Projet MA016

Jean-Louis DU

Pour Janvier 2024

1 Sujet 2 : Planification de tâches

Une mission, une journée ou un projet bien organisé peut être beaucoup plus rapide et efficace que lorsque cela ne l'est pas.

C'est pour cela que l'on va se pencher sur le problème de comment trouver un ordre pour accomplir une liste de tâches données.

On va ici s'intéresser à un ensemble de tâches qui sont interdépendantes (dans le sens où on aura des liens du type une tâche A doit être réalisée avant de pouvoir commencer une tâche B). On va donc essayé de trouver un ordre optimal pour effectuer ses tâches en prenant en compte le temps nécessaire pour accomplir chaque tâche.

On considère le cas où il n'y a pas de dépendance circulaire, sinon il est impossible de trouver un ordre.

On va tenter de résoudre ce problème en le représentant sous la forme d'un graphe.

1.1 Sans le temps

Tout d'abord on va commencer par se donner une structure. Les Nodes représenteront les tâches a effectuer, elles seront caractérisé par une valeur/identifiant qui sera un entier. Les Edges représenteront les liens de dépendances entre deux tâches (par exemple l'edge (1,2) signifie que la tâche 1 doit être accompli avant de commencer la tâche 2).

Listing 1: Structure utilisée

```
// Definition d'un sommet (Node)
typedef struct Node{
        int value;                  // Valeur ou identifiant du sommet
        struct Node* nextnode;      // Pointeur vers le sommet suivant,
                                      NULL si il n'existe pas
}Node;

// Definition d'un cote (Edge)
typedef struct Edge{
        Node *src;                  // Pointeur vers le sommet source
        Node *dest;                 // Pointeur vers le sommet destination
        struct Edge* nextedge;      // Pointer sur l'edge suivant
}Edge;

// Definition du graphe (Graph)
typedef struct Graph{
        int nb_nodes;
        int nb_edges;
        Node* nodes;                // Pointeur vers le premier node
        Edge* edges;                // Pointeur vers le premier edge
}Graph;
```

Dans cette structure les Nodes et les Edges sont représentés par des listes chaînées. Contrairement à l'implémentation avec un tableau dynamique, de cette manière on peut ajouter autant de Nodes et d'Edges que l'on veut sans se soucier de la mémoire, dans le sens où l'on n'aura pas à recopier le tableau lorsque la capacité maximale du tableau sera atteinte à l'aide d'un "realloc".

De cette manière le graphe peut être décrit par le nombre de Nodes qu'il contient, le nombre d'Edge qu'il contient et les listes d'Edges et de Nodes (pour y accéder il suffit de pointer le premier élément de la liste).

On peut ensuite créer des fonctions pour initialiser, compléter, afficher et vider le graphe. Pour compléter le graphe on a notamment créé les fonctions :

Listing 2: Fonction pour ajouter des Nodes et des Edges

```
// Ajouter une Node
void add_node(Graph* g, int value);

// Ajouter un Edge
void add_edge(Graph* g, int src, int dest);
```

Avec tout cela, on est capable d'effectuer les opérations de bases sur un graphe, on peut maintenant représenter le problème que l'on veut résoudre dans un Graph.

Malheureusement on se rend compte que, bien que pratique pour manipuler un graphe, la structure "Graph" n'est pas adaptée pour trouver l'ordre de réalisation des tâches.

Pour palier à ce problème on va transformer notre structure de graphe en une structure de tableau. Maintenant que le nombre de de Nodes et d'Edges sont fixés, on n'a pas à s'embêter avec des "realloc".

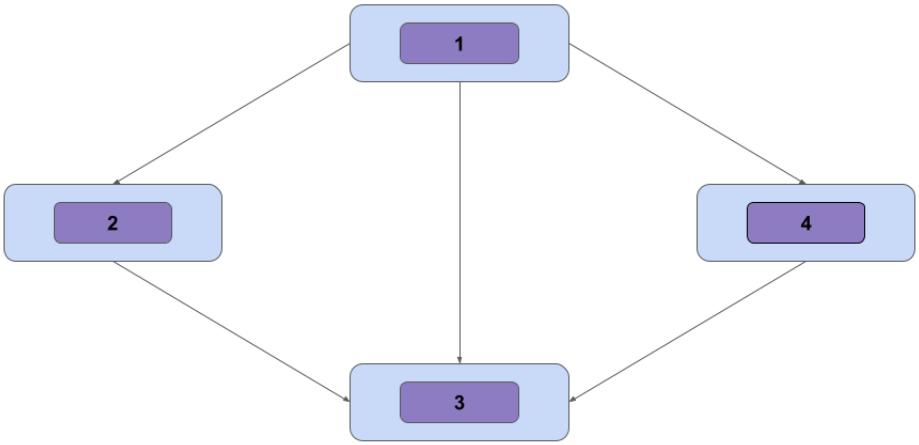
Listing 3: Structure utilisée pour trouver l'ordre

```
// Definition pour le Edge d'un tableau

typedef struct EdgeTab{
        int src;            // Valeur de la Node source
        int dest;           // Valeur de la Node destination
}EdgeTab;

// Definition d'une structure pour construire l'ordre

typedef struct OrderTab{
        int nb_nodes;
        int nb_edges;
        int* tab_nodes;
        EdgeTab* tab_edges;
}OrderTab;
```

Avec ce tableau on peut maintenant afficher l'ordre correspondant.

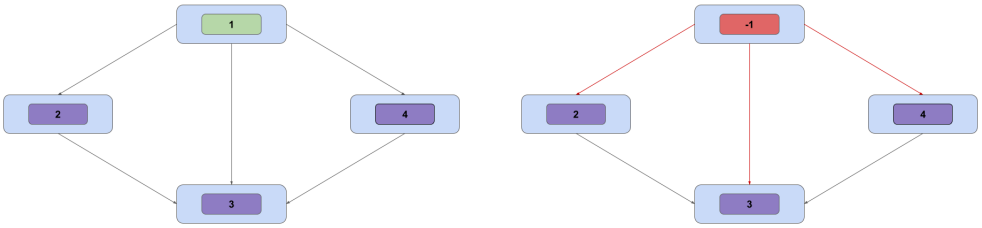
L'idée est de parcourir le tableau de Nodes jusqu'à en trouver un qui ne possède pas de source, c'est à dire que cette tâche n'a pas de pré-requis à accomplir. Une fois le Node trouvé, on l'affiche et on le marque comme parcouru (on lui assigne la valeur -1).
On mets ensuite à jour le tableau d'Edges, maintenant que le Node a été parcouru, on peut maintenant enlever tous les Edges l'ayant pour source. Concrètement cela veut dire que la contrainte liée à ce Node est maintenant levé.
On répète cela jusqu'à que tous les Nodes soient affichés.

On va étudier un exemple pour que cela soit plus clair.

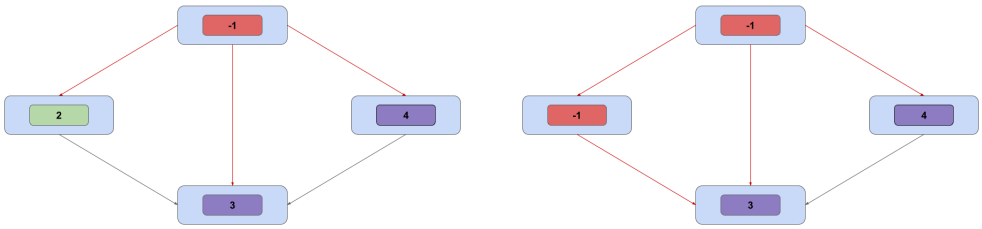


On va étudier le schéma ci-dessus composé de 4 Nodes représentés par les rectangles et 4 Edges représentés par les flèches.

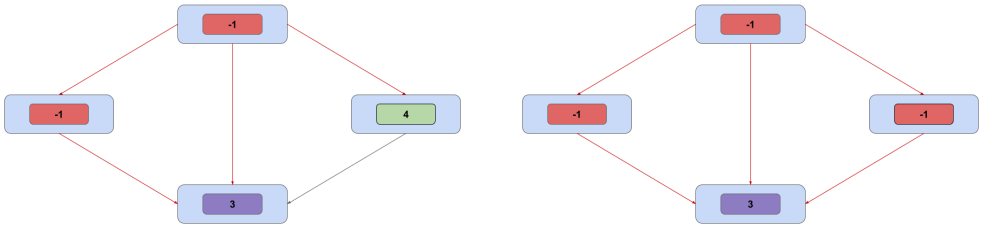
On va détailler un peu comment l'algorithme fonctionne.



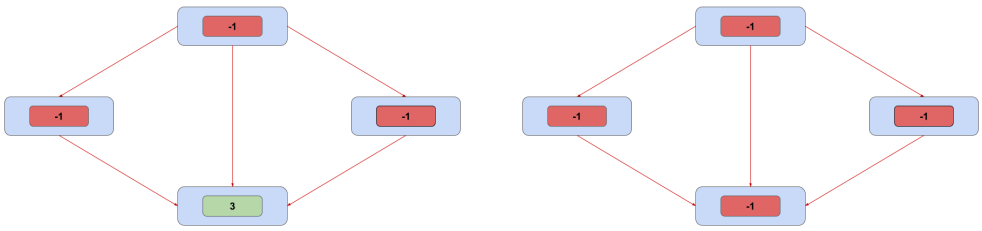
Le Node 1 n'a pas de source, on le marque donc comme visité (on lui assigne la valeur -1), on l'affiche et on enlève les Edges de source 1 (on lui assigne un EdgeTab (-1,-1)).



Puisque l'Edge (1,2) a été enlevé le Node 2 n'a plus de source, on peut donc exécuter la tâche 2 et faire la même chose que précédemment.

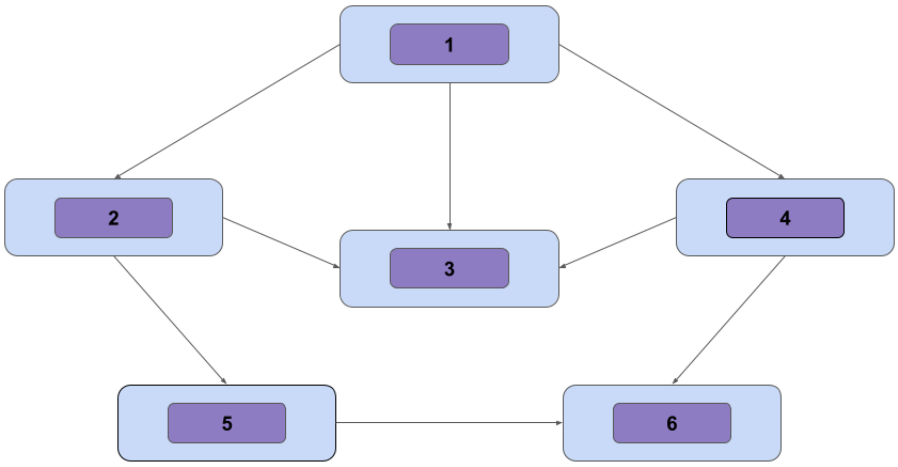


On fait la même chose. On remarque qu'à l'étape précédente on aurait pu exécuter la tâche 4 au lieu de la tâche 2, mais l'algorithme parcourt les Nodes dans l'ordre dans lequel ils ont étés rentrés.



On traite la dernière Node de la même façon. Maintenant que tous les Nodes ont été traités, l'algorithme se finit. Ainsi un ordre possible est 1, 2, 4, 3.

On peut se rajouter des Nodes et des Edges pour avoir un problème plus complexe. On peut par exemple traité le graphe suivant :



En utilisant l'algorithme précédent on obtient pour ordre : 1, 2, 4, 3, 5, 6.

1.2 Avec le temps

On va reprendre les mêmes structures que précédemment en y ajoutant un temps pour chaque Node qui représente la durée nécessaire pour effectuer cette tâche.

Listing 4: Structure utilisée

```
// Definition d'un sommet

typedef struct Node2{
        int value;                  // Valeur du sommet (Node)
        double time;                // Temps requis pour effectuer la tache
        struct Node2* nextnode;     // Pointeur vers le sommet suivant,
                                       NULL si il n'existe pas
}Node2;

// Definition d'un cote

typedef struct Edge2{
        Node2 *src;                 // Pointeur vers le sommet source
        Node2 *dest;                // Pointeur vers le sommet destination
        struct Edge2* nextedge;     // Pointeur sur l'Edge suivant
}Edge2;

// Definition du graphe

typedef struct Graph2{
        int nb_nodes;
        int nb_edges;
        Node2* nodes;               // Pointeur vers le premier Node
        Edge2* edges;               // Pointeur vers le premier Edge
}Graph2;
```

Cette fois ci, la fonction pour ajouter un Node comprend un argument en plus.

Listing 5: Structure utilisé

```
// Ajouter une Node

void add_node2(Graph2* g, int value, double time);
```

Comme tout à l'heure cette structure n'étant pas adaptée pour trouver l'ordre, on va donc la convertir en une structure de tableau.

Listing 6: Structure utilisée pour l'ordre

```
// Definition pour le Edge d'un tableau

typedef struct EdgeTab2{
        int src;            // Valeur de la Node source
        int dest;           // Valeur de la Node destination
}EdgeTab2;

// Definition pour le Node d'un tableau

typedef struct NodeTab2{
        int value;
        double time;
}NodeTab2;

// Definition d'une structure pour construire l'ordre

typedef struct OrderTab2{
        int nb_nodes;
        int nb_edges;
        NodeTab2* tab_nodes;
        EdgeTab2* tab_edges;
}OrderTab2;
```

Maintenant on a un tableau de Nodes, composé de sa valeur et de son temps, ensuite on a un tableau d'Edges composé de la valeur source et de la valeur destination comme dans la partie sans le temps.

L'idée cette fois est de créer un autre tableau de Nodes qui va contenir la valeur de la Node et le temps minimum qu'on mets pour la réaliser, ce temps va être actualiser au fur et à mesure.

Pour commencer on va créer un dictionnaire valeur/temps minimum appelé tmin (on aurait pu créer le tableau directement mais vérifier la valeur des Nodes du tableau une par une peut se révéler très coûteuse).
Pour cela on va supposer que les valeurs des Nodes sont comprises entre 0 et 100 (on pourra modifier la borne supérieure si besoin). On initialise à 0 les valeurs présentes dans le graphe, et -1 sinon.

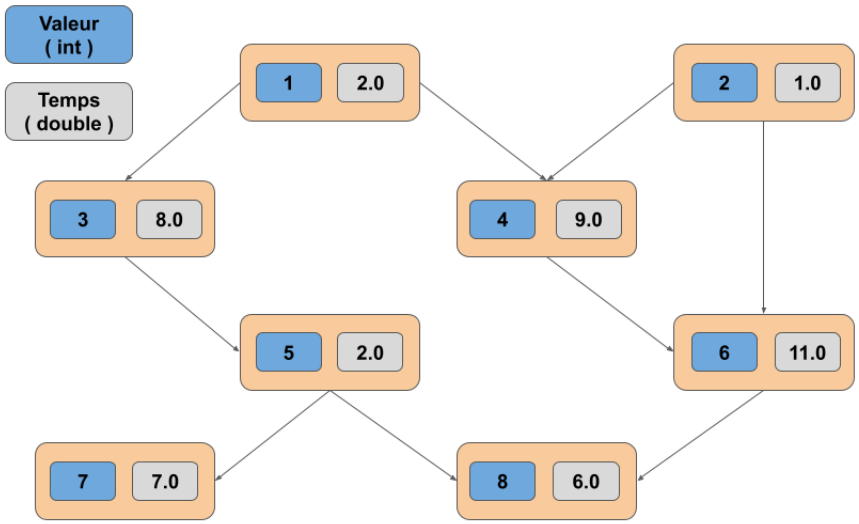
On reprends la même procédure que dans la partie sans le temps, à la seule différence que l'on actualise le temps minimum de tmin à chaque Node traité. En effet, une fois avoir vérifié qu'un Node i n'a pas de source, on ajoute le temps pour effectuer cette tâche au temps minimum de i dans le dictionnaire. Ce temps minimum vaut 0 si le Node ne possède de source traitée, et vaut le plus grand des temps minimum des sources déjà traités sinon.

Une fois que toutes les Nodes ont été traités, on peut maintenant tout ranger

dans un tableau de Node.
Maintenant on a dans un tableau la valeur du Node ainsi que le temps minimum requis pour effectuer chaque tâche.

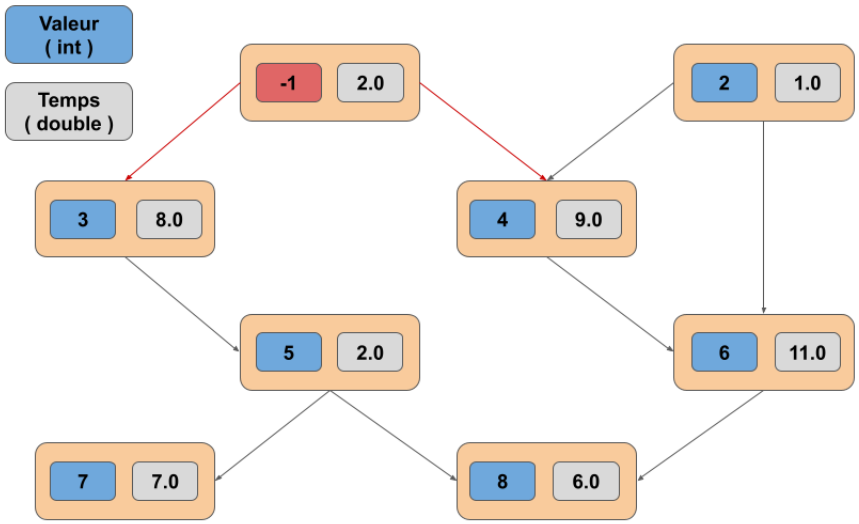
Pour afficher le bon ordre, il nous suffit de ranger les temps minimums du tableau par ordre croissant.
Pour cela on va faire appel à une routine de tri implémentée lors du TP2.

Pour mieux comprendre l'algorithme, prenons l'exemple suivant.



Dans ce problème, on a 8 Nodes et 9 Edges disposés comme ci-dessus.

On utilise la même méthode que précédemment sur ce graphe. La donnée de temps nous servira pour mettre à jour tmin.



On veut maintenant implémenter le temps dans le problème, on a donc crée un dictionnaire tmin qui ressemble à cela pour les premières étapes de la résolution. Les valeurs en vert correspondent à une Node traitée.

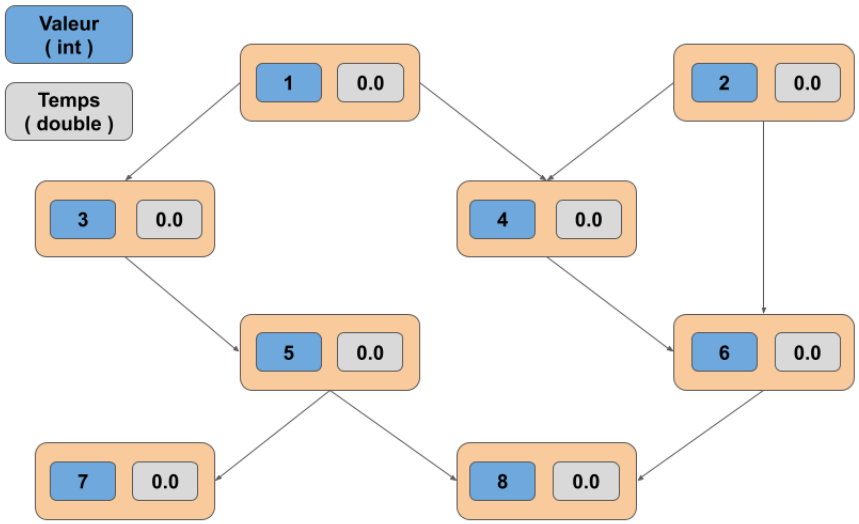
Valeur	0	1	2	3	4	5	6	7	8	9	10	...	100
Temps Minimum	-1	0	0	0	0	0	0	0	0	-1	-1	...	-1

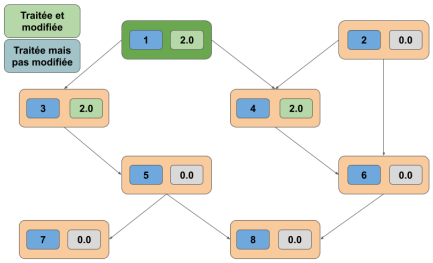
Valeur	0	1	2	3	4	5	6	7	8	9	10	...	100
Temps Minimum	-1	2.0	0	2.0	2.0	0	0	0	0	-1	-1	...	-1

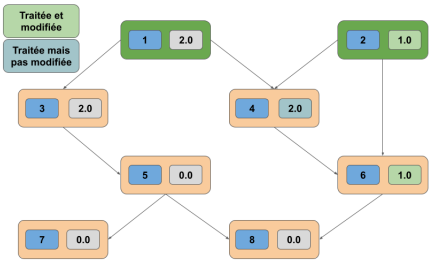
Valeur	0	1	2	3	4	5	6	7	8	9	10	...	100
Temps Minimum	-1	2.0	1.0	2.0	2.0	1.0	0	0	0	-1	-1	...	-1

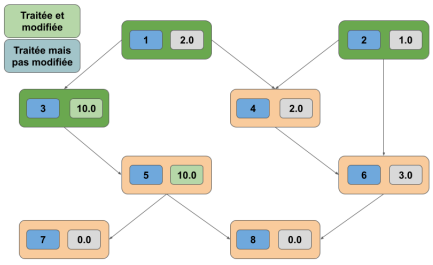
Valeur	0	1	2	3	4	5	6	7	8	9	10	...	100
Temps Minimum	-1	2.0	3.0	10.0	2.0	1.0	10.0	0	0	-1	-1	...	-1

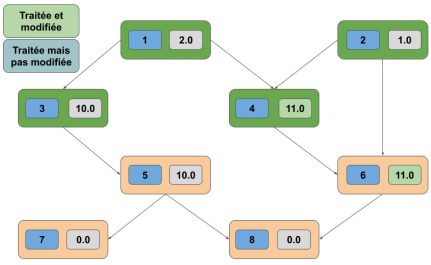
Pour mieux comprendre l'idée de l'algorithme on va le représenter comme si les temps minimums se font dans un graphe.
(Les graphes se lisent de gauche à droite, de haut en bas)

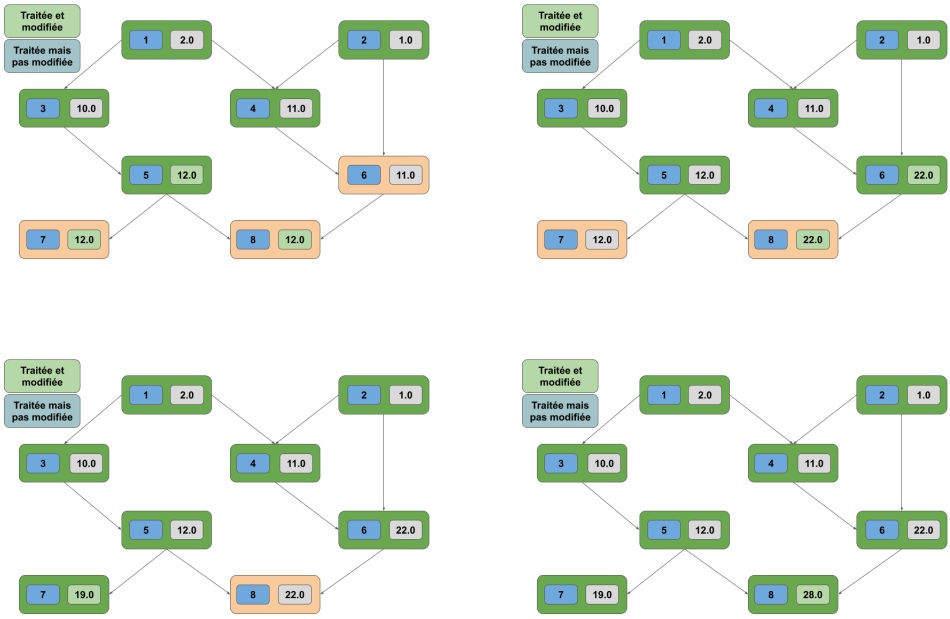












Ainsi après avoir rangés les temps par ordre croissant, on se retrouve avec pour ordre : 2, 1, 3, 4, 6, 7, 5, 8.