I esercitazione

NOTA BENE: non leggere questa soluzione senza aver provato prima autonomamente l'esercizio a partire dal testo dell'esercizio nel file "es1_teseo-testo*"

Sommario:

- Agenti e ambienti
  - Teseo e il labirinto versione base: formulazione PEAS
- Agenti per il problem solving
  - Teseo e il labirinto versione base: formulazione con ambiente osservabile
- Algoritmi di ricerca euristica

1. Teseo e il labirinto

Caso base: Teseo con mappa
Assunzioni:
a. Teseo ha una mappa e sa dove si trova (ambiente accessibile)
b. Teseo può spostarsi in una casella adiacente se non c'è un muro nel mezzo (le azioni sono deterministiche)
c. Lo stato obiettivo è chiaramente indicato sulla mappa e riconoscibile (goal test)

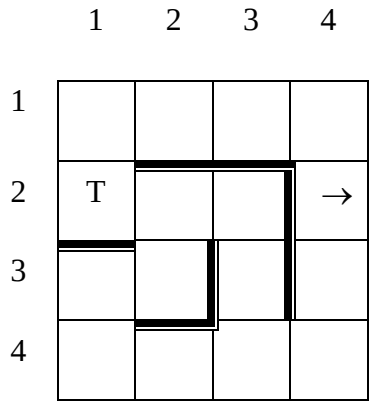
Formulazione PEAS

P-Performance
E-Environment
A-Actions|Attuatori
S-Sensors |Percezioni

Misura di Prestazioni: +1 per ogni spostamento (premiato l'agente che esce prima dal labirinto su una media di labirinti diversi, i.e. costo del cammino).
Ambiente: labirinto
Azioni: ↑ → ↓ ← (spostamenti/mosse di una casella in orizzontale e verticale)
Percezioni: mappa, posizione di sé stesso e dell'uscita

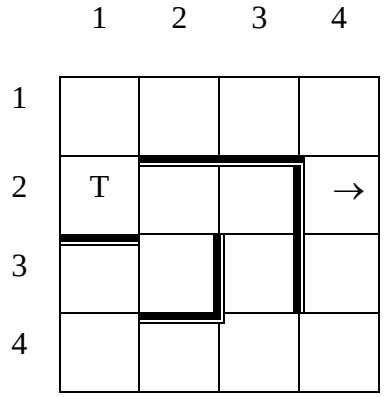
Tipo di ambiente

Statico (alternative: dinamico, semidinamico)
Deterministico (alternative: stocastico, strategico)
Osservabile (alternative: parzialmente osservabile)
Discreto (alternativa: continuo)
Non episodico (alternativa: episodico)
Mono agente (alternativa: multi-agente)



Tipo di agente

Reattivo? No
Con stato? Si
Con obiettivo? Si
Con funzione di utilità? No



Siamo nell'ambito degli **Agenti per il Problem solving** che sono in grado di pianificare una intera sequenza di mosse prima di agire. I passi sono:

- Formulazione del problema come un problema di ricerca
- Ricerca della soluzione nello spazio degli stati

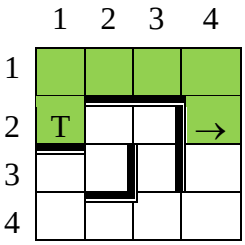
Formulazione del problema

Stati: (R, C) coordinate nella griglia (Riga, Colonna)
Stato iniziale: (2,1)
Goal-Test: (2, 4)?
Azioni: $\uparrow \rightarrow \downarrow \leftarrow$ (spostamenti/mosse di una casella in orizzontale e verticale)
Azioni((R, C)) $\subseteq \{\uparrow \rightarrow \downarrow \leftarrow\}$
Modello di transizione:
Un solo stato successore.
Risultato((R, C) , $\uparrow$) = (R-1, C), purché non ci sia un muro*
Risultato((R, C) , $\rightarrow$) = (R, C+1), purché non ci sia un muro*
Risultato((R, C) , $\downarrow$) = (R+1, C), purché non ci sia un muro*
Risultato((R, C) , $\leftarrow$) = (R, C-1), purché non ci sia un muro*
Costo azione: 1
Costo cammino: somma costo dei passi/spostamenti (di tutte le azioni)

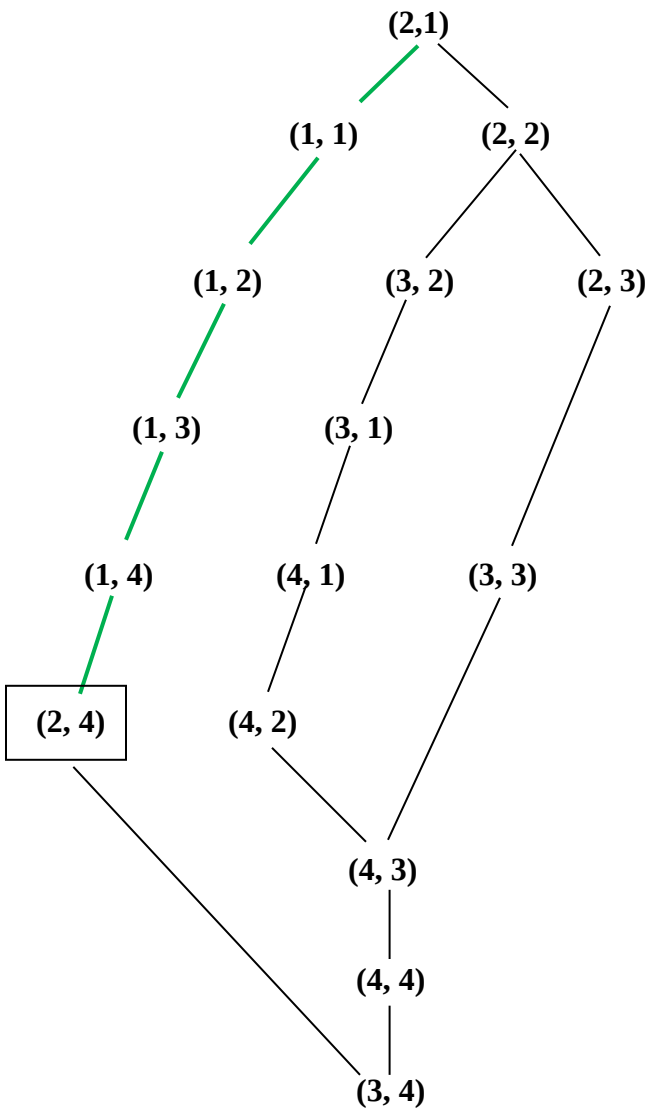
(*) Nel codice Python per realizzare questo ci sono rimozioni dalla lista azioni.

NOTA: gli script Python sono nel file "*esercitazione1.py*" divisi in EX.1 (ESERCIZIO 1) e EX.2 (ESERCIZIO 2) nel seguito (in base all'ordinamento delle azioni).
EX1. È svolto qui (ordine $\uparrow \rightarrow \downarrow \leftarrow$ o suo inverso).
EX2. È lasciato a voi come esercizio ulteriore (ordine $\uparrow \downarrow \rightarrow \leftarrow$ o suo inverso).

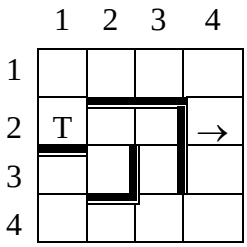
NOTA: come detto nella lezione 1 (slide <Esercitazioni>), il codice Python è di ausilio aggiuntivo agli esercizi, non è "obbligatorio" per l'esame. Permette una autoverifica (esercizi, anche diversi da questi cambiando alcuni elementi per verificare la vostra soluzione con quella del codice) e di stimolo per chi voglia andare oltre implementando altri casi etc. Si suggerisce di provare comunque su carta la soluzione e usare le soluzioni Python come controllo, anche in condizioni diverse (e.g. diverso stato iniziale, goal, muri, …). Esercitarsi anche alla velocità (vedi esempio il caso A* a seguire) vi aiuta poi a svolgere gli esercizi con confidenza in breve tempo.



Spazio stati:

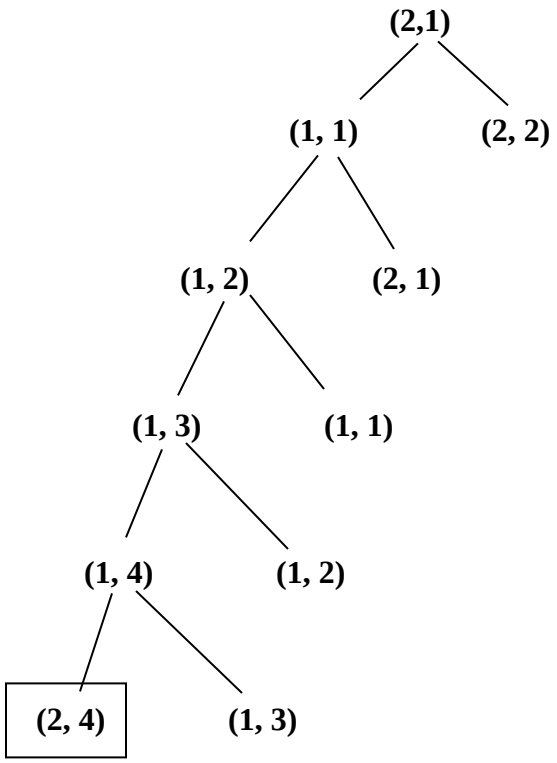


Gli archi sono tutti nei due sensi. Quindi ci sono cicli.
Il cammino ottimo è segnato in verde. Il costo del cammino ottimo è 5.
Faq: si noti che ogni stato compare una sola volta.

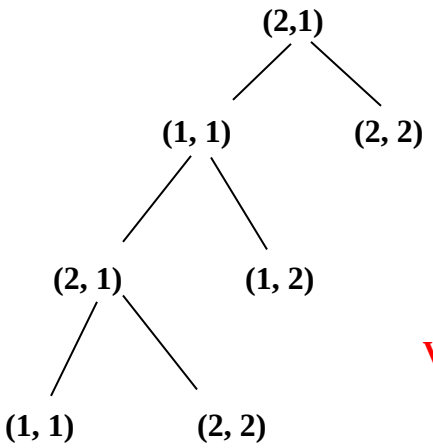


Ricerca in profondità con l'ordinamento ↑ → ↓ ←

Se si assume di avere questo ordine delle mosse sullo stesso livello di profondità [inserimento in reverse (lifo) sullo stesso livello, tra i figli].
N.B. non c'è in Python!!! (ma è quanto visto su slide del corso/AIMA su albero A,B, C,….)

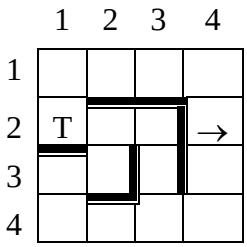


Con un diverso ordinamento delle azioni: ↑ ↓ → ←



Va in loop

Ecc.



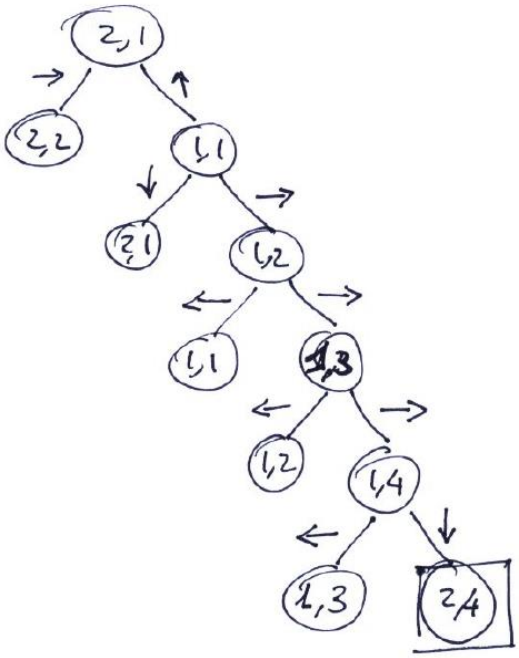
Ricerca in profondità con l'ordinamento ↑ → ↓ ←

Se si assume di avere ordine LIFO (come alg. in Python) l'esecuzione va in ciclo su caselle (2,1) e (2,2) . **Va in loop** (da testo è DF ad albero)

Python: ToyProblems.py --> classe Labirinto_Teseo di base (chiamato teseo1 in "*esercitazione1.py*")

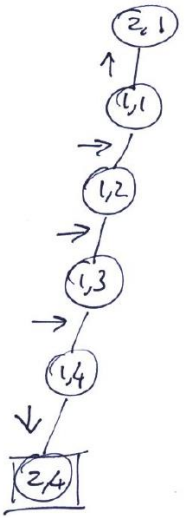
Ricerca in profondità con l'ordinamento ←↓→↑ (*inverso* rispetto a prima)

Ordine LIFO (di estrazione, append per inserimento, come alg. in Python), ad albero:



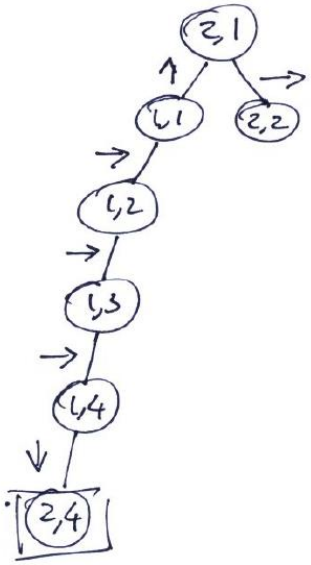
Soluzione EX.1 con depth_first_search_tree e actions = ['LEFT','DOWN','RIGHT','UP']
Lista delle azioni nel cammino: ['UP', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']
Lista degli stati nel cammino: ['(2,1)', '(1,1)', '(1,2)', '(1,3)', '(1,4)', '(2,4)']
Costo della soluzione: 5

Ricerca in profondità <u>Ricorsiva</u> con l'ordinamento ↑ → ↓ ←



Soluzione EX.1 con recursive_depth_first_search e actions = ['UP','RIGHT','DOWN','LEFT']
Lista delle azioni nel cammino: ['UP', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']
Lista degli stati nel cammino: ['(2,1)', '(1,1)', '(1,2)', '(1,3)', '(1,4)', '(2,4)']
Costo della soluzione: 5

Ricerca in profondità con "<u>schema</u> algoritmo di A" (*f* = - node.depth, su/a grafo) con l'ordinamento ↑ → ↓ ←
Estrazione FIFO da coda con priorità in base ad *f*



Soluzione EX.1 con astar_search e actions = ['UP','RIGHT','DOWN','LEFT']
Lista delle azioni nel cammino: ['UP', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']
Lista degli stati nel cammino: ['(2,1)', '(1,1)', '(1,2)', '(1,3)', '(1,4)', '(2,4)']
Stati esplorati: ['(2,1)', '(1,1)', '(1,2)', '(1,3)', '(1,4)']
Costo della soluzione: 5

Conclusioni:
DF non è completa, non è ottima come visto dagli esempi. Infatti, la soluzione dipende dall'ordine fissato per le mosse e dall'algoritmo di sorting della coda (vedi esempi sopra); si può trovare lo stato goal o meno, si può avere un costo ottimo o meno, ma *non vale in generale*. Trovare il goal per uno specifico ordinamento non implica trovarlo in generale. Altri esempi di cambio ordine: provateli.

Ma NON cambiano le proprietà generali: costo in tempo e memoria, completezza, ottimalità. Questi sono i concetti importanti che fanno la differenza ai fini dell'uso in AI. Ecco perché ci focalizziamo sul provare le proprietà in generale a prescindere da specifici ordinamenti delle azioni o di preferenza a pari condizioni (livello, valore di *f*, …)

D'altra parte NON si potrebbe scegliere un algortitmo di sorting/ordine azioni *ad hoc* per ottimizzare il problema poiché alla creazione dell'algoritmo non si conosce il problema e mi devo basare sull'analisi generale che mi da garanzie di costo/ottimalità etc. in qualsiasi caso (quindi con analisi al caso pessimo).

DF a profondità limitata (DL):

Completa se con giusto limite (Ottimale? *No*). Esempi di prove dell'algoritmo:

Soluzione EX.2 con limited depth_first_search_tree, actions =
['LEFT','RIGHT','DOWN','UP'], **max_depth = 5**
Lista delle azioni nel cammino: ['UP', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']
Lista degli stati nel cammino: ['(2,1)', '(1,1)', '(1,2)', '(1,3)', '(1,4)', '(2,4)']
Costo della soluzione: 5

Soluzione EX.1 con limited depth_first_search_tree, actions
=['UP','RIGHT','DOWN','LEFT'], **max_depth = 5**
Lista delle azioni: ['UP', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']
Lista degli stati: ['(2,1)', '(1,1)', '(1,2)', '(1,3)', '(1,4)', '(2,4)']
Costo della soluzione: 5
Nota: quindi non più ciclo come prima per DF (con questo ordinamento di azioni ↑→↓←)

Provate con profondità limite diversa (ad esempio 10).

Soluzione EX.1 con limited depth_first_search_tree, actions =
['UP','RIGHT','DOWN','LEFT'], **max_depth = 10**
Lista delle azioni: ['RIGHT', 'LEFT', 'RIGHT', 'LEFT', 'UP', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']
Lista degli stati: ['(2,1)', '(2,2)', '(2,1)', '(2,2)', '(2,1)', '(1,1)', '(1,2)', '(1,3)', '(1,4)', '(2,4)']
Costo della soluzione: 9

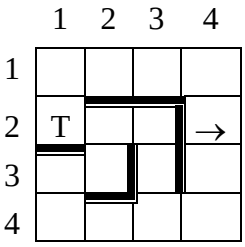
Approfondimento iterativo. Completo? Si. **Ottimale?** Si.

Ricerca euristica

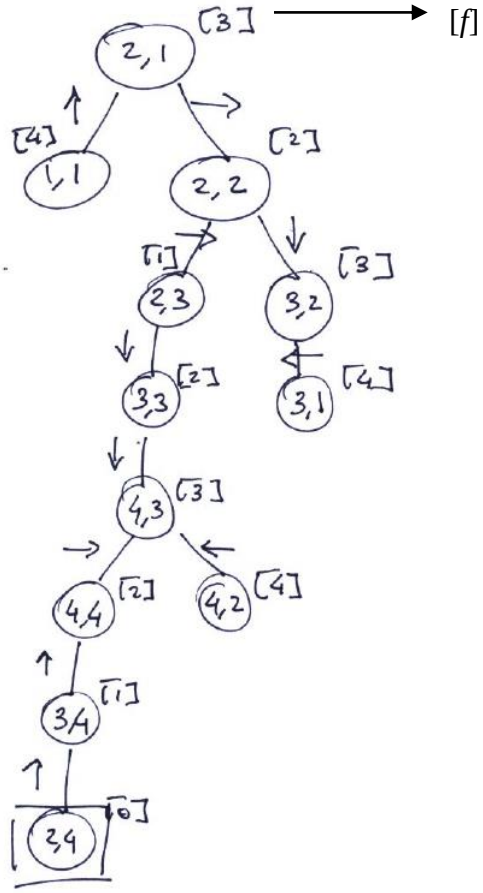
Supponiamo ora di avere un modo di valutare gli stati. Avendo la mappa e sapendo dove si trova l'uscita ....
Quale potrebbe essere una buona funzione di valutazione euristica?
Stima della distanza dalla soluzione com Manhattan Distance (MD)

$$h((R, C)) = MD((R, C), (R_g, C_g)) = |R - R_g| + |C - C_g|$$

[in alternativa la distanza in linea d'aria, ma è meno informata]



Greedy Best First (a grafo): un Best first con f(n) = h(n)
Con l'ordinamento ↑ → ↓ ←



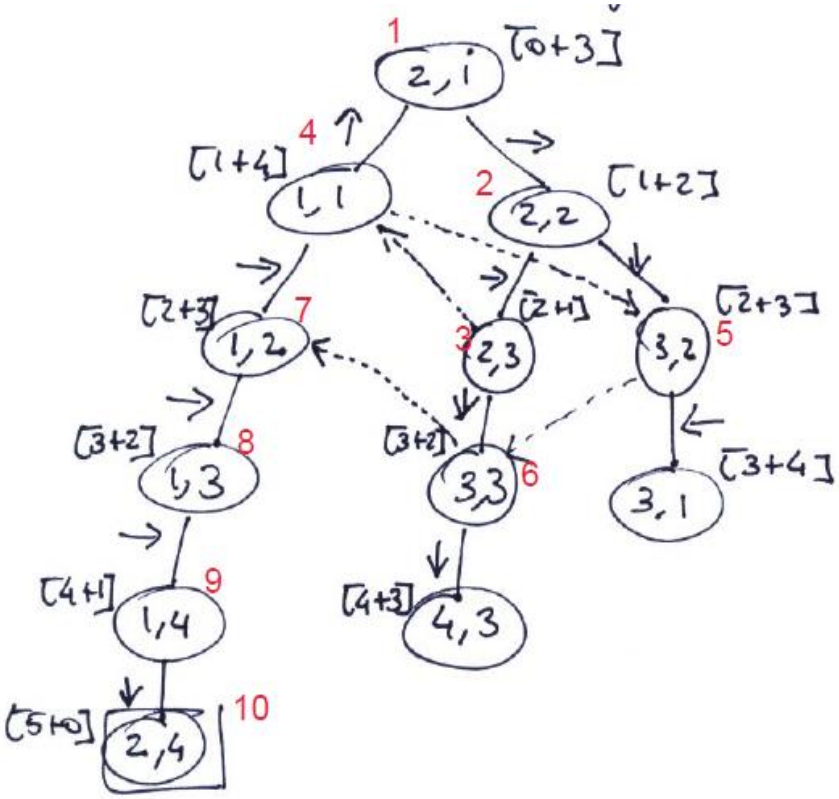
FIFO tra pari valori di *f*

Soluzione EX.1 con astar_search, actions = ['UP','RIGHT','DOWN','LEFT'], f = h
Lista delle azioni: ['RIGHT', 'RIGHT', 'DOWN', 'DOWN', 'RIGHT', 'UP', 'UP']
Lista degli stati: ['(2,1)', '(2,2)', '(2,3)', '(3,3)', '(4,3)', '(4,4)', '(3,4)', '(2,4)']
Stati esplorati: ['(2,1)', '(2,2)', '(2,3)', '(3,3)', '(3,2)', '(4,3)', '(4,4)', '(3,4)']
Costo della soluzione: 7

Commento: «Su grafi» completa per spazi finiti ma non ottimale: qui l'esempio di costo 7

Algoritmo A: Best first con f(n) = g(n) + h(n) (sempre con ordine ↑ → ↓ ←)

Nel caso la strategia non indichi una preferenza si segue una politica FIFO (tra pari valori di *f*) (come da testo). Caso ricerca "a grafo" (vedi testo):



Nota: ordine FIFO tra f di pari valore ("salti" tra i nodi dell'albero - - - >)

Numeri in rosso per ordine di esplorazione (vedi anche lista sotto degli Stati esplorati)

Soluzione EX.1 con astar_search, actions = ['UP','RIGHT','DOWN','LEFT'], f = g+h
Lista delle azioni: ['UP', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']
Lista degli stati: ['(2,1)', '(1,1)', '(1,2)', '(1,3)', '(1,4)', '(2,4)']
Stati esplorati: ['(2,1)', '(2,2)', '(2,3)', '(1,1)', '(3,2)', '(3,3)', '(1,2)', '(1,3)', '(1,4)']
Costo della soluzione: 5

Commenti: Trova il cammino soluzione: (2,1) (1,1) (1,2)(1,3)(1,4)(2,4) di costo 5
(o il piano [*su, destra, destra, destra, giù*]).

Trova sempre una soluzione? SI. Completo anche nel caso di ricerca-albero.
Possiamo dire in generale che trova sempre la soluzione migliore? Si perché l'euristica è ammissibile.

Ammissibile: per ogni n $h(n) \leq h^*(n)$

Perché è ammissibile (sottostima)?

Se il nostro agente fosse in grado di passare attraverso i muri (problema meno vincolato) in un numero di passi pari alla MD arriverebbe alla soluzione, ma purtroppo non lo può fare. Quindi ci mette sempre un numero di passi maggiore, o almeno uguale.

È anche monotona (consistente): riflettere sul come (con le possibili mosse e euristica MD)

Monotona: $h(n) \leq h(n') + c(n, a, n')$ ovvero : $h(n) - h(n') \leq c(n, a, n')$

La *f* cresce lungo tutti i cammini.

Quindi di fatto abbiamo usato un algoritmo A*.
Ci garantisce l'ottimalità anche evitando di rivisitare gli stati già visitati (usando graph search) perché espande i nodi in ordine di f crescente e quando arriva al nodo goal siamo sicuri di avere già incontrato e testato tutti i nodi con f minore.

Ricerca locale (come ricerca dell'uscita)

Classe Labirinto_Teseo_Local
Usando **Hill climbing** e per valore della funzione obiettivo la -MD (perché si massimizza, quindi come minimizzare MD a zero)

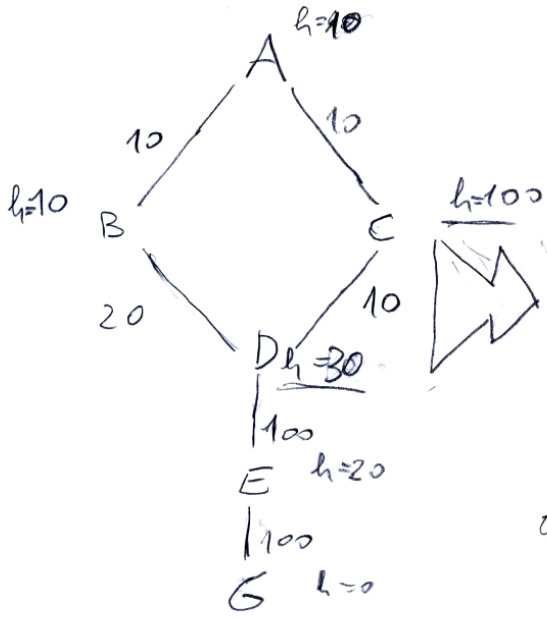
Si percorre (2,1) → (2,2) → (2,3) (poiché il (3,3) non migliora il problem.value)
E si ferma in massimo locale in (2,3) (angolo del muro)

In PYTHON: stampa solo (2,3) (sempre che non si vogliano inserire dei "print" lungo il percorso, nella zona dell'istruzione "current=neighbord")

Stocastica ce la può fare; ad esempio con **Simulated annealing**: in effetti provando temperatura iniziale (k) =100 inziale e con T che decade in modo esponenziale (come descritto nei commenti del codice), trova una soluzione (ma non ottima) in, a esempio, 27 mosse (o anche minori), ma dopo un po' di ripartenze.

Controesempio: euristica non consistente

Perché serve questa proprietà più forte di monotonia (e quindi crescita della f lungo tutti i cammini)? Riusciamo a immaginarci un caso in cui la *h* è ammissibile ma con graph search non trovo la soluzione ottimale? (algoritmo ricerca-grafo con A)

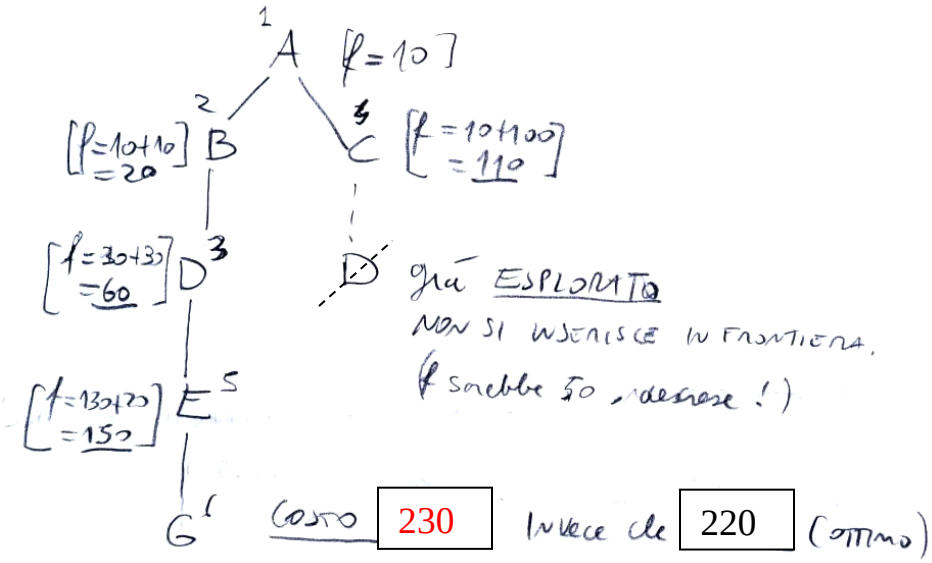


Euristica consistente**?**

$h(C) \leq 10 + h(D)$ **?**

NO h cade troppo bruscamente, e.g. h(C) troppo alta

OTTIMO: A C D E G
COSTO: 220



Dove era il problema? Nell'avere espanso D prima di C: se l'euristica fosse stata monotona non decrescente, C (con f così più bassa di almeno 60, cioè con valori di h sotto 40 e di f da 50 in giù) sarebbe espanso prima del nodo D a sinistra (con f=60) e non avremmo trovato D come già presente in lista esplorati su un path non ottimale. Così l'altro D, con f da 50 (20+30) in giù, entrerebbe in frontiera, il primo D espanso sarebbe quello sul cammino migliore (l'altro può essere non considerato con il controllo su esplorati), trovando alla fine la soluzione ottima!
Warning: Ed IV AIMA: con il nuovo schema di alg. di A* si troverebbe l'ottimo anche in questo caso (al costo di più possibili nodi ridondanti e sub-ottimali in frontiera) .