Università degli Studi di Perugia
DIPARTIMENTO DI MATEMATICA E INFORMATICA
[LM-18] INFORMATICA

Academic Year: 2023/2024

Course: Artificial Intelligent Systems - Intelligent Model Project

Professor: Stefano Marcugini

Ongoing assignment: Intelligent Application Development

Student: Lorenzo Mariotti

ID: 369094

Sommario

1. Problema salto del cavallo pigro

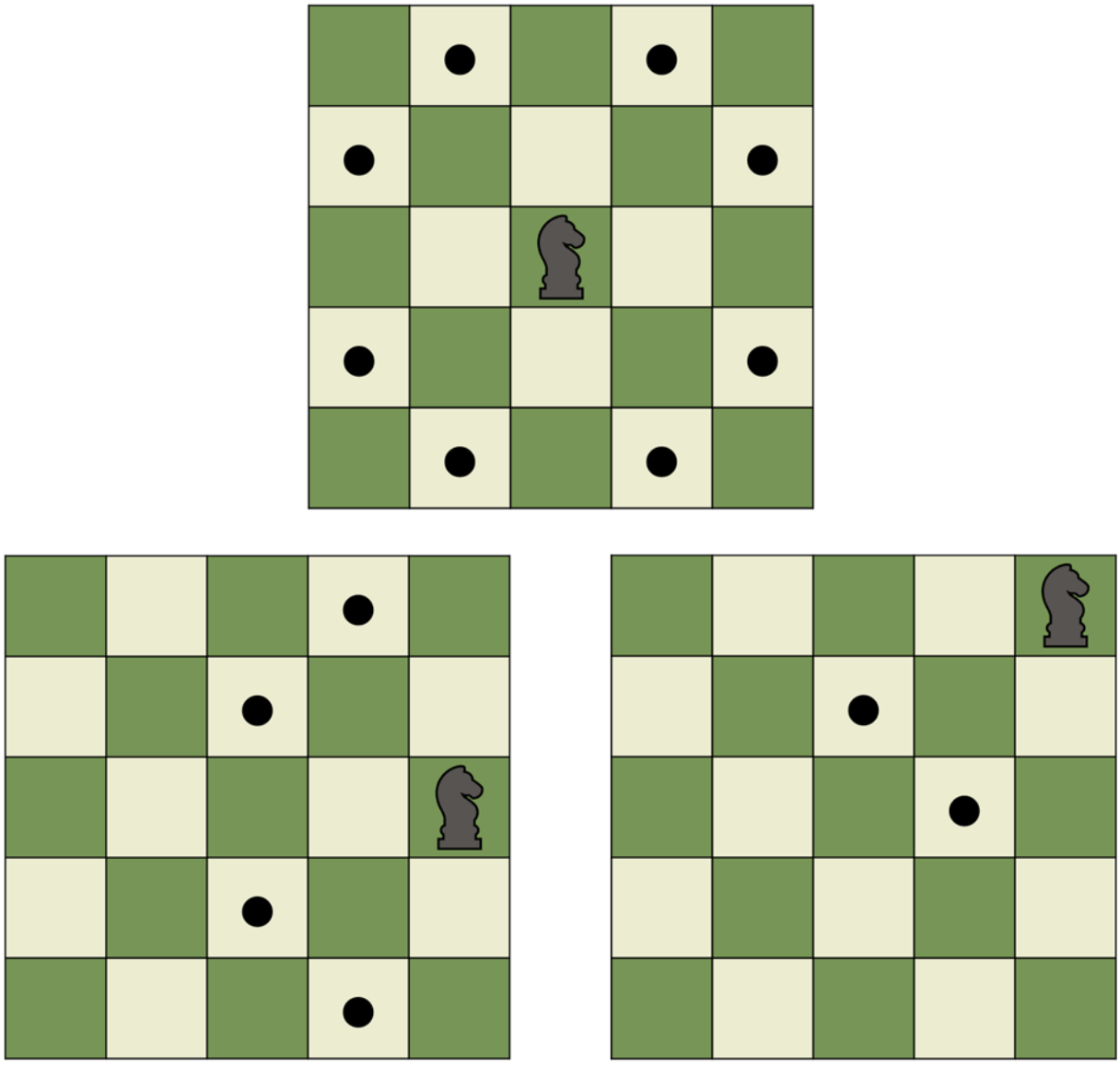
1.1. Enunciato

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.

1.2. 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).

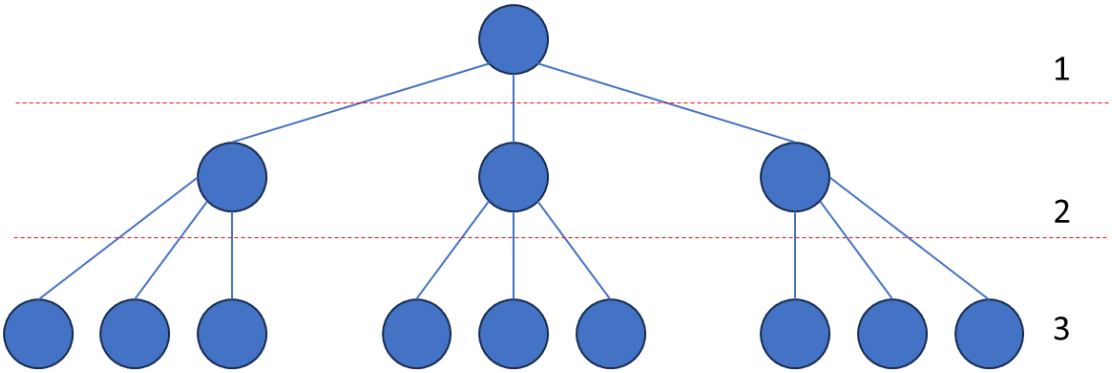
Esempio su una scacchiera 5x5:



2. Soluzione

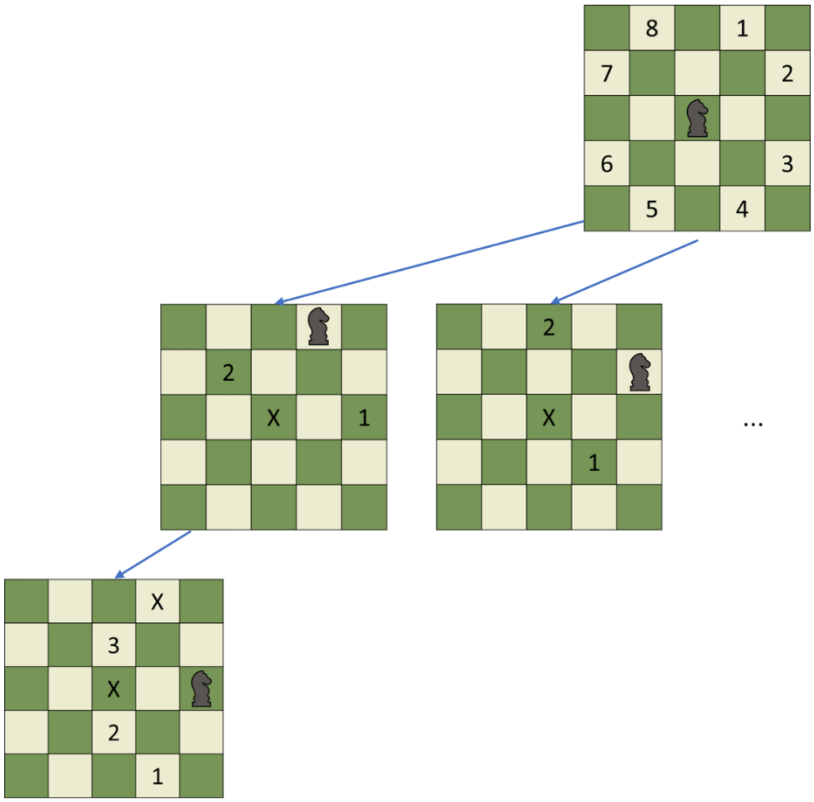
2.1. Logica dell'algoritmo

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:



2.2. Implementazione

2.2.1. 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);;
```

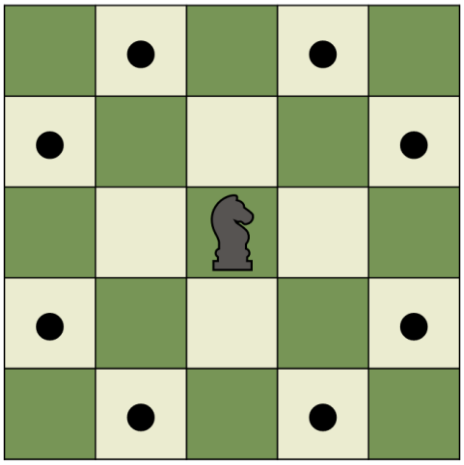
Una casella della scacchiera è rappresentata dalle sue coordinate (x,y) dove il punto (0,0) rappresenta il punto più in alto a sinistra della scacchiera mentre il punto (N-1,N-1) rappresenta il punto più in basso a destra

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

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)

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)
];;
```



2.2.2.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;;
```

2.2.3.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:

$$Cell(x,y): 0 \leq x < N \ \wedge \ 0 \leq x < N$$

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);;
```

2.2.4.Risoluzione Breadth First Search

```
let extend path n =
  (*print_path path; *)

  (*
    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)));;

(*
  Implementazione dell'algorimo 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]];;
```