Relazione per il progetto di Ragionamento Automatico

143095 Zuccato Francesco

Sommario

Introduzione al problema

Nella presente relazione sono presentate due implementazioni diverse, in programmazione predicativa, per la ricerca di soluzioni di un problema NP-hard. Il sorgente è stato realizzato sia per Minizinc (uno dei più noti SAT-solver) sia in ASP (Answer Set Programming).

Il codice Minizinc è stato eseguito con il solver Gecode, 6.3.0.

Il codice ASP è stato eseguito con clingo 5.4.0 (grounder: gringo 5.4.0, solver: clasp 3.3.5).

Il timeout è stato impostato a 5 minuti.

L'istanza del problema in oggetto è la seguente:

14. La comunità europea finanzia i progetti PNRR a cui possono partecipare le sedi universitarie italiane. Per semplicità consideriamo quelle a Nord, a partire da Bologna (inclusa). Basandosi su google maps o simili, crea una matrice delle distanze in Km tra le Università suddette (Bologna, Modena, Parma, Ferrara, ..., Genova, ..., Udine, Trieste). Considera solo multipli di 10km approssimando per difetto (Udine Trieste sono 70 km ad esempio). Non è importante un realismo al 100%.

    Il PNRR consiste in 1 Milione di Euro complessivo per 4 tematiche. La suddivisione dei fondi per tematiche è un dato di input (esempio 40% per la prima, 30% per la seconda, 10% per la terza, 20% per la quarta). Come altro dato di input ogni università considerata ha un elenco non vuoto di tematiche di interesse. Diciamo da una a tre ciascuna. Per ogni tematica ci deve essere una sede proponente (hub) e al massimo 5 altre sedi (spoke) a ciascuna delle quali si possono associare altre università distanti meno di 100km. Gli spoke di una stessa tematica invece devono essere distanti almeno 100km. Un hub di una tematica non può essere nè hub nè spoke di altre tematiche. Uno spoke può essere spoke di al massimo 2 tematiche. Ogni sede può entrare solo nelle tematiche di interesse dichiarate.

    L'hub ha il 30% dei fondi complessivi della tematica. Il rimanente 70% è diviso equamente tra tutti gli spoke e affliati.

    Si vuole creare una struttura di hub (uno per tematica), spoke e associate che garantisca almeno m (dato in input, poniamo 50.000 Euro ad esempio) a ogni sede e tale che nessuna sede incameri più di M (anche questo in input, poniamo 150.000 Euro ad esempio).

Analisi dei requisiti

Dal problema si evidenziano molte entità che possono essere costanti (valore fissato, uguale per ogni istanza), parametri (valore variabile, ma fissato per ogni istanza) o variabili (valore variabile in esecuzione):

1. Il numero di tematiche (denominate 'sections' nel codice)
    - Costante, intero: s=4
2. Il numero di università
    - Parametro, intero: s ≤ n ≤ 20
3. La matrice delle distanze
    - Costante, matrice 20*20 di interi: *distance*
    - È stata generata in Python dalle coordinate delle città, per massimizzare la realtà
    - NB: Se n < 20 si usano le prime n università e quindi solo la sottomatrice n*n
4. Il denaro totale da suddividere
    - Parametro, intero: *tot_money* = 10^i, con 3 ≤ i ≤ 6
5. La distanza minima tra spokes
    - Costante, intero: d < 100 km
6. La distanza massima tra affiliate e spokes/hub
    - Costante, intero: d ≥ 100 km
7. Il numero massimo di hub e spokes per tematica
    - Costanti, interi: *max_num_hubs* = 1, *max_num_spokes* = 5
8. L'insieme delle percentuali di ripartizione del denaro:
    - Tra tematiche: Parametro, array di s interi: *sections_perc* = [40, 30, 10, 20] o [33, 27, 25, 15]
    - Tra sedi hub e non_hub, Costanti, interi: *hubs_perc* = 70*, non_hubs_perc* = 30.
    - NB: sebbene i valori siano delle percentuali, sono memorizzati come interi per:
        - Generalità: Non tutti i solver sanno gestire i float (es. Chuffed)
        - Problemi di memoria: con un budget di € 1mln, anche solo usando interi e divisione intera, ci sono problemi di overflow, meglio non usare i float in ogni caso
9. Il numero minimo e massimo di interessi per ogni università
    - Parametro: coppia di interi *(min_interests, max_interests)* = (1,3) o (2,4)
    - In realtà sono dei parametri per il codice Python che genera random la matrice degli interessi
10. La matrice degli interessi delle università
    - Parametro, matrice 20*4 di booleani: *interested*
    - Ne sono state due generate in Python, usando i parametri (1,3) e (2,4)
11. La quota di denaro ricevuta da ogni università
    - Variabile, array di n interi: *money_gained* ≥ 0
    - È una variabile ma in realtà il suo valore è calcolato direttamente dalle altre variabili:
    - $\text{money\_gained}[\text{i}] = \sum_{\text{j}=1}^{\text{s}}(\text{is\_hub}(\text{i},\text{j}) * \text{money\_hubs} + \text{is\_non\_hub}(\text{i},\text{j}) * \text{money\_non\_hubs})$
        - NB se università i non è affiliata/spoke/hub in j allora is_hub(i,j) = is_non_hub(i,j) = 0
12. La quota di denaro minima per ogni università, che è la **variabile principe del problema**
    - Variabile, intero: 0 ≤ *min_money* ≤ (*tot_money / n)*
    - Un obiettivo secondario del problema è l'equità, ovvero massimizzare parametro *min_money:*
        - CSP: iniziare con *min_money* piccolo e aumentarlo finchè non si trova più soluzione
        - COP: max{ min{ *money_gained* } } (per ottimizzare si può inizializzare *min_money* con un valore non piccolissimo, per cercare di escludere dall'inizio alcune soluzioni)
13. La quota di denaro massima per ogni università
    - Costante, intero: *max_money* = (3/10) * *tot_money*
    - Teoricamente ci si aspetta sia un parametro, in realtà di fatto è stato usato come una costante:
        - per riuscire garantire la quota minima a tutti, è difficile ottenere un'alta quota massima
        - il 70% sarà sempre per gli hub, quindi il massimo sarà sempre abbastanza grande

Considerazioni:

(1) Lower-bound: max_money ≥ tot_money * max{ sections_perc } * hub_perc

- Ciò implica, nella prima serie di test, usando *sections_perc* = [40, 30, 10, 20] si ha:
  - max_money ≥ tot_money * 40% * 70% = tot_money * 28%
  - Indipendentemente da n, più del 25% del denaro andrà solo ad un'unica università!
  - Per questo motivo si è deciso di fare un'altra serie di test abbassando max{ *sections_perc* }
- Nella seconda serie di test, usando *sections_perc* = [33, 27, 25, 15] si ha:
  - max_money ≥ tot_money * 33% * 70% = tot_money * 23,1%
  - Che è un buon miglioramento, avendo ridotto del 5% il lower bound precedente, e che implica un aumento del denaro disponibile da dividere, ovvero più equità.

(2) Riduzione delle percentuali di ripartizione:

- Tra tematiche: al massimo si potrebbero usare tutte 4 le tematiche al 25% (in generale a 1/s), ma il problema diventerebbe più semplice, meno generico e meno realistico
  - NB: in ogni caso comunque il lower bound è 25% * 70% = 17,5%.
- Tra hub e non: si potrebbe cambiare ad esempio con un 60%-40% la distribuzione tra hub e non, ma dal testo si evince che 70%-30% è una costante del problema.

Preprocessing e generazione tabelle

Scelta delle università

La prima cosa da fare è scegliere con quali università realizzare trattare. Per semplicità e s.p.d.g. è stato scelto di utilizzare solo un'università per ogni città. Inoltre, sempre per semplicità, le coordinate utilizzate per calcolare le distanze sono delle città in generale, non specifiche delle università.

Sono state identificate venti città universitarie nel Nord Italia. Le relative coordinate sono state estrapolate da un progetto su GitHub che, nel file *italy_geo.xlsx*, contiene le coordinate di tutti i comuni italiani al 2018.

Scelte le università in oggetto, si può procedere alla generazione delle due tabelle.

Tabella delle distanze

File: distances.py

Per generare la tabella si inizia con l'importare il file csv delle coordinate delle città. Poi si è calcolato la reale distanza geografica tra ogni coppia di città (la formula è omessa) e la si è memorizzata in matrice 20x20, arrotondando i valori ad ogni 10km.

Infine, è stato sufficiente esportare tale matrice sia in un csv (su cui poi, con delle banali regex, si ottiene una matrice secondo la sintassi di mzn) sia in un file lp, stampando per ogni coppia (i,j), con i < j, '*distance(i,j,km).*', aggiungendo anche il vincolo di simmetria.

Tabella degli interessi

File: interested.py

A differenza di prima non esiste un'unica tabella giusta, dato che gli interessi sono generati randomicamente. Si è scelto di generare due diverse tabelle, una dove ogni università ha da 1 a 3 interessi, l'altra da 2 a 4. Inoltre, si genera un seed random per garantire la riproducibilità.

L'implementazione è piuttosto semplice: per ogni università si genera un valore intero $interests$ tra [min, max], che sarà il numero totale di interessi per l'università corrente, e poi si generano interi tra [1,s] fino a che non se ne ottengono esattamente $interests$ distinti.

Terminata la generazione, il resto è una stampa degli output secondo le sintassi di mzn ed lp nei due file.

Una volta generate le due tabelle, per semplicità, esse sono semplicemente copiate ed incollate nei file progetto.lp e progetto.mzn anche se sicuramente sarà possibile compilare ed eseguire più file lp/mzn assieme.

Le due tabelle si assumono quindi presenti e saranno omesse da ulteriori considerazioni nelle pagine seguenti.

Programma ASP

File: progetto.lp

Per scrivere ASP si è iniziato dalle costanti e dai parametri, ovvero definendo le relazioni più banali: il budget totale, l'incasso minimo e massimo per ogni università, i domini per le università e le tematiche, le percentuali di condivisione tra le tematiche e tra hub e spoke/affiliati.

Prima di proseguire si introduce qualche definizione e si specifica la seguente premessa: per brevità, si lasceranno impliciti i vincoli di dominio $sections\_dom(S)$, $universities\_dom(U)$ e simili.

Definizione 1: due università U1, U2 si definiscono 'vicine' se è vera la seguente relazione:

$$in\_range(U1, U2) :- distance(U1, U2, KM), KM < 100$$

Definizione 2: università U 'partecipa'' alla tematica S se è vera una delle seguenti relazioni:

$$hub(S, U), spoke(S, U), affiliated(S, U).$$

Definizione 3: università U è 'libera' per S se U non partecipa ad S.

Dopo queste le precedenti definizioni si può spiegare come si sono dichiarate le tre relazioni più importanti: $hub(S, U), spokes(S, U), affiliated(S, U)$, dove S sta per section (tematica) e U per università. Assieme alla dichiarazione si aggiungono nella stessa riga alcuni vincoli: (a) di numerosità: (1,1) per gli hub, (0,5) per gli spoke e (0,n) per le affiliate (nessun vincolo), (b) di unicità nella partecipazione, (c) di interessamento dell'università a quella tematica.

Ora si può iniziare con l'aggiunta dei vincoli, alcuni sono piuttosto semplici:

- si nega che un'università U, che è già hub per la tematica S1, sia hub o spoke anche per S2:

$$:-hub(S1, U), hub(S2, U), S1 \neq S2. \text{ e } :-hub(S1, U), spoke(S2, U), S1 \neq S2$$

- si nega la distanza < 100 tra due spoke: $:- spoke(S, U1), spoke(S, U2), in\_range(U1, U2), U1 \neq U2$

Il prossimo vincolo sulla distanza è più interessante ed è espresso dalla seguente implicazione: $affiliated(S, U1) \rightarrow \exists U2 : ((spoke(S, U2) \vee hub(S, U2)) \wedge in\_range(U1, U2))$. Si noti che non è un vincolo universale, ovvero possono esistere coppie di università distanti dove una è affiliata e l'altra è spoke o hub per la stessa tematica. Ciò significa che sarebbe sbagliato usare lo stesso approccio di negazione usato per le distanze tra spoke, perché, se per una tematica S esistessero due spoke a distanza almeno 210 allora per S l'insieme delle affiliate sarebbe sempre vuoto. Anche una strategia opposta, per il motivo duale, sarebbe incorretta: con l'implicazione: [(se esiste spoke o hub U1 per S, U1 'vicina' U) allora (U è affiliata ad S)] si avrebbero tutte le università libere ed interessate ad S affiliate ad S, ma il punto non è vero, questa è una possibilità, non un obbligo.

L'approccio usato si basa sulla creazione di una relazione ausiliaria $can\_be\_affiliated(S, U)$, che è valida per (U,S) se esiste U1, spoke o hub per S, vicina ad U. (idem con spoke al posto di hub):

$$can\_be\_affiliated(S, U) :- hub(S, U1), in\_range(U, U1)$$

Ora non resta che imporre che la formula: $affiliated(S, U) \rightarrow can\_be\_affiliated(S, U)$, che si trascrive:

$$:- not\ can\_be\_affiliated(S, U), affiliated(S, U)$$

Fatto ciò, L'ASP trova tutte e sole le soluzioni che rispettano i vincoli di partecipazione indicati. Ora è necessario includere i vincoli monetari per cercare di massimizzare l'equità e per non lasciare nessuna università libera, che al momento è ancora contemplato.

In realtà, usando le funzioni aggregate $\#sum$ e $\#count$ tutti i passaggi sono piuttosto banali, anche se necessitano di un paio di relazioni diverse, per garantire anche un minimo di leggibilità ed evitare di riscrivere pezzi di codice uguali in più parti. Inoltre, per brevità, di seguito si indicherà con S&A l'insieme di spoke ed affiliate ad una certa tematica (implicita). Si può quindi definire:

- il denaro spettante a ciascuna tematica:

$$money\_for\_section(S, T * P/100) :- section\_perc(S, P), total\_money(T)$$

- il denaro spettante all'hub della tematica (similmente anche $money\_for\_non\_hubs$ per S&A):

$$money\_for\_hubs(S, (TS * P/100)) :- money\_per\_section(S, TS), hubs\_perc(P).$$

- il numero di S&A per ogni tematica S, ottenuto contando gli spoke e gli affiliati di S:

$$num\_of\_share\_owners(S, SP + AF) :- SP = \#count\{U : spoke(S, U)\}, AF = \#count\{U : affiliated(S, U)\}$$

Ora è necessario aggiungere un vincolo molto importante sulle istanze piccole. Bisogna negare che il numero di S&A per ogni tematica S sia maggiore di zero. Dai vincoli precedenti è ammissibile, sia spokes sia affiliate non hanno un numero minimo per tematica, ma implicherebbe la possibilità di "sprecare" denaro dato che nessuno ne avrebbe diritto. Tale situazione la si nega banalmente con: $:- num\_of\_share\_owners(S, 0)$.

Si può proseguire con i passaggi, calcolando la singola quota spettante ad ogni S&A per ogni tematica:

$$share\_per\_section(S, (MON/OWN)) :- money\_for\_non\_hubs(MON), num\_of\_share\_owners(S, OWN)$$

Non resta che sommare, per ogni università, le varie quote a cui ha diritto in quanto hub, spoke o affiliata. Il modo più semplice è di agglomerare tutto in una relazione "pozzo" su cui applicarci poi sum. Si noti che le relazioni rappresentate nell'ASP non sono altro che sottoinsiemi di $\mathbb{N}^k$ (dove k è l'arità). Questo per sottolineare che sono insiemi e non multi-insiemi, quindi, se si desidera agglomerare più informazioni assieme è necessario essere sicuri di produrre solo valori univoci, altrimenti si perderebbe informazione. Siccome è possibile che un'università riceva la stessa quota da più tematiche diverse, nella relazione è necessario conservare la tematica di provenienza, a differenza del ruolo, dato che per ogni coppia (U,S) esso è univoco.

Il predicato è così definito (la prima riga va riscritta anche cambiando $spoke(S, U)$ con $affiliated(S, U)$):

$$gains\_share(S, U, SHARE) :- share\_per\_section(S, SHARE), spoke(S, U)$$

$$gains\_share(S, U, SHARE) :- money\_for\_hubs(S, SHARE), hub(S, U)$$

Per calcolare il denaro incassato è sufficiente applicare la funzione aggregata:

$$money\_obtained(U, MONEY) :- MONEY = \#sum\{SHARE : gains\_share(S, U, SHARE)\}$$

Infine, come ultima cosa i vincoli sul denaro incassato da ogni università:

$$:- money\_obtained(U, M), min\_money(MIN), MONEY < MIN.$$
$$:- money\_obtained(U, M), max\_money(MAX), MONEY > MAX.$$

Programma Minizinc

File: progetto.mzn

Per il programma in mzn l'approccio risolutivo non si discosta molto dall'ASP. Una differenza è che le tre relazioni affiliated, spoke e hub sono codificate in un'unica matrice $role$ n*s.

Ogni elemento di $role[i,j]$ (università i, tematica j) è così codificato: 0 se non partecipante ("nothing"), 1 se affiliata, 2 se spoke, 3 se hub. Questa codifica, oltre che compatta, definisce implicitamente un ordine numerico ai ruoli, come effettivamente è nella realtà. Questo sia consente di compattare alcune parti di codice (ad es. se una proprietà deve valere per spoke e hub basterà dire $role[i,j] \geq spoke$), sia potrebbe aiutare nel solving.

Le relazioni costanti n, s, min, max e tot money ecc sono $par\ int$. Similmente, sections_perc ora è un $array\ of\ par\ int$. Tutte le relazioni utilizzate per la divisione del budget sono state trasformate in $array\ of\ var\ int$: $num\_of\_share\_owners,\ money\_for\_non\_hubs,\ money\_for\_hubs,\ money\_obtained$. Inoltre, per un pretty-print dell'output, si è aggiunto un array di stringhe con i nomi delle città.

Premessa: esclusi i primi vincoli esemplificativi, per brevità, i domini delle variabili nei forall potrebbero essere omessi. In tal caso vale la seguente regola generale: $i\ in\ 1..n$ (o i1, i2 ecc) è usata per indicare le università, $j\ in\ 1..s$ (o j1, j2 ecc) è usata per indicare le tematiche.

Alcuni dei vincoli di partecipazione si possono realizzare con un paio di semplici constraint forall:

- ogni università può partecipare solo alle tematiche a cui è interessata (o è interessata o non partecipa)

$$forall(i\ in\ 1..n, j\ in\ 1..s)\ \{\ (role[i,j] ==\ nothing) \vee\ interested[i,j]\ \};$$

- i vincoli di numerosità per hub (1,1) e section (0,5) sono banali con un count:

$$forall(j\ in\ 1..s)\ \{\ count(role[1..n,j], hub) == 1\ \};\ \ forall(j\ in\ 1..s)\ \{\ count(role[1..n,j], spoke) \leq 5\ \};$$

- con il count si può negare anche lo spreco di denaro (nessuna affiliata o spoke per una tematica):

$$forall(j\ in\ 1..s)\ \{\ count(role[1..n,j], affiliated) +\ count(role[1..n,j], spoke) > 0\ \}$$

- l'unico altro ruolo che può avere un hub è essere affiliato:

$$forall(i,j1,j2\ where\ ((role[i,j1] == hub) \wedge (j1 \neq j2))\ )\ \{\ role[i,j2] \leq affiliated)\ \};$$

- per limitare i ruoli degli spoke si può usare la stessa strategia usata nell'LP: se la cardinalità massima di spoke è n se ne definiscono n+1 diversi e si nega tale possibilità:

$$forall(i,j1,j2,j3\ where\ (\ j1 \neq j2\ \wedge\ j2 \neq j3\ \wedge\ j1 \neq j3)\ )$$

$$\{\ not\ (\ (role[i1,j] \geq spoke) \wedge (role[i2,j] \geq spoke) \wedge (role[i3,j] \geq spoke)\ )\ \};$$

  - NB: pe questo vincolo erano sufficiente usare il $\neq$ anziché il $\geq$, ma si è convenuto usare quest'ultimo per (possibili) motivi di efficienza dato che taglia anche alcune possibilità con gli hub

- per negare che due spoke possano essere 'vicini' è semplice, e la struttura del vincolo ricalca i precedenti:

$$forall(i1,i2,j\ where\ ((role[i1,j] == spoke) \wedge (role[i2,j] == spoke) \wedge (i1 \neq i2)))\ \{\ distance[i1,i2] \geq 100\ \};$$

- per obbligare invece un'affiliata ad avere uno spoke/hub vicino è un po' più complesso. Il *core* del problema è lo stesso enunciato affrontando lo stesso problema nell'ASP: è una condizione esistenziale. Fortunatamente, in Minizinc è presente la primitiva *exists()*, che permette di esprimere facilmente il vincolo:

$$forall(i,j\ where\ (role[i,j] == affiliated))\ \{\ exists(i1\ where\ role[i1,j] \geq spoke)\ \{\ distance[i,i1] < 100\ \}\};$$

Ora, anche per il mzn sono stati definiti tutti i vincoli di partecipazione, e, come nel caso dell'LP, non resta che implementare il codice che spartisca il denaro secondo le regole fornite.

Nei prossimi vincoli si ometterà di specificare sempre forall ma è sempre presente, e continua a valere la regola specificata prima sull'uso degli indici (i per università, j per tematiche).

Rispetto all'ASP, due predicati sono stati rimossi: $money\_for\_section$ e $money\_for\_non\_hubs$, servivano solo per memorizzare dei passaggi intermedi che nel codice mzn sono stati inglobati dalla funzione, così definita:

$$calc\_money\_per\_section(tot\_money, num\_of\_share\_owners, sections\_perc, participation\_perc) =$$

$$((tot\_money\ * section\_perc * participation\_perc)\ div\ (100 * 100 * num\_of\_share\_owners));$$

Prima di poter utilizzare la funzione è necessario calcolare il numero di S&A per ogni tematica:

$$forall(j)\ \{\ num\_of\_share\_owners == \ count(role[1..n, j], spoke) + count(role[1..n, j], affiliated)\ \};$$

Ora si ha a disposizione tutti i dati necessari per calcolare quanto distribuirà ogni tematica:

$$money\_for\_hubs[j] == calc\_money\_per\_section(tot\_money, 1, sections\_perc[j], hubs\_perc)$$

$$share[j] == calc\_money\_per\_section(tot\_money, num\_of\_share\_owners[j], sections\_perc[j], non\_hubs\_perc)$$

Non resta che eseguire tre somma-se: se l'università i ha un ruolo per tematica j: si aggiunge il dovuto al totale. Per implementarla si è utilizzata un'altra funzione ausiliare $indexes\_of(val, array)$, che ritorna un array della stessa lunghezza di quello in input, dove, per ogni elemento restituisce i se $val = array[i]$ else -1.

Per ottenere a quanto ammonta il denaro ricevuto da ogni università non resta che usare tre sum (e sommarli). Ognuno di essi somma tutti gli elementi dell'array t.c. $indexes\_of()$ non ha ritornato -1, ovvero solo quelli per cui l'università partecipa (per S&A bisogna sostituire $money\_for\_hubs$ con $share$, e, al posto di hub, una volta $spoke$ e l'altra $affiliated$):

$$money\_gained[i] == sum([if\ j \geq 0\ then\ money\_for\_hubs[j]\ else\ 0\ endif\ |\ j\ in\ indexes\_of(hub, role[i, 1..s])$$

Infine, bisogna imporre che il denaro ricevuto da ogni università rispetti i limiti dati:

$$(calc\_money\_gained[i] \geq min\_money) \wedge (calc\_money\_gained[i] \leq max\_money)$$

Ottimizzazioni

È noto che l'aggiunta di vincoli semplici da calcolare, anche se già implicati da altri, può impattare positivamente sui tempi di esecuzione. Uno è già stato specificato nel caso del codice MiniZinc, dove nel vincolo sul numero di spokes diversi si sono inclusi gli hub con la disuguaglianza, anche se non necessario.

Ma, oltre a questo dettaglio, si è identificato un predicato molto utile per diminuire i tempi di esecuzione. Siccome è richiesto che ogni università riceva almeno min_money, ciò banalmente implica anche che ogni università deve partecipare ad almeno una tematica, perché altrimenti il suo ricavo sarebbe zero.

Con i vincoli attuali ci si accorge che un'università incassa zero solo quando si verifica che $0 < min\_money$, ergo la soluzione trovata non va bene. Il punto è che per fare questo controllo è necessario aver già calcolato quanto ricava l'università in questione (e, molto probabilmente, il solver avrà già fatto fatto il medesimo calcolo per tutte le altre università). Siccome sono necessari un bel po' di passaggi: chiamare la funzione $calc\_money\_per\_section$ (s volte), chiamare $indexes\_of$ e fare i sum (3*n volte), sommarli ed infine verificare il range di $money\_gained$ (n volte, se a fallire è l'ultima).

Tutti questi calcoli si rilevano inutili a monte se un'università non partecipa a nessuna tematica. Tale proprietà è talmente banale da verificare che è immediata a vista d'occhio, è sufficiente che ci sia una riga di soli zeri.

Nel minizinc è sufficiente scrivere:

$$forall(i\ in\ 1..n)\ \{\ sum[i, 1..s] > 0\ \};$$

Un discorso simile si può fare anche con ASP. Per implementarlo definiamo il predicato ausiliario $participate(U)$ e lo implichiamo a vero se esiste un S t.c. U partecipa ad S. Infine neghiamo la sua negazione:

$$participate(U) :- affiliated(S, U), sections\_dom(S), universities\_dom(U).$$

$$participate(U) :- spoke(S, U), sections\_dom(S), universities\_dom(U).$$

$$participate(U) :- hub(S, U), sections\_dom(S), universities\_dom(U).$$

$$:- not\ participate(U), universities\_dom(U).$$

A dimostrazione dell'utilità del vincolo si mostrano a titolo di esempio quattro tempi di esecuzione diversi (MZN/ASP, con/senza), usando i seguenti parametri:

$$total\_money = 1{,}000, min\_money = 15, max\_money = 300, s = 4, interested\_seed = 0.9100653423283941$$

	MiniZinc (solve satisfy)	ASP (tempo di grounding escluso)
con vincolo	SAT in 1.5 s	SAT in 2.0 s
senza vincolo	UNKOWN in 300.0 s	SAT in 57.7 s

Certamente è solo un'euristica ed un solo test non è certamente un campione statistico affidabile, ma sembra alquanto difficile poter affermare che il vincolo non possa essere utile. In tutti i tempi di esecuzione dei test elencati al prossimo capitolo tale vincolo aggiuntivo è sempre incluso.

Inoltre, una condizione necessaria (ma non sufficiente) per la consistenza del sistema, è sicuramente che:

$$min\_money * n \leq tot\_money$$

Sembra una condizione banale da verificare, ma se non si aggiunge tale vincolo i solver non si rendono conto dell'inconsistenza, mentre, con il vincolo, tutti verificano subito (1-2 secondi) un modello inconsistente!

Usando l'istanza di prima ($se\ min > 50 \rightarrow UNSAT$) : ASP va in timeout con $min = 60$.

MiniZinc fa (molto) peggio… timeout: (1) default: con $min = 150$ (2) first_fail, indomain_random e Luby(63): con $min = 130$, (3) first_fail, indomain_random e geometric(1.5,100) con $min = 140$.

Problemi dovuti agli interi grandi

Nella consegna è richiesto $total\_money = 1\ mln$. Purtroppo, però con i numeri grandi come quelli richiesti ci sono stati non pochi problemi.

ASP

Nell'ASP i problemi sono dati dalla fase di grounding iniziale. Usando però 1mln il timeout di 5 minuti scade senza che nemmeno sia terminato il grouding. E lo stesso vale anche per 100,000 e 10,000. Quindi ci si è dovuti accontentare di usare $total\_money = 1{,}000$ che necessita di circa 6 secondi per il grounding.

MINIZINC

Anche per il MiniZinc sono sorti dei problemi. In questo caso però sono dovuti semplicemente ad overflow. Dato che si sono usati gli interi per rappresentare le percentuali per calcolare ad esempio il 70% è necessario fare $total\_money * 70\ div\ 100$. Il problema è che il primo prodotto fa crescere ancor di più il numero iniziale e, in Gecode 6.3.0. causa un errore di overflow.

Per ovviarlo si sono invertiti i fattori: $total\_money\ div\ 100 * 70$. Ed infatti l'overflow non accade, tant'è che per $total\_money \geq 500{,}000$ si usa questa formula invece della precedente. L'ultima strategia - che sembra un'ottima soluzione per ovviare l'overflow - in realtà, fa sorgere un problema ancora maggiore.

Il punto è che div è l'operatore di divisione intera, e tronca un eventuale resto. Il che significa che nel caso in cui la divisione abbia resto i decimali andrebbero persi. Moltiplicando, si ottengono dei valori che sono influenzati e si propaga l'errore. Errore che invece non si propagherebbe moltiplicando prima e dividendo dopo.

Tutto questo per dire che è altamente sconsigliato di usare $total\_money \geq 500{,}000$ in quanto potrebbe dare come risultato UNSAT quando invece il problema è SAT (per moltissimi problemi basta scalare con $total\_money = 100{,}000\ o\ 10{,}000$ senza perdere troppo soluzioni).

Un esempio ($s = 4, seed = 0.9100653423283941$):

- MiniZinc: UNSAT in 39 s: $total\_money = 1{,}000{,}000, min\_money = 74{,}000, max\_money = 300{,}000$
- MiniZinc: SAT in 2 s: $total\_money = 100{,}000, min\_money = 7{,}400, max\_money = 30{,}000,$

In realtà ovviamente se esiste una soluzione con 100,000 deve esistere anche con 1,000,000: si mantengono gli stessi ruoli e il denaro che riceverà ogni università basterà moltiplicarlo x10.

Per questi motivi non si è usato 1mln. Per consentire un paragone più accurato possibile si è deciso si eseguire anche il MZN con $total\_money = 1{,}000$ come ASP. Tuttavia, si noti che, shiftando di 10 il tot_money, se il min_money non è un multiplo di 10 è necessario troncare il valore per difetto o per eccesso.

Un esempio ($s = 4, seed = 0.9100653423283941$):

- MiniZinc: SAT in 33 s: $total\_money = 1{,}000, min\_money = 17, max\_money = 300$
- MiniZinc: SAT in 8 s: $total\_money = 10{,}000, min\_money = 173, max\_money = 300$
- MiniZinc: UNKOWN in 300 s: $total\_money = 1{,}000, min\_money = 18, max\_money = 300$

Con 1,000 non si possono ovviamente trovare soluzioni con un minimo pari solo a 17 o 18, mentre con 10,000 è possibile usare anche 171,172,..,179.

Test e tempi di esecuzione

Premesse

Di seguito sono presentati i tempi esecuzioni di un CSP in MZN e ASP con due istanze diverse: una su un grafo degli interessi più sparso (il primo) e uno più denso (il secondo). Inoltre, sempre su quest'ultimo, sono presentati anche dei test di COP con l'obiettivo di massimizzare min_money.

Per il MiniZinc sono presentate le esecuzioni con tre tecniche diverse: (1) default, oppure con first_fail, indomain_random e (2) luby(63) o (3) geometric(1.5, 100). Sono state provate anche altre possibilità di restart, varchoice o constrain choice, ma si sono scelte queste in quanto molte volte si sono rilevati le migliori

Ovviamente per ogni problema esiste un valore k per min_money t.c. tutti i valori minori di esso sono SAT (potrebbe essere anche k=0, se ad esempio la matrice degli interessi è vuota) mentre i maggiori uguali sono UNSAT. Su istanze con n=20 spesso tale valore non si riesce a calcolare con precisione in quanto tutti i valori nel suo intorno portano a timeout.

Un'ultima osservazione: se tot_money=k trova una soluzione allora è soluzione anche per tot_money=k*mul, dove il ricavato di ogni università va moltiplicato per mul. Si noti che il viceversa non è valido.

CSP-1: Grafo sparso

Matrice degli interessi generata: *python3 interested.py 4 1 3 0.3355233901788055*

$n = 20, s = 4, max\_money = 300, total\_money = 1{,}000$

	Output	Time (s)	Min Money	Restart	Var Choice	Constrain Choice
MZN	SAT	1	16	default	default	default
MZN	SAT	1	16	luby(63)	first_fail	indomain_random
MZN	SAT	1	16	geometric(1.5, 100)	first_fail	indomain_random
ASP	SAT	9	16			
MZN	?	300	17	default	default	default
MZN	?	300	17	luby(63)	first_fail	indomain_random
MZN	?	300	17	geometric(1.5, 100)	first_fail	indomain_random
ASP	UNSAT	42	17			
MZN	UNSAT	11	20	default	default	default
MZN	?	300	20	luby(63)	first_fail	indomain_random
MZN	?	300	20	geometric(1.5, 100)	first_fail	indomain_random
ASP	UNSAT	38	20			

Percentuali per tematica: [40%, 30%, 10%, 20%]

In questa istanza il grafo delle distanze è sparso: ogni università è interessata da 1 a 3 tematiche su 4.

Un grafo come così formato riduce le combinazioni generabili che sono soluzione (ad es. la terza è interessata solo alla quarta tematica essa deve sicuramente deve parteciparvi).

Nell'esempio si nota come il problema con min_money=16 sia banalmente risolvibile, mentre con 17 tutte e tre le esecuzioni di MiniZinc portano a timeout. ASP invece, in un tempo anche relativamente breve, riesce a concludere che il problema è UNSAT.

Si è provato infine ad aumentare min_money=20, avendo già noto che tale istanza sia UNSAT, per vedere se MZN ora termina. Un po' a sorpresa, l'unico a riuscirci (anche molto velocemente!) è il solve satisfy di default.

CSP-2: Grafo denso

Matrice degli interessi generata: *python3 interested.py 4 2 4 0.9100653423283941*

$n = 20, s = 4, max\_money = 30\% * total\_money$, $total\_money = 1{,}000$

	Output	Time (s)	Min Money	Restart	Var Choice	Constrain Choice
MZN	SAT	5	16	default	default	default
MZN	SAT	5	16	luby(63)	first_fail	indomain_random
MZN	SAT	5	16	geometric(1.5, 100)	first_fail	indomain_random
ASP	SAT	86	16			
MZN	?	300	17	default	default	default
MZN	SAT	35	17	luby(63)	first_fail	indomain_random
MZN	?	300	17	geometric(1.5, 100)	first_fail	indomain_random
ASP	SAT	250	17			
Cambiando $total\_money = 10{,}000$:						
MZN	?	300	173	default	default	default
MZN	SAT	47	173	luby(63)	first_fail	indomain_random
MZN	SAT	9	173	geometric(1.5, 100)	first_fail	indomain_random
MZN	SAT	193	174	luby(63)	first_fail	indomain_random
MZN	?	300	174	geometric(1.5, 100)	first_fail	indomain_random
MZN	?	300	175	luby(63)	first_fail	indomain_random

Percentuali per tematica: [33%, 27%, 25%, 15%]

In questa istanza il grafo delle distanze è denso: ogni università è interessata da 2 a 4 tematiche su 4.

Anche in questo caso abbiamo che con $min\_money = 16$ si trova una soluzione facilmente con MiniZinc, mentre ASP fa un po' di fatica impiegando più di un 1 minuto e 20 secondi.

Aumentando il minimo a 17 l'insieme delle soluzioni diminuisce, e diventa difficile anche per MiniZinc trovare una soluzione. Il restart con $luby(63)$ è l'unico tra i MZN a trovare una soluzione, probabilmente grazie ad un restart al momento giusto nel posto giusto. L'ASP, sebbene molto più lento dei vari MZN già con il minimo precedente, anche in questo caso riesce a trovare una soluzione, quasi allo scadere dei 5 minuti.

Aumentando ancora il minimo a 18 tutti, sia ASP sia MZN, vanno in timeout (risultati omessi dalla tabella).

Siccome la consegna iniziale richiedeva un tot_money=1,000,000, si è aumentato da 1,000 a 10,000, per aumentare la granularità delle soluzioni. In ASP, come già spiegato, non è possibile (il grounding va in timeout).

Ciò che emerge è che anche in questo caso il migliore è $luby(63)$, l'unico a riuscire a trovare una soluzione con $min\_money = 174$. Diminuendo di 1 a 173, anche il $geometric(1.5, 100)$ trova una soluzione.

Fanalino di coda il solve satisfy di default che va in timeout già con 170 e 171.

COP: Grafo denso

Rispetto al CSP, nella tabella sono omesse le seguenti colonne: output (se SAT: sono riportati i risultati, se UNSAT: 'UNSAT', se timeout: '?'), VarChoice e ConstrainChoice (i cui valori sono uguali alle tabelle precedenti: default se restart è default, indomain_random e first_fail altrimenti).

L'obiettivo è di massimizzare il minimo tra i ricavi delle università. Tale funzione si è rilevata migliore rispetto al minimizzare la stdev, in quanto quest'ultima trovava stdev piccole ma con un minimo molto più basso.

Si noti che la media di ricavi non è detto sia 500.00, ciò è dovuto agli errori di troncamento.

Sommando tali resti: $((tot\_money/n\ - mean) * n)$ si ottiene il totale del denaro andato "perso".

Ad esempio se mean=499.40: (1000/20 - 499.4) * 20 = 0.6 * 20 = 12,00 € persi.

	Time (s)	**Tot Money**	**Min Money**	**Min Found**	**Mean**	**STD**	**Restart**
MZN	300	10,000	160	165	499.80	674.53	default
MZN	300	10,000	160	173	499.75	674.46	luby(63)
MZN	300	10,000	160	165	500.00	703.12	geometric(1.5, 100)
ASP	300	1,000	16	17	49.35	67.74	
MZN	300	10,000	170	?	?	?	default
MZN	300	10,000	170	173	499.75	674.57	luby(63)
MZN	300	10,000	170	?	?	?	geometric(1.5, 100)
ASP	300	1,000	17	17	49.6	67.62	

Percentuali per tematica: [33%, 27%, 25%, 15%]

Dalla tabella si evidenzia un tempo sempre pari a 300, quindi nessuna delle esecuzioni riesce a terminare.

L'ASP trova la soluzione con min=17 sia imponendo il minimo a 16 sia a 17, come aveva fatto anche nel CSP.

Per MiniZinc accade l'opposto: imponendo min=173 con geometric o min=174 con luby il CSP trova una soluzione (si riveda la tabella della pagina precedente), mentre imponendo la massimizzazione del minimo, partendo da 170, il geometric va in timeout senza trovare nulla, mentre luby trova min=173, che è peggiore del CSP con min=174.

Questi fenomeni inducono a pensare che, se il tempo è limitato, è meglio usare un CSP con un parametro "difficile" che un COP con un parametro "facile" da minimizzare o massimizzare strada facendo.

Questo probabilmente perché molti sotto rami dell'albero con il CSP sono rimossi subito se non rispettano il vincolo, mentre con il COP all'inizio esegue del lavoro inutile trovando soluzioni ma migliorative della funzione di costo ma in realtà sono facili da scoprire.