Mineure Informatique: PERT Maker

BOYER Timothé, GRASSET Emilien, HACINI Malik

9 Novembre 2023

Table des Matières

1 Introduction

Ce document relate la conception de PERT Maker. PERT Maker est une solution logicielle d'utilisation de la méthode PERT de gestion de projet. La méthode PERT fournit une méthode et des moyens pratiques pour décrire, représenter, analyser et suivre de manière logique les tâches et le réseau des tâches à réaliser dans le cadre d'une action à entreprendre ou à suivre. Un graphe de dépendances est utilisé.

Pour chaque tâche, sont indiquées une date de début et de fin au plus tôt et au plus tard. Le graphe permet aussi de déterminer le chemin critique qui conditionne la durée minimale du projet. Le but est de trouver la meilleure organisation possible pour qu'un projet soit terminé dans les délais, et d'identifier les tâches critiques, c'est à dire les tâches qui ne doivent souffrir d'aucun retard sous peine de retarder l'ensemble du projet.

Ce logiciel est le fruit d'un travail en groupe de longue haleine. Nous avons, pendant près de 2 mois, concu pas à pas le logiciel. Nous avons évidemment rencontrés beaucoup de difficultés tout au long de la conception, que nous détaillerons.

Ce rapport s'articule en 4 sections : Architecture, Organisation du travail en groupe, Conception et Tests.

2 Architecture

Le projet se décompose en 3 grandes parties :
L'Algorithmique des Graphes, la gestion du format CSV, et la génération de LaTeX. Nous avons découpé chacune de ces parties, et finalement organisé le logiciel en 6 modules. Chacun d'entre eux traite d'une partie spécifique du projet. Ils sont ensuite tous utilisés dans un fichier principal,nommé main, qui fait office de logiciel final.

3 Travail en Groupe

Ce projet comporte beaucoup d'étapes, et pour le concevoir de la manière la plus efficace possible, nous avons du trouver un moyen facile et rapide de collaborer, en présentiel ou à distance. Pour cela, nous avons choisi Github.
Nous avons crée un dépôt Github dédié au projet, sur lequel nous avons travaillé durant toute la conception. Chacun travaillait ensuite localement sur un clone du dépôt dans son IDE, et cela a rendu le partage de fichiers et le travail en parallèle simple, agréable et sûr.
Nous avons ensuite réparti les tâches entre les 3 membres du groupe, en suivant l'architecture établie au préalable. Le but était d'utiliser le plus efficacement possible les capacités de chacun, en le placant sur des tâches qu'il maîtrise.

Rpéartition du travail		
Tâche	Auteur principal	Aides
Graphes	Malik	Timothé
Cycle detector	Malik	-
CSV to graphe	Timothé	Emilien
CSV verifier	Timothé	Emilien
graphe to latex	Emilien	Malik
chemins critiques	Malik	Timothé,Emilien
Main	Malik	Timothé
Manuel utilisateur	Emilien	Malik
Rapport	Timothé	Malik,Emilien

Cette répartition n'est cependant pas absolue.
Dans la pratique, nous avons pu organiser beaucoup de séances de travail en présentiel. Alors, chacun apportait son aide au travail des autres, ce qui facilitait notamment la mise en commun de fichiers écrits en parallèles.

Aussi, plus la fin du projet approchait, plus nous devions recouper tout notre travail. Il était Alors nécéssaire que tout le groupe dispose d'une bonne compréhension du code, afin de repérer et corriger les failles.

4 Conception

4.1 Bases des Graphes

Nous avons démarrré la conception en reprenant la classe Graphe définie en TP. Cependant, nous lui avons apporté quelques modifications pour coller au projet. Premièrement, nous avons rendu les graphes pondérés : la durée de chaque tâche le poids de chaque arrête vers les tâches qui lui succèdent. Ensuite, nous avons ajouté une deuxième méthode de représentation d'un graphe : Sa matrice d'adjadcence pondérée. Cela nous a été particulièrement utile pour la détermination du chemin critique. Voici donc la nouvelle classe Graphe :

```
1
2  class DiGraphe:
3  def __init__(self,noeuds: list[str],arcs_ponderes : set[tuple[str, str, int]])->None:
4      """Construit un graphe orienté à partir des deux ensembles le définissant :
5      noeuds et arcs (pondérés). On suppose que le graphe est connexe. On le représentera
6      En utilisant sa matrice d'adjacence (array numpy), et un dictionnaire des listes d'adjacence.
7
8      Args:
9          noeuds (dict[int,str]): noeuds du graphe orienté. La clé est un entier (pour ordonner les
10         et la valeur une chaine de caractères (l'information que porte le noeud)
11
12         arcs_ponderes (set[tuple[int,int,int]): arcs du graphe orienté.
13         L'entier représente la pondération de l'arc.
14     """
15     noeuds_dict=dict()
```

```
16
17      i=0
18      for noeud in noeuds:
19          noeuds_dict[i]= noeud
20          i+=1
21
22
23      mat_adj=np.full((len(noeuds),len(noeuds)), np.inf)
24
25      keys=list(range(len(noeuds)))
26      vals=list(noeuds_dict.values())
27      for arc in arcs_ponderes:
28          ligne=keys[vals.index(arc[0])]
29          col=keys[vals.index(arc[1])]
30          mat_adj[ligne,col]=arc[2]
31
32      for i in range(len(noeuds)):
33          mat_adj[i,i]=0
34
35
36      self.dict_adj={key: [keys[vals.index(a[1])] for a in arcs_ponderes if a[0]==value] for key,val
37      self.noeuds=noeuds_dict
38      self.mat_adj=mat_adj
```

4.1.1 Détécteur de Cycle

Le détécteur de cycle dans un Graphe orienté, lui aussi établi en TP, était un indispensable pour ce projet. En effet, si un graphe de tâches est cyclique, le projet associé est incohérent, et nous devons le faire remarquer. Le détécteur fonctionne de manière récursive, en se basant sur le parcours en profondeur d'un graphe. En voici l'implémentation :

```
1
2   def cycle_detector_recursive_part(g:DiGraphe, noeud, visites: set, pile_recursive: list)->bool:
3   """Partie récursive du détecteur de cycle. Se base sur le DFS
4
5   Args:
6       g (DiGraphe): Graphe orienté à traiter
7       noeud (_type_): Noeud de départ du parcours
8       visites (set): La liste des noeuds visités
9       pile_recursive (list): La pile des noeuds du parcours en cours.
10
11  Returns:
12      bool: True si le graphe a un cycle à partir du noeud de départ, False sinon.
13  """
14
```

```
    visites.add(noeud)
    pile_recursive.append(noeud)

    for voisin in g.dict_adj[noeud]:

        if voisin not in visites:
            if cycle_detector_recursive_part(g, voisin, visites, pile_recursive):
                return True
        elif voisin in pile_recursive:
            return True

    pile_recursive.remove(noeud)
    return False


def cycle_detector(g: DiGraphe)->bool:
    """Détécteur de cycle dans un graphe orienté

    Args:
        g (DiGraphe): Graphe à traiter
    Returns:
        bool: True si le graphe a un cycle à partir du noeud de départ, False sinon.
    """
    visites=set()
    pile_recursive=[]
    for noeud in list(g.noeuds):
        if noeud not in visites:
            if cycle_detector_recursive_part(g, noeud, visites, pile_recursive):
                return True

    return False
```

A ce stade de la conception, nous avons débuté en parallèle les modules de gestion de CSV, et ceux d'algorithmes plus poussés sur les graphes.

5 Gestion du Format CSV

TIMOTHE A faire

6 Algorithmes plus complexes sur les Graphes

Pour appliquer la méthode PERT à un Graphe, nous devons en déterminer le chemin critique (chemin le plus long), et les dates de références pour chaque tâche. Notre processus de réfléxion fut le suivant:
- Conception d'un algorithme déterminant les distances et chemins les plus courtes vers chaque

tâche du graphe. (algorithmes de plus court chemin)
- Extraction du chemin critique et des dates de références.

Naivement, nous avons premièremnt pensé qu'il suffisait d'opposer les poids (les passer en négatifs) à un graphe, puis lui appliquer n'importe quel algorithme de recherche de plus court chemin. Cela ne fut malheureusement pas le cas.

6.1 Choix de l'algorithme de plus court chemin

Notre première idée fut d'implémenter l'algorithme de Dijkstra. Nous l'avons fait, avant de se rendre compte d'un problème majeur : l'algorithme de Dijkstra est un algorithme glouton. En conséquence, il est nécéssaire que tous les poids soient positifs pour assurer le fonctionnement.

Après plusieurs recherches et essais infructueux, nous avons donc décidé d'implémenter l'algorithme de Bellman Ford : un algorithme de recherche de plus court chemin similaire à Dijkstra, mais fonctionnel avec les poids négatifs. Cet algorithme est moins efficace que celui de Dijkstra (en complexité temporelle), mais il permet de concrétiser notre approche initiale. En voici l'implémentation :

```python
def bellmanFord(g: DiGraphe, source:int)->tuple[dict,dict]:
"""Performe l'algortihme de Bellman-Ford sur un graphe pondéré. Parcours un graphe à partir
d'une source, et en détérmine les plus courts chemins vers chaque noeud.

Args:
    g (DiGraphe): graphe à traiter
    source (int): noeud de départ

Returns:
    tuple[dict,dict]: le dict des distances , le dict des prédécésseurs (pour reconstruire le chem
"""
distances = {}
predecesseurs = {}
for noeud in g.noeuds:
    distances[noeud] = np.inf
    predecesseurs[noeud] = None
distances[source] = 0

for i in range(len(g.noeuds)-1):
    for j in g.noeuds:
        for k in g.dict_adj[j]:
            if distances[k] > distances[j] + g.mat_adj[j,k]:
                distances[k]  = distances[j] + g.mat_adj[j,k]
                predecesseurs[k] = j
return distances, predecesseurs
```

6.2 Extraction du chemin critique

Grâce à l'algorithme de Bellman Ford, il est aisé d'obtenir la durée incompressible d'un projet : il s'agit de la distance à là tâche finale, en partant de la tâche de départ, renvoyé par l'algorithme quand exécuté sur le graphe avec les poids opposés (donc négatifs). On peut aussi reconstruire le chemin exact en utilisant le dictionnaire des prédécésseurs.

```python
def chemin_critique(g: DiGraphe,    source: int, arrivee: int)->tuple[list,float]:
"""Renvoie les arcs et noeuds d'un des chemins critiques (le plus long) d'un DiGraphe d'une source
ainsi que sa durée. S'appuie sur la fonction bellmanFord


Args:
    g (DiGraphe): graphe à traiter
    source (int): noeud source
    arrivee (int): noeud d'arrivée

Returns:
    tuple[list,float]: Les arcs du chemin, et sa durée (distance)
"""
etape_chemin=arrivee
#On crée un graphe dont les poids sont les opposés des poids du graphe d'origine.
#De cette manière, la recherche dU chemin le plus long (chemin critique) dans le graphe d'origine
#revient à la recherche du chemin le plus court dans ce nouveau graphe.
g_oppose=copy.deepcopy(g)
g_oppose.mat_adj=-g_oppose.mat_adj
distances,pred=bellmanFord(g_oppose, source)
arcs=[]
noeuds=[]
while etape_chemin!=source:
    pred_actuel=pred[etape_chemin] #On remonte le chemin à l'envers (à partir des prédécésseurs)
    arcs.append((pred_actuel,etape_chemin))
    noeuds.append(pred_actuel)
    etape_chemin=pred_actuel
arcs=arcs[::-1]
noeuds=noeuds[::-1]
return arcs,noeuds, -distances[arrivee]
```

Nous décidons d'ailleurs de retourner en plus des noeuds le composant, les arcs du chemins critique. Cela n'était pas présent dans la première version de la fonction, mais nous l'avons ajouté pour assister la visualisation du graphe en LaTeX.

6.2.1 Limitations

Notre algorithme de chemin critique est limité. Par défaut, l'algorithme de Bellman Ford retourne le "premier" chemin critique qu'il croise. Cependant, il est possible que plusieurs chemins critiques

(donc de même distance) existent dans un même graphe.
Dans ce cas, notre fonction de recherche de chemin critique n'en renverra qu'un seul, et certaines tâches critiques ne seront pas "reconues". Cependant, l'impact n'est pas déterminant: les portions de chemins non reconnues seront de durée égale au portions de chemins critique. Alors, les dates de référénce pour chaque tâche resteront inchangés, ce qui représente le coeur de la méthode PERT.

6.3 Dates de référence.

Pour extraire les dates de références pour chaque tâche, il suffit aussi de s'appuyer sur les résultats de l'algorithme de Bellman Ford sur un graphe. L'obtention de la date de fin au plus tard etait moins aisée. En voici le détail de l'implémentation :

```
def dates_tot_tard(g: DiGraphe,duree_finale:int,noeuds_critiques)->dict[tuple[float,float,float]]:
    """Renvoie les dates de référence de chaque tâche d'un graphe.
    S'appuie sur la fonction bellmanFord. La durée de la tâche finale
    est un paramètre nécéssaire, car n'ayant pas de voisin, sa durée n'est pas encodée dans
    les arcs menants à ses voisins, contrairement aux autres tâches.

    Args:
        g (DiGraphe): graphe à traiter
        duree_finale (int): durée de la tâche finale

    Returns:
        dict[tuple[float,float,float]]: dictionnaire des dates. 3 dates pour chaque tâche
    """

    dates=dict()

    #On crée un graphe dont les poids sont les opposés des poids du graphe d'origine.
    #De cette manière, la recherche des distances les plus longues revient à la recherche
    #des distances les plus courtes dans ce nouveau graphe.
    g_oppose=copy.deepcopy(g)
    g_oppose.mat_adj=-g_oppose.mat_adj

    distances_plus_longues,pred=bellmanFord(g_oppose,0)


    for i in range(len(g.noeuds)): #On ne prend pas la tache finale (pas de voisin)
        #car sa duree est en paramètre de la fonction.

        if i!=(len(g.noeuds)-1):
            voisin=g.dict_adj[i][0]
            duree=g.mat_adj[i,voisin]
        else:
            duree=duree_finale

```

```
35
36          #On renvoie les 3 dates de références pour chaque tâche : début au plus tôt, fin au plus
37          #Les deux premières dates sont aisées à obtenir.
38          date_debut_tot=-distances_plus_longues[i]
39          date_fin_tot=date_debut_tot+duree
40
41          #Pour la date de fin au plus tard, c'est plus compliqué. Si la tâche
42          #n'est pas dans le chemin critique, il s'agit d'une tâche parallèle.
43          #On doit donc trouver son lien avec le chemin critique.
44          fils_min=noeuds_critiques[0]
45
46          if i==len(g.noeuds)-1: #La tâche finale est particulière. Elle est dans le chemin critique
47              date_fin_tard=-distances_plus_longues[i]+duree
48          elif i not in noeuds_critiques: #Pour les noeuds n'appartenant pas au chemin critique
49              fils_critiques=[fils for fils in g.dict_adj[i] if fils in noeuds_critiques] #On trouve
50              for fils_actuel in fils_critiques:
51                  if noeuds_critiques.index(fils_actuel)>=noeuds_critiques.index(fils_min): #On cho
52                      date_fin_tard=-distances_plus_longues[fils_actuel]
53          else:
54              date_fin_tard=-distances_plus_longues[i]+duree
55
56          dates[i]=(date_debut_tot,date_fin_tot,date_fin_tard)
57      return dates
```

7 Analyse

A ce stade de la conception, tout les algorithmes principaux d'extraction des informations d'un projet à partir d'un CSV, puis du traitement de celles-ci étaient bien avancés. Nous avons alors décidé de démarrer la conception du fichier principal du logiciel. Son rôle est d'interagir avec l'utilisateur, et de générer le code LaTeXde l'analyse de son projet.

7.1 Visualisation d'un graphe

La première étape de la rédaction de l'analyse en LaTeX est la visualisation du graphe de tâches. En particulier, nous devons mettre en évidence le chemin critique, ce que nous faisons en le colorant en rouge. Pour cela, nous utilisons grandement le langage dot du logiciel Graphviz et le package LaTeX dot2tex. En voici l'implémentation:

```
1  from Graphes import*
2
3  #Traduction d'un graphe en LaTeX
4  def graphe_to_latex(g: DiGraphe,chemin_critique: list=[])->str:
5      """Donne l'écriture en langage dot d'un graphe de tâches de la classe DiGraphe.
6      Colore aussi en rouge les chemins critiques donnés.
7
```

```
    Args:
        g (DiGraphe): graphe orienté à traiter

    Returns:
        str: code dot du graphe, directement utilisable en LaTeX
    """
    adj=g.dict_adj
    dot="""\\begin{center}
\\begin{tikzpicture}[scale=0.6, every node/.style={scale=0.6}]
\\begin{dot2tex}[codeonly]\n"""

    dot+= """
    digraph G{ \n""" + " "*3 + "rankdir=LR \n" + " "*3

    for noeud, voisins in adj.items():
        for voisin in voisins:
            dot+=f"{g.noeuds[noeud]}"
            dot+=f' -> {g.noeuds[voisin]} [label="{int(g.mat_adj[noeud,voisin])}"'
            if (noeud,voisin) in chemin_critique: #on colore en rouge les arcs du chemin critique
                dot+=' color="red"'
            dot+="]; \n" +"   "
    dot+="} \n"
    dot+="""\\end{dot2tex}
\\end{tikzpicture}
\\end{center}\n"""

    return dot
```

8 Tests