



UNIVERSITÀ DI PERUGIA
Dipartimento di Matematica e Informatica



LAUREA MAGISTRALE IN INFORMATICA
CORSO DI COMPUTATIONAL INTELLIGENCE

Ant Colony Optimization for Open Shop Scheduling Problem

Professore

Prof. Marco Baiocchi

Studente

Cristian Cosci
(Matricola: 350863)

Anno Accademico 2021-2022

Indice

1	Descrizione e implementazione del problema	3
1.1	Open Shop Scheduling Problem (OSSP)	3
1.2	Ant Colony Optimization (ACO)	5
1.2.1	Pseudocodice dell'algoritmo	6
1.3	Rappresentazione dell'OSSP	10
1.3.1	La funzione obiettivo	10
1.4	Funzione euristica	10
1.5	Ottimizzazione soluzione utilizzando Local Search	11
1.6	Tuning dei Parametri	11
1.6.1	Alpha e Beta	11
1.6.2	w_ib e w_bs	12
1.6.3	rho	12
1.6.4	n_ants e num_gen	12
1.7	Risultati ottenuti su istanze di prova	13
1.7.1	Risultati su Istanze di dimensione 4 x 4	14
1.7.2	Risultati su Istanze di dimensione 5 x 5	14
1.7.3	Risultati su Istanze di dimensione 7 x 7	14
1.8	Ulteriori Informazioni	14

Capitolo 1

Descrizione e implementazione del problema

Il progetto implementato ha l'obiettivo di risolvere il problema dell'**Open Shop Scheduling (OSSP)** utilizzando l'algoritmo di ottimizzazione **Ant Colony Optimization**.

1.1 Open Shop Scheduling Problem (OSSP)

Open-shop scheduling o Open-shop scheduling problem (OSSP) è un problema di ottimizzazione nell'informatica e nella ricerca operativa. È una variante dello scheduling ottimale dei job.

In un problema generale di scheduling, ci vengono dati n job J_1, J_2, \dots, J_n con tempi di elaborazione variabili, che devono essere eseguiti su m macchine m_1, m_2, \dots, m_m , mentre si cerca di ridurre al minimo il **makespan** - la lunghezza totale dello scheduling (ovvero, tempo impiegato da tutti i job per essere eseguiti).

Nella variante specifica nota come open-shop scheduling, ogni lavoro è costituito da un insieme di operazioni (detti anche task) O_1, O_2, \dots, O_m le quali devono essere elaborate in un ordine arbitrario (non ci sono vincoli di ordine di esecuzione tra i task appartenenti allo stesso job) ciascuna in una macchina specifica.

L'input per l'OSSP consiste in un **insieme di n job (ciascuno costituito da m task)**, un **altro insieme di m workstation (macchine)** e una **tabella bidimen-**

sionale contenente la quantità di tempo che ciascun lavoro deve trascorrere su ciascuna workstation, il quale corrisponde al costo necessario ad eseguire un dato task sulla macchina indicata (questo perchè ogni task ha una macchina specifica in cui deve essere eseguito). Un esempio di istanza di input è disponibile nella Figura 1.1.

	Task_0		Task_1		Task_2		Task_3	
JOB	Machine	Cost	Machine	Cost	Machine	Cost	Machine	Cost
Job_1	1	10	3	4	0	8	2	11
Job_2	3	7	2	11	1	5	0	8
Job_3	0	8	1	12	2	13	3	6
Job_4	2	12	0	5	3	3	1	7

Figura 1.1: Esempio di istanza di input per OSSP con 4 job e 4 macchine. Ogni coppia Machine-Cost indica il costo di esecuzione del task (associato al job) per essere eseguito sulla macchina indicata.

Ciascun job può essere elaborato solo su una workstation alla volta e ciascuna workstation può elaborare un solo job alla volta.

Tuttavia, a differenza del problema job shop classico, l'ordine in cui avvengono le fasi di lavorazione (esecuzione dei task) è libero. L'obiettivo è assegnare un tempo t ad ogni task, il quale indica l'istante in cui quest'ultimo inizierà ad essere elaborato dalla workstation incaricata, in modo che:

- non vengano assegnati due job alla stessa workstation contemporaneamente,
- nessun job sia assegnato a due workstation contemporaneamente e
- ogni job sia assegnato a ciascuna postazione per il tempo desiderato.

La misura abituale della qualità di una soluzione è il suo makespan, la quantità di tempo dall'inizio dello scheduling (la prima assegnazione di un job a una workstation) alla sua fine (il tempo di fine esecuzione dell'ultimo job sull'ultima workstation).

Il problema dell'open-shop scheduling può essere risolto in tempo polinomiale per istanze che hanno solo due workstation o solo due job. Può anche essere risolto in tempo polinomiale quando tutti i tempi di elaborazione, diversi da zero, sono uguali:

in questo caso il problema diventa equivalente alla colorazione di un grafo bipartito che ha i job e le macchine come suoi vertici, e che ha un arco per ogni coppia job-macchina che ha un tempo di elaborazione diverso da zero. Il colore di un arco nella colorazione corrisponde al segmento di tempo in cui è pianificata l'elaborazione di una coppia job-macchina.

Per tre o più macchine, o tre o più job, con tempi di elaborazione variabili, il problema di scheduling open-shop è NP-hard.

1.2 Ant Colony Optimization (ACO)

Nell'informatica e nella ricerca, l'Ant Colony Optimization (ACO) è una tecnica probabilistica per la risoluzione di problemi computazionali che possono essere ridotti alla ricerca su grafo.

ACO è uno dei migliori algoritmi di ottimizzazione basato su Swarm Intelligence ed è ispirato al comportamento delle formiche.

L'idea è quella di simulare una colonia di formiche con lo scopo di risolvere problemi di ottimizzazione.

Nel mondo naturale, le formiche di alcune specie (inizialmente) vagano in modo casuale e, dopo aver trovato il cibo, tornano alla loro colonia mentre rilasciano tracce di feromone. Se altre formiche trovano un percorso del genere, è probabile che non continuino a viaggiare a caso, ma piuttosto che seguano il percorso, e che lo rafforzino se alla fine trovano cibo.

Col tempo, tuttavia, la scia di feromone inizia ad evaporare, riducendone così la forza attrattiva. Più tempo impiega una formica per viaggiare lungo il sentiero e tornare indietro, più tempo ha il feromone per evaporare. Un cammino breve, in confronto, viene percorso più frequentemente, e quindi la densità dei feromoni diventa maggiore sui percorsi più brevi rispetto a quelli più lunghi. L'evaporazione del feromone ha anche il vantaggio di evitare la convergenza verso una soluzione localmente ottimale. Se non ci fosse alcuna evaporazione, i percorsi scelti dalle prime formiche tenderebbero ad essere eccessivamente attraenti per le successive. In tal caso, l'esplorazione dello spazio delle soluzioni sarebbe vincolata. L'influenza dell'evaporazione dei feromoni nei sistemi di formiche reali non è chiara, ma è molto importante nei sistemi artificiali. Il risultato complessivo è che quando una formica trova un buon (cioè breve) percorso

dalla colonia a una fonte di cibo, è più probabile che altre formiche seguano quel percorso e un feedback positivo alla fine porta molte formiche a seguire un unico percorso. L'idea dell'algoritmo della colonia di formiche è di imitare questo comportamento con "formiche simulate" che camminano attorno al grafo che rappresenta il problema da risolvere.

Negli algoritmi di ottimizzazione delle colonie di formiche, una formica artificiale è un semplice agente computazionale che cerca buone soluzioni per un dato problema di ottimizzazione. Per applicare un algoritmo di colonia di formiche, il problema di ottimizzazione deve essere convertito nel problema di trovare il percorso più breve su un grafo pesato.

Nella prima fase di ogni iterazione, ogni formica costruisce stocasticamente una soluzione, ovvero l'ordine in cui devono essere seguiti gli archi nel grafo. Nella seconda fase vengono confrontati i percorsi trovati dalle diverse formiche. L'ultimo passaggio consiste nell'aggiornare i livelli di feromoni su ciascun arco.

1.2.1 Pseudocodice dell'algoritmo

Algorithm 1 Ant Colony Optimization Pseudocode

```

1: initialize pheromone matrix
2: for  $g \leftarrow 1$  to max_gen do
3:   for  $i \leftarrow 1$  to num_ants do
4:      $s_i \leftarrow \text{create\_solution}()$ 
5:   end for
6:    $\text{pheromone\_evaporation}()$ 
7:    $\text{update\_best\_solution}()$ 
8: end for
9: return best solution ever found

```

Inizializzazione

La fase di inizializzazione per la matrice dei ferormoni è abbastanza semplice: all'inizio si ha lo stesso valore τ_0 (valore iniziale per il pheromone) per tutti gli archi, e le formiche scelgono solo in base all'euristica θ (il feromone all'inizio non ha nessun contributo nella scelta delle formiche).

create_solution()

Algorithm 2 create_solution()

Ogni formica parte da un percorso vuoto π
while $|\pi| < n$ **do**
 seleziona un task c_j non appartenente al percorso π
 aggiungi c_j a π
end while
aggiungi il costo del nuovo arco al costo totale del cammino π
return π

Come fa la formica a scegliere un percorso?

La scelta del prossimo nodo da visitare è una scelta probabilistica ed è influenzata da due valori:

- **Feromone** τ_{ij} ("tau")
- **Funzione euristica** θ_{ij} ("teta")

dove i il nodo corrente (ultimo task nel cammino) e j è il task successivo.

I valori di feromone sono memorizzati in una matrice che è gestita dalla colonia: τ_{ij} rappresenta quanto la colonia ritiene buono passare da un task i ad un task j (è un numero reale. In genere più è alto e più la colonia ritiene buono questo arco).

Il valore di euristica è un'informazione esterna che valuta la bontà della scelta e la sua formulazione **dipende dal problema che si sta cercando di risolvere**.

Probabilistic Choice: La probabilità di scegliere j come task successivo del percorso è data dalla seguente formula:

$$p_{i \rightarrow j} = \frac{\tau_{ij}^\alpha \theta_{ij}^\beta}{\sum_{k \in NV} \tau_{ij}^\alpha \theta_{ij}^\beta} \text{ for } j \in NV \quad (1.1)$$

dove NV è l'insieme dei task non ancora presenti nel cammino (non ancora nella soluzione).

Per scegliere un numero basandosi sulle probabilità si fa utilizzo della tecnica della **roulette wheel**.

Il valore probabilistico non esiste in natura ma viene aggiunto per migliorare le pre-

stazioni.

La scelta del prossimo task (in base al valore probabilistico) avviene in due passaggi:

1. calcolare $p(i \rightarrow j)$ per tutti task non ancora visitati
2. selezionare un task tra essi con il metodo della roulette wheel

Evaporazione

L'evaporazione è simulata nel seguente modo:

$$\tau_{ij} \leftarrow (1 - \rho) \forall i, j \quad (1.2)$$

ρ (rho) è un valore piccolo.

Esempio: $\rho = 0.05$; $1-\rho = 0.95$ -> è diminuito del 5% .

L'evaporazione è una sorta di decadimento esponenziale del feromone.

Aggiornamento

L'aggiornamento non esiste in natura. Questo processo da una piccola reward (ri-compensa) ad ogni coppia c_i, c_j che appare in buone soluzioni.

Quali sono le buone soluzioni?

In natura se ne distinguono 2:

1. **Best_So_Far** -> è la migliore soluzione di sempre s_{bs} (sarà anche il risultato finale restituito dall'algoritmo)
2. **Iteration_Best** -> è la migliore soluzione trovata in questa generazione s_{ib}

Algorithm 3 rewards(s, k)

- 1: **for all** c_i, c_j che appaiono in s **do**
 - 2: $\tau_{ij} \leftarrow \tau_{ij} + K/Ls$
 - 3: **end for**
-

Ls è il costo della soluzione s.

Ci sono 3 possibilità:

1. $\text{reward}(s_{bs}, w_{bs})$
2. $\text{reward}(s_{ib}, w_{ib})$
3. $\text{reward}(s_{bs}, w_{bs})$ e $\text{reward}(s_{ib}, w_{ib})$

w_{bs} e w_{ib} sono pesi che dipendono dal problema e devono essere calibrati. Ci sono anche varianti in cui non viene premiata una sola soluzione ma ad esempio anche la seconda migliore ecc...

I parametri di ACO

- α -> indica quanto è importante τ (ferormone) nella scelta delle formiche
- β -> indica quanto è importante θ (valore euristico) nella scelta delle formiche
- ρ -> indica quanto velocemente evapora il feromone (influenza l'evaporazione: tanto più è alto, tanto più velocemente evapora il feromone)
- n_ants (numero di formiche) -> può influenzare la bontà dell'Iteration_Best: più è alto il numero di formiche e più tentativi vengono fatti ad ogni iterazione. Facendo più tentativi è più probabile che vengano fuori soluzioni migliori ad ogni generazione (solitamente sta tra 20 e 50).
- w_{ib} -> indica quale soluzione viene premiata maggiormente: "premiare l'Iteration_Best significa premiare le novità" -> **Exploration**
- w_{bs} -> indica quale soluzione viene premiata maggiormente: "premiare il Best_So_Far significa spingere verso una convergenza" -> **Exploitation**
- max_gen

La matrice dei feromoni rappresenta la memoria collettiva della colonia.

La ricompensa non è della soluzione in sé, ma è data alle componenti della soluzione (archi/edges).

I valori euristici aiutano le formiche a scegliere, soprattutto all'inizio, perchè all'inizio il feromone non dà contributo.

1.3 Rappresentazione dell'OSSP

Una delle parti fondamentali, necessaria all'implementazione, è la definizione di un'opportuna rappresentazione per il problema di scheduling Open-Shop.

Dovendo ACO lavorare con una rappresentazione a grafo, la scelta più ovvia è stata quella di rappresentare il problema come un **grafo diretto completamente connesso**. I vari task compongono i vertici e un cammino è dato da una sequenza di task da eseguire in tale ordine. Spetterà poi all'algoritmo trovare il cammino migliore (senza ripetizioni), tale per cui si ha il makespan minore.

1.3.1 La funzione obiettivo

La definizione della funzione obiettivo deve rispettare la rappresentazione del problema e ha il compito di restituire il makespan (costo dello scheduling) data una soluzione.

Il funzionamento della funzione obiettivo consiste nell'andare ad assegnare (secondo l'ordine dato dal cammino nel grafo) i task alla rispettiva macchina, andando ovviamente a rispettare tutti i vincoli imposti dal problema, come:

- una macchina può eseguire solo un task alla volta
- un job può essere assegnato ad una sola macchina alla volta.

Nel caso in cui nel cammino sia presente un arco che collega due task dello stesso job, oppure vi sia un task assegnato ad una macchina che ne sta ancora processando un altro, il task verrà messo in attesa di esecuzione finchè la macchina non sarà libera.

1.4 Funzione euristica

La scelta della funzione euristica, utilizzata per calcolare la probabilità che ha ciascun task di essere scelto come successivo nella soluzione, dipende dal problema che si sta risolvendo. In questo caso il valore euristico θ per ciascun nodo è dato da $1/Ctime_j$ dove $Ctime_j$ è il tempo necessario ad eseguire il task j .

1.5 Ottimizzazione soluzione utilizzando Local Search

Per migliorare ulteriormente lo scheduling ottenuto utilizzando ACO, si è deciso di applicare un ulteriore algoritmo di ricerca locale sulla soluzione come la **Local Search**. La **Local Search** è un algoritmo utilizzato per problemi di ottimizzazione discreta. In particolare, partendo da un elemento dello spazio di ricerca X del problema, si procede esplorando il suo vicinato. Per esplorare il vicinato di un elemento si esegue una perturbazione su quest'ultimo. Per **perturbazione** si intende una piccola modifica che permette di spostarsi leggermente sullo spazio di ricerca. Questa operazione dipende necessariamente dal tipo di problema che si sta risolvendo, nel caso dell'OSSP la perturbazione consiste in una swap tra due elementi dello scheduling. L'algoritmo utilizzato è la Iterated Local Search con a base la Local Search Best Improvement. Questo tipo di ottimizzazione locale permette di ottenere buoni risultati.

1.6 Tuning dei Parametri

È ovviamente necessario un tuning dei parametri per l'algoritmo ACO. In particolare, i parametri con principale influenza sul funzionamento dell'algoritmo sono:

1. **alpha** e **beta**;
2. **w_ib** e **w_bs**;
3. **rho**;
4. **num_ants** e **num_gen**.

1.6.1 Alpha e Beta

Una scelta comunemente adottata è quella di limitare e rendere uguale i due parametri. In questo modo le formiche sono equamente influenzate nella scelta del cammino dal feromone che dall'euristica.

Si possono notare alcune differenze nella qualità della soluzione trovata dall'algoritmo nel caso in cui questi due valori vengano variati, tuttavia, non è stato trovato un

particolare setup in cui si hanno performance mediamente migliori. Si è quindi deciso di assegnare un valore di 1 sia per alpha che per beta.

1.6.2 **w_ib** e **w_bs**

I parametri **w_ib** e **w_bs** indicano il reward assegnato (in incremento in termini di feromone) alla miglior soluzione trovata all'iterazione corrente e alla miglior soluzione in generale. Con questi parametri è quindi possibile gestire il rapporto **exploration/exploitation** dell'algoritmo. I migliori risultati sono ottenuti utilizzando un rapporto di $\frac{3}{5}$ per **w_ib** e $\frac{1}{5}$ per **w_bs**. In questo modo si tende a favorire la fase di exploration portando l'algoritmo a convergere più lentamente. Per andare a raffinare ulteriormente il rapporto è possibile cambiare i valori utilizzando un numero superiore a 5 per il denominatore. Tuttavia, per questa fase di test si è deciso di procedere con questa semplice configurazione.

1.6.3 **rho**

Il parametro **rho** definisce la velocità di evaporazione del feromone sui cammini già percorsi dalle formiche. Nel caso di un valore molto piccolo di rho l'algoritmo potrebbe non convergere se il numero di generazioni non è sufficientemente grande. Nel caso in cui il valore **rho** è molto grande si tende a forzare una convergenza veloce (scartando ipotetiche buone soluzioni), la quale porta a scarsi risultati.

1.6.4 **n_ants** e **num_gen**

I parametri **n_ants** e **num_gen** permettono di gestire il numero di formiche della colonia e il numero di generazioni eseguite. In questo caso si è rivelato necessario gestire il valore principalmente in base alla dimensione dell'istanza da risolvere, prestando attenzione all'aumento della complessità temporale dell'algoritmo. All'aumentare del valore per questi due parametri aumenta il tempo necessario a terminare l'algoritmo di ricerca ma le soluzioni trovate sono migliori per istanze relativamente grandi (mentre per le istanze piccole non risulta necessario avere una colonia e un numero di generazioni molto grande). Come precedentemente riportato, il valore di **n_ants** è posto tra 20 e 50, mentre il numero di generazioni è stato posto in proporzione alla

dimensione dell'istanza (da 400 a 600). Tuttavia, per le istanze 7x7 si è deciso di limitare il numero di formiche a 10 in quanto il tempo richiesto a terminare l'algoritmo risulta eccessivo (nonostante con un numero maggiore si sarebbero potuti ottenere risultati migliori). Per quanto riguarda le istanze 4x4 e 5x5, non è stato necessario aumentare il numero di formiche oltre le 10, in quanto le soluzioni restituite sono state molto buone anche utilizzando questo valore.

1.7 Risultati ottenuti su istanze di prova

In questa sezione sono riportati i risultati ottenuti con molteplici istanze di **Open Shop Scheduling**, di varia dimensione, pubblicate nel lavoro di *E. Taillard*, "*Benchmarks for basic scheduling problems*" presente al seguente link <http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html> e descritte nel seguente articolo <http://mistic.heig-vd.ch/taillard/articles.dir/Taillard1993EJOR.pdf>. Nelle tabelle successive sono riportati i parametri utilizzati in base alla dimensione dell'istanza e successivamente i risultati restituiti dall'implementazione di ACO appena descritta.

Nelle tabelle dei risultati è presente la miglior soluzione trovata nel lavoro di Taillard e i migliori risultati ottenuti utilizzando ACO in combinazione alla Local Search (oltre al tempo necessario per terminare l'esecuzione).

Parametri ACO			
	4 x 4	5 x 5	7 x 7
alpha	1	1	1
beta	1	1	1
rho	0.5	0.5	0.5
tau	0.05	0.04	0.05
w_ib	3/5	3/5	3/5
w_bs	1/5	1/5	1/5
n_ants	10	10	10
num_gen	400	500	600

1.7.1 Risultati su Istanze di dimensione 4 x 4

Istanze 4 x 4				
istanza	upper bound	ACO results	ACO+LS	time
<i>ta44_1</i>	193	202	193	<i>19.26s + 24.7s</i>
<i>ta44_2</i>	236	253	236	<i>29.85s + 36.0s</i>
<i>ta44_3</i>	271	274	272	<i>39.35s + 40.1s</i>
<i>ta44_4</i>	250	250	250	<i>30.96s + 70.4s</i>
<i>ta44_5</i>	295	295	295	<i>45.46s + 59.2s</i>

1.7.2 Risultati su Istanze di dimensione 5 x 5

Istanze 5 x 5				
istanza	upper bound	ACO results	ACO+LS	time
<i>ta55_1</i>	300	347	318	<i>146.0s + 132s</i>
<i>ta55_2</i>	262	285	268	<i>101.0s + 145s</i>
<i>ta55_3</i>	328	360	344	<i>146.6s + 106s</i>
<i>ta55_4</i>	310	333	333	<i>158.6s + 182s</i>

1.7.3 Risultati su Istanze di dimensione 7 x 7

Istanze 7 x 7				
istanza	upper bound	ACO results	ACO+LS	time
<i>ta77_1</i>	438	551	451	<i>959s + 2245s</i>
<i>ta77_2</i>	449	550	468	<i>773s + 2295s</i>

1.8 Ulteriori Informazioni

Il codice del progetto è liberamente disponibile nella seguente repository GitHub https://github.com/CristianCosci/Ant_Colony_Optimization_for_OSSP.

Oltre al progetto stesso, sono disponibili degli script per creare delle istanze di input per l'OSSP ricavandole da questo sito <http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html>.

È possibile utilizzare anche un ottimizzatore dei parametri, il quale permette di trovare la combinazione ottimale di parametri con cui è possibile raggiungere l'upper bound noto per una data istanza (è ovviamente necessario conoscere a priori l'upper bound).

I parametri ottenuti possono poi essere utilizzati nello script principale **main.py** e vedere tutte le informazioni relative allo scheduling trovato, come ad esempio il gantt e il costo del makespan (vedi Figura 1.2). È anche possibile visualizzare il processo di ricerca dell’algoritmo ACO visualizzando un grafico in cui sono riportati i valori di media, massimo e minimo delle soluzioni trovate dalle formiche ad ogni generazione.

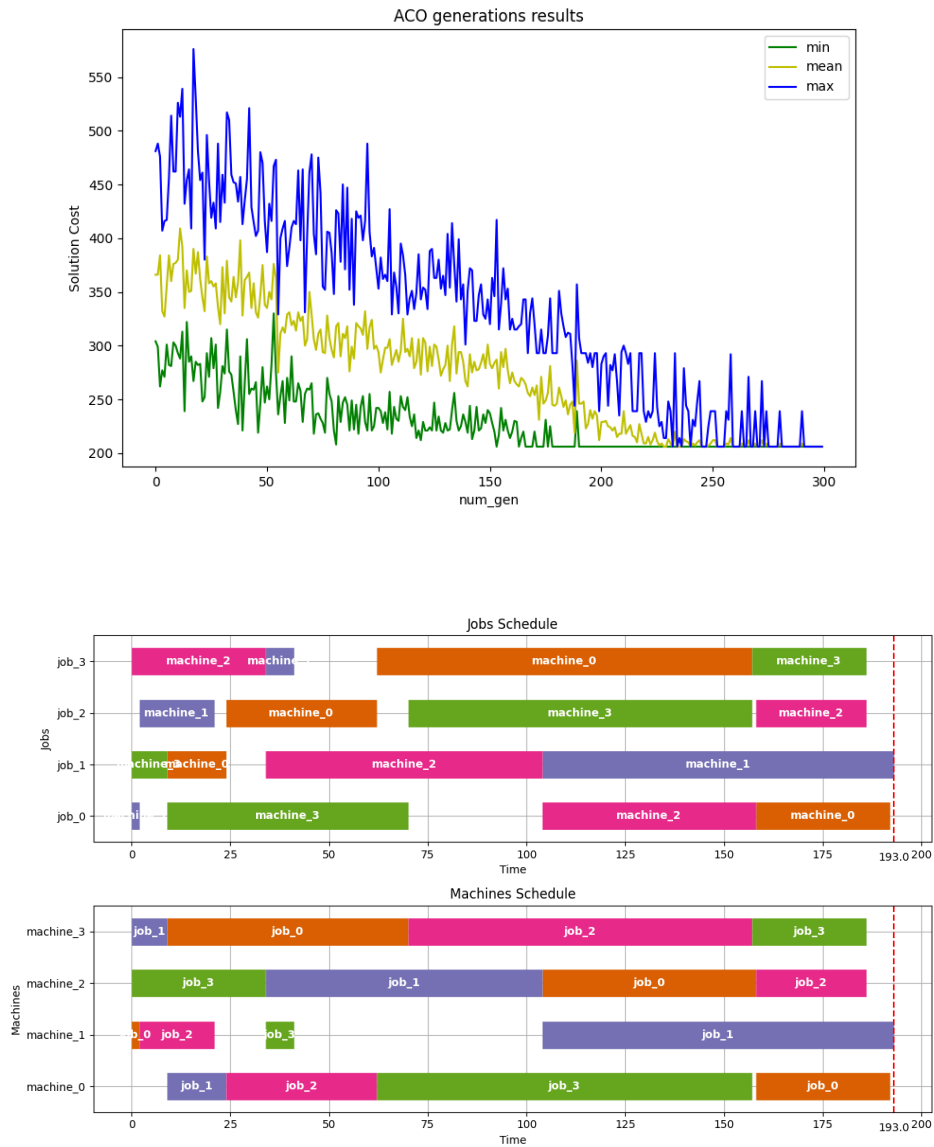


Figura 1.2: Gantt per un’istanza di input data da 4 job e 4 macchine.