Techniques Algorithmiques et Programmation

TP noté – 2h10 (+43min si tiers-temps)

Recherche d'un maximum local en 2D

Jeudi 11 avril 2024

Consignes

Vous avez le droit de consulter une seule ressource sur Internet : les notes de cours. Vous pouvez utiliser vos notes personnelles de cours et de TD, ainsi que vos programmes réalisés en TP. Vous pouvez utiliser du brouillon. Les outils basés sur l'intelligence artificielle (comme *ChatGPT* ou *Copilot*) ainsi que les forums ou dépôts (comme *Discord* ou *GitHub*) ne font pas partie des documents autorisés. Ils sont **interdits**. C'est une épreuve **individuelle**, vous n'avez pas le droit de communiquer avec vos voisins, proches ou lointains. Tout manquement à ces consignes sera considéré comme une fraude pouvant donner lieu à des **sanctions**[1].

La notation prendra principalement en compte la correction de votre code, c'est-à-dire, s'il passe les tests avec succès, et de façon moindre :

- sa lisibilité (présentation et commentaires),
- ses performances, que vous pouvez tester avec la commande `time`,
- l'absence de fuite mémoire, que vous pouvez détecter grâce à `valgrind`.

Votre code doit pouvoir être compilé sans erreur ni avertissement du compilateur.

Ce que vous devez télécharger et déposer sur Moodle

1. Téléchargez et décompressez l'archive `tp.tgz` depuis la page du cours sous Moodle.
2. Éditez **uniquement** le fichier `tp2.c`.
3. En fin d'épreuve, déposez le fichier `tp2.c` sous Moodle, et uniquement lui, non compressé.

1 Objectif du TP noté

Il s'agit de déterminer un maximum **local** dans un tableau à deux dimensions. Plus précisément, on considère une grille `G` **carrée** `n` × `n` dont chaque valeur est de type `double` et comprise dans l'intervalle $[0, 1]$. On suppose de plus que le bord de la grille `G` (c'est-à-dire l'ensemble des positions ayant une coordonnée à `0` ou à `n-1`) ne contient que la valeur 0.0, ce qui permet de chercher un maximum local qui **n'est pas** sur le bord.

Soient `i` et `j` deux indices avec `1` ⩽ `i` ⩽ `n-2` et `1` ⩽ `j` ⩽ `n-2`. On dit que `G[i][j]` est un *maximum local* si `G[i][j]` est supérieur ou égal aux **huit** valeurs voisines, c'est-à-dire `G[i][j]` ⩾ `G[i1][j1]` pour tous `i1`, `j1` tels que `i-1` ⩽ `i1` ⩽ `i+1` et `j-1` ⩽ `j1` ⩽ `j+1`. La *position* de ce maximum local est la paire `(i,j)`, où `i` représente le numéro de ligne et `j` le numéro de colonne, tous deux entre `1` et `n-2` inclus. Dans l'exemple ci-après où `n` vaut `5`, il y a deux maximums locaux : `G[2][1] = 0.4` et `G[1][3] = 0.8`. Votre programme lancé sur cet exemple devra trouver l'un de ces deux maximums locaux (on demande d'en trouver un seul, n'importe lequel).

i \ j	0	1	2	3	4
0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.3	0.4	0.8	0.0
2	0.0	0.4	0.2	0.0	0.0
3	0.0	0.1	0.0	0.1	0.0
4	0.0	0.0	0.0	0.0	0.0

1. Si cela n'est pas déjà fait, vous pourrez consulter plus tard la charte des examens et son pdf complet de mars 2024.

Vous avez trois algorithmes à programmer, correspondant à trois méthodes différentes pour trouver un maximum local qui n'est pas sur le bord. L'algorithme **naïf** consiste à parcourir séquentiellement `G` pour trouver un maximum local. L'algorithme du **gradient** part de la position médiane `(n/2,n/2)`, puis teste si la position courante `(i,j)` contient un maximum local. Si cela n'est pas le cas, on déplace la position courante vers une position voisine de valeur strictement supérieure à `G[i][j]` et on recommence depuis cette nouvelle position, jusqu'à atteindre un maximum local. Enfin, l'algorithme **récursif** est basé sur une stratégie diviser-pour-régner. Il consiste à découper la grille en quatre sous-grilles carrées et à continuer récursivement dans une seule d'entre elles. Cette méthode garantit une complexité en $O(n)$, alors que les algorithmes naïf et du gradient ne peuvent faire mieux que $\Omega(n^2)$ dans le pire des cas.

2 Structure du code

Le fichier à éditer et à remettre à la fin du TP est `tp2.c` et seulement lui. Vous n'avez pas à modifier les autres fichiers. Vous pouvez consulter et utiliser les types et routines présentés dans `tp2.h` et `tp2-tools.c`. Votre programme doit, s'il est exécuté sur la ligne de commande avec un nom de fichier test, permettre d'appliquer un des algorithmes. Le correcteur doit pouvoir tester vos algorithmes en jouant ses propres fichiers de tests. La gestion de la ligne de commande (chargement des fichiers, affichage des résultats, génération de fichiers de tests, *etc.*) est déjà programmée dans la fonction `main()`. Vous n'avez donc à programmer que les trois algorithmes demandés. Vous pouvez programmer d'éventuelles fonctions intermédiaires, si besoin.

Un fichier `Makefile` est fourni, permettant de compiler avec la commande `make tp2` et de nettoyer avec `make clean`. Ensuite, `tp2` peut être exécuté pour générer des grilles, avec une commande de la forme `./tp2 [gen0|gen1|gen2] [nombre entier]`, ou pour tester un des algorithmes, avec une commande de la forme `./tp2 [naif|grad|rec] [fichier]`.

Voici une exécution sur le fichier `g0.txt` (fourni) correspondant à la grille `5×5` de l'exemple précédent. Il montre les 9 cases de la grille centrées sur le maximum local détecté par l'algorithme. On visualise bien que la case centrale est un maximum local (notez que l'algorithme aurait pu détecter l'autre maximum local en parcourant la grille différemment).

```
./tp2 naif g0.txt
loading time=0"0000
n=5
                 j=3
    0.00000000  0.00000000  0.00000000
i=1 0.40000000  0.80000000  0.00000000
    0.20000000  0.00000000  0.00000000

computing time=0"0000
```

Trois générateurs de grilles aléatoires sont aussi pré-programmés : `gen0`, `gen1` et `gen2`. Il vous est fortement conseillé de tester de nombreux exemples, en particulier pour l'algorithme récursif, le plus délicat à mettre au point. Par exemple, pour tester sur une grille de taille 8×8, utilisez des commandes du type :

```
./tp2 gen1 8 > g.txt
./tp2 naif g.txt
./tp2 grad g.txt
./tp2 rec g.txt
```

Pour mesurer la vitesse des algorithmes, il faut prendre des exemples significatifs avec `n=3000`, voire `n=5000`. Comptez 5s à 10s pour le temps de chargement de tels fichiers. Le générateur `gen0` produit une grille où chaque case contient une valeur aléatoire entre 0 et 1, indépendante des autres. Utilisez `gen1` pour générer des instances difficiles pour l'algorithme naïf et `gen2` pour générer des instances difficiles pour l'algorithme du gradient. Si vous programmez correctement l'algorithme récursif, il doit être difficile de le mettre en défaut.

3 Implémentation des trois algorithmes

Dans la suite on notera `G[A..B]` la sous-grille carrée de `G` où `A` est la position du coin en haut à gauche et où `B` est la position du coin en bas à droite. Les coins `A` et `B` sont inclus dans la sous-grille. La grille complète `G` correspond donc à la sous-grille `G[A..B]` avec `A=(0, 0)` et `B=(n-1, n-1)`.

3.1 Algorithme naïf

L'algorithme **naïf** consiste à parcourir séquentiellement `G` pour trouver un maximum local.

Exercice 1 – Fonction `voisin_max`

Écrivez la fonction `position voisin_max(grid G, position p)` qui renvoie `p` si `p` est la position d'un maximum local dans `G`, et qui sinon, renvoie un voisin `q` de `p` telle que la valeur dans `G` en `q` est strictement plus grande que celle en `p`. Ainsi, `voisin_max(G, p)` vaut `p` si et seulement s'il y a un maximum local en `p`.

Exercice 2 – Algorithme naïf

On suppose qu'il existe un maximum local de `G` situé dans une sous-grille carrée `G[A..B]` de taille $k \times k$, où ni la position `A`, ni la position `B` n'est sur le bord de `G`. Écrivez la fonction `position algo_naif(grid G, position A, position B)` qui renvoie la position d'un maximum local de `G` situé dans la sous-grille `G[A..B]`, en parcourant toutes les positions de `G[A..B]`.

3.2 Algorithme du gradient

L'algorithme du **gradient** cherche un maximum local directement dans `G`. Il part de la position médiane $(\texttt{i}, \texttt{j}) = (\lfloor n/2 \rfloor, \lfloor n/2 \rfloor)$, puis teste si la position courante `(i,j)` est un maximum local. Si ce n'est pas le cas, on déplace la position courante vers une position voisine de valeur strictement supérieure à `G[i][j]` et on recommence depuis cette nouvelle position. Remarquez que le chemin produit par l'algorithme du gradient ne peut traverser le bord (à cause de la valeur `0.0` sur le bord) ni revenir sur lui-même à cause de la décroissance des valeurs rencontrées. Il converge donc vers un maximum local.

Exercice 3 – Algorithme du gradient

Écrivez la fonction `position algo_grad(grid G, int n)` qui renvoie la position d'un maximum local de `G` de taille $n \times n$, en utilisant l'algorithme du gradient.

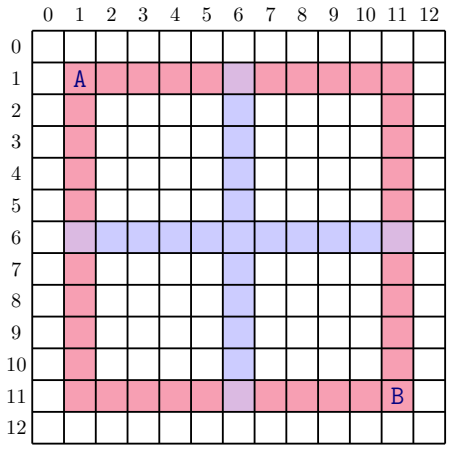
3.3 Algorithme récursif

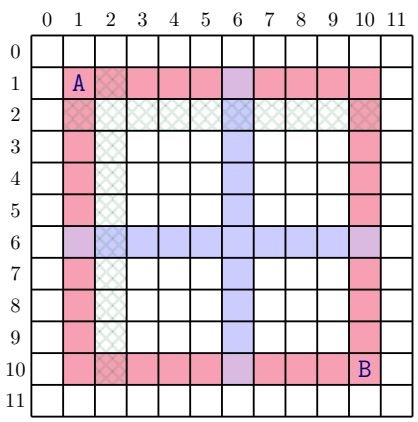
Pour une sous-grille carrée `G[A..B]` de taille $k \times k$ dans `G`, on définit plusieurs zones :

- Le *bord* de `G[A..B]` est l'ensemble des positions `(i,j)` de `G[A..B]` situées sur la même ligne ou colonne que `A` ou `B`. Sur la figure ci-dessous, le bord est indiqué en rose.
- La *croix médiane* de `G[A..B]`, indiquée en bleu, est l'ensemble des positions de `G[A..B]` situées sur la ligne ou la colonne médiane entre `A` et `B`, *i.e.*, les positions (i, j) avec $i = \texttt{A.i} + \lfloor k/2 \rfloor$ ou $j = \texttt{A.j} + \lfloor k/2 \rfloor$.
- Enfin, seulement dans le cas où k est pair (figure de droite pour $k = 10$, zone hachurée en vert), l'*extension du bord* contient les positions (i, j) de `G[A..B]` avec $i = \texttt{A.i} + 1$ ou $j = \texttt{A.j} + 1$.

On appelle *cadre* de `G[A..B]` l'ensemble des positions qui sont soit sur le bord, soit sur la croix médiane, soit (si k est pair) sur l'extension du bord.

Si on supprime le cadre de `G[A..B]`, on obtient quatre sous-grilles carrées appelées *quartiers*. L'intérêt de l'extension du bord, quand k est pair, est de conserver quatre quartiers **carrés**. Ils sont de taille 4×4 sur la figure de gauche, et de taille 3×3 sur celle de droite.



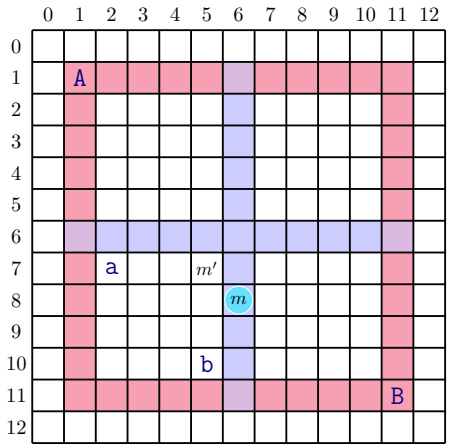


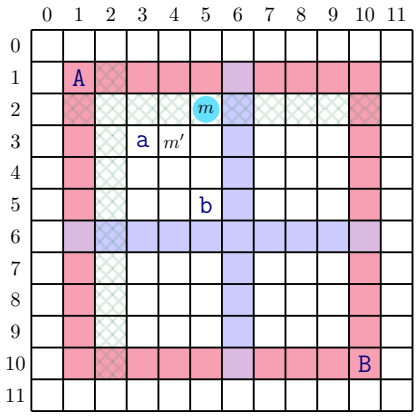
Exercice 4 – Valeur maximale sur les parties colorées

Écrivez une fonction `position position_max_cadre(grid G, position A, position B)` qui renvoie une position de valeur maximale parmi les positions se trouvant sur le cadre de `G[A..B]`. Votre algorithme ne doit parcourir que les cases du cadre, et **pas** l'ensemble des cases de `G[A..B]`.

Soit m une valeur du cadre de `G[A..B]` maximale parmi les positions du cadre. Deux cas sont possibles :

- Soit m est un maximum local de `G`, et l'algorithme renvoie cette position.
- Soit on peut trouver $m' > m$ dans une position voisine de celle de m, située dans l'un des 4 quartiers. Le quartier contenant m' est de la forme `G[a..b]` où `a` et `b` sont ses deux coins. On peut alors montrer que `G[a..b]` contient un maximum local de `G`. L'algorithme se rappelle donc récursivement sur la sous-grille `G[a..b]`. Les deux figures ci-dessous illustrent ce second cas (dans les cas k impair et k pair).





Le principe de l'algorithme est donc le suivant :

Algorithme **algo_rec**(G, A, B)

Entrée : Une grille G et deux positions A et B délimitant la sous-grille $G[A..B]$.
Sortie : La position d'un maximum local de $G[A..B]$.

1. Si la grille $G[A..B]$ est de taille 4 ou moins, renvoyer **algo_naif**(G, A, B).
2. Calculer le maximum m du cadre de $G[A..B]$.
3. Si c'est un maximum local de G, renvoyer la position de m.
4. Sinon, déterminer les coins a, b du quartier contenant un élément strictement plus grand que m.
5. Renvoyer **algo_rec**(G, a, b).

Exercice 5 – Algorithme récursif

Écrivez la fonction `position algo_rec(grid G, position A, position B)` qui renvoie la position d'un maximum local dans la sous-grille `G[A..B]`, en utilisant l'algorithme récursif décrit ci-dessus.