Intelligenza Artificiale e Laboratorio

Anno Accademico 2017/18

Liccardo Francesco, Mittone Gianluca, Scaletta Marco

Programmazione Prolog e ASP

Implementazione di strategie di ricerca con Prolog

Tutti gli algoritmi di seguito descritti sono stati realizzati cercando di rendere il codice il più leggibile e semplice possibile: questo ha portato alla creazione di strutture molto modulari, formate da regole piccole, chiare e ben specializzate. Tale approccio si è rivelato determinante nelle fasi di debugging, testing e aggiornamento del codice semplificando e riducendo di molto i tempi e la quantità di lavoro richiesta per le suddette operazioni, aumentando però la verbosità generale dell'implementazione.

Si è inoltre scelto di limitare l'output degli algoritmi solamente al primo risultato, dato che è garantito essere ottimo. A livello implementativo tale politica è stata realizzata mediante l'eliminazione dei punti di scelta del backtracking considerati superflui, attraverso l'uso del *cut*.

Strategie non informate

Iterative Deepening

L'implementazione dell'algoritmo risulta semplice e naturale grazie al meccanismo di backtracking offerto da Prolog. Alla base vi è una ricerca a profondità limitata standard: nel caso lo stato in esame risulti finale la regola termina con successo, altrimenti viene esplorato uno stato successivo se esso non supera il livello di taglio. Esplorare uno stato significa generare una azione valida per esso, applicarla, controllare di non aver scoperto una configurazione già precedentemente analizzata ed, in tal caso, ripetere il procedimento dall'inizio. Tale algoritmo è denominato `ricerca_profondita_limitata` e viene eseguito ripetutamente aumentando il livello di taglio nel caso si superi tale valore senza che venga trovata una soluzione; questa ciclicità viene gestita dalla regola `ricerca_profondita_iterativa`.

L'inizializzazione dello stato iniziale e del livello di taglio sono invece delegate a `iterativeDeepening`, che è la regola da richiamare per avviare la ricerca:

```
iterativeDeepening(Soluzione)
```

dove `Soluzione` è la variabile che conterrà l'eventuale risultato.

La sequenza di azioni che conducono dallo stato iniziale a quello finale viene elaborata mediante una strategia backward: una volta raggiunto un obiettivo la lista soluzione viene costruita percorrendo a ritroso il ramo che ha portato al goal, aggiungendo in testa alla lista tutte le azioni che vengono trovate durante il cammino.

Strategie basate su euristica

Algoritmo A*

Essendo questo algoritmo più complesso rispetto al precedente, si rende necessaria la creazione di una struttura dati ad hoc per associare ad un determinato stato del problema tutte le informazioni ad esso correlate. Si definisce così un nodo:

```
nodo(Stato, DistanzaInizio, StimaEuristica, StimaTotale, Percorso)
```

dove:

- `Stato`: è uno stato proprio del dominio del problema;
- `DistanzaInizio`: è la distanza di Stato dallo stato iniziale;
- `StimaEuristica`: è la distanza stimata da Stato allo stato finale;
- `StimaTotale`: è la somma di `DistanzaInizio` e `StimaEuristica`;
- `Percorso`: è la lista delle azioni che hanno portato a `Stato` dallo stato iniziale.

Per avviare la ricerca è sufficiente digitare

```
aStar(Soluzione)
```

con `Soluzione` variabile che conterrà l'eventuale soluzione del problema. Questa regola inizializza lo stato iniziale, calcola la stima euristica della sua distanza dallo stato finale, inserisce tutte queste informazioni in un nodo e richiama `esplora`, che implementa 'ad alto livello' il cuore di `aStar`. Formalmente:

```
esplora(Frontiera, NodiEsplorati, Soluzione)
```

dove:

- `Frontiera` (o open set): la lista di nodi da essere valutati;
- `NodiEsplorati` (o close set): la lista di nodi già valutati;
- `Soluzione`: lista di mosse che hanno portano allo stato finale.

Il primo passo ad essere eseguito è il controllo della corrispondenza tra lo stato con il valore minimo di `StimaTotale` (quindi il più promettente per A*) e uno stato finale. In caso positivo l'esecuzione termina e viene restituito all'utente il campo `Percorso` del nodo in oggetto; altrimenti si procede alla generazione di tutti i suoi successori ed alla loro analisi, con conseguente aggiornamento della `Frontiera`, per poi ripetere esplora ricorsivamente.

La generazione dei successori utilizza il predicato `findall` per ottenere la lista completa delle azioni valide nello stato esaminato. Viene generato un nodo per ognuno dei possibili stati risultato dell'applicazione delle stesse e ne vengono compilati tutti i campi.

Ora la regola `analizza` si occuperà di implementare il principio della 'disuguaglianza triangolare': esaminerà ogni singolo nodo successore appena generato e in base a `Stato` e `StimaTotale` deciderà se debba essere inserito in `Frontiera`, `NodiEsplorati` o se non vada considerato. L'inserimento di un nodo in `Frontiera` avviene sempre in modo ordinato: il primo nodo della lista sarà sempre quello con il valore minimo di `StimaTotale`, semplificando così le operazioni di `esplora`. L'esecuzione ricomincia quindi dal controllo dello stato finale.

La scelta di adottare una struttura dati così completa per formalizzare un nodo ha il pregio di dover eseguire molti calcoli solamente una volta nella storia dello stesso, permettendo di realizzare delle regole ben specializzate e semplici, andando però ad appesantire la quantità di informazioni che l'algoritmo deve memorizzare e trasmettere da una regola all'altra.

Algoritmo IDA*

Questo algoritmo riutilizza la struttura `nodo` descritta precedentemente per associare ad un determinato stato del problema tutte le informazione ad esso correlate. Per avviare la ricerca è sufficiente digitare

```
idaStar(Soluzione)
```

come primo passo controlla se vi è la presenza di un limite soglia di un'esecuzione precedente dell'algoritmo. Successivamente, indipendentemente dal passo precedente, inizializza lo stato iniziale, calcola la stima euristica della sua distanza dallo stato finale, asserisce il limite soglia inizializzandolo con il valore `StimaEuristica` dello stato iniziale e infine chiama la regola `idaStarR`. Formalmente:

```
idaStarR(Stato, StimaEuristica, Soluzione)
```

dove:

- `Stato`: è uno stato proprio del dominio del problema;

- `StimaEuristica`: stima euristica del nodo;
- `Soluzione`: variabile che conterrà il percorso della soluzione.

All'interno del corpo della regola recupera l'informazione relativa al limite soglia, memorizzandola all'interno della variabile `Limite`, in seguito rimuove l'informazione per inizializzarla nuovamente al valore infinito (nel nostro caso a 99999, in quanto prolog non prevede una costante che identifica un valore che allegoricamente rappresenta l'infinito). Dopo aver eseguito questa sequenza di istruzioni, l'algoritmo invoca la regola `esplora` che rappresenta il "cuore" del processo. Formalmente:

```
esplora(Nodo, NodiEsplorati, Limite, Soluzione)
```

dove:

- `Nodo`: nodo da controllare;
- `NodiEsplorati`: la lista di nodi già valutati;
- `Limite`: limite soglia dell'attuale iterazione;
- `Soluzione`: lista di mosse che hanno portano allo stato finale.

Come primo passo controlla se la `StimaTotale` del nodo è minore o uguale del limite soglia, in caso positivo effettua un'ulteriore analisi che controlla se il nodo che sta analizzando è un nodo obiettivo. Se quest'ultimo test viene superato l'algoritmo termina e viene restituito all'utente il campo `Percorso` del nodo in oggetto; altrimenti si procede alla generazione del nodo successore, per poi ripetere l'esplora ricorsivamente.
Nel caso in cui nel primo passo la `StimaTotale` è maggiore del limite soglia viene controllato il valore soglia della prossima iterazione, chiamando la regola `verifica_limite`. Formalmente:

```
verifica_limite(StimaTotale)
```

dove:

- `StimaTotale`: stima totale del nodo che ha superato il limite soglia dell'esecuzione.

Innanzitutto recupera il valore soglia della prossima iterazione e lo memorizza nella variabile `ProssimoLimite`, successivamente verifica se quest'ultimo è minore o uguale della `StimaTotale`. In caso positivo il valore del prossimo limite soglia non verrà modificato, invece in caso negativo il valore limite della prossima iterazione viene inizializzato a `StimaTotale`.

Si ricorda che all'interno del prossimo valore soglia dovrà sempre esserci il valore minore fra tutte le stime dei costi dei nodi che hanno superato il valore soglia durante iterazione.
L'implementazione dell'algoritmo è stata possibile sfruttando i due predicati `assert` e `retract` che hanno l'effetto di modificare l'insieme delle clausole del programma (alla successiva invocazione del goal). Nel caso specifico di fallimento dovuto al superamento del valore soglia è stato possibile salvare (con `assert`) il valore della stima totale del costo del nodo prima di fare backtracking.

Domini

Prima di discutere i risultati dei test degli algoritmi implementati si rende necessario esplicitare su quali domini essi siano stati messi alla prova e di come essi siano stati modellati. Ai fini di questa relazione sono stati analizzati due modelli di dominio: il *labirinto* ed il *gioco dell'8*, entrambi in quattro varianti caratterizzati da livelli di difficoltà diversi.

Il *labirinto* è stato modellato attraverso dei fatti che ne descrivono il numero di righe, di colonne, le caselle occupate, la posizione di partenza e quella di arrivo. Una posizione è denotata da `pos(Riga, Colonna)` dove i parametri sono autoesplicativi. Sono rese disponibili due possibili scenari: uno da 10x10 caselle ed uno da 20x20, ed ognuno di essi con due disposizioni delle posizioni di partenza ed arrivo, una semplice ed una difficile.

Il gioco dell'8 invece mette a disposizione solo tre fatti: la configurazione di partenza, quella di arrivo ed il lato della griglia di gioco. Una posizione viene rappresentata mediante una lista che elenca il numero delle tessere da destra verso sinistra, dall'alto verso il basso, con una x nella posizione vuota. Anche questo dominio è stato realizzato in due varianti: una con una griglia 3x3 (gioco dell'8) ed una con una griglia 4x4 (gioco del 15), entrambe con due posizioni di partenza, una semplice ed una difficile.

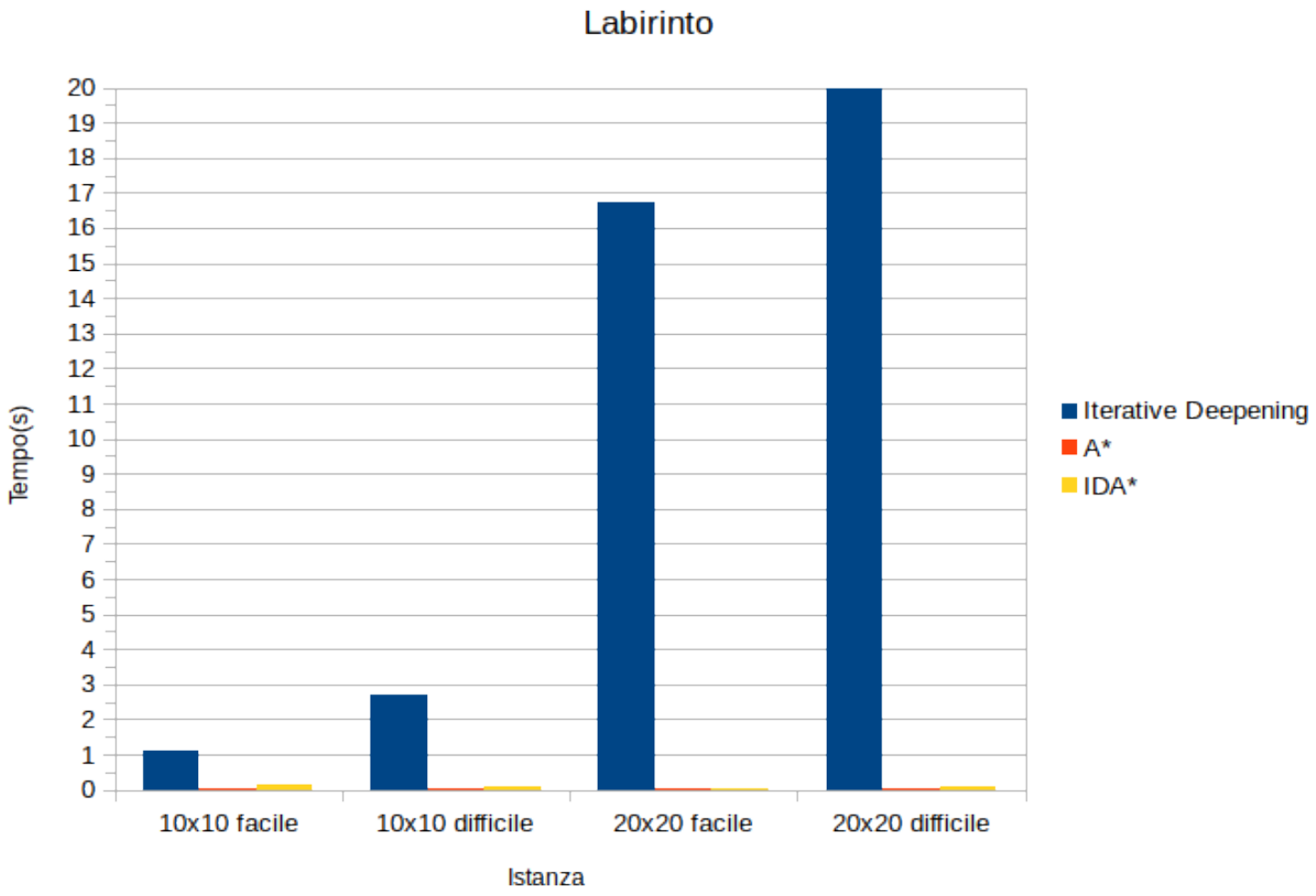
Ogni dominio è inoltre affiancato da un file contenente le relative azioni, che sono sempre quattro:

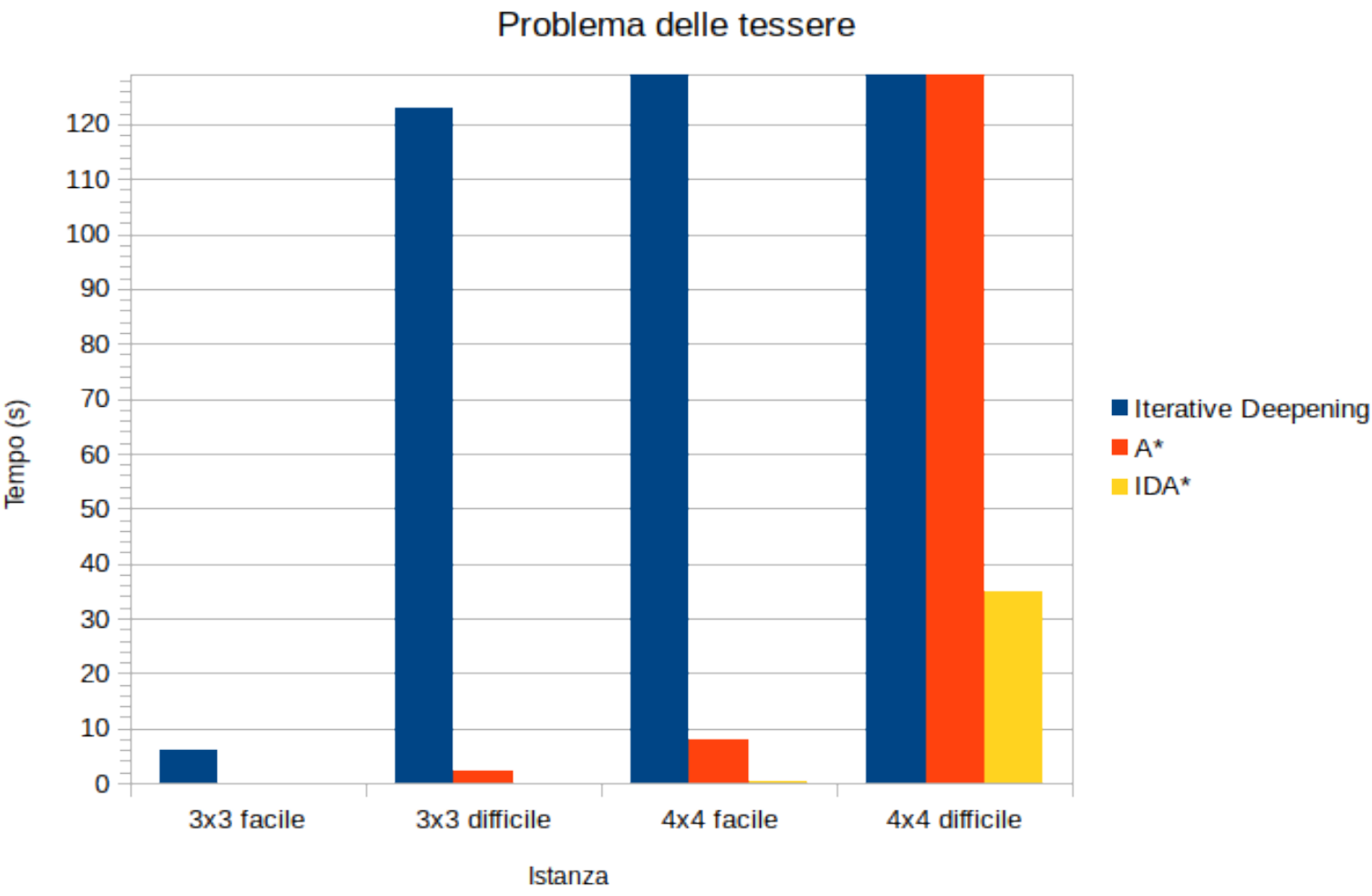
- `applicabile(Azione, Posizione)`: determina la validità di Azione in Posizione;
- `trasforma(Azione, Posizione, NuovaPosizione)`: applica Azione a Posizione;
- euristica(Posizione, StimaEuristica): calcola la stima euristica da Posizione alla posizione finale;
- `costoSpostamento(PosizionePartenza, PosizioneArrivo, Costo)`: calcola il costo di spostamento da PosizionePartenza a PosizioneArrivo.

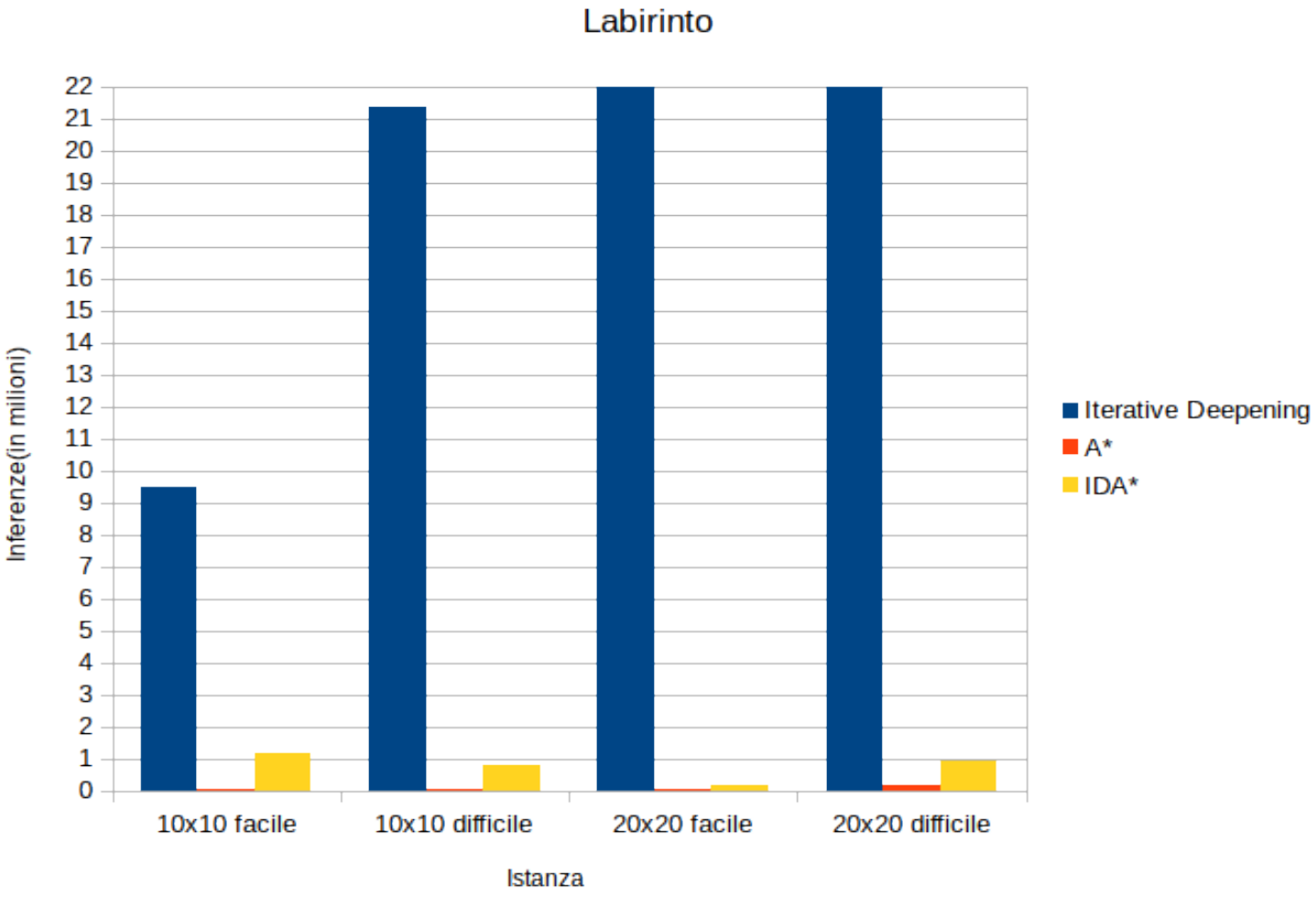
In tutti i test come funzione euristica è stata utilizzata la *Distanza di Manhattan* ed il costo di spostamento, data la tipologia di domini, è stato considerato unitario.

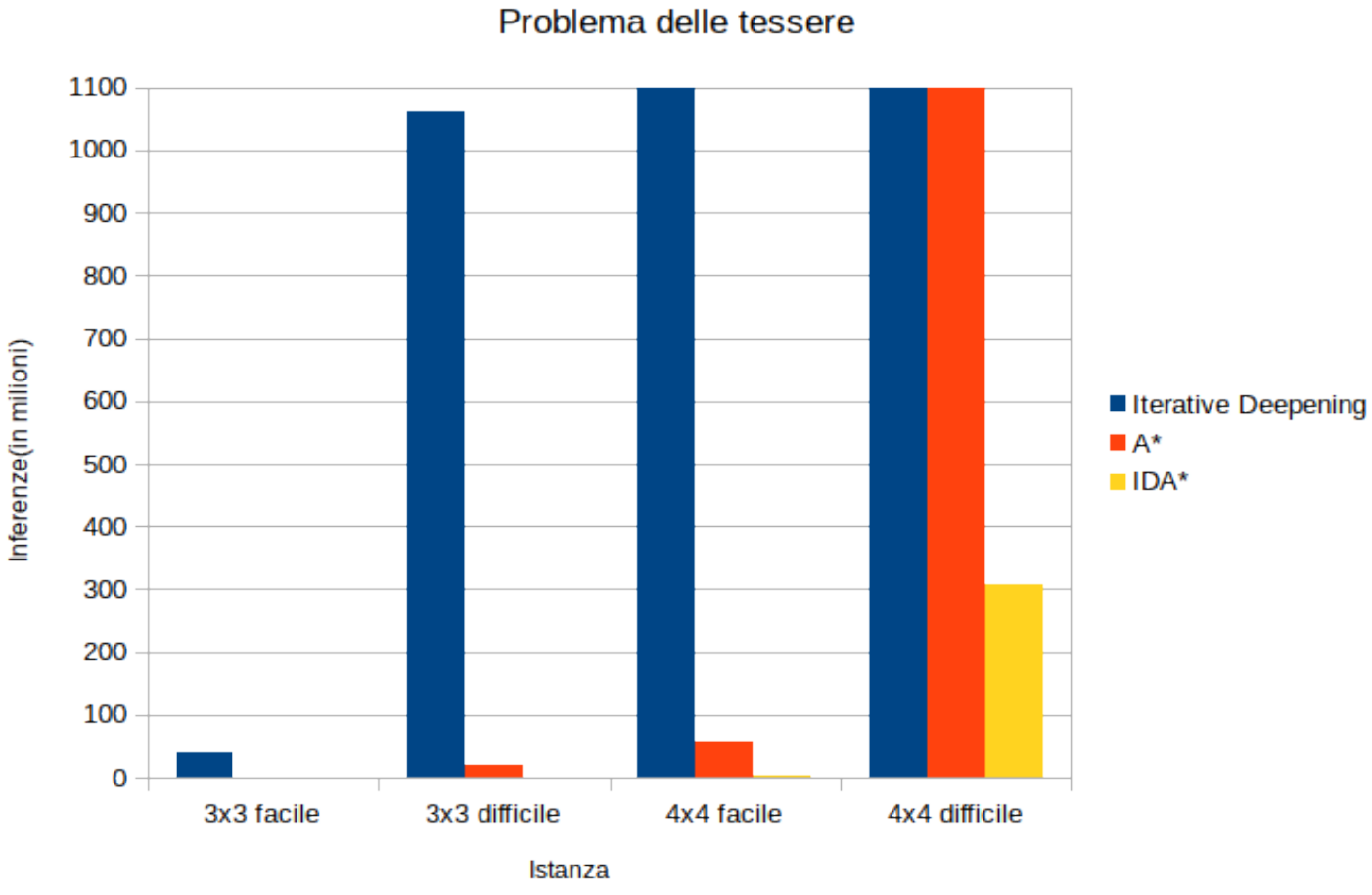
Tutte le prove effettuate hanno confermato una netta superiorità di *IDA** rispetto agli altri algoritmi, in tutti i possibili scenari. Solo *A** riesce parzialmente a competere con esso, non riuscendo tuttavia a risolvere in tempi ragionevoli il gioco del 15 partendo dalla posizione difficile. Si dimostra invece una completa incapacità di *iterative deepening* di risolvere qualsiasi problema che sia poco più che basilare: i suoi tempi di esecuzione salgono troppo rapidamente ed il numero di inferenze eseguite anche. Il caso più evidente di questo è sicuramente il test del 3x3 difficile: *iterative deepening* arriva a toccare le 1.061.310.227 inferenze in 122,934 secondi, contro le 18.345.153 in 2,477 si *A** e le 623.902 in 0,076 di *IDA**. Nota a margine, *A** riesce a comportarsi meglio di *IDA** fintantoché lo scenario rimane semplice, questo per via della sua logica più semplice ed intuitiva, ma appena la complessità aumenta non è più neanche paragonabile.

Seguono, nelle prossime pagine, grafici e tabelle riassuntive dei test condotti.


Labirinto
Tempo(s)
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
10x10 facile
10x10 difficile
20x20 facile
20x20 difficile
Istanza
Iterative Deepening
A*
IDA*


Problema delle tessere
Tempo (s)
120
110
100
90
80
70
60
50
40
30
20
10
0
3x3 facile
3x3 difficile
4x4 facile
4x4 difficile
Istanza
Iterative Deepening
A*
IDA*


Labirinto
Inferenze(in milioni)
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
10x10 facile
10x10 difficile
20x20 facile
20x20 difficile
Istanza
Iterative Deepening
A*
IDA*


Problema delle tessere
Inferenze(in milioni)
1100
1000
900
800
700
600
500
400
300
200
100
0
3x3 facile
3x3 difficile
4x4 facile
4x4 difficile
Istanza
Iterative Deepening
A*
IDA*

	Labirinto			
Strategia	10x10 facile	10x10 difficile	20x20 facile	20x20 difficile
Iterative Deepening	1.085	2.721	16.721	—
A*	0.014	0.007	0.009	0.026
IDA*	0.129	0.088	0.028	0.104

Tabella 1: Tempi di esecuzione (espressi in secondi)

	Tessere			
Strategia	3x3 facile	3x3 difficile	4x4 facile	4x4 difficile
Iterative Deepening	6.152	122.934	—	—
A*	0.137	2.477	8.040	—
IDA*	0.017	0.076	0.237	34.913

Tabella 2: Tempi di esecuzione (espressi in secondi)

	Labirinto			
Strategia	10x10 facile	10x10 difficile	20x20 facile	20x20 difficile
Iterative Deepening	9 449 000	21 322 628	141 789 264	—
A*	31 283	56 044	27 841	135 345
IDA*	1 189 329	798 983	192 960	902 824

Tabella 3: Inferenze eseguite

	Tessere			
Strategia	3x3 facile	3x3 difficile	4x4 facile	4x4 difficile
Iterative Deepening	39 220 382	1 061 310 227	—	—
A*	682 069	18 345 153	54 955 687	—
IDA*	138 358	623 902	1 876 229	306 883 838

Tabella 4: Inferenze eseguite

Constraint Satisfaction Problem con ASP (Answer Set Programming)

Calendario delle lezioni

Consegna

Formulare e risolvere il seguente problema di soddisfacimento di vincoli: creazione di un **calendario settimanale di lezioni** universitarie con allocazione nelle aule. Si ipotizzi la presenza di:

- studenti appartenenti ad almeno tre anni differenti
- almeno 3 corsi per ciascun gruppo (anno) di studenti
- almeno 3 docenti che svolgono, ciascuno, almeno 2 corsi
- un numero adeguato di aule per lo svolgimento delle lezioni, da considerare il parametro principale per la valutazione dei risultati ottenuti.

Soluzione

Il file `lesson_csp.cl` contiene la soluzione del problema di soddisfazione di vincoli.

Scelte implementative

Costanti

– `num_classrooms`: numero massimo di aule, definito in base al numero di professori;

– `num_years`: numero di anni in cui i corsi sono suddivisi;

– `num_courses_for_year`: numero di corsi per ogni anno;

– `num_prof`: numero di professori;

– `num_courses_for_prof`: numero di corsi per ogni professore;

– `days`: numero di giorni settimanali in cui si svolgono le lezioni;

– `lessons_per_day`: numero di lezioni al giorno;

– `num_lessons_per_course`: numero di lezioni settimanali per ogni corso

– `num_slot_ids`: numero di slot temporali disponibili per le lezioni settimanali;

– `num_years`: numero di anni in cui i corsi sono suddivisi.

Predicati

– `num_max_lessons_per_slot(Max)`: definisce il numero massimo (`Max`) di lezioni che possono avere luogo nello stesso slot temporale, cioè contemporaneamente;

– `num_max_lessons_per_slot(Max)`: definisce il numero massimo (`Max`) di lezioni che possono avere luogo nello stesso slot temporale, cioè contemporaneamente;

– `classroom_id(Id)`: `Id` è l'identificativo di un'aula;

– `course(Id,Year)`: assegnazione di un corso (`Id`) a un anno (`Year`);

– `teaching(IdCourse,IdProf)`: assegnazione di un corso (`IdCourse`) a un professore (`IdProf`);

– `location(Classroom,Slot)`: definizione di una locazione data da un aula (`Classroom`) e da uno slot temporale (`Slot`);

– `lesson(CourseId,location(Classroom,Slot))`: assegnazione di una locazione (`location(Classroom,Slot)`) a un corso (`CourseId`);

Vincoli

– Il numero di aule non deve essere inferiore a 1 e non deve essere superiore a `num_max_classrooms`.

– Devono essere rispettati i limiti minimi di corsi per ogni anno, di corsi per ogni professore e di lezioni settimanali per ogni corso.

– Ogni corso può essere assegnato a uno e un solo anno.

– Ogni corso può essere assegnato a uno e un solo professore.

– Una lezione di corso non può avere luogo in due aule diverse durante stesso slot temporale.

– Due lezioni di due corsi diversi non possono avere luogo nella stessa aula durante lo stesso slot temporale.

– Un professore non può tenere due corsi diversi durante lo stesso slot temporale.

– Il numero dei corsi non può essere inferiore al numero di professori per il numero di corsi per professore.

– Non è possibile che il numero totale di lezioni che devono essere svolte superi il numero di lezioni che possono essere svolte nello stesso slot temporale per gli slot di tempo disponibili, quindi, cioè che

```
num_max_lessons_per_slot(Max_per_slot),
    Max_per_slot*num_slot_ids <
num_courses_for_year*num_years*num_lessons_per_course.
```

Risultati

I seguenti sono i valori assegnati alle costanti.

Costanti	Valore
# anni	12
# corsi per anno	3
# corsi	12
# professori	6
# corsi per professore	2
# lezioni per corso	3
# giorni di lezione	5
# lezioni al giorno	5
# slot per le lezioni	25

Tabella 5: Valori delle costanti

Risultato con 2 aule: il primo risultato presentato è quello con **2 aule** a disposizione per le lezioni. Seguono le tabelle che rappresentano le conseguenti assegnazioni dei corsi agli anni, dei corsi ai professori, il calendario settimanale e i tempi di esecuzione.

Anno	Corsi		
1	2	5	6
2	4	9	11
3	3	7	10
4	1	8	12

Tabella 6: Assegnazione dei corsi agli anni

Professori	Corsi
1	1 4
2	2 10
3	5 11
4	6 8
5	3 7
6	9 12

Tabella 7: Assegnazione dei corsi ai professori

	Giorno 1	**Giorno 2**	**Giorno 3**	**Giorno 4**	**Giorno 5**
Lezione 1	P C A 6 9 1 1 1 2	P C A 3 11 1 2 10 2	P C A 2 10 1 4 8 2	P C A 6 9 1 ∅ ∅ 2	P C A 1 4 1 4 8 2
Lezione 2	P C A 3 5 1 6 12 2	P C A 3 11 1 1 4 2	P C A 3 5 1 4 6 2	P C A ∅ ∅ 1 ∅ ∅ 2	P C A 1 4 1 4 6 2
Lezione 3	P C A 1 1 1 5 7 2	P C A 4 8 1 ∅ ∅ 2	P C A 2 2 1 5 7 2	P C A 6 12 1 5 3 2	P C A ∅ ∅ 1 ∅ ∅ 2
Lezione 4	P C A 5 7 1 6 9 2	P C A ∅ ∅ 1 ∅ ∅ 2	P C A 2 2 1 4 6 2	P C A 3 5 1 5 3 2	P C A ∅ ∅ 1 ∅ ∅ 2
Lezione 5	P C A ∅ ∅ 1 ∅ ∅ 2	P C A 2 10 1 6 12 2	P C A 2 2 1 ∅ ∅ 2	P C A 3 11 1 5 3 2	P C A 1 1 1 ∅ ∅ 2

Tabella 8: Calendario settimanale

Time	0.154
CPU Time	0.087

Tabella 9: Prestazioni (in secondi)

Risultato con 1 aula: per l'implementazione che è stata scelta deve valere

$$\#\text{slot per le lezioni} \cdot \#\text{aule} \geq \#\text{lezioni per corso} \cdot \#\text{corsi}$$

altrimenti il problema non è soddisfacibile. Quindi con solo **1 aula** a disposizione ci si aspetta che il problema non sia soddisfacibile, dal momento che il numero totale di lezioni

$$\#\text{slot per le lezioni} \cdot \#\text{aule} = 25 \cdot 1 = 25$$

mentre

$$\#\text{lezioni per corso} \cdot \#\text{corsi} = 3 \cdot 12 = 36.$$

Risulta infatti che non esista nessun **Answer Set** che soddisfi tutti i vincoli. Le seguenti sono le prestazioni.

Time	0.015
CPU Time	0.017

Tabella 10: Prestazioni (in secondi)

É stato eseguito un test senza il precedente vincolo ed è stato interrotto dopo 2904.860 secondi (circa 48 minuti) di esecuzione, ne consegue che la scelta di inserire tale vincolo sia la migliore.

Risultato con più di 2 aule: dal momento che il problema è soddisfacibile con 2 aule ci si aspetta che lo sia anche con un numero maggiore di aule. Tutti i test effettuati con più di 2 aule hanno avuto successo con diverse prestazioni, le seguenti tabelle presentano alcuni esempi.

3 aule	
Time	0.200
CPU Time	0.200

6 aule	
Time	0.870
CPU Time	0.870

10 aule	
Time	2.541
CPU Time	2.540

Tabella 11: Prestazioni (in secondi)

Pianificazione: problema del trasporto

Il secondo esercizio da risolvere attraverso il paradigma ASP è il *problema del trasporto*: si tratta del dominio del trasporto aereo di merci descritto nel **Cap.10.1** del *Russell e Norvig*. È richiesto un'applicazione a problemi di varie dimensioni.

$$
\begin{aligned}
&Init(At(C_1,\ SFO) \land At(C_2,\ JFK) \land At(P_1,\ SFO) \land At(P_2,\ JFK)\\
&\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)\\
&\quad \land Airport(JFK) \land Airport(SFO))\\
&Goal(At(C_1,\ JFK) \land At(C_2,\ SFO))\\
&Action(Load(c,\ p,\ a),\\
&\quad \text{PRECOND: } At(c,\ a) \land At(p,\ a) \land Cargo(c) \land Plane(p) \land Airport(a)\\
&\quad \text{EFFECT: } \neg\, At(c,\ a) \land In(c,\ p))\\
&Action(Unload(c,\ p,\ a),\\
&\quad \text{PRECOND: } In(c,\ p) \land At(p,\ a) \land Cargo(c) \land Plane(p) \land Airport(a)\\
&\quad \text{EFFECT: } At(c,\ a) \land \neg\, In(c,\ p))\\
&Action(Fly(p,\ from,\ to),\\
&\quad \text{PRECOND: } At(p,\ from) \land Plane(p) \land Airport(from) \land Airport(to)\\
&\quad \text{EFFECT: } \neg\, At(p,\ from) \land At(p,\ to))
\end{aligned}
$$

Figura 1: Descrizione PDDL di un problema di pianificazione del trasporto di merci presente nel **Cap.10.1** del *Russell e Norvig*.

Descrizione generale del problema

- **Dominio**, definito tramite assiomi
  - Un insieme $C = \{c_i \mid \texttt{Cargo}(c_i)\}$ di merci da trasportare, con $i = 1 \dots m$, dove $|C| = m$.
  - Un insieme $P = \{p_j \mid \texttt{Plane}(p_j)\}$ di aerei, con $j = 1 \dots n$, dove $|P| = n$.
  - Un insieme $A = \{a_k \mid \texttt{Airport}(a_k)\}$ di aeroporti, con $k = 1 \dots l$, dove $|A| = l$.
- **Fluenti**
  - $At_{plane} = \{\texttt{at}(\mathrm{p}_i, \mathrm{a}_k) \mid 1 \leq i \leq n, 1 \leq k \leq l\}$, dove p_i è l'aereo che si trova nell'areoporto a_k
  - $At_{cargo} = \{\texttt{at}(\mathrm{c}_i, \mathrm{a}_k) \mid 1 \leq i \leq m, 1 \leq k \leq l\}$, dove c_i è la merce che si trova nell'areoporto a_k

- $In = \{\texttt{in(c}_i\texttt{,p}_k\texttt{)} \mid 1 \leq i \leq m, 1 \leq k \leq n\}$, dove $\texttt{c}_i$ è la merce che si trova nell'aereo $\texttt{p}_k$

- **Azioni**

  - $\texttt{fly(p}_i\texttt{,a}_{from}\texttt{,a}_{to}\texttt{)}$, dove $\texttt{p}_i$ è l'aereo che vola dall' aeroporto $\texttt{a}_{from}$ all'areoporto $\texttt{a}_{to}$.

    * PRE: $\texttt{at(p}_i\texttt{,a}_{from}\texttt{) \& Plane(p}_i\texttt{) \& Airport(a}_{from}\texttt{)}$ $\texttt{\& Airport(a}_{to}\texttt{)}$
    * POST: $\texttt{not at(p}_i\texttt{,a}_{from}\texttt{) \& at(p}_i\texttt{,a}_{to}\texttt{)}$

  - $\texttt{load(c}_i\texttt{,p}_j\texttt{,a}_k\texttt{)}$, dove $\texttt{c}_i$ è la merce che viene caricata sull'aereo $\texttt{p}_j$ nell'areoporto $\texttt{a}_k$.

    * PRE: $\texttt{at(p}_j\texttt{,a}_k\texttt{) \& at(c}_i\texttt{,a}_k\texttt{) \& Cargo(c}_i\texttt{) \& Plane(a}_j\texttt{)}$ $\texttt{\& Airport(a}_k\texttt{)}$
    * POST: $\texttt{not at(c}_i\texttt{,a}_k\texttt{) \& in(c}_i\texttt{,p}_j\texttt{)}$

  - $\texttt{unload(c}_i\texttt{,p}_j\texttt{,a}_k\texttt{)}$, dove $\texttt{c}_i$ è la merce che viene scaricata dall'aereo $\texttt{p}_j$ nell'areoporto $\texttt{a}_k$.

    * PRE: $\texttt{at(p}_j\texttt{,a}_k\texttt{) \& in(c}_i\texttt{,p}_j\texttt{) \& Cargo(c}_i\texttt{) \& Plane(a}_j\texttt{)}$ $\texttt{\& Airport(a}_k\texttt{)}$
    * POST: $\texttt{at(c}_i\texttt{,a}_k\texttt{) \& not in(c}_i\texttt{,p}_j\texttt{)}$

- **Goal**

  - $G \subseteq At_{cargo}$ è l'insieme dei fluenti che devono essere soddisfatti al fine di risolvere il problema di pianificazione. *(Si noti che non è stato usato il simbolo di inclusione stretta, poichè è possibile definire un insieme di goal contenente tutti i possibili fluenti in At_{cargo}, naturalmente, a meno che vengano definiti domini triviali, non è possibile trovare una soluzione.)*

Implementazioni

Si è scelto di risolvere il problema del trasporto in due modi diversi: il primo, `cargoSequential.cl`, prevede la possibilità di eseguire, per ogni istante di tempo, una sola azione:

```
1{occurs(A,S): action(A)}1:- level(S).
```

Questo vincolo implica che deve essere eseguita **almeno** e **al più** un'azione `A` nello stato `S` . Quindi, ad esempio, solo un aereo alla volta può caricare le merci al proprio interno, scaricarle o effettuare un volo tra due aeroporti.

Naturalmente si tratta di un approccio poco realistico e non efficiente al problema, dal momento che ci si aspetterebbe che gli aerei siano indipendenti tra loro. Per questo motivo una seconda implementazione, `cargoParallel.cl` prevede la possibilità di effettuare più azioni nello istante di tempo

```
1{occurs(A,S): action(A)}:- level(S).
```

L'assenza di un vincolo superiore significa che è possibile eseguire nello stesso stato `S` un numero arbitrario di azioni.

Essendo i vincoli più rilevanti nella risoluzione del problema, si mette in evidenza che in entrambe le implementazioni:

- *L'aereo può eseguire una sola azione alla volta*: non può caricare/scaricare due merci diverse nello stesso istante e naturalmente non può volare in una certa direzione e contemporaneamente effettuare altre azioni.
- *Capienza illimitata*: il numero di merci che è possibile caricare su un aereo non è limitato. Si è scelto questo approccio perchè si tratta di un semplice rilassamento di vincoli rispetto al problema con *capienza limitata*, da cui, quindi, non si discosta molto, pur potendo essere definito in modo più semplice.
- *Eliminazione di azioni useless*: dal momento che l'azione

  ```
  fly(Plane,From,To)
  ```

  non viola nessun vincolo quando `From=To`, ma si tratta di un'*azione useless* (non è una *no-op* perchè `Plane` effettua un volo), `fly` è stata definita come segue

  ```
  action(fly(Plane,From,To)) :- plane(Plane), airport(From),
         airport(To), From!=To.
  ```

  imponendo come nuovo vincolo che non si possa mai effettuare un volo da e verso lo stesso areoporto.

Il tempo

Si pone un limite al tempo di esecuzione del piano attraverso una congiunzione di vincoli relativi ai fluenti:

```
holds(Fluent,lastLev)
```

dove `Fluent` è il fluente che si vuole rendere valido entro e non oltre il tempo `lastLev`. Inoltre gli effetti di ogni azione eseguita nel tempo `S` sono validi nel tempo `S+1`.

Cargo Sequential

Oltre alla definizione attraverso ASP delle precondizioni e postcondizioni è stato necessario definire due *vincoli di persistenza*

```
holds(F,S+1):-holds(F,S),fluent(F),level(S),not -holds(F,S+1).
```

Questo vincolo implica che se al tempo S è il fluente F è valido, e non è possibile dimostrare che al tempo successivo (S+1) F non sia più valido (not -holds(F,S+1)), allora si può affermare che la validità di F persista anche al tempo S+1.

Inoltre deve essere definito

```
-holds(F,S+1):- -holds(F,S),fluent(F),level(S),not holds(F,S+1).
```

Questo secondo vincolo implica che, se al tempo S, il fluente F non è valido, e non è possibile dimostrare che al tempo successivo (S+1) F sia valido (not holds(F,S+1)), allora si può affermare che la non validità di F persista anche al tempo S+1.

Sono inoltre stati definiti dei vincoli globali tali che al tempo S non sia possibile che:

1. lo stesso aereo si trovi in due areoporti diversi.

```
:-  holds(at(Plane,Airport1),S),
holds(at(Plane,Airport2),S),
Airport1!=Airport2.
```

2. la stessa merce si trovi in due areoporti diversi.

```
:-  holds(at(Cargo,Airport1),S),
holds(at(Cargo,Airport2),S),
Airport1!=Airport2.
```

3. due aerei diversi contengano la stessa merce.

```
:-  holds(in(Cargo,Plane1),S),
holds(in(Cargo,Plane2),S),
Plane1!=Plane2.
```

Si è optato per questo approccio in modo che si possa considerare come fallimento sia l'errata definizione dei fluenti validi al tempo 0, ma anche la definizione inconsistente del goal. Per esempio, se si pone che al tempo 0 un aereo si trovi in due areoporti diversi, il fallimento è immediato, infatti nessuna soluzione è da prendere in considerazione. Vale lo stesso nel caso si ponga come goal che, ad esempio, una stessa merce si trovi in due areoporti diversi.

Cargo Parallel

Dal momento che il numero di azioni eseguibili in un singolo stato non è limitato a 1, è necessario definire vincoli globali in modo che tale rilassamento del problema non crei inconsistenze. Nonostante sia intuitivo pensare di rispettare tale regola tramite il seguente vincolo:

```
:- occurs(Action1,S), occurs(Action2,S), Action1!=Action2.
```

In realtà questo vincolo elimina il rilassamento voluto, dal momento che non è possibile eseguire più di un azione alla volta, anche rispetto ad aerei diversi. Quindi è necessario definire i vincoli globali facendo riferimento in modo specifico agli aerei e alle merci: non è possibile che al tempo `S`

1. lo stesso aereo carichi/scarichi due merci diverse.

```
:-  occurs(load/unload(Cargo1,Plane,_),S),
occurs(load/unload(Cargo2,Plane,_),S),
Cargo1!=Cargo2.
```

2. lo stesso aereo effettui un qualche volo e carichi una qualche merce.

```
:-  occurs(fly(Plane,_,_),S),
occurs(load(_,Plane,_),S).
```

3. lo stesso aereo effettui un qualche volo e scarichi una qualche merce.

```
:-  occurs(fly(Plane,_,_),S),
occurs(unload(_,Plane,_),S).
```

4. lo stesso aereo scarichi una qualche merce e carichi una qualche merce.

```
:-  occurs(unload(_,Plane,_),S),
occurs(load(_,Plane,_),S).
```

Si è potuto omettere alcuni elementi sfruttando le proprietà del carattere "_", che ha il significato sintattico di poter essere sostituito con qualsiasi elemento che rispetti il pattern matching: non è necessario in questo caso rendere espliciti i vincoli relativi alle precondizioni e postcondizioni, poichè devono essere già rispettati.

Nonostante le regole 1, 2 e 3, nel caso *parallelo* possano essere equivalentemente sostituite da regole che considerino le azioni e non i fluenti, si è preferito mantenere lo stesso approccio del caso precedente.

Soluzione e fallimento

Per imporre che sia soddisfatta la validità di un insieme di fluenti si definisce, innanzitutto, la regola

$$\texttt{goal :- holds(F}_1\texttt{,S}_1\texttt{), ..., holds(F}_n\texttt{,S}_n\texttt{)}$$

Dove F_i ed S_i sono rispettivamente il fluente e il tempo dell'i-esimo sottogoal.

Poi si pone il vincolo

$$\texttt{:- not goal}$$

Se non è possibile rendere valido `goal` in almeno un modello il problema non è risolvibile.

Tutte le possibili soluzioni al problema sono date dai possibili modelli che rendono valido `goal`.

Esempi

Sono presi in considerazione 6 possibili scenari, con 3 tipi di dominio diversi (*piccolo, medio, grande*), e 2 difficoltà diverse per ogni dominio (*facile, difficile*).

- **Dominio**

	$\lvert P\rvert$	$\lvert C\rvert$	$\lvert A\rvert$
Piccolo	1	2	3
Medio	2	4	5
Grande	2	8	9

- **Difficoltà** *Tutte* le merci si trovano in un areoporto diverso da quello di destinazione. Questo perchè altrimenti si avrebbero alcune configurazioni triviali per alcune merci: la soluzione al sottogoal è una lista vuota di azioni.
  - Facile
    1. Ogni aereo si trova al tempo `0` in un areoporto con almeno una merce.
    2. Più merci possono trovarsi nello stesso areoporto al tempo `0`.
    3. Le merci hanno la stessa destinazione.

    (il punto 2 è definito in questo modo perchè non si è voluto presentare problemi banali, come quello in cui tutte le merci si trovano nello stesso areoporto e hanno la stessa destinazione)
  - Difficile
    1. Ogni aereo si trova al tempo `0` in un areoporto senza merci.

2. Tutte le merci si trovano in areoporti diversi al tempo `0`.
3. Tutte le merci hanno una destinazione diversa.

Le seguenti tabelle presentano il numero minimo di passi per ogni configurazione, rispettivamente, per la soluzione sequenziale (un'azione alla volta) e per quella parallela (più azioni alla volta per aerei diversi). Come si può notare per il dominio piccolo, essendo presente un solo aereo, le due soluzioni sono equivalenti.

	Numero minimo di passi					
	Piccolo		Medio		Grande	
Strategia	Facile	Difficile	Facile	Difficile	Facile	Difficile
Sequenziale	5	6	10	12	20	—
Parallelo	5	6	5	6	10	12

	Tempo di esecuzione					
	Piccolo		Medio		Grande	
Strategia	Facile	Difficile	Facile	Difficile	Facile	Difficile
Sequenziale	0.008	0.008	0.081	0.081	2428.557	—
Parallelo	0.008	0.008	0.023	0.022	9.486	21.396

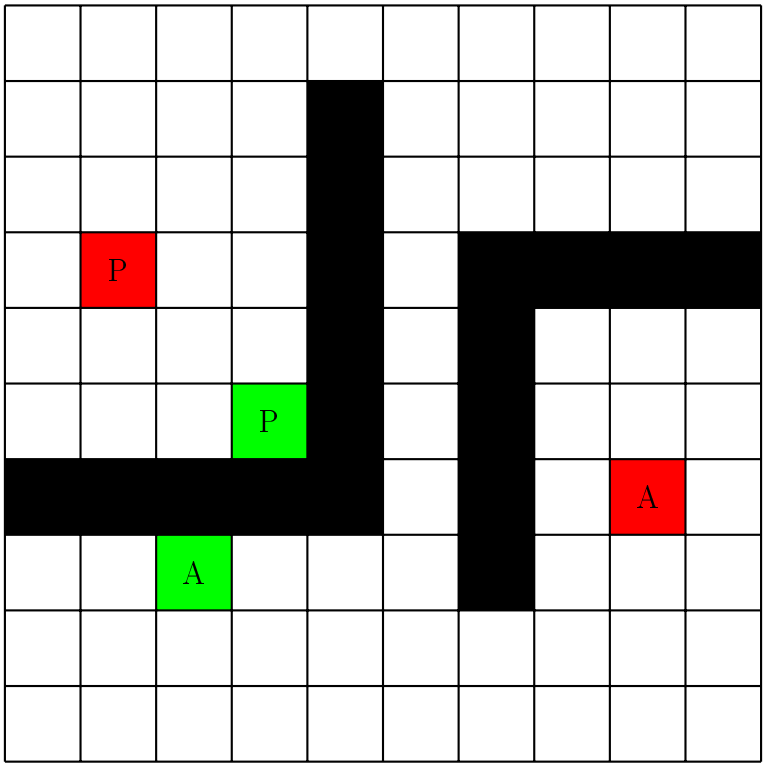
Appendice

Scenari dei test degli algoritmi Prolog

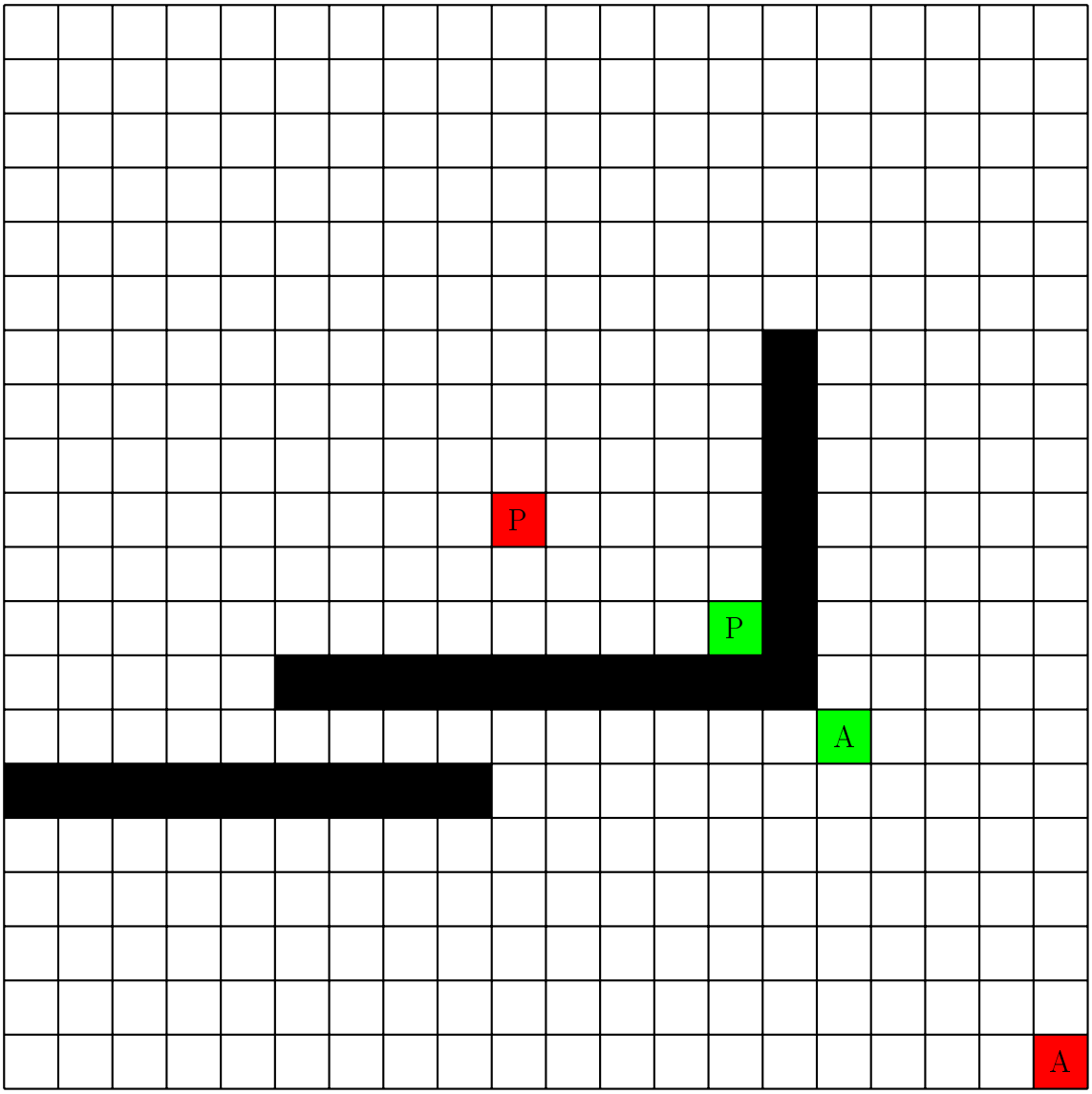
Labirinto

In rosso sono indicate le posizioni difficili, in verde quelle facili; P e A indicano le caselle di partenza ed arrivo.

Labirinto 10x10



Labirinto 20x20



Gioco delle tessere

Gioco dell'8 facile

3	4	1
6		8
2	7	5

Gioco dell'8 difficile

	3	7
6	8	1
2	5	4

Gioco dell'15 facile

7	2	1	8
5	10	4	12
6	3		15
9	13	14	11

Gioco dell'15 difficile

15	5	3	11
6	8	2	1
9	12		4
13	7	10	14

Configurazioni problema del trasporto

Dominio Piccolo

Facile

START

A1	A2	A3
C1 P1	**C2**	

GOAL

A1	A2	A3
C2	**C1**	

Difficile

START

A1	A2	A3
C1	**C2**	P1

GOAL

A1	A2	A3
C2	**C1**	

Dominio Medio

GOAL

A1	A2	A3	A4	A5
C2	**C1**			

GOAL

A1	A2	A3	A4	A5
C2	**C1**			