



UNIVERSITÀ DI PERUGIA  
Dipartimento di Matematica e Informatica



LAUREA MAGISTRALE IN INFORMATICA  
CORSO DI ARTIFICIAL INTELLIGENT SYSTEM

# Relazione progetto d'esame

## SOLUZIONE GIOCO DEL 15

*Professore*

**Prof. Stefano Marcugini**

*Studente*

**Cristian Cosci**  
**(Matricola: 350863)**

---

Anno Accademico 2021-2022

# Indice

<b>1</b>	<b>Introduzione al problema</b>	<b>3</b>
<b>2</b>	<b>Descrizione codice</b>	<b>4</b>
2.1	Definizione del problema . . . . .	4
2.2	Implementazione Grafo . . . . .	11
2.3	Algoritmi di ricerca . . . . .	13
2.3.1	Best first . . . . .	15
2.3.2	A* . . . . .	17
<b>3</b>	<b>Test del programma</b>	<b>21</b>
3.1	Istanze di test . . . . .	21
3.2	Confronto vari algoritmi . . . . .	22
<b>4</b>	<b>Conclusioni</b>	<b>24</b>

# Capitolo 1

## Introduzione al problema

Il lavoro svolto ha l'obiettivo di realizzare un programma che data una configurazione del gioco del 15 (vedi Figura 1.1) determini, se esiste, una sequenza di mosse che porti alla soluzione. Per fare ciò ho deciso di utilizzare vari algoritmi a partire da un algoritmo di ricerca **best first** ad un algoritmo **A\***. Nei capitoli successivi saranno spiegate le varie implementazioni e verranno confrontati i vari algoritmi.



Figura 1.1: Configurazione stato goal gioco del 15.

# Capitolo 2

## Descrizione codice

Data una tabella di dimensione  $4 \times 4$  con 15 tessere (ogni tessera ha un numero da 1 a 15) e uno spazio vuoto. L'obiettivo è posizionare i numeri sulle tessere in ordine utilizzando lo spazio vuoto. Possiamo far scorrere lo spazio vuoto scambiandolo con le quattro tessere adiacenti (sinistra, destra, sopra e sotto).

Nei paragrafi seguenti verrà descritta l'implementazione del programma, scritto nel linguaggio di programmazione funzionale **OCaml**.

Il programma partendo da un'istanza di input, che rappresenta lo stato iniziale del problema, mediante un algoritmo di ricerca espande il grafo dei cammini (muovendosi tra le mosse possibili) cercherà di risolvere la configurazione, arrivando all'obiettivo (vedi Figura 2.1).

### 2.1 Definizione del problema

Per prima cosa ho deciso la rappresentazione di ogni nodo nel grafo e quindi di ogni stato nel problema.

Nel mio caso ogni stato rappresenta una configurazione della tabella. Ho quindi definito per comodità un nuovo tipo chiamato **stato** il quale è costituito da una lista di interi. La lista di interi rappresenta in sequenza le caselle della tabella, in ordine dalla prima in alto a sinistra all'ultima a destra. La casella vuota è rappresentata dal numero 0.

### Initial Configuration

2	1	3	4
5	6	7	8
9	10	11	12
13	14	15	

### Final Configuration

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figura 2.1: Configurazione iniziale di esempio e configurazione obiettivo.



Figura 2.2: Tipo stato: una lista di interi.

Ho successivamente definito tutte le funzioni di comodo per effettuare i movimenti della casella vuota. Per ogni movimento ho definito una funzione:

- **muoviSU**;
- **muoviGIU**;
- **muoviDX**;
- **muoviSX**.

Ogni funzione prende in input uno stato, determina la posizione della cella vuota e, se è possibile effettuare tale mossa, ritorna un nuovo stato con la cella vuota nella nuova posizione (la mossa consiste nello scambiare la cella vuota con quella che si trova nella direzione dello spostamento di quest'ultima), altrimenti restituisce un'eccezione (come nel caso in cui la cella vuota è in uno dei bordi della tabella e la mossa preveda di muovere in quella direzione).

Le quattro funzioni di movimento utilizzando altre due funzioni di comodo:

- **findBlank**: dato uno stato restituisce l'indice in cui si trova la casella vuota.
- **scambia**: data una lista di interi e due indici, ritorna una lista di interi con i valori nei due indici scambiati di posizione.

```

(* funzioni di movimento con opportuni controlli*)
(* stato -> stato *)
let muoviSU (St(state)) =
  let blankIndex =
    findBlank (St(state))
  in if blankIndex < 4 then raise Fail
  else St(scambia state blankIndex (blankIndex-4));;

let muoviGIU (St(state)) =
  let blankIndex =
    findBlank (St(state))
  in if blankIndex > 11 then raise Fail
  else St(scambia state blankIndex (blankIndex+4));;

let muoviDX (St(state)) =
  let blankIndex =
    findBlank (St(state))
  in if (blankIndex mod 4) = 3 then raise Fail
  else St(scambia state blankIndex (blankIndex+1));;

let muoviSX (St(state)) =
  let blankIndex =
    findBlank (St(state))
  in if (blankIndex mod 4) = 0 then raise Fail
  else St(scambia state blankIndex (blankIndex-1));;

```

Figura 2.3: Funzioni di movimenti.

```

(* funzione che permette di trovare l'indice della casella vuota *)
(* stato -> int *)
let findBlank (St(state)) =
  let rec aux index = function
    [] -> raise Fail
    | x::rest -> if x = 0 then index
                  else aux (index+1) rest
  in aux 0 state;;

(* funzione che permette di scambiare due caselle con indice x e y*)
(* 'a list -> int -> int -> 'a list *)
let scambia state x y =
  let rec aux index x y = function
    [] -> []
    | hd::rest ->
        if index = x then (List.nth state y)::aux (index+1) x y rest
        else if index = y then (List.nth state x)::aux (index+1) x y rest
        else hd::aux (index+1) x y rest
  in aux 0 x y state;;

```

Figura 2.4: Funzioni di comodo per i movimenti.

È necessario anche verificare che l'istanza di input sia effettivamente risolubile, in quanto non sempre una configurazione del gioco del 15 ha soluzione. In generale, per una data griglia di dimensione  $N$ , possiamo verificare se un puzzle  $N \times N$  è risolubile o meno seguendo delle semplici regole:

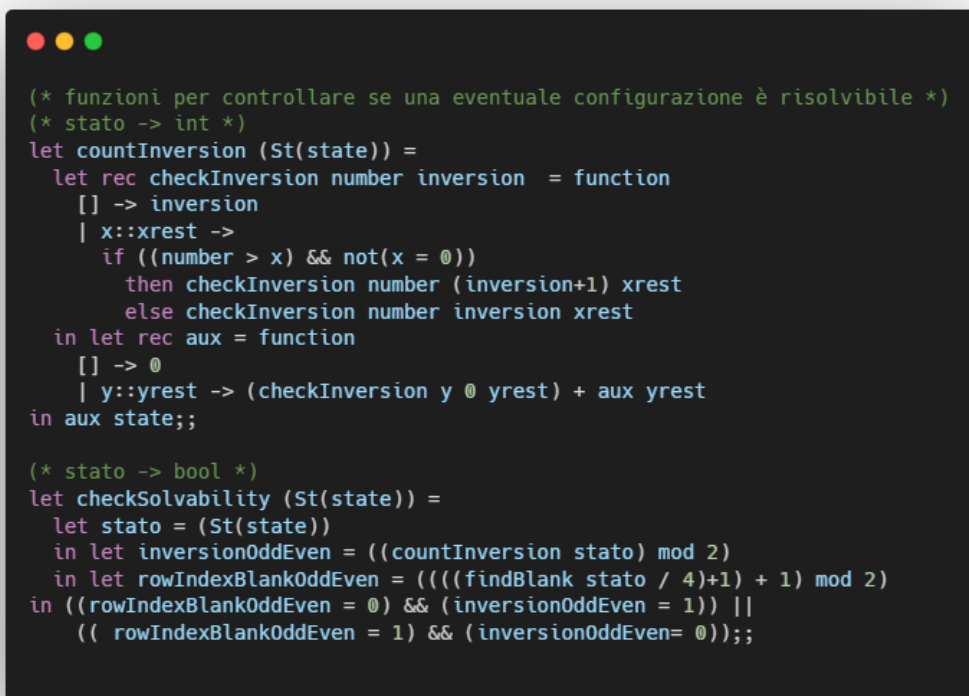
1. Se  $N$  è dispari, l'istanza del puzzle è risolubile se il numero di inversioni è pari nello stato di input.
2. Se  $N$  è pari (**il mio caso (4x4)**), l'istanza del puzzle è risolubile se
  - lo spazio vuoto è su una riga pari contando dal basso (penultimo, quartultimo, ecc.) e il numero di inversioni è dispari.
  - lo spazio vuoto è su una riga dispari contando dal basso (ultimo, terzultimo, ecc.) e il numero di inversioni è pari.
3. Per tutti gli altri casi, l'istanza del puzzle non è risolubile.



## Che cos'è un'inversione?

Se consideriamo le tessere scritte in una singola riga (lista di interi nel mio caso) invece di essere distribuite in N righe, una coppia di tessere (a, b) formano un'inversione se a compare prima di b ma  $a > b$ . Per esempio in Figura 2.1, considerando le tessere scritte in una riga, in questo modo: 2 1 3 4 5 6 7 8 9 10 11 12 13 14 15 0, si ha solo 1 inversione cioè (2, 1).

Ho per questo definito una funzione **checkSolvability** che preso uno stato in input restituisce un booleano che ci dice se la configurazione è o meno risolubile. La stessa funzione utilizza un'altra funzione di comodo che ritorna il numero di inversioni dato uno stato.

A screenshot of a code editor with a dark background and light-colored text. The code is written in OCaml and defines two functions: `countInversion` and `checkSolvability`. The `countInversion` function takes a state and returns the number of inversions. The `checkSolvability` function takes a state and returns a boolean indicating if the configuration is solvable. The code uses pattern matching and recursive calls to calculate the number of inversions and check the solvability conditions.

```
(* funzioni per controllare se una eventuale configurazione è risolubile *)
(* stato -> int *)
let countInversion (St(state)) =
  let rec checkInversion number inversion = function
    [] -> inversion
  | x::xrest ->
    if ((number > x) && not(x = 0))
    then checkInversion number (inversion+1) xrest
    else checkInversion number inversion xrest
  in let rec aux = function
    [] -> 0
  | y::yrest -> (checkInversion y 0 yrest) + aux yrest
  in aux state;;

(* stato -> bool *)
let checkSolvability (St(state)) =
  let stato = (St(state))
  in let inversionOddEven = ((countInversion stato) mod 2)
  in let rowIndexBlankOddEven = (((findBlank stato / 4)+1) + 1) mod 2
  in ((rowIndexBlankOddEven = 0) && (inversionOddEven = 1)) ||
    (( rowIndexBlankOddEven = 1) && (inversionOddEven= 0));;
```

Figura 2.5: Funzioni per controllare se una determinata istanza è o meno risolubile.

Qui di seguito ho riportato degli esempi di risultati ottenuti testando la risolubilità di alcune istanze per lo stato iniziale del problema.

13	2	10	3
1	12	8	4
5	0	9	6
15	14	11	7

$N = 4$  (Even)

Position of 0 from bottom = 2 (Even)

Inversion Count = 41 (Odd)

→ Solvable

Figura 2.6: Esempio di controllo istanza.

6	13	7	10
8	9	11	0
15	2	12	5
14	3	1	4

$N = 4$  (Even)

Position of 0 from bottom = 3 (Odd)

Inversion Count = 62 (Even)

→ Solvable

Figura 2.7: Esempio di controllo istanza.

3	9	1	15
14	11	4	6
13	0	10	12
2	7	8	5

$N = 4$  (Even)

Position of 0 from bottom = 2 (Even)

Inversion Count = 56 (Even)

→ Not Solvable

Figura 2.8: Esempio di controllo istanza.

Ovviamente il controllo per verificare se una data istanza è risolvibile o meno va fatto alla singola istanza di input. Questo perchè data un'istanza risolvibile qualsiasi altra istanza derivata da mosse legali è ancora risolvibile.

## 2.2 Implementazione Grafo

Ho deciso di rappresentare il grafo implicitamente mediante la **funzione successori** che dato uno stato  $S$  calcola la lista degli stati che si possono ottenere applicando ad  $S$  tutti gli operatori applicabili ad  $S$  (ovvero le mosse per spostare la cella vuota). Oltre a ciò, ho definito un nuovo tipo di dato per identificare il tipo **grafo**.

```
(* definizione del tipo di grafo *)
type 'a graph = Graph of ('a -> 'a list);;
```

Figura 2.9: Definizione tipo grafo.

```

(* funzione successori che rappresenta il grafo *)
(* stato -> stato list *)
let giocoDell15 state =
  let rec aux = function
    [] -> []
  | f::rest ->
    try f state :: aux rest
    with Fail -> aux rest
  in aux [muoviSU; muoviGIU; muoviDX; muoviSX];;

(* g = Graph giocoDell15 *)
(* stato graph *)
let g = Graph giocoDell15;;

```

Figura 2.10: Funzione successori che rappresenta il grafo.

Ho inoltre implementato delle funzioni che permettono di stampare il singolo stato oppure un intero cammino in un grafo (una lista di stati).

```

(* funzioni di comodo per stampa layout *)
(* stato -> unit *)
let stampaStato (St(state)) =
  let stampaNum x = print_int(x); if x>9 then print_string(" ") else print_string(" ")
  in let aCapo x = if (x mod 4) = 3 then print_newline()
  in let rec aux index = function
    [] -> print_newline()
  | x::rest -> stampaNum x; aCapo index; aux (index+1) rest
  in aux 0 state;;


(* stato list -> unit *)
let rec stampaCammino = function
  [] -> print_newline()
| x::rest -> stampaStato x; stampaCammino rest;;

```

Figura 2.11: Funzioni per la stampa dei cammini e degli stati.

## 2.3 Algoritmi di ricerca

Nelle sottosezioni successive sono riportati i due algoritmi di ricerca utilizzati risolvere il gioco del 15. Entrambi sono algoritmi che funzionano mediante l'espansione dei nodi seguendo l'ordine di una coda di priorità. Quest'ultima è ordinata mediante una funzione di valutazione che differisce a seconda dell'algoritmo. Il codice di entrambi gli algoritmi è lo stesso, l'unica differenza sta nella funzione **confrontaCammino** la quale permette di ordinare la coda di priorità secondo la funzione di valutazione scelta. Il codice dell'algoritmo di ricerca è il seguente:



```
(* algoritmo di ricerca *)
(* stato -> stato graph -> stato list *)
let search (St(state)) (Graph succ) =
  let estendi cammino (Graph succ) =
    stampaCammino cammino;
    List.map (function x -> x::cammino)
      (List.filter (function x -> (not (List.mem x cammino))) (succ (List.hd cammino)))
  in let rec search_aux = function
    [] -> raise NotFound
    | cammino::rest ->
      if isGoal (List.hd cammino)
      then List.rev cammino
      else search_aux (List.sort confrontaCammino (rest @ ((estendi cammino (Graph succ)))))
  in search_aux [(St(state))];;
```

Figura 2.12: Funzione di ricerca per la soluzione del gioco del 15.

Le funzioni di valutazione dei due algoritmi e le altre funzioni di comodo sono descritte nelle sottosezioni dedicate a ciascun algoritmo.

Per verificare se la ricerca è arrivata alla soluzione, ad ogni passaggio, prima dell'espansione viene confrontato lo stato attuale con quello obiettivo. La funzione **isGoal** restituisce quindi un booleano che indica se ci troviamo nello stato goal o meno.

```

(* funzione test obiettivo *)
let isGoal state =
  let statoGoal = St[1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;0]
  in state = statoGoal;;

```

Figura 2.13: Funzione che verifica se lo stato attuale è lo stato obiettivo.

È anche necessaria una funzione **confrontaCammino** che permette di eseguire le comparazioni per l'ordinamento. Quest'ultima sarà la stessa per tutti gli algoritmi di ricerca, quello che differisce l'una dall'altra è la funzione **distanza** utilizzata.

```

(* funzione di comparazione per l'ordinamento della coda di priorità *)
(* stato list -> stato list -> int *)
let confrontaCammino cammino1 cammino2 =
  let piuvicino (cammino1, cammino2) =
    (distanza cammino1) < (distanza cammino2)
  in if List.hd cammino1 = List.hd cammino2
     then 0
     else if piuvicino (cammino1, cammino2)
        then -1
        else 1;;

```

Figura 2.14: Funzione di comparazione utilizzata dall'algoritmo di ordinamento.

La funzione **confrontaCammino** restituisce 0, -1, 1 a seconda di quale dei due cammini passati in input è più vicino allo stato obiettivo. Questa funzione permette il paragone che viene sfruttato nella funzione di ordinamento per la coda di priorità.

### 2.3.1 Best first

Per quanto riguarda l'algoritmo **Best First** la ricerca viene effettuata espandendo per primi i nodi secondo l'ordine dato da **una funzione euristica**. La funzione euristica rappresenta il costo stimato del cammino più conveniente dallo stato attuale allo stato obiettivo. Una proprietà fondamentale che l'euristica deve avere è quella di **ammissibilità e consistenza**. In sostanza l'euristica non deve mai sbagliare la stima per eccesso, in quanto le euristiche ammissibili sono ottimiste per natura (il costo da uno stato allo stato goal è inferiore a quello reale). Un'euristica si dice consistente se per ogni nodo  $n$  e ogni successore  $n'$  di  $n$  generato da un'azione, il costo stimato per raggiungere l'obiettivo partendo da  $n$  non è superiore al costo di passo per raggiungere  $n'$  sommato al costo stimato per andare da lì all'obiettivo (disuguaglianza triangolare). **Ogni euristica consistente è anche ammissibile.**

Le più famose euristiche utilizzabili per il gioco del 15 sono quella della somma delle distanze di Manhattan oppure la somma delle caselle fuori posto. Ho deciso personalmente di implementarle entrambe e di metterle a confronto. Ho definito quindi una funzione **distanza** che calcola il valore di costo per ciascuno stato.

In Figura 2.15 è riportata la funzione distanza basata sull'euristica delle **tessere fuori posto**. Proprio come suggerisce il nome, questa euristica restituisce il numero di tessere che non sono nella loro posizione finale. La cella vuota non viene presa in considerazione durante il calcolo di questo valore euristico. Questa euristica, come mostrato nel capitolo conclusivo, è tuttavia la più lenta e verrà esplorata un'enorme quantità di nodi per raggiungere lo stato obiettivo rispetto ad altre euristiche.

In Figura 2.16 è riportata la funzione distanza basata sull'euristica delle distanze di Manhattan: il valore è ritornato dalla funzione è la somma delle distanze assolute lungo l'asse verticale e orizzontale di ogni casella rispetto alla sua posizione nello stato obiettivo. Questa euristica è sicuramente ammissibile in quanto non può sovrastimare il numero di mosse necessarie a raggiungere lo stato obiettivo. La distanza di Manhattan è più veloce della distanza di Hamming (caselle fuori posto) spiegata sopra.

```

(* Best First utilizzando caselle fuori posto *)
(* stato list -> int *)
let distanzaBF_Misplaced cammino =
  let misplaced x index =
    if (x = (index + 1)) then 0 else 1
  in let heuristic (St(state)) =
    let rec aux_Hn index count_Hn = function
      [] -> count_Hn
      | num::rest ->
        if num = 0 then
          aux_Hn (index+1) (count_Hn) rest
        else
          aux_Hn (index+1) (count_Hn + (misplaced num index)) rest
    in aux_Hn 0 0 state
  in (heuristic (List.hd cammino));;

```

Figura 2.15: Funzione di valutazione basata sul numero di caselle fuori posto.

```

(* Best First utilizzando Manhattan *)
(* stato list -> int *)
let distanzaBf_Manhattan cammino =
  let distanzaManhattan x1 x2 y1 y2 =
    ((abs(x1-x2)) + (abs(y1 -y2)))
  in let heuristic (St(state)) =
    let colonna x = x mod 4
    in let riga x = x / 4
    in let rec aux_Hn index count_Hn = function
      [] -> count_Hn
      | num::rest ->
        if num = 0 then
          aux_Hn (index+1) (count_Hn) rest
        else
          aux_Hn (index+1) (count_Hn + (distanzaManhattan (colonna index) (colonna (num-1)) (riga
index) (riga (num-1)))) rest
    in aux_Hn 0 0 state
  in (heuristic (List.hd cammino));;

```

Figura 2.16: Funzione di valutazione basata su distanza di Manhattan.



### 2.3.2 A\*

Per quanto riguarda l'algoritmo **A\*** la ricerca viene effettuata espandendo per primi i nodi secondo l'ordine dato da una funzione di valutazione che non è costituita soltanto dall'euristica. Per ordinare la coda di priorità in questo algoritmo, si utilizzano in combinazione sia una **funzione euristica  $h(n)$**  che una **funzione costo di cammino  $g(n)$** . La funzione di valutazione  **$f(n)$**  è quindi uguale a  **$h(n) + g(n)$** . Nelle due figure successive sono riportate le due funzioni utilizzando le stesse euristiche utilizzate per la ricerca Best First. L'unica differenza è che il valore precedente è ora sommato al valore di costo del cammino (ovvero al costo per arrivare fino allo stato che si sta valutando, in questo caso corrisponde al numero di mosse effettuate).

```
(* A* utilizzando caselle fuori posto + costo cammini *)
(* stato list -> int *)
let distanzaA_Misplaced cammino =
  let misplaced x index =
    if (x = (index + 1)) then 0 else 1
  in let heuristic (St(state)) =
    let rec aux_Hn index count_Hn = function
      [] -> count_Hn
      | num::rest ->
        if num = 0 then
          aux_Hn (index+1) (count_Hn) rest
        else
          aux_Hn (index+1) (count_Hn + (misplaced num index)) rest
    in aux_Hn 0 0 state
  in let costocammino cammino =
    let rec aux_Gn count_Gn = function
      [] -> count_Gn
      | x::rest -> aux_Gn (count_Gn + 1) rest
    in aux_Gn 0 cammino
  in (heuristic (List.hd cammino)) + (costocammino cammino);;
```

Figura 2.17: Funzione di valutazione basata sul numero di caselle fuori posto e il costo di cammino.

```

(* A* utilizzando Manhattan + costo cammini *)
(* stato list -> int *)
let distanzaA_Manhattan cammino =
  let distanzaManhattan x1 x2 y1 y2 =
    ((abs(x1-x2)) + (abs(y1-y2)))
  in let heuristic (St(state)) =
    let colonna x = x mod 4
    in let riga x = x / 4
    in let rec aux_Hn index count_Hn = function
      [] -> count_Hn
      | num::rest ->
        if num = 0 then
          aux_Hn (index+1) (count_Hn) rest
        else
          aux_Hn (index+1) (count_Hn + (distanzaManhattan (colonna index) (colonna (num-1)) (riga
index) (riga (num-1)))) rest
    in aux_Hn 0 0 state
  in let costocammino cammino =
    let rec aux_Gn count_Gn = function
      [] -> count_Gn
      | x::rest -> aux_Gn (count_Gn + 1) rest
    in aux_Gn 0 cammino
  in (heuristic (List.hd cammino)) + (costocammino cammino);;

```

Figura 2.18: Funzione di valutazione basata su distanza di Manhattan e il costo di cammino.

Come si può verificare in Tabella 3.1 nel Capitolo 3, nonostante l'algoritmo A\* con l'utilizzo delle distanze di Manhattan e il costo dei cammini abbia una buona convergenza verso la soluzione per la soluzione del gioco del 15 non è sufficiente. Infatti, per problemi del puzzle con dimensione inferiore, la distanza di Manhattan è più che sufficiente, tuttavia per un puzzle 4x4 l'algoritmo (per istanze da risolvere particolarmente difficili) può impiegare molto tempo alla risoluzione. Si necessita quindi di un'euristica in grado di approssimare ancora meglio il costo della soluzione, permettendo all'algoritmo di convergere più velocemente. Ho così deciso di implementare l'euristica dei **conflitti lineari**:

*due tessere 'a' e 'b' sono in conflitto lineare se si trovano nella stessa riga o colonna, anche le loro posizioni di obiettivo sono nella stessa riga o colonna e la posizione di obiettivo di una delle tessere è bloccata dall'altra tessera in quella riga (sono in posizione invertita rispetto al loro obiettivo).*

Un conflitto lineare fa sì che due mosse vengano aggiunte al valore euristico finale ( $h$ ), ciò perchè una tessera deve spostarsi per far posto alla tessera che ha lo stato obiettivo dietro la tessera spostata (oppure girarvi intorno) risultando in 2 mosse che mantengono l'ammissibilità dell'euristica. Il conflitto lineare è sempre combinato con la distanza di Manhattan e ogni conflitto lineare aggiungerà 2 mosse alla distanza di Manhattan, quindi il valore  $h(n)$  è:

**Distanza Manhattan + 2\*numero di conflitti lineari.**

Il conflitto lineare combinato con la distanza di Manhattan è significativamente più veloce dell'euristica precedente e gli enigmi 4x4 possono essere risolti usandolo in un discreto lasso di tempo. Proprio come in precedenza, non si considera la tessera vuota quando calcoliamo questa euristica. In Figura 2.19 è mostrata la funzione per calcolare il valore  $f(n) = h(n) + g(n)$  utilizzando questa nuova definizione. Infine nel Capitolo 3 verranno messe a confronto le prestazioni dei vari algoritmi con le diverse funzioni di valutazione.

```
(* A* utilizzando linear conflict + Manhattan + costo cammini *)
(* stato list -> int *)
let distanzaA_conflict cammino =
  let distanzaManhattan x1 x2 y1 y2 =
    ((abs(x1-x2)) + (abs(y1-y2)))
  in let heuristic (St(state)) =
    let colonna x = x mod 4
    in let riga x = x / 4
    in let rec aux_Hn index count_Hn = function
      [] -> count_Hn
      | num::rest ->
        if num = 0 then
          aux_Hn (index+1) (count_Hn) rest
        else
          aux_Hn (index+1) (count_Hn
            + (distanzaManhattan (colonna index) (colonna (num-1)) (riga index) (riga (num-1)))
            + (checkLinearConflict num index (St(state)))) rest
    in aux_Hn 0 0 state
  in let costocammino cammino =
    let rec aux_Gn count_Gn = function
      [] -> count_Gn
      | x::rest -> aux_Gn (count_Gn + 1) rest
    in aux_Gn 0 cammino
  in (heuristic (List.hd cammino)) + (costocammino cammino);;
```

Figura 2.19: Funzione di valutazione basata su distanza di Manhattan + Linear Conflict e il costo di cammino.

```

(* int -> int -> int -> int -> int *)
let checkConflict num1 num2 index1 index2 =
  let colonna x = x mod 4
  in let riga x = x / 4
  in let rigaGoal x = (x - 1) / 4
  in let colonnaGoal x = (x - 1) mod 4
  in let posizioneGoal x = x - 1
  in if (((riga index1 = riga (index2)) && ((rigaGoal num1 = rigaGoal num2))) ||
    ((colonna index1 = colonna (index2))) && (colonnaGoal num1 = colonnaGoal num2)) &&
    ((index1 > index2 && ((posizioneGoal num1) < (posizioneGoal num2)))))
  then 2 else 0 ;;

(* int -> int -> stato -> int *)
let checkLinearConflict x index (St(state)) =
  let rec aux count indexY = function
    [] -> count
  | hd::tail ->
    if indexY = index then
      count
    else
      if hd = 0 then
        aux (count) (indexY+1) tail
      else
        aux (count + checkConflict x hd index indexY) (indexY+1) tail
  in aux 0 0 state;;

```

Figura 2.20: Funzioni di comodo per calcolare i conflitti lineari.

# Capitolo 3

## Test del programma

In questo capitolo sono riportati degli esempi di istanze di test con il cammino per la soluzione di diversa lunghezza. Inizialmente ho voluto paragonare le performance dell'algoritmo Best First con A\* utilizzando le prime due euristiche implementate. Ho deciso di preparare 6 istanze di test con difficoltà crescente in modo da valutare e confrontare le performance di risoluzione dell'algoritmo. Successivamente saranno mostrate altre 2 istanze particolarmente difficili e saranno risolte con l'euristica dei conflitti lineari per mostrarne il netto aumento delle performance.

### 3.1 Istanze di test

Qui di seguito sono riportate 6 istanze di test ottenute muovendo casualmente la cella vuota per 5, 10, 15, 20, 25 e 35 mosse.

- **Stato obiettivo:** let statoIniziale = St[1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;0];;
- **5 mosse:** let statoIniziale = St[1;0;3;4;5;2;6;7;9;10;11;8;13;14;15;12];;
- **10 mosse:** let statoIniziale = St[5;1;3;4;2;10;6;7;9;14;11;8;13;0;15;12];;
- **15 mosse:** let statoIniziale = St[5;1;3;4;2;10;0;7;9;14;6;11;13;15;12;8];;
- **20 mosse:** let statoIniziale = St[5;1;4;7;2;10;3;11;9;14;0;6;13;15;12;8];;
- **25 mosse:** let statoIniziale = St[5;1;4;7;2;10;3;11;9;14;0;8;13;15;6;12];;
- **35 mosse:** let statoIniziale = St[5;10;1;7;14;0;4;11;2;15;3;12;9;13;8;6];;

## 3.2 Confronto vari algoritmi

Ho quindi testato le varie istanze con la ricerca Best First e A\* con le varie euristiche. Ovviamente il campione di test non è significativo e i risultati non sono attendibili, questo confronto è comunque una buona metrica per identificare le prime differenze. Nella Tabella 3.1 ho riportato la lunghezza del cammino e il tempo necessario alla risoluzione del problema. Qui di seguito è presente una legenda per leggere al meglio la tabella stessa:

- **BF\_mis**: ricerca Best First con euristica caselle fuori posto;
- **BF\_man**: ricerca Best First con euristica delle distanze di Manhattan;
- **A\*\_mis**: ricerca A\* con euristica caselle fuori posto + costo di cammino;
- **A\*\_man**: ricerca A\* con euristica delle distanze di Manhattan + costo di cammino.

profondità soluzione	BF_mis	BF_man	A*_mis	A*_man
<b>5 mosse</b>	immediata, cammino ottimo	immediata, cammino ottimo	immediata, cammino ottimo	immediata, cammino ottimo
<b>10 mosse</b>	immediata, cammino con 16 mosse	immediata, cammino ottimo	immediata, cammino ottimo	immediata, cammino ottimo
<b>15 mosse</b>	> 5 min, no risoluzione	immediata, cammino ottimo	immediata, cammino ottimo	immediata, cammino ottimo
<b>20 mosse</b>	no risoluzione	1,30sec, cammino con 30 mosse	9sec, cammino ottimo	immediata, cammino ottimo
<b>25 mosse</b>	no risoluzione	2sec, cammino con più di 50 mosse	400sec, cammino ottimo	2sec, cammino ottimo
<b>35 mosse</b>	no risoluzione	no risoluzione	no risoluzione	97sec, cammino ottimo

Tabella 3.1: Comparazione performance algoritmi di ricerca.

Come si può notare all'aumentare della difficoltà dell'istanza di input i vari algoritmi reagiscono più e meno bene. Per quanto riguarda l'euristica migliore si può vedere come utilizzando le distanze di Manhattan si converga prima alla soluzione. Per quanto riguarda l'algoritmo di ricerca,  $A^*$  riesce a trovare i cammini ottimi in quanto può sfruttare la funzione di costo di cammino la quale influisce sulla scelta dei cammini più corti.

Tuttavia è possibile migliorare ulteriormente i risultati utilizzando l'euristica relativa ai conflitti lineari con l'algoritmo  $A^*$ . Ho preparato quindi altre due istanze ancora più difficili e le ho provate a risolvere utilizzando l'algoritmo in questione.

- **40 mosse:** let statoIniziale = St[14;5;10;7;0;4;1;11;2;15;3;12;9;13;8;6];;
- **45 mosse:** let statoIniziale = St[14;5;10;7;4;15;1;11;2;13;3;12;9;8;6;0];;

Qui di seguito è riportato il confronto del miglior  $A^*$  precedente con questa nuova implementazione  **$A^*_\text{conflict}$**

profondità soluzione	<b><math>A^*_\text{conflict}</math></b>	<b><math>A^*_\text{man}</math></b>
<b>35 mosse</b>	29 sec, cammino ottimo	97sec, cammino ottimo
<b>40 mosse</b>	67 sec, cammino ottimo	> 10 min, no risoluzione
<b>45 mosse</b>	114 sec, cammino ottimo	> 10 min, no risoluzione

Tabella 3.2: Comparazione performance algoritmi di ricerca.

Come si può vedere le prestazioni sono particolarmente migliori, con questa euristica è possibile risolvere il puzzle in un tempo accettabile.

## Capitolo 4

### Conclusioni

Il programma implementato è quindi in grado di risolvere il gioco del 15. L'algoritmo migliore per la risoluzione risulta essere  $A^*$  utilizzando come euristica la distanza di Manhattan + i conflitti lineari e come  $g(n)$  il costo del cammino. Ovviamente più la soluzione dell'istanza iniziale è lunga più sarà il tempo necessario alla risoluzione, crescendo con un ordine di grandezza esponenziale. In alcuni casi non si arriva ad una soluzione in tempo utile in quanto la memoria richiesta per mantenere la coda di priorità è esponenziale. Un algoritmo come  $A^*$  è in grado di risolvere ogni istanza del gioco del puzzle con 9 caselle in pochissimo tempo, all'aumentare della dimensione (come nel mio caso) il costo della memoria diventa enorme e non sempre si arriva ad una soluzione immediata. È consigliabile risolvere il gioco del puzzle per dimensioni superiori a 3 con algoritmi che fanno uso di meno memoria, come l'algoritmo  $IDA^*$  che espande iterativamente la profondità del cammino di ricerca. Altri approcci per migliorare il tempo di risoluzione sono relativi all'utilizzo di altre euristiche come ad esempio la tecnica del Pattern Database.