1 L'algorithme de Graham

L'algorithme de Graham permet de calculer l'enveloppe convexe (voir Figure 1) d'un nuage de n points (dans le plan) avec une complexité en $\Theta(n \log(n))$. Voir [1, section 33.3] pour une présentation de l'algorithme. Une implantation en C est donnée ci-dessous.

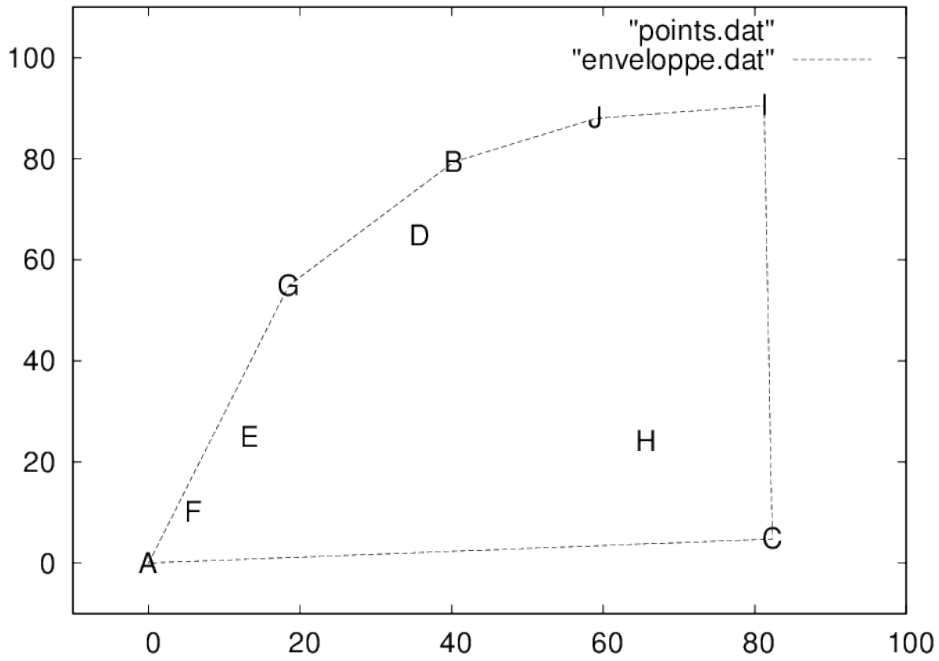

FIGURE 1 – L'enveloppe convexe d'un nuage de $N = 10$ points.

En dehors du point A, qui est placé à l'origine, tous les autres points ont des coordonnées strictement positives. Le point A fait donc partie de l'enveloppe convexe. On trie les autres points par angle polaire croissant. En d'autres termes, on les considère dans l'ordre suivant :

$$C\,,\ H\,,\ I\,,\ J\,,\ F\,,\ D\,,\ E\,,\ B\,,\ G\,.$$

Si on les énumère dans cet ordre-là, on remarque une propriété des points qui constituent l'enveloppe : à chaque virage, on tourne à gauche. L'enveloppe convexe se construit en utilisant une pile, initialisée avec les trois premiers points :

$$[A, C, H]\,.$$

On remarque qu'on tourne à gauche en C. Le point suivant considéré est I. Si on va de C à I en passant par H, on tourne à droite en H. On en déduit que H ne fait pas partie de l'enveloppe et on le dépile. Par contre, si on va de A à I en passant par C, on tourne à gauche. On empile donc I :

$$[A, C, I]\,.$$

Le point suivant considéré est J. Si on va de C à J en passant par I, on tourne à gauche en I. On empile donc J :

$$[A, C, I, J]\,.$$

Le point suivant considéré est F. Si on va de I à F en passant par J, on tourne à gauche en J. On empile donc F :

$$[A, C, I, J, F]\,.$$

Le point suivant considéré est D. Si on va de J à D en passant par F, on tourne à droite en F. On dépile donc F. Si maintenant on va de I à D en passant par J, on tourne à gauche en J. On empile donc D :

$$[A, C, I, J, D]\,.$$

Le point suivant considéré est E. Si on va de J à E en passant par D, on tourne à gauche. On empile donc E :

$$[A, C, I, J, D, E]\,.$$

Le point suivant considéré est B. Si on va de D à B en passant par E, on tourne à droite. On dépile donc E mais on n'a pas fini de dépiler. On voit que si on va de J à B en passant par D, on tourne encore à droite. On dépile donc D. Enfin, si on va de I à B en passant par J, on tourne à gauche. On empile donc B :

$$[A, C, I, J, B]\,.$$

Dernier point considéré : G. Si on va de J à G en passant par B, on tourne à gauche. On empile donc G. Il n'y a plus de points. La pile contient l'enveloppe convexe :

$$[A, C, I, J, B, G]\,.$$

```
#include <assert.h>
#include <stdbool.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

struct point {
    double x; /* abscisse */
    double y; /* ordonnée */
    char c;   /* identificateur */
};

/*
 * On veut trier les sommets par angle polaire croissant.
 * Retourne -1, 0, 1 suivant que A est avant B, aligné avec B ou après B.
 */

int angle_croissant (struct point* A, struct point* B)
{   double d = A->y * B->x - A->x * B->y;
    return d < 0? -1 : d > 0? 1 : 0;
}

/*
 * Retourne true si le chemin O -> A -> B fait un virage à gauche en A.
 */
```

```
bool tourne_a_gauche (struct point* O, struct point* A, struct point* B)
{   double d = (A->y - O->y) * (B->x - O->x) - (A->x - O->x) * (B->y - O->y);
    return d < 0;
}

#define N 20
#define MAX_COORDINATES 100
#define SCENARIO 8787231

int main ()
{   struct point T[N];
    struct point* E[N];
    int i, e;

    srand48 (SCENARIO);
/* On crée N points. Le point A est en (0,0) */
    T[0].x = 0;
    T[0].y = 0;
    T[0].c = 'A';
    for (i = 1; i < N; i++)
    {   T[i].x = drand48 () * MAX_COORDINATES;
        T[i].y = drand48 () * MAX_COORDINATES;
        T[i].c = 'A' + i;
    }
/* On les trie par angle polaire croissant */
    qsort (T+1, N-1, sizeof (struct point),
           (int (*) (const void*, const void*))&angle_croissant);
/* On ne gère pas le cas de deux points alignés */
    for (i = 1; i < N-1; i++)
        assert (angle_croissant (&T[i], &T[i+1]) != 0);
/* On stocke dans E les adresses des sommets de l'enveloppe convexe de T */
    E[0] = &T[0];
    E[1] = &T[1];
    E[2] = &T[2];
    e = 2;
    for (i = 3; i < N; i++)
    {
/* On peut montrer que e est toujours >= 1 */
        while (! tourne_a_gauche (E[e-1], E[e], &T[i]))
            e -= 1;
        e += 1;
        E[e] = &T[i];
    }
/* L'enveloppe convexe est dans E[0..e] */
    return 0;
}
```

Le programme précédent n'est pas très bien écrit. Il repose sur l'utilisation d'une pile.

Question 1. Quel est le type des éléments empilés et dépilés ? Quelles variables du programme servent à la gestion de cette pile ? Quelle partie du programme utilise cette pile ?

Question 2. Proposer une structure de données séparée pour la pile avec une interface minimale.

Question 3. Implanter cette pile avec un tableau puis avec une liste chaînée.

2 V'Lille

Ce sujet est inspiré par un Projet de Fin d'Études GIS.

Une ville contient une vingtaine de stations de V'Lille. Un camion parcourt régulièrement les stations. Il prélève des vélos dans les stations trop pleines et en dépose dans les stations trop vides. Pendant son déplacement d'une station à l'autre, des usagers se présentent aux stations, soit pour déposer, soit pour emprunter un vélo mais ce n'est pas toujours possible : lorsqu'ils cherchent à déposer un vélo dans une station pleine ou à emprunter un vélo dans une station vide. On cherche à minimiser l'insatisfaction des usagers.

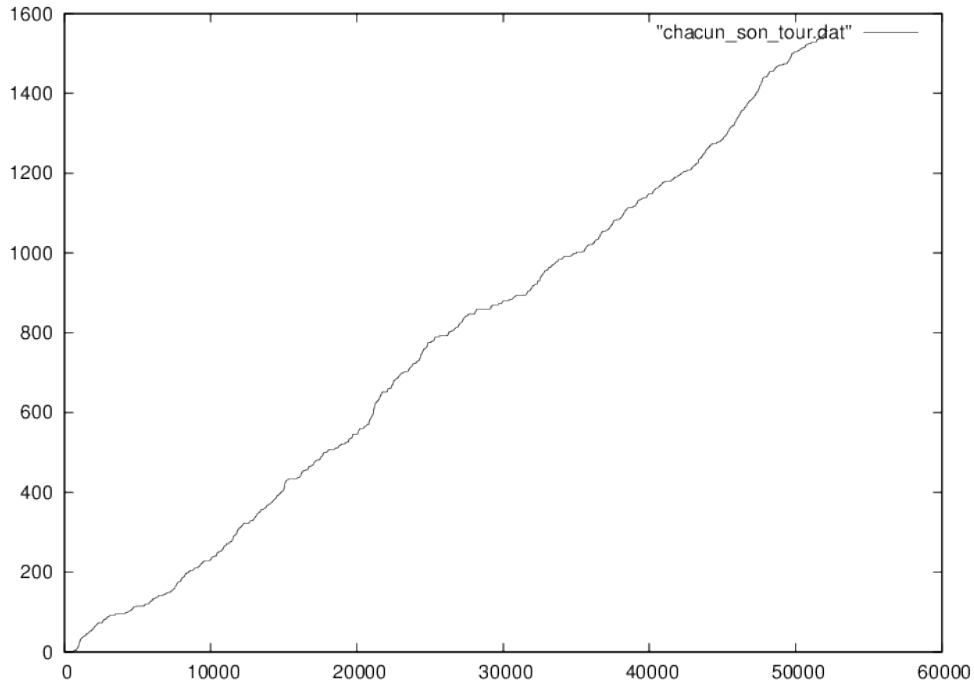

FIGURE 2 – En abscisse, le temps. En ordonnée, le nombre d'insatisfaits. Le camion parcourt les stations les unes à la suite des autres, sans se soucier du nombre de vélos effectivement présents.

La structure de données suivante représente la ville. Le champ `time` donne l'heure (il s'agit d'un temps abstrait). Le champ `tab` contient le tableau des stations. Le champ `truck` contient le camion. Le champ `file` contient une file de priorité de stations : à chaque fois qu'il quitte une station, le camion se rend dans la station la plus prioritaire.

```
#define NB_STATIONS 20
struct ville {
    double time;
    struct station tab [NB_STATIONS];
    struct camion truck;
    struct file_priorite_station file;
};
```

La structure de données suivante représente une station. Le champ `num` contient le numéro de la station, c'est-à-dire l'indice de la station dans le champ `tab` de la ville. Les champs `x` et `y` donnent les coordonnées cartésiennes de la station (ils sont compris entre 0 et `MAX_COORDINATES`). Le champ `velos` contient le nombre de vélos présents dans la station (entre 0 et `MAX_VELOS`). Le champ `derniere_visite` contient l'heure à laquelle la dernière visite a eu lieu (initialement `PAS_DE_VISITE`). Le champ `insatisfaction` contient le nombre d'usagers qui sont repartis mécontents de cette station depuis le début de la journée. Les autres champs sont techniques.

```
#define MAX_VELOS 20
#define MAX_COORDINATES 100
#define PAS_DE_VISITE -1
struct station {
    int num;
    int x, y;
    int velos;
    double derniere_visite;
    int insatisfaction;
    double frequence;
    int indice;
};
```

La structure de données suivante représente le camion. Le champ `num` contient le numéro de la dernière station visitée par le camion (initialement 0). Le champ `velos` contient le nombre de vélos présents dans la remorque (entre 0 et `MAX_VELOS`).

```
struct camion {
    int num;
    int velos;
};
```

Le programme principal est donné ci-dessous. Tant que la journée n'est pas finie, on défile la station la plus prioritaire et on stocke son adresse dans la variable globale `station_courante`. On déplace alors le camion vers cette station. Pendant le déplacement du camion, des usagers se présentent dans toutes les stations pour déposer ou emprunter des vélos. Le nombre d'usagers dépend de la durée du trajet, qui dépend de la distance parcourue par le camion. Une fois arrivé à la nouvelle station, on équilibre le nombre de vélos entre le camion et la station (par exemple, si le camion transporte 5 vélos et la station contient 13 vélos, on équilibre les deux quantités, de telle sorte que chacun contienne $(5 + 13)/2 = 9$ vélos). La station est remise dans la file et on recommence. À chaque itération, on affiche le nombre d'insatisfaits.

```
struct station* station_courante;

#define TMAX (1000 * MAX_COORDINATES * 0.521405433)
#define SCENARIO 65467868

int main ()
{   struct ville city;
    int grrr;

    station_courante = &city.tab [0];
    init_ville (&city, &chacun_son_tour, SCENARIO);
    while (city.time < TMAX)
    {   station_courante = defiler_priorite_station (&city.file);
```

```
        deplace_camion (&city, station_courante);
        equilibre_camion_et_station (&city.truck, station_courante);
        enfiler_priorite_station (&city.file, station_courante);
        enquete (&city, &grrr);
        printf ("%f %d\n", city.time, grrr);
    }
    return 0;
}
```

Les fonctions de priorité sont du type suivant. Elles retournent `true` si A est plus prioritaire que B et `false` sinon.

```
typedef bool fonction_de_priorite_station (struct station* A, struct station* B);
```

Question 4. Écrire une fonction de priorité `chacun_son_tour`, qui se contente de s'assurer que les stations sont toutes visitées les unes à la suite des autres, sans se soucier du nombre de vélos effectivement présents.

Un exemple de simulation est donné Figure 2.

Question 5. Votre fonction implante-t-elle une relation d'ordre *total*? Pensez-vous que le code donné dans le support de cours fonctionne avec une relation d'ordre partiel ?

Question 6. Proposer une meilleure fonction de priorité.

Question 7. Proposer un découpage du projet en fichiers. Écrire les fichiers d'entête.

Références

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Dunod, Paris, 2ème edition, 2002.