Le prolème Min Makespan - Partie 2

Axel COUSSEAU, Minh-Hoang DANG

January 17, 2020

1 Complexité en temps

1.1 Génération des instances

1.1.1 Mode I_m

```
print("Saisissez m: ")
m = int(input())
tasks = np.concatenate((
    [m for i in range(3)], # 3 itérations
    np.repeat([m + i for i in range(1, m)], 2) # 2*m itération
))

print(tasks)
instance = np.concatenate(([m, len(tasks)], tasks))
```

Donc la génération de l'instance I_m prend $\mathcal{O}(3+2m)$ temps.

1.1.2 Mode I_R

```
instances = []
print("Saisissez n, m, k, min, max. Par exemple: 'n m k min max'")
n, m, k, low, high = [int(item) for item in input().split()]
print(n, m, k, low, high)
for i in range(k):
    instances.append( # append prend O(1) temps
        # Génération un array de n élémént prend O(n) temps
        np.concatenate(([m, n], np.random.random_integers(low, high, size=n)))
    )
```

Donc la génération de l'instance I_m prend $\mathcal{O}(kn)$ temps.

1.2 Copie en mémoire

La fonction suivante permet de transformer une chaîne de caractère (fournie par I_f ou I_c:

```
def getTokenIfValid(seq: str):
    """Parse l'entrée et transforme en array"""
```

```
    tokens = seq.split(":")
    m, n, tasks = int(tokens[0]), int(tokens[1]), tokens[2:len(tokens)]
    if(n != len(tasks)):
        print("Il n'y a pas assez de tâches, il en faut n(", n, ")")
        return None

    return tokens
```

La complexité dépend de la fonction `split` de Python, donc le code source est comme suit:

```
    while (maxcount-- > 0) {
        while (i < str_len && STRINGLIB_ISSPACE(str[i]))
            i++;
        if (i == str_len) break;
        j = i; i++;
        while (i < str_len && !STRINGLIB_ISSPACE(str[i]))
            i++;
```

On peut déduire facilement que `split` s'éxécute en $\mathcal{O}(s)$ temps, avec s étant la longueur de la chaîne de caractères en question. Finalemet, La copie en mémoire des instances prend $\mathcal{O}(s)$ temps.

1.3 Les algorithmes pour le problème Min-Makespan

1.3.1 LSA

```
def LSA(m: int, tasks: list):
    """Les tâches seront traités dans l'ordre tel qu'elles sont fournies"""

    M = dict([(i, []) for i in range(1, m+1)])

    for task in tasks: # n itération
        availIdx = 1
        loads = dict(map(lambda item: (item[0], sum(item[1])), M.items()))
        availIdx = min(loads, key=loads.get)
        M[availIdx].append(task)

    return M
```

Le mapping consiste à parcourir chacun de m machines puis faire une somme des durées. - d'_1: `loads` contient 1 tâche, - d'_2: `loads` contient 2 tâches - ... - d'_n: `loads` contient n tâches

A chaque itération on fait `min` et `sum` du nombre de tâches au courant. Le nombre d'itérations au total est donc 2 fois (2 opérations) la somme d'une suite arithmétique $(1, 2, ..., n)$, donc $\frac{2n(n+1)}{2} = n(n+1) = n^2 + n$. Au final, `LSA` prend $\mathcal{O}(n^2)$ temps.

1.3.2 LPT

```
def LPT(m: int, tasks: list):
    """Les tâches seront traités dans l'ordre décroissant de leur durée"""
```

```
    M = dict([(i, []) for i in range(1, m+1)])
    # Trier selon l'ordre décroissant, O(n*log(n)) mergesort
    tasks = np.sort(tasks, kind="mergesort")[::-1]
    #tasks = np.sort(tasks)[::-1] # O(n^2) quicksort par défaut

    for task in tasks:
        availIdx = 1
        loads = dict(map(lambda item: (item[0], sum(item[1])), M.items()))
        availIdx = min(loads, key=loads.get)
        M[availIdx].append(task)

    return M
```

`LPT` est juste une variation de `LST`, en ordonnant les tâches dans l'ordre décroissant, qui prend $\mathcal{O}(n^2)$ ou $\mathcal{O}(nlog(n))$ selon le choix de l'algorithme de tri. Au final, `LPT` prend $\mathcal{O}(n^2)$ temps.

1.3.3 MyAlgo

```
def MyAlgo(m: int, tasks: list):
    """Affecter les tâches en utilisant le principe du First-Fit Algorithm.
    On va essayer de faire m bins de capacité ~opt """

    M = dict([(i, []) for i in range(1, m+1)])
    opt = np.ceil(len(tasks)/m)
    tasks = tasks = np.sort(tasks, kind="mergesort")

    #
    macIdx = 1
    for task in tasks: # n itérations
        if(macIdx < m - 1):
            while(sum(M[macIdx]) <= opt):
                M[macIdx].append(task) # O(1)
            macIdx += 1
        else:
            M[macIdx].append(task)

    return M
```

Le mapping consiste à parcourir chacun de m machines puis faire une somme des durées. - d'_1: `loads` contient 1 tâche, - d'_2: `loads` contient 2 tâches - ... - d'_n: `loads` contient n tâches

A chaque itération on fait `sum` du nombre de tâches au courant. Le nombre d'itérations au total est donc la somme d'une suite arithmétique $(1, 2, ..., n)$, donc $\frac{n(n+1)}{2} = \frac{n^2+n}{2}$. Au final, `MyAlgo` prend $\mathcal{O}(n^2)$ temps.

2 Résultats

2.1 Comparaison du ratio LSA / LPT pour I_m

Théoriquement, nous avons trouvé que le ratio ne dépasse pas $\lim_{m\to\infty} \frac{4m-1+x}{3m} \approx \frac{4}{3}$.

L'exécution sur différents taille de m montre que le ratio converge vers 1.5 pour LSA et $\frac{4}{3}$ pour LPT quand m devient grand.

m	10	100	1000	10000
ratio LSA	1.433	1.493	1.499	1.5
ratio LPT	1.3	1.33	1.333	1.3333
ration MyAlgo	1.3	1.33	1.333	1.3333

2.2 Comparaison des performances des trois algorithmes

2.2.1 I_f et I_c

I_f et I_c sont la même chose pratiquement donc nous allons tester avec les données du TD (ex 3.2)

```
Choisissez le mode: 0 - fichier, 1 - clavier, 2 - I_m, 3 - I_R
0
Saisissez le nom du fichier
test.txt
Borne inférieure "maximum" =  7
Borne inférieure "moyenne" =  13.0
Résultat LSA =  15
Ratio LSA =  1.154
Résultat LPT =  13
Ratio LPT =  1.0
Résultat MyAlgo =  13
Ratio MyAlgo =  1.0
```

LPT et MyAlgo sont meilleurs que LSA.

2.2.2 I_R

Pour ce test, nous fixons le nombre de machines m et varions le nombre de tâches n:

n, $m = 10$	10	100	1000
ratio LSA	1.705	1.078	1.007
ratio LPT	1.705	1.007	1.001
ration MyAlgo	1.705	1.078	1.007

LPT est le plus efficace pour I_R

3 Conclusion

`MyAlgo`, en variant l'algorithme `First-Fit` est le plus rapide (les trois s'exécutent en $\mathcal{O}(n^2)$ mais `MyAlgo` prend 2 fois moins d'itérations que les deux autres. Pour les instances I_c, I_f et I_m, `LPT` et `MyAlgo` donnent le même ratio mais `MyAlgo` est préférable en temps d'éxécution. Pour I_R, `LPT` est préférable pour obtenir le meilleur ratio.

In [ ]: