Progetto di Ragionamento Automatico

Roberto Borelli
borelli.roberto@spes.uniud.it

Indice

1 Introduzione

Presentazione del progetto. L'obiettivo di questo lavoro è quello di risolvere un semplice problema di ottimizzazione utilizzando opportune tecniche di *knowledge rapresentation*. In particolare, dopo aver presentato il problema vedremo come risolverlo in programmazione logica utilizzando l'*answer set programming* (ASP), vedremo la sua codifica Minizinc e infine vedremo la formulazione in programmazione lineare intera. Nell'ultima sezione confronteremo tutte le soluzioni trovate effettuando opportuni test delle performance sui vari modelli.

La richiesta del problema. Nel problema proposto dobbiamo trovare una pianificazione ottimale delle gare di un atleta in modo che alcuni vincoli siano rispettati. Vengono date le informazioni di n gare che si svolgeranno durante l'anno corrente, di ogni gara conosciamo il giorno in cui si svolgerà, la sua lunghezza (ossia i chilometri che l'atleta dovrà correre) e la distanza da casa del luogo in cui si terrà l'evento. Vogliamo trovare un sottoinsieme di gare che soddisfi i seguenti vincoli:

[UNICITY] Se ci sono più gare in uno stesso giorno, l'atleta parteciperà al più ad una di esse.

[15KM] Se l'atleta corre ad una gara con lunghezza $\geq$ 15km, l'atleta non parteciperà a gare di lunghezza maggiore o uguale a 10km per almeno 10 giorni.

[MAR-MAR] Se l'atleta partecipa ad una maratona, aspetta almeno 6 settimane (42 giorni) per partecipare ad una nuova maratona.

[MAR-MZ] Se l'atleta partecipa ad una maratona, aspetta almeno 4 settimane (28 giorni) per partecipare ad una mezza maratona.

[MZ-MZ] Tra due mezze maratone l'atleta aspetta almeno 3 settimane (21 giorni).

[BUDGET] Supponendo che la benzina costi C euro/litro e che la macchina con cui viaggia l'atleta percorra in media K km/litro. Sia B il budget di soldi a disposizione. Vogliamo verificare che complessivamente i soldi spesi in benzina per andare e tornare dalle gare non superino la soglia B.

Tra tutte le possibili combinazioni di gare vogliamo inoltre scegliere solo quelle che massimizzano la distanza percorsa dall'atleta. Da quest'ultima richiesta capiamo dunque che ci troviamo di fronte ad un problema di ottimizzazione.

Alcuni limiti superiori. Per ottimizzare la risoluzione dei modelli che discuteremo in seguito, pensando al vincolo [UNICITY], possiamo immediatamente pensare a quale sarà il valore massimo che alcune variabili assumeranno. In particolare, supponiamo che data un'istanza abbiamo calcolato una soluzione ammissibile che soddisfa i vincoli sopra descritti, allora possiamo sicuramente affermare che:[1]

[UBCT] La corsa totale dell'atleta sarà minore o uguale a c dove si calcola c nel seguente modo. Sia g un giorno in cui c'è almeno una gara. Sia c_g la lunghezza massima di una gara programmata per il giorno g, allora $c = \sum_g c_g$.

[UBDT] La distanza totale che l'atleta percorrerà (in auto) sarà minore o uguale a d dove d è calcolato analogamente a c. Sia g un giorno in cui c'è almeno una gara. Sia d_g la distanza da casa massima di una gara programmata per il giorno g, allora $d = 2\sum_g d_g$.

Ipotesi. Le assunzioni effettuate sono le seguenti:

1. La lunghezza e la distanza da casa di una gara sono dei numeri interi.
2. Una maratona è lunga 42 km e una mezza è lunga 21km.

[1] Nella notazione che segue, UB sta per Upper Bound, CT sta per Corsa Totale, DT per Distanza Totale.

2 Modello ASP

Partiamo col modellare il problema in ASP. In questo caso l'input è fornito da una serie di atomi della forma:

```
gara(id, giorno, lunghezza_gara, distanza_da_casa)
```

`id` è un oggetto non necessariamente numerico che identifica univocamente le n gare e grazie a questo campo possiamo codificare anche due gare i e j che sono state programmate per lo stesso giorno, che hanno la stessa distanza da casa e che sono lunghe uguali. Si noti che di fronte a questa situazione, per soddisfare i vincoli del problema e per massimizzare la funzione obiettivo, se una soluzione include i allora esiste un'altra soluzione con le stesse proprietà dove non compare i, ma compare la gara j. Il risultato verrà fornito attraverso il seguente predicato.

```
partecipa(id)
```

Seguono le definizioni di alcuni predicati base.

```
giorno(1..366).    % enumerazione dei giorni
lunghezza(1..50).  % lunghezza della gara
maratona(42).      % lunghezza di una maratona
mezza(21).         % lunghezza di una mezza maratona
distanza(1..1000). % distanza da casa
```

Con il predicato `soglia_distanza` memorizziamo la distanza massima percorribile rispettando il vincolo del problema [BUDGET]. Vale la pena notare che è importante che la divisione venga effettuata come ultima operazione, in modo da minimizzare la perdita di informazione dovuta all'aritmetica sui numeri interi.

```
budget(30000).      % soldi (euro) a disposizione per i viaggi
costo_benzina(190). % costo in centesimi della benzina
km_litro(12).       % quanti chilometri percorro con un litro
soglia_distanza(S) :- S=((B*100*K)/C), km_litro(K), costo_benzina(C), budget(B).
```

Descriviamo ora le definizioni dei predicati principali. Codifichiamo il vincolo [UNICITY] tramite un *cardinality constraint*: se G è un giorno, l'atleta parteciperà al più ad una gara nel giorno G.

```
% [UNICITY]
0{partecipa(I) : gara(I,G,L,D)}1 :- giorno(G).
```

Ora dobbiamo codificare i vincoli [15KM], [MAR-MAR], [MAR-MZ] e [MZ-MZ]. Prendiamo ad esempio il caso del vincolo [MAR-MZ] (gli altri casi sono analoghi). Diciamo che se l'atleta partecipa alla maratona `I1` il giorno `G1`, allora l'atleta non può partecipare ad una mezza maratona `I2` il cui giorno è nell'intervallo $(\texttt{G1}, \texttt{G1} + 28]$.

```
% vincolo [15km]
:- partecipa(I2), gara(I2,G2,L2,D2), L2>=10,
partecipa(I1), gara(I1,G1,L1,D1), L1 >= 15, G2 > G1, G2 <= G1 + 10.

% vincolo [MAR-MAR]
:- partecipa(I2), gara(I2,G2,L2,D2), maratona(L2),
partecipa(I1), gara(I1,G1,L1,D1), maratona(L1), G2 > G1, G2 <= G1 + 42.

% vincolo [MAR-MZ]
:- partecipa(I2), gara(I2,G2,L2,D2), mezza(L2),
partecipa(I1), gara(I1,G1,L1,D1), maratona(L1), G2 > G1, G2 <= G1 + 28.

% vincolo [MZ-MZ]
:- partecipa(I2), gara(I2,G2,L2,D2), mezza(L2),
partecipa(I1), gara(I1,G1,L1,D1), mezza(L1), G2 > G1, G2 <= G1 + 21.
```

Tramite l'aggregato `#sum` calcoliamo la distanza totale da casa (andata e ritorno) delle gare a cui l'atleta parteciperà e la corsa totale che dovrà sostenere. Ricordiamo che la semantica dell'aggregato è basata sugli insiemi e non sui multi-insiemi quindi dobbiamo considerare nei due casi rispettivamente le coppie `<distanza, id>` e `<lunghezza, id>` che sono uniche in quanto l'attributo `<id>` identifica univocamente ogni gara.

```
distanza_totale(T) :- P = #sum{D,I : partecipa(I), gara(I,G,L,D)}, T = 2*P.
corsa_totale(T)    :- T = #sum{L,I : partecipa(I), gara(I,G,L,D)}.
```

Gli upper bound [UBCT] e [UBDT] vengono calcolati tramite l'ausilio dell'aggregato `#max`. In questo caso non è necessario considerare le coppie `<lunghezza, id>` e `<distanza, id>` all'interno dell'aggregato.

```
% [UBCT]
max_c_giorno(G,MC) :- MC = #max{0; L : gara(I,G,L,D)}, giorno(G).
ubct(T) :- T = #sum{MC,G : max_c_giorno(G, MC)}.
:- ubct(C1), corsa_totale(C2), C2 > C1.

% [UBDT]
max_d_giorno(G,MD) :- MD = #max{0; D : gara(I,G,L,D)}, giorno(G).
ubdt(T) :- P = #sum{MD,G : max_d_giorno(G, MD)}, T = 2*P.
:- ubdt(T1), distanza_totale(T2), T2 > T1.
```

Il vincolo [BUDGET] viene codificato dal seguente codice che asserisce che non può essere che si partecipi ad un insieme di gare tali che la loro somma delle distanze raddoppiata (consideriamo andata e ritorno) superi il budget acconsentito.

```
:- distanza_totale(T), soglia_distanza(T1), T > T1.
```

La funzione obiettivo viene codificata grazie allo statment di ottimizzazione `#maximize`.

```
#maximize{T : corsa_totale(T)}.
```

Concludiamo mostrando l'output nella forma di un predicato ternario `partecipa`.

```
partecipa(G,L,D) :- partecipa(I), gara(I,G,L,D).
#show partecipa/3.
#show corsa_totale/1.
#show distanza_totale/1.
#show soglia_distanza/1.
```

3 Modello Minizinc

```
include "globals.mzn";
par int: budget = 30000;       % euro
par int: costo_benzina = 190;  % centesimi
par int: km_litro = 12;
par int: soglia_distanza = floor((budget * 100 * km_litro) / costo_benzina);

par int: maratona = 42;
par int: mezza = 21;
par int: max_giorno = 366;
par int: max_lunghezza = 50;
par int: max_distanza = 1000;
```

Oltre ai parametri sopra citati, l'input sarà costituito da tre array (lunghi n) `giorni`, `lunghezze` e `distanze` in cui il significato è che fissato $1 \leq i \leq n$ c'è una gara il giorno `giorni[i]` lunga `lunghezze[i]` e distante da casa `distanze[i]`.

```
par int: n=28;
array [1..n] of par 1..max_giorno: giorni;
array [1..n] of par 1..max_lunghezza: lunghezze;
array [1..n] of par 1..max_distanza: distanze;
```

Tramite un array aggiungiamo una variabile booleana `partecipa[i]` per ogni $1 \leq i \leq n$ con il seguente significato:

$$\texttt{partecipa[i]} = \begin{cases} \texttt{true} & \text{l'atleta parteciperà alla gara } i \\ \texttt{false} & \text{altrimenti} \end{cases} \tag{1}$$

```
array [1..n] of var bool: partecipa;
```

Per codificare il vincolo [UNICITY] la scelta migliore è quella di utilizzare un constraint globale. In particolare andremo ad utilizzare una variante del noto `alldifferent` chiamata `alldifferent_except_0` che forza un insieme di variabili a prendere valore differente con l'eccezione che è ammesso che multipli valori siano pari a 0. Consideriamo il multi-insieme $G = \{\texttt{partecipa[j]*giorni[j]} \mid j = 1 \dots n\}$. Sia k il numero di gare a cui l'atleta non parteciperà, allora lo 0 comparirà k volte in G (in breve $0 \in^k G$) e per ogni gara i a cui l'atleta partecipa vale `giorni[i]` $\in^{m_i} G$ per $m_i > 0$. Il constraint `alldifferent_except_0(G)` forzerà le variabili `participa[i]` ad assumere valori tali che vale $m_i = 1$ per ogni i in cui vale `partecipa[i]`.

```
% [UNICITY]
constraint alldifferent_except_0([partecipa[j]*giorni[j] | j in 1..n]);
```

Implementiamo ora i vincoli [15KM], [MAR-MAR], [MAR-MZ] e [MZ-MZ]. Prendiamo il caso di [MAR-MZ]: imponiamo che se il giorno g l'atleta partecipa ad una maratona, la somma delle partecipazioni delle mezze maratone che occorrono tra il giorno $g + 1$ e $g + 28$ (estremi inclusi) deve essere nulla. Ossia ogni mezza maratona j che occorre in cui quei giorni dovrà avere un contributo `partecipa[j]` nullo.

```
% [15KM]
constraint forall(i in 1..n)
    (partecipa[i] == true /\ lunghezze[i] >= 15 ->
        (sum (j in 1..n where giorni[j] > giorni[i] /\ giorni[j] <= giorni[i] + 10
            /\ lunghezze[j]>=10)
                (partecipa[j])) == 0
    );

% [MAR-MAR]
```

```
constraint forall(i in 1..n)
    (partecipa[i] == true /\ lunghezze[i] == maratona ->
        (sum (j in 1..n where giorni[j] > giorni[i] /\ giorni[j] <= giorni[i] + 42
            /\ lunghezze[j] == maratona)
                (partecipa[j])) == 0
    );

% [MAR-MZ]
constraint forall(i in 1..n)
    (partecipa[i] == true /\ lunghezze[i] == maratona ->
        (sum (j in 1..n where giorni[j] > giorni[i] /\ giorni[j] <= giorni[i] + 28
            /\ lunghezze[j] == mezza)
                (partecipa[j])) == 0
    );

% [MZ-MZ]
constraint forall(i in 1..n)
    (partecipa[i] == true /\ lunghezze[i] == mezza ->
        (sum (j in 1..n where giorni[j] > giorni[i] /\ giorni[j] <= giorni[i] + 21
            /\ lunghezze[j] == mezza)
                (partecipa[j])) == 0
    );
```

Calcoliamo l'upper bound [UBDT] e imponiamo il vincolo [BUDGET].

```
par int: ubdt = 2 * (sum(g in 1..max_giorno)
                  (max([0]++[distanze[i] | i in 1..n where giorni[i] == g])));
var 0..ubdt: distanza_totale = sum (i in 1..n)
(partecipa[i] * distanze[i] * 2);
constraint distanza_totale <= soglia_distanza;
```

Massimizziamo la corsa totale e implementiamo l'upper bound [UBCT].

```
par int: ubct = sum(g in 1..max_giorno)
                 (max([0]++[lunghezze[i] | i in 1..n where giorni[i] == g]));
var 0..ubct: corsa_totale = sum (i in 1..n)(partecipa[i] * lunghezze[i]);
solve maximize corsa_totale;
```

Per completezza mostriamo come stampare l'output alla maniera di ASP. Con il seguente codice la scelta delle gare da effettuare sarà stampata tramite una serie di atomi della forma vista in sezione 2 `partecipa(giorno,lunghezza,gara)`.

```
output[
    (if (fix(partecipa[i]) == true) then
        ("partecipa("++show(giorni[i])++","++show(lunghezze[i])++","++show(distanze
            [i])++")")
    else ""
    endif)++
    (if (i == n) then
        ("corsa_totale("++show(corsa_totale)++")"++
        "distanza_totale("++show(distanza_totale)++")"++
        "soglia_distanza("++show(soglia_distanza)++")")
    else ""
    endif) | i in 1..n];
```

3.1 Euristiche di ricerca

In questa sottosezione descriviamo due possibili strategie di ricerca che poi valuteremo in sezione 5.4.

Nella prima strategia (che chiameremo S1) cerchiamo le variabili `partecipa`, assegnando un valore random tra 0 e 1 procedendo in ordine in base alla variabile che ha più vincoli associati. Imponiamo inoltre un restart lineare che ci aiuterà nel caso in cui all'inizio dell'albero di ricerca sia stata presa una scelta sbagliata. Notiamo che affinché il restart sia sensato ci devono necessariamente essere delle scelte non deterministiche altrimenti rivisiteremmo esattamente lo stesso ramo.

```
1 solve :: bool_search(partecipa,occurrence,indomain_random)
2       :: restart_linear(2*n)
3 maximize corsa_totale;
```

La seconda strategia (che chiamiamo S2) è molto simile a S1 ma cambia l'ordine con cui processiamo le variabili. Prenderemo le variabili in ordine in base al numero dei vincoli associati diviso il numero di volte che hanno causato un fallimento.

```
1 solve :: bool_search(partecipa,dom_w_deg,indomain_random)
2       :: restart_linear(2*n)
3 maximize corsa_totale;
```

4 Modello ILP

Il problema presentato in sezione 1 ammette una modellazione in programmazione lineare intera (PLI) ossia un modello COP in cui la funzione obiettivo e i vincoli sono delle disequazioni lineari e le variabili possono assumere solo valori interi.

Consideriamo un input di n gare dove una gara è una tripla del tipo (g_i, l_i, d_i). Sia B il budget massimo a disposizione, sia C il costo al litro della benzina e sia K il numero di chilometri che la macchina percorre con un libro. Introduciamo nel modello una variabile binaria x_i per ogni gara $i = 1 \dots n$. Il significato è lo stesso dell'array `partecipa` usato in sezione 3: $x_i = 1$ sse la gara i-ma sarà inclusa nella soluzione e $x_i = 0$ sse la gara i-ma non farà parte della soluzione. Il modello di PLI è allora il seguente.

$$\max \sum_{i=1}^{n} x_i l_i \tag{2}$$

soggetto a:

$$x_i + x_j \leq 1 \qquad \forall i, j \in [n], i \neq j, g_i = g_j \tag{3}$$

$$x_i + x_j \leq 1 \qquad \forall i, j \in [n], i \neq j, l_i \geq 15, l_j \geq 10, g_i < g_j \leq g_i + 10 \tag{4}$$

$$x_i + x_j \leq 1 \qquad \forall i, j \in [n], i \neq j, l_i = \text{maratona}\ , l_j = \text{maratona}\ , g_i < g_j \leq g_i + 42 \tag{5}$$

$$x_i + x_j \leq 1 \qquad \forall i, j \in [n], i \neq j, l_i = \text{maratona}\ , l_j = \text{mezza}\ , g_i < g_j \leq g_i + 28 \tag{6}$$

$$x_i + x_j \leq 1 \qquad \forall i, j \in [n], i \neq j, l_i = \text{mezza}\ , l_j = \text{mezza}\ , g_i < g_j \leq g_i + 21 \tag{7}$$

$$\sum_{i=1}^{n} x_i d_i 2 \leq S \qquad S = \frac{BK}{C} \text{ con } B,\ K,\ C \text{ parametri di input} \tag{8}$$

$$\sum_{i=1}^{n} x_i d_i 2 \leq UBDT \qquad UBDT = 2 \sum_{g=1}^{366} \max \{d_i \mid g_i = g\} \quad \text{(calcolato in pre-processing)} \tag{9}$$

$$\sum_{i=1}^{n} x_i l_i \leq UBCT \qquad UBCT = \sum_{g=1}^{366} \max \{l_i \mid g_i = g\} \quad \text{(calcolato in pre-processing)} \tag{10}$$

La funzione obiettivo (equazione 2) massimizza la corsa totale dell'atleta. I vincoli 3,4,5,6,7,8 codificano rispettivamente i vincoli del problema [UNICITY], [15KM], [MAR-MAR], [MAR-MZ], [MZ-MZ] e [BUDGET]. Se i e j sono due gare allora una disequazione del tipo

$$x_i + x_j \leq 1 \tag{11}$$

significa che al più solamente una gara tra i e j sarà inclusa nella soluzione. I vincoli 9 e 10 implementano gli upper bound [UBDT] e [UBCT]. Notiamo infine che fissata un'istanza del problema, i vincoli saranno al più dell'ordine di $O(n^2)$ quindi una quantità polinomiale rispetto all'input. Avendo ora a disposizione un modello di PLI, abbiamo deciso di testare due implementazioni di seguito descritte.

4.1 Codifica in Minizinc

A partire dal modello PLI abbiamo creato un nuovo modello Minizinc differente da quello visto in sezione 3. Anche questo nuovo modello può essere dunque risolto con un constraint solver come Gecode. Il nuovo modello minizinc avrà stessa funzione obiettivo, stessi parametri e stesse variabili, e stessi upper bound, cambia solo come codifichiamo i vincoli. Il vincolo [UNICITY] è codificato dal seguente codice.

```
% [UNICITY]
constraint forall(i in 1..n, j in i+1..n)
  (if (giorni[i] == giorni[j]) then (partecipa[i] + partecipa[j]<= 1) endif);
```

Il vincolo [MAR-MZ] è codificato invece nel seguente modo.

```
% [MAR-MZ]
constraint forall(i,j in 1..n where i!=j)
  (forall(l in 1..max_giorno)
    (if(lunghezze[i] == maratona /\ lunghezze[j] == mezza /\
        giorni[j] <= giorni[i]+28 /\ giorni[j] > giorni[i]) then
        (partecipa[i] + partecipa[j] <= 1)
    endif)
  );
```

I vincoli [MAR-MAR], [15KM] e [MZ-MZ] si codificano in maniera del tutto analoga e per brevità omettiamo il codice che risulterebbe solo un duplicato. In sezione 5 valuteremo quale dei due modelli si comporta meglio. Possiamo fin da subito notare un'enorme differenza: il modello della sezione 3 faceva uso di un constraint globale per codificare il vincolo [UNICITY] mentre questo no, quindi potrebbe essere ragionevole aspettarsi che il modello minizinc visto in questa sezione abbia performance peggiori a parità di solver.

4.2 Codifica per uno specifico ILP solver

Abbiamo creato un programma C, che riceve in input le informazioni sulle n gare e stampa in output il modello PLI in formato *CPLEX lp*. Per risolvere il modello *lp* creato è ora sufficiente usare un solver di PLI come Cplex o Glpk. In sezione 5 valuteremo le performance con il solver open-source Glpk che implementa una versione *vanilla*[2] del simplesso unita all'algoritmo *branch and cut*. Con questa ulteriore implementazione possiamo dunque testare:

- Come un ILP solver (che riceve un modello ILP in un formato come CPLEX lp) si comporta rispetto ad un constraint solver che riceve in input un modello Minizinc che codifica disequazioni lineari (come il modello appena discusso in sezione 4.1).
- Come un constraint solver che lavora su un modello che sfrutta le piene potenzialità di Minizinc (e quindi riceve in input un modello che fa uso di constraint globali ad esempio) si comporta rispetto ad un ILP solver.

[2]Ossia implementa solo il simplesso, senza applicare ulteriori ottimizzazioni.

5 Valutazione delle performance

In questa sezione ci occupiamo di testare tutti i modelli delle sezioni precedenti in modo da capire come si comportano su un determinato insieme di istanze.

Configurazioni usate. Descriviamo ora una breve nomenclatura che utilizzeremo nelle prossime tabelle per indicare le configurazioni testate. Con il termine configurazione intendiamo una coppia composta da un modello e un solver che useremo per la ricerca della/e soluzioni. Nel caso dei modelli Minizinc, specifichiamo nella configurazione anche la strategia di ricerca.

[Asp] Modello Asp (descritto in sezione 2) + solver clingo.

[Mzn Cop] Modello Minizinc che usa vincoli non lineari e vincoli globali (sezione 3) + solver Gecode + strategia `solve maximize`.

[Mzn Ilp] Modello Minizinc costruito a partire dalla formulazione ILP (sezione 4.1) + solver Gecode + strategia `solve maximize`.

[Glpk] Modello ILP codificato in *CPLEX lp* (sezione 4.2) + solver Glpk.

5.1 Generazione delle istanze

Per testare le configurazioni abbiamo generato pseudo-casualmente (con un programma C) 30 istanze che sono descritte in tabella 1. In particolare le istanze 1-5 sono le istanze *EASY* e dovrebbero essere risolte agilmente da qualsiasi configurazione. Le istanze 26-30 sono le istanze *HARD* e sono create in modo che ogni configurazione non termini in meno di 300 secondi (exceeding timeout). Le istanze *MEDIUM* sono invece le istanze 6-26. Per controllare la difficoltà delle varie istanze abbiamo impostato i 3 seguenti parametri nella procedura di generazione pseudo-casuale:

[n] Il numero di gare presenti. Più ci sono gare più la difficoltà di risoluzione sale.

[q] La probabilità che per un giorno g in cui è stata schedulata una gara, esistano delle altre gare programmate per il giorno g. Se q fosse 0, ossia se non ci sono giorni duplicati, il vincolo [UNICITY] sarebbe sempre soddisfatto e quindi le istanze sarebbero più facili. Similmente se il q fosse 1, avremmo n gare tutte schedulate per lo stesso giorno e la soluzione ottima si potrebbe trovare banalmente con un algoritmo che lavora in tempo lineare. Quindi da una prima analisi risulta che le istanze difficili saranno quelle in cui il valore di q è vicino a 0.5 .

[$\frac{D}{U}$] Consideriamo il vincolo [BUDGET]. Sia B il budget di soldi a disposizione, sia S il doppio[3] della somma di tutte le distanze da casa delle gare generate, sia K il numero di chilometri che il veicolo percorre con un litro e sia C il costo della benzina. Se avessimo a disposizione un numero di soldi pari o maggiore a $\frac{SC}{K}$ allora le istanze sarebbero più semplici perché avremmo sempre abbastanza soldi per coprire il costo di qualsiasi distanza e il vincolo [BUDGET] sarebbe sempre soddisfatto. Possiamo anche fare un'affermazione più forte ossia possiamo dire che se U[4] è l'upper bound [UBDT], anche con un numero di soldi maggiore o uguale a $\frac{UC}{K}$ il vincolo [BUDGET] sarebbe sempre soddisfatto[5]. Il valore $\frac{D}{U}$ è la percentuale di chilometri rispetto a U che il budget ci permette di percorrere rispettando il vincolo [BUDGET]. Nella generazione delle istanze, calcoleremo il valore di U, imposteremo il valore $\frac{D}{U}$ e poi B verrà calcolato di conseguenza. Fissata una percentuale $\frac{D}{U}$ e fissato S, chiamando D il valore che realizza la percentuale, allora tutte le soluzioni avranno una somma delle distanze totali (andata e ritorno) minore a D. Se $\frac{D}{U}$ è 1 il vincolo [BUDGET] è sempre soddisfatto mentre se $\frac{D}{U}$ è 0, allora l'unica soluzione è non partecipare a nessuna gara. Anche in questo caso i valori centrali di $\frac{D}{U}$ renderanno l'istanza più difficile rispetto ai valori agli estremi.

[3] Dobbiamo considerare andata e ritorno.
[4] Notiamo che varrà $S = U$ solo nel caso in cui nell'istanza generata le gare siano tutte in giorni diversi.
[5] Vale $\frac{UC}{K} \leq \frac{SC}{K}$ siccome in generale $U \leq S$.

In tabella 1 vediamo le istanze generate: 5 EASY, 5 MEDIUM e 5 HARD. In tutte le istanze inoltre è stata posto il costo C della benzina pari a 183 centesimi. I valori S, U, D, q e $\frac{D}{U}$ non fanno parte dell'istanza, ma vengono riportati a scopo descrittivo dell'istanza e di come la stessa è stata generata.

Vale la pena notare che nelle istanze in cui q è alto (vicino a 0.5) allora il valore di U sarà in generale più basso rispetto a S. Se q è alto, ci sarà un alto numero di giorni duplicati e quindi riusciamo a fornire un miglior upper bound in quanto per ogni giorno g (in cui c'è almeno una gara) solo la distanza di una gara contribuirà ad U.

Istanza	n	B	S	U	D	$\frac{D}{U}$	q
ISTANZE EASY							
1	5	444	3376	3240	2916	0.9	0.1
2	10	1162	11148	9532	7625	0.8	0.2
3	15	775	14660	7268	5087	0.7	0.3
4	20	481	20394	5268	3160	0.6	0.4
5	25	638	29084	10466	4186	0.5	0.5
ISTANZE MEDIUM							
6	30	2519	28456	18358	16522	0.9	0.1
7	32	1713	38304	14046	11236	0.8	0.2
8	35	783	37466	7340	5138	0.7	0.3
9	40	615	46766	6730	4038	0.6	0.4
10	42	430	39572	7064	2825	0.4	0.5
11	45	2187	38344	15940	14346	0.9	0.1
12	50	1869	51172	15322	12257	0.8	0.2
13	52	3077	60148	28828	20179	0.7	0.3
14	55	1324	55876	14474	8684	0.6	0.4
15	60	840	57762	13786	5514	0.4	0.5
16	62	4060	69830	29586	26627	0.9	0.1
17	65	5058	74024	41462	33169	0.8	0.2
18	70	2674	76956	25056	17539	0.7	0.3
19	72	1808	70832	19768	11860	0.6	0.4
20	75	910	71270	14934	5973	0.4	0.5
21	80	5237	85290	38160	34344	0.9	0.1
22	82	4899	81162	40160	32128	0.8	0.2
23	85	3202	85498	30004	21002	0.7	0.3
24	90	2163	89474	23640	14184	0.6	0.4
25	95	894	90500	14660	5864	0.4	0.5
ISTANZE HARD							
26	150	13749	161106	100178	90160	0.9	0.1
27	250	12628	256048	103516	82812	0.8	0.2
28	350	10906	353782	102168	71517	0.7	0.3
29	450	11553	461990	126266	75759	0.6	0.4
30	550	6978	564942	114400	45760	0.5	0.5

Tabella 1: Descrizione di 30 istanze generate pseudo-casualmente

5.2 Esecuzione dei test

In tabella 2 riportiamo i valori generati dai test delle configurazioni su tutte le istanze. Quando non conosciamo il valore della colonna *Ottimo* per un'istanza, indichiamo un intervallo a cui l'ottimo appartiene; tale intervallo è della forma $[a, b]$ dove a è il miglior valore trovato da una configurazione e b è il valore [UBCT]. Nella colonna *Time* abbiamo utilizzato il simbolo `e` (exceeding timeout) quando la configurazione non è terminata sulla relativa istanza in un tempo di 300s; nel caso in cui in questo tempo non sia stata trovata nemmeno una soluzione che soddisfi i vincoli allora nella

colonna *Best* scriviamo il carattere `x`, altrimenti riportiamo il valore della migliore soluzione trovata. Se nella colonna *Time* leggiamo un numero allora il valore che troviamo nella colonna *Best* è l'ottimo del problema.

		Glpk		**Mzn Cop**		**Mzn Ilp**		**Asp**	
	Ottimo	**Best**	**Time (s)**	**Best**	**Time (s)**	**Best**	**Time (s)**	**Best**	**Time (s)**
ISTANZE EASY									
1	108	108	0.127	108	0.087	108	0.080	108	0.027
2	115	115	0.129	115	0.064	115	0.214	115	0.107
3	177	177	0.124	177	0.069	177	0.500	177	0.756
4	139	139	0.123	139	0.066	139	0.861	139	2.428
5	138	138	0.129	138	0.0748	138	1.330	138	7.741
ISTANZE MEDIUM									
6	327	327	0.142	327	1.113	327	1.839	327	e
7	230	230	0.133	230	0.073	230	2.279	230	79.460
8	148	148	0.135	148	0.084	148	2.610	148	16.429
9	179	179	0.137	179	0.102	179	3.366	179	36.220
10	98	98	0.137	98	0.0753	98	4.463	98	20.162
11	395	395	0.144	395	0.109	395	4.391	386	e
12	265	265	0.218	265	3.605	265	5.863	265	e
13	349	349	0.247	349	14.542	349	5.911	340	e
14	298	298	0.163	298	1.814	298	6.765	297	e
15	219	219	0.226	219	1.525	219	8.278	216	e
16	535	535	0.261	499	e	535	8.401	493	e
17	566	566	0.259	566	70.851	566	17.325	329	e
18	408	408	1.490	403	e	408	62.762	381	e
19	322	322	2.088	242	e	322	e	275	e
20	268	268	0.335	268	52.059	268	20.987	242	e
21	629	629	1.343	503	e	629	216.446	568	e
22	537	537	1.969	522	e	537	120.950	446	e
23	591	591	4.835	574	e	591	111.859	407	e
24	435	435	107.706	355	e	389	e	308	e
25	219	219	3.397	219	92.470	219	31.584	210	e
ISTANZE HARD									
26	937	928	e	525	e	793	e	x	e
27	[993, 2237]	993	e	142	e	894	e	x	e
28	[953, 2149]	953	e	124	e	794	e	x	e
29	[1055, 2963]	1055	e	70	e	x	e	x	e
30	[1113, 2889]	1113	e	86	e	x	e	x	e

Tabella 2: Performance delle configurazioni Glpk, Mzn Cop, Mzn Ilp e Asp sulle 30 istanze.

5.3 Discussione

In tabella 3 analizziamo i dati ottenuti. La configurazione Glpk si è rilevata la migliore, ottenendo solamente 5 exceeding timeout sulle istanze *HARD* (che ricordiamo essere progettate per causare exceeding timeout) e risolvendo in meno di 5 secondi tute le istanze 1-25 tranne la 24 che ha richiesto circa 100 secondi. La configurazione Mzn Ilp si è comportata molto bene: oltre ai 5 exceeding timeout sulle istanze *HARD*, si sono verificati solamente altri due casi nelle istanze 19 e 24. Notiamo che nel caso dell'istanza 19 il valore trovato corrisponde comunque al valore ottimo e quindi considerando le sole istanze *MEDIUM* che hanno generato `e`, la media dei rapporti $\frac{\text{valore trovato}}{\text{valore ottimo}}$ (da qui in avanti chiameremo questo valore *precisione*) è 0.947.

Contrariamente alle aspettative, la configurazione Mzn Cop si è comportata peggio di quella appena discussa. In questo caso ci sono ben 5 exceeding timeout in più e la precisione sulle istanze *MEDIUM* diminuisce a 0.890. Guardando le 5 istanze *HARD*, la configurazione Mzn Ilp riesce a trovare una soluzione solo per le prime tre, la configurazione Mzn Cop trova una soluzione per tutte e 5 ma i valori migliori trovati nelle istanze 26, 27 e 28 sono nettamente inferiori a quelli trovati da Mzn Ilp.

La configurazione Asp genera exceeding timeout su ben 21 configurazioni. Notiamo però che la precisione (0.888) è comunque abbastanza elevata. Molto spesso quando si verifica un exceeding timeout il valore migliore trovato è molto vicino al vero ottimo e il modello stabile che realizza tale valore viene trovato velocemente. Notiamo infine che per nessuna delle istanze HARD la configurazione Asp riesce a trovare un modello.

Concludiamo questa analisi sottolineando che, pur avendo visto due codifiche in Minizinc con performance molto diverse, non possiamo concludere immediatamente che una sia meglio dell'altra. Con Gecode, la codifica simil-ilp (configurazione Mzn Ilp) si è comportata meglio in generale della codifica con il vincolo globale, ma nelle ultime due istanze HARD (le istanze 29 e 30) non ha trovato una soluzione in 300 secondi. Il fatto che la configurazione Mzn Cop abbia comunque trovato delle soluzioni nelle istanze 29 e 30 può indicare che la propagazione del constraint `alldifferent_except_0` sia una procedura pesante che non vede i suoi benefici sulle istanze piccole, per la quale lavora meglio la codifica simil-ilp, ma al contrario nelle istanze più difficili mostra i suoi punti di forza. A titolo di esempio, il valore ottimo della prima istanza HARD (l'istanza 26) è stato trovato con la configurazione ⟨ modello Minizinc con vincolo globale, solver COIN-BC, strategia `solve maximize` ⟩ in meno di 200 secondi e invece la configurazione con la stessa strategia di ricerca, lo stesso solver e come modello il modello Minizinc simil-ilp ossia la configurazione ⟨ modello Minizinc simil-ilp, solver COIN-BC, strategia `solve maximize` ⟩ non è riuscita a trovare una soluzione nell'istanza 26; quindi in questo caso potremmo pensare che il solver *COIN-BC* lavori molto bene e sfrutti la potenza della propagazione dei vincoli globali. I modelli Minizinc simil-ilp e quello con il vincolo globale ammettono quindi esattamente le stesse soluzioni tuttavia per valutare quale dei due sia meglio occorrerebbe un'analisi più fine che valuti vari fattori tra cui la strategia di ricerca, i solver presi in considerazione e in particolare come vengono propagati i vincoli globali.

	Glpk	**Mzn Cop**	**Mzn Ilp**	**Asp**
Numero di exceeding timeout	5	12	7	21
Ottimi trovati in meno di 300 secondi	25	18	23	9
Ottimi trovati in exceeding timeout	0	0	1	2
Modelli non trovati	0	0	2	5
Precisione in exceeding timeout sulle istanze *MEDIUM*	/	0.890	0.947	0.888

Tabella 3: Analisi dei risultati delle configurazioni Glpk, Mzn Cop, Mzn Ilp e Asp sulle 30 istanze.

5.4 Euristiche di ricerca

In questa sezione prendiamo in considerazione le varie strategia di ricerca discusse in sezione 3.1. In particolare studieremo brevemente se, modificando la strategia di ricerca della configurazione Mzn Cop riusciamo ad ottenere risultati migliori per quanto riguarda le istanze HARD. Le istanze HARD

sono state tutte risolte dalla configurazione Mzn Cop, tuttavia il vlaore dell'ottimo trovato non è per nulla soddifacente e ci chiediamo se una strategia di ricerca ad-hoc possa aiutare a risolvere il problema. In questa veloce analisi prenderemo in considerazione solo le istanze 16-30 ossia i casi in cui la configurazione ha avuto le maggiori difficoltà. La scelta è motivata dal fatto che le strategie di ricerca vedono i loro risultati nelle istanze più difficili e non su quelle più facili, nelle quali, potrebbe addirittura verificarsi un peggioramento in termini di tempo.

Introduciamo due nuove configurazioni:

[Mzn Cop S1] Configurazione Mzn Cop con strategia S1 (sezione 3.1).

[Mzn Cop S2] Configurazione Mzn Cop con strategia S2 (sezione 3.1).

In tabella 4 riportiamo i risultati ottenuti. Vediamo che dove la configurazione Mzn Cop aveva riportato un exceeding timeout, le due nuove configurazioni riescono generalmente a trovare un valore migliore. Nelle istanze HARD notiamo un notevole miglioramento rispetto alla configurazione Mzn Cop, a titolo di esempio nell'istanza 30, entrambe le configurazioni riescono a trovare un valore di circa 10 volte superiore rispetto a quello trovato dalla configurazione senza strategia di ricerca. Per quanto riguarda le istanze MEDIUM, avevamo ottenuto una precisione (la media dei rapporti tra i valori trovati e i veri ottimi, sulle istanze MEDIUM che generano exceeding timeout) di 0.890. Con le configurazioni Mzn Cop S1 e Mzn Cop S2 otteniamo invece una precisione rispettivamente di 0.964 e 0.954 e quindi anche nelle istanze 16-25 il miglioramento è rilevante. Le strategie S1 e S2 presentate in sezione 3.1 si mostrano quindi pressochè equivalenti ed entrambe migliorano le performance sulle istanze più difficili.

		Mzn Cop		**Mzn Cop S1**		**Mzn Cop S2**	
	Ottimo	**Best**	**Time (s)**	**Best**	**Time (s)**	**Best**	**Time (s)**
ISTANZE MEDIUM							
16	535	499	e	535	e	530	e
17	566	566	70.851	562	e	516	e
18	408	403	e	405	e	401	e
19	322	242	e	322	180.406	313	e
20	268	268	52.059	268	e	253	e
21	629	503	e	583	e	598	e
22	537	522	e	537	e	512	e
23	591	574	e	508	e	556	e
24	435	355	e	415	e	389	e
25	219	219	92.470	210	e	219	e
ISTANZE HARD							
26	937	525	e	799	e	745	e
27	[993, 2237]	142	e	735	e	778	e
28	[953, 2149]	124	e	699	e	709	e
29	[1055, 2963]	70	e	822	e	795	e
30	[1113, 2889]	86	e	868	e	865	e

Tabella 4: Performance delle configurazioni Mzn Cop, Mzn Cop S1 e Mzn Cop S2 sulle istanze 16-30.