Relazione progetto di Intelligenza Artificiale e Laboratorio

Modulo: Programmazione Logica, Ragionamento Non Monotono, Answer Set Programming

Andreata Chiara, Gerbaudo Luca, Racca Andrea Rocco

a.a. 2019/2020

Strategie di ricerca

Ricerca Iterative Deepening

Si tratta di una ricerca a profondità limitata in cui il limite inizialmente è posto a 1 e ad ogni iterazione viene incrementato.
Ad ogni iterazione viene richiamata la ricerca in profondità limitata a partire dallo stato iniziale fino a quando non si soddisfano le condizioni per la terminazione, il limite viene incrementato di 1 e la ricerca a profondià limitata viene richiamata.
La ricerca termina se viene raggiunto uno stato goal, o in caso contrario se il limite supera il numero massimo di celle contenute nella matrice del labirinto.
Se una soluzione è raggiungibile essa è anche ottima, poichè il cammino percorso per raggiungere lo stato di goal è garantito essere il più breve.

Analisi dei risultati variando le dimensioni del labirinto e la disposizione degli ostacoli

Per testarne l'efficienza, l'algoritmo è stato testato su labirinti simili a quello visto a lezione ma di dimensione diversa: abbiamo mantenuto uguale la disposizione della cella start, e degli ostacoli ad essa vicino, e abbiamo spostato la cella goal con i relativi ostacoli sul limite inferiore destro della griglia.
Con labirinti di dimensioni ridotte (fino a 10x10) i tempi di computazione sono nell'ordine di qualche secondo, ma già aumentando di poco le dimensioni (12x12), si passa subito nell'ordine dei minuti. Il labirinto è stato testato fino ad una dimensione 15x15 dove i tempi sono aumentati considerevolmente, addirittura sopra l'ora di computazione.

Dimensione Labirinto	Tempo di Esecuzione
10x10	2”
11x11	4”
12x12	30”
13x13	21’ 11”
14x14	>60’ (terminato con arresto forzato)
15x15	>60’ (terminato con arresto forzato)

Table 1: Tempi misurati durante le prove di ID

Nel caso in cui lo stato iniziale sia abbastanza vicino allo stato finale, anche se la dimensione del labirinto aumenta notevolmente, il tempo di esecuzione rimane nell'ordine dei secondi, anche con labirinti 100x100.
Allo stesso modo, nel caso in cui lo stato iniziale si trova molto lontano dallo stato finale ma siano presenti molte celle occupate a creare un 'percorso quasi obbligato' l'esecuzione termina nel giro di pochi secondi anche con labirinti di dimensioni più elevate (es. 15x15).

Analisi dei risultati in presenza di più uscite

L'algoritmo di ricerca Iterative Deepening è stato testato su più matrici in cui erano presenti più stati goal. L'algoritmo resitituisce in ogni caso lo stato goal più vicino allo stato di partenza.
I test sono inoltre stati ripetuti scambiando l'ordine degli stati goal nel codice senza notare alterazioni nei risultati.

Possibilità che nessuna delle uscite sia raggiungibile dalla posizione iniziale

L'algoritmo di ricerca Iterative Deepening è stato infine testato su matrici in cui gli ostacoli non permettevano il raggiungimento dello stato goal.
Il controllo posto sul limite, che impedisce che aumenti oltre la soglia massima, garantisce che vi sia una condizione di terminazione anche nel caso in cui nessuno stato goal non sia raggiungibile. La soglia massima che il limite aumentando durante la computazione può raggiungere è pari al prodotto delle dimensioni della matrice, ovvero pari al numero di caselle contenute nel labirinto.

Ricerca IDA*

Iterative deepening A* è un algoritmo euristico ed è in grado di trovare il cammino minimo fra un nodo iniziale e un nodo finale indicato come goal. L'algoritmo è una variante dell'Iterative Deepening, illustrato in precedenza, e migliora le prestazioni di A* per quanto riguarda l'uso della memoria, ma rispetto ad esso ha un tempo computazionale notevolmente più elevato proprio a causa della poca memoria utilizzata.
Esattamente come nell'Iterative Deepening, l'algoritmo viene eseguito più volte con una soglia sempre più grande finchè non viene trovata la soluzione. In questo caso la soglia assume il valore della funzione di valutazione: $f(n) = g(n) + h(n)$ dove g è il costo accumulato per raggiungere il nodo n, mentre h è una stima euristica del costo necessario per arrivare alla soluzione partendo da n.
$h(n)$ assume il valore uguale al numero di azioni minime che da n portano ad uno stato goal senza tener conto degli ostacoli che può incontrare: idealmente, rappresenta lo spostamento lungo l'asse delle ascisse e delle ordinate per raggiungere uno stato finale.

Analisi dei risultati variando le dimensioni del labirinto e la disposizione degli ostacoli

Per testarne l'efficienza come per l'iterative deepening l'algoritmo è stato testato su labirinti simili a quello visto a lezione ma di dimensione diversa, quindi abbiamo mantenuto uguale la disposizione

della cella start e degli ostacoli ad essa vicino e abbiamo spostato la cella goal con i relativi ostacoli ad essa vicino sul limite inferiore destro della griglia.
Con labirinti di dimensioni ridotte (fino a 15x15) i tempi di computazione sono nell'ordine di qualche secondo, ma già aumentando di poco le dimensioni (20x20), si passa subito nell'ordine dei minuti. Il labirinto è stato testato fino ad una dimensione massima 25x25 dove i tempi computazionali sono arrivati a decine di minuti.

Dimensione Labirinto	Tempo di Esecuzione
10x10	Immediato
15x15	3"
18x18	25"
19x19	48"
20x20	1' 27"
21x21	4' 16"
25x25	22' 04"
27x27	>60' (terminato con arresto forzato)

Table 2: Tempi misurati durante le prove di IDA*

Nel caso in cui lo stato iniziale sia abbastanza vicino allo stato finale, anche se la dimensione del labirinto aumenta notevolmente, il tempo di esecuzione rimane nell'ordine dei secondi anche con labirinti 100x100.
Anche nel caso in cui lo stato iniziale sia molto lontano dallo stato finale, ma il numero di celle occupate sia elevato e crei un 'percorso quasi obbligatorio', l'esecuzione termina nel giro di pochi secondi anche con labirinti di dimensioni molto maggiori.

Analisi dei risultati in presenza di più uscite

L'algoritmo di ricerca IDA* è stato testato su labirinti contenenti più uscite, quindi più stati goal.
In questo caso l'algoritmo ha sempre trovato il cammino minimo per raggiungere il goal più vicino, i tempi computazionali sono simili a quelli precedentemente descritti in base alla posizione della cella iniziale e finale e alla dimensione.

Possibilità che nessuna delle uscite sia raggiungibile dalla posizione iniziale

L'algoritmo di ricerca IDA* è stato testato su labirinti senza uscite, ovvero con l'uscita bloccata da un muro di ostacoli o fuori dalla griglia.
In questo caso l'algoritmo termina restituendo $false$, in quanto non gli è possibile trovare una soluzione. I tempi computazionali rimangono simili a quelli precedentemente descritti, quindi con il variare della dimensione della griglia, il tempo di esecuzione aumenta notevolmente.

Ricerca A*

L'algoritmo A* realizza un'evoluzione di una ricerca in ampiezza che permette di associare ad ogni nodo una stima del percorso migliore per raggiungere uno stato goal.
Essa viene calcolata con la seguente funzione: $f(n) = g(n) + h(n)$. $g(n)$ rappresenta il costo effettivo del camino dal nodo iniziale ad n, mentre $h(n)$ rappresenta una stima euristica del costo necessario per raggiungere uno stato goal a partire dal nodo n.
$h(n)$, come per IDA*, assume il valore uguale al numero di azioni minime che da n portano ad uno stato goal senza tener conto degli ostacoli che può incontrare: idealmente, rappresenta lo spostamento lungo l'asse delle ascisse e delle ordinate per raggiungere uno stato finale.

Trattandosi di una ricerca in ampiezza, A* si occupa di espandere il nodo selezionato in base alla funzione euristica andandone a visitare interamente i rispettivi figli, il grafo è analizzato per livelli. Questo algoritmo tiene altresì conto dei nodi precedentemente visitati, evitando di andarli ad espandere nuovamente nel caso si visitino nelle iterazioni successive alla prima.
Dei tre algoritmi realizzati per la ricerca di percorsi nei labirinti, A* è senz'altro il più rapido a trovare una soluzione. Questa performance è dovuta all'utilizzo di una coda nella quale A* salva i nodi visitati andandoli ad ordinare in modo decrescente in base al valore della funzione sopra citata. Per ordinare tale coda è stato utilizzato l'algoritmo di ordinamento "insertion sort".
A* è un algoritmo ottimo e completo. Ottimo in quanto garantisce di raggiungere uno stato finale seguendo il cammino migliore e quindi meno costoso, e completo perchè permettere di raggiungere in ogni caso lo stato goal, a condizione che questo sia presente.

Analisi dei risultati variando le dimensioni del labirinto e la disposizione degli ostacoli

Come per gli algoritmi precedentemente descritti, A* è stato testato su labirinti differenti per dimensioni e disposizioni degli ostacoli.
Durante l'esecuzione su diversi labirinti è subito saltato all'occhio quanto questo algoritmo sia il migliore dei tre implementati. È stato confrontato con IDA* su un labirinto 25x30, nel quale vi è una strada "obbligatoria" da seguire per raggiungere l'uscita, ed ha trovato soluzioni in poco più di 3 secondi, mentre IDA*, non è riuscito a terminare nemmeno dopo un'ora di computazione, ed è stato arrestato in maniera forzata.
Abbiamo messo alla prova A* su labirinti di dimensioni sempre maggiori ottenendo buoni risultati fino alla dimensione 30x30, per la quale l'algoritmo ha trovato la soluzione dopo 34 secondi.
Per questi labirinti di grosse dimensioni è stato necessario estendere la dimensione dello stack utilizzato da SWI-PROLOG per eseguire la ricerca, poiché giungeva in poco ad occupare più di 10GB, quando la dimensione di default è di solamente 1GB.
Tentando di risolvere un labirinto 40x40, invece, è stato necessario terminare in maniera forzata l'esecuzione dell'algoritmo in quanto giungeva ad occupare sopra i 12GB di memoria portando vistosi rallentamenti al PC.
L'analisi di questi risultati ha permesso di dimostrare quanto A* sia rapido ed efficiente, anche se può generare problemi quando si deve andare a salvare un numero eccessivo di elementi nella coda dei nodi visitati.

Dimensione Labirinto	Tempo di Esecuzione
10x10	immediato
20x20	24"
30x30	1' 35"
40x40	>12GB di RAM (terminato con arresto forzato)

Table 3: Tempi misurati durante le prove di A*

Analisi dei risultati in presenza di più uscite

Come per Iterative Deepening, anche con A* in caso di più stati finali viene raggiunto quello più vicino allo stato iniziale, l'euristica ha infatti un valore minore per lo stato goal raggiungibile con meno spostamenti, non viene tenuto conto della presenza di ostacoli.

Possibilità che nessuna delle uscite sia raggiungibile dalla posizione iniziale

Essendo una ricerca in ampiezza, A* tiene conto dei nodi visitati, salvandoli in una lista apposita, e in questo modo consente all'algoritmo di terminare una volta visitato interamente il grafo anche

se non è presente uno stato goal. Ciò consente di non far proseguire la ricerca all'infinito in cerca di una soluzione impossibile da trovare.

ASP

Implementazione

Per prima cosa abbiamo inserito i fatti necessari a rappresentare: numero di settimane, giorni e ore in cui andrà suddiviso il calendario del corso, quali insegnamenti, docenti e quali relazioni tra questi ultimi due elementi sono presenti.

- Numero di settimane disponibili: $settimana(1..24)$.
- Numero di giorni disponibili: $giorno(1..6)$.
- Numero di ore disponibili: $ore(1..8)$.
- Elenco di insegnamenti disponibili:
  - $insegnamento(nomeInsegnamento1)$.
  - $insegnamento(nomeInsegnamento2)$.
  - ....
  - $insegnamento(nomeInsegnamentoN)$.
- Lista dei docenti:
  - $docente(nomeDocente1)$.
  - $docente(nomeDocente2)$.
  - ....
  - $docente(nomeDocenteN)$.
- Lista delle relazioni Docente-Insegnamento:
  - $insegna(nomeInsegnamento1, nomeDocente1)$.
  - $insegna(nomeInsegnamento2, nomeDocente2)$.
  - ....
  - $insegna(nomeInsegnamentoN, nomeDocenteN)$.

Successivamente siamo passati alla creazione delle lezioni: esse sono definite, per ogni settimana del calendario, mediante il numero di giorni di lezione contenuti nella settimana e, per ogni giorno, il totale delle ore. Questa suddivisione è dettata dal dover gestire settimane e giorni con orari diversi. La settimana 7 e 16, a differenza delle altre nelle quali vi sono lezioni solo il venerdì e il sabato, sono full-time (dal lunedì al sabato). Inoltre, il sabato deve contenere 4 o 5 ore.
Per implementare queste informazioni, abbiamo creato una regola per ogni specifica tipologia di combinazione giorno-settimana: una regola per gestire le ore dei venerdì in settimane part-time, una seconda per la gestione del sabato in tali settimane, e regole per gestire i giorni in settimane full-time.

```

% Definizione degli slot del calendario, lezioni, in base alla settimana
% Settimana 7, giorni lun-ven, max 8h di lezione
lezione(I,S,G,O) :-
    insegnamento(I), settimana(S), giorno(G), ora(O),
    S == 7, G < 6 , si_svolge(I,S,G,O).

% Settimana 7, sabato, max 5h di lezione
lezione(I,S,G,O) :-
    insegnamento(I), settimana(S), giorno(G), ora(O),
    S == 7, G == 6, O < 6, si_svolge(I,S,G,O).

% Settimana 16, giorni lun-ven, max 8h di lezione
lezione(I,S,G,O) :-
    insegnamento(I), settimana(S), giorno(G), ora(O),
    S == 16, G < 6,  si_svolge(I,S,G,O).

% Settimana 16, sabato, max 5h di lezione
lezione(I,S,G,O) :-
    insegnamento(I), settimana(S), giorno(G), ora(O),
    S == 16, G == 6, O < 6, si_svolge(I,S,G,O).

% Settimana normale, venerdi, max 8h di lezione 9

lezione(I,S,G,O) :-
    insegnamento(I), settimana(S), giorno(G), ora(O),
    S != 7, S!=16, G == 5, si_svolge(I,S,G,O).

% Settimana normale, sabato, max 5h di lezione
lezione(I,S,G,O) :-
    insegnamento(I), settimana(S), giorno(G), ora(O),
    S != 7, S!=16, G == 6, O < 6, si_svolge(I,S,G,O).
```

Per la gestione delle singole ore è stato necessario definire un aggregato, il quale determina che ogni slot orario contiene un unico insegnamento per evitare sovrapposizioni. Tale aggregato è formato da una tripla: $< settimana, giorno, ora >$.

```
% Il calendario viene suddiviso in slot orari, ciascuno contrassegnato dalla tripla
    settimana,giorno,ora
% A ciascuno slot del calendario viene eventualmente assegnato un insegnamento
{ si_svolge(I, S, G, O) : insegnamento(I) } 1 :- settimana(S), giorno(G), ora(O).
```

Al fine di tracciare le singole ore all'interno del calendario a ciascuna di esse è stato assegnato un identificatore mediante la regola denominata *idOra*. È assegnato un id anche alle ore in cui non sono presenti lezioni, ciò non comporta errori nei risultati perchè la distribuzione delle lezioni nelle ore non è determinata da questa regola, ma permette di avere un codice più snello.

```
% Ogni ora, o slot, del calendario viene contrasseganata con un id progressivo nel
    soddisfacimento di alcuni vincoli
% gli idOra vanno da 1 per la prima ora del primo giorno della prima settimana ed
    aumentano progressivamente
% fino alla ultima ora del sesto giorno della ventiquattresima settimana
```

```
4
5 idOra(S, G, O, ID) :-
6     settimana(S), giorno(G), ora(O),
7     ID = (S - 1) * 48 + (G - 1) * 8 + O.
```

Terminata la creazione dello scheletro del progetto, siamo passati alla scrittura della regola "goal", in essa vi sono i vincoli tra il numero di ore di lezione per materia e il numero totale fornito dal testo del progetto. Per fare ciò abbiamo sfruttato la funzione *count* per tenere il conto delle ore di lezione di ciascuna materia.

Dopo questa fase iniziale di programmazione, abbiamo eseguito il programma constatando la correttezza delle regole e dei predicati. Abbiamo notato che la soluzione proposta da *clingo* rispettava tutti i vincoli inseriti, pur essendo semplici e minimali.
Dato che l'output fornito da *clingo* non è molto intuitivo e facile da leggere, abbiamo deciso di utilizzare uno script in *python* in grado di parsificare la soluzione e stamparla a video in modo più comprensibile.

```
#SETTIMANA 1
        VEN
                1 : presentazione_master
                2 : presentazione_master
                3 : pm
                4 : pm
                5 : ict
                6 : pm
                7 : pm
                8 : ict

        SAB
                1 : pm
                3 : pm
                4 : pm
                5 : pm

#SETTIMANA 2
        VEN
                1 : pm
                2 : pm
                3 : pm
                4 : lm
                5 : pm
                6 : lm
                7 : lm
                8 : lm
```

Figure 1: Esempio di output parsificato da python.

Utilizzo di un problema giocattolo per l'implementazione dei vincoli

Appurata la correttezza del codice precedentemente scritto, abbiamo proseguito con la scrittura dei vincoli rigidi.
Per far fronte a tempistiche troppo elevate di esecuzione, che molto spesso intralciavano e ostacolavano il nostro lavoro, abbiamo deciso di lavorare con un "toy-problem" ovvero con una versione molto ridotta del progetto. Questa versione giocattolo si compone di: 8 corsi, 5 docenti e 6 settimane di cui la terza e la quinta full-time. Le ore di lezione per le settimane full-time e part-time rimangono le stesse del problema originale, uno dei docenti tiene due corsi e un secondo docente ne tiene tre.
L'implementazione e la sperimentazione dei vincoli rigidi su questo sistema ridotto è stata notevolmente agevolata: testare uno stesso vincolo sul problema giocattolo, infatti, piuttosto che sul prob-

lema originale ha permesso di lavorare con tempi di computazione al più nell'ordine dei secondi. Dal momento che il toy-problem rappresenta un ambiente molto ridotto, soprattutto per quanto riguarda il numero di corsi, è stato utilizzato per implementare e sviluppare solo i primi 4 dei 7 vincoli rigidi e poi opportunamente modificato per soddisfare i vincoli del problema in esame.

Analisi implementazione dei vincoli

Vincolo #1

```
%--- Vincolo 1 -- max 4 ore al giorno per docente ---
:- insegnamento(I),settimana(S), giorno(G),ora(O), lezione(I, S, G, O),
   #count{I1: lezione(I1, S, G, O), insegnamento(I1), docente(D), insegna(I1,D)} > 4.
```

Il vincolo #1 è stato implementato tramite l'operatore *#count* per effettuare un conteggio sulle ore che un docente può tenere in un certo giorno di una certa settimana. Il conteggio di tali ore è garantito essere non maggiore di 4 poichè si tratta di un integrity constraint.

Vincolo #2

```
%--- Vincolo 2 -- ciascun insegnamento min 2 e max 4 ore al giorno ---
:- insegnamento(I),settimana(S), giorno(G),ora(O), lezione(I, S, G, O), #count{O1:
    lezione(I, S, G, O1), ora(O1)} < 2.
:- insegnamento(I),settimana(S), giorno(G),ora(O), lezione(I, S, G, O), #count{O1:
    lezione(I, S, G, O1), ora(O1)} > 4.
```

Come il precedente, anche per l'implementazione di questo vincolo sono stati utilizzati gli integrity constraints: un insegnamento in un giorno di una settimana non può avere meno di due ore e non più di quattro.

Vincolo #3

```
%--- Vincolo 3 -- 2 ore di presentazione master il primo giorno ---
v3 :- lezione(presentazione_master, 1, 5, 1), lezione(presentazione_master, 1, 5, 2).
```

L'implementazione di questo vincolo è stata piuttosto semplice: è bastato imporre che le prime due ore in assoluto fossero occupate dal "corso presentazione_master" : la prima e la seconda ora del giorno 5 (venerdì) della prima settimana.

Vincolo #4

```
%--- Vincolo 4 -- 2 blocchi da 2 ore per recuperi ---
:- lezione(recupero, S, G, O), O < O2, lezione(recupero, S, G, O2), O+1 != O2.
```

Anche per implementare questo vincolo abbiamo utilizzato gli integrity constraints per obbligare in un dato giorno di una data settimana le ore assegnabili a "recupero" in slot consecutivi. Nella regola "goal" è inoltre stato aggiunto il corso "recupero" vincolato ad avere 4 ore totali.

Vincolo #5

```
%--- Vincolo 5 --  insegnamento Project Management deve concludersi entro prima
    settimana full-time --
:- settimana(S), giorno(G), ora(O), S > 7, lezione(pm, S, G, O).
```

Per realizzare questo vincolo abbiamo semplicemente imposto che non fosse possibile inserirlo in settimane precedenti la settima mediante l'uso degli integrity constraints.

Vincolo #6

```
%--- Vincolo 6 --  la prima lezione di progmulti collocata prima del termine lezioni di lm --
v6 :- settimana(S_ACC), giorno(G_ACC), ora(O_ACC), lezione(progmulti, S_ACC, G_ACC, O_ACC),
    settimana(S_LM), giorno(G_LM), ora(O_LM), lezione(lm, S_LM, G_LM, O_LM),
    idOra(S_ACC, G_ACC, O_ACC, N_ACC), idOra(S_LM, G_LM, O_LM, N_LM), N_ACC < N_LM.
```

Questo vincolo è realizzato sfruttando l'dentificatore univoco per le ore *idOra* in modo da imporre che vi sia almeno un'ora di "Accessibilità e usabilità nella progettazione multimediale" (progmulti) con id minore di un'ora di "Linguaggi di markup" (lm), e quindi che vi sia almeno un'ora di progmulti prima che finisca lm.

Vincolo #7

```
%--- Vincolo 7 -- ogni insegnamento precedente deve concludersi prima della prima ora del successivo --
% Ovvero ogni idOra degli insegnamenti successivi non devono essere minori degli idOra degli insegnamenti precedenti

% prima ora di lcs dopo ultima ora di ict
:-  settimana(S_LCS), giorno(G_LCS), ora(O_LCS), lezione(lcs, S_LCS, G_LCS, O_LCS), idOra(S_LCS, G_LCS, O_LCS, N_LCS),
    settimana(S_ICT), giorno(G_ICT), ora(O_ICT), lezione(ict, S_ICT, G_ICT, O_ICT), idOra(S_ICT, G_ICT, O_ICT, N_ICT),
    N_LCS < N_ICT.
```

Il codice del vincolo #7 non è riportato per intero perchè molto lungo e ripetitivo.
Scrivendo il vincolo rigido #7, abbiamo notato che implementandolo in un'unica regola, i tempi di esecuzione erano molto elevati, tanto che, dopo ore di computazione, siamo stati obbligati ad interrompere l'esecuzione in modo forzato. Abbiamo provato a riscrivere la regola più volte, arrivando ad ottenere un risultato stilisticamente discreto con poche righe di codice e molto comprensibile ma il tempo di computazione non si è ridotto, anzi è aumentato. Dividendo la regola in molte sotto-regole (come mostrato nell'estratto di codice), invece, i tempi di esecuzione si sono ridotti notevolmente, portando a termine l'esecuzione in un paio di minuti.

```
% v7 la prima lezione di insegnamento_successivo deve avvenire dopo ultima di insegnamento_precedente
%   slotTempo(insegnamento_successivo) > slotTempo(insegnamento_precedente)
v7 :- settimana(S_SLC), giorno(G_SLC), ora(O_SLC), lezione(lcs, S_SLC, G_SLC, O_SLC), slotTempo(S_SLC, G_SLC, O_SLC, N_SLC),
    settimana(S_ICT), giorno(G_ICT), ora(O_ICT), lezione(ict, S_ICT, G_ICT, O_ICT), slotTempo(S_ICT, G_ICT, O_ICT, N_ICT),
    settimana(S_PSAWDMI), giorno(G_PSAWDMI), ora(O_PSAWDMI), lezione(psawdmI, S_PSAWDMI, G_PSAWDMI, O_PSAWDMI), slotTempo(S_PSAWDMI, G_PSAWDMI, O_PSAWDMI, N_PSAWDMI),
    settimana(S_PSAWMII), giorno(G_PSAWMII), ora(O_PSAWMII), lezione(psawmII, S_PSAWMII, G_PSAWMII, O_PSAWMII), slotTempo(S_PSAWMII, G_PSAWMII, O_PSAWMII, N_PSAWMII),
    settimana(S_PBD), giorno(G_PBD), ora(O_PBD), lezione(pBD, S_PBD, G_PBD, O_PBD), slotTempo(S_PBD, G_PBD, O_PBD, N_PBD),
    settimana(S_TSS), giorno(G_TSS), ora(O_TSS), lezione(tss, S_TSS, G_TSS, O_TSS), slotTempo(S_TSS, G_TSS, O_TSS, N_TSS),
    settimana(S_LM), giorno(G_LM), ora(O_LM), lezione(lm, S_LM, G_LM, O_LM), slotTempo(S_LM, G_LM, O_LM, N_LM),
    settimana(S_PM), giorno(G_PM), ora(O_PM), lezione(pm, S_PM, G_PM, O_PM), slotTempo(S_PM, G_PM, O_PM, N_PM),
    settimana(S_MD), giorno(G_MD), ora(O_MD), lezione(md, S_MD, G_MD, O_MD), slotTempo(S_MD, G_MD, O_MD, N_MD),
    settimana(S_TSMD), giorno(G_TSMD), ora(O_TSMD), lezione(tsmd, S_TSMD, G_TSMD, O_TSMD), slotTempo(S_TSMD, G_TSMD, O_TSMD, N_TSMD),
    settimana(S_SM), giorno(G_SM), ora(O_SM), lezione(sm, S_SM, G_SM, O_SM), slotTempo(S_SM, G_SM, O_SM, N_SM),
    settimana(S_PGDI), giorno(G_PGDI), ora(O_PGDI), lezione(pgdi, S_PGDI, G_PGDI, O_PGDI), slotTempo(S_PGDI, G_PGDI, O_PGDI, N_PGDI),
    settimana(S_EIS), giorno(G_EIS), ora(O_EIS), lezione(eis, S_EIS, G_EIS, O_EIS), slotTempo(S_EIS, G_EIS, O_EIS, N_EIS),
    settimana(S_FT), giorno(G_FT), ora(O_FT), lezione(ft, S_FT, G_FT, O_FT), slotTempo(S_FT, G_FT, O_FT, N_FT),
    settimana(S_AEID), giorno(G_AEID), ora(O_AEID), lezione(aeid, S_AEID, G__AEID, O_AEID), slotTempo(S_AEID, G__AEID, O_AEID, N_AEID),
    settimana(S_G3D), giorno(G_G3D), ora(O_G3D), lezione(grafica3D, S_G3D, G_G3D, O_G3D), slotTempo(S_G3D, G_G3D, O_G3D, N_G3D),
    N_SLC > N_ICT,          % prima ora di lcs dopo ultima ora di ict
    N_PSAWDMI > N_SLC,      % prima ora di psawdmI dopo ultima ora di lcs
    N_PSAWMII > N_PSAWDMI,  % prima ora di psawmII dopo ultima ora di psawdmI
    N_TSS > N_PBD,          % prima ora di tss dopo ultima ora di pBD
    N_SLC > N_LM,           % prima ora di lcs dopo ultima ora di lm
    N_MD > N_PM,            % prima ora di md  dopo ultima ora di pm
    N_TSMD > N_MD,          % prima ora di tsmd dopo ultima ora di md
    N_SM > N_PM,            % prima ora di sm dopo ultima ora di sm
    N_PGDI > N_PM,          % prima ora di pgdi dopo ultima ora di sm
    N_FT > N_EIS,           % prima ora di ft dopo ultima ora di eis
    N_AEID > N_FT,          % prima ora di aeid dopo ultima ora di ft
    N_G3D > N_EIS.          % prima ora di grafica3D dopo ultima ora di eis
```

Figure 2: Prima versione del vincolo rigido #7 prima di essere suddiviso.

Analisi implementazione vincoli auspicabili

Vincolo auspicabile #1

```
%--- Vincolo auspicabile 1 -- Distanza tra prima e ultima ora di lezione < 6 settimane --
:-  settimana(S_1), giorno(G_1), ora(O_1), lezione(C_1, S_1, G_1, O_1),
    settimana(S_2), giorno(G_2), ora(O_2), lezione(C_2, S_2, G_2, O_2),
    C_1 == C_2,
    S_2 - S_1 > 5.
```

Nell'implementazione di questo vincolo abbiamo sfruttato nuovamente le proprietà degli integrity constraints per rendere impossibile avere più di 5 settimane tra qualunque lezione di uno stesso corso.

Vincolo auspicabile #2

```
%--- Vincolo auspicabile 2 -- Insegnamenti crossmedia e smm devono iniziare nella settimana 16 --
% entrambe le lezioni si svolgono dalla settimana 16, ed entrambe hanno almeno 1 ora in quella settimana
% vincolo suddiviso in due sottoregole per ridurre il tempo di esecuzione.
:-  lezione(crossmedia, S, G, O), giorno(G), ora(O), settimana(S), S < 16.
:-  lezione(smm, S, G, O), giorno(G), ora(O), settimana(S), S < 16.
v2aa :- lezione(crossmedia, 16, G, O).
v2ab :- lezione(smm, 16, G, O).
```

Per implementare il vincolo auspicabile #2 abbiamo deciso di suddividerlo in più sottovincoli per rendere più snello e leggibile il codice e per velocizzare i tempi di esecuzione. Sfruttando gli integrity constraint abbiamo reso impossibile inserire "crossmedia" e "smm" in settimane precedenti la sedicesima, e successivamente abbiamo vincolato entrambe le lezioni ad avere almeno uno slot nella settimana 16.

Vincolo auspicabile #3

```
%--- Vincolo auspicabile 3 -- Ogni insegnamento successivo deve iniziare 4 ore dopo il precedente --
propedeuticita(ict,pBD).
propedeuticita(tsmd,smm).
propedeuticita(cpub,vgp).
propedeuticita(tss,psawdmI).
:- propedeuticita(P1,P2), lezione(P2,S,G,O), idOra(S,G,O,ID1), #count{ID2:lezione(P1,S1,G1,O1),idOra(S1,G1,O1,ID2),ID2 < ID1} != 4.
```

Per realizzare questo vincolo è stato necessario introdurre alcuni nuovi fatti, nella forma $< propedeuticita(insegnamento1, insegnamento2). >$, per indicare che il primo insegnamento è propedeutico al secondo e deve quindi iniziare prima del successivo. Il vincolo sull'ordine di inizio, e sulle 4 ore di distanza tra i corsi, è posto tramite un integrity constraint e l'operatore *#count*.

Vincolo auspicabile #4

```
%--- Vincolo auspicabile 4 -- la distanza fra ultima lezione di psawdmI  e la prima di psawmII non deve superare le due settimane --
1{ultima_settimana_psawdmI(S)} 1:-lezione(psawdmI,S,G,O), #max {SS:lezione(psawdmI,SS,GG,OO)} = S.
1{prima_settimana_psawmII(S)} 1:-lezione(psawmII,S,G,O), #min {SS:lezione(psawmII,SS,GG,OO)} = S.
:- ultima_settimana_psawdmI(S1), prima_settimana_psawmII(S2), S2 - S1 > 2.
```

Il vincolo auspicabile #4 è stato da noi suddiviso in sottovincoli. In un primo momento abbiamo utilizzato degli aggregati, e due nuovi predicati, per indicare che vi può essere una sola settimana contrassegnata come ultima per il corso psawdmI (Progettazione e sviluppo di applicazioni web su dispositivi mobile I). Tramite l'operatore $\#max$ abbiamo indicato che tale settimana deve avere il numero massimo, ed essere dunque l'ultima per quel corso. In modo analogo abbiamo gestito la prima settimana in cui è presente una lezione del corso psawmII (Progettazione e sviluppo di applicazioni web su dispositivi mobile II). Abbiamo aggiunto in seguito un integrity constraint per vincolare la distanza tra l'inizio di un corso e la fine del secondo non superiore a 2 settimane.

Conclusioni

Durante lo sviluppo del progetto abbiamo notato che con l'inserimento di nuovi vincoli o con la modifica di essi, il tempo di esecuzione cambia notevolmente.
Dopo aver verificato la correttezza di tutte le regole, grazie all'output prodotto dal parser in *python*, abbiamo deciso di testare il tempo di esecuzione con varie combinazioni di presenza di vincoli. I vincoli rigidi sono sempre stati mantenuti tutti in ogni esecuzione, mentre quelli auspicabili sono stati man mano aggiunti o rimossi per testare il comportamento in presenza e assenza di essi.
Abbiamo notato che alcune regole non modificano molto il tempo di computazione, mentre il vincolo auspicabile #3, allunga sensibilmente il periodo di calcolo. Aggiungendo i vincoli auspicabili #1 e #3 la durata è vicina all'ora, invece, con tutti i vincoli rigidi e auspicabili il programma termina in circa 10 minuti restituendo una soluzione congrua alle aspettative.

Vincoli	Tempo di Esecuzione
VR	8”
VR + v1a	22”
VR + v2a	9”
VR + v3a	1’ 29”
VR + v4a	10”
VR + v1a + v2a	15”
VR + v1a + v3a	53’ 31”
VR + v1a + v4a	26”
VR + v2a + v3a	5’ 49”
VR + v3a + v4a	13”
VR + v1a + v2a + v3a	30’ 28”
VR + v1a + v2a + v4a	14”
VR + v1a + v3a + v4a	21’ 34”
VR + v2a + v3a + v4a	40”
TUTTI I VINCOLI	11’ 01”

VR: tutti i vincoli rigidi
v1a: vincolo auspicabile #1
v2a: vincolo auspicabile #2
v3a: vincolo auspicabile #3
v4a: vincolo auspicabile #4

Table 4: Tempi di esecuzione misurati durante le prove