Sistemi Intelligenti: Il Job Shop Scheduling Problem

Di Mergoni Alberto e Braghini Marco

1. Introduzione

In un panorama competitivo in cui l'efficienza operativa è vitale, la capacità di pianificare in modo ottimale le risorse è diventata un elemento chiave per il successo. Il Job Shop Scheduling Problem (JSSP) costituisce un nodo cruciale nell'ambito dell'ottimizzazione combinatoria, con impatti significativi su vari settori.

Le componenti principali sono**:**

- **Task**: I Task rappresentano le unità di lavoro da completare. Ogni task è composto da una sequenza di operazioni che devono essere eseguite in un determinato ordine e su determinate macchine.
- **Lavori (Jobs)**: I lavori sono le singole attività che compongono un task. Ogni lavoro ha una durata di esecuzione e richiede una specifica macchina per essere eseguita.
- **Macchine**: Le macchine sono le risorse su cui vengono eseguiti i job. Ogni macchina può eseguire un solo job alla volta e può avere vincoli di capacità e disponibilità.

L'obiettivo del JSSP è trovare una sequenza di assegnazioni delle operazioni alle macchine che minimizzi il tempo totale di completamento di tutti i lavori. Il tempo totale di completamento, noto anche come makespan, è il tempo trascorso dal momento in cui inizia la prima operazione fino al completamento dell'ultima operazione.

Il JSSP ha numerose applicazioni pratiche in settori quali la produzione industriale, la logistica, i trasporti e la gestione delle risorse informatiche. Proprio quest'ultimo è stato l'ambito di nostro interesse, in particolare l'allocazione di risorse nelle CPU. Il JSSP si presta esattamente a risolvere questo problema, e proprio questo ci ha portato a studiarlo.

Il JSSP è un problema notoriamente complesso a causa delle molte variabili e vincoli coinvolti:

- **Dipendenze Temporali**: Le operazioni possono dipendere l'una dall'altra in termini di sequenza di esecuzione, cioè una operazione deve essere completata prima che un'altra possa iniziare.
- **Capacità delle Macchine**: Ogni macchina ha una capacità massima di lavoro e non può eseguire più di una operazione contemporaneamente.
- **Variabilità delle Durate delle Operazioni**: Le operazioni possono avere durate diverse e possono richiedere macchine diverse per essere eseguite.
- **Numero di Lavori e Macchine**: Il numero di lavori e macchine può variare considerevolmente da un'istanza del problema all'altra, influenzando la complessità computazionale del problema.

Dal punto di vista della teoria della complessità computazionale, il JSSP è un problema NP-completo, il che significa che è almeno altrettanto difficile quanto il più difficile dei problemi NP (Non-deterministic Polynomial time). Ciò implica che non esiste un algoritmo efficiente (in tempo polinomiale) per risolvere il JSSP in generale, specialmente quando il numero di lavori e macchine diventa grande.

2. Approccio Generale

L'approccio adottato per risolvere il problema del JSSP utilizza il paradigma del Constraint Satisfaction Problem (CSP). Questo metodo si basa sulla definizione di variabili decisionali, vincoli e obiettivi da ottimizzare. In particolare:

- **Variabili Decisionali**: Le variabili decisionali rappresentano gli oggetti o le entità di interesse nel problema. Ogni variabile ha un dominio di valori che può assumere.
- **Dominio delle Variabili**: Il dominio di una variabile è l'insieme di tutti i possibili valori che può assumere. Questo dominio può essere finito o infinito e può essere discreto o continuo.
- **Vincoli**: I vincoli rappresentano le relazioni tra le variabili decisionali e limitano le combinazioni valide di valori per queste variabili. I vincoli possono essere di diversi tipi, come vincoli di uguaglianza, vincoli di disuguaglianza, vincoli di assegnazione, vincoli aritmetici, vincoli logici, ecc.

Il CSP è un metodo che sfrutta una struttura fattorizzata, questo porta a vantaggi, come la succintezza del codice, ma anche a svantaggi, come l'aumento della complessità di calcolo.

La risoluzione del CSP da noi usata coinvolge la ricerca di una soluzione che soddisfi tutti i vincoli e fa uso di diversi approcci per risolvere un CSP:

- Backtracking: Un algoritmo di ricerca esaustiva che esplora ricorsivamente lo spazio delle soluzioni, facendo scelte per le variabili decisionali e controllando se queste scelte violano i vincoli.
- Consistenza dei Vincoli: Prima di iniziare la ricerca della soluzione, è possibile applicare tecniche per ridurre lo spazio delle soluzioni, ad esempio utilizzando la consistenza dei vincoli per eliminare valori dai domini delle variabili.
- Propagazione dei Vincoli: Durante la ricerca della soluzione, è possibile utilizzare tecniche di propagazione dei vincoli per eliminare valori dai domini delle variabili in base alle informazioni disponibili.

3. Modellazione del Problema

Per analizzare appieno il problema è stato prima di tutto fondamentale analizzare l'environment. Quest'ultimo è caratterizzato da:

- **Single/Multi agent** dato che in base al numero di macchine che si sceglie (1 o più) si ha un solo agente decisionale o di più.

- **Osservabilità totale** dato che il modello descrive completamente il sistema e specifica tutte le informazioni necessarie per trovare una soluzione ottimale al problema.
- **Statico** dato che tutte le informazioni sono conosciute a priori e non cambiano nel tempo.
- **Discreto** poiché rappresentano istanti di tempo discreti in cui le attività possono iniziare.
- **Deterministico** siccome questo modello assume che la durata di ciascuna attività sia deterministica (cioè, nota con certezza)
- **Episodico/sequenziale** dato che rappresentiamo una sequenza di attività che devono essere eseguite si in ordine (un job può essere eseguito solo se quello prima è stato fatto), ma quest'ordine varia in base alla soluzione che ci permette di avere il makespan minimo.

Successivamente abbiamo analizzato la struttura dell'agente. L'obbiettivo principale è minimizzare il makespan. Questo può essere interpretato come un obiettivo di massimizzazione della funzione, in cui l'agente cerca di minimizzare il tempo necessario per completare tutte le attività. Pertanto, l'approccio può essere considerato più simile a un **agente basato sulle utilità** (Utility Based Agent).

Per modellare il problema del JSSP utilizzando il metodo CSP abbiamo fatto uso del programma MiniZinc. Questo programma ci fornisce un'interfaccia standard per il CSP, in modo semplice e intuitivo, utilizzando solutori specializzati che implementano algoritmi efficienti per trovare soluzioni ottimali o approssimate.

Per fare ciò è necessario definire le variabili decisionali, i vincoli e la funzione obiettivo.

Come variabili di input abbiamo usato:

- n_macchine: intero che indica quante macchine sono a disposizione
- n_tasks: intero che indica il numero di task che le macchine devono eseguire
- n_job_per_task: intero che indica da quanti job è composta ogni task
- macchine_per_job: array che assegna a ogni job la macchina che dovrà eseguirlo
- durata_jobs: array che indica la durata di ogni job

Le variabili decisionali invece includono:

- min_durata: indica il tempo minimo in cui i task possono essere compiuti, ovvero il tempo totale per completare il task più lungo
- max_durata: indica il tempo massimo in cui i task possono essere compiuti, ovvero il tempo totale per completare tuti i task uno in fila all'altro
- inizio_jobs: array in cui sono contenuti tutti i possibili tempi di inizio di ogni job (questi vanno da 0 a max_durata)
- fine_jobs: array in cui sono contenuti tutti i possibili tempi di fine di ogni job (questi vanno da 0 a max_durata)
- tempo_completamento: indica il tempo per completare tutti i task, ovvero tutti i job (questo è compreso tra min_durata e max_durata)

I vincoli vengono specificati attraverso la funzione "constraint" e nel nostro caso applicati a tutte le variabili grazie alla funzione "forall".

Nel nostro programma abbiamo applicato i seguenti vincoli:

- Tutti i job devono avere un tempo di completamento positivo
- I job devono iniziare da un valore positivo maggiore o uguale a zero
- Il tempo di fine di un job è dato dal suo tempo di inizio più la sua durata
- Posso eseguire un job solo se ho eseguito tutti quelli prima dello stesso task
- Una macchina può far partire solo un job per volta e durante il tempo di esecuzione non può farne partire un altro (per realizzare questo vincolo abbiamo usato la funzione no_overlap inserendo i valori di inzio e durata di due job assegnati a una stessa macchina)
- Il tempo di completamento non può essere minore del tempo degli intervalli di durata di tutti i job, ovvero il tempo di completamento è il valore uguale al tempo di fine dell'ultimo job che viene eseguito

La funzione obiettivo è quella di minimizzare il tempo totale di completamento dei job, per fare questo sfruttiamo le potenzialità di MiniZinc usando il risolutore "resolve minimize" + la variabile, nel nostro caso tempo_completamento.

Grazie a questo otterremo la/le soluzioni possibili con i tempi di completamento minori.

Una volta fatto ciò, grazie a un susseguirsi di stringhe stampiamo il risultato in modo più comprensibile specificando il tempo di completamento e i parametri più significativi per ogni job.

4. Valutazione

Abbiamo eseguito una serie di test utilizzando diverse istanze del problema con varie dimensioni e complessità al fine di valutare l'efficacia e l'efficienza del nostro modello nella risoluzione del JSSP. Siamo estremamente soddisfatti della qualità delle soluzioni ottenute, poiché garantiamo la ricerca della soluzione ottimale. Tuttavia, il tempo di esecuzione rimane un aspetto critico che richiede un'attenzione particolare.

Per condurre i test, abbiamo mantenuto costante il numero di job per task uguale al numero di task, al fine di assicurare la massima obiettività possibile. Abbiamo utilizzato il solver Chuffed dopo averne testati diversi. Inizialmente, Chuffed ha mostrato prestazioni comparabili agli altri solvers su istanze di piccole dimensioni, ma si è dimostrato il più performante man mano che le dimensioni dei problemi aumentavano.

task	job per task	durata	macchine	tempo Chuffed
5	5	5s	5	462 ms
5	5	5s	3	478ms
5	5	50s	5	1s 319ms
5	5	50s	3	1s 444ms
8	8	5s	8	1s 301ms
8	8	5s	5	2s 498ms
8	8	50s	8	12s 716ms
8	8	50s	5	19s 760ms
10	10	5s	10	4min 508 s
10	10	5s	5	14m 23s
10	10	50s	10	2min 5 s
10	10	50s	5	2 ore +
15	15	5s	15	56s 111ms
15	15	5s	10	2 ore +
15	15	50s	15	1 ora 31min

Come previsto, abbiamo osservato che il tempo di risoluzione varia in base alla complessità del problema, aumentando con il numero di task e la durata dei job, ma diminuendo con l'aumentare del numero di macchine.

Il JSSP è noto per la sua elevata complessità computazionale, e come evidenziato dai nostri test, può richiedere anche diverse ore per essere risolto.

Abbiamo esplorato l'opzione di sviluppare euristiche di ricerca locale (hill climb e plateau), ma queste hanno mostrato performance inferiori rispetto all'utilizzo di un set di tempo di completamento che varia dalla durata minima a quella massima oppure non si sono implementate bene con il codice.

Considerando che il programma spesso ottiene risultati accettabili in pochi secondi e il tempo successivo è dedicato all'esplorazione più ampia dello spazio delle soluzioni, potrebbe essere vantaggioso implementare un troncamento dopo un certo periodo di tempo. Tuttavia, abbiamo scelto di mantenere il codice invariato per garantire sempre la ricerca della soluzione ottimale.

Per i futuri sviluppi, potremmo valutare la parallelizzazione del processo di risoluzione per sfruttare al meglio le risorse disponibili, esplorare l'utilizzo di solutori più potenti o studiare ulteriori euristiche specifiche del dominio per guidare la ricerca verso soluzioni più promettenti.