

Progetto di Ricerca Operativa

Filippo Landi

20 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 che fornisce una soluzione ammissibile in Java, si trova su GitHub al seguente indirizzo:

<https://github.com/Cippo95/progetto-ro>

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*¹ 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:

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

- 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.
 - 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.²

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.

²N.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 codice³:

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 (durata e valore sono numeri interi random tra 1 e 100).
4. Istanziamento da file: se è usato l'argomento -f creo i job e le precedenze caricando da file.

³Nel 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 eseguire.
 - 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 Alcuni commenti

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*.

Se interessati al linguaggio dot:

[https://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language))

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.

8.4 Qualità della soluzione

La soluzione fornita dal mio programma attraverso la greedy è non ottima, viene sempre data la precedenza ai job a ratio minore, non considerando i job nelle precedenze: potrebbero essere presenti dei job molto buoni che aspettano dei job *ready* con ratio peggiore rispetto agli altri dello stesso turno. Avrebbe quindi senso in certi casi far eseguire dei job peggiori del migliore in un dato turno: bisognerebbe valutare le qualità dei job che stanno aspettando quelli correnti.

Sono stati dimostrati metodi efficienti che risolvono il problema all'ottimo in caso che questo abbia precedenze particolari, ad esempio delle catene o degli alberi[2].

In caso di precedenze casuali, senza una struttura particolare, ma sempre acicliche per garantire l'ammissibilità, il problema è stato dimostrato *NP-Hard*[2][3].

Per avere soluzioni migliori quindi dovrei irrobustire la mia greedy o applicare delle metaeuristiche, ma al tempo dell'esame mi sono fermato alla greedy.

9 Lavori post-esame

Nei paper che letto (velocemente) fino ad ora, se ho capito bene venivano esplorati casi di euristiche più o meno robuste: non venivano affrontate metaeuristiche.

Inizialmente ragionandoci su, una ricerca locale sulla soluzione greedy o su una soluzione con randomicità nella scelta dei job *ready* (in stile *GRASP*) non mi ispirava troppo.

Come possibile intorno alla ricerca mi immaginavo un semplice swap tra lavori e non credevo portasse a molto. Ho testato questo sulla soluzione greedy, per esempio usando l'esempio di caso pessimo (file che riporto nel repository) e non cambiava nulla, perché questo intorno, che io esploro esaurivamente, non gode della proprietà di raggiungibilità. Magari in casi diversi avrebbe funzionato ma in questo semplice caso... no.

Continuando a ragionare, ho notato che gli algoritmi genetici mi ispiravano per questo problema. Prima di partire con l'implementazione ho fatto qualche ricerca e ho trovato uno studio recente a riguardo[4].

Il problema mostrato nel paper è estremamente simile al mio e mi ha ispirato a continuare il progetto... a questo punto per sfizio personale.

L'algoritmo genetico che ho implementato si ispira a quanto fatto nel paper, consiglio di leggere il paper velocemente per cogliere le differenze:

1. Crea la popolazione iniziale con l'individuo generato dalla greedy e il resto random (buona enumerazione).
2. Seleziona due genitori con *binary tournament scheme*: se ho capito bene si scelgono a caso due genitori (e l'ho implementato così).
3. Genera un figlio applicando il crossover:
 - Il crossover *ad un punto*: eredito la prima parte dal padre e il resto dei lavori per *posizione relativa* (come fatto a lezione nella GA per il TSP).
 - Dopo il crossover controllo l'ammissibilità.
 - Nel paper viene anche fatta una mutazione, io la salto (scelta sicuramente opinabile).
4. Se il figlio è ammissibile valuta la fitness del figlio, in caso contrario riparto dallo step 2.

5. Applica una ricerca locale sul figlio: la faccio esaustiva di tipo swap e la faccio più volte seguendo una filosofia simile alla *Variable Depth Search*. . . questo step è sicuramente estremamente pesante (a seconda dei parametri messi).
6. Sostituisce il peggiore membro della popolazione col figlio (se è effettivamente peggiore del figlio).
7. In caso il figlio sia il migliore della popolazione aggiorna la migliore soluzione.
8. Se uno dei criteri di stop è raggiunto ritorna la miglior soluzione altrimenti ritorna allo step 2: i criteri di stop sono un tot di generazioni a priori (nel paper sono $100 \times \text{numero di lavori}$) o tot generazioni senza miglioramento (nel paper $10 \times \text{numeri di job}$), quest'ultimo lo devo ancora implementare.

Sto ottenendo buoni risultati dove la soluzione greedy alle volte viene migliorata anche di molto.

Devo però fare refactoring del codice, scegliere bene alcuni parametri e controllare che tutto funzioni bene.

I tempi di calcolo per l'algoritmo genetico sono estremamente più lunghi dell'algoritmo greedy a seconda dell'istanza (e dei parametri da me impostati).

Ho dovuto ristrutturare parecchio il codice, infatti questa nuova versione si troverà in *JobMainGA.java*.

Aggiornerò i file di progetto (e probabilmente anche questo documento) su GitHub una volta finito. . . direi in un paio di giorni.

10 Dove trovare i paper

Spesso i paper si trovano su siti che richiedono di creare un account o pagare. . . cercando bene ci sono siti che li rendono pubblicamente consultabili e scaricabili.

Paper su 2-machine scheduling, ma parla ampiamente del problema di single machine scheduling, degli algoritmi ottimi per precedenze particolari e altro[2]:

https://www.researchgate.net/publication/225108781_An_Experimental_Study_of_Algorithms_for_Weighted_Completion_Time_Scheduling

Paper che spiega NP-Hardness dei problemi come il mio[3]:

<https://econpapers.repec.org/paper/agseureia/272176.htm>

Paper sull'uso di algoritmi genetici a problemi simili al mio[4]:

https://www.researchgate.net/publication/261035802_Genetic_local_search_algorithm_for_minimizing_the_total_completion_time_in_single_machine_scheduling_problem_with_release_dates_and_precedence_constraints

Riferimenti bibliografici

- [1] URL: <https://mathematica.stackexchange.com/a/613>.
- [2] Ivan Baev, Waleed Meleis e Alexandre Eichenberger. “An Experimental Study of Algorithms for Weighted Completion Time Scheduling”. In: *Algorithmica* 33 (mag. 2002), pp. 34–51. DOI: 10.1007/s00453-001-0103-x.
- [3] J. K. Lenstra e A. H. G. Rinnooy Kan. *COMPLEXITY RESULTS FOR SCHEDULING CHAINS ON A SINGLE MACHINE*. 1979. URL: <https://EconPapers.repec.org/RePEc:ags:eureia:272176>.
- [4] Ichraf Zaidi, Mehdi Mrad e Talel Ladhari. “Genetic local search algorithm for minimizing the total completion time in single machine scheduling problem with release dates and precedence constraints”. In: *2013 5th International Conference on Modeling, Simulation and Applied Optimization, ICMSAO 2013* (apr. 2013), pp. 1–4. DOI: 10.1109/ICMSAO.2013.6552617.