**Università degli Studi di Torino**

Dipartimento di Informatica

Corsi di Laurea Magistrale in Informatica



**Relazione del progetto di Intelligenza**

**Artificiale e Laboratorio**

Salvatore Coluccia

a.a. 2020/2021

Indice

[1. Prolog 3](#_Toc34327020)

[1.1 Iterative Deepening 3](#_Toc34327021)

[1.2 IDA\* 4](#_Toc34327022)

[1.3 A\* 4](#_Toc34327023)

[1.4 Benchmark e considerazioni 5](#_Toc34327024)

[1.4.1 Labirinto 5](#_Toc34327025)

[1.4.2 Considerazioni 7](#_Toc34327026)

[2.ASP 8](#_Toc34327027)

[2.1 Calendario torneo sportivo 8](#_Toc34327028)

[2.2 Trasporto Aereo 9](#_Toc34327029)

# Prolog

Per la parte del progetto relativa al linguaggio Prolog ho implementato 3 algoritmi di ricerca: Iterative Deepening, A\* e IDA\*.

Iterative Deepening appartiene alla famiglia degli algoritmi di ricerca non informata e quindi non utilizza alcun tipo di conoscenza del problema al di là della definizione del problema stesso.

A\* e IDA\* appartengono invece alla famiglia degli algoritmi di ricerca informata che quindi utilizzano una conoscenza aggiuntiva (in questo caso un’euristica) in modo da espandere prima i nodi che soddisfano determinate condizioni.  
La funzione euristica utilizzata è stata la distanza di Manhattan (*L1(P1,P2) = |x1-x2|+|y1-y2|*) perché soddisfa le condizioni di **ammissibilità** e **consistenza**.  
Un’euristica è ammissibile se non sbaglia mai per eccesso la stima del costo per arrivare all’obiettivo. Un’euristica è consistente (o monotona) se, per ogni nodo *n* e ogni successore *n0* di *n*, il costo stimato per raggiungere l’obiettivo partendo da *n* non è superiore al costo di passo per arrivare a *n0* sommato al costo stimato per andare da lì all’obiettivo.



***Figura 1.1:*** *La linea verde rappresenta la distanza euclidea mentre le altre indicano alcune possibili rappresentazioni geometriche della distanza di Manhattan (tutte equivalenti).*

Il dominio utilizzato per testare questi algoritmi è quello del ***labirinto***.

## Iterative Deepening

L’algoritmo di Iterative Deepening si basa sull’algoritmo di ricerca in profondità limitata. L’idea sulla quale si basa è quella di eseguire iterativamente l’algoritmo di ricerca in profondità limitata incrementando il limite ad ogni iterazione fino a trovare la soluzione (se esiste).

La ricerca a profondità limitata permette di non espandere rami infiniti che non portano ad alcuna soluzione in quanto al raggiungimento del limite imposto l’algoritmo termina di espandere altri nodi.

L’algoritmo di Iterative deepening è un algoritmo completo ed ottimo in quanto trova sempre una soluzione se esiste e la soluzione trovata è quella ottima.

Nel caso in cui non esista alcuna soluzione non è in grado di accorgersene perché aumenterebbe all’infinito il limite di profondità.

Nell’implementazione si è deciso di tenere traccia dei nodi già espansi per evitare di ripercorrere rami già percorsi. Questo aumenta lievemente l’ammontare della memoria necessaria durante l’esecuzione dell’algoritmo ma riduce il tempo necessario ad eseguire l’intero algoritmo.

## IDA\*

IDA\* è un algoritmo di ricerca informata che implementa una variante dell’algoritmo di iterative deepening ma utilizzando una funzione euristica per scegliere la profondità massima dell’iterazione corrente:

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

*g(n) = costo effettivo dal nodo iniziale al nodo n*

*h(n) = costo stimato dal nodo n al nodo finale. Ricordo che la stima è stata effettuata utilizzando la distanza di Manatthan*

si sceglie quindi il min(f(n)) tra tutti i nodi che hanno superato la soglia imposta nell’iterazione precedente.

Nell’implementazione attuata si tiene traccia dei nodi già visitati in modo da evitare di ripercorrere gli stessi path e quindi in modo da ottimizzare i tempi di esecuzione.

Una caratteristica di IDA\* è quella di richiedere meno memoria rispetto all’algoritmo A\* in quanto deve ricordare esclusivamente i nodi del path corrente (oltre a quelli già visitati nella specifica iterazione).

Nel caso del dominio del labirinto si è scelto di assegnare dei costi diversi ai movimenti tra le varie caselle. È stato definito un costo di default pari a 1 per tutti i movimenti che però cambia per i movimenti di specifiche caselle, questo va ad influenzare il valore che assume la funzione g e rende meno vantaggiose le soluzioni che passano da quelle caselle.

## A\*

A\* è un algoritmo di ricerca informata di tipo best-first e cioè che espande sempre il nodo più “vantaggioso” localmente.

Il termine “vantaggioso” è definito in base al valore della funzione euristica f(n) = g(n) + h(n) che è implementata come per l’algoritmo IDA\*.

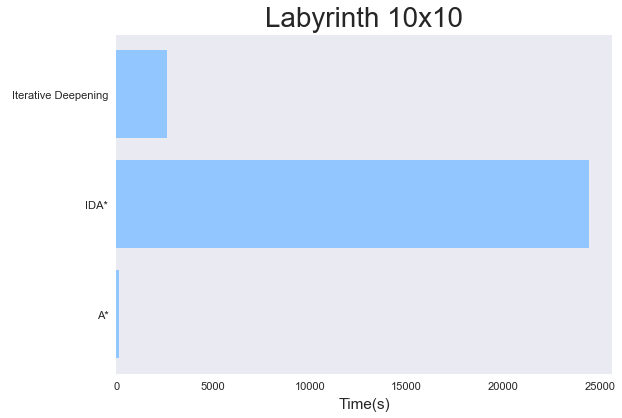
L’algoritmo A\* è stato implementato utilizzando una coda con priorità dove ad ogni nodo è assegnata la rispettiva priorità f(n). Ad ogni iterazione viene quindi espanso dalla coda il nodo con priorità più bassa fino a quando non viene soddisfatto lo stato goal.

A\*, rispetto a IDA\*, richiede molta più memoria in quanto deve ricordare tutti i nodi della coda più quelli già esplorati (che non devono essere riespansi).

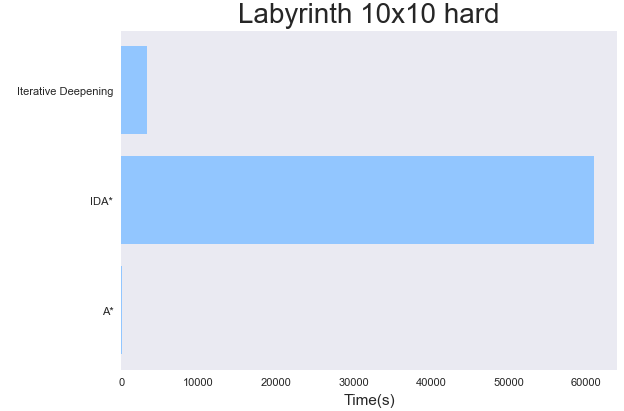
A\* è un algoritmo completo per domini finiti in quanto prima o poi la coda diventerà vuota e l’algoritmo terminerà negativamente

## Benchmark e considerazioni

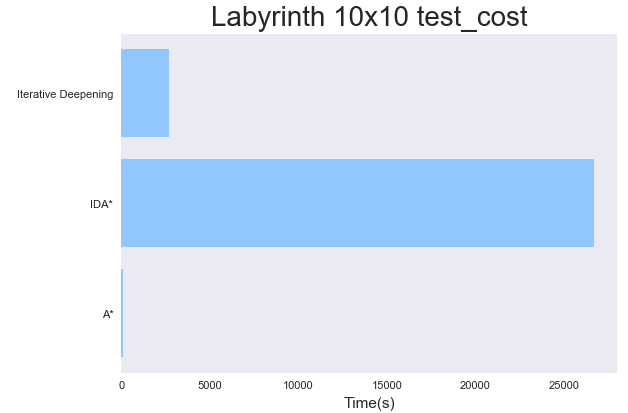
### 1.4.1 Labirinto

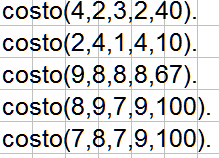


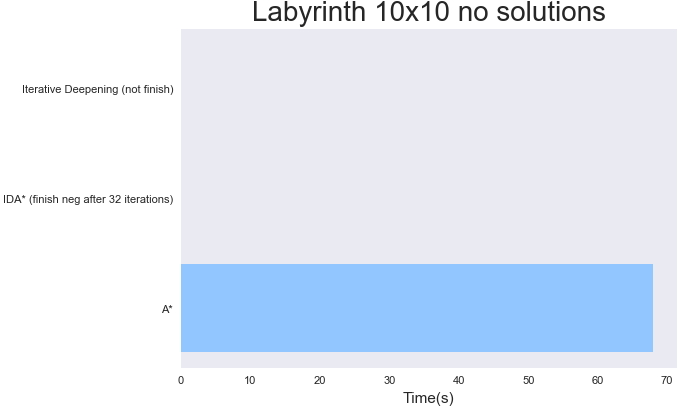




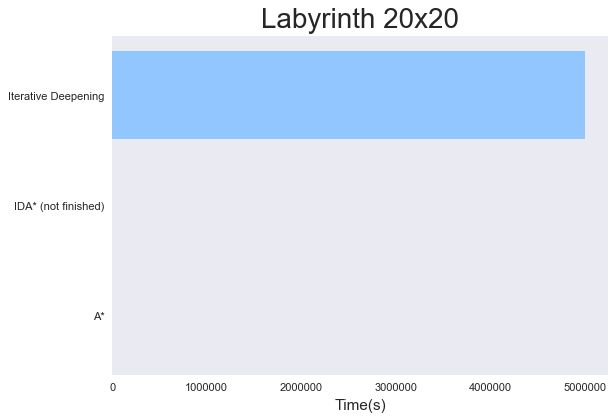


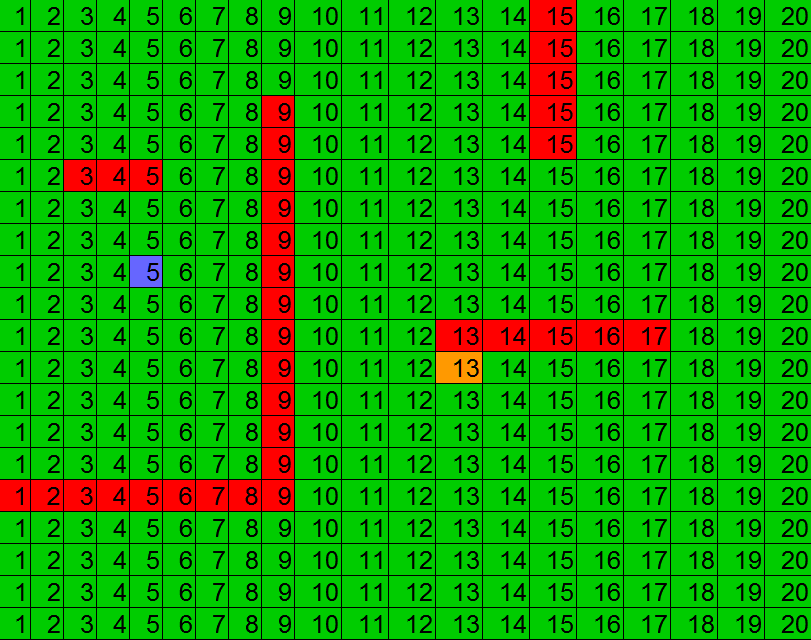


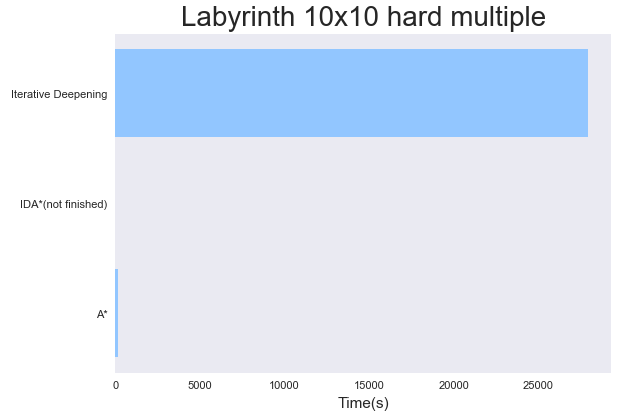














### 1.4.2 Considerazioni

Dai numerosi test condotti possiamo notare che in questo dominio non esiste un solo algoritmo che prevale sugli altri in termini di tempo di esecuzione.

Possiamo constatare come l’algoritmo di ricerca A\* sia quello migliore rispetto agli altri due in quanto nonostante sia meno performante in termini di costo di memoria risulta essere il più veloce in termini di tempo impiegato.

Si vede anche come sia quasi sempre l’algoritmo di IDA\* quello con i tempi peggiori, possiamo ricondurre questa inefficienza a diversi fattori:  
- All’ordine col quale sono state scritte le regole  
- All’euristica utilizzata  
- All’istanza del problema e all’ordine col quale sono stati definiti i fatti

In generale possiamo anche notare come l’unico algoritmo che riesce sempre a terminare anche nella casistica senza soluzioni sia A\*.

# 2.ASP

Per la parte ASP ho implementato le soluzioni per i problemi del calendario di un torneo sportivo e per il problema del trasporto aereo descritto nel capitolo 10.1 del Russell e Norvig.

## 2.1 Calendario torneo sportivo

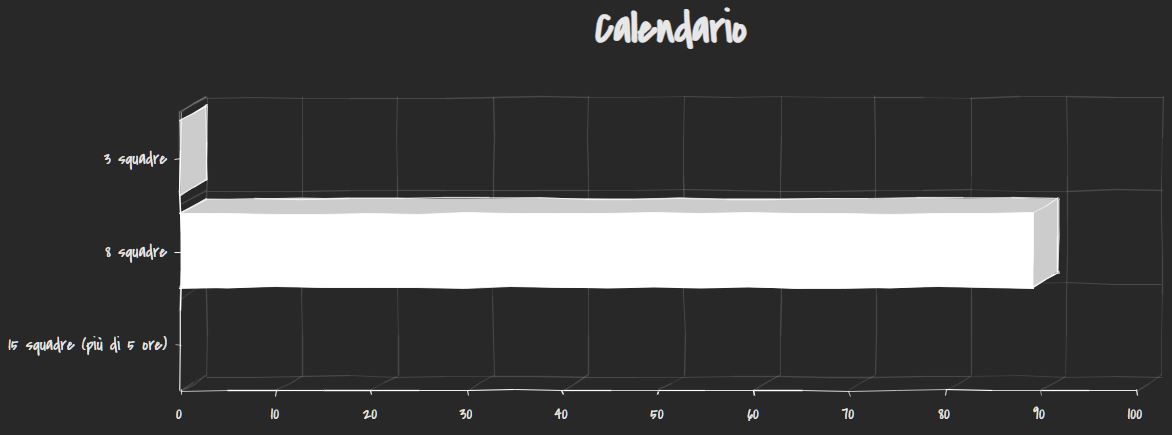
Questo tipo di problema è un classico problema di soddisfacimento di vincoli.  
Per implementarlo ho utilizzato i costrutti:

* *citta(<nomeCitta>)* : per dire quali sono le città del dominio
* *squadraLocation(<nomeSquadra>,<nomeCitta>)* : per indicare la città di casa di una squadra
* *squadra(<nomeSquadra>)* : per indicare le squadre del dominio

Per la definizione dei vincoli sono stati utilizzati sia il costrutto *:-* sia quello per la definizione dei vincoli di cardinalità:

*Es. % Ogni squadra deve affrontare una volta in casa ogni altra squadra*

*numteam-1 {match(S1,S2,C,G) : squadra(S2),citta(C),giornata(G)} numteam-1 :- squadra(S1).*



I test sono stati eseguiti su 3 istanze diverse del dominio.  
Le istanze si differenziano per il numero di squadre (e implicitamente per il numero di giornate = *numteam\*(numteam-1)*).

Possiamo notare come il tempo impiegato dall’algoritmo per risolvere il problema cresca in modo esponenziale rispetto al numero di squadre coinvolte.

## 2.2 Trasporto Aereo

Il problema del trasporto Aereo, a differenza del precedente, è un problema di pianificazione.

Per implementarlo ho seguito lo schema suggerito a lezione e quindi suddividendo i costrutti implementati in:

**Azioni**: *Es. action(carica(Merce,Aereo,Aereoporto)): merce(Merce),aereo(Aereo),aeroporto(Aereoporto).*

**Fluenti**: *Es. fluent(posizione(X,Y)):- aeroporto(Y), merce(X).*

**Effetti**:

*Es. holds(in(Merce,Aereo),Stato+1), not holds(posizione(Merce,Aereoporto),Stato) :- occurs(carica(Merce,Aereo,Aereoporto),Stato), state(Stato).*

**Precondizioni**:

*Es*. :- *occurs(carica(Merce,Aereo,Aereoporto),Stato), not holds(posizione(Merce,Aereoporto),Stato).*

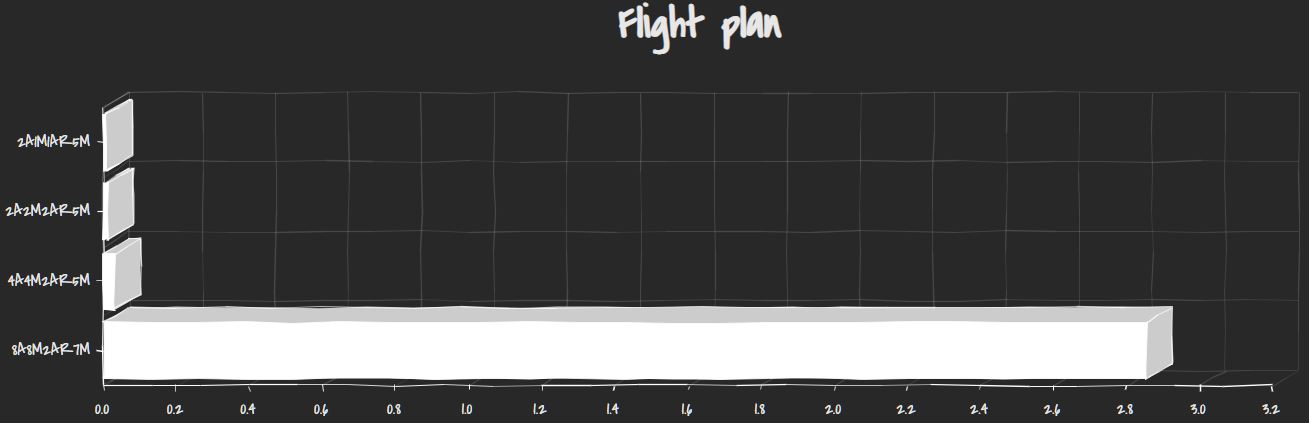
**Regole di persistenza**: *Es. -holds(F,S+1):- fluent(F),state(S),-holds(F,S),not holds(F,S+1).*

**Regole causali**:

*Es. -holds(posizione(X,Aereoporto1),Stato):-*

*fluent(posizione(X,Aereoporto1)),state(Stato), aereo(X), holds(posizione(X,Aereoporto2),Stato), aeroporto(Aereoporto2), Aereoporto1!=Aereoporto2, not holds(posizione(X,Aereoporto1),Stato).*

**Stato iniziale e goal.**



I test sono stati condotti su diverse istanze del dominio andando a modificare il numero di Aerei, il numero di Aeroporti, il numero di merci e la loro posizione iniziale.

Possiamo notare come nelle prime tre istanze del problema i tempi di esecuzione siano tutto sommato simili mentre l’ultima istanza (quella più complessa) ha richiesto un tempo di esecuzione molto più alto.  
Possiamo ricondurre questo drastico calo delle prestazioni ai parametri dell’istanza testata in quanto si è deciso di lasciare invariato il numero di aerei ma di raddoppiare sia il numero di aeroporti sia il numero di merci, inoltre le merci sono state disposte “lontane” dagli aeroporti di partenza degli aerei.