1. Notices d'utilisation :

a) Notice pour graphique :

repere [f distance] [ensemble] [partitions] [couleurs] [mode] [dimension x] [dimension y] [rangs] [points];;

Légende :

[f distance] = Fonction de distance : distance_euclidienne ou distance_manhattan
[ensemble] = Ensemble sous forme d'un Arbre Binaire de Recherche :
Voir « Notice pour ensemble »
[partitions] = Nombre de partitionnement désiré sous forme d'un entier
[couleurs] = Liste des couleurs désirés pour chaque partitionnement
[mode] = Mode d'interactivité désiré : Fixe ou Dynamique
[dimension x] = Dimension x de la fenêtre sous forme d'un entier
[dimension y] = Dimension y de la fenêtre sous forme d'un entier
[rangs] = Taille des rangs horizontaux et verticaux sous forme d'un entier
[points] = Taille du rayon des points sous forme d'un entier

b) Notice pour ensemble :

create_bst [points]
ou
create_bst_random [min] [max] [nb points]

Légende :

[points] = Liste des points désirés sous forme de liste de flottants
[min] = Liste des cordonnées minimums sous forme de flottants
[max] = Liste des cordonnées maximums sous forme de flottants
[nb points] = Nombres de points désirés sous forme d'un entier

2. Description des structures de données :

Les données sont stockées dans un Arbre Binaire de Recherche générique codé ainsi :
`type 'a bst = Empty | Cons of (('a * int) * 'a bst * 'a bst);;`
Avec `Empty` correspond au sous-arbre vide et `Cons` à un nœud de deux sous-arbres et d'un couple `('a * int)` d'une valeur et de son poids.

Pour l'ajout, nous disposons d'une fonction :
`let rec add = fun elem bst`
qui procède à l'ajout de l'`elem` dans le `bst` donné.

Pour la suppression, nous disposons d'une fonction :
`let rec remove = fun elem bst`
qui procède à une suppression de l'`elem` par remontée gauche dans le `bst` donné.

3. Algorithmes principaux :

a) Fonctions principales de k_moyenne :

```
(*Fonction d'ajout de l'élèment donné dans la partition la plus proche*)
let add_in_l_bst = fun f (l_p_a, l_bst) e ->
      let d_min = (d_min_in_list f l_p_a e)
      in
            let rec aux = fun (l_p_a, l_bst) acc ->
                  match (l_p_a, l_bst) with
                  | ([], []) -> (acc, d_min)
                  | ((p_a::ll_p_a), (bst::ll_bst)) ->
```

```
                        let d = (f p_a e)
                        in
                              if (d = d_min) then (aux (ll_p_a, ll_bst)
(add_in_list (add e bst) acc))
                              else
      (aux (ll_p_a, ll_bst) (add_in_list (remove e bst) acc))
                  | _ -> failwith "add_in_l_bst : Erreur de construction"
            in (aux (l_p_a, l_bst) [])
;;
```

Le but de cette fonction est d'ajouter l'élément e dans la partition la plus proche : nous récupérons la distance entre le point moyen le plus près et e. Puis nous cherchons la partition correspondant à la distance la plus proche, nous renvoyons alors la liste des partitions modifiée et la distance minimum.

```
(*Fonction d'ajout des élèments d'un ABR donné dans leurs partitions les plus
rapprochés*)
let partition_bst = fun f bst (l_p_a, l_bst) ->
      let rec aux = fun bst l_bst somme_d ->
            match bst with
            | Empty -> ((l_points_average f l_bst), l_bst, somme_d)
            | bst ->
                  let e = (elem_root bst)
                  in
                        let (new_l_bst, d) = (add_in_l_bst f (l_p_a, l_bst) e)
                        in (aux (remove e bst) new_l_bst (d +. somme_d))
      in (aux bst l_bst 0.)
;;
```

Le but de cette fonction est d'ajouter à la liste des partitions chacun des éléments de bst grâce à add_in_l_bst et de renvoyer ses nouveaux points moyens, la nouvelle liste de partitions et la somme des distances minimales.

```
(*Fonction d'obtention de la nouvelle liste d'ABR selon la liste de ses points
moyens*)
let rec partition = fun f (l_p_a, l_bst, somme_d_l_bst) ->
      let rec aux = fun l_bst acc somme_d ->
            match l_bst with
            | [] -> (acc, somme_d)
            | (bst::ll_bst) ->
                  let (_, new_l_bst, d) = (partition_bst f bst (l_p_a, acc))
                  in (aux ll_bst new_l_bst (d +. somme_d))
      in
            let (new_l_bst, new_somme_d_l_bst) = (aux l_bst l_bst 0.)
            and new_l_p_a = (l_points_average f l_bst)
            in
                  if (new_somme_d_l_bst >= somme_d_l_bst) then new_l_bst
                  else (partition f (new_l_p_a, new_l_bst, new_somme_d_l_bst))
;;
```

Le but de cette fonction est de modifier la liste des partitions, partition par partitions jusqu'à ce que la somme des sommes des distances minimales entre les éléments de chaque partition et leurs points moyens respectif ne varie plus.

```
(*Fonction de formation de k nuages selon un ABR et une fonction de distance*)
let k_moyenne = fun f bst k l_color ->
      let (l_pa_ini, l_bst_ini) = (partition_ini bst k)
      in
            let (l_pa, l_bst, somme_d) = (partition_bst f bst (l_pa_ini,
l_bst_ini))
            in (partition f (l_pa, l_bst, somme_d))
;;
```

Le but de cette fonction est lancé le programme en initialisant le bst en une liste de bst vide contenant partition par partitions jusqu'à ce que la somme des sommes des distances minimales entre les éléments de chaque partition et leurs points moyens respectif ne varie plus.

```
(*Fonction d'obtention du point moyen d'un ABR donné*)
let point_average = fun f bst ->
      let rec aux = fun bst k acc ->
            match bst with
            | Empty when (k = 0.) -> failwith "point_average : erreur division
par zero"
            | Empty ->
                  let acc_x = (List.nth acc 0) and acc_y = (List.nth acc 1)
                  in ((acc_x /. k)::(acc_y /.k)::[])
            | bst ->
                  let e = (elem_root bst)
                  in
                        let x = (List.nth e 0)                    and y =
(List.nth e 1)
                        and acc_x = (List.nth acc 0)  and acc_y = (List.nth acc
1)
                        in (aux (remove e bst) (k +. 1.) ((x +. acc_x)::(y +.
acc_y)::[]))
      in (aux bst 0. [0.;0.])
;;
```

Le but de cette fonction est de calculer le points moyens d'un ensemble donné, pour cela, nous faisons les sommes des coordonnées x et y puis nous les divisons par le nombre de points de l'ensemble. Nous renvoyons ensuite le point moyen.

```
(*Fonction d'obtention de la liste des points moyens d'une liste d'ABR donné*)
let l_points_average = fun f l_bst->
      let rec aux = fun l_bst acc ->
            match l_bst with
            | [] -> acc
            | (bst::ll_bst) -> (aux ll_bst (add_in_list (point_average f bst)
acc))
      in (aux l_bst [])
;;
```

Le but de cette fonction est de calculer les points moyens de toutes les partitions et de les renvoyer sous la forme d'une liste.

4. Choix liés à la programmation

Nous avons désiré laisser le libre choix à l'utilisateur dans la manière d'utiliser ce programme. Il a :

- le choix des fonctions de distance : Manhattan ou Euclidienne
- le choix dans la création de son ensemble :
  - Il peut aussi bien entrer lui-même une liste de points définis
  - Mais également obtenir une liste de points pseudo-aléatoires en entrant des minimums et maximums pour chacune des coordonnées.
- le choix dans le nombre de partitionnement effectué
- le choix dans les couleurs des nuages de points dessinés
- le choix dans le mode d’interactivité : Dynamique ou Fixe
- le choix des dimensions de la fenêtre
- le choix dans la tailles des rangs et des points.

Nous avons désiré partir sur des fonctions récursives terminales pour la plupart pour obtenir un meilleur rendement de temps.

Nous avons désiré privilégier la lisibilité du code en structurant les fonctions sous les mêmes formats, en commentant chacune des fonctions, ainsi qu'en divisant le code en plusieurs fichiers .ml pour permettre à tous les potentiels lecteurs de bien comprendre la division des tâches du programmes.

L'ajout des fonctions `elem_root` et `next` ont comme but de faciliter l'accès et le parcours de l'Arbre Binaire de Recherche. Le type `direction` permettant de choisir entre le sous ABR `Left | Right`.

L'ajout de la fonction `add_in_list` permet de faciliter la lisibilité du code.

L'objectif dans ce programme était de bien diviser les tâches entre chacune de ces fonctions. Pour cela, les fonctions permettaient de gérer un élément puis un groupe de ses éléments jusqu'à obtenir une fonction permettant de gérer le plus grand groupe de cet élément.

Ainsi, sur ce principe, nous avons une fonction `points_average` qui renvoie le point moyen d'un ABR et `l_points_average` qui renvoie la liste des points d'une liste d'ABR.

Nous divisons également les tâches ainsi :
1) `d_min_in_list` permet de trouver la distance la plus proche entre un élement et les points moyens d'une liste d'ABR
2) `add_in_l_bst` permet d'ajouter cet élément dans le point moyen le plus proche trouver par la fonction précédente.
3) `partition_bst` permet d'ajouter tous les éléments d'un ABR grâce à la fonction précédente.
4) `partition` permet d'ajouter tous les éléments de chaque ABR de la liste d'ABR donné tant qu'il y a des variations de points moyens.

Ainsi nous avons diviser les tâches en 4 fonctions remplissant chacune d'entre elles une tâche bien définie.

Pour la partie graphique, nous avons décidé de créer des fonctions prenant des fonctions en paramètre permettant d'obtenir des fonctions génériques :

`extremite_l_bst` prend en paramètre les fonctions `max min` et `extremite_x_bst` `extremite_y_bst` permettant d'avoir une fonction capable d'obtenir les extrémisées aussi bien minimales que maximales des coordonnées x ou y.

Nous avons divisé la partie graphique en deux temps :
- La partie traçage du graphique adapté à l'ensemble donné
- La partie traçage de l'ensemble

Pour la partie main, nous avons créé une fonction `repere` permettant d'entrer les choix attendus tel que le mode d'interactivité codé sous la forme d'un type `mode` permettant de choisir entre `Fixe | Dynamique`.

5. Difficultés rencontrées

Les difficultés ont surtout été ressenti dans la compréhension globale de l'algorithme des k-moyennes. l'implémentation du multi-ensemble un fois effectué, le blocage s'est fait sentir lors du début du codage de k-moyenne. Une entre-aide entre groupes a été nécessaire pour réussir à en sortir quelque chose. Puis le travail régulier a fini par rendre plus clair le problème et les solutions sont venus avec le temps.

La partie graphique fut également laborieuse à cause du problème d'adaptation du graphique à l'ensemble donné.
Le plus gros blocage fut lié à un problème de séquençage (avec « ; ») dans une conditionnelle. Puisse que nous n'avions pas malheureusement pas eu le temps de voir les mots-clés **begin** et **end** en Travaux Pratiques.

L'implémentation d'un repère en mode dynamique fut également compliqué.

6. Jeux d’essai

Nous vous invitons tester le programme avec le fichier exécutable « main.byte ».
Les commandes présentes sont dans le fichier main.ml :

```
(*Exemple fixe*)
repere distance_euclidienne (create_bst_random [-25.;-25.] [25.;25.] 20) 3
[blue;red;green] Fixe 1280 720 10 5;

(*Exemple dynamique*)
repere distance_euclidienne (create_bst_random [-25.;-25.] [25.;25.] 20) 3
[blue;red;green] Dynamique 1280 720 10 5;

(*Exemple 5 nuages*)
repere distance_euclidienne (create_bst_random [-25.;-25.] [25.;25.] 50) 5
[blue;red;green;black;cyan] Dynamique 1280 720 10 5;

(*Exemple positif*)
repere distance_euclidienne (create_bst_random [5.;5.] [25.;25.] 15) 3
[blue;red;green] Dynamique 1280 720 10 5;

(*Exemple négatif*)
repere distance_euclidienne (create_bst_random [-25.;-25.] [-5.;-5.] 20) 3
[blue;red;green] Dynamique 1280 720 10 5;

(*Exemple distance de Manhattan*)
repere distance_manhattan (create_bst_random [-25.;-25.] [25.;25.] 20) 3
[blue;red;green] Dynamique 1280 720 10 5;

(*Exemple du sujet*)
repere distance_euclidienne (create_bst [[1.;2.];[1.;1.];[3.;6.]]) 2 [blue;red]
Dynamique 1280 720 10 5;;
```