

# Progetto di Ricerca Operativa

Filippo Landi

13 dicembre 2021

## Sommario

Il mio progetto per il corso di *Ricerca Operativa*.

## 1 Introduzione

Ho scelto il progetto numero 53 “**Single machine scheduling, weighted completion times with precedence constraints**”:

*“Sono dati  $n$  jobs  $J=\{j_1..j_n\}$  di cui è noto per ciascuno il tempo di esecuzione  $d^j$  e il peso  $w^j$  (non correlato con  $d^j$ ). Inoltre esistono delle regole di precedenza fra coppie di jobs descritte da un grafo aciclico. Il tempo di completamento del job  $j$ ,  $C^j$  è definito come l’istante di fine lavorazione del job. Ogni job, una volta iniziata la lavorazione, va portato a termine senza interruzioni. Si determini la sequenza di esecuzione dei jobs sull’unica macchina disponibile che minimizza la somma pesata dei tempi di completamento dei jobs nel rispetto delle precedenze.”*

Ho scritto il codice per risolvere il problema in Java.

## 2 Come usare il codice

Per usare il programma generando dei job con precedenze non cicliche casuali:

```
java JobMain -j [NUMERO]
```

Dove *NUMERO* è il numero dei job.

Per usare il programma caricando da file *csv* i job e le precedenze (guardare *jobs.csv* per un esempio):

```
java JobMain -f [FILE]
```

Dove *FILE* è il path a un file *csv*.

Per mostrare a terminale cosa succede ad ogni *turno*<sup>1</sup> di computazione:

```
java JobMain [OPZIONI] -debug
```

Di seguito spiego in maniera informale i vari ragionamenti e scelte fatte durante la progettazione: riporterò i punti più importanti, il codice sorgente stesso è ampiamente commentato e autoesplicativo. Buona lettura.

### 3 Prime osservazioni

Il problema di *single-machine scheduling*  $1||\sum_j w_j C_j$  visto alla slide 33 di *11a.EuristicheGreedy\_2021.ppt* è estremamente simile al mio: devo aggiungere le regole di precedenza tra coppie di job descritte da un *grafo diretto aciclico* e “il gioco è fatto”. Di seguito abbrevierò *grafo diretto aciclico* in *dag* (dall’inglese *direct acyclic graph*). Nella slide viene suggerito che quel problema si risolve all’ottimo con una greedy basata sul rapporto durata su peso in ordine non decrescente.

### 4 Chi va piano va sano e va lontano

Il codice finale è il frutto di una serie di raffinamenti:

1. Sono partito risolvendo il problema  $1||\sum_j C_j$ : il problema più semplice di *single-machine scheduling* riportato nelle slide che si risolve all’ottimo con una greedy detta *Shortest Processing Time*, che prende i lavori per durate non decrescenti.

Questo mi ha permesso di ragionare su:

- Una prima implementazione dei lavori in una classe *Job*.
- Una inizializzazione casuale dei job nel main con numero di istanze definito da argomento alla chiamata al programma.

---

<sup>1</sup>Leggere la sezione sull’implementazione del main per capire cosa intendo per *turno*.

- La realizzazione della greedy: un sorting per durata non decrescente.
2. Ho adattato il codice per risolvere il  $1||\sum_j w_j C_j$ , che si è rivelato quasi identico al precedente: ho aggiunto il rapporto durata su valore che chiamo *ratio* ai job e ho cambiato la sort considerando esso non decrescente.
  3. Ho adattato il codice per risolvere il problema delle precedenze.  
Qui ho ragionato su:
    - Come realizzare un *dag* casuale.
    - Come gestire le precedenze derivate dal *dag*.
  4. Infine per arricchire il progetto:
    - Ho implementato un metodo per passare i lavori e le precedenze tramite un (unico) file *.csv*.
    - Ho implementato la scrittura, ad ogni avvio del programma, di un file *.dot* che mostra graficamente le precedenze.

Andiamo più nel dettaglio dei vari componenti.

## 5 Implementazione dei lavori

I lavori sono stati realizzati nella classe *Job* in *Job.java* che implementa la classe *Comparable*, spiego poi il perché.

Ho deciso di realizzare gli attributi *span* (durata) e *value* (valore) come interi in quanto nelle lezioni erano tali, ovviamente si può cambiare avendo premura di modificare il necessario: ricordo che Java è fortemente tipato. Il *ratio* definito come *span/value*, necessario per il secondo problema, è necessariamente di tipo *Double* per avere i valori decimali. Ho anche aggiunto *index*: un indice di tipo intero per identificare i job.

La classe *Comparable* permette di definire il metodo *compareTo* che definisce un metodo per confrontare una collezione di oggetti, infatti l'insieme dei Job (*jobs*) sarà realizzato come collezione di oggetti (lista). Per esempio l'istruzione:

```
Collections.sort(LISTA);
```

riordina i *Job* secondo la politica di *compareTo*, questo si può usare per risolvere i problemi senza precedenze, mentre nel codice finale userò l’istruzione:

```
Collections.min(LISTA);
```

per identificare direttamente il *Job* col *ratio* minimo.

## 6 Realizzare un dag

Da una rapida ricerca online ho trovato che un grafo diretto è aciclico se esiste un ordinamento dei vertici che rende la sua matrice di adiacenza triangolare inferiore, la diagonale principale è nulla per non avere cicli sui vertici stessi (self-loops).[1]

L’idea è che i nodi hanno un grado e solo nodi di grado inferiore si possono connettere a nodi di grado superiore: creando un ordine si evitano cicli.

Anche alla slide n.48 di *09.Flussi1\_2021.ppt* del corso si parla di questo concetto: “Nei grafi aciclici è possibile dare una buona enumerazione ai nodi. Un grafo aciclico mappa una relazione di ordine parziale (antisimmetrica e transitiva) che si riflette nella buona enumerazione.”.

Realizzo quindi per il mio codice una matrice triangolare inferiore casuale con diagonale nulla.<sup>2</sup>

Faccio inoltre notare che al codice non interessa che la matrice abbia tale particolare forma, come spiegherò più avanti esso ricava semplicemente il *grado* (o *priorità*) dei job:

1. Se questi gradi rispettano i vincoli di un dag allora funziona tutto come deve, ed è sempre così con la mia matrice casuale (di cui spiego successivamente l’implementazione).
2. Se questi gradi non rispettano i vincoli, il programma non riuscirà a trovare un lavoro candidato e quindi terminerà dando errore (che gestisco scrivendo a schermo “errore di precedenze”): può capitare caricando da file dei lavori con precedenze errate.

---

<sup>2</sup>N.B.: in realtà come già detto mi posso ricondurre a questa matrice tramite un riordinamento dei vertici, non è detto che in generale abbia questa particolare forma (ad esempio caricando da file).

## 6.1 La matrice triangolare inferiore casuale

La matrice essendo triangolare è quadrata e avrà cardinalità dettata dal numero dei job.

Si può leggere questa matrice per righe dicendo:

**“Il job associato alla riga i aspetta il job associato alla riga j?”**

- Se c'è un 1 allora sì,
- se c'è un 0 allora no.

Ho implementato nella classe `RandomDagGenerator` un generatore per una matrice triangolare inferiore casuale, riga per riga:

- Prima della diagonale “lancia una moneta” per assegnare 0 o 1 (probabilità a 0.5 ma può essere cambiata).
- Dalla diagonale principale fino a fine riga assegna 0 (si potrebbe dire che fa un “zero-fill”).

## 7 Spiegazione di JobMain

Il main si trova in `JobMain` e fa molte cose. Spiego cosa fa per blocchi, si possono riconoscere guardando i commenti nel codice<sup>3</sup>:

1. Definizione di variabili: molte variabili sono dichiarate qui, altre dentro i blocchi che le usano esclusivamente: da quel che ho capito non è *best practice* fare così, sarebbe meglio sapere dall'inizio tutte le variabili usate nel programma, ma mi pareva troppo confusionario poi.
2. Gestione argomenti: qui controllo gli argomenti e setto alcuni boolean per eseguire le azioni richieste.
3. Istanziamento random: se è usato l'argomento `-j` creo i job e le precedenze in maniera random.
4. Istanziamento da file: se è usato l'argomento `-f` creo i job e le precedenze caricando da file.

---

<sup>3</sup>Nel codice ho commentato in maiuscolo le sezioni qui riportate.

5. Creazione del grafico *.dot*: creo il grafico *.dot* con la sua sintassi.
6. Stampo i lavori e la matrice.
7. Calcolo le priorità dei job: ciclo e conto il numero di job che ogni job aspetta, questo ne determina la *priorità* o *grado*:
  - I lavori con grado 0 sono quelli che non aspettano nessuno, quindi possono essere eseguiti.
  - I lavori con grado maggiore di zero aspettano qualcuno, quindi non possono essere eseguiti.
  - I lavori con grado -1 sono i lavori già eseguiti, così da non riprenderli in considerazione.
8. Creo la lista *ready* di lavori a grado 0.
9. Algoritmo principale: cicla per il numero di lavori (li deve eseguire tutti), questi sono i *turni*:
  - (a) Aggiorna la lista dell'*ordine di esecuzione migliore* aggiungendo l'indice del job col ratio minore.
  - (b) Rimuove il job scelto dalla lista *ready*.
  - (c) Aggiorna le priorità: il job stesso va a -1 e viene diminuito il grado di chi lo aspettava.
  - (d) Aggiunge i nuovi lavori grado 0 alla lista *ready*.
  - (e) Ritorna al primo step (fino a fine ciclo).
10. Restituisco l'ordine di esecuzione migliore dei job.
11. Stampo i tempi di completamento dei job.

## 8 Commenti ulteriori

### 8.1 Caricamento da file

Come detto ho aggiunto la capacità di caricare da file *csv* dei lavori, basta scrivere i vari campi separati da virgole: il nome, la durata, il valore e le varie precedenze (queste separate da “;”). Come accennato bisogna avere premura che le precedenze siano un grafo aciclico, se no il codice arriva ad un punto che non trova nessun job candidato e termina l'esecuzione.

## 8.2 Il grafico .dot

Per vedere il grafico bisogna avere qualche programma capace di interpretare il linguaggio dot: io uso *XDot*, ma ci sono anche utility online. Il grafico, oltre ad essere carino, può aiutare ad individuare cicli in caso le precedenze non rispettino i vincoli propri di un *dag*.

## 8.3 Il modello matematico

Non ci ho pensato molto quindi spero che il mio ragionamento sia corretto: direi che si può creare prendendo il modello del problema 1||  $\sum_j w_j C_j$  visto alla slide 33 di *11a.EuristicheGreedy-2021.ppt* e aggiungendo dei vincoli come disequazioni sui tempi di esecuzione dei job che ne limitino l'esecuzione dopo i job a cui devono dare la precedenza, stile:  $t_{j_1} \geq t_{j_2} + d_{j_2} \forall j_1, j_2 \in J$  (come viene fatto coi *task* alla slide n.90 di *09.Flussi1-2021.ppt*). Credo quindi serva anche un lavoro finto  $j_0$  che esegua al tempo 0, di durata 0 e quindi *ratio* 0.

## Riferimenti bibliografici

[1] URL: <https://mathematica.stackexchange.com/a/613>.