Intelligenza Artificiale e Laboratorio

Relazione progetto Prolog anno 2019/2020

Studenti:

Sanfilippo Paolo

Giuseppe Biondi

Antonio Surdo

Traccia:

Si richiede di implementare le seguenti strategie di ricerca:

- Strategie non informate:
  - iterative deepening
- Strategie basate su euristica:
  - algoritmo IDA*
  - algoritmo A*

applicandole al problema del labirinto descritto a lezione, in cui un sistema intelligente è collocato in uno spazio di n righe e m colonne, in cui sono posti degli ostacoli. Il sistema si trova in una delle celle del labirinto e, muovendosi all'interno dello stesso, deve raggiungere una casella di uscita. Il sistema può muoversi nelle quattro direzioni (nord, sud, est, ovest) e non in diagonale e, ovviamente, non può raggiungere una cella contenente un ostacolo. Il sistema intelligente conosce la configurazione del labirinto (dimensioni, posizione degli ostacoli, uscite).

Strategie non informate

Iterative deepening

L'implementazione di questo algoritmo sfrutta l'utilizzo della ricerca in profondità limitata dove la soglia viene incrementata ad ogni iterazione in cui non raggiunge un risultato finale.

La ricerca in profondità limitata è divisa in due parti:

- Nella prima parte andiamo ad inizializzare con la variabile S la posizione iniziale da cui si partirà all'interno del labirinto e chiamiamo la seconda parte passandogli:
  - S: posizione iniziale
  - Soluzione: la variabile in cui ci aspettiamo di ricevere il risultato della ricerca
  - [S]: una lista contenente i nodi che abbiamo visitato, essendo all'inizio conterrà solo quello iniziale
  - Soglia: l'attuale valore di soglia per questa iterazione

```
17  depth_limit_search(Soluzione,Soglia):-
18    iniziale(S),
19    dfs_aux(S,Soluzione,[S],Soglia).
```

Iterative deepening - figura 1. Ricerca in profondità, parte 1

- Nella seconda parte avverrà la ricerca ricorsiva in profondità dove la riga 21 rappresenta la condizione di successo: se si verifica finale(S), la nostra Soluzione inizia a ricostruirsi a ritroso partendo dalla lista vuota.
  Altrimenti finché la Soglia è positiva continuiamo la discesa ricorsivamente prendendo la prima Azione applicabile nello stato S, costruendo il nuovo nodo ottenuto con lo spostamento Azione e verificando che non si tratti di un nodo già visitato.
  La soluzione sarà costruita tramite backtracking aggiungendo in testa alla lista AzioniTail, l'Azione che è stata eseguita in questo step.

```
21  dfs_aux(S,[],_,_):-finale(S).
22  dfs_aux(S,[Azione|AzioniTail],Visitati,Soglia):-
23    Soglia>0,
24    applicabile(Azione,S),
25    trasforma(Azione,S,SNuovo),
26    \+member(SNuovo,Visitati),
27    NuovaSoglia is Soglia-1,
28    dfs_aux(SNuovo,AzioniTail,[SNuovo|Visitati],NuovaSoglia).
```

Iterative deepening - figura 2. Ricerca in profondità, parte 2

Ora che abbiamo visto la ricerca in profondità possiamo passare alla parte iterativa dell'algoritmo, ovvero dove incrementiamo la soglia ad ogni iterazione.

Per ottenere questo risultato semplicemente verifichiamo se è risolvibile con l'attuale soglia e in caso contrario quest'ultima viene incrementata di 1 e si riesegue la ricerca iniziale con il nuovo valore.

```
13  iterative_deepening_aux(Soluzione,Soglia):-
14    depth_limit_search(Soluzione,Soglia),!.
15  iterative_deepening_aux(Soluzione,Soglia):-
16    NuovaSoglia is Soglia+1,
17    iterative_deepening_aux(Soluzione,NuovaSoglia).
```

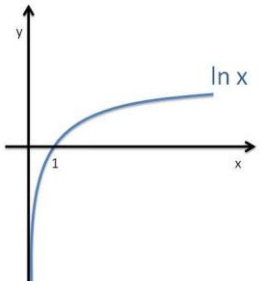
Iterative deepening - Figura 3. Parte iterativa

Tuttavia, questo tipo di approccio risulta infinito nel momento in cui si presenta un caso in cui non è possibile raggiungere un risultato finale, per questo scopo abbiamo ricercato delle condizioni di uscita, ovvero un Upper bound che raggiunto il quale decidiamo che una soluzione non è più ottenibile in tempi ragionevoli.

La nostra idea è stata di giocare con le dimensioni dei labirinti in maniera tale da avere una condizione di uscita dinamica in base al tipo di problema.

La prima idea è stata di semplicemente moltiplicare il numero di colonne per il numero di righe per assicurarci di non visitare più di quante caselle effettivamente esistano (ignorando la possibilità che alcune siano occupate) ma questa soluzione si è dimostrata insoddisfacente in quanto questo valore di soglia cresce proporzionalmente alle dimensioni del labirinto.

Abbiamo pensato di poter sfruttare la funzione del logaritmo naturale, come mostrato in figura 4 questo ci permette di avere una crescita controllata. Infatti, all'aumentare delle dimensioni del labirinto la crescita rallenta permettendoci di ottenere un valore che per essere raggiunto non richiede tempi troppo elevati.



Iterative dIterative deepening - Figura 4 . Funzione logaritmo naturale

Il risultato della funzione logaritmica è poi moltiplicato per una costante che è stata calcolata sulla sperimentazione. Questa costante ci permette di avere un valore di UpperBound sufficientemente grande per labirinti piccoli ma abbastanza piccolo per labirinti troppo grandi che richiederebbero una notevole quantità in termini di tempo.

Applicata questa modifica il codice rappresentato in figura 3 diventa come segue:

```
13  iterative_deepening_aux(Soluzione,Soglia):-
14    depth_limit_search(Soluzione,Soglia),!.
15  iterative_deepening_aux(Soluzione,Soglia):-
16    num_colonne(NumCol),
17    num_righe(NumRighe),
18    NuovaSoglia is Soglia+1,
19    NuovaSoglia < (log(NumCol * NumRighe) * 6),
20    iterative_deepening_aux(Soluzione,NuovaSoglia).
```

Iterative deepening - Figura 5. Parte iterativa con controllo soglia

Nota: il cut a riga 14 nella figura precedente elimina eventuali percorsi alternativi di costo pari o maggiore

Per concludere questo algoritmo può essere eseguito tramite la chiamata iterative_deepening(Soluzione).

Come possiamo vedere in figura 6 questa chiamata va a iniziare la ricerca a partire dalla Soglia 1.

```
1  iterative_deepening(Soluzione):-
2    iterative_deepening_aux(Soluzione,1).
```

Iterative deepening - Figura 6. Inizio algoritmo

Strategie informate

Algoritmo IDA*

Come l'iterative deepening questo algoritmo viene ripetuto più volte con una soglia crescente. Tuttavia, questa soglia non è più incrementata di un valore come nel caso precedente ma è calcolata cercando la minima funzione di valutazione tra i percorsi inesplorabili.

Per la funzione di valutazione abbiamo indicato con G la funzione costo, che ci rappresenta la lunghezza del percorso dallo stato iniziale al nodo attuale e con H la distanza di Manhattan (o geometria del taxi) come nostra euristica. La nostra funzione di valutazione F sarà ottenuta tramite la somma di queste due. F = G + H

In questa nuova ricerca in profondità distingueremo tre casi particolari:

- Se ci troviamo nel nodo finale:
  Siamo giunti alla soluzione del problema, la soluzione sarà la lista di azioni per giungere in questo nodo
- Se ci troviamo in un nodo la cui funzione F è minore della nostra attuale soglia:
  Possiamo proseguire con la ricerca in profondità perché ancora non abbiamo superato la nostra soglia.
- Se ci troviamo in un nodo la cui funzione F è maggiore della nostra attuale soglia:
  Non possiamo più proseguire in profondità, dobbiamo controllare se la funzione di valutazione di questo nodo può essere usata come nuova soglia per la prossima iterazione

Per comodità abbiamo rappresentato i nodi come una clausola nodo (P, Az) dove P rappresenta la posizione del nodo e Az è una lista di azioni eseguite dalla radice per giungere a questa posizione.

Inoltre, utilizziamo i predicati di prolog asserta/retract per salvare dinamicamente i valori di prossimoThreshold che rappresenta la soglia da utilizzare nella prossima iterazione e thresholdCheck che, come vedremo, sarà utilizzato per una condizione di uscita in una situazione senza un nodo finale raggiungibile.

Partiamo quindi a vedere in dettaglio la ricerca in profondità.

```
ricerca_in_profondita(NodiDaVisitare,Threshold,Espansi,Res)
```

IDA* - Figura 1. Ricerca in profondità

Come vediamo in figura 2, la riga n°50 rappresenta il primo caso che abbiamo accennato. Quando raggiungiamo un nodo che si unifica con la clausola di finale(N) allora unifichiamo il valore della soluzione (Res) con Az che rappresenta il percorso per giungere in tale nodo, ovvero la nostra soluzione.

```
50  ricerca_in_profondita([nodo(N,Az)|_],_,_,Az):-finale(N),!.
51  ricerca_in_profondita([nodo(N,Az)|Figli],Threshold,Espansi,Res):-
52    calcola_F(nodo(N,Az),F),
53    F =< Threshold,!,
54    generaFigli(nodo(N,Az),Espansi,FigliDiN),
55    append(FigliDiN,Figli,NodiDaVisitare),
56    ricerca_in_profondita(NodiDaVisitare,Threshold,[nodo(N,Az)|Espansi],Res).
57  ricerca_in_profondita([nodo(N,Az)|Figli],Threshold,Espansi,Res):-
58    prossimoThreshold(ProssimoThreshold),
59    calcola_F(nodo(N,Az),F),
60    (F < ProssimoThreshold ; ProssimoThreshold == Threshold),!,
61    retract(prossimoThreshold(ProssimoThreshold)),
62    asserta(prossimoThreshold(F)),
63    ricerca_in_profondita(Figli,Threshold,Espansi,Res).
64  ricerca_in_profondita([_|Tail],Threshold,Espansi,Res):-
65    ricerca_in_profondita(Tail,Threshold,Espansi,Res),!.
```

IDA* - Figura 2. Ricerca in profondità dettagliata

A partire dalla riga 51, inizia il secondo caso particolare, quando dobbiamo continuare la ricerca in profondità. In questo caso generiamo i figli del nodo corrente e li aggiungiamo in testa ai nodi da visitare. Nella chiamata ricorsiva di riga 56 aggiorniamo il valore di Espansi che rappresenta la lista di nodi da cui siamo già passati e che utilizzeremo all'interno di generaFigli per non passare da nodi già visti con una funzione di costo (G) minore rispetto al percorso attuale

Infine, nel terzo caso particolare, quando ci troviamo in un nodo la cui funzione F è maggiore del nostro threshold, dobbiamo aggiornare il prossimo valore di threshold. Questa si aggiorna se rientra in uno dei due casi a riga 60:

- Se la F è minore del ProssimoThreshold, quest'ultimo allora non è il minimo che stiamo cercando
- Se il ProssimoThreshold è uguale a quello attuale allora abbiamo appena iniziato la nuova iterazione e dobbiamo cambiare la prossima soglia.

Se nessuna delle due condizioni è verificata si esegue semplicemente la chiamata ricorsiva ignorando l'attuale nodo.

Questa ricerca sarà eseguita iterativamente, aggiornando ad ogni ciclo il nuovo valore di Soglia memorizzato in prossimoThreshold. Tuttavia, come nel caso dell'algoritmo

precedente è necessaria una condizione di uscita qual'ora il programma dovesse essere sottoposto a problemi senza soluzione.

Per questo abbiamo pensato di utilizzare un valore memorizzato dinamicamente in thresholdCheck, inizializzato a 0, contenente ad ogni giro il valore del vecchio Threshold cosicché se per due iterazioni otteniamo lo stesso valore di nuova soglia significa che siamo bloccati e abbiamo visto tutti i percorsi raggiungibili.

```
21  ida_aux(Nodo,Res):-
22    prossimoThreshold(Threshold),
23    generaFigli(Nodo,[],Figli),
24    retractall(thresholdCheck(_)),
25    asserta(thresholdCheck(Threshold)),
26    ida_aux2(Nodo,Figli,Threshold,[Nodo],Res),!.
27
28  ida_aux2(_,NodiDaVisitare,Threshold,Espansi,Res):-
29    ricerca_in_profondita(NodiDaVisitare,Threshold,Espansi,Res),!.
30  ida_aux2(Nodo,_,_,_,Res):-
31    thresholdCheck(Vs),
32    prossimoThreshold(Ps),
33    (Vs =\= Ps ; Vs =:= 0),!,
34    ida_aux(Nodo,Res).
```

IDA* - Figura 3. Controllo Threshold

Come vediamo in figura 3 a partire dalla riga 21 si inizializzano i valori da cui cominciare la ricerca, in particolare si aggiorna la clausola che traccia il valore di thresholdCheck con il valore di soglia che sarà utilizzata nella iterazione che sta per iniziare e in caso essa fallisca in riga 29, si passerà da riga 33 dove troviamo il controllo sopra accennato.

Per concludere questo algoritmo può essere eseguito tramite la chiamata ida_star (Soluzione).

Come possiamo vedere in figura 4 questa chiamata va a iniziare la ricerca resettando eventuali clausole rimaste sul threshold (in caso di esecuzioni ripetute) e inizializzandole pari a 0.

```
10  ida_star(Soluzione):-
11    iniziale(Start),
12
13    retractall(prossimoThreshold(_)),
14    retractall(thresholdCheck(_)),
15
16    asserta(prossimoThreshold(0)),
17    asserta(thresholdCheck(0)),
18    ida_aux(nodo(Start,[]),Soluzione).
```

IDA* - Figura 4. Inizio algoritmo

Algoritmo A*

A differenza dell'algoritmo precedente, in questo abbiamo deciso di rappresentare la funzione di costo e l'euristica all'interno della clausola nodo, la quale diverrà come segue:

nodo(Pos,G,H,Actions)

dove Pos indica la posizione, G la funzione di costo, H l'euristica e Actions l'insieme di azioni per giungere in questo stato.

Questo algoritmo si basa sull'uso di due liste:

- Opens: rappresenta l'insieme di nodi che dobbiamo esplorare.
- Closed: rappresenta l'insieme di nodi già chiusi.

Come nell'algoritmo precedente la funzione di costo, rappresentata con G, sarà data dal numero di azioni necessarie per arrivare in tale nodo dallo stato iniziale mentre l'euristica sarà la distanza di Manhattan.

Iniziamo a vedere in dettaglio la parte di ricerca. Questa è stata divisa in due blocchi di codice.

Il primo sarà invocato tramite una chiamata:

search(Opens,Closed,CurrentMin,Result)

Dove Opens e Closed sono le liste sopra accennate, CurrentMin rappresenta il nodo in cui ci troviamo e Result è il risultato che verrà restituito al termine.

```
36  search(_,_,nodo(CurrentMin,_,_,Actions),Actions):-finale(CurrentMin),!.
37  search(Opens,Closed,CurrentMin,Result):-
38    subtract(Opens,[CurrentMin],NewOpens),
39    generateChild(CurrentMin,Figli),
40    search_aux(NewOpens,[CurrentMin|Closed],Figli,ResOpens,ResClosed),
41    a_star_aux(ResOpens,ResClosed,Result).
```

A* - Figura 1. Ricerca in ampiezza, parte 1

In questo blocco andremo a rimuovere il nodo attuale CurrentMin dalla lista dei nodi Opens per spostarlo in quella dei Closed, ci costruiremo la lista dei nodi raggiungibili da questa posizione, questa lista è denominata Figli nella figura 1 – riga 39, e avvieremo la seconda parte di ricerca che si occuperà di smistare i figli tra le due liste Open/Closed.

Nella seconda parte di ricerca, invocata a riga 40 della figura1, distingueremo principalmente due casi:

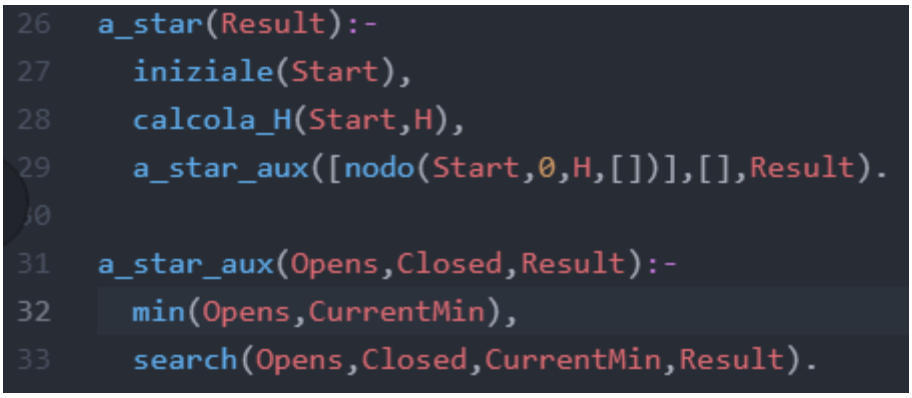
- Nel primo caso ci troviamo in un figlio non chiuso, quindi semplicemente lo aggiungiamo in testa agli aperti e chiamiamo ricorsivamente.
- Nel secondo incontriamo un nodo che è già stato chiuso, in questo caso dobbiamo considerare la funzione di valutazione. Se la funzione tramite il nuovo percorso è minore rispetto a quella del nodo già chiuso allora eliminiamo tale nodo dai chiusi e aggiungiamo quello in cui ci troviamo negli aperti, altrimenti passiamo direttamente alla chiamata ricorsiva.

```
46  search_aux(O,C,[],O,C).
47  search_aux(Open,Closed,[F|Figli],ResOpens,ResClosed):-
48    \+isMember(F,Closed),!,
49    search_aux([F|Open],Closed,Figli,ResOpens,ResClosed).
50  search_aux(Open,Closed,[nodo(Nodo,G,H,Actions)|Figli],ResOpens,ResClosed):-
51    F is G + H,
52    getNodeFromList(Nodo,Closed,nodo(Nodo,GNodo,HNodo,AzioniNodo)),
53    FNodo is GNodo + HNodo,
54    F < FNodo,!,
55    subtract(nodo(Nodo,GNodo,HNodo,AzioniNodo),Closed,NewClosed),
56    search_aux([nodo(Nodo,G,H,Actions)|Open],NewClosed,Figli,ResOpens,ResClosed).
57  search_aux(Open,Closed,[_|Figli],ResOpens,ResClosed):-
58    search_aux(Open,Closed,Figli,ResOpens,ResClosed).
```

A* - Figura 2. Ricerca in ampiezza, parte 2

Come vediamo in figura 2, dal momento che isMember controlla soltanto la posizione di due nodi ci serve una funzione per recuperare le informazioni di tale nodo dalla lista closed, questo avviene a riga 52 grazie alla chiamata getNodeFromList.

Terminiamo mostrando dove viene iniziata questa fase di ricerca e come vengono inzializzate le liste Open/Closed.



A* - Figura 3. Inizio algoritmo A*

Come vediamo nella figura 3, in riga 32 avviene la ricerca del nodo con funzione di valutazione minore che sarà restituito nella variabile CurrentMin dalla quale inizieremo la ricerca.

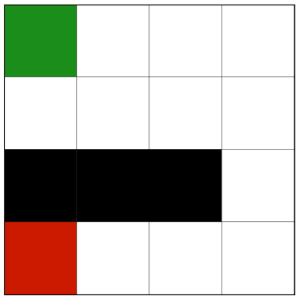
In fine l'algoritmo inizia creando le liste Opens/Closed rispettivamente con una lista contenente solo il nodo radice e la lista vuota.

I nostri labirinti

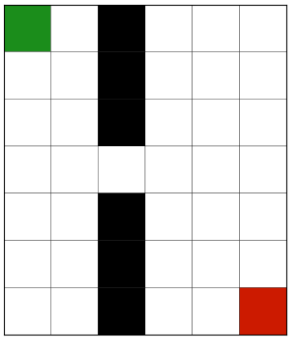
Per la realizzazione delle nostre mappe abbiamo implementato due programmi scritti in Python. Il primo è un generatore di labirinti totalmente casuale dal quale abbiamo creato i labirinti più complessi. Il secondo è un programma che dato un labirinto scritto in Prolog ci costruisce un'immagine automatica costruendo una tabella in cui le caselle assumono i seguenti colori:

- Bianche: caselle su cui possiamo muoverci
- Nere: i muri che non possiamo attraversare
- Verde: la posizione iniziale da cui partiamo
- Rosso: la casella di arrivo, nei casi studio in cui la casella di arrivo è fuori dal labirinto essa non sarà rappresentata

Labirinti piccoli:

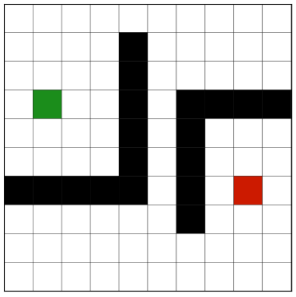


Labirinto 1 - labirinto 4x4

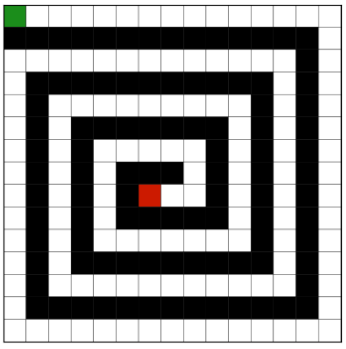


Labirinto 2 – labirinto 7x6

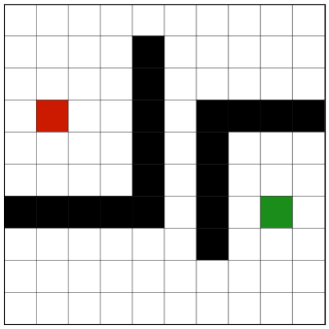
Labirinti medi:



Labirinto 3 – labirinto 10x10

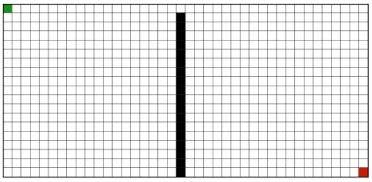


Labirinto 4 – labirinto 15x15

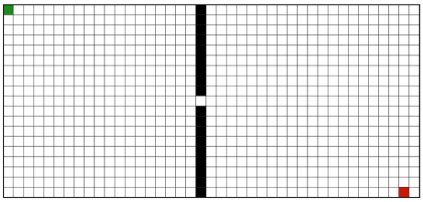


Labirinto 5 – labirinto inizio/fine invertiti

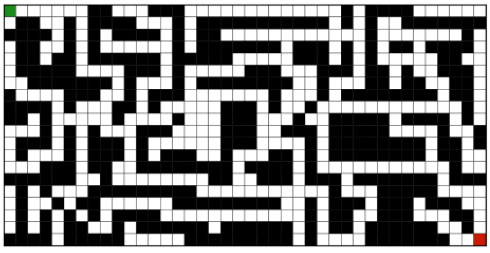
Labirinti grandi:



Labirinto 6 – labirinto 19x40

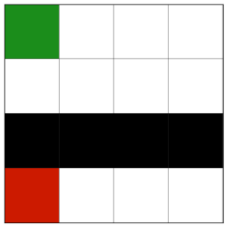


Labirinto 7 – labirinto 19x41

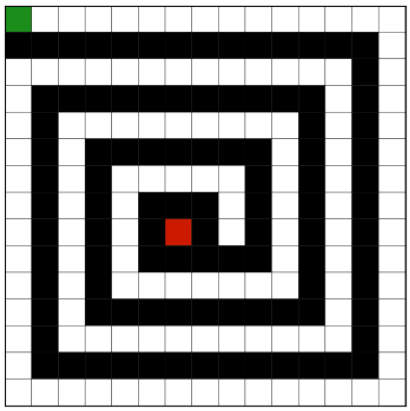


Labirinto 8 – labirinto 20x40

Labirinti con soluzione non raggiungibile:



Labirinto 9 – labirinto chiuso 4x4

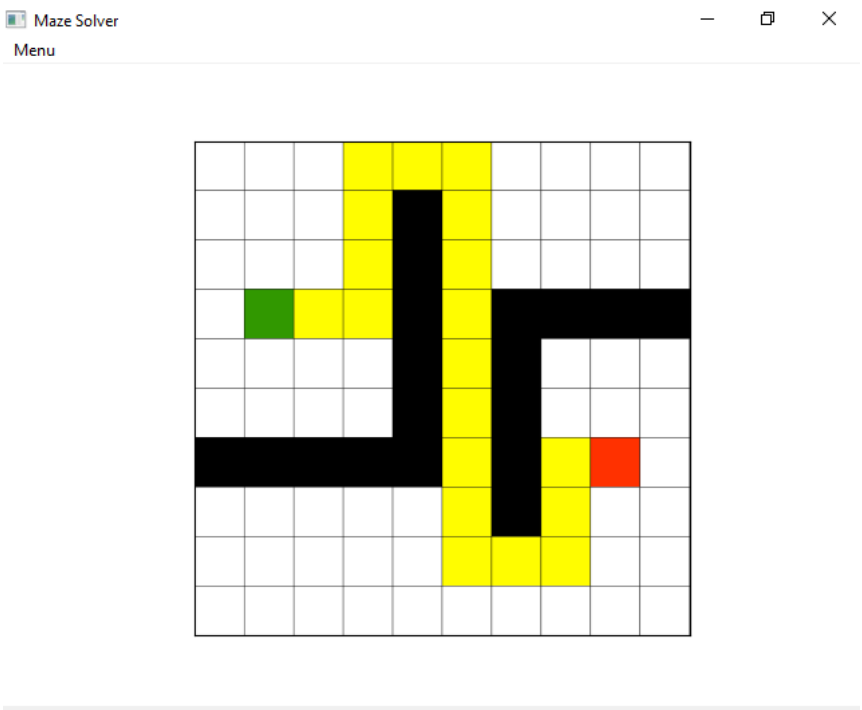


Labirinto 10 – labirinto chiuso 15x15

Applicazione Python

Per concludere il nostro lavoro abbiamo realizzato una semplice interfaccia scritta in Python che ci permette di:

- Selezionare il labirinto per visualizzarlo a video
- Scegliere con quale algoritmo vorremmo risolverlo
- Mostrare il percorso della soluzione in giallo dopo aver avviato la risoluzione



Conclusioni

Dai test effettuati abbiamo riscontrato che l'algoritmo A* è il più efficiente in termini di tempo.

Rispetto all'IDA* ha il vantaggio di non rivisitare i nodi già espansi una volta e lo svantaggio di consumare più memoria, questo svantaggio anche su labirinti più grossi, come nel Labirinto8, non mostra differenze.

Il peggiore in termini di tempo è L'Iterative Deepening, in particolare in labirinti molto aperti, come nel caso di Labirinto6 e Labirinto7, essendo che ha molti percorsi alternativi da esplorare richiede una quantità di tempo elevata.

- **Completezza:**
  Tutti gli algoritmi garantiscono un risultato, abbiamo implementato labirinti privi di soluzioni come i numeri8 e 9. Gli algoritmi terminano restituendo fallimento, ovvero l'impossibilità di trovare una soluzione.
- **Correttezza:**
  La correttezza è garantita per A* e IDA* che restituiranno sempre una soluzione corretta.

  Nel caso dell'Iterative Deepening, per via del criterio di terminazione adottato è possibile che venga interrotto prima di raggiungerla, quindi restituirà fallimento prima anche se è presente una soluzione nel caso di labirinti troppo grandi.

  Questo non si verifica nell'IDA* dove il criterio di stop è stato calcolato usando le variazioni di threshold, in questo caso con l'aumentare delle dimensioni del labirinto aumenta il tempo necessario per risolverlo ma garantisce di poter sempre arrivare alla soluzione quando esiste.