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

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.
Ho utilizzato due funzioni euristiche:

- Distanza di **Manhattan**: *L1(P1,P2) = |x1-x2|+|y1-y2|*
- Distanza di **Chebyshev**: *Linf=max(|x1-x2|+|y1-y2|)*

Entrambe soddisfano 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.

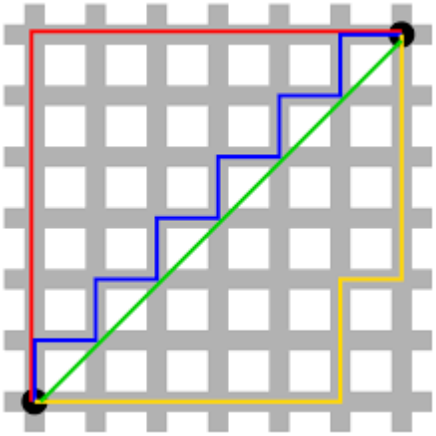


Figura 1.1: *La linea verde rappresenta la distanza euclidea mentre le altre indicano alcune possibili rappresentazioni geometriche della distanza di Manhattan (tutte equivalenti).*

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

1.1 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, non ho volutamente inserito un meccanismo di limitazione della ricerca proprio per “verificare” la non terminazione dell’algoritmo.

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

1.2 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 l’euristica scelta

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.

1.3 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 uno heap dove ad ogni nodo è assegnata la rispettiva priorità f(n). Ad ogni iterazione viene quindi espanso dallo heap il nodo con priorità più bassa fino a quando non viene soddisfatto lo stato goal.

Utilizzando questo tipo di struttura dati si ottiene un incremento prestazionale notevole (rispetto ad utilizzare una coda) in quanto si può utilizzare un operatore per il getMin di complessità O(1) e un operatore di inserimento di complessità O(log(n)), con una coda invece le complessità sarebbero state rispettivamente O(n) e O(1).

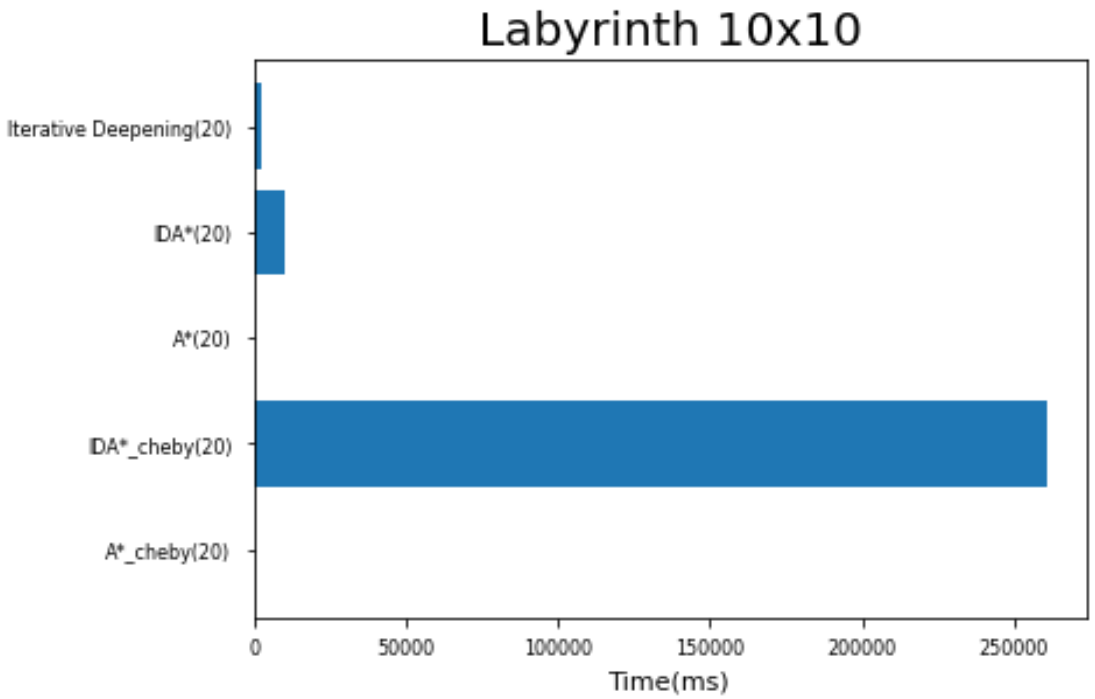
A*, rispetto a IDA*, richiede molta più memoria in quanto deve ricordare tutti i nodi della coda più quelli già esplorati (che non devono essere riespansi).

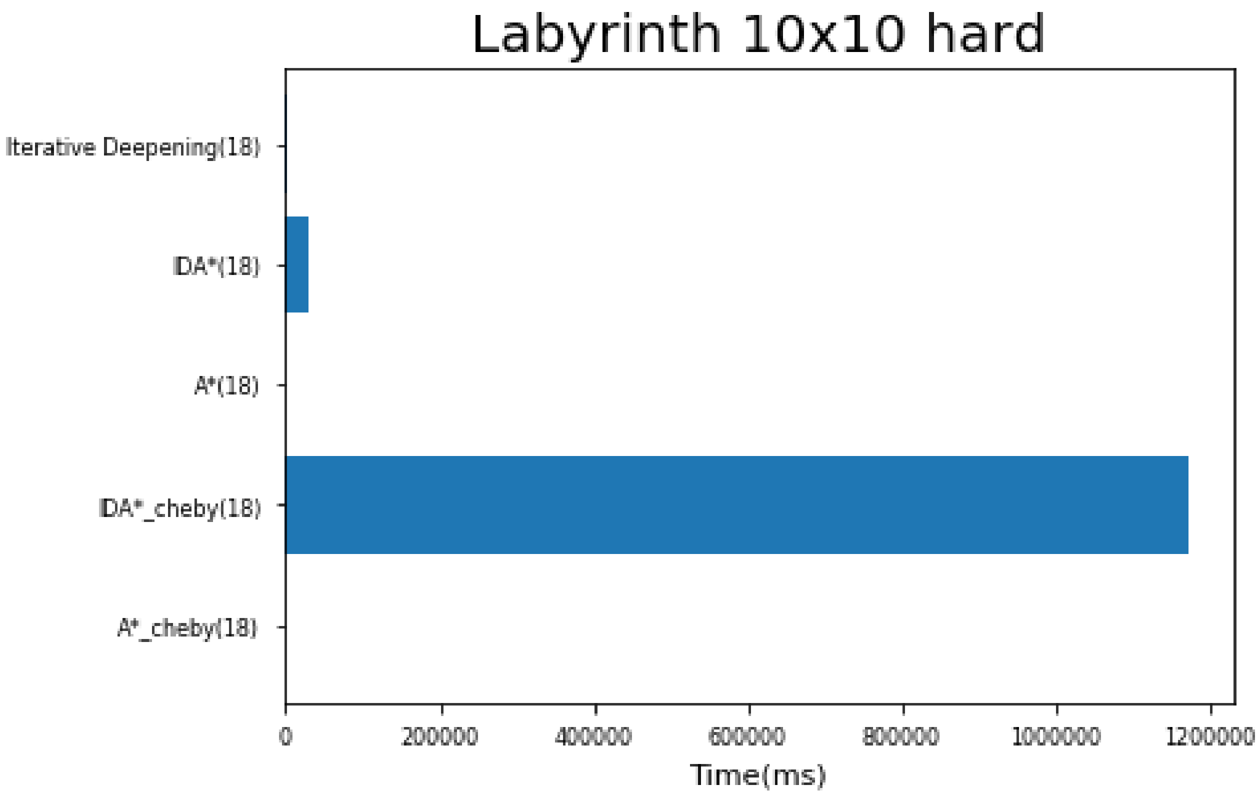
A* è un algoritmo completo per domini finiti in quanto prima o poi la coda diventerà vuota e l'algoritmo terminerà negativamente

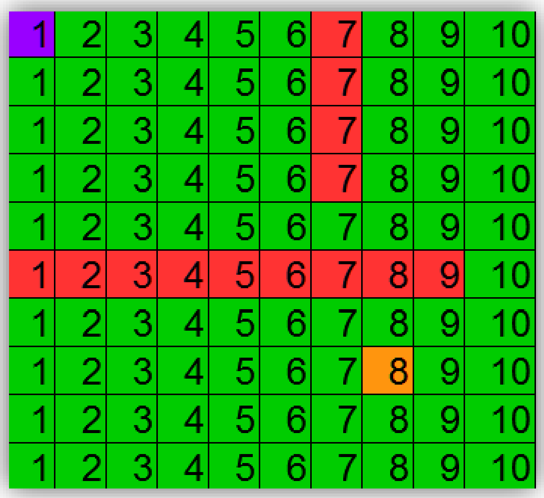
1.4 Benchmark e considerazioni

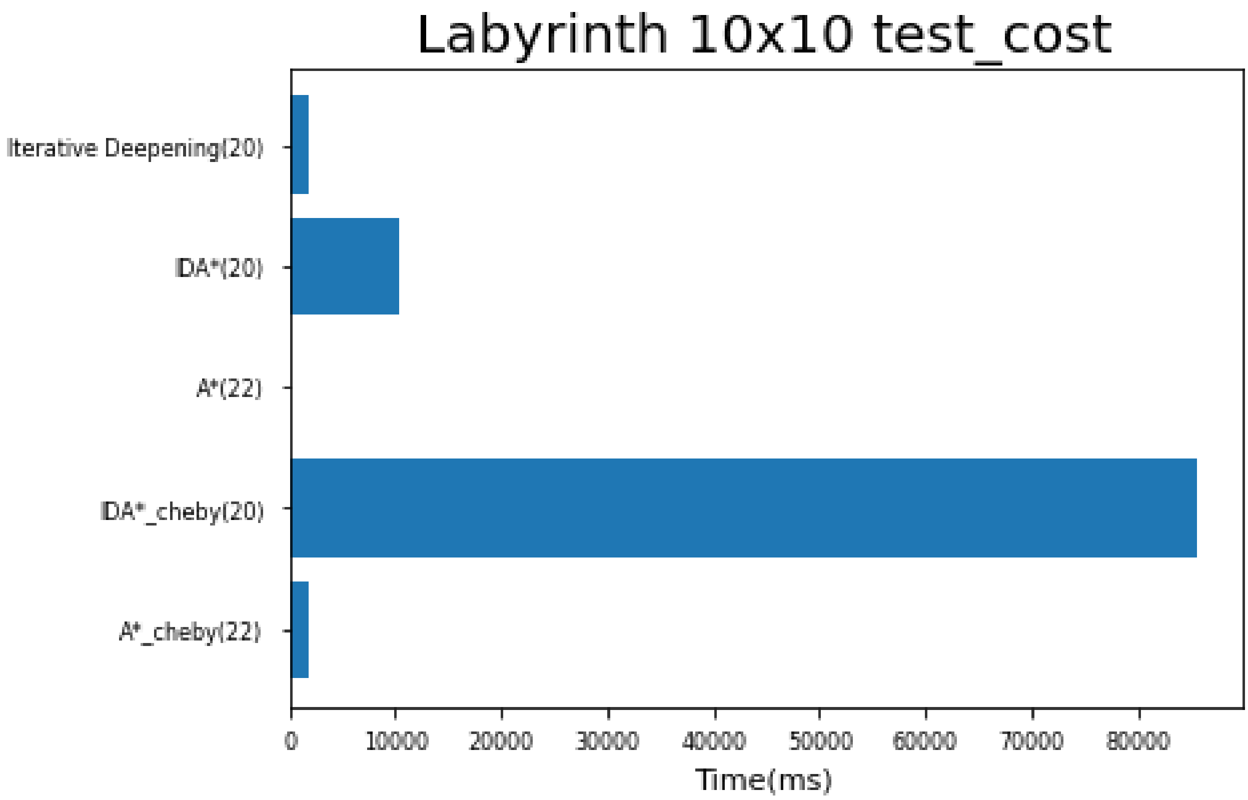
LEGENDA: blu = partenza; rosso = ostacolo; giallo = libero; arancione = uscita.

1.4.1 Labirinto




Labyrinth 10x10 hard
Iterative Deepening(18)
IDA*(18)
A*(18)
IDA*_cheby(18)
A*_cheby(18)
0
200000
400000
600000
800000
1000000
1200000
Time(ms)


1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10



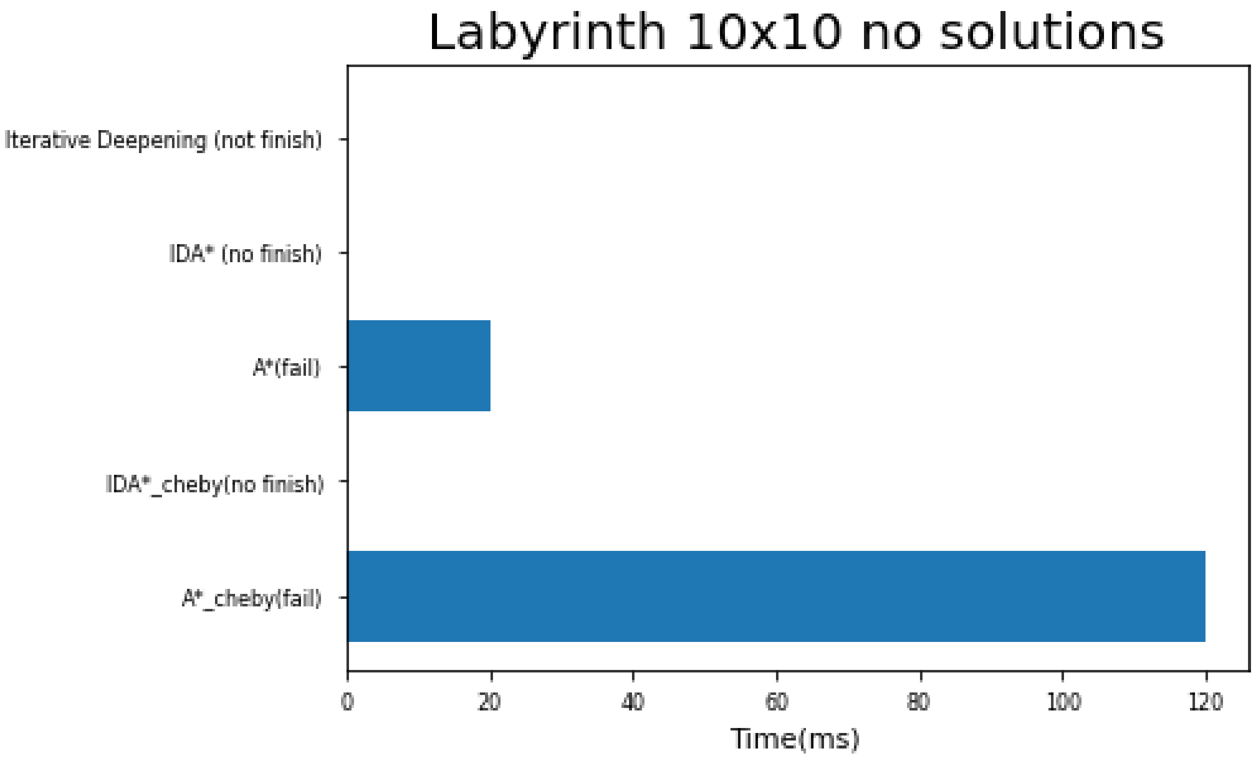
costo(4,2,3,2,40).

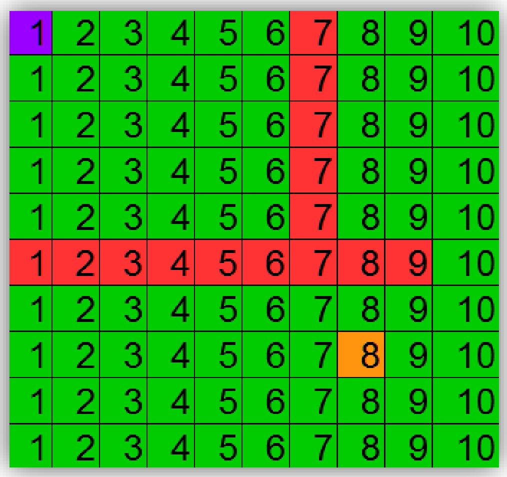
costo(2,4,1,4,10).

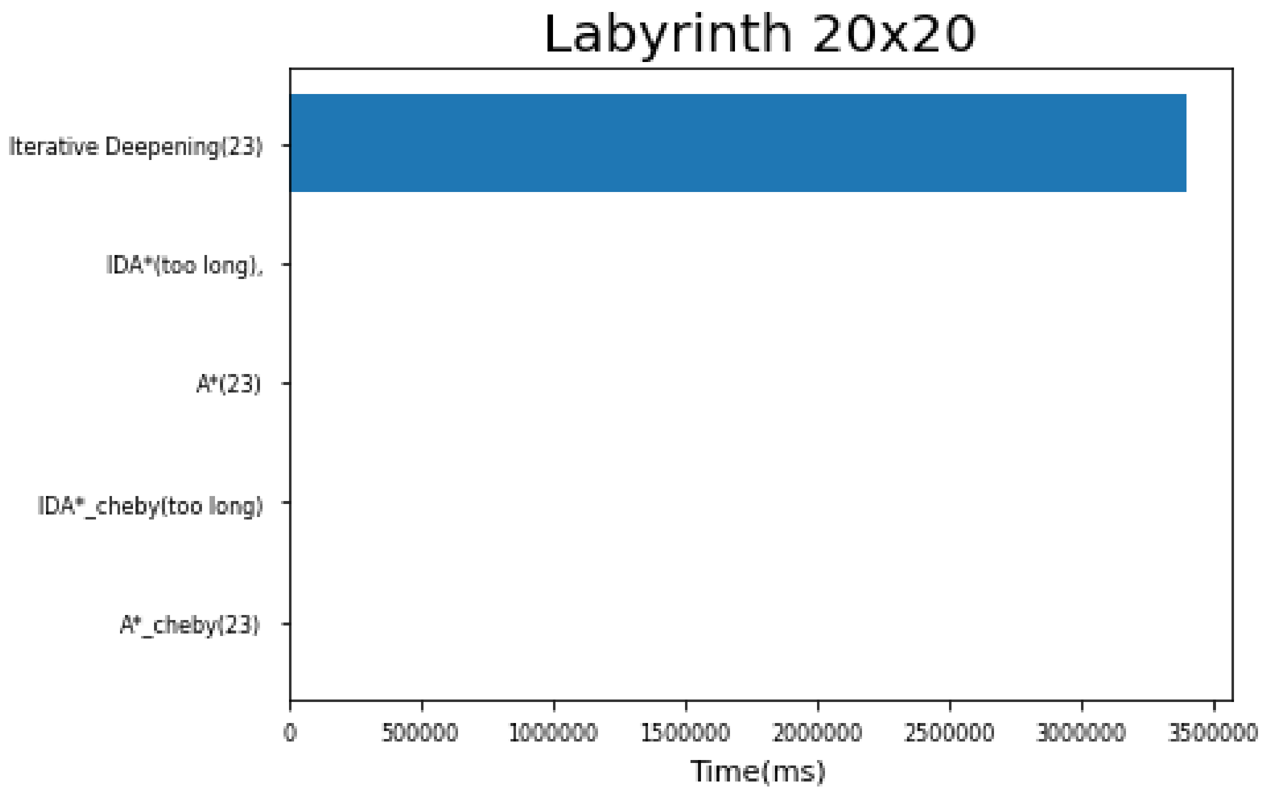
costo(9,8,8,8,67).

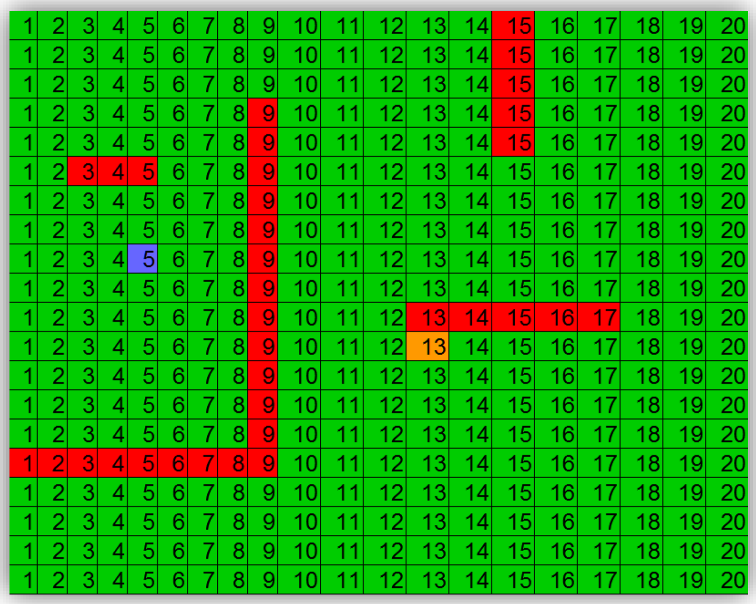
costo(8,9,7,9,100).

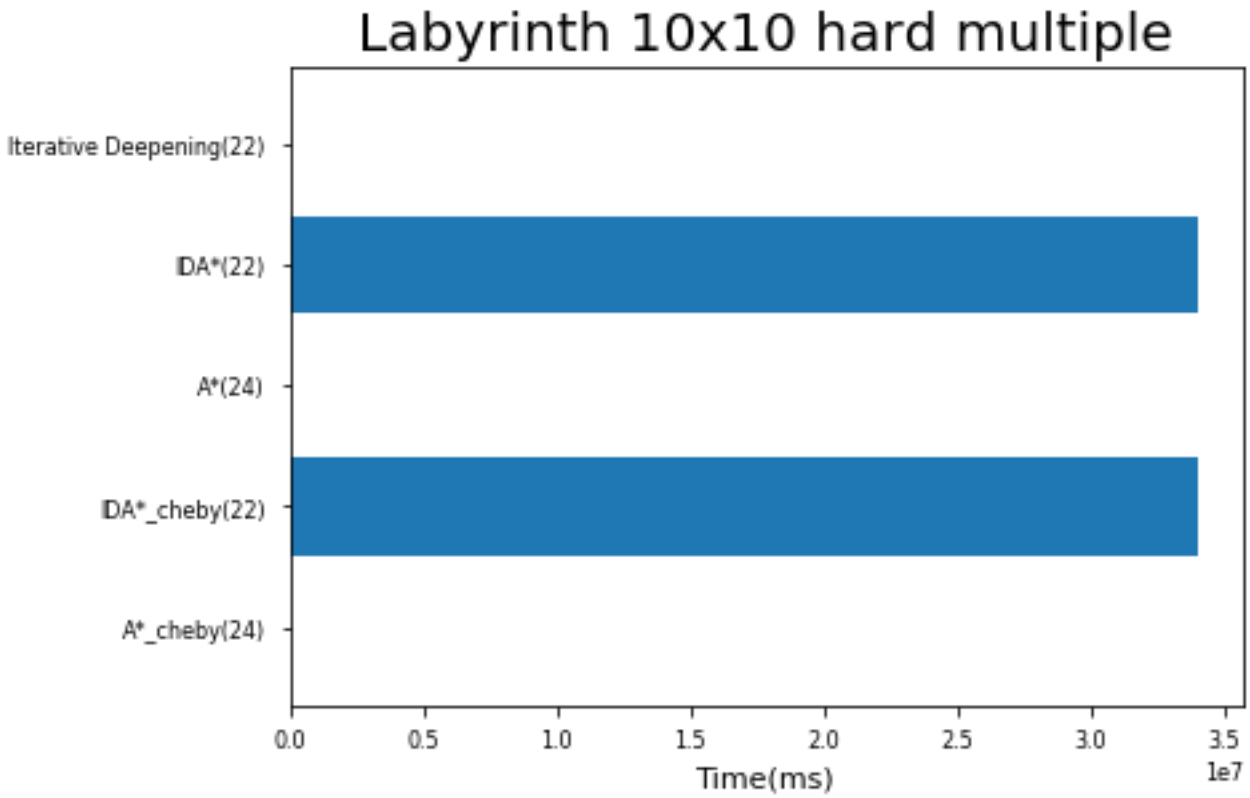
costo(7,8,7,9,100).


Labyrinth 10x10 no solutions
Iterative Deepening (not finish)
IDA* (no finish)
A*(fail)
IDA*_cheby(no finish)
A*_cheby(fail)
0
20
40
60
80
100
120
Time(ms)


1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10


Labyrinth 20x20
Iterative Deepening(23)
IDA*(too long),
A*(23)
IDA*_cheby(too long)
A*_cheby(23)
0
500000
1000000
1500000
2000000
2500000
3000000
3500000
Time(ms)





1.4.2 Considerazioni

Dai test condotti possiamo subito osservare come gli algoritmi testati abbiano performance nettamente differenti in termini di tempo ma molto simili in termini di soluzione trovata.

Possiamo constatare come l'algoritmo di ricerca A*, 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 IDA* quello con i tempi peggiori in quanto ogni volta deve rieffettuare la ricerca ripartendo dalla radice (ma con un taglio più profondo).

Possiamo anche notare che l'unico algoritmo che riesce sempre a terminare anche nella casistica senza soluzioni sia A*.

Tra le due euristiche utilizzate quella che porta a risultati migliori è la distanza di Manhattan mentre con quella di Chebyshev i tempi si dilatano notevolmente.
Questo era prevedibile poiché per IDA* questa euristica porta ad un incremento del taglio più "lento" e quindi ad una dilatazione dei tempi totali, per A*questo è spiegabile dal fatto ad esempio che le celle [0,0]-[10,10] e [0,0]-[1,10] hanno la stessa distanza anche se la cella [1,10] in realtà è più vicina alla [0,0] (considerando il numero minimo di celle da percorrere per arrivare alla cella [0,0]).

Notiamo anche come nel caso in cui ci siano costi differenti per le azioni effettuabili, l'unico algoritmo di ricerca che ne "beneficia" è la ricerca A* in quanto questo cambia la priorità nella coda delle azioni possibili. Nell'algoritmo IDA* invece questo sostanzialmente non cambia nulla in quanto l'euristica serve per scegliere il nuovo taglio ma che è sempre scelto come il minimo tra quelli che superano il taglio attuale.

2.ASP

Per il progetto relativo alla parte sull'Answer Set Programming ho implementato una mia soluzione relativa al problema a vincoli per la programmazione del *"Master in Progettazione e Management del Multimedia per la Comunicazione"* .

Elenco qui le scelte progettuali che secondo me sono state più importanti:

1. Ho modellato i concetti con i seguenti termini:

CONCETTO	IMPLEMENTAZIONE ASP
Insegnate	insegnante(X).
Insegnamento	insegnamento(Nome,Prof,OreTot).
Vincolo di propedeuticità	propedeutica(X,Y). *Indica che X è propedeutico per Y*
Tempo	giornata(Num,Ore). *Num indica il giorno; Ore può essere 8 o 5*
Lezione	lezione(Giornata,Professore,Insegnamento,Ora). *La lezione di Insegnamento tenuta da Professore si svolgerà nella giornata numero Giornata nella Ora ora*

2. Ho strutturato il problema cercando di seguire l'approccio **Generate and Test** caratterizzante del paradigma ASP:
    a. Ho prima definito delle regole che generassero i termini *lezione(G,P,I,O)* vincolandole nella cardinalità in base al fatto che fosse un giorno da 8 ore oppure un giorno di al più 5 ore
    b. Successivamente ho definito tutti i constraint (regole senza testa) necessari per scartare i modelli inconsistenti con i requisiti imposti dal progetto.
3. Ho risolto i vincoli sulla cardinalità facendo largo uso della notazione *Min{<termine>:<dominio>}Max* per indicare il numero minimo e massimo di quei termini che potevano comparire nell'answer set in output.
4. Per implementare requisiti più complessi come quelli relativi al fatto che la prima lezione di una materia deve iniziare dopo le prime 4 ore di un'altra, ho utilizzato gli operatori *#min* e *#max* di

Clingo. Grazie a questi sono riuscito a recuperare le giornate relative alla "prima lezione" e "ultima lezione" di uno specifico insegnamento e quindi ad utilizzarle per specificare i relativi vincoli.

5. **TEST**: Per convincermi del fatto che l'answer set generato in output fosse quello corretto (oltre che analizzare a mano quello che veniva stampato in console) ho implementato un tool in Python che tramite l'utilizzo della libreria Pandas controlla uno per uno tutti i vincoli rigidi richiesti nei requisiti. Questo mi ha permesso anche di individuare inconsistenze non evidenti ma che evidenziavano un errore di programmazione nella definizione dei vincoli.

Come possiamo notare nella tabella sottostante, misurando la CPU Time alterando solamente il numero di vincoli auspicabili attivi nella definizione del problema, il tempo totale di esecuzione varia discretamente solo nel caso in cui ci sono tutti i vincoli attivi (circa +2 secondi). Questo delta è riconducibile al fatto che il solver scarterà molti più modelli di quanti non ne scarta nella versione senza vincoli auspicabili.

Tutti i vincoli attivi	Vincoli auspicabili non attivi	Solo i primi 3 vincoli auspicabili
4.375s	2.797s	2.766s

3.Clips

Il progetto implementato tramite CLIPS è relativo alla definizione di un agente intelligente che sappia giocare al gioco della battaglia navale (secondo le regole e i vincoli specificati nel relativo documento).

3.1 Descrizione soluzione

L'approccio adottato è stato quello di cercare di definire delle regole che ricalcassero quanto più fedelmente possibile il ragionamento di un giocatore umano.

È stato quindi definito un nuovo template *deduced-cell* avente la stessa struttura del template *k-cell* e che è servito ad identificare le celle il cui contenuto è stato dedotto dall'agente tramite una GUESS.

Le mosse di tipo GUESS sono state utilizzate per tutti i casi in cui l'agente poteva dedurre il contenuto di una determinata cella.
Ad esempio: se l'agente è a conoscenza che il contenuto della cella [0,0] è LEFT e quello della cella [0,2] è RIGHT, può effettuare una operazione di GUESS nella cella [0,1].

Le mosse di tipo FIRE sono invece state utilizzate non solo per casistiche certe, ma anche per casi in cui era solo probabile che ci fosse una casella piena.
Nello specifico, le mosse di tipo FIRE sono state utilizzate anche nel caso in cui l'agente non abbia, in un determinato momento, una conoscenza tale da permettergli di prendere una decisione più sensata di una casuale: in tal caso (ad esempio nei test senza conoscenza iniziale) l'agente sceglie una casella casuale facendosi "influenzare" esclusivamente dai fatti di tipo *k-per-row* e *k-per-col* (regola: *random-fire-smart*)

Usare le mosse di tipo FIRE nei casi di incertezza, come quello più estremo relativo a nessuna conoscenza iniziale, permette eventualmente di scoprire nuovi fatti che possono portare l'agente a poter effettuare altre azioni di GUESS o FIRE più intelligenti.

Una critica che si potrebbe avanzare è che una fire-ko incide molto di più di una guess-ko nel punteggio e quindi magari avrebbe più senso usare le fire in caso di certezza e usare le guess in caso di incertezza. Questa sarebbe un'osservazione perfettamente sensata anche se bisognerebbe valutare quanto incide il

fatto che una guess non ci permette effettivamente di vedere cosa c'è su una casella e quindi non aggiunge una conoscenza reale del mondo. Inoltre bisognerebbe definire euristicamente o casualmente il contenuto della cella alla quale si vuole applicare GUESS.
Per sperimentare queste osservazioni è stata utilizzata anche un'altra implementazione dell'agente (file *3_Agent_fireguess_switched.clp*) che chiameremo Algo2.
Le principali differenze riguardano:

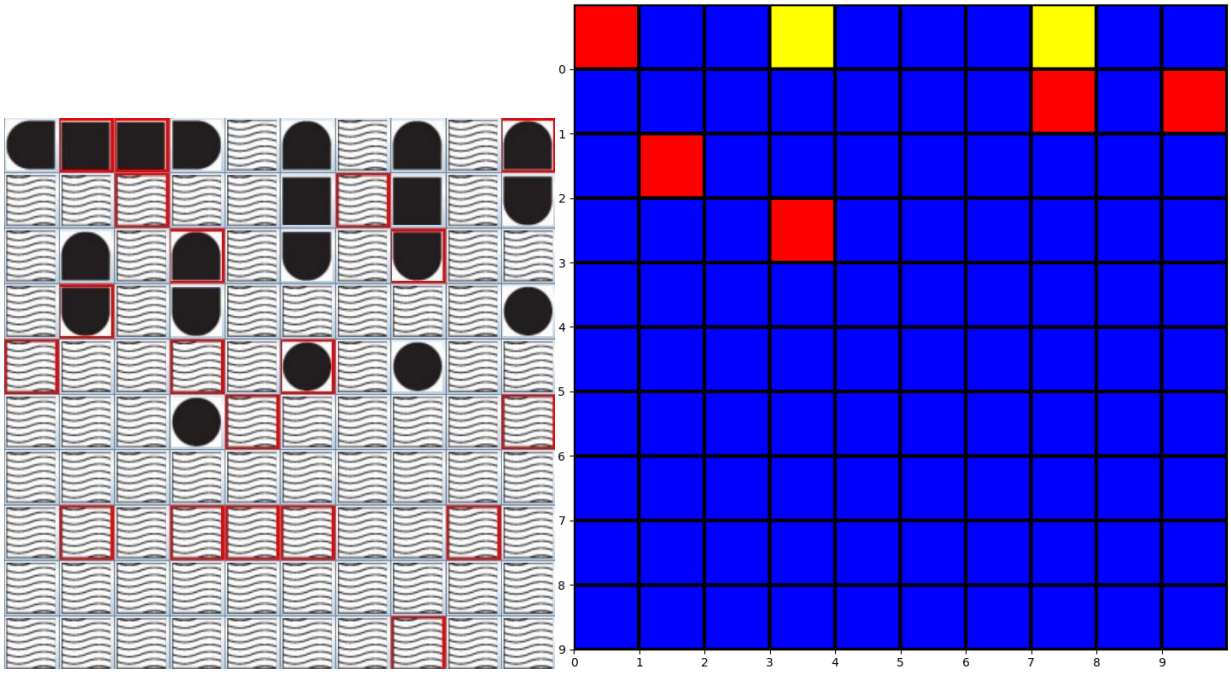
- Si usano le guess anche in caso di incertezza, scegliendo euristicamente il completamento più "corto", quindi si ipotizza meno volte middle e si preferisce top,bot,left o right (il contenuto inferito lo si trova nel fact *deduced-cell*). Ha senso in quanto le navi più corte sono più frequenti.
- È implementato un meccanismo di backtracking usando le azioni di tipo UNGUESS nei casi in cui l'agente si accorga che il guess effettuato in uno step precedente non può essere giusto

3.2 Test e risultati

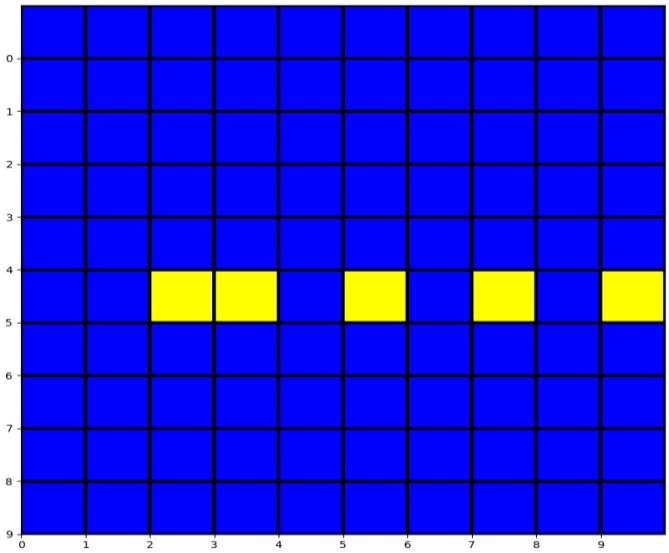
Sono stati effettuati dei test su due mappe diverse e con versioni aventi della conoscenza iniziale o meno. Per ogni caso vengono riportate le statistiche finali.

Le celle di colore giallo rappresentano le guess, quelle di colore rosso rappresentano le fire.

3.2.1 Test A



SCORE: 100 FIRE_OK:5 FIRE_KO:0 GUESS_OK:2 GUESS_KO:0 SAFE:6 SINK:6



Versione senza conoscenza iniziale

SCORE: -265

FIRE_OK: 1

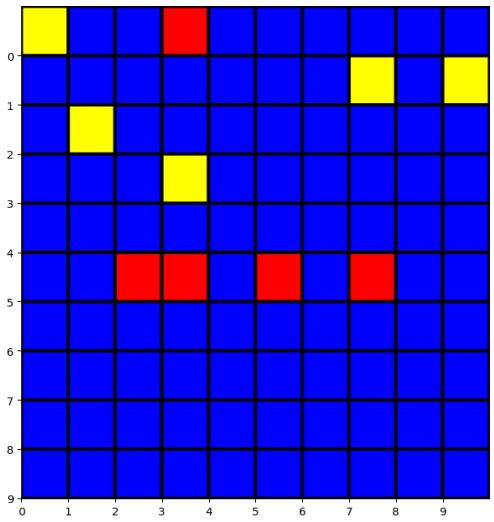
FIRE_KO: 4

GUESS_OK: 0

GUESS_KO: 0

SAFE: 19

SINK: 1



Versione con conoscenza – Algo2

SCORE: 25

FIRE_OK: 2

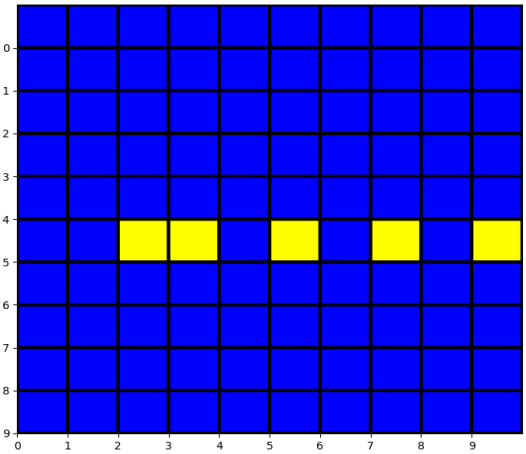
FIRE_KO: 3

GUESS_OK: 5

GUESS_KO: 0

SAFE: 6

SINK: 6



Versione senza conoscenza – Algo2

SCORE: -265

FIRE_OK: 1

FIRE_KO: 4

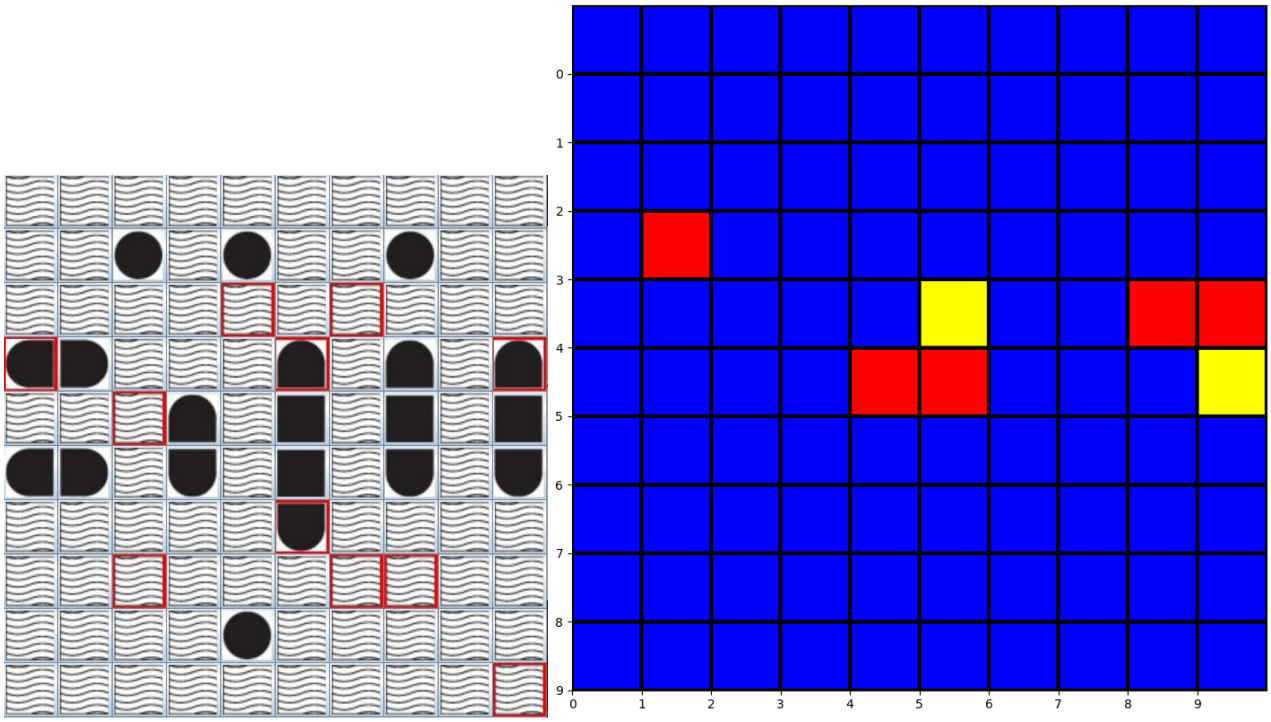
GUESS_OK: 0

GUESS_KO: 0

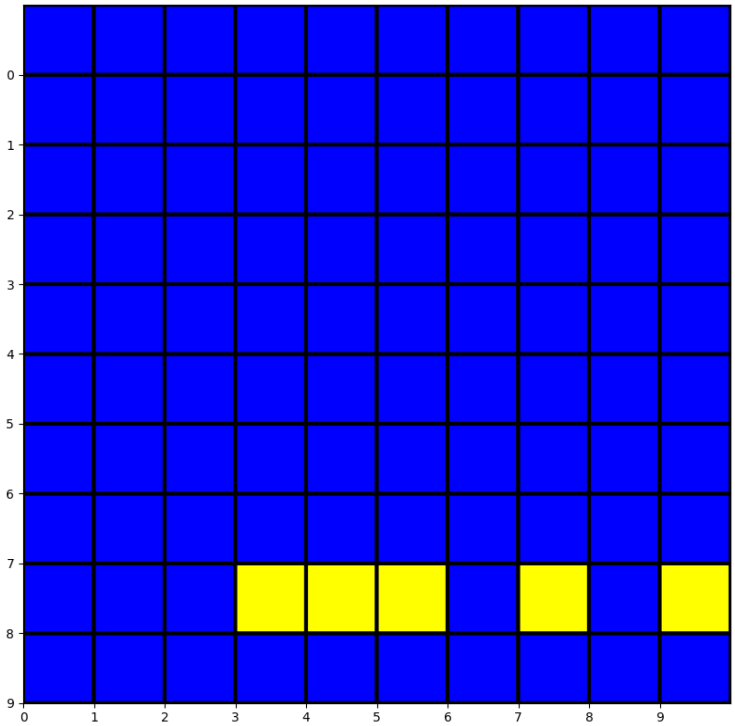
SAFE: 19

SINK: 1

3.2.2 Test B



SCORE: -65 FIRE_OK:3 FIRE_KO:2 GUESS_OK:2 GUESS_KO:0 SAFE:11 SINK:3



Versione senza conoscenza iniziale

SCORE: -265

FIRE_OK: 1

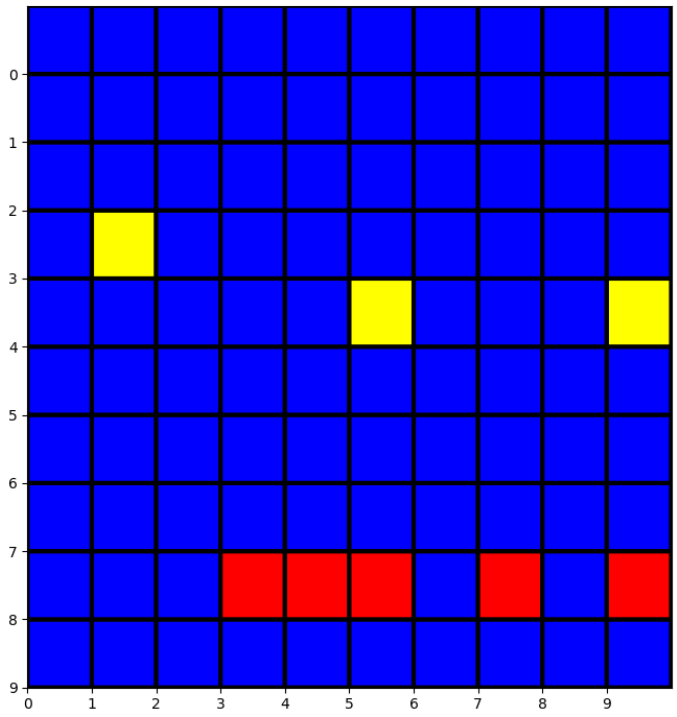
FIRE_KO: 4

GUESS_OK: 0

GUESS_KO: 0

SAFE: 19

SINK: 1



Versione con conoscenza – Algo2

SCORE: -150

FIRE_OK: 1

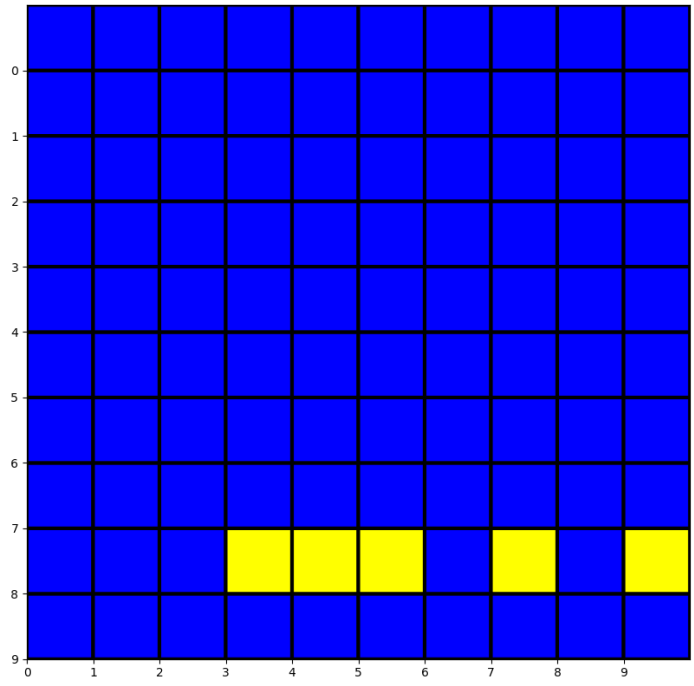
FIRE_KO: 4

GUESS_OK: 3

GUESS_KO: 0

SAFE: 12

SINK: 2



Versione senza conoscenza – Algo2

SCORE: -265

FIRE_OK: 1

FIRE_KO: 4

GUESS_OK: 0

GUESS_KO: 0

SAFE: 19

SINK: 1

3.2.3 Risultati

*la colonna *Caso* è da leggere in questo modo: prima lettera indica il test (A o B), la seconda indica l'algoritmo (1 o 2) e l'ultima indica C=conoscenza iniziale, N=non conoscenza iniziale

Caso	Score	Fire ok	Fire ko	Guess ok	Guess ko	Safe	Sink
A – 1 – C	100	5	0	2	0	6	6
A – 1 – N	-265	1	4	0	0	19	1
B – 1 – C	-65	3	2	2	0	11	3
B – 1 – N	-265	1	4	0	0	19	1
A – 2 – C	25	2	3	5	0	6	6
A – 2 – N	-265	1	4	0	0	19	1
B – 2 – C	-150	1	4	3	0	12	2
B – 2 – N	-265	1	4	0	0	19	1

Dalla tabella riassuntiva possiamo notare come i limiti del sistema siano evidenti nelle due mappe utilizzate. L'agente che da lo score medio più alto è la prima versione, notiamo però che il miglioramento è percepibile solo per i casi in cui c'è conoscenza iniziale, negli altri il punteggio non cambia tra le due implementazioni.

In particolare si può notare come le prestazioni in caso di non conoscenza iniziale siano particolarmente basse, questo è probabilmente spiegabile dal fatto che la strategia adottata dall'agente è sempre quella di effettuare delle fire pseudo-random con la speranza di acquisire maggiore conoscenza nello step successivo.

4.Reti Bayesiane

4.1 Reti Bayesiane statiche

L'obiettivo di questa sperimentazione è quello di comparare l'algoritmo di Variable Elimination visto a lezione utilizzandolo su diverse reti e con diverse query.

4.1.1 Implementazione

Per implementare l'algoritmo ho utilizzato Java8 e la libreria aima-core.

Per la parsificazione delle reti ho utilizzato il progetto bnParser presentato a lezione.

Per implementare la parte relativa ai diversi ordinamenti e al pruning della rete ho direttamente modificato la libreria aima per la quale avevo precedentemente importato direttamente i sorgenti.

Ho testato la correttezza delle inferenze confrontando le distribuzioni restituite dal mio script con quelle restituite da SamIam.

Questi sono i macro step che ho seguito:

1. Ho scaricato e scelto alcune reti di interesse dal sito https://www.bnlearn.com/bnrepository/ importandole prima in SamIam e successivamente direttamente nel mio progetto (in formato xmlbif)

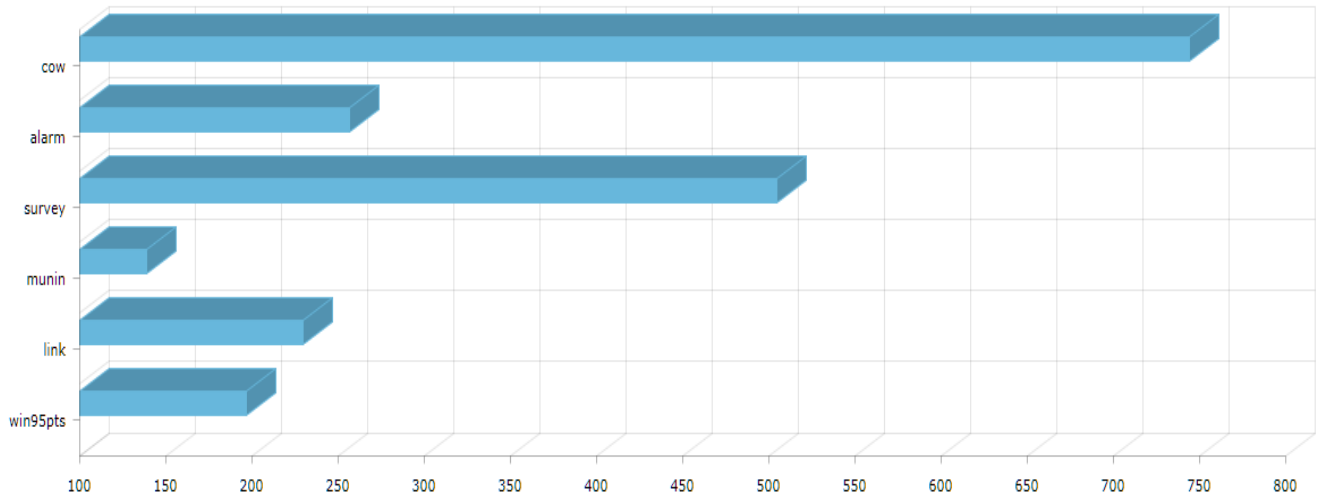
2. Ad ogni rete sono stati poi applicati tre algoritmi di pruning in sequenza:
    a. Pruning dei nodi irrilevanti secondo il Teorema1 visto a lezione (non appartenenti agli ancestors di {X} U {E}
    b. Pruning dei nodi irrilevanti secondo il Teorema2 visto a lezione (m-separation)
    c. Pruning degli archi irrilevanti secondo il teorema 6.5 visto a lezione
3. Per ogni rete ho poi eseguito l'algoritmo di variable elimination utilizzando tre diversi tipi di ordinamento delle variabili:
    a. Topologico inverso
    b. MinDegree
    c. MinFill
4. Ho infine salvato i tempi di ogni computazione(nanosecondi) della variable elimination in un file di output con il quale ho poi generato delle statistiche e dei grafici

4.1.2 Benchmark e risultati

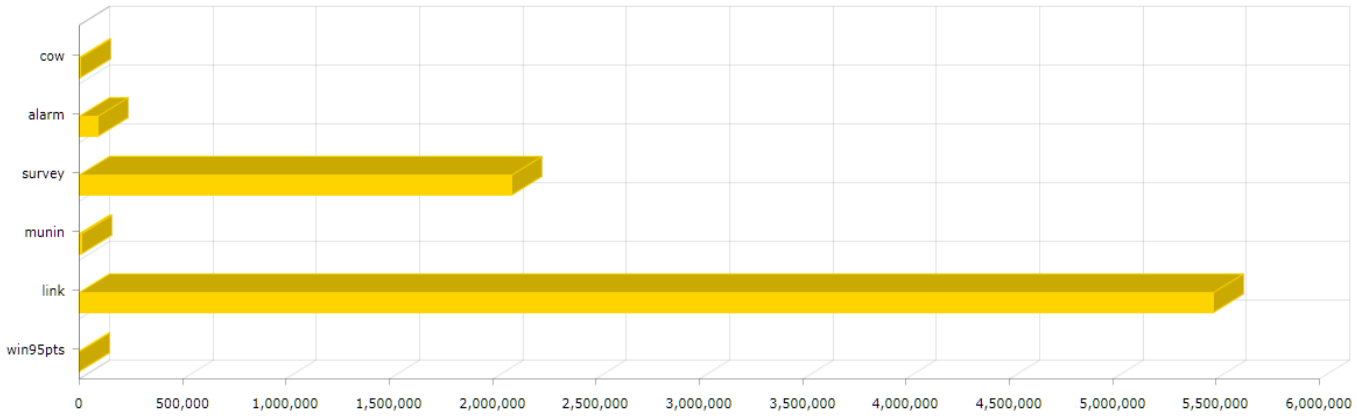
Questi sono i test che ho effettuato e i risultati che ho ottenuto (evidenzio in rosso il peggior risultato di un esperimento).

#	Query variables	Evidence	Network	Order	Computational time (nanoseconds)
1.1	Pregnancy	Blood = true	Cow.xml	Inverse topological	640400
1.2	Pregnancy	Blood = true	Cow.xml	MinDegree	7672701
1.3	Pregnancy	Blood = true	Cow.xml	MinFill	641200
1.4	Pregnancy	Blood = true Scan = false	Cow.xml	Inverse topological	787000
1.5	Pregnancy	Blood = true Scan = false	Cow.xml	MinDegree	1573300
1.6	Pregnancy	Blood = true Scan = false	Cow.xml	MinFill	672501
2.1	S	T="Car"	Survey.xml	Inverse topological	2033900
2.2	S	T="Car"	Survey.xml	MinDegree	4200300
2.3	S	T="Car"	Survey.xml	MinFill	5642199
2.4	A	O="Emp"	Survey.xml	Inverse topological	1360700
2.5	A	O="Emp"	Survey.xml	MinDegree	670600
2.6	A	O="Emp"	Survey.xml	MinFill	591401
3.1	SHUNT	HR="LOW"	Alarm.xml	Inverse topological	61100200
3.2	SHUNT	HR="LOW"	Alarm.xml	MinDegree	18473099
3.3	SHUNT	HR="LOW"	Alarm.xml	MinFill	11343600
3.4	MINVOLSET	VENTTUBE="LOW"	Alarm.xml	Inverse topological	356000
3.5	MINVOLSET	VENTTUBE="LOW"	Alarm.xml	MinDegree	469799
3.6	MINVOLSET	VENTTUBE="LOW"	Alarm.xml	MinFill	330500
3.7	MINVOLSET	VENTLUNG="ZERO" VENTTUBE="LOW"	Alarm.xml	Inverse topological	1234300
3.8	MINVOLSET	VENTLUNG="ZERO" VENTTUBE="LOW"	Alarm.xml	MinDegree	1206100
3.9	MINVOLSET	VENTLUNG="ZERO"	Alarm.xml	MinFill	1037101

		VENTTUBE=”LOW”			
4.1	DIFFN_DISTR	DIFFN_M_SEV_DIST=”NO”	Munin.xml	Inverse topological	920400
4.2	DIFFN_DISTR	DIFFN_M_SEV_DIST=”NO”	Munin.xml	MinDegree	892200
4.3	DIFFN_DISTR	DIFFN_M_SEV_DIST=”NO”	Munin.xml	MinFill	293501
4.4	DIFFN_DISTR	DIFFN_DUMMY_1=”Dummy”	Munin.xml	Inverse topological	4340000
4.5	DIFFN_DISTR	DIFFN_DUMMY_1=”Dummy”	Munin.xml	MinDegree	681199
4.6	DIFFN_DISTR	DIFFN_DUMMY_1=”Dummy”	Munin.xml	MinFill	636100
4.7	DIFFN_DISTR	DIFFN_TYPE=”Sens”	Munin.xml	Inverse topological	214300
4.8	DIFFN_DISTR	DIFFN_TYPE=”Sens”	Munin.xml	MinDegree	348000
4.9	DIFFN_DISTR	DIFFN_TYPE=”Sens”	Munin.xml	MinFill	88100
4.10	DIFFN_DISTR R_ULN_LD_EW	DIFFN_TYPE=”Sens”	Munin.xml	Inverse topological	151400
4.11	DIFFN_DISTR R_ULN_LD_EW	DIFFN_TYPE=”Sens”	Munin.xml	MinDegree	711700
4.12	DIFFN_DISTR R_ULN_LD_EW	DIFFN_TYPE=”Sens”	Munin.xml	MinFill	158801
5.1	Z_2_a_f	D0_10_d_p=”A”	Link.xml	Inverse topological	16611994500
5.2	Z_2_a_f	D0_10_d_p=”A”	Link.xml	MinDegree	1109199
5.3	Z_2_a_f	D0_10_d_p=”A”	Link.xml	MinFill	1400001
5.4	Z_2_a_f	Z_2_a_m=”F”	Link.xml	Inverse topological	235100
5.5	Z_2_a_f	Z_2_a_m=”F”	Link.xml	MinDegree	192501
5.6	Z_2_a_f	Z_2_a_m=”F”	Link.xml	MinFill	135600
6.1	PrtCbl	TnrSpply=”Low”	Win95pts.xml	Inverse topological	51500
6.2	PrtCbl	TnrSpply=”Low”	Win95pts.xml	MinDegree	67800
6.3	PrtCbl	TnrSpply=”Low”	Win95pts.xml	MinFill	63500
6.4	NtSpd	PrtDriver=”Yes”	Win95pts.xml	Inverse topological	498100
6.5	NtSpd	PrtDriver=”Yes”	Win95pts.xml	MinDegree	219100
6.6	NtSpd	PrtDriver=”Yes”	Win95pts.xml	MinFill	234200



Migliori tempi di computazione per rete divisi per 1000



*Peggiori tempi di computazione per rete (*rete link troncata a 5500000) divisi per 1000*

4.2 Reti Bayesiane dinamiche

L'obiettivo di questa sperimentazione è quello di implementare e testare l'algoritmo di rollup filtering per le reti Basyesiane dinamiche.

5.1.1 Implementazione

L'implementazione si basa sull'utilizzo della libreria aima-core ed in particolare usando la classe di base DynamicBayesianNetwork (che ho esteso tramite la mia classe IalabDBN) per la rappresentazione di una DBN.

La classe IalabDBN estende il concetto di DynamicBayesianNetwork offrendo anche la possibilità di utilizzare il rollup filtering tramite multiple chiamate al metodo *forward* che permette di avanzare al prossimo slice della rete.

Il metodo forward utilizza una versione modificata del metodo di Variable Elimination che ritorna i factors calcolati invece di moltiplicarli, in questo modo possono essere riutilizzati nell'esecuzione sul prossimo slice (mettendoli nella posizione più a destra nella formula di sumout di VE).

5.1.2 Benchmark e risultati

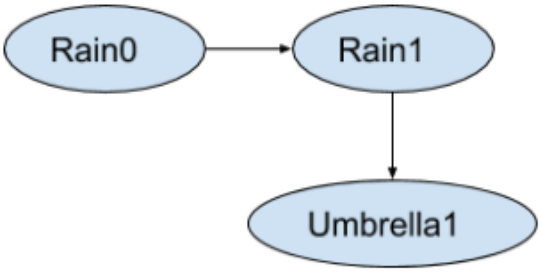
Ho effettuato i test usando 5 DBN diverse.

Per ogni esperimento ho definito le evidenze, le query variables e l'ordinamento utilizzato dall'algoritmo di VE.

Per verificare la correttezza delle inferenze ho poi utilizzato l'algoritmo di Particle Filtering controllando che i risultati fossero almeno simili.

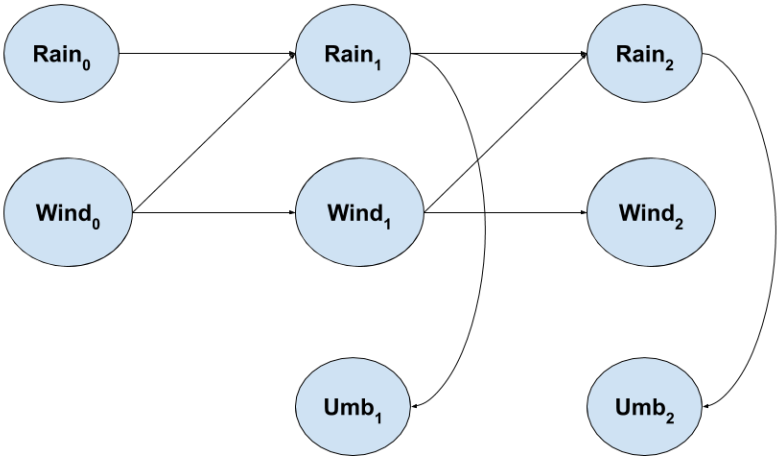
Questi sono i risultati ottenuti.

(* Inv.Top. = Inverse topological; Fill=Elimination fill; MinD=Elimination Min Degree)



- **Variabili di evidenza**: Umbrella (U)
- **Variabili di stato**: Rain (R)

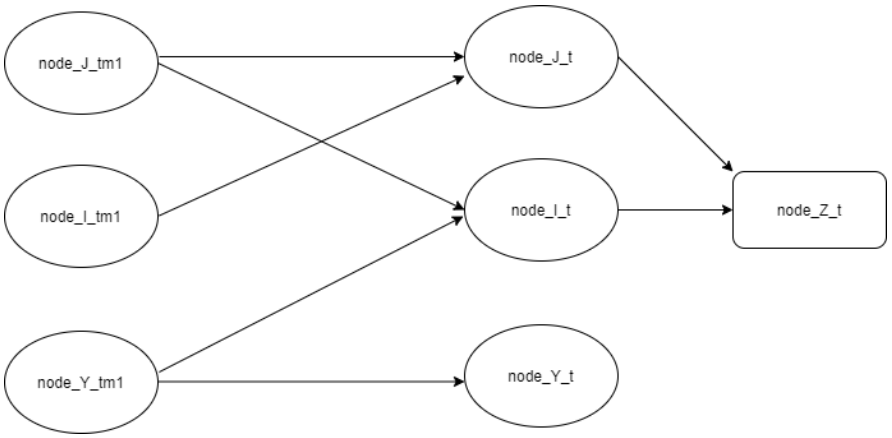
Query	E - t0	E – t1	E – t2	E – t3	E – t4	VE ord.	Time
Rain_t	U=T	U=T				Inv.Top.	4293500
Rain_t	U=T	U=T				Fill	3649201
Rain_t	U=T	U=T				MinD	273800
Rain_t	U=F	U=T	U=T	U=T	U=F	Inv.Top.	452600
Rain_t	U=T	U=T	U=T	U=T	U=F	Fill	1208000
Rain_t	U=T	U=T	U=T	U=T	U=F	MinD	456800



Variabili di evidenza: Umbrella

Variabili di stato: Rain, Wind

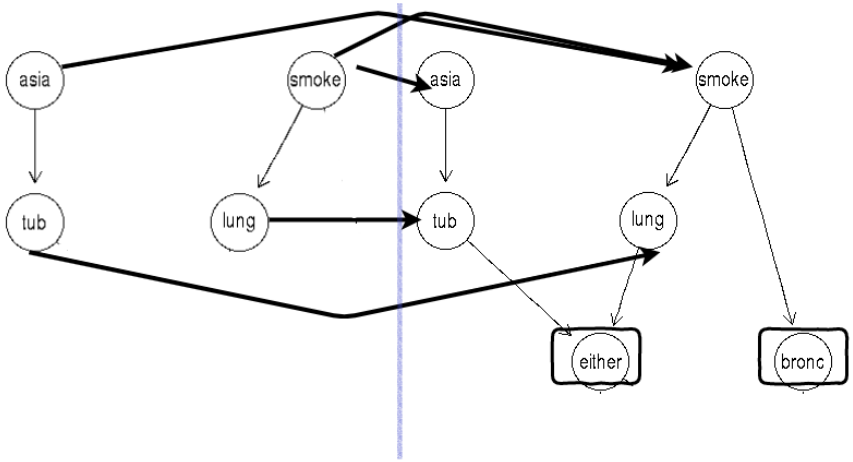
Query	E - t0	E – t1	VE ord.	Time
Rain, Wind	U=T	U=T	Inv.Top.	231300
Rain, Wind	U=T	U=T	Fill	553500
Rain, Wind	U=T	U=T	MinD	1715701



Variabili di evidenza: Z

Variabili di stato: I, J, Y

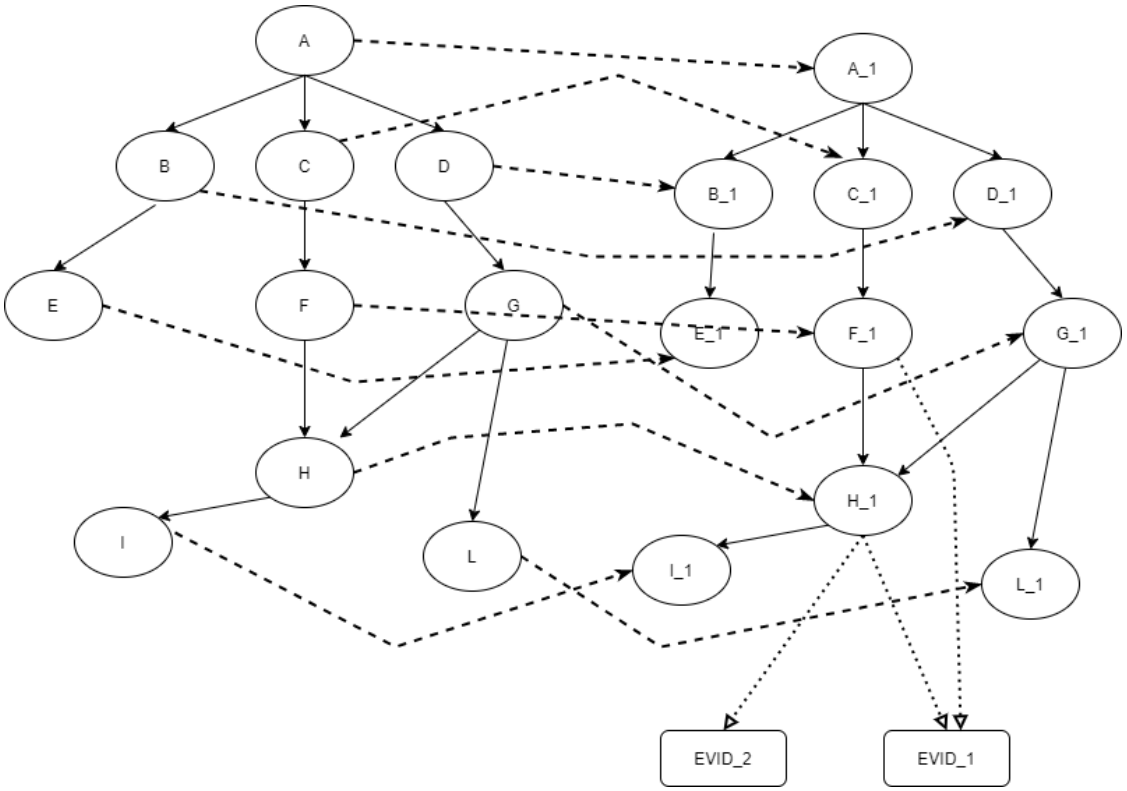
Query	E - t0..n	VE ord.	Time
Y	Z=T|Z=F	Inv.Top.	1330199
Y	Z=T|Z=F	Fill	438300
Y	Z=T|Z=F	MinD	324199
I	Z=T|Z=F|Z=F|Z=T| Z=F| Z=F| Z=F| Z=F| Z=T	Inv.Top.	864099
I	Z=T|Z=F|Z=F|Z=T| Z=F| Z=F| Z=F| Z=F| Z=T	Fill	2243700
I	Z=T|Z=F|Z=F|Z=T| Z=F| Z=F| Z=F| Z=F| Z=T	MinD	4432101



Variabili di evidenza:
Either (E), Bronc (B)

Variabili di stato: Asia, Lung, Smoke, Tub

Query	E - t0	E – t1	VE ord.	Time
Asia	E=T; B=F	E=T; B=T	Inv.Top.	786700
Asia	E=T; B=F	E=T; B=T	Fill	817100
Asia	E=T; B=F	E=T; B=T	MinD	745300



Variabili di evidenza: Evid1 (E1), Evid2 (E2)

Variabili di stato: A, B, C, D, E, F, G, H, I, L

Query	E - t0	E – t1	VE ord.	Time
B	E1=T; E2=T	E1=F; E2=T	Inv.Top.	18720900
B	E1=T; E2=T	E1=F; E2=T	Fill	90727800
B	E1=T; E2=T	E1=F; E2=T	MinD	92899200