Sorbonne Université

Data structure and algorithms

Task scheduling

Méline Trochon
Anatole Vercelloni

Teacher : Didier Smeth

Novembre 2022 — Janvier 2023

Table des matières

1 Introduction

1.1 Présentation du problème du bonhomme de neige

Quand l'hiver arrive, chaque année, parfois il neige. Et quoi de mieux que de construire un bonhome de neige quand les conditions sont idéales. Problème : Vous vous souvenez de ce qu'il faut faire mais impossible de vous rappeler dans quelle ordre le construire. Heureusement, avec un algoritme d'ordonnancement, vous allez pouvoir vous assurez que votre bonhome ne sera pas difforme.

FIGURE 1 – Fred : un bonhomme de neige qui n'attend que vous pour être construit

Nous avons donc l'ensemble de tâche ci-dessous :

task	
t_0	placer le bonnet
t_1	placer la tête
t_2	placer l'œil gauche
t_3	placer l'œil droit
t_4	placer le nez
t_5	placer l'écharpe
t_6	placer le gant gauche
t_7	placer le gant droit
t_8	placer le bras gauche
t_9	placer le bras droit
t_{10}	empiler le corps et la tête
t_{11}	placer le bouton 1
t_{12}	placer le bouton 2
t_{13}	construire la tête
t_{14}	construire le corps

:

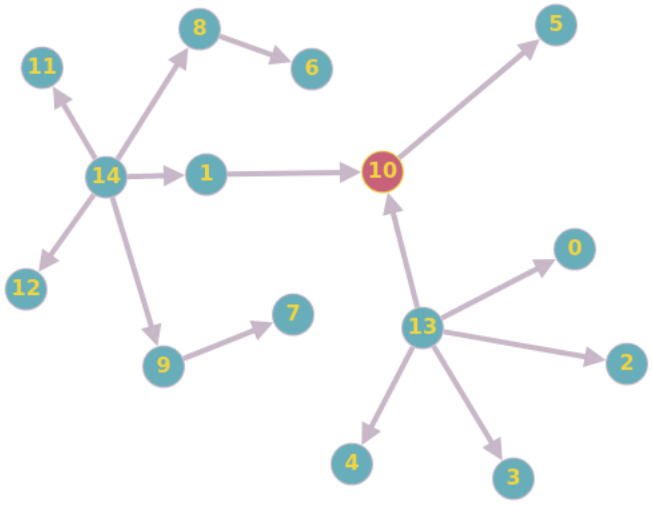

FIGURE 2 – représentation sous forme de graphe du problème du bonhomme de neige

1.2 Formalisation

Comme on peut le voir ci-dessus, une représentation du problème par un graphe le rend beaucoup plus accessible. Le but de ce projet va donc être de définir une structure de donnée ainsi que des algorithmes adaptés à la résolution d'un problème d'ordonnancement. Le problème se pose de cette manière : nous avons un ensemble de tâche $t_i, i \in \mathbb{N}$,

à éxecuter. Ces tâches ont des dépendances les unes par rapport aux autres, et l'objectif est donc de trouver l'odre dans lequel nous allons pouvoir éxecuter ces tâches.
Dans une première partie, nous nous concentrerons sur la manière de structurer nos données (ensemble de tâches, dépendances, ..), puis nous verrons un cas avec un exécutable et des durées de tâches toutes égales et enfin, dans une derniere partie nous étudierons le cas ou il y a une infinité d'éxécutables avec des tâches à durées variables.

2 Modélisation du problème

Comme on a pu le voir en introduction, le choix du graphe comme structure de données est assez naturel. En effet, nous pouvons attribuer à chaque tâche un sommet du graphe, et les dépendances sont représentées par des arcs orientés entre les tâches. Ainsi, si une tache t_i doit être éffectuée avant une tâche t_j, on aura dans notre graphe l'arc (t_i, t_j). Il ne restera plus qu'à suivre les arcs pour trouver une exécution valide (en fait, on remarque que cela revient à faire un tri topologique d'un graphe).

2.1 Structure de données

Considérons un ensemble de tâches $T = \{t_i\}$, $i \in \mathbb{N}$ et $A \subseteq \{(x, y) \mid (x, y) \in T^2,\ x \neq y\}$ un ensemble d'arcs.
On peut alors définir le graphe $G = (T, A)$

L'enjeux est de savoir comment nous allons stocker ce graphe. Un graphe d'ordonnancement est obligatoirement un graphe ordonné, par ailleurs, le graphe est usuellement peu dense. Ainsi, implémenter une matrice d'adjacence nous ferait prerdre de la mémoire inutilement, en plus de perdre du temps pour chercher tous les successeurs.

Par ailleurs, la taille des implémentations n'étant pas énorme, ce n'est ni la mémoire ni la rapidité de l'algorithme qui détermine un algorithme d'ordonnancement.

Nous avons donc fait le choix d'utiliser une liste d'adjacence. C'est à dire qu'à chaque sommet t_i nous lui associons une liste composée de tous ses successeurs.

Nous implémentons notre code en C++, cela nous donne les classes suivantes :

```
class Task{
    int id;
    vector<int> *next;
    int visited;
}
```

```
class ListTask{
    int size;
    vector<Task> data;
}
```

Ainsi, comme décrit ci-dessus, chaque tâche est représentée par un identifiant, un entier ainsi que sa liste de successeurs. Et le graphe est représenté par une ListTask. Nous avons fait le choix d'utiliser un pointeur sur un vecteur, afin que la taille en mémoire de chaque Task soit identique pour le vector 'data' de ListTask.

Pour pouvoir tester des problèmes, on passe par un fichier dans lequel la première ligne est dédiée au nombres de tâches, suivie de leur liste d'adjacence. Par exemple, pour le problème du bonhomme de neige, on a :

```
15 0
0
1 10
2
3
4
5
6
7
8 6
9 7
10 5
11
12
13 0 2 3 4 10
14 1 8 9 11 12
```

3 Première partie : exécution de tâches de même durée séquentiellement

3.1 Tri topologique

Nous pouvons facilement remarquer, et comme indiqué dans l'énoncé du projet, qu'une manière de résoudre notre problème d'ordonnancement équivaut à réaliser un tri topologique du graphe G.

En effet, pour un graphe acyclique orienté G(T,A), ts $= (t_0, ..., t_n)$ est un tri topologique si et seulement si, pour tout $(t_i, t_j) \in$ ts, $i < j$, t_i n'est pas dans la liste des successeurs de t_j

Cela remplit parfaitement nos conditions de dépendances entre les tâches, on peut donc s'intéresser à l'algorithme du tri topologique, dont voici le pseudo-code :

Algorithm 1 visit(t,L)

```
t.colour = gris
for t' in succ(t) do
  if t' est blanc then
    visit(t',L)
  end if
end for
t.colour = noir
L.push_back(t)
```

Algorithm 2 topological_sort(G(T,A))

```
L ← []
for t in T do
  if t est blanc then
    visit(t,L)
  end if
end for
return L
```

L'idée générale de l'algorithme est de parcourir les sommets du graphe, puis, pour chaque sommet, on s'intéresse à leur liste de successeurs. La difficulté est de savoir où est-ce que l'on est passé dans le graphe, et c'est pour cela que nous avons un champs colour (visited dans notre code) qui sert de marquage. Au départ, toutes les tâches sont blanches, puis les tâches en cours de visite sont grises, enfin, les tâches complètement traitées sont noires. Ainsi, cela permet de ne pas visiter deux fois la même tâche et permet le bon fonctionnement de l'algorithme.

3.2 Exemple d'application : le bonhomme de neige

Notre algorithme étant terminé, ne perdons pas le nord et testons le sur notre objectif initial, il nous suffit de rentrer les différentes tâches avec leurs listes de successeurs et on obtient le résultat ci-dessous

```
14 13 12 11 9 8 7 6 4 3 2 1 10 5 0
```

On obtient donc un enchaînement de tâches, et après un rapide coup d'œil à la figure 2, on s'assure de la validité de notre résultat. Plus qu'à s'exécuter, donc première étape construire le corps, houla ça va prendre du temps ça...

Quelle idée de construire un bonhomme de neige seul, ça va prendre une éternité, on va s'autoriser à être aidé par d'autres constructeurs. Et puis, un peu de réalisme, construire la tête ne prendra pas le même temps que de placer le bonnet, changeons un peu le problème.

Pour la suite, nous considérerons qu'il y a une infinité d'exécutables (vous avez vraiment beaucoup d'amis) et que les tâches ont des durées variables.

4 Deuxième partie : éxecution de tâches en parallèle

4.1 Implémentation de l'algorithme

On repart donc du même problème, à la différence près que plusieurs tâches peuvent être éxecutées simultanément en respectant toujours les contraintes des dépendances. on aura donc n tâches exécutées pour chaque pas de temps t_i,

$i \leq |T|, n \in \mathbb{N}$

Idée : On remarque, que pour le premier pas de temps t_1, sont executés en même temps toutes les tâches qui n'ont pas de prédécesseur (dans G). Puis, pour le deuxième temps t_2, toutes les tâches qui n'ont pas de prédecessurs (sans compter celles qu'on a prise au temps t_1)

On peut donc imaginer un algorithme qui récupère à chaque itération, les tâches qui n'ont pas de prédécesseurs et les retire du graphe. Problème, notre implémentation en liste d'adjacence permet l'accès aux successeurs mais pas aux prédecesseurs. On va donc devoir implémenter une fonction $invert()$ qui modifiera cette propriétée.

Algorithm 3 invert(G(T,A))

```
nG ← ∅
for t in T do
  if t not in nG then
    nG.insert(t)
  end if
  for succ in t.next() do
    if succ not in nG then
      nG.insert(succ)
    end if
    (succ.next()).push_back(t)
  end for
end for
return nG
```

On va pouvoir maintenant implémenter l'algorithme $multi\_process\_scheduling$, qui à partir d'un graphe G(T,A) retourne l'odre de l'éxecution. Pour la valeur de retour, on va avoir une liste de liste de tâches (une liste de tâche pour chaque temps t_i). Pour simplifier, on décide d'utiliser le champ visited des tâches (qui ne sert pas ici) et d'y stocker le temps t_i. On obtient donc l'algorithme suivant

Algorithm 4 $multi\_process\_scheduling(G(T, A))$

```
nG ← invert(G)
tmp ←[]
inc ← 0
for i = 1 to T.size() do
  tmp ←[]
  for t in T do
    if t.next() is empty then
      t.visited() ← i
      tmp.push_back(t)
      inc++
    end if
  end for
  for t in tmp do
    T.erase(t)
  end for
  if inc >= G.size() then
    break;
  end if
end for
return nG
```

4.2 Le bonhomme de neige : le retour

Avec cette deuxième version terminée, nous nous empressons de la tester sur notre construction de bonhomme de neige. On met les résultats dans un fichier par soucis de visibilité, et on obtient

```
1| 13 14
2| 0 1 2 3 4 8 9 11 12
3| 10 6 7
4| 5
```

On voit, par ailleurs, que l'on finit notre bonhomme de neige en 4 pas de temps, ce qui correspond en fait à la longueur du plus long chemin. On peut également dire que 9 constructeurs suffisent pour cette execution, cela correspond au nombre maximum de tâches exécutées en parallèle, qui se situe au temps 2.

5 Troisième partie : exécution de tâches à durées variables

Dans cette partie, nous allons considérer que chaque tâche à une durée d, $d \in \mathbb{N}^*$, nous allons donc voir comment modifier notre algorithme en conséquence, puis quels résultats nous obtenons.

5.1 Implémentation de l'algorithme

Idée : On repart de l'algorithme précédent (Algorithm 4), et on veut, au lieu de supprimer directement une tâche qui n'a pas de prédécesseurs, attendre d tours de boucle correspondant à sa durée d'exécution. On va donc avoir une boucle qui représentera le pas de temps, et à chaque pas de temps, on va prendre les tâches qui n'ont pas de prédécesseurs (et qui ne sont pas déjà prises, c'est-à-dire en cours d'exécution), puis on décrémente la durée de toutes les tâches prises, enfin on supprime celles dont la durée est égale à 0, on obtient donc :

Algorithm 5 $multi\_process\_schedulingu\_duration(G(T, A))$

```
nG ← invert(G)
tmp ←[]
inc ← 0
i ← 1
while tmp not empty or inc < T.size do
  for t in T do
    if t.next() is empty and t not in tmp then
      t.visited() ← i
      tmp.push_back(t)
      inc++
    end if
  end for
  for t in tmp do
    t.duration -= 1
    if t.dur<0 then
      tmp.erase(t)
      T.erase(t)
    end if
  end for
  i++
end while
return nG
```

5.2 Le bonhomme de neige : le final

Après avoir finalisé notre algorithme, il est temps de l'essayer sur notre problème.

Nous avons donc rajouté un booléen dans le fichier après le nombre de tâches. 0 pour signifier que les temps sont tous fixés à 1, 1 pour dire que les temps sont variables, et sont donc écrit après châque numéro de tâche. Cela donne donc :

```
15 1
0 1
1 5 10
2 2
3 2
4 2
5 3
6 2
7 2
8 1 6
9 1 7
10 1 5
11 1
12 1
13 7 0 2 3 4 10
14 9 1 8 9 11 12
```

Par exemple, on voit que l'on met un temps de 1 pour faire la t_0, c'est à dire placer le bonnet, et un temps de 9 pour faire t_{14}, c'est à dire construire le corps.
On obtient donc le résultat ci-dessous

```
1  début de 13 début de 14
2
3
4
5
6
7
8   fin de 13
9  début de 0   début de 2   début de 3   début de 4
10    fin de 14  fin de 0
11   début de 1   début de 8   début de 9   début de 11 début de 12  fin de 2    fin de 3    fin de 4
12    fin de 8    fin de 9    fin de 11  fin de 12
13   début de 6   début de 7
14
15    fin de 6    fin de 7
16    fin de 1
17   début de 10
18    fin de 10
19   début de 5
20
21
22    fin de 5
```

On finit donc notre bonhomme en 22 temps, on peut y voir
Nous avons enfin réussi à résoudre notre problème, et nous savons qu'il nous faut *insérer le temps final* pour faire notre bonhomme de neige. Il nous reste plus qu'à appeler nos amis à la rescousse et nous pourrons le construire.

6 Conclusion

En conclusion, l'ordonnancement des tâches est une question résolvable en théorie pour les cas simples avec une solution optimale.

Mais la question d'ordonnancement peut être une question difficile à résoudre dans beaucoup de cas. Par exemple, la durée d'une tâche peut ne pas être su, ou avec une marge d'erreur. Il peut aussi y avoir un ordre de priorité et des tâches qui ne sont exécutables qu'à partir d'un temps t, ce qui implique que l'algorithme doit couper l'exécution d'une tâche en plusieurs parties. Nous pouvons aussi imaginer que les ressources pour l'exécution d'une tâche sont limitées ou finies (c'est-à-dire que potentiellement, toutes les tâches ne pourront pas être exécutées).

La question de l'ordonnancement est un problème important dans quasiment tous les domaines et il est difficile voir impossible de créer un algorithme exhaustif tant les besoins et les problématiques des clients peuvent être différents.