# Intelligenza Artificiale e Laboratorio

Anno Accademico 2019/20

Giuseppe Mazzone
Sante Altamura

# Programmazione logica: Prolog e ASP

## Implementazione di strategie di ricerca con Prolog

L'approccio seguito per la realizzazione del progetto ha permesso la creazione di algoritmi per implementare strategie di ricerca non informate e basate su euristiche, applicandole al problema del labirinto. 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*.

Per testare le strategie di ricerca è possibile effettuare la consult del loader `['loader.pl']` ed eseguire uno dei seguenti comandi:

- `playid` per l'Iterative Deepening;
- `playida` per l'Iterative Deepening A*;
- `playa` per l'A*;

### 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 che si basa su un limite fornito come variabile dinamica, inizialmente inizializzato a 1:

```
:-dynamic(limite/1).
limite(1).
```

Inizialmente l'interprete prolog prende in considerazione la prima clausola a cui si da il parametro di output "Soluzione" poi considera e cerca di dimostrare il predicato ricercaPath. Dopo c'è un cut in modo tale da rendere le scelte definitive, cioè, se è possibile una soluzione per ricercaPath allora ignora tutte le altre possibili scelte in cui aumenta di uno il limite

```
iterativeDeepening(Soluzione):-
    ricercaPath(Soluzione),
    !,
    retract(limite(_)),
    assert(limite(1)).
```

Nel passo base, effettua una ricerca in profondità con limite quello fornito dinamicamente sopra riportato:

```
ricercaPath(Soluzione):-
limite(N),
Limite is N,
write(Limite),nl,
iniziale(S),
ricerca_depth_limitata(S,[],Limite,Soluzione).
```

Se la ricercaPath non ricade nel passo base, cioè l'interprete prolog non è riuscito a trovare uno stato finale in base al limite corrente viene eseguita la seconda scelta della ricercaPath che permette di aumentare il limite di 1 e di controllare che non si sia giunti alla massima profondità:

```
ricercaPath(Soluzione):-
limite(N),
N1 is N+1,
massimaProfondita(D),
N1 =< D,
retract(limite(_)),
assert(limite(N1)),
ricercaPath(Soluzione).
```

Esplorare uno stato attraverso 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_depth_limitata` e viene invocato con il livello di taglio che,

durante la discesa in profondità nell'albero, non deve superare.

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 informate

### Euristiche implementate

### Algoritmo IDA*

Questo algoritmo permette di evolvere la ricerca di Iterative Deepening andando ad utilizzare una euristica per stimare il numero della soglia ad ogni passo di ricerca. Tale strategia utilizza una particolare struttura `nodo` per associare ad un determinato stato del problema la stima euristica **f(n)**:

```
ida_nodo(Stato, StimaEuristica)
```

dove:

- `Stato`: è uno stato proprio del dominio del problema dal quale vogliamo stimare la distanza;
- `StimaEuristica`: è la distanza stimata da Stato allo stato finale;

Tale struttura è fornita in maniera dinamica ed è definita dal seguente predicato extra-logico:

```
:-dynamic(ida_nodo/2).
```

Al primo passo di ricerca la soglia è data dalla stima euristica calcolata tra il nodo iniziale e il nodo finale, è importante notare la presenza del CUT che in questo caso ci permette di eliminare tutti gli altri percorsi dal ramo di risoluzione ottenendo quindi solo una soluzione:

```
idaStar(Sol):-
    iniziale(S),
    heuristic(S, _, EuristicaIniziale),
    ida(S, Sol, [S], 0, EuristicaIniziale),
    !.
```

Ad ogni iterazione viene effettuata la ricerca in profondità secondo la soglia determinata dall'euristica. Successivamente al passo iniziale, la stima euristica corrisponderà al minimo:

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

per tutti i nodi che superano la soglia del passo precedente. Se la ricerca in profondità fallisce allora vengono presi in considerazione i predicati successivi, altrimenti, viene correttamente istanziato il parametro di output. Nel caso vengano considerati i predicati successivi, il funzionamento per il ricalcolo della soglia è il seguente: vengono prese le F(S) per tutti gli stati attraverso il predicato `findall` e tra queste vengono considerate solo quelle che superano la soglia precedente (attraverso la `exclude`) ed inserite in OverEuristicaLista; ordinate le euristiche dalla più piccola alla più grande, viene presa la soglia che corrisponde alla minima euristica trovata e si chiama ricorsivamente `ida` sulla nuova soglia.

```
ida(S, Sol, Visitati, CostoPercorsoS, Euristica):-
    ida_search(S, Sol, Visitati, CostoPercorsoS, Euristica);
    findall(FS, ida_nodo(_, FS), EuristicaLista),
    exclude(>=(Euristica), EuristicaLista, OverEuristicaLista),
    sort(OverEuristicaLista, ListaOrdinata),
    nth0(0, ListaOrdinata, NuovaEuristica),
    retractall(ida_nodo(_, _)),
    ida(S, Sol, Visitati, 0, NuovaEuristica).
```

Clausola che rappresenta il caso base della ricerca in profondità, se lo stato è finale.

```
ida_search(S, [], _, _, _):-
    finale(S).
```

Se lo stato non è finale, avviene una ricerca in profondità, con limite dettato dall'euristica. Viene scelta un'azione applicabile nello stato S, si trova uno stato nuovo che verifichi il predicato `trasforma` si verifica che non appartenga agli stati visitati e si calcola il costo per passare al nuovo stato; per questo tipo di esperimento il costo è sempre unitario. Il costo del percorso per il nuovo stato `CostoPercorsoNuovaS` è dato da:

$$CostoPercorsoS + Costo$$

mentre il costo del percorso per il nuovo stato `FNuovoS` è dato da:

$$CostoPercorsoNuovaS + CostoHeuristicNuovaS$$

viene asserito un predicato che mi permette di tracciare `SNuovo` con la sua `F(SNuovo)`; se la profondità del nuovo stato è minore dell'euristica limite si chiama ricorsivamente ida_search sul nuovo stato.

```
ida_search(S, [Az|AltreAzioni], Visitati, CostoPercorsoS, Euristica):-
    applicabile(Az, S),
    trasforma(Az, S, NuovoS),
    \+member(NuovoS, Visitati),
    costo(S, NuovoS, Costo),
    CostoPercorsoNuovaS is CostoPercorsoS + Costo,
    heuristic(NuovoS, _, CostoHeuristicNuovaS),
    FNuovoS is CostoPercorsoNuovaS + CostoHeuristicNuovaS,
    assert(ida_nodo(NuovoS, FNuovoS)), FNuovoS =< Euristica,
    ida_search(NuovoS, AltreAzioni, [NuovoS|Visitati],
    CostoPercorsoNuovaS, Euristica).
```

### Algoritmo A*

Essendo questo algoritmo basato sulla ricerca in ampiezza e non essendo tale ricerca naturale per prolog la sua implementazione risulta più complessa, infatti si rende necessaria la creazione di una struttura dati ad hoc simile a quella precedente per associare ad un determinato stato del problema tutte le informazioni ad esso correlate. Si definisce così un nodo:

```
nodo(S, AzioniPerS, CostoAttuale, StimaEuristica)
```

dove:

- `S`: è uno stato proprio del dominio del problema;
- `AzioniPerS`: è la lista delle azioni che hanno portato a `Stato` dallo stato iniziale.
- `CostoAttuale`: è la distanza di Stato dallo stato iniziale;
- `StimaEuristica`: è la distanza stimata da Stato allo stato finale;

La clausola seguente permette di inizializzare il problema indicando lo stato iniziale e calcolando l'euristica iniziale a partire da tale stato S. Subito dopo viene invocato astarRicerca sul nodo di partenza, trovata una soluzione essa sarà l'unica ad essere prodotta grazie al CUT:

```
astar(Soluzione):-
    iniziale(S),
    heuristic(S, _, HeuristicS),
    astarRicerca([nodo(S,[],0,HeuristicS)],[],Soluzione),
    !.
```

Nel caso base di astarRicerca se ci si trova nello stato finale, le azioni da restituire come parametro di output sono proprio le azioni per raggiungere S

```
astarRicerca([nodo(S,AzioniPerS,_,_)|_],_,AzioniPerS):-
    finale(S),!.
```

Mentre il caso induttivo permette effettivamente di eseguire la ricerca in ampiezza, che a differenza di quella standard, ordina la lista dei nuovi stati possibili secondo le F(S) di tali nodi privilegiando quelle con stima minore:

```
astarRicerca([nodo(S,AzioniPerS,CostoAttuale,
HeuristicAttuale)|CodaStati],
Visitati,Soluzione):-
    findall(Az,applicabile(Az,S),ListaAzioniApplicabili),
    generaStatiFigli(nodo(S,AzioniPerS,CostoAttuale,HeuristicAttuale),
    [S|Visitati],ListaAzioniApplicabili,StatiFigli),
    append(CodaStati,StatiFigli,NuovaCoda),
    predsort(a_star_comparator,NuovaCoda,CodaOrdinata),
    astarRicerca(CodaOrdinata,[S|Visitati],Soluzione).
```

Nel caso base la lista azioni applicabili è vuota allora, banalmente, la lista di stati figli è vuota.

```
generaStatiFigli(_,_,[],[]).
```

Nel caso induttivo, ad ogni chiamata viene posizionato in testa alla lista dei figli il nodo rappresentato dal costo del suo percorso e dal valore dell'euristica.

```
generaStatiFigli(nodo(S,AzioniPerS,CostoAttuale,HeuristicAttuale)
    ,Visitati,[Az|AltreAzioni],[nodo(SNuovo,[Az|AzioniPerS],
    NuovoCostoPerS,NuovaEuristicaPerS)|AltriFigli]):-
    trasforma(Az,S,SNuovo), \+member(SNuovo,Visitati), !,
    costo(S, SNuovo, Costo), NuovoCostoPerS is CostoAttuale + Costo,
    heuristic(SNuovo,_,NuovaEuristicaPerS),
    generaStatiFigli(nodo(S,AzioniPerS,CostoAttuale
    ,HeuristicAttuale),Visitati,AltreAzioni,AltriFigli).
```

Fa riferimento al caso in cui SNuovo è gia stato visitato ma è necessario proseguire la ricerca.

```
generaStatiFigli(Nodo,Visitati,[_|AltreAzioni],AltriFigli):-
    generaStatiFigli(Nodo,Visitati,AltreAzioni,AltriFigli).
```

# Mappe Utilizzate

- Per la creazione delle mappe è stato utilizzato un generatore di labirinti implementato in Java.
- Per la creazione delle mappe in prolog, viene generato dallo stesso tool un file testuale con le posizioni occupate.
- Le posizioni iniziali `I` e finali `F` sono state scelte a posteriori dai progettisti.

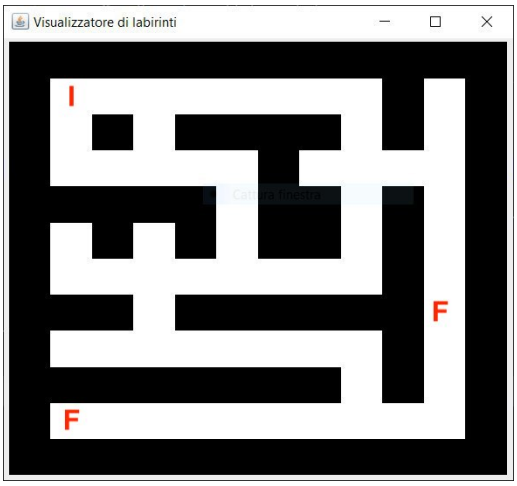



Figura 1: 10x10 A

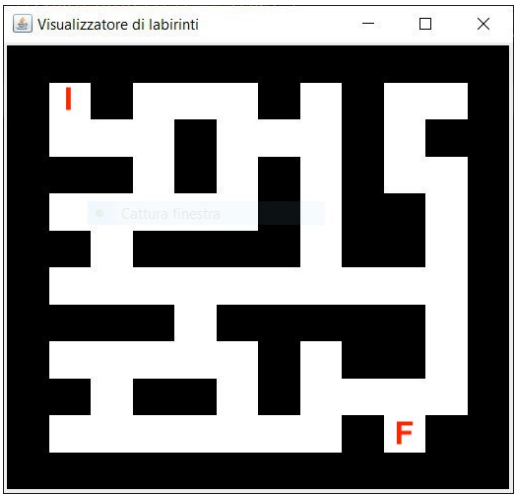



Figura 2: 10x10 B

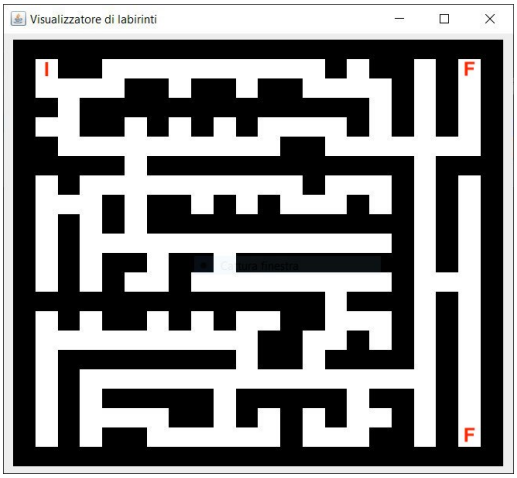



Figura 3: 20x20 A

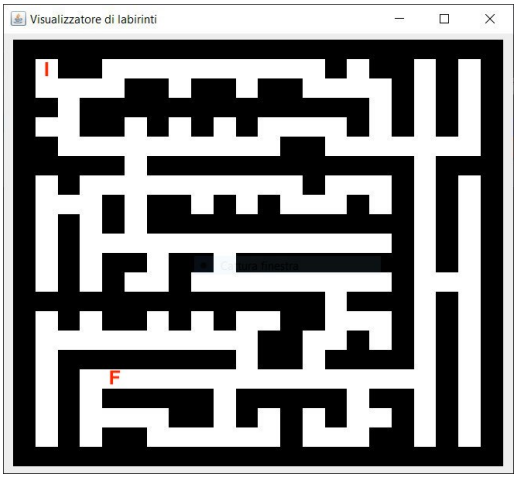



Figura 4: 20x20 B

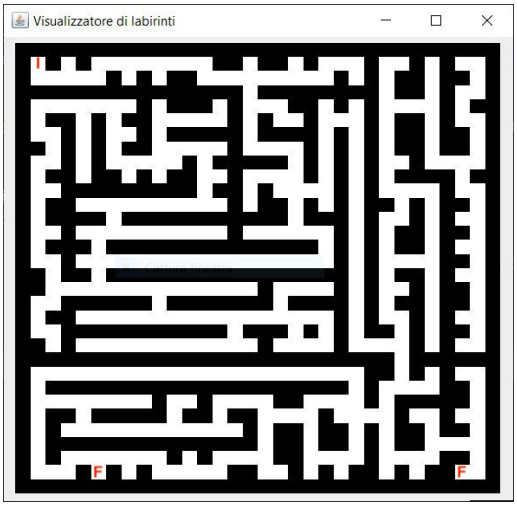



Figura 5: 30x30 A

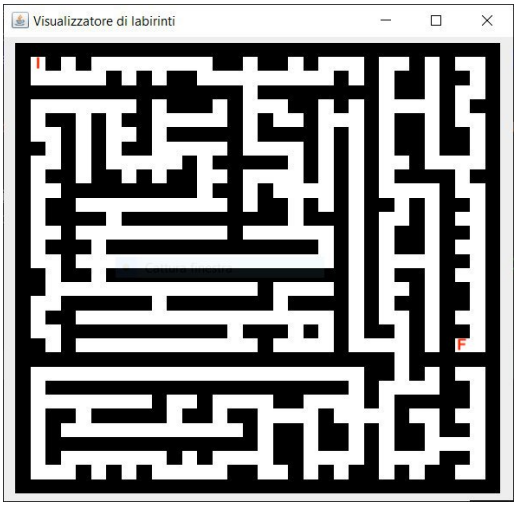



Figura 6: 30x30 B

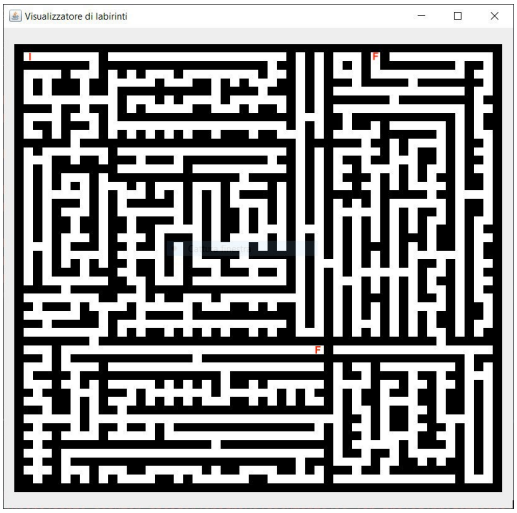



Figura 7: 50x50 A

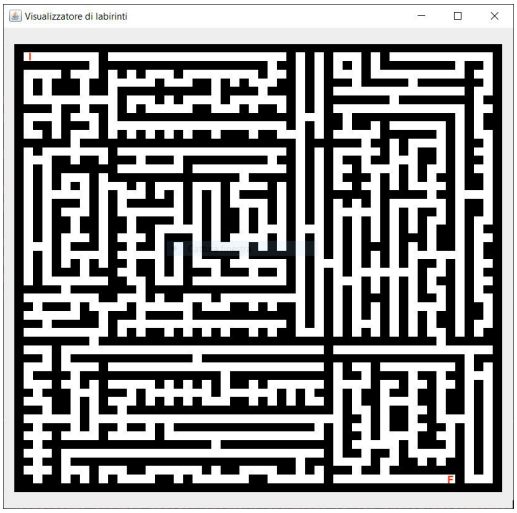



Figura 8: 50x50 B

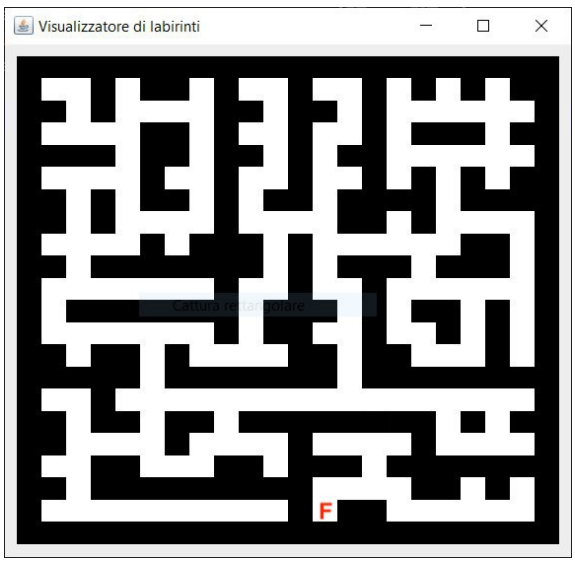
# Risultati Ottenuti

- In tutti i test effettuati, la strategia di ricerca informata $A*$ si è dimostrata superiore a tutte le altre: è stata la strategia più efficiente in termini di tempo. Questo vale sia per l'euristica basata sulla distanza di Manatthan sia per quella basata sulla distanza Euclidea, tenendo conto che sono due euristiche ammissibili per A*, in quanto la rendono ammissibile. Intuitivamente, si può anche capire il perchè in questo caso A* è computazionalmente ottimo: ha una **stima ottimistica** del costo del percorso attraverso ogni nodo considerato ed inoltre, trattandosi di una ricerca in ampiezza, non espande i nodi già espansi.
- L'Iterative Deepening A* risulta avere tempi di computazione buoni se la ricerca avviene su mappe di piccole-medie dimensioni. Uno svantaggio rispetto ad A* si può notare in relazione all'euristica utilizzata: la distanza euclidea, fa aumentare i tempi di esecuzione di IDA* per tutte le mappe, risultando anche più oneroso dell'Iterative Deepening. Invece, utilizzando la distanza di Manatthan come euristica, il tempo di computazione risulta essere notevolmente ridotto.

Tutte le strategie implementate, rispettano la:

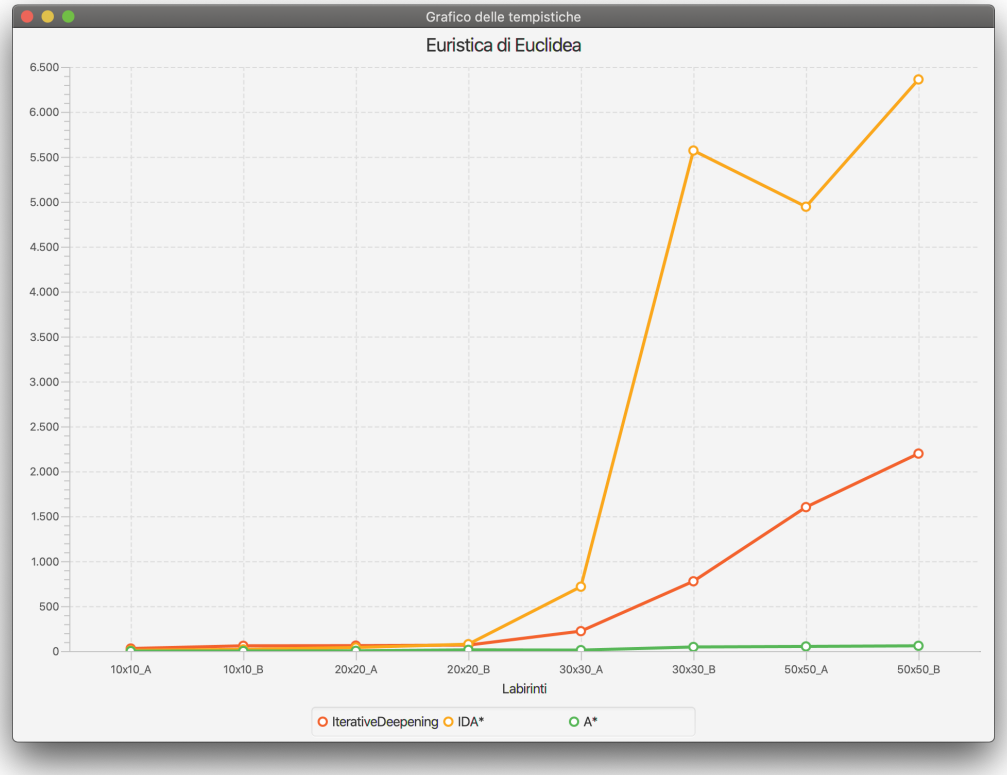
- **Correttezza**;
- **Completezza**;
- **Condizione di terminazione**;

Le strategie di ricerca sono state testate anche su mappe per le quali non è possibile raggiungere lo stato finale, come ad esempio:



| | Euclidea | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Strategia | 10x10A | 10x10B | 20x20A | 20x20B | 30x30A | 30x30B | 50x50A | 50x50B |
| Iterative Deepening | 29 | 59 | 61 | 67 | 222 | 777 | 1602 | 2198 |
| IDA* | 9 | 19 | 40 | 77 | 717 | 5570 | 4945 | 6361 |
| A* | 0 | 1 | 3 | 14 | 11 | 46 | 52 | 59 |

Tabella 1: Inferenze eseguite



| | Manhattan | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Strategia | 10x10A | 10x10B | 20x20A | 20x20B | 30x30A | 30x30B | 50x50A | 50x50B |
| Iterative Deepening | 29 | 59 | 61 | 67 | 222 | 777 | 1602 | 2198 |
| IDA* | 0 | 1 | 1 | 2 | 27 | 143 | 929 | 914 |
| A* | 3 | 3 | 2 | 3 | 8 | 32 | 47 | 61 |

Tabella 2: Inferenze eseguite