TP #7 & #9 – APL2

9 et 16 janvier 2019
Erwan Kerrien

Rappel : si vous avez des questions sur ce TP ou sur le cours, n'hésitez pas à m'envoyer un mail à Erwan.Kerrien@inria.fr (je consulte plus rarement mon mail Erwan.Kerrien@univ-lorraine.fr).

Ce TP est prévu sur 4h, et est donc à faire sur les séances des 9 et 16 janvier. La partie 2 concernant les tris et la lecture des points dans un fichier est à me rendre à l'issue du TP du 9 janvier, soit pour le 10 janvier 8h. La partie 3 concernant l'implantation du k-d tree est à me rendre à l'issue du TP du 16 janvier, soit le 17 janvier 8h.

Tout ce qui apparaît sous une section **Exercice** est à me rendre (voir pp. 4, 6 et 7).

Vous pouvez, si vous le souhaitez et si vous le jugez utile, m'envoyer un texte de commentaires. Ne pas le faire ne vous exposera à aucun retrait de points. En revanche, ne pas commenter vos codes d'une manière ou d'une autre pourrait le faire.

Les fichiers de réponse (code source, texte...) sont à déposer sur arche.

1 Objectifs du TP

Ce TP a pour objectif d'implémenter une structure de partitionnement de l'espace appelée *k-d tree*. Même si cette structure peut s'étendre à k dimensions (d'où le *k-d* dans le nom), on se restreindra ici à 2 dimensions, c'est-à-dire au plan.

Comme nous l'avons vu en cours, un arbre binaire de recherche permet d'accélérer les recherches de valeurs dans un ensemble (typiquement grand). Le fait que les données soient organisées, c'est-à-dire triées, permet d'éviter la recherche exhaustive et réduit ainsi la complexité qui devient sous-linéaire, et plus précisément logarithmique.

Ce genre de structure permet aussi d'accélérer les recherches, dans un ensemble, de la valeur la plus proche d'une valeur donnée. C'est typiquement le genre de requête qu'on va faire à de très nombreuses reprises pour la gestion des collisions dans un monde 3D où les objets sont représentés par des surfaces triangulées : pour savoir si un point va rentrer en collision avec un objet du monde, on va chercher le sommet le plus proche de ce point, parmi tous les sommets de tous les objets du monde. On ne peut se permettre une recherche exhaustive et on va donc utiliser ce genre de structure arborescente pour organiser l'espace et accélérer les requêtes de voisinage.

Les k-d trees en fournissent un exemple, et nous allons en voir une première implantation dans un monde 2D.

Nous avons un ensemble de N points donnés dans le plan (en 2D), stockés dans un tableau P. Chaque point est identifié par ses deux coordonnées horizontale x et verticale y.

Le k-d tree organise cet ensemble de points sous forme d'un arbre binaire défini de manière récursive :

- si l'ensemble ne contient aucun point, le k-d tree est vide
- sinon, on prend une direction (horizontale ou verticale) et on trouve le point médian suivant cette direction : par exemple, si la direction est la direction horizontale, ce sera le point de l'ensemble tel que la moitié des points est à sa gauche et l'autre moitié est à sa droite (si c'est la direction verticale, la moitié est en en-dessous, et l'autre moitié au-dessus). Le nœud contient le point médian, et on place deux-sous-arbres : le sous-arbre gauche est le k-d tree du sous-ensemble des points à gauche (respectivement au-dessous), et le sous-arbre droit est le k-d tree du sous-ensemble des points à droite (respectivement au-dessus). Ces deux k-d trees sont formés avec la direction orthogonale (on passe de l'horizontal au vertical et vice-versa).
- on débute avec l'ensemble des points au complet et la direction horizontale.
- À une étape donnée, si le nombre de points est pair, le sous-arbre gauche traitera un point de plus que le sous-arbre droit.

Afin de fixer les idées, prenons un exemple. On considère l'ensemble des 13 points :

$$P = [(0.9, 5.3), (2.3, 2.3), (4.3, 4.9), (1.7, 6.0), (2.5, 7.6), (6.0, 8.4), (6.4, 5.3), (8.5, 5.9), (9.0, 2.2), (9.6, 4.9), (7.9, 6.5), (6.4, 7.3), (9.9, 8.2)]$$

Ces points sont représentés sur la figure 1.

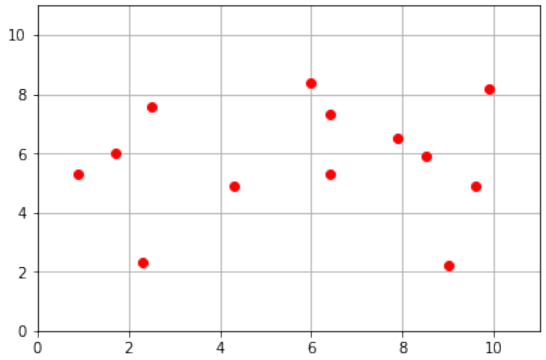


FIGURE 1 – Ensemble de points de départ

On va donc construire le premier nœud de l'arbre. Pour ce faire on choisit la direction horizontale et on ordonne les points selon les x croissants, ce qui donne :

$$P = [(0.9, 5.3), (1.7, 6.0), (2.3, 2.3), (2.5, 7.6), (4.3, 4.9), (6.0, 8.4), (6.4, 5.3), (6.4, 7.3), (7.9, 6.5), (8.5, 5.9), (9.0, 2.2), (9.6, 4.9), (9.9, 8.2)]$$

Le point médian est celui qui sépare cet ensemble en deux parties égales, c'est-à-dire le point d'indice 6 (en commençant à 0) : (6.4, 5.3). Cette séparation se matérialise par la ligne verticale verte de la figure 2. On stocke dans ce premier nœud

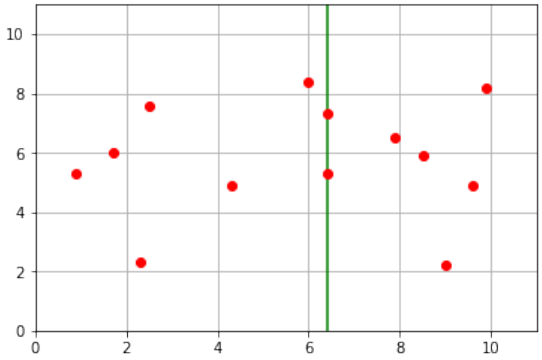


FIGURE 2 – Le point (6.4,5.3) est médian selon l'axe horizontal : 6 points sont à gauche et 6 sont à droite (y compris l'autre point sur cet axe). Cela permet de créer la racine du k-d tree (premier nœud, à droite)

la direction (0 pour l'axe horizontal) et le point médian (6.4, 5.3). Le sous-arbre gauche est le k-d tree généré par les 6 points à gauche, mais avec la direction verticale. On ordonne donc ces 6 points selon les y, ce qui donne :

$$P_g = [(2.3, 2.3), (4.3, 4.9), (0.9, 5.3), (1.7, 6.0), (2.5, 7.6), (6.0, 8.4)]$$

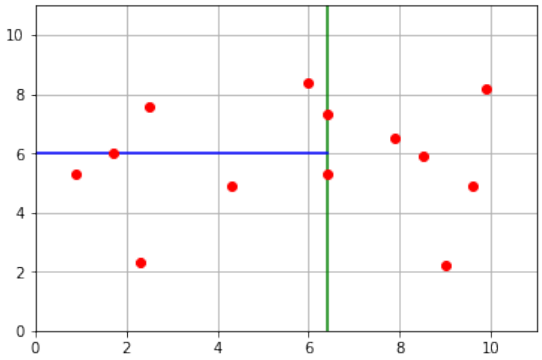
Comme il y a 6 points, le point médian pourrait être celui d'indice 2 ou celui d'indice 3. On choisit celui d'indice 3, (1.7, 6.0), ce qui sépare l'ensemble en 3 points en-dessous et 2 points au-dessus, comme indiqué sur la figure 3

On fait de même avec l'ensemble des points à droite : le point (7.9, 6.5) est le point médian, ainsi que représenté sur la figure 4

Et on itère, jusqu'à obtenir des ensembles vides de points de chaque côté. On obtient alors le résultat de la figure 5.

Dans une première partie, nous allons mettre en place les opérations de base au TP : définition d'un point, lecture d'un fichier de points, fonctions permettant de faire du tri en C.

Dans une deuxième partie, nous construirons le k-d tree et nous écrirons la fonction permettant de trouver le point le plus proche d'un point donné, en exploitant la structure du k-d tree.



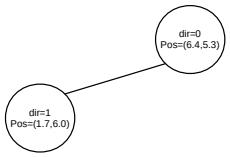
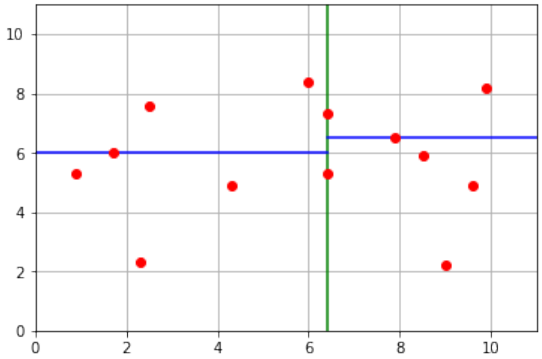


FIGURE 3 – Le point (1.7,6.0) est médian selon l'axe vertical pour les points de gauche : 3 points sont en-dessous et 2 points sont au-dessus. On obtient la racine du sous-arbre gauche.



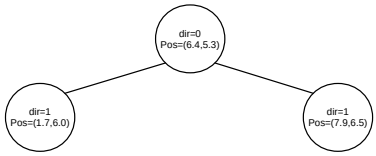


FIGURE 4 – Le point (7.9,6.5) est médian selon l'axe vertical pour les points de droite : 3 points sont en-dessous et 2 points sont au-dessus. On obtient la racine du sous-arbre droit.

2 Prolégomènes

Pour construire un k-d tree, nous avons besoin de

1. définir comment stocker un point
2. générer ou lire un ensemble de points dans un fichier
3. pouvoir trier ces points suivant leur composante x ou y

2.1 Définition d'un point

Nous allons simplement utiliser une structure et définir le point par :

```
1 struct Point
2 {
3   double x,y;
4 };
5 typedef struct Point Point;
```

ce qui peut se raccourcir en

```
1 typedef struct
2 {
3   double x,y;
4 } Point;
```

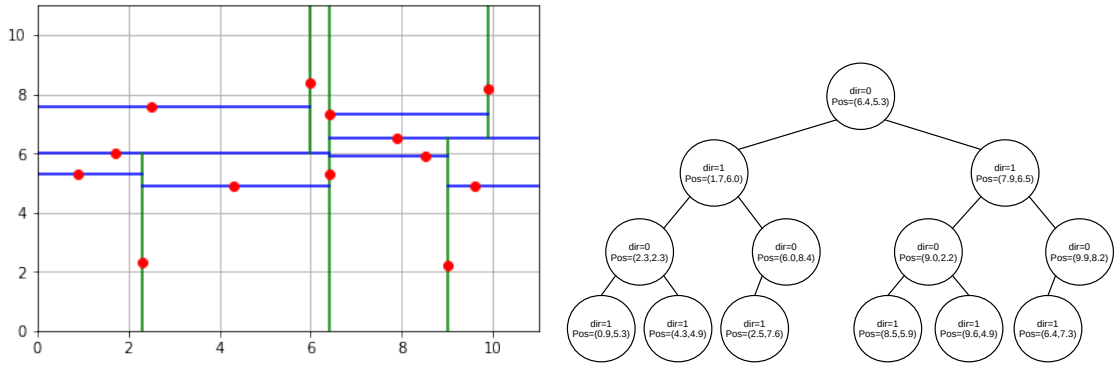


FIGURE 5 – Résultat final de la construction du k-d tree.

2.2 Lecture/écriture d'un ensemble de points

Une bonne manière de tester sur un grand nombre de points est de les générer aléatoirement. Le problème est que si la génération se fait à l'intérieur du programme, un bug éventuellement rencontré ne sera pas forcément répétable puisque les ensembles de points sont différents à chaque exécution. De plus, il est bon de pouvoir travailler avec un ensemble de points fixés, sur lesquels nous pouvons prédire à la main ce que doit donner le résultat. C'est pour cela qu'il est très utile de pouvoir stocker des ensembles de points dans des fichiers.

Il faut décider d'un format pour le fichier, et je vous propose le suivant :

— la première ligne ne contient qu'un nombre entier positif : le nombre de points stockés dans le fichier

— puis chaque ligne contient les coordonnées x et y d'un point, chacune étant donnée par un nombre réel

Par exemple, le fichier suivant contient 13 points :

```
13
1.7009472761773259 5.9989510783920768
4.3225136326265119 4.8971353587215463
6.3519619574546642 5.2673452278912745
6.0330189466630202 8.4454147557008152
2.3356180974913845 2.3116527834495777
0.9162957877462243 5.2816470690451780
9.6220666354624864 4.8714393074025581
7.9343434786118303 5.9987943693989862
9.0272704926446412 2.2469344140248997
9.9274354381148875 8.1953156731069630
8.4538131805387380 5.9405227871334754
2.5304195576023405 7.6345625275906936
6.3880725234691393 7.3496676270615620
```

Vous trouverez sur arche le fichier `genRndPoints.c` qui permet de générer aléatoirement N points (par défaut $N = 1000000$) dans un carré $[0, 10] \times [0, 10]$ (voir les variables `xmin`, `xmax`, `ymin`, et `ymax` dans ce fichier pour changer les dimensions de cette zone). Les points sont ensuite écrits dans le fichier passé en premier argument, suivant le format ci-dessus.

Exercice : Lecture d'un fichier de points

Écrivez la fonction `readPoints` qui permet de lire ce type de fichier. La fonction aura le prototype suivant :

```
1  Point *readPoints(char fname[], unsigned int *N);
```

et prendra donc en entrée une chaîne de caractères `fname`, qui indique le nom du fichier à lire, et renverra les points lus dans un tableau de `Point`. La taille de ce tableau (nombre de points) sera renvoyée grâce au deuxième argument `N`, passé par variable.

Le travail minimum demandé est une version qui réussit à lire les 13 points du fichier donné en exemple ci-dessus. Une note supplémentaire viendra avec la gestion correcte des erreurs : fichier impossible à ouvrir, erreur de format (pas d'entier en tête de fichier, ou pas le bon nombre de coordonnées). En cas d'erreur, la fonction `readPoints` renverra un pointeur vide (`(Point *)NULL`) et `*N` sera mis à zéro.

2.3 Tri en C

Les différents tris que vous avez vus en APL1 sont disponibles en C dans la librairie standard. On utilisera ici le *quicksort* qui est l'un des plus efficaces en moyenne. Il est implanté dans la fonction `qsort`.

Pour commencer, lisez la documentation de la fonction `qsort` en tapant `man qsort` dans un terminal. Vous pouvez voir qu'il est nécessaire d'inclure le fichier `stdlib.h` pour l'utiliser (directive `#include <stdlib.h>`). Ensuite, son prototype vous est donné :

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

- Le premier argument (`base`) est le tableau à trier. Il est indiqué comme étant de type `void *` qui est un type pointeur dit *générique*. Ceci est possible car toute variable pointeur occupe la même taille en mémoire (ce qu'il faut pour stocker une adresse). En revanche, on ne peut pas déréférencer ce type de pointeur et il faut fait une conversion explicite (*explicit cast*) avant de pouvoir le déréférencer. Nous en verrons un exemple plus bas.
- Le deuxième argument (`nmemb`) indique le nombre d'éléments dans le tableau
- Le troisième argument (`size`) indique la taille occupée en mémoire par un élément du tableau (utiliser `sizeof`).
- Le quatrième argument doit vous paraître étrange et mérite quelques précisions.

Chaque fonction définie en C a un nom. Ce nom désigne une variable et on peut effectivement utiliser une fonction comme une variable. La question se pose du type de cette variable. La réponse tient en deux points : 1) c'est un pointeur, 2) le type du pointeur est donné par le prototype de la fonction.

Dans le cas de la fonction `qsort`, le quatrième argument doit donc être le nom d'une fonction dont le prototype est de prendre deux arguments, chacun de type `const void *`, et de renvoyer un `int`.

L'exemple `exTri1.c` permet ainsi de trier un tableau de `float`.

```
#include <stdio.h>
#include <stdlib.h>

int cmpFloat(const void *a, const void *b)
{
  const float *fa=(const float *)a, *fb=(const float *)b;
  if (*fa < *fb) return -1;
  if (*fa > *fb) return 1;
  return 0;
}

int main(int argc, char *argv[])
{
  // on vérifie qu'on a au moins un nombre passé en argument
  if (argc < 2)
    {
      fprintf(stderr,"Erreur : il faut au moins un nombre\n");
      return 1;
    }
  // allocation du tableau qui va stocker les nombres passés en argument
  float *tab=malloc((argc-1)*sizeof(float));
  // conversion chaîne vers float de chaque argument, et stockage dans le tableau
  int i;
  for (i=0; i<argc-1; i++)
    tab[i] = strtof(argv[i+1],0);

  // affichage du tableau initial
  printf("Tableau initial\n");
```

```
29    for (i=0;i<argc-1; i++)
30      printf("%f ", tab[i]);
31    printf("\n");
32
33    // tri du tableau en utilisant la fonction cmpFloat pour la comparaison
34    qsort(tab,argc-1, sizeof(float), cmpFloat);
35
36    // affichage du tableau trié
37    printf("Tableau trié\n");
38    for (i=0;i<argc-1; i++)
39      printf("%f ", tab[i]);
40    printf("\n");
41
42    // libération propre de la mémoire allouée
43    free(tab);
44    return 0;
45  }
```

La fonction de comparaison de deux `float` est `cmpFloat`. On peut voir (ligne 4) qu'elle obéit au prototype demandé par `qsort`. La conversion des arguments est faite en ligne 6, ce qui permet le déréférencement dans les lignes suivantes pour accéder aux valeurs passées en argument. Comme demandé pour un tri croissant, la fonction renvoie -1 si le premier argument est inférieur au deuxième (ligne 7), renvoie 1 si le premier argument est supérieur au deuxième (ligne 8) et renvoie 0 si les deux arguments sont égaux (ligne 9). On remarquera que d'une part on n'a pas besoin de `else` puisque la fonction se termine dans chaque `if`, et d'autre part qu'on termine par le cas d'égalité car la comparaison de deux `float` est numériquement peu fiable à cause des erreurs d'arrondis qui interviennent dans les calculs. On prendra donc soin de ne jamais tester l'égalité stricte de deux `float` en C.

Petite question : comment pourrions-nous effectuer un tri décroissant ?

Exercice : Tri de points

Vous écrirez une fonction `cmpX` qui permet de trier des `Point` suivant leurs coordonnées x croissantes, et une fonction `cmpY` qui fait la même chose mais avec la coordonnée y. Vous testerez ces fonctions sur un ensemble de points (par exemple généré à la main ou avec `genRndPoints` et lu avec `readPoints`).

3 k-d Trees

Nous avons à présent ce qu'il nous faut pour coder des k-d trees. Vous répondrez aux questions suivantes.

Exercice : Structure

1. Comme nous l'avons vu en introduction, un nœud doit contenir : une direction (horizontale ou verticale) qui indique le sens de la coupe, un point qui indique la position de la coupe et un sous-arbre gauche et un sous-arbre droit. Proposez une `struct` qui permet de stocker cela. On pourra s'inspirer de ce que nous avons fait pour les listes chaînées. Vous nommerez cette structure `kdTree`.
2. Écrivez une fonction `newKdTree` qui prend en paramètres une direction (encodée comme un `int` qui vaut 0 pour la direction horizontale et 1 pour la direction verticale) et un point (encodé comme un `Point`) et qui alloue un `kdTree`, le remplit avec la direction et le point passés en argument, initialise les sous-arbres gauche et droite au pointeur `(kdTree*)NULL` et renvoie un pointeur vers la structure ainsi allouée.
3. Écrivez une fonction `delKdTree` qui prend en argument un pointeur vers un `kdTree` et libère proprement la mémoire (y compris les sous-arbres gauche et droit).
4. Écrivez une fonction `genKdTree` qui génère un k-d tree à partir d'un ensemble de points. L'ensemble de points sera passé en argument comme un tableau de `Point`. On passera donc également en argument le nombre de points `N`, et on donnera aussi en argument la direction de coupe pour le k-d tree (sous forme d'un `int` comme ci-dessus).
   La fonction suivra les étapes suivantes :
   (a) si l'ensemble de points passé en arguments est vide ($N = 0$), alors renvoyer l'arbre vide `(kdTree*)NULL`.
   (b) sinon, trier l'ensemble des points en utilisant la fonction `cmpX` si la direction vaut 0 et `cmpY` sinon.

(c) puis créer, avec `newKdTree` un nœud avec la direction courante, et le point médian (indice $N/2$ en valeur entière, que je note $|N/2|$).

(d) puis générer le k-d tree pour les points d'indice 0 à $|N/2|-1$ ($|N/2|$ premiers points) et le stocker dans le sous-arbre gauche.

(e) puis générer le k-d tree pour les points d'indice $|N/2|+1$ à $N-1$ ($N-|N/2|$ derniers points) et le stocker dans le sous-arbre droit.

(f) renvoyer l'arbre créé.

3.1 Plus proche voisin (PPV)

En dernier lieu, nous allons voir comment exploiter le k-d tree pour rechercher le plus proche voisin d'un point, que nous nommerons Q.

On pourrait croire qu'il suffit de descendre dans le bon sous-arbre à chaque étape, comme dans le cas d'une recherche classique, mais on cherche ici le plus proche voisin et cela ne permet pas de sélectionner un seul sous-arbre à tous les coups. Pour illustrer mon propos, je détaille les premières étapes de cette recherche dans l'exemple de l'introduction.

On commence par entrer dans l'arbre, ce qui nous permet d'initialiser le point le plus proche au point stocké sous la racine et la distance minimale à la distance à ce point.

Cela donne la situation de la figure 6. On doit descendre dans chaque sous-arbre qui peut potentiellement contenir des points plus proches que ce point initial. Cet ensemble est matérialisé par le cercle orange sur la figure. On descend donc systématiquement dans le sous-arbre contenant le point Q (c'est le centre du cercle!). Mais on voit qu'on doit aussi descendre dans le sous-arbre droit car le cercle y déborde.

Lors de la descente dans le sous-arbre gauche, la situation est différente (voir figure 7). On teste si le point courant est plus proche que le plus proche voisin actuel : ce n'est pas le cas ici (ligne en tirets), donc on n'a pas besoin de mettre à jour le cercle. On va ensuite descendre dans le sous-arbre gauche, qui contient Q, mais on voit ici qu'on n'a pas besoin de descendre dans le sous-arbre droit car le cercle ne l'intersecte pas : on peut donc l'exclure dans la suite.

La situation est la même dans le sous-arbre droit sous la racine (voir figure 8). Avec cette deuxième étape, on a donc évité au total 4 tests sur les 13 points de l'ensemble!

Exercice : PPV

Vous implanterez cette procédure sous la forme de la fonction `PPV` qui prendra en argument le k-d tree, le point Q, ainsi que le plus proche voisin et la distance minimale qui seront tous les deux passés par variable (ceci permet l'appel récursif de la fonction).

Le test pour savoir si le cercle courant déborde sur un sous-arbre est :

— `(P.x-Q.x)^2 < dmin^2` si la direction du nœud courant est 0

— `(P.y-Q.y)^2 < dmin^2` sinon

Vous testerez cette fonction sur divers ensembles de points, et comparerez notamment le résultat à une recherche exhaustive (en qualité et en temps).

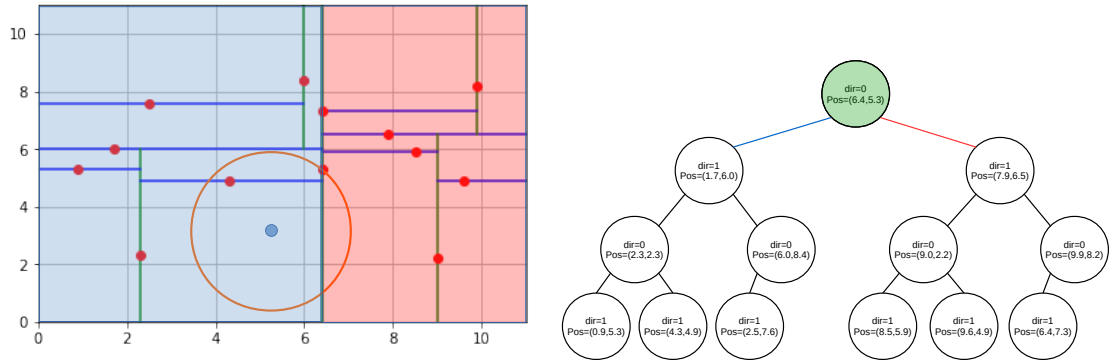


FIGURE 6 – On initialise le point le plus proche à celui stocké sous la racine et la distance minimale comme la distance à ce point. Ensuite, on descend systématique dans le sous-arbre qui contient le point Q. On voit cependant ici que des points plus proches peuvent se situer dans le sous-arbre droit (ensemble matérialisé par le cercle orange). Il nous faut donc aussi descendre dans le sous-arbre droit.

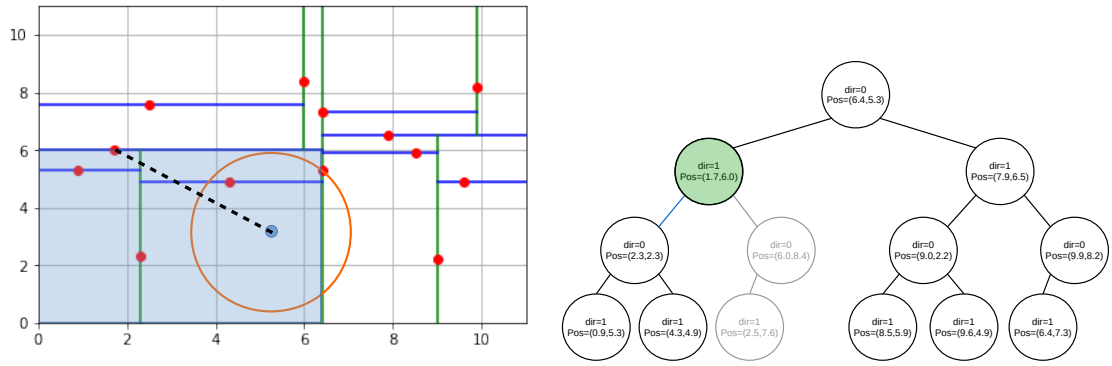


FIGURE 7 – Lors de la descente dans le sous-arbre gauche, on n'a pas besoin de mettre à jour le point le plus proche car le point stocké est plus éloigné. On descendra systématiquement dans le sous-arbre gauche qui contient Q, mais on n'aura pas besoin de descendre dans le sous-arbre droit puisque le cercle n'y déborde pas : on évite ainsi 2 tests dans la recherche.

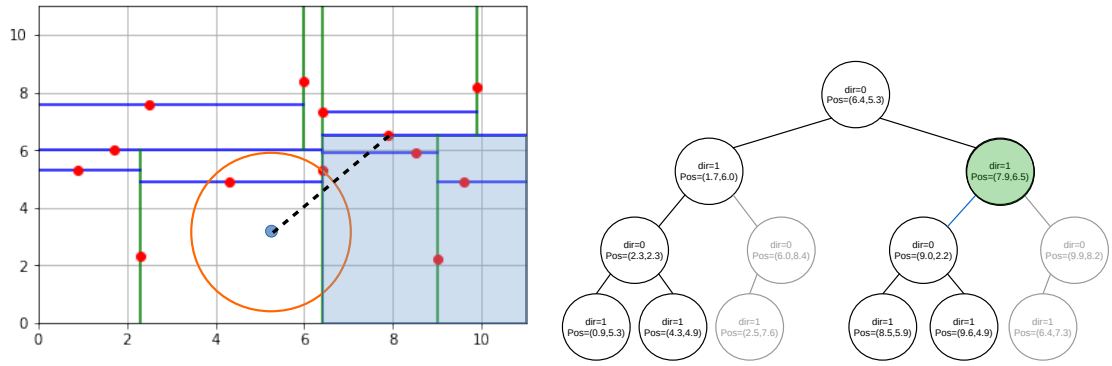


FIGURE 8 – Lors de la descente dans le sous-arbre droit, on n'a pas besoin de mettre à jour le point le plus proche car le point stocké est plus éloigné. On descendra systématiquement dans le sous-arbre gauche qui contient Q, mais on n'aura pas besoin de descendre dans le sous-arbre droit puisque le cercle n'y déborde pas : on évite ainsi 2 tests dans la recherche.