

TP Algo2 – session 8 – Arbres couvrants de poids minimum

23 mai 2025

Objectif d'apprentissage : implémenter l'algorithme de Prim (sur 2 séances de TP).

On repartira du code réalisé pour le tas (TP1 et TP2) et les graphes (TP7) en l'adaptant :

- la structure de tas sera utilisée en tant que file de priorité : il s'agira d'un tas min plutôt que d'un tas max : il faudra inverser le sens des comparaisons;
- le tableau du tas contiendra des pointeurs sur des sommets plutôt que des entiers;
- la structure de graphe est étendue pour inclure un poids pour chaque arête.

Il faudra utiliser le code suivant pour les définitions et implémenter chaque méthode (voir l'archive fournie) :

```
#ifndef MST_H
#define MST_H

#include <stddef.h>

/* Graph data structure */

typedef struct sgraph_vertex
{
    size_t degree;
    struct sgraph_vertex **neighbors;
    int *weights;
    // Data for MST procedure
    int distance;
    int index; // index in the heap queue
    struct sgraph_vertex *parent;
} graph_vertex;

typedef struct sgraph
{
    size_t size;
    graph_vertex **vertices;
} graph;

/*
 * Create an empty graph
 */
void graph_create (graph * self);

/*
 * Destroy a graph
 */
void graph_destroy (graph * self);

/*
 * Create a vertex, add it in the graph and return it
 */
```

```

    */
graph_vertex *graph_add_vertex (graph * self);

/*
 * Add an edge between two vertices (does not check if edge already exists)
 */
void graph_add_edge (graph_vertex * source, graph_vertex * destination,
                    int weight);

/* Queue data structure */

typedef struct
{
    size_t size;
    int *priorities;
    graph_vertex **vertices;
} queue;

/*
 * Create an empty priority queue
 */
void queue_create (queue * self);

/*
 * Destroy a queue
 */
void queue_destroy (queue * self);

/*
 * Size of the priority queue
 */
size_t queue_size (queue * self);

/*
 * Remove and return the vertex with minimum value in the priority queue
 */
graph_vertex *queue_extract_min (queue * self);

/*
 * Decrease the value of a vertex key at index i
 */
void queue_decrease_key (queue * self, size_t i, int key);

/*
 * Insert a vertex in the priority queue
 */
void queue_insert (queue * self, graph_vertex * vertex, int key);

/* Minimum Spanning Tree algorithm */

/*
 * Run Prim's algorithm to set parent and return the weight of the MST
 */
int mst_prim (const graph * self, graph_vertex * source);

```

```
#endif // MST_H
```

L'objectif de cette première séance sur ce sujet est d'implémenter et tester les fonctions de base (toutes les méthodes sauf la dernière). Comme précédemment, lorsqu'une fonction ne peut terminer son exécution (valeur absente ou argument à `NULL`), la fonction renverra une valeur par défaut : 0 si le type de retour est `int`, `NULL` si c'est un pointeur, `false` si c'est un booléen. On supposera cependant que si les pointeurs vers des structures `queue` ou `graph` sont définis (non nuls), alors les structures représentent des files de priorités ou des graphes valides et qu'il n'y a pas besoin de le vérifier dans `mst.c` (mais il faut s'assurer que les fonctions manipulent correctement ces structures dans `mst_tests.c`).

L'évaluation (facultative et à titre indicatif) consistera à soumettre deux fichiers : les implémentations (`mst.c`) et les tests unitaires (`mst_tests.c`). Si une fonction auxiliaire paraît utile pour les tests, c'est dans ce fichier de test qu'il faudra la rajouter.

L'évaluation de `mst.c` se fait sur la base du nombre d'avertissements générés par `gcc`, par `valgrind`, de la validité du code en utilisant les tests unitaires enseignants et sur le respect des conventions de codage avec `indent`. Si le fichier `mst_tests.c` soumis détecte un bug dans les implémentations enseignantes de référence, un bonus sera attribué.

Les deux fichiers doivent être soumis **tels quels** pour permettre le traitement et l'évaluation automatique (non compressés, non archivés, non renommés).