Techniques Algorithmiques et Programmation

TP noté – 2h10 (+43' si 1/3-temps)

Paire de points les plus proches

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 également vous servir de brouillon. Les outils ayant recours à l'intelligence artificielle (comme *ChatGPT* ou *Copilot*) ainsi que les forums ou dépôts (comme *Discord* ou *GitHub*) ne font pas partie des documents autorisés et sont par conséquent interdits. C'est une épreuve **individuelle**, vous n'avez pas le droit de communiquer avec vos voisins, proches ou lointains. Nous vous rappelons que 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.

Objectif du TP noté

Il s'agit de coder l'algorithme probabiliste vu en TD et permettant de calculer la paire de points les plus proches, en temps linéaire en moyenne.

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 `tp.c`.
3. En fin d'épreuve, déposez le fichier `tp.c` sous Moodle, et uniquement lui, non compressé.

Description de l'algorithme

L'algorithme prend en entrée une suite de points P du plan et retourne la paire de points les plus proches, selon la distance euclidienne.

Son principe est de visiter les points dans l'ordre $p_0, p_1, \ldots, p_t, \ldots, p_{n-1}$, et de vérifier à chaque étape t si p_t est à une distance $< d_{\min}$ d'un des points déjà visités. Ici, $d_{\min}$ est la distance des deux plus proches points visités jusqu'alors. Ainsi, initialement, $d_{\min} := d(p_0, p_1)$. Si on réussit à visiter tous les points sans trouver de meilleure paire, c'est que la paire réalisant $d_{\min}$ est la paire recherchée. S'il y a un échec à l'étape t, c'est-à-dire s'il existe un point $q \in \{p_0, \ldots, p_{t-1}\}$ tel que $d(p_t, q) < d_{\min}$, alors on met à jour la meilleure paire trouvée ainsi que $d_{\min}$, puis on recommence la visite de chaque point, toujours dans l'ordre $p_0, p_1, \ldots, p_{n-1}$, jusqu'à les parcourir tous sans aucun échec.

Les performances dépendent grandement de l'ordre des visites des points, mais aussi de l'efficacité à détecter les échecs. On a vu en TD qu'un simple ordre aléatoire uniforme sur les points produira un échec à l'étape t avec une probabilité $O(1/t)$.

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

La recherche d'un point q déjà visité proche de p_t est réalisée efficacement par l'usage d'une table de hachage, notée T_δ, où δ est un paramètre réel positif qui va dépendre de la valeur courante $d_{\min}$. La table T_δ permet simplement de stocker la liste des points appartenant à une *cellule* d'indice (i, j), la région carrée du plan de coté δ correspondant à $[i\delta, (i+1)\delta[\times [j\delta, (j+1)\delta[$. Dit autrement, à l'étape t, $T_\delta[(i, j)]$ contient tous les points q d'indice $< t$ appartenant à la cellule (i, j).

Ci-dessous figure le pseudo-code de l'algorithme, où R est la paire recherchée et où $x(p_t), y(p_t)$ sont les coordonnées du point p_t.

Algorithme pppp_random(P)

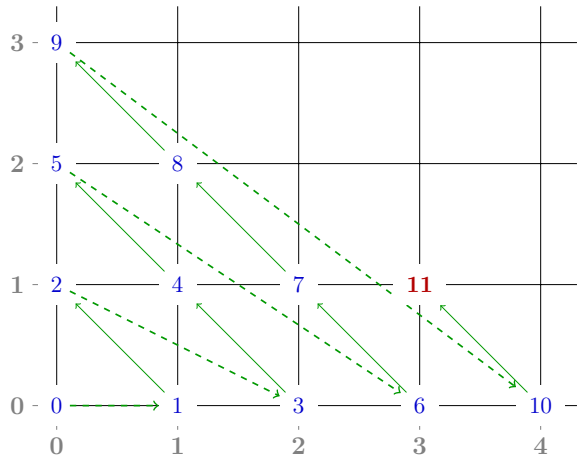
Entrée : Une suite $P = (p_0, p_1, \ldots, p_{n-1})$ de $n \geqslant 2$ points du plan.
Sortie : La paire de points les plus proches, selon la distance euclidienne.

1. Poser $d_{\min} := d(p_0, p_1)$ et $R := \{p_0, p_1\}$.
2. Créer une table de hachage T_δ de cellules de paramètre $\delta := d_{\min}/\sqrt{2}$.
3. Pour chaque point $p_t \in P$ dans l'ordre $t := 0, 1, \ldots, n-1$:
   (a) Soit $(i, j) := (\lfloor x(p_t)/\delta \rfloor, \lfloor y(p_t)/\delta \rfloor)$ la cellule du point p_t.
   (b) Chercher dans la table T_δ le point q qui, parmi les cellules "voisines" de (i, j), soit le plus proche de p_t.
   (c) Si q existe[2] et si $d(p_t, q) < d_{\min}$, poser $d_{\min} := d(p_t, q)$, $R := \{p_t, q\}$ et aller en (2).
   (d) Ajouter p_t à $T_\delta[(i, j)]$.
4. Renvoyer R.

Plus en détails

En TD nous avons vu qu'à chaque instant de l'algorithme, $T_\delta[(i, j)]$ contient au plus un seul point. On a aussi montré que les cellules "voisines" de (i, j) comme écrit en (3b) sont simplement les 25 cellules centrées autour de (i, j), c'est-à-dire les cellules $(i \pm a, j \pm b)$ avec $a, b \in \{0, 1, 2\}$. En fait, 4 de ces cellules sont inutiles.

Pour implémenter T_δ on aura besoin de coder chaque paire d'entiers $(i, j) \in \mathbb{N}^2$ vers un entier unique noté $\mathsf{map}(i, j)$. En représentant chaque paire (i, j) par un point d'une grille, on définit $\mathsf{map}(i, j)$ comme la somme des longueurs des diagonales du type $(t, 0) \to (0, t)$, partant de $(0, 0)$ et jusqu'à atteindre le point (i, j) sans le compter (voir la figure). Ici, la longueur d'une diagonale est le nombre de points de la grille qu'elle contient. Dans l'exemple ci-dessous, $\mathsf{map}(3, 1) = 11$ car il y a exactement 11 points sur les diagonales situées avant $(3, 1)$: les points indiqués en bleu sur la figure.



On peut montrer que $\mathsf{map}(i, j) = j + \sum_{t=1}^{i+j} t$. Il faudra bien sûr implémenter cette fonction efficacement, par une formule close par exemple.

2. Toutes les cellules "voisines" de (i, j) pourraient être vides.

Table de hachage

Il vous est fourni une implémentation de table de hachage, voir `htable.h`. Cette structure de données permet d'écrire des paires `(key,value)` et aussi d'extraire `value` pour une `key` donnée. Dans notre implémentation, `key` est un `int` et `value` est un `void*`. La compréhension des détails de cette implémentation n'est pas nécessaire pour faire le TP.

Pour pouvoir utiliser cette table dans pppp_random, il faudra l'avoir créée avec `T = ht_create()`. Puis pour réaliser l'opération $T_\delta[(i,j)] := q$ par exemple, soit l'insertion du point q dans la cellule (i,j), il faudra :

(1) calculer l'entier `h = map(i, j)`, puis

(2) écrire la paire `(h, &q)` dans `T` grâce à `ht_write(T, h, &q)`.

La lecture de $T_\delta[(i,j)]$ se réalise similairement avec `ht_read(T, map(i, j))`.

Fonctions à compléter

Comme indiqué plus haut, vous n'avez à éditer que le fichier `tp.c`. La liste des fonctions que vous avez à réaliser, ainsi que les détails des implémentations, sont précisés dans le source `tp.c`.

Compilation et tests

Utilisez `make` pour compiler. Veillez à développer un code se compilant sans erreur ni avertissement.

Pour tester vos algorithmes, le programme se lance par `./main [n]` où `n` est le nombre de points à générer. Pour tester l'algorithme naïf, commencez avec peu de points (par exemple, entre 3 et 10 points au départ). Vous pouvez alors déplacer les points avec la souris : le programme met à jour la paire de points trouvée par votre algorithme. Pour tester les algorithmes `pppp_random` et `pppp_random2`, comparez les résultats avec l'algorithme naïf. La différence de temps d'exécution entre l'algorithme naïf et l'algorithme `pppp_random` n'apparaît que pour de grandes valeurs de `n` (typiquement 10 000 ou plus). La touche `'s'` permet de modifier la taille des points pour une meilleure lisibilité.