Progetto di Automated Reasoning
Shy Travelling Salesmen Problem

Bruniera Alvise

Università degli studi di Udine

March 23, 2023

Abstract

Scrivi un programma in minizinc e ASP in grado di trovare una soluzione al seguente problema (STSP). Prepara una batteria di 30 istanze casuali, includendo alcune più facili, alcune medie ed alcune che eccedano il timeout di 5 minuti. Possibilmente sperimenta con diverse strategie. Nel caso di COP riporta il risultato migliore ottenuto entro il timeout.

Le strade di una città sono rappresentate mediante una matrice di adiacenza, $M[x, y] = c$ significa che tra l'incrocio x e l'incrocio y c'è una strada di lunghezza c km. Nella rappresentazione logica useremo invece fatti del tipo `strada`(x, y, c). Due auto partono dallo stesso incrocio prendendo due strade diverse. Si trovi un piano per le due auto tale da permettere a ciascuna di effettuare un circuito Hamiltoniano, senza mai fermarsi e senza mai passare per un incrocio allo stesso tempo (inteso con una certa tolleranza, che so mezz'ora, vedi tu come formularlo in modo sensato) dell'altra auto. Non è necessario che ritornino al punto di partenza allo stesso momento ma entrambe lo fanno.As si, assumiamo ce le auto vadano costantemente alla stessa velocità.per comodità metterei 1 minuto a km. E le lunghezze delle strade le farei intere.

Ho soprannominato il problema *"Shy" Travelling Salesmen Problem* (STSP) perché è simile al TSP, ma con il requisito che i due commessi viaggiatori "timidi" mantengano le distanze entro una certa tolleranza che chiamerò "timidezza". Sempre per similitudine con TSP, nel caso di COP ho deciso di ottimizzare la somma del tempo di percorrenza delle due macchine.

I modelli si trovano rispettivamente nei file `minizinc/progetto.mzn` e `asp/progetto.lp`. I risultati dei benchmark, invece si trovano nella cartella `results/`. Gli script `data_generator.py`, `bench-minizinc.fish` e `bench-asp.fish` permettono di generare un batch di istanze benchmark e di eseguirli (il primo richiede python, mentre gli altri due fish-shell).

Contents

1 Introduzione al problema

Prima di affrontare le soluzione è necessario chiarire alcuni dettagli sul problema, che la consegna non specifica.

Per cominciare, non viene specificato come rappresentare nella matrice quando due incroci *non* sono collegati. Per comodità ho deciso di rappresentarlo inserendo un numero negativo nella matrice, e considerando invece 0 come una strada. Usare numeri negativi ha facilitato lo sviluppo, in questo modo per attivare o disattivare un a strada in un test era questione di cambiare un segno. Quanto allo zero, si tratta di una decisione arbitraria, ad ogni modo non rende più semplice il problema. Questo non ha riscontri nella codifica ASP, se non nel permettere fatti del tipo `strada`$(x, y, 0)$, perché le strade assenti sono semplicemente assenti.

Un'altra precisazione riguarda la direzionalità e la riflessività del grafo. La consegna non specifica che il grafo debba essere indiretto, ache se nel TSP classico è così. Per semplificare la generazione delle istanze di benchmark ho deciso di usare un grafo indiretto, che non richiede di assicurarsi che $m[x, y] = m[y, x]$. Questo ha avuto degli effetti marginali sul modello utilizzato. Per quanto riguarda la riflessivita, se anche ci fosse un arco che connette un nodo a se stesso, questo non sarebbe mai utilizzato in un circuito perché violerebbe il requisito di attraversare ogni incrocio una volta sola, quindi non fa differenza che siano ammessi o meno.

La direzionalità del grafo ed ammettere archi di lunghezza 0 sono un po' meno naturali rispetto alla formulazione classica di TSP, ma sono comunque riconducibili al mondo reale come la presenza di ostacoli in un solo lato della strada, e la presenza di incroci molto ravvicinati, ad esempio in una rotonda.

I casi benchmark forniti sono "standardizzati" nel senso che tutti gli archi assenti sono -1 e non sono presenti archi riflessivi.

Sia la tolleranza agli incontri, che il nodo di partenza, sono indicati nell'input del problema e possono essere alterati a piacere. Per semplicità i benchmark usano sempre gli stessi valori. In particolare usano 6 minuti per la tolleranza, scelta in modo puramente arbitrario.

Per quanto riguarda il valore da ottimizzare, ho scelto di minimizzare la somma dei tempi di percorrenza totali. Ho considerato di utilizzare il massimo dei tempi, ma mi sembrava meno interessante come soluzione. In usando il massimo i due circuiti sarebbero stati in un certo senso "disaccoppiati" permettendo a quello più corto di crescere arbitrariamente senza cambiare il punteggio finche non supera il più lungo. Ho considerato anche di massimizzare il distacco tra i due viaggiatori, ma questo avrebbe reso inutile il valore della tolleranza e sarebbe diventato in generale un problema meno realistico.

2 Minizinc

2.1 Modellazione

A metà dello sviluppo ho notato che la libreria standard di minizinc fornisce alcuni global constraint per verificare che un vettore di precedenze rappresenti un circuito in un grafo. Ho deciso di non utilizzarla per più motivi, anzitutto perché questo avrebbe eliminato metà della consegna, inoltre perché ero già a buon punto e ottenevo comunque prestazioni soddisfacenti senza utilizzarlo.

Circuito Hamiltoniano Per rappresentare il circuito utilizzo semplicemente un vettore bidimensionale `path` di tipo `array [1..2,1..n+1] of var 1..n` in cui sono indicati per le due macchine la sequenza di nodi da attraversare.

Per verificare che sia un Hamiltoniano utilizzo 5 vincoli. Il primo richiede che il primo e l'ultimo nodo di entrambi i percorsi siano uguali al nodo di partenza. Due richiedono che tutti i nodi nel percorso siano diversi (utilizzando il global constraint `all_different`). E gli ultimi due controllano che per ogni coppia di nodi consecutivi nel percorso esista l'arco nella matrice.

Su tutti questi vincoli ho utilizzato l'annotazione `::domain` per forzare la domain propagation, anche se sugli ultimi due non sembrava avere effetti. Su tutti gli altri vincoli del modello, qualsiasi tipo di propagazione provassi o non aveva effetti, o peggiorava le prestazioni o veniva direttamente rifiutata dal solver.

Timidezza Per verificare che le due macchine non si incontrassero entro una tolleranza ho utilizzato un secondo vettore `time` di tipo `array [1..2,1..n] of var 0..max_time` in cui `max_time` è un grezzo upper bound dei tempi ottenuto sommando tutti i valori positivi della matrice `m`. Questo limite dovrebbe aiutare il solver a risolvere il modello. Inoltre avrebbe permesso di utilizzare le ottimizzazioni del compilatore superiori ad `-O2`, che poi non sono state usate perché per qualche ragione era più veloce con le ottimizzazioni di default.

Il vettore viene popolato prima inserendo 0 nelle celle corrispondenti al nodo di partenza e poi incrementalmente utilizzando l'ultimo valore inserito nel vettore, ed il peso del prossimo arco da attraversare. Il vettore è stato etichettato con l'annotazione `::is_defined_var` poiché non serve fare ricerca su questi valori, ma dipendono direttamente dal percorso, il chè ha un osservabile impatto sulle prestazioni.

Dopo averli calcolati, semplicemente un vincolo controlla che i tempi di visita corrispondenti a ciascun nodo abbiano una differenza (assoluta) maggiore della variabile `limit`.

Come symmetry break (implicato dal vincolo sui tempi) utilizzo il vincolo `constraint path[1,2] > path[2,2]`. Se il grafo fosse stato indiretto avrei potuto aggiungere anche il vincolo `constraint path[1,2] > path[2,n]`, ma in un grafo indiretto questo potrebbe rendere insoddisfacibile un'istanza, ad esempio se tutti i nodi entranti in `start` partono da nodi di indice più alto di tutti i nodi uscenti.

Ho sperimentato anche con altri metodi per calcolare i tempi e con altri vincoli ridondanti (alcuni dei quali sono ancora commentati nel file della soluzione), ma quelli appena illustrati hanno fornito i risultati migliori.

Goal Come obbiettivo sommo i pesi di tutti gli archi attraversati da entrambe le macchine (che saranno sempre positivi) e minimizzo il valore risultante. Questo valore viene stampato in output e rappresenta l'ottimo raggiunto.

2.2 Euristiche

Queste euristiche sono la parte del progetto in cui mi sono concentrato maggiormente e che hanno influito di più sulla qualità del risultato.

Ho cominciato cercando di ottimizzare le euristiche un input piccolo, e dopo aver trovato un buon risultato, lo ho provato su un input molto grande ed ho scoperto che le euristiche per gli input piccoli sono pessime per quelli grandi e viceversa.

Tuttavia, ho anche notato che è possibile utilizzare espressioni e condizionali all'interno delle euristiche. Quindi non ero costretto ne a scegliere di ottimizzare per una sola dimensione ne a fare compromessi, ma potevo utilizzare un approccio ibrido.

Piccole istanze Per le piccole istanze, ho notato che un approccio esaustivo e naive produce i risultati migliori, rispetto a tecniche più probabilistiche.

Trattandosi di un circuito, ha senso che le variabili vengano popolate in `input_order`. Ma per la scelta del valore `indomani_reverse_split` non ho una spiegazione, ho il sospetto che funzioni bene solo perché sono istanze casuali. Ho anche notato che `indomani_reverse_split` non è più veloce di `indomani_split` ad esaurire la ricerca (cambia solo se iniziare dalla metà superiore od inferiore) ma nei casi testati trova l'ottimo prima.

Con un approccio del genere non ha senso utilizzare riavvii. Un condizionale selezione l'euristica `restart_none` per le istanze piccole.

Utilizzare questa euristica per input molto grandi ha risultati pessimi, non riesce a trovare velocemente un buon ottimo ed ovviamente non è possibile

esaurire la ricerca velocemente.

Grandi istanze Per le istanze grandi, il risultato migliore lo ho trovato con un approccio "aggressivamente probabilistico" basato su una semplice strategia di Large Neighborhood Search (LNS). Questo approccio ottiene risultati sorprendentemente buoni, mentre tentare una ricerca più esaustiva non avrebbe senso su istanze oltre una certa dimensione.

Utilizzando l'euristica `relax_and_reconstruct`. Ad ogni riavvio le variabili hanno una certa probabilità di essere mantenute per l'esplorazione successiva. Ho notato che istanza molto grandi funzionano meglio con probabilità più alte mentre quelle più piccole con probabilità più basse. Quindi la probabilità è calcolata dalla dimensione del grafo, e limitata al massimo a 90 %, che viene raggiunta da istanze con `n >= 70`.

L'esplorazione avviene con una strategia `first_fail` per la scelta delle variabili, e `indomain_random` per la scelta dei valori. Utilizzare questa strategia fa una notevole differenza nel trovare velocemente un ottimo.

Per i riavvii ci sono due strategie principali: riavvii rapidi, per evitare di "incagliarsi" in un ramo sbagliato e sfruttare di più la LNS; e riavvii lenti, per esplorare meglio un ramo ed assicurarsi di non perdersi alcuni ottimi. Entrambe le tecniche hanno vantaggi e svantaggi. La ricerca con la sequenza di Luby (con l'annotazione `restart_luby`) permette di utilizzare entrambe le tecniche, alternando fasi di riavvii molto rapidi, ed alcuni riavvii esponenzialmente sempre più lenti. Si ottenevano buoni risultati anche con i riavvii esponenziali, ed una base molto bassa (ad esempio 1.00001), però a contrario della sequenza di Luby, questa ad un certo punto smette di fare riavvii rapidi e rimane "bloccata" su un ottimo.

Utilizzare queste euristiche per un piccolo input non da buoni risultati. Non solo non si arresta mai perché continuando a riavviare, e procedendo randomicamente, non riesce ad esaurire la ricerca. Ma sotto una certa dimensione di input è addirittura più lento della ricerca esaustiva nel trovare l'ottimo.

Risultato finale La soglia oltre la quale gli input sono considerati "grandi" è stata trovata per tentativi. Corrisponde alla dimensione oltre la quale la ricerca eccede il limite di 5 minuti, che sulla mia macchina corrisponde a `n > 14`, ma non esiste una soglia "universale".

Se oltre alla dimensione degli input si varia anche la tolleranza agli incontri, il tempo potrebbe aumentare ed eccedere il limite. Tuttavia, nei test effettuati si è visto che in questi casi in cui l'istanza è piccola ma comunque eccede il limite, la ricerca esaustiva trova un ottimo prima di LNS. Avere

questa tolleranza come parametro non era richiesto dalla consegna, quindi non ho studiato approfonditamente cosa succede al suo variare.

In realtà, utilizzare multithreading avrebbe permesso di spostare questa soglia più in alto. Ma per qualche ragione, la LNS diventava più lenta in multithreading, quindi come compromesso non è stato utilizzato.

Un possibile sviluppo futuro potrebbe essere una selezione più accurata dell'euristica da utilizzare, analizzando meglio la macchina ed il problema, invece che limitarsi ad una condizione sulla dimensione del grafo.

Questa euristica selettiva permette al modello di affrontare con buoni risultati sia problemi medio-piccoli (che costituiscono la maggior parte dei benchmark forniti). Che casi molto grandi, di cui sono stati forniti tre esempi, rispettivamente con `n = 30`, `n = 50`, ed `n = 100`. Per ottenere buoni risultati anche con input ancora più grandi bisognerebbe modificare l'espressione `min(90, 55 + (n div 2))` per raggiungere probabilità più alte di 90 % con input più grandi di `n = 100`.

```
solve
  ::relax_and_reconstruct([path[j,i] | i in 1..n+1, j in
   1..2], min(90, 55 + (n div 2)))
  ::if n > 14 then
    int_search(path, first_fail, indomain_random)
   else
    int_search(path, input_order, indomain_reverse_split)
   endif
  ::if n > 14 then
    restart_luby(n div 4)
   else
    restart_none
   endif
  minimize total;
```

3 ASP

3.1 Modellazione

A causa di una piccola svista che non valeva la pena di risolvere, nella codifica per ASP i nodi sono numerati partendo da 0, mentre la matrice per minizinc è indicizzata partendo da 1. Le istanze di benchmark sono esattamente le stesse, solo numerati in modo diverso. Se proprio si volesse avere soluzioni allineate sarebbe sufficiente incrementare di 1 tutti gli indici nelle istanze fornite.

Circuito Hamiltoniano Come primo approccio ho tentato di "tradurre" direttamente la soluzione minizinc, implementando una sorta di vettore come una serie di fatti del tipo `step(Z,Y,X)`, che significa al passo `Y`, la macchina `Z` si trova al nodo `X`. Inoltre c'erano alcune differenze nel come produce l'elenco dei nodi. Questa soluzione incompleta si può trovare nel file `asp/incompleto.lp`.

Alla fine è stata abbandonata perché le prestazioni non erano soddisfacenti, ho invece optato per una versione modificata dell'implementazione di Hamilton Circuit vista a lezioni. L'unica differenza (a parte i nomi delle funzioni) è l'aggiunta della variabile `Z` per indicare di quale macchina si tratta.

```
{go(Z,X,Y)} :- strada(X,Y,_), car(Z).

seen(Z,X) :- go(Z,S,X), start(S), car(Z).
seen(Z,Y) :- seen(Z,X), go(Z,X,Y), car(Z).

1{go(Z,X,Y) : strada(X,Y,_)}1 :- node(X), car(Z).
1{go(Z,X,Y) : strada(X,Y,_)}1 :- node(Y), car(Z).
:- node(X), not seen(Z,X), car(Z).
```

Timidezza Per modellare la tolleranza agli incontri introduco una nuova funzione

3.2 Confronto con minizinc

4 Risultati dei benchmark