UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CSP solver

Autore:
Edoardo Rustichini

N° Matricola:
7076614

Corso:
Intelligenza artificiale

Docente corso:
Paolo Frasconi

Indice

Elenco delle figure

1 Introduzione

Assignment

Si scriva (in un linguaggio di programmazione a scelta) un generico solver per problemi di sodisfacimento di vincoli basato su backtracking e MAC, capace di generare tutte le soluzioni per un problema assegnato. Si consideri quindi il problema descritto in 6.1.2 di RN 2021 e lo si risolva enumerando tutte le soluzioni compatibili con i vincoli e scegliendone una a costo minimo. Si applichi il metodo ad almeno tre istanze diverse.

L'obbiettivo del progetto è implementare un solver generico per problemi di soddisfacimento dei vincoli, che riesca ad enumerare tutte le possibili soluzioni e selezionandone una a costo minimo; inoltre è previsto si applichi il solver ad un problema di job-shop-scheduling.
Per l'implementazione ho scelto il linguaggio `Python`.
La relazione è strutturata come segue:

- breve introduzione teorica dell'argomento trattato
- descrizione dell'implementazione
- analisi esperimenti
- considerazioni finali e spunti per sviluppi futuri.

1.1 Introduzione teorica

1.1.1 Problemi CSP

I problemi di soddisfacimento dei vincoli (CSP: *Constraint Satisfaction problem*) sono problemi definiti su un insieme di variabili $\mathcal{X} = \{X_1 : i = 1 \dots n\}$, ognuna associata ad un dominio $D_i \neq \emptyset$ che corrisponde ai valori ammessi per quella variabile. Sulle variabili del problema viene definito un insieme di vincoli $\mathcal{C}$ che possono essere pensati come relazioni che specificano quali valori le variabili coinvolte possono prendere.

Una caratteristica importante per questo tipo di problemi è la loro *rappresentazione fattorizzata*, che permette l'uso di tecniche **generiche** ed **indipendenti** dal dominio specifico, a differenza dei problemi di ricerca classici che spesso richiedono conoscenze specifiche.

In questi problemi si deve di pensare agli stati come **assegnamenti** delle variabili del problema: l'obbiettivo è trovare un assegnamento completo e consistente, ovvero tutte le variabili devono essere associate ad un valore (e quindi assegnate) e nessuno dei vincoli è violato.

Un aspetto fondamentale dei CSP è il fatto che, quando un assegnamento parziale viola un vincolo, possiamo scartare estensioni di quell'assegnamento, evitando di esplorare grandi porzioni dello spazio degli stati.

Propagazione dei vincoli Negli algoritmi risolutori di CSP viene impiegata un tipo di inferenza chiamata **propagazione dei vincoli** che sfrutta i vincoli per eliminare valori non consistenti dai domini delle variabili; questo permette di ridurre lo spazio di ricerca. Spesso si rappresenta un problema CSP come **grafo dei vincoli**, dove ogni nodo rappresenta una variabile, e gli archi che connettono variabili rappresentano i vincoli.

Legata a questa rappresentazione è il concetto fondamentale di **consistenza d'arco**, una proprietà per cui, per ogni valore nel dominio *attuale*di una variabile X_i deve esistere almeno un valore nel dominio di una variabile X_j che soddisfa il vincolo (*arco* nel grafo dei vincoli) tra queste due variabili. L'AC-3 è l'algoritmo più noto per controllare e garantire questa proprietà; lo fa controllando tutti gli archi del problema e rimuovendo i valori inconsistenti con i vincoli presenti. Dopo l'applicazione di questo, se non sono rilevate inconsistenze, si ottiene un nuovo CSP equivalente all'originale, ma con domini ridotti.

Ricerca della soluzione La ricerca della soluzione può essere eseguita usando una ricerca sistematica nel quale considero come stati un assegnamento parziale, e come azioni l'estensione degli assegnamenti. Per fare questo si può usare l'algoritmo di **backtracking** che sceglie e assegna una variabile non ancora assegnata, e verifica se questo assegnamento porta ad una violazione dei vincoli: se questo accade l'algoritmo torna al passo precedente ("*backtrack*") e prova un altro valore. Questa ricerca anche se corretta e completa può risultare inefficiente su problemi complessi, e per

questo si possono decidere di usare molte tecniche di ottimizzazione, euristiche e tipi di inferenza, tra cui:

- ordinamento di variabili (per esempio MRV), usate per determinare l'ordine con il quale scegliere la prossima variabile da esaminare; si usa approccio fail-first per potare sezioni inutili dell'albero di ricerca
- ordinamento di valori: decidere ordine con quale scegliere il prossimo valore da esaminare, si usa approccio fail-last per arrivare velocemente ad una soluzione
- inferenza: fare inferenza mentre si sta cercando la soluzione può essere utile per inferire sui domini delle variabili adiacenti a quella scelta (MAC)

1.1.2 Problemi job-shop scheduling

Tra i vari problemi che possono essere modellati come CSP rientra il *job-shop scheduling problem* (*JSS*): sono problemi di ottimizzazione dove l'obiettivo è quello di realizzare una pianificazione di alcuni *lavori* che devono essere svolti su un insieme di *risorse/macchine*. Generalmente si vuole che la soluzione minimizzi il *makespan*, ovvero la durata complessiva della sequenza. Per l'esercizio assegnato è però richiesto di risolvere una versione semplificata proposta nel 6.1.2 del RN, dove si può vedere come un unico lavoro che ha più operazioni.

Si può descrivere nel seguente modo:

- ogni lavoro corrisponde ad una variabile, i quali valori possibili rappresentano l'istante di inizio di quel lavoro (in minuti)
- i vincoli possono essere di precedenza, per imporre un ordine tra alcune operazioni, o disgiuntivi, per impedire il sovrapporsi nel tempo di diverse operazioni

Nell'esempio si sta considerando una parte dell'assemblaggio di un'automobile, che prevede:

1. installare gli assi
2. fissare le ruote
3. stringere i dadi per ogni ruota
4. fissare i copriruota
5. fare un'ispezione finale

Ogni compito ha una durata conosciuta, utile per definire i vincoli di precedenza ricavabili dall'ordine nella lista. Inoltre è presente un vincolo di disgiunzione tra l'installazione dei due assi poichè non possono essere eseguiti contemporaneamente.

2 Descrizione dell'implementazione

Il progetto è diviso neis eguenti moduli:

- `problem.py`: definisce il problema
- `constraint.py`: definisce i tipi di vincoli semplici
- `solver.py`: definisce il solver che usa il backtracking per ricercare le soluzioni
- `job-scheduling.py`: estende la classe `Problem` per modellare specificamente problemi di job-shop scheduling
- `main.py`: è il file di esecuzione principale che crea e risolve le istanze del problema
- `CPMpy.py`: usa il modulo `cpmp` per risolvere lo stesso problema e verificare la correttezza.

2.1 Rappresentazione di un CSP

Il modulo `problem` è la parte del progetto che organizza e definisce tutti gli elementi necessari di un CSP. Per creare un nuovo problema è sufficiente inizializzare un oggetto di questa classe, e successivamente usare i metodi forniti per aggiungere variabili e vincoli sulle variabili

Quando si aggiunge una variabile, automaticamente viene prima controllato se è già stata definita una variabile con quel nome: se non è presente, allora si aggiunge. L'insieme di vincoli `constraints` associati al problema è memorizzato mantenendo un dizionario che associa ogni variabile alla lista di vincoli in cui viene coinvolta, ricordando in questo modo una lista di adiacenza per il grafo dei vincoli, il che offre il vantaggio di accedere facilmente ed in modo rapido a tutti i vincoli che riguardano una specifica variabile, accelerando la verifica della consistenza di un assegnamento.

2.2 Vincoli

I vincoli sono l'argomento centrale dei CSP. La classe `Constraint` rappresenta un vincolo generico n-ario ed ha come attributi la lista di variabili coinvolte (`variables`)e la funzione (`constraint-func`) che determina se un vincolo è soddisfatto per un assegnamento. Il metodo importante di questa classe è `is-staisfied` per determinare se il vincolo è soddisfatto.

Sapendo che qualsiasi vincolo n-ario si può ricondurre ad un vincolo binario, ho deciso anche di implementare la classe `BinaryConstraint` che estende `Constraint` e impone che le variabili coinvolte siano esattamente 2. Per questo tipo di vincolo ho deciso che, nel caso in cui non tutte le variabili coinvolte sono state già assegnate, il vincolo è considerato soddisfatto.

La scelta di usare (`constraint-func`) permette di definire qualsiasi relazione senza creare classi dedicate.

2.3 Algoritmo Risolutivo

2.3.1 Struttura del Solver

La classe `Solver` rappresenta l'implementazione vera e propria dell'algoritmo di backtracking che permette di trovare le soluioni del CSP; questa opera su un'istanza di `Problem`. L'algoritmo è implementato, come richiesto, basandosi su MAC, ovvero una tecnica, che usa AC-3, per fare inferenza mentre la ricerca procede. La gestione di aggiunta e rimozione delle inferenze è stata gestita considerando un'inferenza come un dizionario che associa ad una variabile i suoi valori da rimuovere, e:

- se si vuole aggiungere l'inferenza al problema, vuol dire che si dovranno rimuovere, per ciascuna variabile, gli elementi presenti nelle inferenze
- se invece si vuole rimuovere un inferenza al problema, vuol dire che si dovranno ripristinare i domini come erano prima dell'aggiunta delle inferenze.

Uso di Euristiche Nel tentativo di migliorare l'efficienza viene usata l'euristica di MRV, che sceglie come variabile da assegnare quella con il dominio più piccolo; in caso di pareggio viene usata degree-heuristic che sceglie la variabile coinvolta in più vincoli. Si può notare che non è stata implementata un'euristica per l'ordinamento dei valori perché è richiesto di enumerare tutte le soluzioni, ovvero esplorare l'intero spazio delle soluzioni, rendendo quindi inutile l'ordine con il quale consideriamo i valori nel dominio.
Da questa osservazione potrebbe anche risultare inutile il fatto di usare MAC, però credo che comunque si possa trarre dei vantaggi da questo perché lo spazio di ricerca viene comunque diminuito perché elimina i rami dell'albero di ricerca che non portano a soluzioni; si noterà quindi un vantaggio se il problema presenta molti vincoli *stretti*, ovvero facilmente *insoddisfacibili*, nel caso invece di problemi più facili forse vedremo meno vantaggi dato che il MAC comunque introduce un overhead.

2.4 Modellazione problema di jss

La classe `JobShopSchedulingProblem` estende `Problem` per specializzarsi nella rappresentazione di job-shop scheduling. In questi tipi di problemi le variabili rappresentano le operazioni, che avranno lo stesso dominio che consiste nei valori da 1 al tempo massimo di esecuzione; ogni operazione è associata alla sua durata tramite l'uso di un dizionario.

2.4.1 Vincoli del jss

I metodi `add-precedence-constraint , add-disjunction-constraint` della classe `JobShopSchedulingProblem` permettono di aggiungere facilmente vincoli di precedenza e disgiunzione, i due vincoli principali per questo tipo di problemi. In particolare questi definiscono la funzione da verificare, che sfrutta le durate delle operazioni, ed infine creano un `BinaryConstraint` adatto che viene aggiunto al problema.

2.5 Uso di dizionari

Come si può leggere dalla descrizione dell'implementazione, la struttura dati che ho usato con maggior frequenza sono i dizionari. Questa scelta è stata motivata dall'efficienza e facilità nella gestione di associazioni di tipo chiave-valore. In Python i dizionari sono implementati usando tabelle hash, che consentono tempi medi di accesso, inserimento e aggiornamento in $O(1)$, e quindi costanti nella maggior parte dei casi. Nel progetto questo risulta molto utile perché ci sono frequenti operazioni di lookup (verifica dei vincoli o consultazione degli assegnamenti).

Le operazioni di ricerca, nel caso peggiore, manteiene invece complessità lineare; questo però nel nostro caso non è un problema perché queste operazioni saranno meno frequenti rispetto a quelle di lookup.

I dizionari però hanno un costo di memoria maggiore rispetto alle alternative, come le liste, perché si devono memorizzare le hash table, tabelle che mantengono le coppie key:valore.

3 Guida: creazione e risoluzione di un CSP

Vediamo il procedimento per creare e risolvere un problema:

1. Creare un oggetto `Problem`
2. Definire e aggiungere le variabili (ed i relativi domini) al problema
3. Aggiungere i vincoli sulle variabili definendo la funzione da soddisfare
4. Creare un oggetto `Solver` associandolo al problema da risolvere
5. Invocare il metodo `getAllSolutions` per ottenere tutte le soluzioni

4 CPMpy

Per verificare la correttezza dell'algoritmo sviluppato ho deciso di risolvere lo stesso problema usando una delle librerie python principali per la modellazione e la risoluzione di problemi CSP, la libreria `cpmpy`.

4.1 Breve introduzione a `cpmpy`

CPMpy (Contraint Programming Models for Python) è una libreria open source che permette di modellare e risolvere problemi di ottimizzazione vincolata in modo semplice ed intuitivo.

Per la precisione, CPMpy agisce da interfaccia tra Python e solver esterni in modo da semplificare la modellazione. Come solver predefinito usa OR-Tools, un framework open source sviluppato da Google per la risoluzione di problemi CP: CPMpy astrae le complessità di questo traducendo il modello scritto in python in un formato adatto a OR-Tools.

Per creare un problema usando questa libreria si deve:

1. definire le variabili: nel caso del problema specificato sono tutte variabili intere con domini finiti, quindi si può usare quindi `intvar()`
2. creare un modello: in questa libreria un oggetto `Model` rappresenta un container per memorizzare una lista di "CPMpy expressions" che nel caso di CSP rappresenteranno i vincoli
3. per aggiungere i vincoli al modello basta usare l'operatore +=;
4. Risolvere il problema: si può usare la funzione `solve()` che restituisce un valore booleano per comunicare se è stata trovata o meno una soluzione; nel nostro caso però si vuole enumerare tutte le soluzioni, quindi si deve usare la funzione `solveAll()` che cerca e conta tutte le soluzioni

4.2 Utilizzo nel progetto

Ho utilizzato la libreria CPMpy per validare la correttezza del solver implementato; in particolare ho verificato che

- il numero totale di soluzioni trovate dal mio solver coincidesse con quello di CPMpy
- il costo minimo fosse lo stesso in entrambi gli approcci

Per fare questo ho definito le 3 diverse istanze del problema di diversa complessità, e risolvendole separatamente usando il mio solver e quello di CPMpy.

In tutti i casi i risultati sono stati identici, e da questo si può intuire che, assumendo come corretti e affidabili OR-Tools e CPMpy

- il fatto che il numero di soluzioni trovate sia identico nei due casi indica che il mio solver esplora lo spazio di ricerca senza perdere soluzioni valide
- il fatto che il costo minimo sia lo stesso dice che il mio solver assegna correttamente i valori

Risultati	**my solver**			**cpmpy**		
	Istanza 1	**Istanza 2**	**Istanza 3**	**Istanza 1**	**Istanza 2**	**Istanza 3**
Soluzioni valide	163592	252448	11490500	163592	252448	11490500
Costo minimo	30	17	39	30	17	39
Soluzioni a costo minimo	163592	6272	414050	163592	6272	414050

Tabella 1: Confronto tra due approcci su tre diverse istanze del problema

5 Conclusioni

In questo progetto ho potuto applicare i concetti teorici studiati nel contesto dei CSP, riuscendo a creare un risolutore concreto e funzionante.

Dopo i test posso dire di aver ottenuto un solver

- affidabile data la validazione ottenuta tramite il confronto con i risultati di una libreria specializzata in questo tipo di problemi.
- semplice ed intuitivo
- facilmente estendibile

Alcune possibili estensioni potrebbero essere

- l'implementazione e l'uso di altre euristiche più avanzate nella ricerca della soluzione
- l'aggiunta di altri tipi di vincoli predefiniti, estendendo quelli base già presenti
- ottimizzazioni nella ricerca

Il diagramma di Gant di una soluzione a costo minimo dell'esempio del problema fornito nel libro è il seguente:

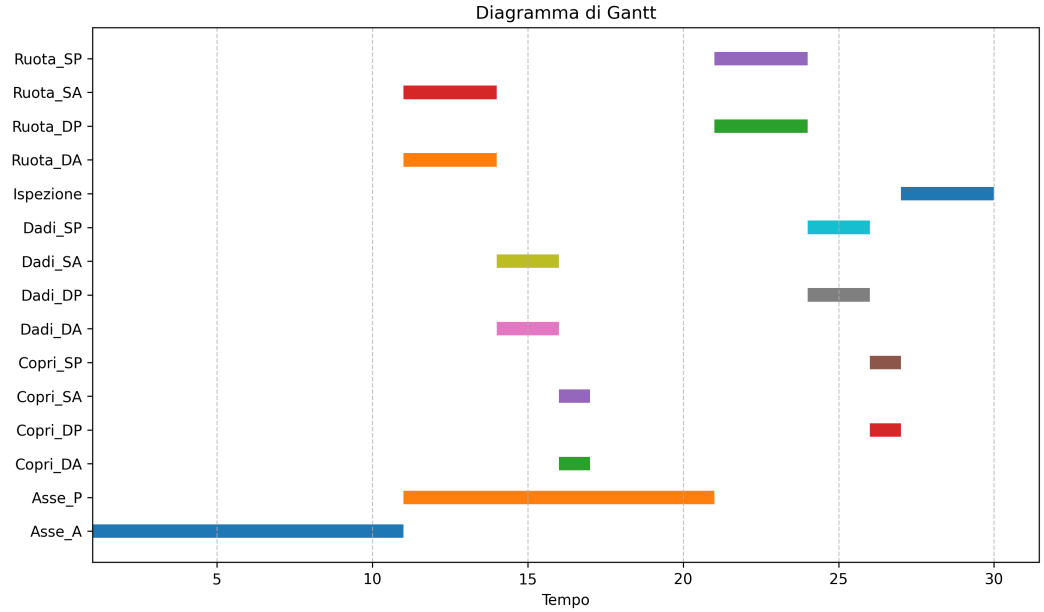

Figura 1: Diagramma di Gant

Come si può vedere i vincoli sono rispettati, e si può notare che alcune operazioni sono sovrapposte perché l'esempio non citava il vincolo aggiuntivo classico di risorse limitate, ma solo il fatto che c'erano quattro lavoratori,

Riferimenti bibliografici

[1] Intelligenza artificiale : un approccio moderno / Stuart J. Russell, Peter Norvig