Università degli Studi di Torino

FACOLTÀ DI INFORMATICA
Corso di Laurea Magistrale in Informatica

Intelligenza Artificiale e Laboratorio

Programmazione logica

Iodice Francesco
Matricola 813359

Carlo Alberto Barbano
Matricola 811588

Valentino Di Cianni
Matricola 800665

Anno Accademico 2020–2021

Contents

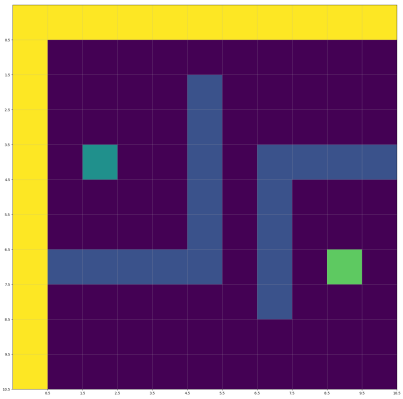
Sommario

A. Prolog

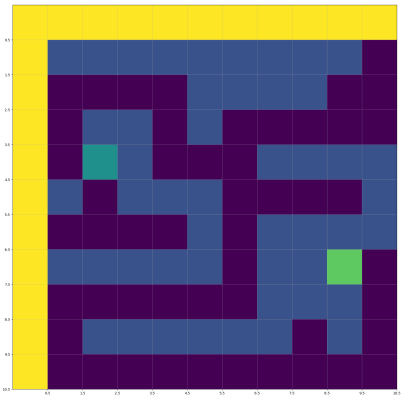
A.1. Introduzione

Di seguito sono illustrate le principali tecniche implementative utilizzate nell'implementazione di alcuni algoritmi di ricerca nello spazio degli stati, con il paradigma della programmazione logica. I risultati delle sperimentazioni sono stati confrontati tra di loro per mettere in evidenza i casi in cui alcuni algoritmi si comportano meglio di altri sulla variazione dello spazio degli stati.
Nello specifico, gli esperimenti sono stati svolti su una serie di labirinti in cui un agente parte da una posizione iniziale e deve raggiungere una posizione finale, evitando gli ostacoli. Sono state utilizzate diversi varianti, tra cui labirinti piccoli e grandi, poco o molto ostacolati e con una o più vie di uscita.
Di seguito una rappresentazione grafici dei labirinti utilizzati:

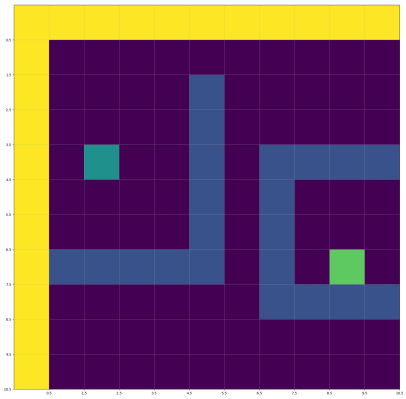
Labirinti 10x10



(A) Semplice

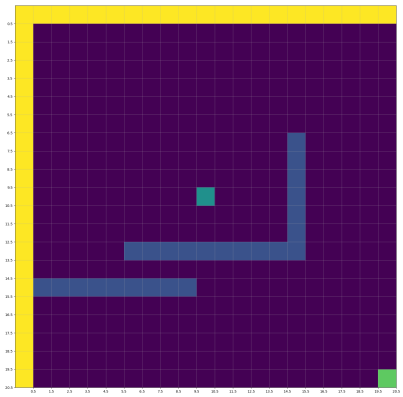


(B) Ostacolato

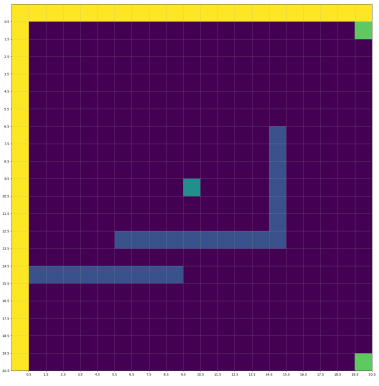


(C) Senza uscita

Labirinti 20x20

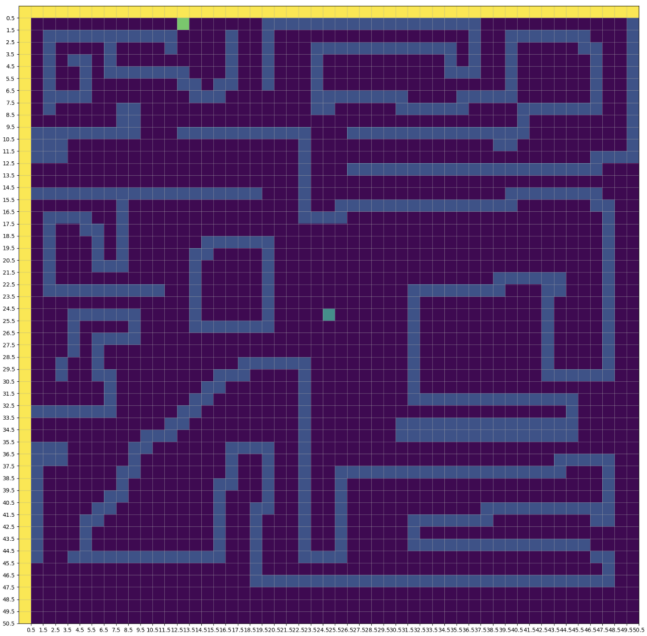


(A) Semplice

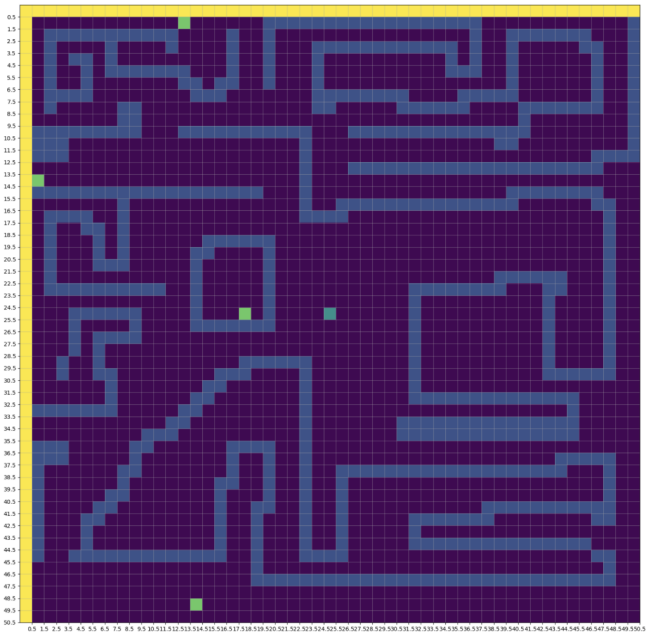


(B) Due uscite

Labirinti 50x50



(A) Semplice



(B) Quattro uscite, di cui una non raggiungibile

A.2. Metodo

A.2.1. Iterative deepening

Iterative deepening (ID) è una strategia di ricerca non informata (*blind*) in cui viene ripetuta una depth-first search, con un parametro *Soglia* che limita la profondità massima della ricerca. Nell'esempio del labirinto, *Soglia* rappresenta il numero massimo di azioni che l'algoritmo può effettuare per andare dal nodo di partenza al nodo goal.
Il funzionamento è il seguente: se *Soglia* vale *t* e la ricerca in profondità non trova nessuna strada di lunghezza $l \leq t$ che termina in uno stato finale, viene ripetuta la ricerca con una nuova Soglia pari a $t + 1$.
Di seguito sono illustrate le principali scelte implementative nello sviluppo dell'algoritmo iterative deepening. I predicati necessari usati per la realizzazione dell'algoritmo sono i seguenti:

- iterative_deepening(Soluzione, Profondita) : clausola di partenza, la clausola ausiliaria e alla terminazione delle ricerca indica se esiste il cammino dalla sorgente alla destinazione, con la profondità presa in parametro.
- iterative_deepening_aux(Soluzione, SogliaIniziale) : effettua la chiamata a depth_limit_search, e in caso di fallimento ricorre con la soglia incrementata di un'unità.
- depth_limit_search(Soluzione, Soglia) : effettua il match con la posizione iniziale e richiama dfs_aux
- dfs_aux(S, [Azione|AzioniTail], Visitati, Soglia) : algoritmo di visita in profondità

A.2.2. A*

A* è un algoritmo di ricerca informato. Estende l'algoritmo di *Dijkstra* con l'utilizzo una funzione euristica per guidare la visita. La funzione di valutazione per un nodo *n* nell'algoritmo A* è la seguente:

$$f(n) = g(n) + h(n)$$

dove *g(n)* rappresenta il costo del cammino dal nodo di partenza al nodo *n*, e *h(n)* rappresenta il costo ipotizzato (euristica) del cammino dal nodo *n* al nodo goal.

Applicando A* al dominio del labirinto sono state utilizzate due diverse euristiche:
- distanza di Manhattan
- distanza Euclidea

La distanza L_1 di **Manhattan** tra due punti: P_1 e P_2 di coordinate (x_1,y_1) e (x_2,y_2) è definita come segue:

$$L_1(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2|$$

La distanza d_E **euclidea** (*in linea d'aria*) è a la misura del segmento avente per estremi i due punti, espressa dalla seguente formula:

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_2)^2 + (y_2 - y_1)^2}$$

Di seguito sono illustrate le principali scelte implementative nello sviluppo dell'algoritmo A*.
I nodi espansi dall'algoritmo sono mantenuti in una lista *key-value* della forma *F-node(S, A, G, H)*

- *F* : valore della funzione euristica applicata al nodo S, nonché uguale a G + H
- *node* : predicato per indicare i dati significativi per ogni nodo
  - S : posizione del nodo all'interno del labirinto ☐ pos(X,Y)
  - A : lista delle azioni necessarie per giungere al nodo S dal nodo iniziale
  - G : valore della funzione G per il nodo S
  - H : valore della funzione H per il nodo S

L'utilizzo di una lista *key-value* rende possibile mantenere facilmente una frontiera ordinata per l'esplorazione degli stati.

I predicati necessari usati per la realizzazione dell'algoritmo sono i seguenti:

- astar() : clausola di partenza, associa un nodo alla posizione iniziale del labirinto, ne valuta il costo e richiama la funzione ausiliaria con in testa alla lista il nodo appena valutato
- astar_aux(N, Open, Closed, Res) : dato un nodo *N*, vengono calcolati tutti i nodi raggiungibili da *N* tramite il predicato *expandNode* e inseriti nelle lista *Open* in maniera ordinata rispetto al valore di F. Il nodo *N* viene inserito nella lista *Closed* e la ricerca prosegue per ricorsione sul nodo aperto con il minimo valore di F. *Open* e *Closed* rappresentano rispettivamente la lista dei nodi aperti e chiusi durante l'esplorazione. Al termine dell'esecuzione *Res* conterrà la sequenza di azioni (se esistente) per arrivare a uno stato finale.
- expandNode(N, Open, Closed, NewOpen) Espande un nodo *N* inserendo tutti i nuovi nodi trovati (che non sono nella lista *Closed*) nella lista *NewOpen*, utilizzando il predicato ausiliario *expandAll*
- expandAll(N, Open, ApplicableActions, Closed, NewOpen) : Dato un nodo *N* e la lista di azioni applicabili a quel nodo *ApplicableActions,* viene applicata l'azione in testa alla lista al nodo N generando N' (tramite il predicato *genNode*).
  Viene verificato che il nodo N' che non sia nella lista dei *Closed*.
  Se il nodo N' corrisponde a una posizione già presente nella lista *Open* viene mantenuto solo se il corrispondente F-score è migliore di quelli trovati in precedenza, altrimenti viene scartato. L'espansione viene applicata usando tutte le azioni in *ApplicableActions*.
- genNode(Action, A, B) : Data l'*Action* e il nodo *A* genero il nodo *B* e ne valuta il costo
- nodeHasBetterScore(N, Open) : Questo predicato è soddisfatto se *1. Open* non contiene un nodo con la stessa posizione di *N 2.* A *N* è associato uno score inferiore rispetto al nodo già presente in *Open* con la stessa posizione.

A.2.3. IDA*

Iterative deepening A* (IDA*) è in grado di trovare il cammino minimo fra un nodo indicato come iniziale e ciascun membro di un insieme di "nodi soluzione" in un grafo pesato.
L'algoritmo è una variante dell' ***iterative deepening depth-first search*** usata per migliorare le prestazioni di A*. Il vantaggio principale di IDA* è infatti l'uso lineare della memoria, al contrario di A* che ha bisogno, nel caso peggiore, di uno spazio esponenziale. D'altro canto, questo algoritmo usa fin troppa poca memoria, che potrebbe essere invece sfruttata per migliorare le prestazioni in termini di tempo.
Le euristiche utilizzate nell'implementazione dell'algoritmo sono le stesse descritte in precedenze quelle basate su:

- distanza euclidea
- distanza di Manhattan

Di seguito sono illustrate le principali scelte implementative nello sviluppo dell'algoritmo IDA*.
I nodi espansi dall'algoritmo sono mantenuti in una lista *key-value* della forma *F-node(S, A, G, H)*

- *F* : valore della funzione euristica applicata al nodo S, nonché uguale a G + H
- *node* : predicato per indicare i dati significativi per ogni nodo
  - S : posizione del nodo all'interno del labirinto ☐ pos(X,Y)
  - A : lista delle azioni necessarie per giungere al nodo S dal nodo iniziale
  - G : valore della funzione G per il nodo S
  - H : valore della funzione H per il nodo S

I predicati principali usati per la realizzazione dell'algoritmo sono i seguenti:

- ida_star() : clausola di partenza, associa un nodo alla posizione iniziale del labirinto, ne valuta il costo e richiama la funzione ausiliaria con in testa alla lista il nodo appena valutato
- ida_star_aux(S, Visited, Bound, Sol) : effettua la chiamata a ida_star_dfs_ordered con il Bound uguale al valore di F associato al primo nodo. Se la ricerca con quel Bound fallisce, vengono trovati dal database tutti i nodi che sono stati esplorati (tramite predicati dinamici), per trovare il minimo tra i bound maggiori di quello attuale e la ricerca viene ripetuta.
- ida_star_dfs_ordered(N, Visited, Bound, [Action | OtherActions]) : predicato utilizzato per effettuare l'espansione dei nodi in maniera ordinata (in base a F-score)
- ida_star_dfs(N, Visited, Bound, Action, Sol) : dal nodo N, applicando Action, trova il nodo N' e verifica che non sia già stato visitato. Verifica che l'F-score associato al nodo N' sia inferiore al Bound corrente, e in caso positivo continua la visita invocando il predicato *ida_star_dfs_ordered.*

Sono presenti anche altri predicati che svolgono funzioni accessorie, come *sort_actions*, *update_min_higher_bound*, *zip* e *unzip*.
L'implementazione dell'algoritmo è stata fatta cercando di ottimizzarne l'esecuzione:

- L'esplorazione dei nodi successori avviene in maniera ordinata in base all'euristica, per tentare strade migliori come prima alternativa. Contrariamente, si sarebbe potuto semplicemente scegliere un'azione a random (i.e. unificando *applicabile(Action, S)*) ma questo avrebbe potuto portare a scelte sub-ottimali.
- Per tenere traccia dei bound della frontiera, è stato fatto uso di predicati extra-logici quali *minHigherBound/1* e *currentBound/1* e del predicato *update_min_higher_bound/1* che si occupa di aggiornare il valore "globale" del prossimo bound durante l'esplorazione.

A.3. Discussione

Sono di seguito riportati i risultati delle simulazioni con gli algoritmi spiegati nella sezione A.2. In presenza di più stati finali per uno stesso labirinto, sono riportati i tempi relativi a ciascuna soluzione (è stato previsto un predicato *multi* che si occupa di ripetere la ricerca, se richiesto, sulle possibili soluzioni diverse). Se una soluzione non è raggiungibile, allora sono riportati i tempi relativi alle altre possibili soluzioni dello stesso labirinto (es. 50x50 con 4 uscite). In mancanza di queste ultime, è riportato il tempo che l'algoritmo impiega a determinare l'assenza di percorsi possibili.

Dimensione Labirinto	Tipo	Tempo
10x10	Ostacolato	~4ms
	Semplice	~2283ms
	Senza uscita	∞
20x20	Semplice	∞
	2 uscite	∞
50x50	Semplice	∞
	4 Uscite	∞

Tab.1: Risultati Iterative Deepening

Come si può osservare nei risultati riportati in Tab.1, quando la soluzione è a una profondità limitata la performance è accettabile. Al crescere della dimensione del labirinto (e quindi della profondità della soluzione) oppure in assenza di una soluzione (che rende necessario esplorare interamente il labirinto) l'algoritmo non termina in tempi ragionevoli. Questo rappresenta un risultato atteso, in quanto la complessità temporale di Iterative Deepening è $O(b^d)$, dove *b* è il branching factor e *d* è la profondità della soluzione.

Dimensione Labirinto	Tipo	Tempo	
		Manhattan	Euclide
10x10	Ostacolato	~7ms	~14ms
	Semplice	~177ms	~598ms
	Senza uscita	∞	∞
20x20	Semplice	~208ms	~14733ms
	2 uscite	< 1ms, ~213ms	~725ms, ~15356ms
50x50	Semplice	∞	∞
	4 Uscite	∞	∞

*Tab.2: Risultati IDA**

Risultati migliori si possono osservare in Tab.2, utilizzando la variante informata IDA*. La soluzione (quando presente) viene raggiunta in tempi ragionevoli nelle varianti 10x10 e 20x20 del labirinto. Nelle versioni più larghe (50x50), l'algoritmo impiega un tempo troppo elevato per terminare, con entrambe le euristiche. La versione senza soluzione del labirinto 10x10 soffre dello stesso problema, poichè per determinare l'assenza di un percorso è necessario esplorare completamente il labirinto, facendo degenerare l'algoritmo in un semplice iterative deepening.

Possiamo notare anche come la scelta della funzione di euristica influisca sul tempo di risoluzione: le performance ottenute sono peggiori impiegando la distanza euclidea d_E al posto di quella di manhattan. Questo peggioramento è dato da principalmente due motivi:

- In un gioco che consente mosse in sole 4 direzioni (come nel caso del labirinto) è preferibile utilizzare la misura di distanza di Manhattan (L1), poiché rispecchia meglio le possibili direzioni di esplorazione
- É preferibile che una funzione euristica ritorni la distanza più grande possibile (ovviamente rimanendo ammissibile, cioè non sovrastimandola), ma poiché $L_1 \geq d_E$, la distanza euclidea può in effetti sottostimare la distanza effettiva di un certo nodo dalla soluzione.

Dimensione Labirinto	**Tipo**	**Tempo**	
		Manhattan	**Euclide**
10x10	Ostacolato	< 1ms	~1ms
	Semplice	~1ms	~1ms
	Senza uscita	~2ms	~1ms
20x20	Semplice	~3ms	~12ms
	2 uscite	< 1ms, ~2ms	~5ms, ~12ms
50x50	Semplice	~42ms	~52ms
	4 Uscite	~2ms, ~42ms, ~43ms	~13ms, ~69ms, ~51ms

*Tab.3: Risultati A**

In Tab.3 sono riportati i risultati per l'algoritmo A*. Quest'ultimo algoritmo ha il pregio di fornire una risposta a tutte le configurazioni provate in tempi ragionevoli, anche se a discapito di una maggiore complessità spaziale rispetto a IDA* ($O(b^d)$ per A* e $O(bd)$ per IDA*, dove *b* è il branching factor e *d* è la profondità della soluzione). Anche in questo caso si può notare come l'utilizzo dell'euristica d_E porti a un peggioramento (seppur minore) nelle performance dell'algoritmo, similmente a quanto avviene con IDA*. Il trade-off tra complessità spaziale e temporale è il criterio di scelta per applicare algoritmi di questo tipo a problemi nel mondo reale, dove spesso A*, seppure completo, richiederebbe una quantità troppo elevata di memoria.

B. ASP

Di seguito sono illustrate le principali tecniche di modellazione e implementazione del problema della generazione del calendario del "Master in Progettazione e Management del Multimedia per la Comunicazione" dell'Università di Torino.
Tutti i requisiti sono descritti nel file ASP/Readme.md.

B.1. Modellazione

Dopo un'analisi dei requisiti richiesti, per la modellazione del problema si è deciso di utilizzare i seguenti predicati per astrarre la realtà descritta :

- *insegnamento**(corso, docente, ore)*** : tripletta in cui è presente il **docente** che deve svolgere il **corso** e il numero totale di **ore** da svolgere per quel corso.

- *giorno**(G)***: indica i giorni settimanali per cui è prevista lezione.
  Per imporre il seguente ordinamento tra i giorni:
  - lun < mar < mer < gio < ven < sab

  sono stati rappresentati come stringhe precedute da una lettera che ne imponeva l'ordinamento secondo l'ordine in cui occorreva la lettera nell'alfabeto.

  ```
  giorno(a_lun; b_mar; c_mer; d_gio; e_ven; f_sab).
  ```

- *orario**(Inizio, Fine)*** : rappresenta le coppie di valori che costituiscono Inizio e Fine di ogni possibile ora accademica.

- *giorni_lun_to_gio**(G)***: sono indicati i giorni in cui è prevista lezione relativi alle settimane full-time.

- *settimane**(S)*** : rappresenta le settimane in cui è prevista lezione

- *settimane_full**(S)*** : rappresenta le settimane full time

- *settimane_part**(S)*** : rappresenta le settimane part time

- *esami_propedeutici **(A, B)*** : rappresenta una propedeuticità tra il corso A e il corso B in particolare il corso B può iniziare solo dopo che il corso A si è concluso

- *propeudicità_specifica**(A,B)*** : analoga al predicato antecedente ma la propeudicità è diversa, ovvero il corso B può iniziare solo dopo che sono state erogate almeno 4 ore del corso A

Per indicare uno slot di un'ora di lezione relativa al calendario del Master, si è utilizzato il seguente predicato:

- slot**(G, I, F, C, D, O, S)** : in cui viene indicata un'ora del corso **C** tenuto dal docente **D** nella settimana **S**, nel giorno **G**, dalle ore **I** alle ore **F**.
  O rappresenta il numero totale di ore da erogare relative al corso **C.**

Per la sua realizzazione è stato realizzato il seguente predicato di supporto, rappresentante un unico slot orario all'interno del calendario scolastico mediante l'utilizzo dei *cardinality constraint* :

```
1 {orario_completo(Settimana, Giorno, Inizio, Fine) } 1 :-
    settimane(Settimana),
    orario(Inizio,Fine),
    giorno(Giorno).
```

Così da avere tutte le combinazione possibili non ripetute delle variabili **Settimana, Giorno, Inizio, Fine**.

Anche predicato **slot/8** è stato realizzato con l'utilizzo del *cardinality constraint* :

```
1{ slot(Giorno, Inizio, Fine, Corso, Docente, Oretotali, Settimana):
    orario_completo(Settimana, Giorno, Inizio, Fine) }:-
    insegnamento(Corso,Docente,Oretotali).
```

Ad ogni insegnamento **I** vengono assegnati da 1 a N slot orari.
Per ogni **I** il limite di N sarà poi determinato da un vincolo che scarta i modelli in cui N_I è diverso dalle ore totali che devono essere erogate dall'insegnamento **I**.

Nelle specifiche è richiesto che nelle prime 2 ore del giorno accademico ci sia la presentazione del master, e 4 ore per eventuali recuperi, a blocchi di 2 ore non nello stesso giorno. Questi 6 slot orari sono stati rappresentati come insegnamenti così da includerli nel mapping descritto e successivamente vincolati in base alle specifiche.

E' stato utilizzato l'aggregato **#count** per effettuare un confronto su più insiemi di slot secondo vari criteri e alcuni predicati di utilità:

- *ore_per_corso_erogate**(Corso, Count)*** : conteggio su quante ore **Count** sono erogate per **Corso**.
- *ore_giornaliere_docente**(Count, Docente, Giorno, Settimana)*** : conteggio su quante ore **Count** sono erogate dal **Docente** nel giorno **Giorno** della settimana **Settimana**.
- *ore_giornaliere_corso**(Count, Docente, Giorno, Settimana)*** : conteggio su quante ore **Count** sono erogate per il corso **Corso** nel giorno **Giorno** della settimana **Settimana**.
- *primo_slot_corso**(Giorno, Inizio, Settimana, Corso)*** : rappresenta il primo slot orario del corso Corso. Lo slot è ottenuto imponendo a 0 un conteggio:
    - **Count_week** : conteggio degli slot orari relativi a Corso nelle settimane posteriori a **Settimana**.
    - **Count_day** : conteggio degli slot orari relativi a **Corso** che a parità di **Settimana** sono distribuiti su giorni successivi **Giorno**.
    - **Count_hours** : conteggio degli slot orari relativi a **Corso** che a parità di Settimana e di **Giorno** avvengono in ore successive.
- *ultima_slot_corso**(Giorno, Inizio, Settimana, Corso)*** : rappresenta l'ultimo slot orario del corso **Corso**. Lo slot è ottenuto in modo analogo al precedente.

B.2. Implementazione e valutazione

I vincoli sono stati principalmente realizzati cercando di descrivere uno scenario che non rispettasse i requisiti del problema, così da escludere quel risultato.
Per l'implementazione dei vincoli è stato creato un subset del problema iniziale.
Dato il problema ***A***(C=25,S=25) dove C sono il numero di corsi da erogare e S il numero di settimane a disposizione è stato generato il problema ***B***(8,8).

La creazione del sottoproblema è stata necessaria per le seguenti motivazioni:

- il tempo necessario alla generazione dell'output del problema A cresceva esponenzialmente con il numero di vincoli inseriti.
- l'analisi completa dell'output, per verificare la consistenza dei vincoli del problema ***A*** richiede un effort consistente, causato dalla numerosa quantità di slot orari generati.

I vincoli di integrità introdotti nel modello per rispettare tutti i requisiti sono 25.
Le prime prove fatte sul problema A con al massimo 4 o 5 vincoli attivi generano un modello in circa 20 secondi. Con l'aggiunta del vincolo:

- *"A ciascun insegnamento per vengono assegnate minimo 2 e massimo 4 ore nello stesso giorno"*

Il tempo di realizzazione del modello sale a circa 1 ora e 25 minuti. Di conseguenza il modus operandi per l'implementazione è stato il seguente: si modella il vincolo sul problema ***B,*** se il vincolo è consistente lo si riporta sul problema ***A,*** se il vincolo non è consistente lo si modificava nel problema ***B***.
Sul modello B oltre ai vincoli di consistenza sulla realtà è stato mantenuto attivo solo 1 vincolo alla volta, quella in corso di realizzazione, così da avere meno tempo di attesa nella generazione del modello.

Per una visione più efficiente dell'output è stato creato uno script in python che mappa l'output prodotto dal solver in una struttura dati ordinata per settimana, giorno e ora di ogni lezione. Di seguito un frammento dell'output prodotto dallo script :

```
Settimana 3

Ven      9:00 - 10:00    Linguaggi di markup
                         Docente: Gena
Ven      10:00 - 11:00   I vincoli giuridici del progetto diritto dei media
                         Docente: Travostino
Ven      11:00 - 12:00   Project management
                         Docente: Muzzetto
Ven      12:00 - 13:00   I vincoli giuridici del progetto diritto dei media
                         Docente: Travostino
Ven      15:00 - 16:00   Linguaggi di markup
                         Docente: Gena
```

L'output formattato è presente nel file di testo **result.txt.**

Dopo aver verificato la consistenza di ogni vincolo sul problema B sono stati riportati tutti sul problema **A** il cui output è stato generato in **12008.787 s ~ 3.20 h**, di cui:

- Reading : **577 s**
- Solving : **11431 s**