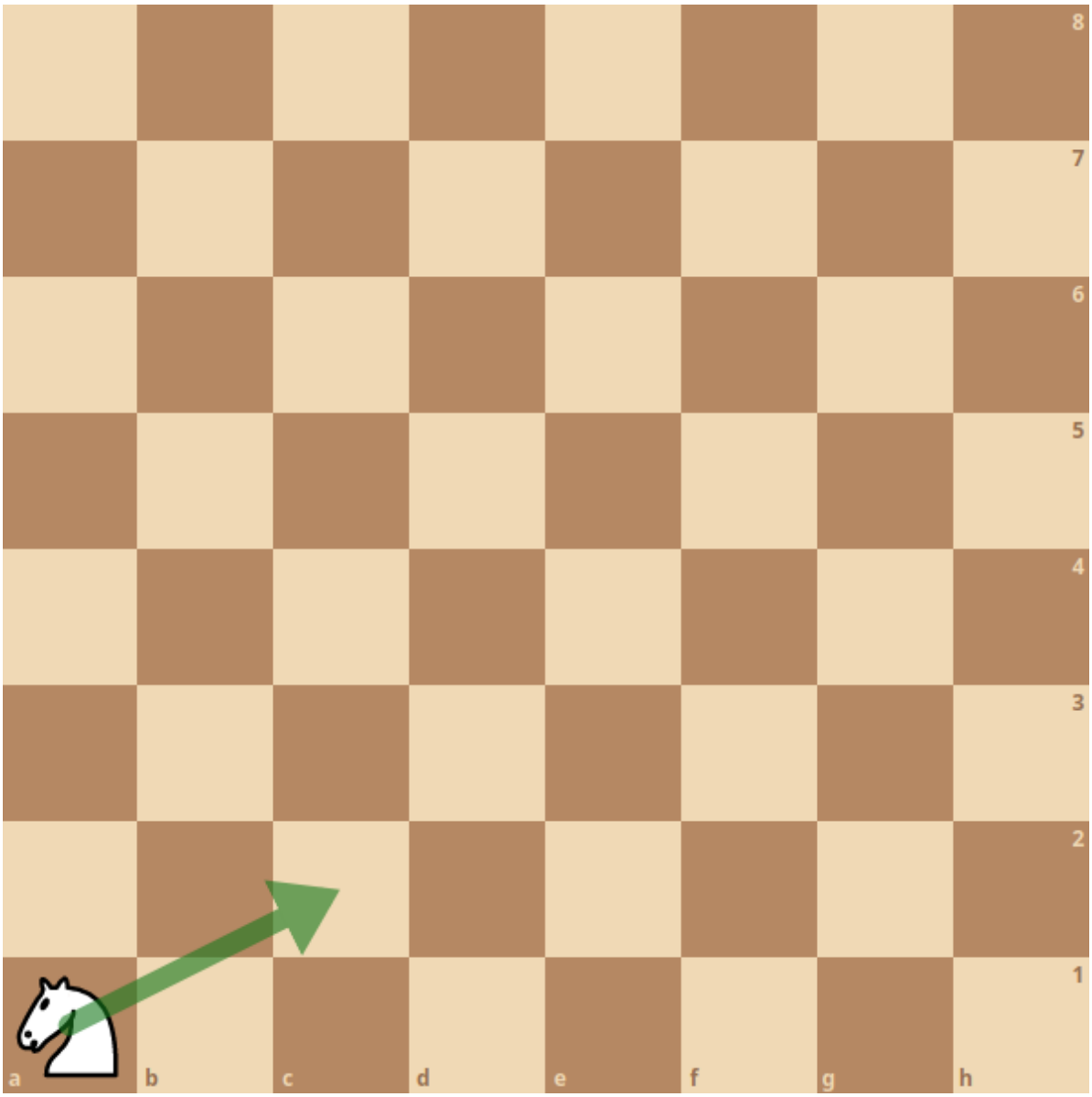
Intelligent Application Development

Problema del salto del cavallo (Ricerca in Ampiezza)

Bocchini Fabio - matricola: 340600 - A.A. 2021/2022



GitHub repository: https://github.com/FabioBocchini/salto_del_cavallo.git

Obiettivo

Data una scacchiera n x n ed un cavallo posizionato su una casella trovare una sequenza di mosse che consenta al cavallo di occupare tutte le caselle delle scacchiera ciascuna esattamente una volta prima di ritornare alla posizione di partenza. Si risolva il problema utilizzando un algoritmo di ricerca in ampiezza.

Esecuzione

Ricerca in Ampiezza

Per prima cosa ho provato a risolvere il problema con una ricerca nel grafo dei cammini, ovvero una ricerca in ampiezza in un grafo dove ogni nodo è un percorso che il cavallo può effettuare. Per fare questo abbiamo di una funzione `camminoSuccessore` che ci permetta di lavorare sui cammini invece che sui singoli nodi di G.

Per prima cosa definiamo il tipo `graph` come funzione che dato in input un elemento di tipo **'a** (nel nostro caso sarà una coppia di interi) restituisce una lista di **'a**

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

`deltaSalti` e `posizioneSicura` ci aiutano a definire la funzione `successori`

`deltaSalti` ci indica quali spostamenti sono possibili da un cavallo su una scacchiera sommando il primo valore della coppia alla x e il secondo alla y della posizione attuale

`posizioneSicura` controlla che una posizione si trovi all'interno della scacchiera, `n` indica il numero di righe e di colonne della scacchiera

```
let deltaSalti = [(1,2);(1,-2);(-1,2);(-1,-2);(2,1);(2,-1);(-2,1);(-2,-1)];;
let posizioneSicura (x,y) n = x > 0 && x <= n && y > 0 && y <= n;;
```

la funzione `successori`

```
let successori n stato =
    let rec successoriAux risultato = function
        |[] -> risultato
        |(dx,dy)::rest ->
            let x = (fst stato) + dx in
            let y = (snd stato) + dy in
            if posizioneSicura (x,y) n then
                successoriAux ((x,y)::risultato) rest
            else
                successoriAux risultato rest
    in successoriAux [] deltaSalti
;;
```

dato uno stato in forma (x, y) in cui si trova il cavallo restituisce una lista stati (coppie (x,y)) raggiungibili dal cavallo andando a sommare ai valori della posizione corrente le coppie di `deltaSalti` e controllando con `posizioneSicura` che non siano fuori dalla scacchiera

la funzione `camminoSuccessore`

```
let camminoSuccessore successori cammino =
    let rec camminoSuccessoreAux = function
        |[] -> []
        |h::t ->
            if List.mem h cammino then
            camminoSuccessoreAux t
        else
            (h::cammino)::(camminoSuccessoreAux t)
    in camminoSuccessoreAux (successori (List.hd cammino))
 ;;
```

prende in input un cammino, ovvero una lista di coppie (x, y) e restituisce una lista di cammini, ovvero una lista di liste di coppie, che rappresentano tutti i cammini che il cavallo può percorrere avendo già percorso il cammino di input e facendo un salto in più

la funzione obiettivo

```
let obiettivo n grandezzaScacchiera inizio risultato =
  (List.length risultato) = grandezzaScacchiera && (List.exists ((=) inizio)
(successori n (List.hd risultato)))
;;
```

controlla che un cammino corrisponda ad una soluzione accettabile del problema. Per farlo basta controllare che la dimensione del cammino sia `n x n`, ovvero che il numero di caselle su cui è passato il cavallo corrisponda all'intera scacchiera (`camminoSuccessore` controlla già che nello stesso cammino non ci siano due caselle uguali), e che sia possibile raggiungere la casella iniziale dall'ultima casella raggiunta. Questo viene fatto utilizzando la funzione `successori` e il primo elemento della lista risultato, perché i nuovi elementi vengono aggiunti in testa, quindi `List.hd` corrisponderà all'ultimo elemento inserito.

ora possiamo effettuare la visita in ampiezza

```
let visitaAmpiezza (Graph successori) obiettivo inizio =
    let rec cerca = function
        |[] -> raise SoluzioneNonTrovata
        |cammino::resto ->
            if obiettivo cammino then
                cammino
            else cerca (resto @ (successori cammino)
    in cerca [[inizio]]
;;
```

il primo argomento è il grafo definito dalla funzione `camminoSuccessore` e utilizza una lista di cammini (ovvero una lista di liste di coppie). Per ogni elemento della lista controlla se il cammino corrisponde ad una soluzione, se lo è allora il cammino viene restituito e l'esecuzione finisce, se non lo è il cammino viene espanso tramite `camminoSuccessore` e i suoi figli vengono aggiunti in coda alla lista utilizzata per la ricerca. La lista è quindi utilizzata come Queue per una ricerca in ampiezza, se i nuovi figli fossero stati aggiunti in testa allora sarebbe stato uno Stack e quindi una ricerca in profondità.

```
let cercaCamminoInAmpiezza ((Graph successori), obiettivo, inizio) =
  List.rev (visitaAmpiezza (Graph (camminoSuccessore successori)) obiettivo
inizio);;

let saltoDelCavallo inizio n =
  time cercaCamminoInAmpiezza ((Graph (successori n)), (obiettivo n (n * n)
inizio), inizio);;
```

Queste due ultime funzioni sfruttano le caratteristiche del linguaggio funzionale per passare i vari argomenti alle funzioni e per invertire la lista risultato (che essendo costruita aggiungendo elementi in testa è al contrario) e per stampare il tempo necessario all'esecuzione.

Il programma può essere eseguito chiamando la funzione `saltoDelCavallo` con argomenti: posizione iniziale in forma (x,y) e numero di colonne della scacchiera

```
saltoDelCavallo (1,1) 8;;
```

Utilizzando questo algoritmo, con 0<n<6, la ricerca non trova soluzioni, come possiamo controllare in effetti non esistono soluzioni per n<6 e per n dispari (infatti su una scacchiera n x n con n dispari ci sono un numero dispari di caselle, e il cavallo, cambiando il colore della casella ad ogni salto, non può tornare sullo stesso colore dopo un numero di salti dispari).

Da n = 6 in poi però possiamo notare un alto tempo d'esecuzione, e soprattutto che l'algoritmo finisce per Stack Overflow. Questo è dovuto alla grande quantità di memoria che viene utilizzata per mantenere la lista di cammini da espandere in `visitaAmpiezza`. Se immaginiamo il grafo dei cammini come un albero infatti, da ogni nodo possiamo avere un massimo di 8 nodi figli (8 possibili mosse del cavallo), quindi al livello n ci sono 8^n cammini di lunghezza n da memorizzare. La soluzione su una scacchiera 6x6 si trova quindi a n = 36, 8^36=3,2e+32!

```
saltoDelCavallo (1,1) 6;;
```

Stack Overflow dopo 20 minuti circa.

```
saltoDelCavallo (1,1) 8;;
```

Stack Overflow dopo 15 minuti circa. Lo Stack Overflow viene raggiunto prima perché essendoci più caselle libere ci sono più mosse disponibili, quindi il branching factor dell'albero è più vicino ad 8.

Per trovare una soluzione ho quindi optato per una ricerca informata.

Ricerca Informata

Per la ricerca informata, come funzione euristica ho usato la regola di Warnsdorff, che sceglie di espandere prima i nodi con meno mosse successive disponibili. Quindi oltre che alla posizione della casella prende anche in considerazione le caselle già visitate

```
let warnsdorff p1 p2 =
    let a = List.length(List.filter (fun x -> not(List.mem x p1)) (successori
(List.hd p1)))     in
    let b = List.length(List.filter (fun x -> not(List.mem x p2)) (successori
(List.hd p2)))
    in

    if a>b then 1
    else if a=b then 0
    else -1
in
```

questa funzione poi verrà usata come criterio d'ordinamento, quindi restituisce 1 se la prima posizione va inserita dopo la seconda, 0 se hanno lo stesso numero di mosse successive disponibili, e -1 se la prima va inserita prima della seconda nella lista.

La funzione `camminoSuccessore` è stata sostituita con `estendi`

```
let estendi cammino =
    List.map (fun x -> x::cammino)
        (List.filter (fun x -> not(List.mem x cammino)) (successori (List.hd
cammino)))
in
```

che svolge lo stesso compito.

La funzione di ricerca diventa quindi

```
let rec cerca = function
    |[] -> raise SoluzioneNonTrovata
    |cammino::resto ->
        if obiettivo cammino then
            stampalista (List.rev cammino)
        else
            cerca ((List.sort warnsdorff (estendi cammino)) @ resto)
in
cerca [[inizio]]
```

che, a differenza di quella precedente utilizza la lista di cammini da visitare è ordinata ogni volta tramite la funzione euristica.

La regola di Warnsdorff funziona bene quando scegliamo come inizio una casella centrale della scacchiera, quindi

```
time saltoDelCavallo ((3,3), 6);;
```

```
(C,3); (A,2); (C,1); (E,2); (F,4); (E,6); (C,5); (A,6); (B,4); (D,5); (F,6);
(E,4); (F,2); (D,1); (B,2); (A,4); (B,6); (C,4); (A,5); (C,6); (E,5); (D,3);
(E,1); (F,3); (D,2); (F,1); (E,3); (F,5); (D,6); (B,5); (D,4); (B,3); (A,1);
(C,2); (A,3); (B,1);
execution time: 0.004433s
```

```
saltoDelCavallo ((4,4), 8);;
```

```
(D,4); (B,3); (A,1); (C,2); (A,3); (B,1); (D,2); (F,1); (H,2); (F,3); (E,1);
(G,2); (H,4); (G,6); (H,8); (F,7); (D,8); (B,7); (A,5); (C,4); (B,2); (D,1);
(E,3); (G,4); (H,6); (G,8); (E,7); (C,8); (A,7); (C,6); (E,5); (D,3); (C,1);
(A,2); (B,4); (A,6); (B,8); (D,7); (F,8); (H,7); (G,5); (H,3); (G,1); (E,2);
(F,4); (H,5); (F,6); (D,5); (B,6); (A,8); (C,7); (E,6); (C,5); (A,4); (C,3);
(E,4); (F,2); (H,1); (G,3); (F,5); (G,7); (E,8); (D,6); (B,5);
execution time: 0.007165s
```

Decisamente migliore alla semplice ricerca in ampiezza.