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 Baioletti

Studente
Cristian Cosci
(Matricola: 350863)

Anno Accademico 2021-2022

Indice

Capitolo 1

Descrizione e implementazione del problema

Il progetto implementato ha l'obiettivo di risolvere il problema dell'**Open Shop Scheduling (OSSP)** utilizzando come 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, ..., J_n con tempi di elaborazione variabili, che devono essere eseguiti su m macchine m_1, m_2, ..., 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, ..., 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	
JOBS	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 feromoni. Se altre formiche trovano un percorso del genere, è probabile che non continuino a viaggiare a caso, ma piuttosto che seguano il percorso, tornando e rafforzandolo se alla fine trovano cibo.
Col tempo, tuttavia, la scia di feromoni inizia ad evaporare, riducendone così la forza attrattiva. Più tempo impiega una formica per viaggiare lungo il sentiero e tornare indietro, più tempo hanno i feromoni per evaporare. Un cammnino 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 dei feromoni 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 ← 1 to max_gen do
3:     for i ← 1 to num_ants do
4:         s_i ← create_solution()
5:     end for
6:     pheromone_evaporation()
7:     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:

$$p_{i->j} = \frac{\tau_{ij}^{\alpha}\theta_{ij}^{\beta}}{\sum_{k \in NV} \tau_{ij}^{\alpha}\theta_{ij}^{\beta}} for j \in NV \tag{1.1}$$

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 probabilitico 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:

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

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 (ricompensa) 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)

```
for all c_i, c_j che appaiono in s do
    τ_ij ← τ_ij + K/Ls
end for
```

Ls è il costo della soluzione s.

Ci sono 3 possibilità:

1. reward(s_{bs}, w_{bs})
2. reward(s_{ib}, w_{ib})
3. reward(s_{bs}, w_{bs}) e 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'Iteretion_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 sè, ma è data alle componenti della soluzione (archi/edges).
I valori euristici aiutano le formiche a scegliere, soprattuto all'inizio, perchè all'inizio il feromone non da 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 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`.
Oltre ad uno script per creare le istanze di input è possibile utilizzare anche un ottimizzatore dei parametri, il quale permette di trovare la combinazione ottimale di parametri con cui è possibile raggiunge il lower bound noto per una data istanza (è ovviamente necessario conoscere a priori il lower 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).

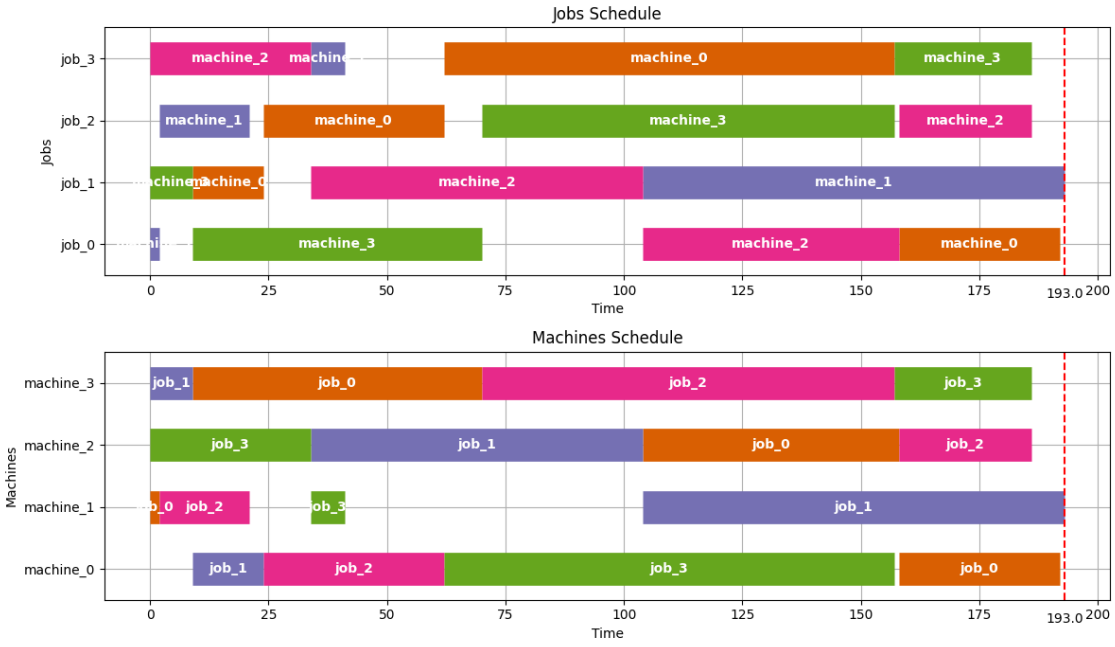

Figura 1.2: Gantt per un'istanza di input data da 4 job e 4 macchine.