Polytech Lille
Université de Lille 1
Sciences et Technologies
Troisième année du Cycle ingénieur
Génie Informatique et Statistique

RAPPORT DE PROJET
GRAPHES COMBINATOIRES

FEUTRIER Thomas
LAPLACE Jordan
Tuteur : SEYNHAEVE Franck

GIS3A

SOMMAIRE

Introduction **3**

Cahier des charges **3**

Modélisation **4**

Explication des termes 5

Méthode de résolution **5**

Structures de données **6**

Structures de données envisagées 6

Matrice d'incidence 6

Table des arcs 7

Table des successeurs 7

Comparaison des structures 8

Principaux algorithmes de résolution **10**

Chargement / Niveau 10

Initialisation de tabNiveau 10

Algorithme de tri des niveaux 11

Tri rapide ou "quicksort" complexité 11

Tri fusion complexité 11

Tri par insertion complexité 11

Comparaison des algorithmes de tri 12

Algorithme de Bellman aller-retour 12

Pseudo-code des algorithmes de résolution **13**

Exemples traités **19**

Mode d'emploi **19**

Conclusion **20**

Introduction

L'objectif de ce projet est de réaliser un programme en langage de programmation C permettant de résoudre des "problèmes d'ordonnancement" vus en cours de graphes combinatoires, c'est à dire qu'on recherche une organisation optimale pour réaliser toutes les tâches, compte tenu de contraintes temporelles et précédence.
Nous présenterons dans un premier temps la modélisation utilisée puis nous expliquerons la méthode de résolution théorique d'un problème d'ordonnancement.
Ensuite, seront examinées les structures de données envisagées puis celle choisie pour la réalisation de ce programme.
Enfin nous expliciterons le but et le code des différentes fonctions qui composent notre projet.

Cahier des charges

Nous devons réaliser un code permettant de lire des données traduisant un problème d'ordonnancement simple et le résoudre.
Un problème d'ordonnancement simple consiste à organiser la réalisation de tâches, compte tenu de leurs contraintes temporelles (délais et contraintes d'enchaînement) .

Tout d'abord, il faut pouvoir lire les données stockées dans un fichier texte de format usuel pour décrire les problèmes d'ordonnancement ; ainsi, le document de départ aura ce format :

- Des lignes commençant par c, suivi d'un espace, contenant des caractères
- Une ligne commençant par p, suivi d'un espace, contenant la description du problème (nombre de sommets)
- n lignes commençant par v (vertex), suivi d'un espace, contenant les informations sur les noeuds
- m lignes commençant par a (arc), suivi d'un espace, contenant les informations sur les contraintes de précédences.

De plus le programme à réaliser doit calculer :

- Date au plus tôt de chacune des tâches
- Durée minimale du projet
- Date au plus tard de chacune des tâches
- Marge de chacune des tâches et mise en évidence des tâches critiques.

Enfin il stockera les résultats dans un fichier texte nommé "results.txt".
Pour finir le code de programme doit être optimisé et propre, car le nombre de sommets peut être de très grande taille (jusqu'à 300 000 tâches).

Modélisation

Le but de ce projet est de régler des problèmes d'ordonnancement simples ; or nous avons vu lors de nos cours que la meilleure modélisation face à ce genre de problème est la "potentiel tâches".

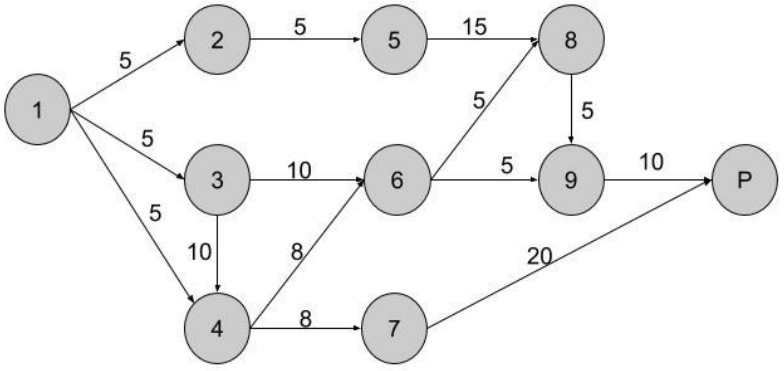
Celle-ci consiste à réaliser un graphe respectant ces règles :

- Les sommets représentent les tâches.
- Les arcs représentent les liens de précédence.
- Les poids des arcs représentent les durées des tâches initiales de ceux-ci.
- On ajoute un sommet sans successeur nommé "P" qui permettra donc de visualiser les durées des tâches qui n'ont pas de successeur.

Par exemple on se trouve face à ce problème d'ordonnancement :

Tâche	T1	T2	T3	T4	T5	T6	T7	T8	T9
Précédence	-	T1	T1	T1, T3	T2	T3, T4	T4	T5, T6	T6, T8
Durée	5	5	10	8	15	5	20	5	10

On obtient donc avec la modélisation "potentiel tâches" le graphe ci dessous :



Une fois la modélisation créée, on recherche ce que l'on appelle :

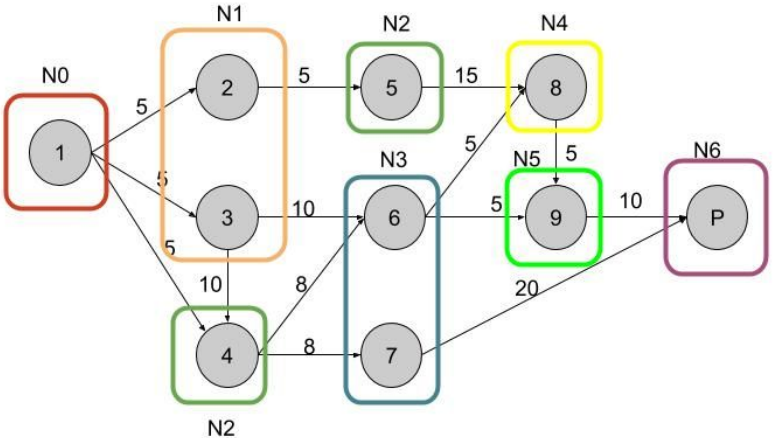
- la date au plus tôt de chaque tâche
- la date au plus tard de chaque tâche

Explication des termes

La date au plus tôt représente le temps auquel une tâche peut être lancée au plus tôt.
Par exemple sur le problème ci-dessus, la tâche 1 peut être lancée au temps 0 au plus tôt, car elle n'a aucune contrainte de précédence.

La date au plus tard d'une tâche représente quant à elle le temps auquel celle-ci peut être lancée au plus tard sans pour autant retarder la fin du projet.

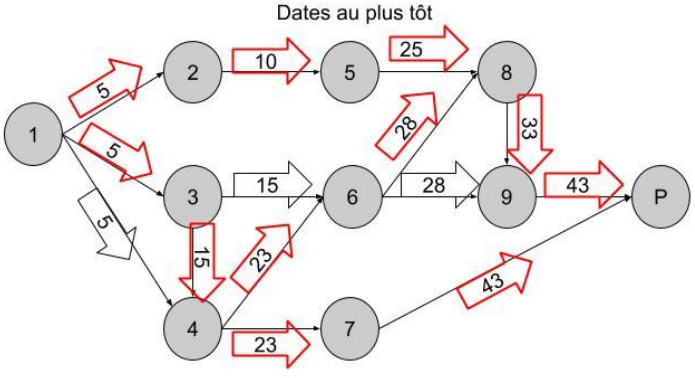
La définition d'un niveau est : si une tâche est de niveau "i", cela veut dire qu'en partant de la tâche initiale (toujours sans précédence), il faut au maximum"i" tâches accomplies pour que celle-ci puisse démarrer.
Exemple :



Méthode de résolution

Nous avons vu que notre programme doit résoudre un problème d'ordonnancement et donc trouver les dates au plus tôt , dates au plus tard et la durée minimale du projet.

La recherche de la date au plus tôt revient à résoudre le problème du plus long chemin, qui correspond à trouver le chemin partant d'une tâche initiale puis qui passe par les arcs les plus lourds de la modélisation, ce qui permet de trouver le potentiel maximum de chaque sommet.



Dans l'exemple ci-dessus, on peut voir les dates au plus tôt, représentées par les flèches rouge entrantes des sommets.

Ainsi, on va utiliser un algorithme de recherche de plus court chemin modifié pour rechercher le plus long chemin.
Nous avons vu deux algorithmes :

- Dijkstra, qui ne marche qu'avec des arcs de poids positifs ou nuls
- Bellman, qui ne marche qu'avec des graphes sans circuit (boucle) et qui nécessite le tri des sommets par ordre topologique (donc par niveau croissant).

Ces deux algorithmes, en temps normal, fournissent l'arborescence couvrante de poids minimal.

Cependant pour Dijkstra, à chaque date au plus tôt trouvée, on doit regarder tous les sommets du problème ; ainsi, pour un nombre de tâches très grand, le temps de traitement sera trop long.

De plus la modélisation "potentiel tâches" ne crée pas de boucle, car une tâche ne peut pas engendrer une tâche avec qui elle a un lien de précédence.
On décide donc, malgré le tri topologique (très coûteux en temps) d'utiliser l'algorithme de Bellman (explication de son fonctionnement dans la partie : Algorithme de Bellman aller-retour).

Afin de calculer les dates au plus tard de chaque sommet, on applique l'algorithme de Bellman sur l'anti-arborescence issue du sommet de fin (dans l'exemple P). Ici, on cherche le potentiel minimum de chaque tâche par rapport à la durée minimale trouvée.

Structures de données

Structures de données envisagées

Matrice d'incidence

Tout d'abord, nous avons rapidement étudié la possible utilisation de matrices d'incidence, qui stockent une information d'incidence entre un sommet et un arc :

- 1 (si le sommet est initial à l'arc)
- -1 (si le sommet est terminal à l'arc)
- 0 (si le sommet et l'arc ne sont pas reliés, non incidents).

Cependant, nous avons vu en cours que ces matrices, pour n sommets et m arcs, occupaient donc n*m cases mémoires et dont la plupart seront vides.
Or, le cahier des charges stipule que nos choix doivent prendre en compte que m et n seront extrêmement grands.

Sur l'exemple ci-dessous, il y a 28 cases occupées sur 10*14=140 cases disponibles :

Matrice d'incidence appliquée à l'exemple

sommet\arcs	1-2	1-3	1-4	2-5	3-4	3-6	4-6	4-7	5-8	6-8	6-9	8-9	7-P	9-P
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
2	-1	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	-1	0	0	1	1	0	0	0	0	0	0	0	0
4	0	0	-1	0	-1	0	1	1	0	0	0	0	0	0
5	0	0	0	-1	0	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	-1	-1	0	0	1	1	0	0	0
7	0	0	0	0	0	0	0	-1	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	-1	-1	0	1	0	0
9	0	0	0	0	0	0	0	0	0	0	-1	-1	0	1
P	0	0	0	0	0	0	0	0	0	0	0	0	-1	-1

Table des arcs

Puis nous avons pensé à utiliser une table des arcs.
C'est une représentation qui nécessite deux tableaux SI (sommets initiaux) et ST(sommets terminaux) de taille n+1 (nombre de tâches +1). L'indice choisi permet d'obtenir le sommet initial et le sommet terminal de l'arc ayant pour numéro "i".

Par exemple, si nous voulons le sommet initial d'un arc, il suffit de regarder SI.
Cette représentation permet de voir le lien de succession et de précédence entre sommets de manière très lente car SI n'est pas totalement triée, ce qui est le cas dans les fichiers de type DIMACS.
En outre, si l'on veut directement voir un sommet et ses successeurs, nous devons explorer le vecteur SI de taille m, ce qui nous paraît beaucoup trop gourmand en temps s'il y a 300 000 arcs.

Tables des arcs appliquée à l'exemple

SI	1	1	1	3	2	3	4	4	5	6	6	8	7	9

SF	2	3	4	4	5	6	6	7	8	8	9	9	P	P

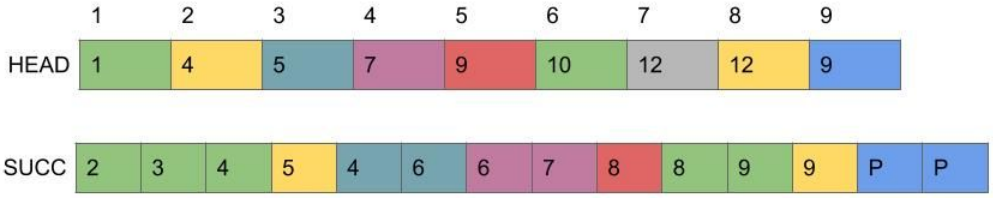
Table des successeurs

Enfin nous avons voulu utiliser la représentation la plus vue en cours, la table des successeurs :

- Succ est une liste rangée contenant successivement pour chaque sommet Xi, l'ensemble de ses sommets adjacents (successeurs) Xj.
- Head[i] = indice du tableau Succ à partir duquel sont rangés les successeurs du sommet Xi.

Cette méthode permet d'accéder à tous les sommets de manière immédiate et donc aussi à tous ses successeurs tout en étant très légère au niveau de la mémoire.

Table des successeurs appliquée à l'exemple



Comparaison des structures

Avant de regarder le tableau récapitulatif des avantages et inconvénients de chaque structures, il faut savoir que :

- m représente le nombre d'arcs
- n représente le nombre de sommets
- m >>n

Type de structures	Avantages	Inconvénients
Matrice d'incidence	-Accès direct aux arcs liés à chaque sommet	-n*m cases mémoires -Beaucoup de cases mémoires inutiles -Accès complexe aux prédécesseurs avec une complexité de O(m*n) et à leurs successeurs
Table des arcs	-Facilité de rangement des arcs -Accès rapide aux informations sur chaque arc	-2*m cases mémoires -Accès complexe aux sommets -Accès complexe aux prédécesseurs d'un sommet avec une complexité de O(m)
Table des successeurs	-Tableau trié des sommets -Accès direct aux successeurs -n+m cases mémoires	-Accès complexe aux prédécesseurs avec une complexité de O(m+n)

Nous avons vu dans la partie Méthode de résolution qu'un accès direct aux sommets et un accès rapide à leurs successeurs et à leurs prédécesseurs étaient cruciaux pour des problèmes avec des n et m très grands.

Ainsi, on observe qu'aucune de ces trois structures données ne respecte toutes les conditions fixées surtout en rapport avec les prédécesseurs. C'est pourquoi nous allons prendre la structure respectant le plus de conditions puis effectuer des changements sur celle-ci.

On sélectionne la table des successeurs car:

- elle peut stocker dans chaque case toutes les informations voulues
- elle permet l'accès le plus rapide aux successeurs d'un sommet

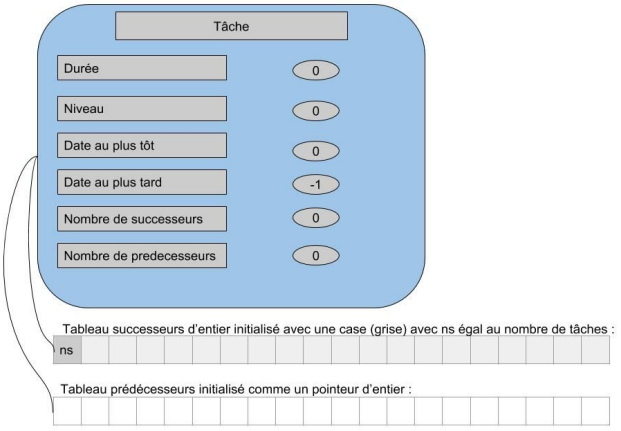
Cependant, pour le calcul de la date au plus tard, nous aurons besoin d'accéder aux prédécesseurs, c'est pourquoi nous avons rajouté un vecteur Pred qui est une liste rangée contenant successivement pour chaque Sommet Xi l'ensemble de ses sommets adjacents (prédécesseurs) Xj.

Cela ajoute des cases mémoires allouées, on passe de m+n à 2m +n. Cependant, les cases des tableaux Succ et Pred stockent des entiers qui ne prennent que 2 ou 4 octets en mémoire, contrairement à 8 octets pour des pointeurs.

De plus on réalise une variante de cette structure :

- nous décidons de stocker directement dans les cases du tableau tabSommet des petits vecteurs dynamiques succ et pred, qui stockent respectivement les indices des successeurs et prédécesseurs dans Tabsommet.
- nous créons dans chaque case deux entiers, nbpred et nbsucc pour savoir combien de successeurs et de prédécesseurs possèdent le sommet regardé.

Ainsi une case de tabSommet devient :



Principaux algorithmes de résolution

Une fois la modélisation faite et la structure de données choisie, nous avons réfléchi aux différents algorithmes que nous allions devoir utiliser ou créer de façon optimale pour diminuer au maximum les cases mémoires utilisées ainsi que la complexité des algorithmes.

Chargement / Niveau

Tout d'abord, nous avons vu dans la partie modélisation que chaque sommet doit stocker sa durée, la date au plus tôt ,la date au plus tard et le niveau de ce sommet.

Donc, il nous faudra créer un programme qui stockera dans notre structure de données les informations présentes dans le fichier DIMACS.

Lors de la lecture des arcs, on pourra comparer le niveau du prédécesseur + 1 et celui du successeur : si ce dernier est plus faible, il suffira de remplacer le niveau actuel du successeur par le niveau du prédécesseur + 1 .

Ainsi, ce programme pourra charger toutes les informations du fichier DIMACS concernant notre problème à résoudre et trouver les niveaux de chaque tâche.

Initialisation de tabNiveau

Pour appliquer l'algorithme de Bellman qui sera expliqué par la suite, il faut avoir un tableau trié par ordre croissant de niveau.
Cependant on ne veut pas modifier l’ordre du vecteur stockant les sommets.

On crée donc un vecteur tabNiveau de taille nombre de sommets +1 rempli d’entiers 0 à nombre de sommets +1.

Ainsi la case 0 possède l’indice de la case de la tâche 0.

tabNiveau

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

Il suffira par la suite de trier ce vecteur tabNiveau par niveau de la case correspondante dans tabSommet.

Algorithme de tri des niveaux

Pour résoudre un problème d'ordonnancement, nous utilisons l'algorithme de Bellman. Or, ce dernier nécessite un tableau trié par niveau. Cette étape de tri n'est pas à négliger car elle est souvent plus coûteuse que l'algorithme de Bellman. Nous nous sommes donc beaucoup renseignés sur le fonctionnement, la complexité et les différents cas d'utilisation des tris.

Nous partons du postulat suivant pour déterminer quel algorithme répondra le mieux à notre problème :

- Les niveaux sont presque triés. Plus on avance dans la lecture, plus les arcs dépendent de tâches de niveau inférieur. En conséquence il y a de fortes chances que le niveau du sommet "1253" soit plus élevé que celui de "10".

- La complexité spatiale doit être prise en compte, car nous devons traiter avec des données de grande taille (300 000 sommets). Un algorithme utilisant des sous tableaux sera donc à éviter.

Tri rapide ou "quicksort" complexité

cas optimal	cas moyen	pire des cas	complexité spatiale
nlog(n)	nlog(n)	n^2	log(n) en moyenne, n dans le pire des cas.

Tri fusion complexité

cas optimal	cas moyen	pire des cas	complexité spatiale
nlog(n)	nlog(n)	nlog(n)	O(n)

Tri par insertion complexité

cas optimal	cas moyen	pire des cas	complexité spatiale
n	n^2	n^2	1

Comparaison des algorithmes de tri

Notre objectif est de trier les sommets par niveau. Notre programme devra traiter avec un nombre de sommets important (300 000). La complexité spatiale est donc un indicateur à prendre en compte.

Le tri fusion est le plus rapide dans la plupart des cas avec une complexité en nlog(n). Mais sa complexité spatiale est trop importante n. Malheureusement, comme dit précédemment, nos données seront déjà presque triées, propriété que cet algorithme n'exploite pas.

Le tri rapide est en moyenne un bon mixte entre rapidité (nlog(n)) et espace mémoire (nlog(n)). Mais encore une fois, cet algorithme n'exploite pas le fait que notre tableau soit presque trié .

Le tri par insertion peut paraître mauvais avec une rapidité en n^2. Mais si les données sont déjà triées, sa complexité passe de n^2 à n, ce qui le rend plus efficace que le tri rapide et fusion. De plus sa complexité spatiale est de 1.

Le Smoothsort est un tri avec une complexité moyenne et dans le pire des cas en nlog(n). Quand les données sont presque triées, la complexité tombe à n. De plus sa complexité spatiale n'est que de 1. Mais l'implémentation est complexe. En effet, nous n'avons jamais utilisé ce tri, la documentation est moins répandue et nous n'avons pas le temps de nous pencher sur cet algorithme. Nous n'utiliserons donc pas le Smoothsort.

Nous avons choisi d'implémenter le tri par insertion car il a la meilleure rapidité et la meilleure complexité spatiale pour répondre au problème.

Algorithme de Bellman aller-retour

Enfin nous créons l'algorithme qui permettra d'obtenir les dates au plus tôt et dates au plus tard. Pour cela, Bellman aller-retour prend en paramètre un tableau trié par ordre de niveau, la taille et bien évidemment le tableau de sommet.

Cet algorithme se déplace par ordre croissant de niveau. A chaque sommet, il compare l'addition de la date au plus tôt et du poids de ce sommet aux dates au plus tôt de ses successeurs : si ces dernières sont plus faibles, alors on les remplace par la date au plus tôt plus le poids du sommet en cours de traitement.

Une fois les dates au plus tôt déterminées, on obtient la durée minimum du projet qui sera la date au plus tôt et la date au plus tard du sommet P.

Pour obtenir les dates au plus tard, on se déplace par ordre décroissant de niveau. A chaque sommet, l'algorithme compare la soustraction : date au plus tard de ce sommet - durée du prédécesseur aux dates au plus tard de ses prédécesseurs ; Si ces dernières sont plus fortes, alors on les remplace par la date au plus tard de ce sommet - durée du prédécesseur.

Pseudo-code des algorithmes de résolution

Fonction: initilialisation de tabSommet et tabNiv en fonction d'un fichier formaté de la façon suivante :
— Des lignes commençant par c, suivi d'un espace, contenant des commentaires,
— une ligne commençant par p, suivi d'un espace, contenant la description du problème. Ici
nous considérerons le nombre de sommets,
— n lignes commençant par v (vertex), suivi d'un espace, contenant les informations sur les
noeuds,
— m lignes commençant par a (arc), suivi d'un espace, contenant les informations sur les
contraintes de précédences

NB : init_noeud initialise les niveaux , successeurs et prédécesseurs de chaque sommet.
Sauf pour les prédecesseurs de p qui sont initialisé dans init_pred_p. P est à la case ns + 1 dans tabSommet.

détail_Initialisation :
poids de s = poids donné;
niveau de s = 0 Valeur calculé lors de l'ajout des arcs
dPlusTot de s = 0
dPlusTard de s = -1 lors de l'algorithme de Bellman change cette variable si elle est égale à -1. Ceci permet de détecter
les sommets dont la date au plus tard doit être initialisée .
nbsucc de s = 0;
nbprec de s = 0;
Action init_noeud(ns , nFichier, tabSommet, tabNiveau){

Données :
-nFichier: chaine de caractère {chaîne de caractères représentant le nom du fichier à lire}

Données/résultats :
-ns : entier {nombre de sommets}

Résultats:
-Vecteur d'entier [1...ns+1] tabNiv
{
- Tableau d'entier représentant un indice dans tabSommet.
- Les indices stockés représentent une tâche.
- Par exemple tabNiv[1] = 10, tabSommets[10] est un prédecesseur de Ni.
- Ce vecteur va être trié en fonction des niveaux des sommets.

- tabNiv[NS + 1] = indice de p (p toujours de niveau maximun).
}

-Vecteur de Noeud [1...ns+1] tabSommet
{tableau contenant toutes les tâches du problème d'ordonnancement courant.}

Variable locale :
-Entier : a,b {variable tampon des noms des sommets}
-chaine de caractère : chaine {chaine de caractère tampon de la ligne courante }
-Caractère : c {premier caractere de chaine }
-Réel : poids poids
- entier : i {indice de boucle }

Tant que le fichier n'a pas été lu en entier :

lire_ligne_suivante(chaine,nFichier)
c <= chaine[0]

{p = nombre de sommets à initialiser}
Si (c = 'p') alors
lire_ns(chaine, ns)

initialisation_des_vecteurs(ns,tabNiveau,tabSommet)

{Initialisation partiel du sommet p}
tabSommet[ns + 1].poids <= 0;
tabSommet[ns + 1].niveau <= 0;
tabSommet[ns + 1].dPlusTot <= 0;
tabSommet[ns + 1].dPlusTard <= -1;
tabSommet[ns + 1].nbsucc <= 0;
tabSommet[ns + 1].nbpred <= 0;

{a = arc à initialiser}
Sinon si (c = 'a')

{a = sommet initial}
{b = sommet final}
lecture_a_b(chaine,a,b)

{Ajout d'un succésseur à a}
tabSommet[a].nbsucc <= tabSommet[a].nbsucc + 1
tabSommet[a].succ[tabSommet[a]).nbsucc] <= b;

{Ajout d'un predécesseur à b}

```
                tabSommet[b].nbpred <= tabSommet[b].nbpred + 1;
                tabSommet[b].pred[tabSommet[b].nbpred] <= a;

                {Mise à jour du niveau de b.On compare le niveau actuel de b et celui de son
predecesseur +1,
                 Si le niveau potentiel est plus grand alors on écrase l'ancien.}
                Si (tabSommet[b].niveau < tabSommet[a].niveau + 1) {
                tabSommet[b].niveau <= tabSommet[a].niveau + 1;

                Fin si


        {Noeuds à initialiser}
        Sinon (c = 'v') {
                lire_a_poids(chaine, a , poids)

                tabSommet[a].poids <= poids;
                tabSommet[a].niveau <= 0;
                tabSommet[a].dPlusTot <= 0;
                tabSommet[a].dPlusTard <= -1;
                tabSommet[a].nbsucc <= 0;
                tabSommet[a].nbpred <= 0;

        Fin si

        fin tant que

        {Initialisation du tableau niveau}
        pour i de 1 à ns faire

        tabNiveau[i] = i + 1;

        fin pour

Fin action


Définition : Permet d'ajouter p en successeur au sommet sans prédécesseur

Action init_pred_p(struct Noeud ** tabSommet , int  ns){
    D/R:
      -Vecteur de Noeud [1...ns+1] tabSommet
       {tableau contenant toutes les tâches du problème d'ordonnancement courant.}

   Donnée:
      -Entier ns {nombre de sommets}
```

```
   VL :
   -Entier i {indice de boucle}

   Pour i de 1 à ns faire :

      {Si le sommet n'a pas de successeur}
      Si(tabSommet[i].nbsucc = 0 ) alors :
         {p devient un successeur de i}
         tabSommet[i].nbsucc  <= 1;
         tabSommet[i].succ[0] <= ns + 1 ;

         {i devient un prédécesseur de p}
         tabSommet[ns + 1 ].nbpred <= tabSommet[ns].nbpred + 1 ;
         tabSommet[ns + 1 ].pred[tabSommet[ns].nbpred] = i;


         //{Mise à jours du niveau de p}
         Si ( tabSommet[ns + 1 ].niveau < tabSommet[i].niveau + 1)

            tabSommet[ns + 1 ].niveau <= tabSommet[i].niveau + 1;

         Fin si
      Fin si
   Fin pour
}
```

Définition : - Algorithme de bellman aller et retour.
- Cette fonction permet, pour chaque sommet , de donner sa date au plus tôt et plus tard.

```
Action bellman(tabNiv , struct Noeud ** tabSommet ,  ns)

   D/R:
   -Vecteur de Noeud [1...ns+1] tabSommet
    {tableau contenant toutes les tâches du problème d'ordonnancement courant.}

   Donnée :
   -Vecteur d'entier [1...ns+1] tabNiv
         {
          - Tableau d'entier représentant un indice dans tabSommet.
          - Les indices stockés représentent  une tâche.
```

```
        - Par exemple tabNiv[1] = 10, tabSommets[10] est un prédécesseur de Ni.
        - Ce vecteur va être trié en fonction des niveaux des sommets.
        - tabNiv[NS + 1 ] = indice de p (p toujours de niveau maximum).
       }

  -Entier ns {nombre de sommets}

  VL:
  -Entier j,i {indice de boucle}


  {Bellman de niveau 0 à niveau p pour calculer la date au plus tôt}
  Pour i de 1 à ns + 1 faire :
    Pour j de 1 à tabSommet[tabNiv[i]].nbsucc faire :
      Si (tabSommet[tabSommet[tabNiv[i]].succ[j]].dPlusTot
         <  tabSommet[tabNiv[i]].dPlusTot
         + tabSommet[tabNiv[i]].poids  ) faire :

          tabSommet[tabSommet[tabNiv[i]].succ[j]].dPlusTot <=
            tabSommet[tabNiv[i]].dPlusTot  + tabSommet[tabNiv[i]].poids ;
      Fin si
    Fin pour
  Fin pour


  {On modifie la date au plus tard de p}
  tabSommet[ns + 1 ].dPlusTard <= tabSommet[ns + 1 ].dPlusTot;

  {Bellman de niveau p à niveau 0 pour calculer la date au plus tard}
  Pour i de ns + 1 à 0 pas  de -1 faire :s
    Pour j de 0  à tabSommet[tabNiv[i]].nbpred faire  :
      Si ( (tabSommet[tabSommet[tabNiv[i]].pred[j]].dPlusTard = - 1 ) OU
        (tabSommet[tabSommet[tabNiv[i]].pred[j]].dPlusTard >
        tabSommet[tabNiv[i]].dPlusTard  - tabSommet[tabSommet[tabNiv[i]].pred[j]].poids)
         ) faire :

            tabSommet[tabSommet[tabNiv[i]].pred[j]].dPlusTard <=
            tabSommet[tabNiv[i]].dPlusTard  -
tabSommet[tabSommet[tabNiv[i]].pred[j]].poids
      Fin si
    Fin pour
  Fin pour
```

```
Action tri_insertion : (ns , tabNiv , tabSommet ) {

   D/R :
   -Vecteur de Noeud [1...ns+1] tabSommet
   {tableau contenant toutes les tâches du problème d'ordonnancement courant.}

   -Vecteur d'entier [1...ns+1] tabNiv
      {
       - Tableau d'entier représentant un indice dans tabSommet.
       - Les indices stockés représentent  une tâche.
       - Par exemple tabNiv[1] = 10, tabSommets[10] est un prédecesseur de Ni.
       - Ce vecteur va être trié en fonction des niveaux des sommets.
       - tabNiv[NS + 1 ] = indice de p (p toujours de niveau maximun).
      }

   Donnée :
   -Entier ns {nombre de sommets}

   VL :
   -Entier tmp {entier tampon de l'indice du sommet courant}
   -Entier cmp {entier tampon du niveau du sommet courant}
   -Entier j,i {indice de boucle}

   {On commence à l'indice 2 car les deux premières tâches sont déjà triées par niveau
croissant}
   Pour i de 3 à ns + 1 faire :

      //On garde en mémoire l'indice du niveau courant
      tmp <= tabNiv[i]

      //On garde en mémoire le niveau courant
      cmp <= tabSommet[tabNiv[i]].niveau

      j <= i ;

      Tant que j > 0 et  tabSommet[tabNiv[j-1]].niveau > cmp Faire :

         tabNiv[j]  <=  tabNiv[j-1] ;
          j  <= j - 1

      Fin tant que
      tabNiv[j] <= tmp ;
   Fin pour
```

Exemples traités

Pour vérifier le bon fonctionnement de notre programme nous l'avons testé sur un problème d'ordonnancement à 9, 30, 120 puis 300000 tâches fournies par les tuteurs.

Nous avons pu comparer les durées minimales, les dates au plus tôt et au plus tard obtenues et ainsi confirmer le bon fonctionnement de notre programme.

Puis, nous avons étudié les durées d'exécution des programmes pour les 3 plus grands cas, pour éviter que le problème de 300 000 tâches ne soit résolu en un temps trop grand, ce qui engendrerait un non respect des contraintes du cahier des charges.

Ainsi nous obtenons :

```
tfeutrie@reuze08 ~/Desktop/PROJET_GC/SD-GC_FEUTRIER_LAPLACE (master) $ time ./main <test30
Entrez le nom du fichier
fin
real    0m0,004s
user    0m0,000s
sys     0m0,000s
tfeutrie@reuze08 ~/Desktop/PROJET_GC/SD-GC_FEUTRIER_LAPLACE (master) $ time ./main <test120
Entrez le nom du fichier
fin
real    0m0,008s
user    0m0,000s
sys     0m0,000s
tfeutrie@reuze08 ~/Desktop/PROJET_GC/SD-GC_FEUTRIER_LAPLACE (master) $ time ./main <test300000
Entrez le nom du fichier
fin
real    0m1,084s
user    0m0,544s
sys     0m0,020s
```

Au final, le temps de résolution ne dépasse pas 2 secondes, même pour un problème complexe comme celui de 300 000 tâches.

Mode d'emploi

Pour utiliser notre programme, il suffit de télécharger l'ensemble des fichiers disponibles sur le git de notre binôme.

Puis il faut entrer la commande :

gcc -o main main.c projet.c -Wall

Après cela, il suffit de saisir dans le terminal se trouvant dans le répertoire téléchargé, ./main.

Il suffit enfin de saisir le nom du fichier de format DIMACS avec l'extension (par exemple .txt) pour que notre programme résolve le problème posé.

```
tfeutrie@reuze08 ~/Desktop/PROJET_GC/SD-GC_FEUTRIER_LAPLACE (master) $ ./main
Entrez le nom du fichier
ordo30.txt
```

Il stockera les caractéristiques de chaque tâche, répertoriera les tâches critiques et la durée minimale du projet dans un fichier texte nommé result.txt.

Conclusion

Pour conclure, nous avons réalisé un programme capable de lire un fichier DIMACS tout en respectant scrupuleusement le cahier des charges pour ce projet, en proposant un code rapide dans le temps et optimisé pour l'espace mémoire utilisé.

Ce projet nous a permis d'améliorer notre méthodologie de projet : se mettre d'accord sur la méthode de résolution, programmer l'avancée du projet et faire face à des erreurs imprévues.

De plus, nous avons développé nos connaissances du langage C mais aussi cela nous a donné l'opportunité d'aborder des problèmes d'ordonnancement très complexes.