Relazione Progetto - Automated Reasoning

Voltan Gabriele
voltan.gabriele@spes.uniud.it

Febbraio 2023

Indice

1 Introduzione

I *Constraint Optimization Problem* sono dei problemi in cui, date delle variabili, dei domini per esse, dei vincoli su quest'ultimi e una funzione *f*, ricerchiamo l'assegnamento per le variabili che rispetta i domini e i vincoli e che minimizza o massimizza il valore della funzione data.

L'obiettivo di questo progetto è la modellazione di COP con due diverse tecniche: la prima è la *Constraint Programming*; la seconda è invece l'*Answer Set Programming*. Per implementare la prima soluzione utilizzeremo *MiniZinc*, mentre invece per la seconda utilizzeremo l'ASP solver *Clingo*.

In questa relazione troviamo: la presentazione del problema con alcune assunzioni fatte; l'implementazione della soluzione al problema con entrambe le tecniche descritte sopra; infine, i risultati ottenuti e le considerazioni finali sul lavoro svolto.

2 Il problema

Postino. All'ufficio postale di Retrograd il postino riceve ogni giorno un certo numero di buste da consegnare. Ci può essere anche più di una busta per indirizzo (pensate ai condomini). E ovviamente anche 0 (non si riceve posta ogni giorno). Questo sarà dunque il dato di input: $n_1, n_2, n_3, ..., n_k$ dove i vari n_j sono gli indirizzi di consegna, eventualmente ripetuti.

La città viene rappresentata con un grafo (non diretto) con archi di diversa lunghezza (numero intero). L'ufficio postale è in uno dei nodi. Le varie abitazioni sono negli altri nodi (indirizzabili mediante gli indirizzi). Ovviamente anche questo è parte dell'input.

Si risolva il problema di consegnare tutte le lettere percorrendo meno strada possibile ritornando alla fine all'ufficio postale. Ma ricordatevi che il postino gira con una borsa a tracolla che può contenere al massimo h buste (ad esempio 50, ma anche questo è un dato di input, dipende un po' dalle dimensioni che la vostra soluzione riuscirà a risolvere). Dunque consegnate quelle h dovrà tornare all'ufficio postale a prendere altre h e cosi' via.

2.1 Assunzioni

Per la modellazione del problema sono state fatte alcune assunzioni in merito ad alcuni aspetti non specificati dal testo. Vengono ora riportate:

- Per semplicità, decidiamo di rappresentare gli indirizzi con dei numeri interi
- Dati n diversi indirizzi di consegna, essi saranno i primi n numeri interi
- Dati n diversi indirizzi di consegna, l'indirizzo dell'ufficio postale sarà l'$n+1$-esimo numero intero
- Sia k il più alto numero di lettere da consegnare ad un singolo indirizzo, assumiamo ragionevolmente che la borsa del postino avrà una capacità di almeno k buste
- Assumiamo che il grafo della città sia un *num_nodi*-clique.

2.2 Vincoli

Dall'analisi del problema presentato sopra possiamo ricavare i vincoli del nostro COP. Vengono ora riportati:

1. Il percorso del postino deve cominciare dall'ufficio postale
2. Il percorso del postino deve terminare all'ufficio postale
3. Il postino deve consegnare tutte le buste assegnate
4. Il postino non può portare trasportare più di h buste contemporaneamente

2.3 Funzione da ottimizzare

Come precedentemente spiegato, nei COP, a differenza dei *Constraint Satisfaction Problem*, abbiamo anche una funzione da minimizzare o massimizzare. Questa funzione, in questo caso, misura la strada che il postino compie per consegnare tutte le lettere.

Come in uno scenario reale, il nostro obiettivo sarà quella di minimizzare questa funzione, ovvero trovare il percorso più breve che permetterà al postino di consegnare tutte le lettere, rispettando i vincoli elencati in precedenza.

3 MiniZinc

In questa sezione viene spiegato il modello per il COP scritto in MiniZinc, analizzando il codice scritto e spiegandolo.

3.1 Parametri

Inizialmente vengono dichiarati i valori che dovranno essere passati in input, ovvero: il grafo della città, rappresentato dai *nodi* e dalle *distanze* tra essi; l'elenco delle *lettere da consegnare*; l'indirizzo dell'*ufficio postale*; la *dimensione della borsa*.

```
int: num_nodi;
int: max_step;
int: ufficio_postale;
int: dim_borsa;

array [1..num_nodi] of int: elenco_lettere;
array [1..num_nodi, 1..num_nodi] of int: distanze;
```

Tra i dati di input, oltre a quelli elencati sopra, viene passato anche *max_step*, ovvero il numero massimo di passi che può compiere il postino. Questo valore non influisce sul risultato teorico del problema, in quanto sarà sufficientemente alto, però, se non calcolato correttamente, con alcune tecniche di ricerca potrebbe peggiorare di molto le performance.

3.2 Variabili

Dopo i parametri del programma, vengono dichiarate le variabili, ovvero i valori che manipolerà effettivamente il modello. In questo problema, le variabili sono soltanto due: il percorso che effettuerà il postino e il costo totale di esso.

```
array [1..max_step] of var 0..num_nodi: percorso;
var int: costo;
```

3.3 Vincoli

Dopo le variabili, vengono dichiarati i vincoli, ovvero la parte più importante del modello.

Il primo vincolo, come indicato nell'elenco descritto nella sezione precedente, richiede che il percorso del postino cominci dall'ufficio postale. Di conseguenza, al t_1, ovvero alla posizione 1 del vettore *percorso*, il postino deve trovarsi all'indirizzo dell'ufficio postale.

```
constraint percorso[1] = ufficio_postale;
```

Il secondo vincolo, molto simile al primo, richiede che il percorso del postino termini all'ufficio postale. Non conoscendo però il tempo t_i alla quale il postino terminerà il suo percorso e non potendo fare in MiniZinc un array di dimensione variabile, ho optato per la seguente implementazione: ho considerato la dimensione del vettore *percorso* pari a *max_step*, numero sufficientemente grande; successivamente ho deciso di riempire con il numero 0 tutte quelle posizioni finali del vettore in cui il postino non si muove più. Così facendo, l'ultima posizione del postino sarà quella cella che precede la sequenza di zeri.

```
constraint exists(i in 1..max_step)
       (percorso[i] == ufficio_postale /\ forall (j in i+1..max_step)(percorso[j] == 0));
```

Il terzo vincolo richiede che tutte le lettere in consegna vengano consegnate nel percorso del postino. Grazie all'assunzione sulla capacità minima della borsa del postino, siamo sicuri che basterà visitare un nodo almeno una volta per soddisfare il vincolo.

```
constraint forall(i in 1..num_nodi where elenco_lettere[i] > 0)
                (exists(j in 1..max_step)(percorso[j] == i));
```

Il quarto vincolo impone al postino di non poter portare con se più di *dim_borsa* lettere contemporaneamente. Considerando gli intervalli ufficio postale - ufficio postale, ne deve sempre esistere uno la cui somma delle lettere consegnate (quindi nodi non ancora visitati) nell'intervallo è minore o uguale alla *dim_borsa*.

```
constraint forall(i in 1..max_step-1 where percorso[i] == ufficio_postale /\ percorso[i+1] != 0)
           (exists(j in i+1..max_step where percorso[j] == ufficio_postale)
               (sum(k in i..j where not exists(z in 1..k-1)(percorso[z] == percorso[k])
                   ((elenco_lettere[percorso[k]])) <= dim_borsa));
```

L'ultimo vincolo definisce la funzione da minimizzare, ovvero la funzione che calcola la distanza percorso da postino durante il suo tragitto.

```
constraint costo = sum(i in 1..max_step-1 where percorso[i+1] != 0)
                     (distanze[percorso[i],percorso[i+1]]);
```

3.4 Obiettivo

Come già spiegato più volte, l'obiettivo di questo COP è minimizzare la funzione *costo*, cercando di trovare un percorso che rispetti i vincoli elencati e che sia il più corto possibile.

```
solve minimize costo;
```

4 Answer Set Programming

In questa sezione viene spiegato il modello per il COP scritto in ASP, analizzando il codice scritto e spiegandolo.

4.1 Fatti

Inizialmente analizziamo i fatti, ovvero quei predicati veri, che rappresentano l'input del problema.

Per rappresentare i vari indirizzi della città utilizzeremo il predicato unario *node*. Per rappresentare l'indirizzo dell'ufficio postale utilizzeremo il predicato unario $ufficio\_postale$. Per rappresentare la capienza della borsa utilizzeremo il predicato unario *dim_borsa*. Per indicare il numero di lettere da consegnare ad ogni indirizzo utilizzeremo il predicato binario $lettere(X, Q)$, in cui X è un nodo e Q è la quantità di lettere da consegnare (0 nel caso dell'ufficio postale e nel caso degli indirizzi che non devono ricevere lettere). Per indicare la distanza tra due nodi X e Y utilizzeremo il predicato ternario $costo\_arco(X, Y, N)$, in cui N rappresenterà appunto la distanza. Infine, come nel modello di MiniZinc in cui abbiamo il parametro *max_step*, avremo il predicato unario *time* che rappresenta i vari passi che potrà effettuare il postino.

4.2 Vincoli

Analizziamo ora i predicati introdotti e le regole scritte al fine di soddisfare i vincoli del nostro problema. Inoltre, analizzeremo anche alcune regole necessarie per un funzionamento corretto e sensato del modello.

Prima di cominciare l'analisi delle varie regole, introduciamo un predicato fondamentale: $postino(X, T)$ che indica la posizione X del postino al tempo T.

- Il primo vincolo richiede che il percorso del postino cominci dall'ufficio postale. La seguente regola garantisce che questo vincolo venga rispettato.

```
1{postino(X, 1): node(X)}1 :- ufficio_postale(X).
```

- Il secondo vincolo richiede che il percorso del postino termini all'ufficio postale. Per soddisfare questo vincolo scriviamo la seguente regola che includerà il predicato $postino(X, T)$. La seguente regola garantisce che, se vale un determinato T ma non il successivo, allora il postino deve trovarsi all'ufficio postale.

  ```
  postino(X, T1) :- not time(T2), time(T1), T2 = T1+1, ufficio_postale(X).
  ```

- Il terzo vincolo richiede che tutte le buste vengano consegnate. Per soddisfare questa vincolo introduciamo la seguente regola.

  ```
  1{postino(N, T): time(T)} :- lettere(N, Q), node(N), Q > 0.
  ```

- Il quarto vincolo riguarda la capienza della borsa del postino. Per soddisfare questo vincolo introduciamo due predicati: il primo, $contabuste(S, T1, T2)$ indica il numero S di buste consegnate dal postino nell'intervallo T1 e T2 (a questi tempi il postino si trova all'ufficio postale); il secondo invece, $visited(X, T1, T2)$, indica se un nodo X è stato visitato nell'intervallo di tempo T1 - T2.

  ```
  visited(X, T1, T2) :- time(T1), time(T2), time(T3), postino(X,T3), T1 < T3 < T2.

  contabuste(S, T1, T2) :- time(T1), time(T2), time(T4), T1 < T4 < T2, postino(Y, T1),
                           postino(Y, T2), not visited(Y,T1,T2), ufficio_postale(Y),
                           S = #sum{N: not visited(X, 1, T3), postino(X,T3), lettere(X, N),
                           node(X) ,time(T3), time(T5), T5 < T3, T1 < T3 < T2}.
  ```

  Infine, come richiesto dal vincolo, scriviamo la regola che indica che non può esistere un $contabuste(S, T1, T2)$ con una S maggiore della dimensione della dimensione della borsa del postino.

  ```
  :- contabuste(S, T1, T2), dim_borsa(N), S > N, time(T1), time(T2).
  ```

- Infine, come specificato sopra, vengono introdotte alcune regole che garantiscono un funzionamento sensato e corretto del modello.

  La seguente regola garantisce che, per ogni $time(T)$ valga esattamente un $postino(X, T)$.

  ```
  1{postino(X,T): node(X)}1 :- time(T).
  ```

  La seguente regola assicura che il postino, ad un determinato tempo T, si trovi in una sola posizione X.

  ```
  :- node(X), node(Y), time(T), postino(X,T), postino(Y,T), X != Y.
  ```

4.3 Obiettivo

Come precedentemente detto, l'obiettivo di questo COP è minimizzare il percorso che il postino compie per consegnare tutte le buste. Per misurare il "costo" percorso svolto utilizziamo il predicato $score(S)$, che dichiariamo come segue:

```
1{score(S)}1 :- S = #sum{N, T, T+1: postino(X,T), postino(Y,T+1),
                         costo_arco(X,Y,N), node(X), node(Y), time(T)}.
```

Il valore di questo predicato va ovviamente minimizzato e per fare ciò scriviamo la seguente riga:

```
#minimize {S: score(S)}.
```

Infine, per mostrare i risultati ottenuti, mostriamo i valori dei predicati veri. Così facendo, renderemo conto del percorso effettuato dal postino, del costo di esso e del numero di buste consegnati in ogni intervallo ufficio postale - ufficio postale, significativo.

```
#show postino/2.
#show score/1.
#show contabuste/3.
```

5 Test e risultati

Dopo la fase di progettazione dei modelli, segue la fase di test dei modelli. La consegna del progetto richiede la preparazione di una batteria di test, di diversa difficoltà. Per ogni livello, ne prepariamo dieci, generati randomicamente utilizzando lo script python *generatore.py*.

5.1 Creazione dei test

In questa sottosezione spieghiamo come vengono generati i dati di test, in base al loro livello di difficoltà. Per fare ciò elenchiamo prima di tutto i parametri che determinano la complessità del problema. Essi sono:

- **Numero di nodi**: maggiore è il numero di nodi, maggiore sarà la dimensione dell'albero di ricerca e di conseguenza, il tempo impiegato dal solver per determinare la soluzione
- **Numero di indirizzi di consegna**: maggiore sarà il numero di indirizzi alla quale il postino dovrà obbligatoriamente recarsi, più complessa sarà l'istanza del problema
- **Dimensione della borsa**: minore sarà la dimensione della borsa, maggiore sarà il numero di volte in cui il postino sarà costretto a tornare all'ufficio postale
- **Max step**: maggiore sarà il numero di step ammessi, più complessa sarà l'istanza del problema

Ora spieghiamo come sono state generate le istanza, in base al loro livello di difficoltà:

- **Easy**: il numero di nodi varia tra 4 e 6; il numero di indirizzi di consegna corrisponde al 30-40-50% (scelto randomicamente) dei nodi totali; la dimensione della borsa sarà pari al massimo numero di lettere da consegnare ad un singolo indirizzo, moltiplicato per il numero totale di nodi; infine, il numero massimo di step è pari a due volte il numero dei nodi.
- **Medium**: il numero di nodi varia tra 6 e 8; il numero di indirizzi di consegna corrisponde al 60-70% (scelto randomicamente) dei nodi totali; la dimensione della borsa sarà pari al massimo numero di lettere da consegnare ad un singolo indirizzo, moltiplicato per 2; infine, il numero massimo di step è pari a due volte e mezzo il numero dei nodi.
- **Hard**: il numero di nodi varia tra 8 e 10; il numero di indirizzi di consegna corrisponde al 80-90% (scelto randomicamente) dei nodi totali; la dimensione della borsa sarà pari al massimo numero di lettere da consegnare ad un singolo indirizzo; infine, il numero massimo di step è pari a quattro volte il numero dei nodi.

La scelta dei valori dei parametri è stata fatta in maniera sperimentale, ovvero provando a generare delle istanze e testandole. L'obiettivo era quello di raggiungere più o meno i tempi richiesti dalla consegna, ovvero qualche secondo per le istanze *easy*, qualche minuto per le istanze *medium*, timeout per le istanze *hard*.

5.2 Esperimenti e risultati

I test condotti sono stati eseguiti su un MacBook Pro con processore M1 e 8GB di RAM. Di seguito viene riportata la tabella 1 con i risultati ottenuti, con il tempo, indicato tra parentesi, espresso in secondi.

5.3 Analisi dei risultati

Analizzando i risultati possiamo notare che l'ASP solver Clingo è sicuramente quello che ha performato nella maniera peggiore, in quanto ha raggiunto il timeout in tutti i test con difficoltà *medium* e *hard*, oltre a due test con difficoltà *easy*.

Analizzando invece i risultati prodotti dai solver utilizzati per il modello scritto in MiniZinc, notiamo che i risultati sono migliori. Gecode, a differenza di Chuffed, raggiunge il timeout più spesso. Però, quando entrambi lo raggiungono, Gecode trova sempre un risultato migliore rispetto a Chuffed. Quest'ultimo infatti, grazie alla sua strategia operativa, è molto veloce a produrre una prima soluzione iniziale, a differenza di Gecode che in molti test di difficoltà *hard* non ne ha prodotta nemmeno una.

Considerati i risultati, e considerato il fatto che non abbiamo riscontrato casi in cui Chuffed ha raggiunto il timeout e gli altri solver no, possiamo considerare Chuffed come il solver più veloce.

Istanza	*Gecode*	*Chuffed*	*Clingo*
Easy 1	12 [0.183]	12 [0.157]	12 [0.443]
Easy 2	28 [0.160]	28 [0.142]	28 [0.386]
Easy 3	22 [0.158]	22 [0.143]	28 [0.357]
Easy 4	30 [0.157]	30 [0.144]	30 [0.582]
Easy 5	10 [0.143]	10 [0.139]	10 [0.396]
Easy 6	31 [0.224]	31 [0.219]	31 [92.315]
Easy 7	29 [0.223]	29 [0.250]	29 [96.127]
Easy 8	31 [0.254]	31 [0.255]	31 [76.738]
Easy 9	23 [0.452]	23 [0.425]	34 [TIMEOUT]
Easy 10	26 [0.373]	26 [0.297]	39 [TIMEOUT]
Medium 1	35 [37.848]	35 [14.660]	60 [TIMEOUT]
Medium 2	54 [124.001]	54 [20.980]	80 [TIMEOUT]
Medium 3	32 [13.250]	32 [1.242]	83 [TIMEOUT]
Medium 4	37 [5.675]	37 [3.809]	78 [TIMEOUT]
Medium 5	34 [TIMEOUT]	34 [169.002]	57 [TIMEOUT]
Medium 6	37 [TIMEOUT]	37 [277.000]	66 [TIMEOUT]
Medium 7	36 [TIMEOUT]	36 [54.334]	71 [TIMEOUT]
Medium 8	45 [TIMEOUT]	45 [164.00]	95 [TIMEOUT]
Medium 9	41 [TIMEOUT]	61 [TIMEOUT]	79 [TIMEOUT]
Medium 10	58 [TIMEOUT]	75 [TIMEOUT]	85 [TIMEOUT]
Hard 1	UNK [TIMEOUT]	178 [TIMEOUT]	UNK [TIMEOUT]
Hard 2	141 [TIMEOUT]	172 [TIMEOUT]	UNK [TIMEOUT]
Hard 3	91 [TIMEOUT]	125 [TIMEOUT]	UNK [TIMEOUT]
Hard 4	UNK [TIMEOUT]	193 [TIMEOUT]	UNK [TIMEOUT]
Hard 5	UNK [TIMEOUT]	153 [TIMEOUT]	UNK [TIMEOUT]
Hard 6	UNK [TIMEOUT]	171 [TIMEOUT]	UNK [TIMEOUT]
Hard 7	UNK [TIMEOUT]	UNK [TIMEOUT]	UNK [TIMEOUT]
Hard 8	UNK [TIMEOUT]	159 [TIMEOUT]	UNK [TIMEOUT]
Hard 9	UNK [TIMEOUT]	203 [TIMEOUT]	UNK [TIMEOUT]
Hard 10	UNK [TIMEOUT]	171 [TIMEOUT]	UNK [TIMEOUT]

Tabella 1: Risultati test