**Academic Year:** 2023/2024

**Course:** Artificial Intelligent Systems - Intelligent Model Project

**Professor:** Stefano Marcugini

**Ongoing assignmen**t: Intelligent Application Development

**Student:** Lorenzo Mariotti

**ID:** 369094

Sommario

[1. Problema salto del cavallo pigro 3](#_Toc157956468)

[1.1. Enunciato 3](#_Toc157956469)

[1.2. Analisi 3](#_Toc157956470)

[2. Soluzione 4](#_Toc157956471)

[2.1. Logica dell’algoritmo 4](#_Toc157956472)

[2.2. Implementazione 5](#_Toc157956473)

[2.2.1. Tipi 5](#_Toc157956474)

[2.2.2. Funzione successori 6](#_Toc157956475)

[2.2.3. Funzione obiettivo 7](#_Toc157956476)

[2.2.4. Risoluzione Breadth First Search 8](#_Toc157956477)

# Problema salto del cavallo pigro

# Enunciato

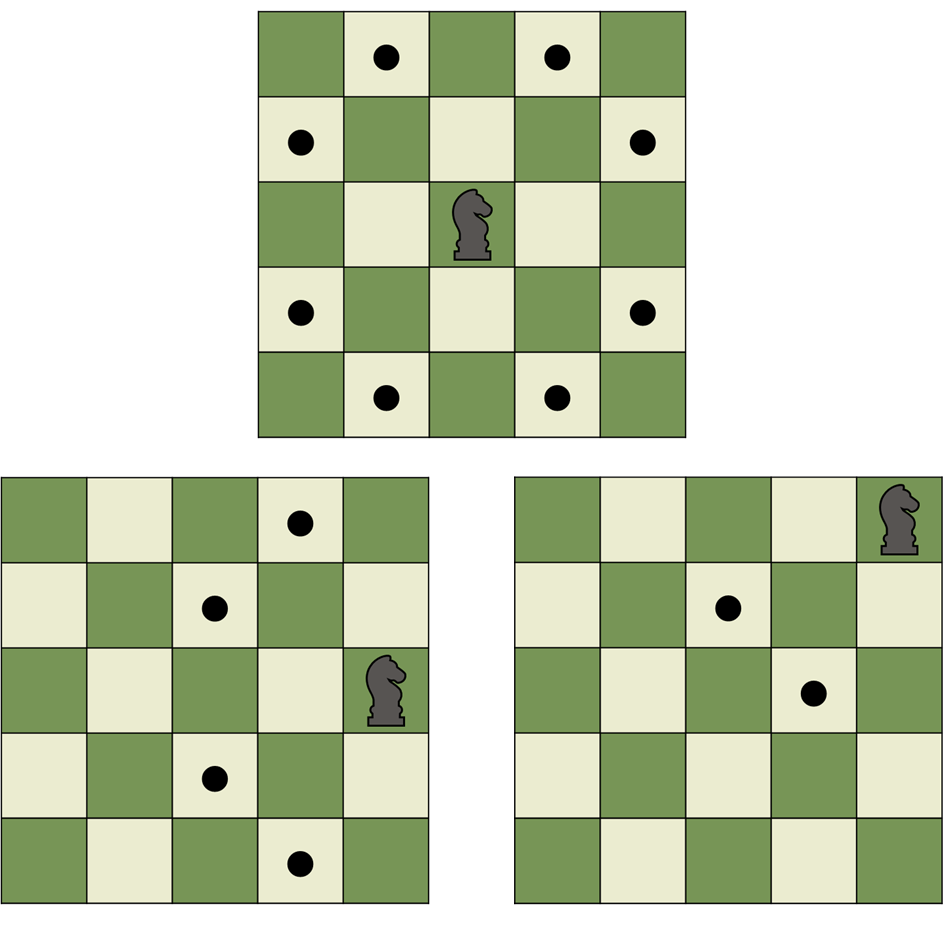
Data una scacchiera NxN ed un cavallo posizionato su una casella trovare una sequenza di mosse che consenta al cavallo di occupare tutte le caselle della scacchiera ciascuna esattamente una volta. Si risolva il problema utilizzando un algoritmo di ricerca in ampiezza.

# Analisi

Il problema consiste nella ricerca di un cammino in un grafo non orientato in cui ogni nodo è rappresentato da una casella ed ogni arco è rappresentato da uno dei movimenti del cavallo.

Nel gioco degli scacchi il cavallo si muove descrivendo una “L”, la sua mobilità è massima quando si trova nel centro (8 possibili movimenti), scende quando si trova ad un lato della scacchiera (4 possibili movimenti) ed è minima quando si trova in un angolo (2 possibili movimenti).

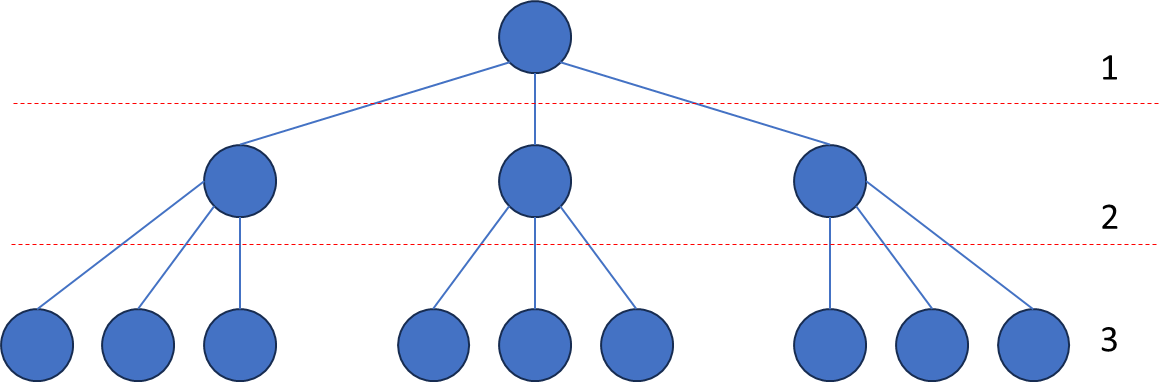
Esempio su una scacchiera 5x5:



# Soluzione

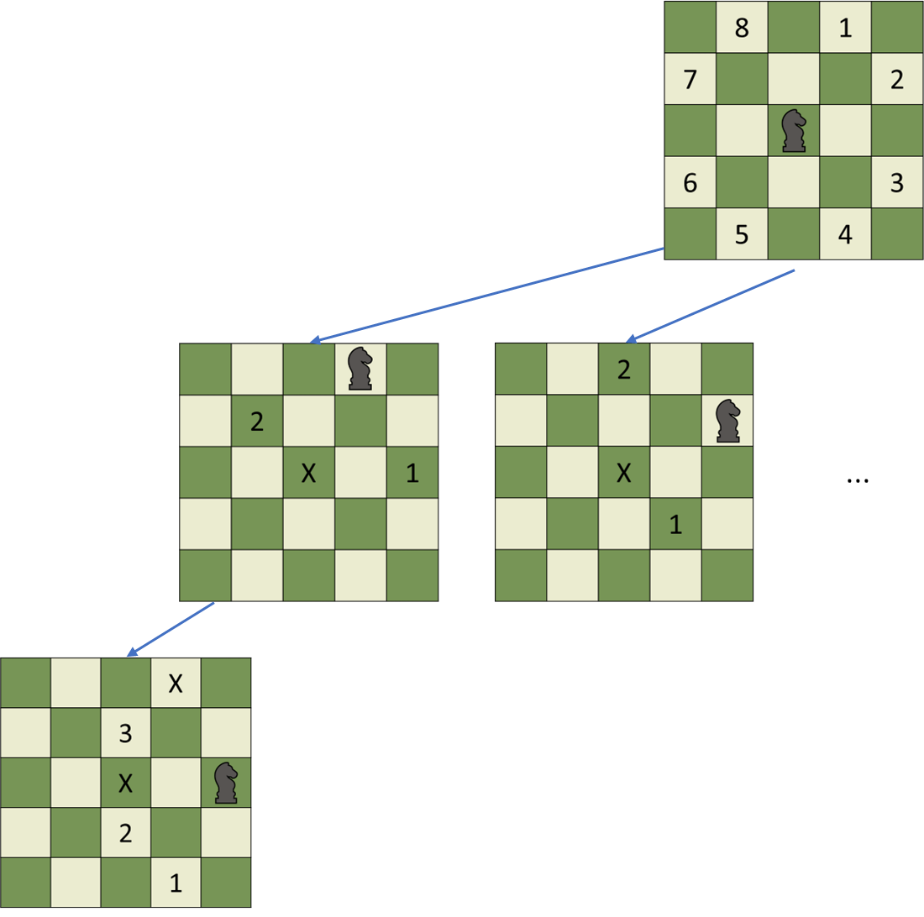
# Logica dell’algoritmo

L’algoritmo di ricerca in ampiezza esplora tutti i nodi alla stessa profondità prima di passare al livello successivo, ciò implica che la soluzione sarà sicuramente ad una profondità ottima ma allo stesso tempo richiederà una quantità di memoria esponenziale per essere trovata.



Nel nostro caso specifico per ogni movimento l’algoritmo esplorerà ogni possibile cella raggiungibile dal cavallo prima di passare al movimento successivo.

Esempio su una scacchiera 5x5 con il cavallo posizionato al centro:



# Implementazione

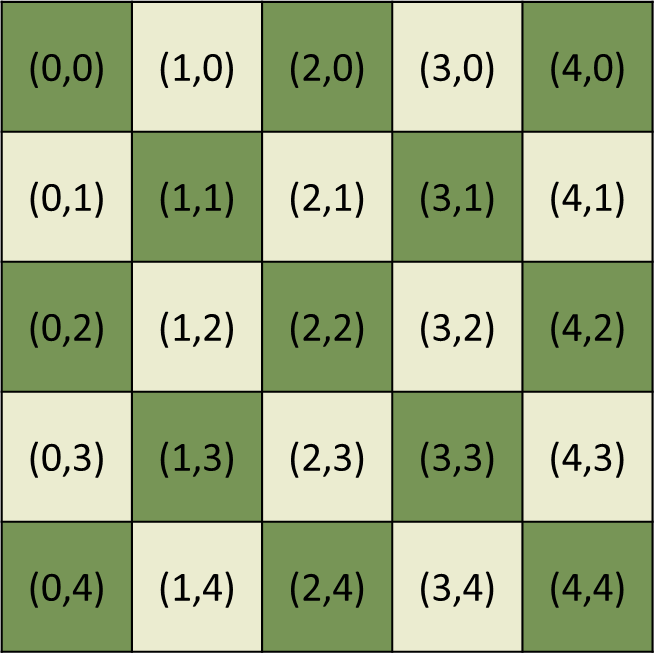
# Tipi

Il grafo è rappresentato da una funzione di successione cioè una funzione che dato un nodo restituisce i vicini del nodo stesso.

type 'a graph = Graph of ('a->'a list);;

Una casella della scacchiera è rappresentata dalle sue coordinate (x,y) dove il punto (0,0) rappresenta il punto più in alto a sinistra della scacchiera mentre il punto (N-1,N-1) rappresenta il punto più in basso a destra

type cell = Cell of (int \* int);;



Esempio su una scacchiera 5x5 con il cavallo posizionato al centro:

# let position = Cell(2,2);;

# let moves =

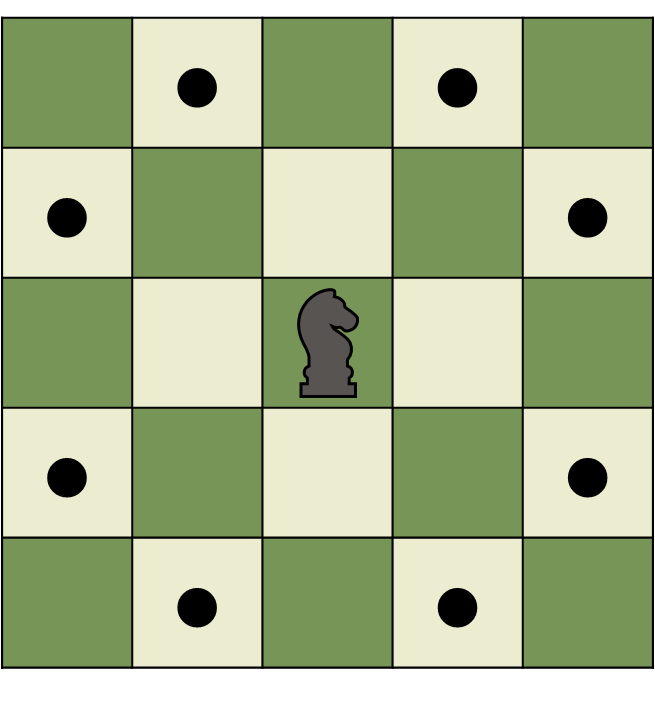
[

  Cell(2,2);Cell(3,0);Cell(4,1);

  Cell(4,3);Cell(3,4);Cell(1,4);

  Cell(0,3);Cell(0,1);Cell(1,0)

];;



# Funzione successori

La funzione successori determina dato un nodo tutti i nodi adiacenti o nel nostro caso specifico caso data una cella determina tutte le celle raggiungibili dal cavallo in una sola mossa.

c = cella attuale

n = dimensioni della scacchiera

c\_list = lista delle celle esplorate

(\* move : cell -> int -> cell list -> cell list \*)

let move c n c\_list =

  let rec aux = function

    [] -> []

    | f::rest -> try (f c n c\_list)::(aux rest)

        with UnvalidPosition -> aux rest in

aux [

move\_up\_rg; move\_rg\_up; move\_rg\_dn; move\_dn\_rg;

move\_dn\_lf; move\_lf\_dn; move\_lf\_up; move\_up\_lf

];;

Le funzioni:

move\_up\_rg, move\_rg\_up, move\_rg\_dn, move\_dn\_rg, move\_dn\_lf, move\_lf\_dn, move\_lf\_up, move\_up\_lf

determinano se il rispettivo movimento è valido (e.s. move\_up\_rg = muovi in alto a destra), se è cosi restituiscono la cella di arrivo altrimenti lanciano l’eccezione **UnvalidPosition**.

Un movimento è valido se:

1. Non porta il cavallo fuori dalla scacchiera;
2. Non finisce in una cella già visitata;

La funzione *“is\_in\_board (\* cell -> int -> bool \*)”* si assicura che le coordinate della cella siano coerenti con le dimensioni della scacchiera.

La funzione *“is\_valid (\* cell -> int -> cell\_list -> bool \*)”* verifica che la cella **c** sia in una posizione della scacchiera valida e che non sia presente nella percorso esplorato.

(\* cell -> int -> bool \*)

let is\_in\_board c n =

  match c with Cell(x, y) -> x >= 0 && y >= 0 && x < n && y < n;;

(\* cell -> int -> cell list -> bool \*)

let is\_valid c n c\_list = (is\_in\_board c n) && (not (List.mem c c\_list));;

(\* cell -> int -> cell list -> cell \*)

let move\_up\_rg c n c\_list = match c with

  Cell(x, y) -> let landing = Cell(x + 1, y - 2) in

    if (is\_valid landing n c\_list) then landing

    else raise UnvalidPosition;;

# Funzione obiettivo

L’obiettivo del problema è quello di far vistare al cavallo tutte le caselle senza passare due volte per la stessa quindi data una lista dei nodi attraversati possiamo dire che questa è una soluzione corretta se:

1. La lunghezza della lista è NxN;
2. La lista non contiene duplicati;

**NOTA:** In questa funzione ho assunto che i nodi nella lista siano sempre delle celle valide cioè:

La funzione ausiliaria *“has\_duplicate (\* ‘a list -> bool \*)”* determina in modo ricorsivo se una lista ha o meno duplicati.

Il caso base della ricorsione è la lista vuota, in questo caso restituisce **false** in quanto non ha trovato alcun duplicato.

Il caso ricorsivo estrae l’elemento in testa alla lista e lo confronta con i restanti elementi, se trova un duplicato allora restituisce **true** altrimenti prosegue sul resto degli elementi.

(\* Verifica se la lista contiene duplicati \*)

(\* has\_duplicate : 'a list -> bool \*)

let rec has\_duplicate lst =

    match lst with

    [] -> false

    | x::rest -> if List.mem x rest then true

                 else has\_duplicate(rest);;

(\* Verifica che sia una lista NxN e non contenga duplicati \*)

(\* goal : cell list -> int -> bool \*)

let goal c\_list n = List.length c\_list = (n \* n) && not (has\_duplicate c\_list);;

# Estensione del percoso

(\* extend: cell list -> int -> cell list list \*)

let extend path n =

  (\*print\_path path; \*)

  (\* Doc OCaml:

    val map : ('a -> 'b) -> 'a list -> 'b list

    map f [a1; ...; an] applies function f to a1, ..., an, and builds the list [f a1; ...; f an] with the results returned by f.

    val filter : ('a -> bool) -> 'a list -> 'a list

    filter f l returns all the elements of the list l that satisfy the predicate f. The order of the elements in the input list is preserved.

  \*)

  List.map (function x -> x::path)

    (List.filter (function x -> not (List.mem x path)) (move (List.hd path) n (List.tl path)));;

La funzione *“extend”* estende il cammino attuale determinando quali sono i prossimi nodi “buoni” da esplorare.

Nel dettaglio:

**move (List.hd path) n (List.tl path)**Chiama la funzione successori passando in input:

* il nodo in testa al percorso;
* la dimensione della scacchiera (per determinare quali percorsi mi porterebbero fuori dalla scacchiera;
* Il resto della lista tranne il nodo inziale (per determinare quali percorsi mi porterebbero ad esplorare un nodo già visitato);

**(List.filter (function x -> not (List.mem x path))**

Filtra i valori ricevuti da **move** escludendo quelli già presenti sul percorso;

**List.map (function x -> x::path)**

Per ognuno dei nuovi nodi buoni genera un percorso dove il nuovo nodo è rappresenta la testa ed il resto del corpo è rappresentato dal percorso attuale;

**Esempio:**

let p = [Cell(1,1); Cell(2,2)];;

val p : cell list = [Cell(1,1); Cell(2,2)]

let k = (List.filter (function x -> not (List.mem x p)) (move (List.hd p) n (List.tl p)));;

- : cell list = [Cell (3, 0); Cell (3, 2); Cell (2, 3); Cell (0, 3)];;

List.map (function x -> x::p) k;;

- : cell list list =

[[Cell (3, 0); Cell (1, 1); Cell (2, 2)];

 [Cell (3, 2); Cell (1, 1); Cell (2, 2)];

 [Cell (2, 3); Cell (1, 1); Cell (2, 2)];

 [Cell (0, 3); Cell (1, 1); Cell (2, 2)]]

# Risoluzione Breadth First Search

(\*

  Implementazione dell'algoritmo di ricerca in ampiezza;

  Prende in ingresso il nodo inziale (in questo caso la cella di partenza) e la dimensione della scacchiera

\*)

let bfs start n =

  let rec search\_aux = function

    [] -> raise NotFound

    | path::rest ->

      if goal path n

        then List.rev path

        else search\_aux (rest @ (extend path n))

  in search\_aux [[start]];;

Il **caso base** della ricorsione è la lista vuota, cioè non ci sono più percorsi da esplorare quindi non è stato possibile determinare una soluzione, in questo caso lancia l’eccezione **NotFound**;

Il **caso ricorsivo** verifica se il percorso in testa alla lista soddisfa la condizione di fine, se cosi è allora restituisce il percorso altrimenti procede in modo ricorsivo estendendolo e aggiungendo il resto dei percorsi in testa, così facendo tutti gli altri percorsi saranno esplorati prima che si torni ad esplorare il percorso attuale.

# Soluzioni alternative

# Risoluzione Depth First Search

(\*

  Implementazione dell'algoritmo di ricerca in profondità;

  Prende in ingresso il nodo inziale (in questo caso la cella di partenza) e la dimensione della scacchiera

\*)

let dfs start n =

  let rec search\_aux = function

    [] -> raise NotFound

    | path::rest ->

      if goal path n

        then List.rev path

        else search\_aux ((extend path n) @ rest)

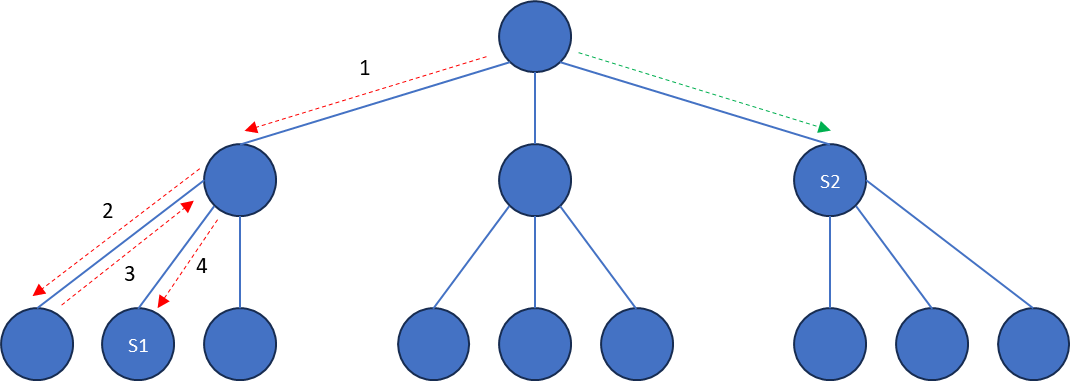
in search\_aux [[start]];;

I nodi vengono esplorati scendendo sempre più in profondità nell’albero per poi risalire per esplorare percorsi alternativi.

Il **caso base** della ricorsione è la lista vuota, cioè non ci sono più percorsi da esplorare quindi non è stato possibile determinare una soluzione, in questo caso lancia l’eccezione **NotFound**;

Il **caso ricorsivo** verifica se il percorso in testa alla lista soddisfa la condizione di fine, se così è allora restituisce il percorso altrimenti procede in modo ricorsivo estendendolo e aggiungendo il resto dei percorsi in coda, così facendo il percorso attuale sarà esplorato nuovamente per primo fino al raggiungimento di una soluzione o un vicolo ceco.

Nel caso generale l’algoritmo di ricerca in profondità può restituire dei percorsi sub ottimi cioè percorsi che portano ad una soluzione “meno ottimale” di un’altra.



S1 ed S2 rappresentano due soluzioni del problema, in rosso il percorso che segue l’algoritmo DFS fino al raggiungimento di una soluzione sub ottima, mentre in verde il percorso per il raggiungimento della soluzione ottima.

Nel caso specifico del problema del **salto del cavallo pigro** ciò non rappresenta un problema in quanto tutte le soluzioni prevedono che il cavallo compia lo stesso numero di passi (deve visitare tutte le caselle una ed una sola volta) quindi tutte le soluzioni si troveranno alla stessa profondità NxN; Inoltre esplorando un percorso alla volta l’algoritmo DFS permette di non tenere in memoria contemporaneamente molteplici percorsi.

# Risoluzione Hill-Climbing

L’algoritmo **A\*** cerca di determinare il percorso ottimo per raggiungere un obiettivo assegnando ad ogni nodo un costo definito da:

Dove:

* g(n) rappresenta il costo del cammino dal nodo iniziale fino al nodo n;
* h(n) è la funzione euristica che stima il costo del cammino dal nodo n fino al nodo obiettivo, trattandosi di una stima non è detto che determini sempre il valore esatto. D’altro canto se esistesse una funzione che determina sempre il costo esatto per raggiungere l’obiettivo non avrebbe senso parlare di “ricerca della soluzione”; L’unico vincolo di h(n) è che non deve mai sbagliare per eccesso il costo per arrivare all’obiettivo.

# Euristica di Warnsdorff

* Una cella Q è accessibile dalla cella P se può essere raggiunta in 1 mossa e Q non è stata visitata;
* S rappresenta l'insieme delle celle accessibili da P (Q ∈ S);
* L'accessibilità di P è la cardinalità delle sue celle accessibili cioè |S|;

**Algoritmo:**

Data una cella P determino S, per ogni cella in S determino la sua accessibilità. Il valore dell'euristica di Warnsdorff sarà il minimo delle accessibilità in S.

**Esempio:**

Nel seguente esempio le celle (1,0) e (4,3) avranno un’accessibilità di 2 mentre le celle (1,2) e (2,3) avranno un’accessibilità di 5.

Per l’euristica di Warnsdorff sono da preferirsi

