

Serpents et échelles

Ce TP est en C. L'objectif est de :

- Réviser le C.
- Écrire un parcours de graphe.
- Trouver une formule pour le nombre de chemins d'un noeud à un autre¹.

Conditions de rendu

Vous devez déposer le zip créé par `make zip` sur le lien suivant avant dimanche 20 avril 2025, 20h00 (heure de Paris) :

<https://nuage04.apps.education.fr/index.php/s/SGWnzdBwx2XZ72F>

Fichiers fournis

- Un `Makefile` comme dans les TP précédents. Vous pouvez toucher à la ligne qui définit `CFLAGS` : ce sont les options de compilation. Vous pouvez par exemple y rajouter (ou enlever) des sanitizers !
- `file` une librairie de files de longueur bornée.
- `plateaux` une implémentation des plateaux : vous y trouverez deux exemples de plateaux, un générateur de plateaux aléatoires, ainsi que quelques fonctions utiles pour manipuler les plateaux.
- `solver.h` : interface qui décrit `solver.c`. C'est ce dernier fichier que vous devez coder !
- `main.c` : votre main à vous. Faites-y ce que vous voulez, je n'y toucherai pas !
- `testeur.o` : une version compilée du testeur. *Elle ne marche que pour les processeurs Intel/AMD. Si vous êtes sous ARM... demandez à des camarades de tester le code sur leur machine; ou si vous n'avez pas de camarades demandez-moi.*

I - Serpents et échelles

1) Rappels de l'école primaire

On s'intéresse au jeu Serpents et Échelles. C'est un jeu de hasard, qui consiste à faire avancer un pion en lançant un D6 et espérer arriver à la fin du plateau en premier.

100	99	98	97	96	95	94	93	92	91
81	82	83	84	85	86	87	88	89	90
80	79	78	77	76	75	74	73	72	71
61	62	63	64	65	66	67	68	69	70
60	59	58	57	56	55	54	53	52	51
41	42	43	44	45	46	47	48	49	50
40	39	38	37	36	35	34	33	32	31
21	22	23	24	25	26	27	28	29	30
20	19	18	17	16	15	14	13	12	11
1	2	3	4	5	6	7	8	9	10

FIGURE V.1 – Un plateau de serpents et échelles (src : J. Erikson)

1. Dans un graphe non-pondéré. Quand on voudra généraliser cette formule aux graphes pondérés, on devra la modifier intelligemment et on obtiendra l'algorithme de Floyd-Warshall.

Plus précisément :

- Le plateau est une suite de case numérotées. Quand on est sur la case i , selon le résultat du dé on peut aller en $i + 1, i + 2, \dots, i + 6$. Si une de ces cases n'existe pas (par exemple si elle se trouve *après* la fin du plateau), on ne peut pas y aller et le mouvement est annulé.
- Bad news! Certaines cases sont des serpents. Quand on s'arrête au « sommet » d'un serpent, le serpent nous fait glisser jusqu'à une case d'avant.
- Good news! Certaines cases sont des échelles. Quand on arrive en bas d'une échelle, on monte à son sommet.
- On va nommer les deux cas précédents des « téléportations ». On n'enchaîne pas les téléportations : par exemple, si monter une échelle nous fait arriver pile sur la queue d'un serpent, on ne descend pas le serpent.
Cette règle permet d'éviter les boucles infinies d'un pion qui serait coincé dans un circuit échelle -> serpent -> échelle -> serpent -> ...

2) Le hasard? Quel hasard?

Pour simplifier, on enlève le hasard : vous *choisissez* si le pion avance de 1, 2,... ou 6 à chaque coup.

3) Adaptation de l'école primaire en C

Pour représenter un plateau à `nb_cases`, nous allons indiquer ses cases de 0 à `nb_cases-1` inclus. Dans chaque case, on stocke l'indice de la case vers laquelle elle nous téléporte (le haut de l'échelle / le bas du serpent). Si la case ne téléporte pas, on stocke son propre indice dedans.

11	10	9	8	7	6
0	1	2	3	4	5

(a) Un plateau à 12 cases

plateau	0	1	2	3	8	5	6	7	8	9	2	11
Indices	0	1	2	3	4	5	6	7	8	9	10	11

(b) Encodage en tableau

FIGURE V.2 – La représentation d'un plateau

On utilise le type suivant pour un plateau :

```

16 struct plateau_s {
17     int* cases;
18     int nb_cases;
19 };
20 typedef struct plateau_s plateau;

```



Remarque. Les deux plateaux ci-dessous sont fournis dans la librairie `plateaux` :

- Le plateau à 100 cases est celui renvoyé par `plateau_enonce` (à ceci près qu'on lui rajoute une case 0 et enlève la case 100).
- Le plateau à 12 cases est celui renvoyé par `plateau_mini`.

Un indice du plateau est appelée une **position**. On dit qu'une position est **valide** si il s'agit d'un indice valide (donc ni négatif ni trop grand).

0. Lisez l'interface `plateaux.h`. Vous pouvez utiliser autant que vous le souhaitez toutes les fonctions qui s'y trouvent!

II - À vous de jouer

1) À faire

On appelle **chemin** un enchainement de déplacements (1 déplacement = avancer le pion de 1 à 6 puis appliquer l'éventuelle téléportation) sur le plateau ; la **longueur** d'un chemin est son nombre de déplacements.

Un **chemin gagnant** est un chemin qui part de la case 0 et arrive à la dernière case.

1. Écrire la fonction `soluble` décrite dans `solver.h`
2. Écrire la fonction `nb_path` décrite dans `solver.h`

2) À propos de la compilation et du testeur

Comme mentionné précédemment, vous pouvez toucher à la variable `CFLAGS` du `Makefile`. Ce sont les options de compilation :

- `--std=gnu11` indique la version précise des librairies à utiliser. Ne modifiez pas cette version.
- `-Wall -Wextra` : vous connaissez. Je ne vois pas pourquoi vous voudriez les enlever.
- `-g -fsanitize=undefined,address` : les sanitizers, qui vous indiquent des comportements « anormaux » de votre code comme un dépassement de capacité, la lecture mémoire d'une mauvaise adresse, l'oubli d'un free... Les sanitizers sont très utiles pour déboguer, mais ils ralentissent votre code.

Les tests du testeur sont faits sur les plateaux mini, enonce... et sur des plateau à au moins 10^6 cases. Sur `soluble` cela ne devrait pas être gênant ; mais sur `nb_path` le temps d'exécution des derniers tests peut dépasser la limite autorisée (1 minute). Pour accélérer votre code, vous pouvez alors enlever les sanitizers. Vous pouvez même mettre `-O2` à la place : c'est une option de compilation qui demande à gcc d'optimiser votre code (ici optimisation de niveau 2).

Pour indication, voici mes temps d'exécution sur ma machine (compilation comprise) :

Options de compilation	Temps d'exécution de <code>make test</code>
Sanitizers (et pas d'optimisation)	~135s, et les deux derniers tests dépassent le temps limite (1min)
Sans sanitizers ni optimisation	~32s
Optimisation <code>-O1</code> (et sans sanitizers)	~16s
Optimisation <code>-O2</code> (et sans sanitizers)	~16s

TABLE V.1 – Durées indicatives d'exécution de `make test` sur un processeur à 3GHz

3) Indications

Cf page suivante. Je vous encourage à d'abord chercher par vous même, puis si vous séchez regarder les indications, puis si vous séchez toujours m'envoyer un mail.

III - Indications

1) Pour soluble

Un graphe se cache derrière le problème. Lequel (que sont ses noeuds? que sont les voisins d'un noeud?)?²

Il s'agit ensuite de trouver des distances dans le graphe. Vous pouvez soit calculer un tableau de distances, soit calculer un tableau de parentés et en déduire la distance. Le tout peut s'effectuer en $\Theta(\text{nb_cases} + 6)\text{nb_cases}$.

2) Pour nb_path

Noter $\text{nb}(\text{len}, \text{pos})$ le nombre de chemins de longueur len qui partent de la position pos et terminent sur la dernière case. Trouver une formule de récurrence qui lit les $\text{nb}(\text{len}, _)$ et les $\text{nb}(\text{len} - 1, _)$. Justifier que $\text{nb}(0, \text{pos}) = \mathbb{1}_{\text{pos}}$ est la dernière case.

Remarquer que des appels récursifs ont lieu plusieurs fois. En déduire une écriture par programmation dynamique; qui devrait avoir une complexité en $\Theta(\text{nb_cases} \times \text{max_path_len})$.

2. Demande-moi si même après avoir cherché vous ne trouvez pas.