe orienté non pondéré.

b) Créer un graphe non pondéré

Vous trouverez dans la table 2 une fonction de création d'un graphe non pondéré non orienté (donc avec une symétrie).

c) Créer un graphe pondéré

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

```
#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 1 – Graphe non pondéré orienté

```
#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 = 12.5;
  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 <= i; j++) {
      v = (float) rand () / RAND_MAX;
      g.tab[num++] = v < taux ? 1 : 0;
      g.tab[max - num] = v < taux ? 1 : 0;
    }
  return g;
}
```

TABLE 2 – Graphe non pondéré non orienté (symétrique)

```
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