UNIVERSITÀ DEGLI STUDI DI UDINE
HIC SUNT FUTURA

DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE

TESI DI LAUREA IN
INTERNET OF THINGS, BIG DATA AND MACHINE LEARNING

# Instradamento di permutazioni su grafi tramite accoppiamenti con applicazione al gioco del 15

CANDIDATO
Aleksa Aleksic

RELATORE
Prof.ssa Carla Piazza

CORRELATORE
Dott. Riccardo Romanello

Anno accademico 2024-2025

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

A mamma e papà, ai sacrifici fatti per consertirmi di scrivere oggi questo pezzo di carta.

# Ringraziamenti

Alle mie sorelle, Marija e Kristina. A Milan e Bogo. Ai ragazzi di UDVP, a Nick, a Pol, a Pit, a Bonni, a Fil Sus, a Gio, a Prince, a Tonno. A Mansu, a Albe, a Thomas. Ai ragazzi di Lenny, a Gioele, a Cri, a Bubu, a Milo, a Daniel, a Stefano, a Edo. A chiunque abbia preso parte, in un modo o nell'altro, a questo percorso, grazie.

# Indice

# Elenco delle figure

# Elenco degli algoritmi

# 1

# Introduzione

## 1.1 Introduzione generale

Nella stesura di questo lavoro di tesi il punto di partenza è stato lo studio della pubblicazione Routing permutations on graphs via matchings" di Noga Alon, F.R.K. Chung e R.L. Graham [1]. In particolare, ci siamo concentrati sulla parte introduttiva relativa al problema delle permutazioni di instradamento su grafi.

A partire da questo spunto, l'obiettivo principale è stato quello di indagare se e in che modo fosse possibile ricondurre il gioco del 15 al quadro teorico delineato dagli autori, così da ottenere strumenti utili per affrontarne la risoluzione (3). Per farlo, è stato necessario innanzitutto definire in modo rigoroso le basi teoriche del problema: dalla formalizzazione delle permutazioni e della loro parità (3), alla modellizzazione del puzzle come grafo, fino ai richiami essenziali alla teoria della complessità (5) e alla caratterizzazione delle classi Pe NP. (5)

Su queste fondamenta si è poi sviluppata l'analisi delle strategie risolutive: dagli algoritmi di ricerca euristica (A*, IDA*) alle ottimizzazioni basate su euristiche come la distanza di Manhattan e i conflitti lineari (4), fino a un approccio alternativo basato su Answer Set Programming (8).

## 1.2 Concetti fondamentali

### 1.2.1 Introduzione alle permutazioni [13]

Una permutazione è un modo di riordinare tutti gli elementi di un insieme. Ad esempio, se l'insieme è $A, B, C$, le permutazioni possibili sono:

$$ABC, ACB, BAC, BCA, CAB, CBA$$

Ogni permutazione corrisponde a una disposizione in cui ciascun elemento compare esattamente una volta.

Formalmente, data una quantità finita $n$, una permutazione su $n$ elementi può essere vista come una funzione biiettiva:

$$\pi : 1, 2, ..., n \to 1, 2, ..., n$$

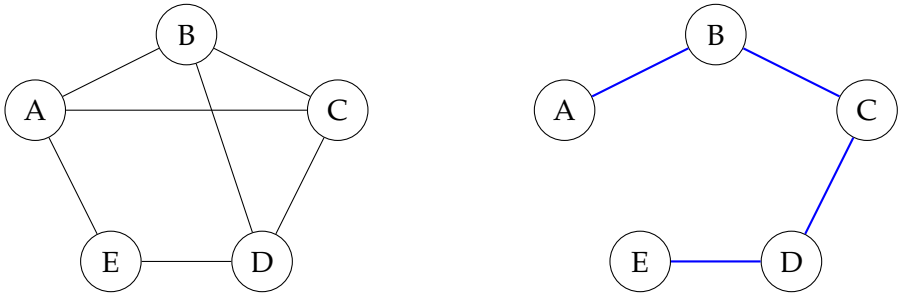
cioè una corrispondenza uno-a-uno da un insieme su sé stesso. In questo caso, il numero totale di permutazioni è $n!$.

### 1.2.2 Spanning tree [4]

Dato un grafo connesso e non orientato $G = (V, E)$, uno spanning tree è un sottografo $T = (V, E)$ tale che:

- $E_T \subseteq E$;
- $T$ è un albero (connesso e aciclico);
- $|E_T| = |V| - 1$;

Uno spanning tree è un sottografo di un grafo connesso che contiene tutti i vertici e li collega senza formare cicli. In altre parole, è una "struttura scheletro" del grafo: mantiene i collegamenti tra tutti i nodi, ma con il numero minimo di archi necessario per rimanere connesso. Ad esempio, se ho un grafo con 4 nodi collegati in rete e vari archi ridondanti, uno spanning tree è una selezione di 3 archi che collega comunque tutti i nodi ma elimina i cicli.




Grafo $G$ Spanning tree di $G$

Figura 1.1: Confronto tra grafo originale e il suo spanning tree

## 1.3 Formalizzazione del problema [1]

Immaginiamo un grafo $G$, i cui nodi sono colorati. Su ciascun nodo è collocata una pedina, anch'essa caratterizzata da un colore, che può coincidere oppure no con quello del nodo che la ospita. Le posizioni delle pedine possono essere interpretate come una permutazione dei nodi del grafo, poiché ogni pedina occupa un nodo distinto e non ci sono sovrapposizioni. Per formulare il problema in maniera rigorosa, fissiamo quindi le seguenti definizioni:

- $G = (V, E)$ un grafo connesso con self-loop in cui $|V| = n$;
- $\pi(v)$ rappresenta la permutazione finale associata al nodo $v$ del grafo;
- $\pi_0(v)$ rappresenta la permutazione iniziale associata al nodo $v$ del grafo;

Si ottiene, quindi, che $\pi$ rappresenta l'insieme delle permutazioni finali (destinazioni) di ogni nodo del grafo e $\pi_0$ rappresenta l'insieme delle permutazioni iniziali (posizioni di partenza) dei nodi del grafo. Vogliamo che a ogni passo valga la seguente relazione:

$$\forall i \forall v (\pi_i(v), \pi_{i+1}(v)) \in E$$

tenendo conto che $\pi_i(v)$ rappresenta la permutazione del nodo $v$ al tempo $i$ e $i+1$, questo vuol dire che una pedina può solo essere scambiata con una pedina adiacente ad essa, quindi la permutazione del nodo $v$ al tempo $i$ e quella al tempo $i+1$ formano un arco del grafo.

Definiamo:

- $P_v(t)$: posizione della pedina con posizione iniziale $v$ al tempo $t$;
- $P_v(t) = \pi_t(\pi_0^{-1}(v))$;
- dove $\{P_v(t) : v \in V\}$ è una permutazione.

Otteniamo quindi il numero minimo di passi per passare da $\pi_0$ a $Id$:

$$rt(G, \pi_0)$$

Di conseguenza definiamo come routing number il valore:

$$rt(G) = \max_{\pi_0} rt(G, \pi_0)$$

come il numero massimo di mosse necessarie per riportare ogni pedina nella sua posizione ottimale con la sequenza più breve possibile.

# 2

# Esistenza del routing number

È possibile dimostrare che il routing number (1.3) esiste sempre. Per compiere tale dimostrazione andiamo a compiere:

- assumiamo che ci venga fornito $T$, un qualunque spanning tree del grafo $G$;
- cerchiamo un nodo foglia $u$ del grafo $T$;
- cerchiamo un nodo che abbia all'interno la pedina che ha come destinazione finale il nodo $u$, ovvero cerchiamo $v$ tale che $\pi(v) = u$;
- portiamo il nodo $v$ a destinazione $v$ scambiando a ogni passo i nodi, in modo che $P_v(\alpha) = \pi(v) = u$ dove $\alpha$ è il tempo finale;
- una volta portato il nodo a destinazione, consideriamo ricorsivamente il grafo $T - \{u\}$;

Per ottimizzare i tempi attuiamo due accorgimenti:

- La prima funzione necessaria è quella che riguarda il recupero di un nodo foglia. Per ottimizzare i tempi possiamo andare ad utilizzare una struttura dati ausiliaria, ad esempio uno stack, andiamo quindi a ordinare i nodi in base al loro grado e andiamo a inserirli in ordine inverso nello stack. In questo modo il primo elemento dello stack è sempre un nodo foglia. Siccome i nodi di un grafo hanno un grado che può essere $[1, n-1]$ possiamo usare l'algoritmo di ordinamento *counting sort* in modo da ordinare la coda in tempo lineare ($O(n)$). A ogni iterazione estraiamo il primo elemento della coda (con un pop()), in modo che venga anche eliminato dalla coda in quanto "sistemato";
- Per ottimizzare il tempo che ci mettiamo a recuperare il nodo che ha come destinazione un nodo $u$, costruiamo un dizionario dove usiamo i nodi come indici e otteniamo per ogni nodo $u$ il nodo che contiene la pedina che deve trovarsi in quel nodo alla fine dell'algoritmo, cioè $v$ tale che $\pi(v) = u$. Supponiamo quindi che $dest[]$ sia un dizionario in modo tale da avere $dest[u] = v$. In questo modo il recupero di tale informazione ha un costo pari a $O(1)$.

Definiamo, a tale scopo, lo pseudocodice per fare quanto appena descritto.

**Algoritmo 1** Calcolo del routing number su uno spanning tree $T$

**Input:** $T$: spanning tree di $G$, $dest$: mappa delle destinazioni finali, $foglie$: coda di nodi ordinata in base al grado dei nodi
**Output:** Numero totale di passi necessari (routing number)

```
1:  function routingNumber(T, foglie, dest)
2:      passi ← 0
3:      if |V(T)| = 0 then
4:          return passi
5:      else
6:          u ← foglie.pop()
7:          v ← dest[u]
8:          passi ← BFS_modificato(T, u, v)
9:          T ← T − {u}
10:         return passi + routingNumber(T, foglie, dest)
11:     end if
12: end function
```

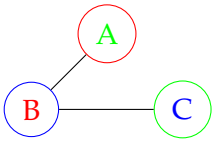



Figura 2.1: Disposizione iniziale del grafo T

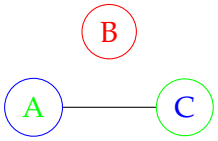



Figura 2.2: Disposizione dopo il primo passo

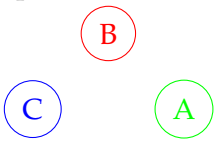



Figura 2.3: Disposizione finale

Esempio di funzionamento dell'algoritmo

### 2.0.1 Funzioni ausiliarie

Perché il tutto (algoritmo 1) funzioni, definiamo le funzioni ausiliarie.

**Algoritmo 2** BFS Modificato per portare il nodo $v$ a destinazione $u$

**Input:** $T$: grafo rappresentato con lista di adiacenza, $u$: nodo destinazione, $v$: nodo iniziale
**Output:** Numero di passi necessari per portare $v$ a $u$

```
1: function BFS_MODIFICATO(T, u, v)
2:     passi ← 0
3:     coda ← Queue()
4:     coda.enqueue(v)
5:     visitato ← Set()
6:     while coda is not empty do
7:         nodo_corrente ← coda.dequeue()
8:         visitato.add(nodo_corrente)
9:         if nodo_corrente = u then
10:            return passi
11:        end if
12:        for all w ∈ Adj(nodo_corrente) do
13:            passi ← passi + 1
14:            coda.enqueue(w)
15:            T ← swap(T, nodo_corrente, w)
16:        end for
17:    end while
18: end function
```

**Algoritmo 3** Funzione SWAP per scambiare le chiavi dei nodi

**Input:** $T$: albero, $nodo\_corrente$: nodo corrente, $w$: nodo con cui scambiare la chiave
**Output:** Albero $T$ con chiavi scambiate

```
1: function MODIFICAALBERO(T, nodo_corrente, w)
2:     temp ← nodo_corrente.chiave
3:     nodo_corrente.chiave ← w.chiave
4:     w.chiave ← temp
5: end function
```

## 2.1 Calcolo della Complessità

L'algoritmo descritto (algoritmo 1) ha una complessità temporale che dipende dalla struttura del grafo e dal numero di nodi $n$. In particolare, possiamo calcolare la complessità come segue:

- Il calcolo del routing number, che involve una BFS per ogni nodo, ha una complessità di $O(n+m)$, dove $n$ è il numero di nodi e $m$ il numero di archi nel grafo.
- In generale, il numero massimo di passi necessari per raggiungere la configurazione finale è di $O(n+m)$, poiché in ogni passo la ricerca di del nodo destinazione viene svolta in $O(1)$ dunque il costo dell'algoritmo equivale al costo della BFS sui nodi.

Pertanto, la complessità complessiva dell'algoritmo è $O(n + m)$, dove $n$ è il numero di nodi nel grafo e $m$ è il numero di archi.

# 3

# Modellazione del gioco del 15 nel contesto di lavoro

Supponiamo di voler modellare il gioco del 15 all'interno del contesto attuale. Possiamo immaginare di avere a che fare con un grafo a reticolo formato da 16 nodi. Supponiamo inoltre che su ogni nodo sia presente un numero da 1 a 15 e che un solo nodo di quelli del grafo assuma il valore di 16. Quello che vogliamo fare è trovare un limite di tempo nel quale riusciamo a portare ogni nodo nella posizione attesa potendo però scambiare solo i nodi adiacenti al 16 con il nodo che lo contiene.

Definiamo quindi:

- un grafo finito e indiretto $G = (V, E)$;
- una funzione $f : V \to \{1, 2, 3, 4, ..., 16\}$.

Supponiamo, inoltre, che a ogni nodo venga assegnato un valore casuale. Definiamo quindi:

- $\pi_0$ la permutazione iniziale delle chiavi nei nodi del grafo;
- $\pi$ la permutazione finale delle chiavi nei nodi;

supponiamo, inoltre, che l'insieme delle permutazioni dei nodi sia ordinato rispetto ai nodi, ad esempio la chiave in prima posizione è situata nel primo nodo ecc...

Definiamo in tal caso $P_v(t)$ la permutazione del nodo $v$ al tempo $t$, ovvero il valore che tale nodo assume al tempo $t$. In questo modo possiamo definire il nodo "vuoto" come il nodo che ha:

$$P_v(t) = 0$$

ipotizzando, come fatto in precedenza, che l'unica mossa possibile sia quella che scambia due nodi adiacenti. In questo caso, inoltre, aggiungiamo tra i vincoli anche che gli unici due nodi che possono essere scambiati sono quelli in cui almeno uno dei due è il nodo vuoto.

Definiamo quindi il concetto di mossa legale come quella mossa che consiste nel muovere una pedina, lungo i vertici, nel nodo vuoto.

$$\exists u, v \in V \text{ t.c. } (u, v) \in E \text{ e } P_u(t) = P_v(t - 1),$$

$$\text{e } P_v(t) = P_u(t-1) = 0 \text{ e } P_w(t-1) = P_w(t)\ \forall w \in V \setminus \{u, v\}.$$

Siccome a ogni mossa scambio due nodi del grafo possiamo vedere una mossa legale come una trasformazione legale del piano di gioco. Ovvero, possiamo immaginare che se al tempo $i$ i nodi del grafo assumono la configurazione $\pi_i$, allora al tempo $i+1$, dopo una mossa legale, assumeranno la configurazione $\pi_{i+1}$.

Possiamo dunque definire una sequenza di configurazioni come:

$$S_m = \{\pi_0, \pi_1, ..., \pi_t\}$$

e possiamo intendere $S_m$ come una sequenza di configurazioni legali se per ogni $i$ abbiamo che $\pi_i$ è una mossa legale applicata a $\pi_{i-1}$. A questo punto definiamo una coppia di configurazioni $\pi_i, \pi_j$ ha una soluzione se $\exists$ una sequenza di configurazioni che porta da $\pi_i$ a $\pi_j$.

## 3.1 Esistenza di una soluzione al gioco del 15

### 3.1.1 Introduzione

Questo capitolo affronta il problema della risolubilità del gioco del 15 attraverso un approccio multidisciplinare che combina teoria matematica e algoritmica computazionale. L'analisi si articola in quattro sezioni principali che esplorano progressivamente gli aspetti teorici e pratici del problema.

La prima sezione stabilisce i fondamenti matematici necessari attraverso lo studio delle permutazioni e della loro parità.

La seconda sezione esamina gli approcci algoritmici classici per la risoluzione automatica del puzzle. Vengono analizzati gli algoritmi di ricerca euristici A* e IDA*.

La terza sezione presenta un tentativo di applicazione della teoria degli spanning tree al problema, esplorando se sia possibile scomporre la risoluzione in sottoproblemi indipendenti.

La quarta sezione conclude con la descrizione di strategie pratiche di risoluzione che, pur non garantendo l'ottimalità, forniscono un compromesso efficace tra complessità computazionale e qualità della soluzione.

### 3.1.2 Permutazioni [13]

Riprendendo l'introduzione alle permutazioni (sezione 1.2.1) definiamo i concetti matematici fondamentali per poter calcolare l'esistenza di una soluzione.

Una permutazione di un insieme di oggetti è un riarrangiamento di questi ultimi in un particolare ordine. Dal punto di vista matematico possiamo definire una permutazione come:

**Definizione 1.** *Una permutazione di un insieme A è una funzione $\sigma_1 : A \to A$ biettiva.*

Dato che siamo interessati ai puzzle, ciò che ci interessa è capire come si muovono i pezzi del puzzle in giro per la scacchiera, quindi rappresentiamo ogni pezzo con un numero, ovvero $[n] = \{1, 2, 3, 4, ..., 15\}$. Un riarrangiamento di questi valori corrisponde a una permutazione per come definita sopra (3).

Siccome le permutazioni sono tipicamente definite su insiemi finiti, rappresentiamo una permutazione semplicemente indicando in che posizione si sposta un valore. Ad esempio $\sigma_1(1) = 2$, o più comodamente, se $\{1, 2, 3\}$ allora:

$$\sigma_1 \leftrightarrow \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$$

quest'ultima è detta notazione ad array di una permutazione. In generale definiamo una permutazione $\sigma_1 : [n] \to [n]$ con u array $2 \times n$:

$$\sigma_1 \leftrightarrow \begin{pmatrix} 1 & 2 & ... & n \\ \sigma_1(1) & \sigma_1(2) & ... & \sigma_1(n) \end{pmatrix}$$

In modo simile possiamo rappresentare una permutazione con un diagramma ad arco, sia: $\sigma = [\sigma(1), \sigma(2), ..., \sigma(n)] = [2, 9, 5, 6, 3, 4, 7, 1, 8]$ una permutazione dell'insieme $\{1, 2, 3, ..., 9\}$, si ottiene:

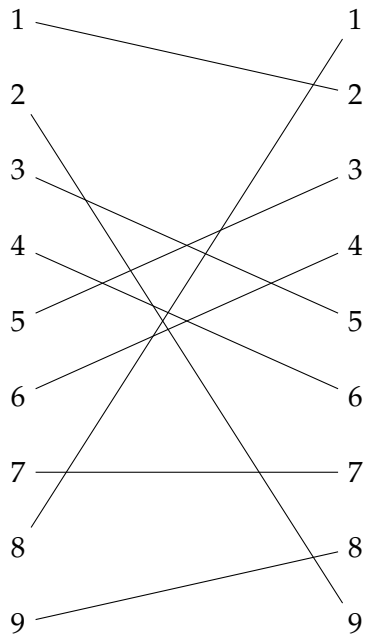



Figura 3.1: Rappresentazione di una permutazione con diagramma ad arco.

Due permutazioni speciali sono la permutazione identità e l'n-ciclo. La prima non fa nulla, ovvero permuta ogni elemento in se stesso, la seconda permuta gli elementi in modo ciclico; ogni elemento viene permutato di una posizione a destra e l'ultimo va in prima posizione.

$$\epsilon \leftrightarrow \begin{pmatrix} 1 & 2 & ... & n-1 & n \\ 1 & 2 & ... & n-1 & n \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & ... & n-1 & n \\ 2 & 3 & ... & n & 1 \end{pmatrix}$$

**Definizione 2.** *Date due permutazioni $\sigma_1, \sigma_2$ la composizione (moltiplicazione) di esse restituisce una terza*

*permutazione, applicando la seconda permutazione all'applicazione della prima su un elemento $k$ dell'insieme:*

$$(\sigma_1 \circ \sigma_2)(k) = \sigma_2(\sigma_1(k))), per\ k \in [n]$$

**Definizione 3.** *Due permutazioni $\sigma_1, \sigma_2$ sono dette inverse se $\sigma_1\sigma_2 = \epsilon$.*

Quest'ultima definizione è particolarmente importante perché permette di disfare una mossa del puzzle.

**Definizione 4.** *Per ogni permutazione $\sigma_1$, esiste ed è unica una permutazione $\sigma_2$ tale che $\sigma_1\sigma_2 = \sigma_2\sigma_1 = \epsilon$, in questo caso $\sigma_2 = \sigma_1^{-1}$.*

Queste definizioni valgono anche per i prodotti (composizioni) di permutazioni:

$$((\sigma_1)(\sigma_2)...(\sigma_k))^{-1} = (\sigma_k)^{-1}...(\sigma_2)^{-1}(\sigma_1)^{-1}$$

**Definizione 5.** *L'insieme di tutte le permutazioni dell'insieme $[n]$ è detto gruppo simmetrico di grado $n$ ed è indicato con $S_n$.*

$$S_n = \{\sigma_1 | \sigma_1\ é\ una\ permutazione\ di\ [n]\}$$

Quando si parla di mosse di un puzzle spesso si usano le notazioni esponenziali, $\sigma_1^{-1}$ si indica con la permutazione inversa di $\sigma_1$, $\sigma_1^m$ vuol dire la permutazione $\sigma_1$ applicata $m$ volte.

**Definizione 6.** *Per ogni $\sigma_1 \in S_n$ esiste un intero positivo $m$ tale che $\sigma_1^m = \epsilon$, il minor $m$ tale che questa relazione valga è detto ordine di $\sigma_1$ e denotato con $ord(\sigma_1)$.*

Tornando all'insieme $[n] \to \{1, 2, 3, ..., 9\}$ se $\sigma = [\sigma(1), \sigma(2), ..., \sigma(n)] = [2, 9, 5, 6, 3, 4, 7, 1, 8]$ allora possiamo indicare tale permutazione con una notazione detta ciclica:

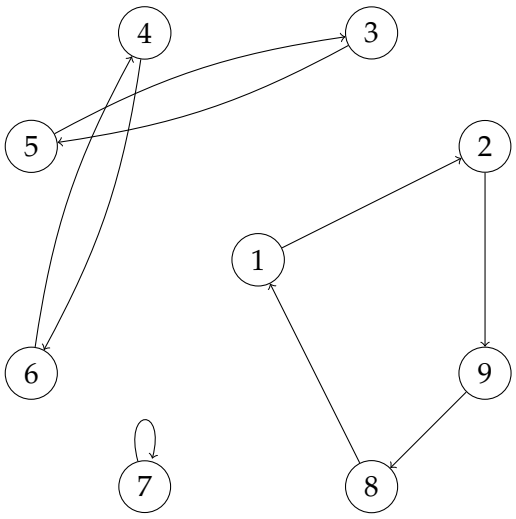



Figura 3.2: Notazione ciclica.

che si ottiene scrivendo gli elementi di ogni ciclo. Ad esempio nel caso descritto sopra definiamo:

$$(a_1, ..., a_{l_1})(a_{l+1}, ..., a_{l_2})...() = (1, 2, 8, 9)(3, 5)(4, 6)(7).$$

Generalmente i cicli che contengono un solo elemento vengono ignorati, quindi ci rimane $(1, 2, 8, 9)(3, 5)(4, 6)$. Siccome una permutazione $(a_1, ..., a_m)$ è detta m-ciclo, in questo caso la permutazione è il prodotto di un 4-ciclo e di due 2-cicli. Per definire la notazione ciclica di una permutazione partiamo dal valore minore, in questo caso 1, scriviamo il numero in cui viene mappato, in questo caso $(1\ 2)$ e continuiamo finchè non chiudiamo il ciclo $(1\ 2\ 8\ 9)$.

Sia $\sigma = [\sigma(1), \sigma(2), ..., \sigma(n)] \in S_n$. Un'inversione è un paio di indici $i \leq j$ tali che $\sigma(i) > \sigma(j)$. Indichiamo $Inv(\sigma)$ l'insieme delle inversioni e $inv(\sigma)$ il numero di inversioni.

La fattorizzazione disgiunta di cicli è la fattorizzazione della permutazione in cicli nella notazione ciclica in cui nessun ciclo ha elementi in comune con un altro ciclo. Ad esempio se $f = (1, 2, 3)(4, 5, 6) \in S_6$ allora $f = gh$ con:

$$g = (1, 2, 3), f = (4, 5, 6).$$

Questo è particolarmente importante in quanto ogni permutazione può essere scritta come prodotto di cicli disgiunti. Ad esempio una trasposizione è un 2-ciclo.

**Definizione 7.** *Ogni permutazione in $S_n$ può essere espressa come prodotto di 2-cicli.*

Un k-ciclo $(a_1...a_{k-1}a_k)$, di dimensione $n$, può essere scritto come prodotto di trasposizioni, cioè $(a_1, ...a_{k-1}, a_k) = (a_1a_2)(a_1a_3)...(a_1a_{k-1})(a_1a_k)$.

*Dimostrazione.* Data una qualsiasi permutazione $\sigma_1 \in S_n$, tale permutazione può essere scritta come prodotto di cicli disgiunti

$$\sigma_1 = (a_1a_2...a_r)(b_1b_2...b_s)(c_1c_2...c_t)$$

e ogni ciclo può essere scomposto in un prodotto di due cicli seguendo la regola definita sopra (7)

$$\sigma_1 = (a_1a_2)(a_1a_3)(a_1a_r)(b_1b_2)(b_1b_3)(b_1b_s)(c_1c_2)(c_1c_3)(c_1c_t)$$

□

Siccome ogni permutazione può essere generata da cicli disgiunti e ogni k-ciclo può essere generato da trasposizioni, le trasposizioni generano $S_n$.

$$(1, 2)(2, 3)(3, 4) = (1, 2, 3, 4)$$

### 3.1.3 Parità delle permutazioni [13]

Ogni permutazione $\sigma_1$ può essere rappresentata come il prodotto di un numero $k$ di 2-cicli. Il numero di 2-cicli che si utilizzano per rappresentare la permutazione definisce la sua parità. Definiamo quindi due insiemi; l'insieme delle permutazioni pari $A_n$ e l'insieme delle permutazioni dispari $O_n$. Entrambi gli insiemi sono sottinsiemi dell'insieme di tutte le permutazioni $S_n$.

**Teorema 1.** *Il numero di mosse per passare da una permutazione a un altra può variare ma la parità rimane uguale.*

Questo vuol dire che se $\sigma_1$ è una permutazione allora:

$$\sigma_1 = \tau_1\tau_2...\tau_r = \sigma_1\sigma_2...\sigma_s$$

dove $\tau_i, \sigma_i$ sono 2-cicli e $r, s$ sono entrambi o pari o dispari. Ogni permutazione può essere rappresentata da un k-ciclo. Ogni k-ciclo può essere rappresentato come il prodotto di $m-1$ trasposizioni (2-cicli). Questo vuol dire che un $m$-ciclo è pari se $m$ è dispari ed è dispari se $m$ è pari. La correttezza di questo deriva da quanto definito sopra. (3.1.2) Considerando la permutazione base $\pi_0 = \{1, 2, 3\}$ e la permutazione finale $\pi = \{3, 2, 1\}$, rappresentabile volendo, come $(1, 3)(2)$, si ottiene che la permutazione finale è ottenibile scambiando semplicemente il 3 con l'1, quindi con un numero dispari di mosse, oppure contando le dimensione dei cicli $|(1, 3)| + |(2)| = 3$ che è dispari. Si dice quindi che la permutazione $\pi$ è dispari. Si può, però, considerare di passare da $\pi_0$ a $\pi$ facendo prima uno scambio tra 2 e 1, poi uno scambio tra 2 e 3 e infine uno scambio tra 1 e 2. In questo caso si ottiene lo stesso risultato ma con 3 mosse. Si nota che il modo per arrivare al risultato è cambiato ma la parità della permutazione è rimasta la stessa. Quindi se pensiamo a una trasposizione come ad una inversione, la parità di $\sigma$ dipende da $inv(\sigma)$

**Teorema 2.** *Ogni permutazione in $A_n$ può essere espressa come prodotto di 3-cicli.*

Se $\sigma_1$ è una permutazione pari, infatti, significa che può essere scritta come prodotto di un numero pari di 2-cicli. Una volta fatto ciò basta raggruppare coppie adiacenti di trasposizioni ottenendo la permutazione sotto forma di prodotto di 3-cicli.

### 3.1.4 Esistenza o meno della soluzione [13]

| | | | |
|---|---|---|---|
| **1** | **2** | **3** | **4** |
| **5** | **6** | **7** | **8** |
| **9** | **10** | **11** | **12** |
| **13** | **14** | **15** | **16** |

(a) Disposizione finale (obiettivo) del gioco del 15.

| | | | |
|---|---|---|---|
| **1** | **3** | **7** | **4** |
| **6** | **2** | **16** | **8** |
| **5** | **9** | **11** | **12** |
| **13** | **10** | **14** | **15** |

(b) Disposizione iniziale del gioco del 15.

L'idea di partenza del gioco è quella di iniziare da una disposizione casuale delle tessere e di riordinarle in modo da ottenere una disposizione voluta. Si possono, quindi, applicare le nozioni di permutazione

per capire se il problema è risolvibile oppure no. Dato un insieme finito di valori $S_n = \{1, 2, 3, ..., 16\}$ e un piano di gioco iniziale, il quale può essere visto come una permutazione del piano di gioco con i numeri iniziali, ci interessa capire se tale permutazione è risolvibile. Ci interessa, quindi, sapere se esiste un numero finito di mosse che porta dalla permutazione del piano di gioco alla soluzione. Data una disposizione iniziale delle pedine, che chiameremo $\sigma_0$, ciò che vogliamo capire è se dalla disposizione $\sigma_0$ si può arrivare alla disposizione $\sigma$, dove come $\sigma$ intendiamo la disposizione finale. Le permutazioni del piano di gioco $\sigma_0$ si possono dividere in due macrocategorie:

- le disposizioni che fissano la casella vuota nella posizione 16;
- le disposizioni in cui la casella vuota non si trova in posizione 16;

Da questa distinzione derivano due teoremi su cui ci basiamo per parlare di risolvibilità del gioco del 15.

**Teorema 3.** *Una permutazione $\sigma_1$ del gioco del 15, che fissa il 16 è risolvibile se e solo se la permutazione $\sigma_1$ è una permutazione pari.*

**Teorema 4.** *Una permutazione $\sigma_1$ del gioco del 15 che non fissa il 16 è risolvibile se e solo se la parità della permutazione coincide con la parità della distanza della pedina 16 dalla sua posizione obiettivo (cella 16).*

Se rappresentiamo la disposizione finale (figura 3.3a) delle tessere come:

$$\sigma = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]$$

allora la disposizione iniziale può essere rappresentata, ad esempio (figura 3.3b), come:

$$\sigma_0 = [1, 3, 7, 4, 6, 2, 16, 8, 5, 9, 11, 12, 13, 10, 14, 15]$$

. Per le proprietà descritte sopra (3.1.2) un insieme di trasposizioni, l'insieme delle trasposizioni può variare ma non la sua parità.

| | | | |
|---|---|---|---|
| **1** | **3** | **7** | **4** |
| **6** | **2** | **16** | **8** |
| **5** | **9** | **11** | **12** |
| **13** | **10** | **14** | **15** |

Figura 3.4: Posizione della cella vuota rispetto alla posizione obiettivo.

Se il 16 è stato spostato di un numero dispari di celle, allora solo un insieme dispari di trasposizioni può portare a tale permutazione. Il primo passo consiste nel calcolare la parità del numero di spostamenti compiuti dalla casella vuota (il valore 16) per passare dalla posizione in $\sigma$ alla posizione in $\sigma_0$ (figura 3.4). Indichiamo tale distanza con $d(\sigma, \sigma_0)$, in questo $d(\sigma, \sigma_0) = 3$ quindi è dispari.

Rappresentiamo la disposizione $\sigma_0$ in notazione ciclica, ottenendo:

$$(2, 3, 7, 16, 15, 14, 10, 9, 5, 6)(1)(4)(8)(11)(12)(13)$$

eliminando i cicli singoli otteniamo un unico ciclo di 10 elementi. Siccome ogni ciclo di dimensione $n$ può essere scritto come prodotto di $n - 1$ trasposizioni, possiamo andare a scriverla come prodotto di trasposizioni nel seguente modo:

$$(2, 3)(3, 7)(7, 16)(16, 15)(15, 14)(14, 10)(10, 9)(9, 5)(5, 6).$$

Notiamo che sono in tutto 9 trasposizioni, quindi un numero dispari di scambi. Abbiamo quindi dedotto che la disposizione $\sigma_0$ è stata ottenuta con un numero dispari di spostamenti della cella vuota. Sappiamo, inolte, che la permutazione $\sigma_0$ può essere generata da un $k-$ciclo con $k$ dispari, dunque tale permutazione è dispari. Siccome tale permutazione può essere ottenuta con un insieme di trasposizioni, e siccome le due parità coincidono, otteniamo che la permutazione $\sigma$ è raggiungibile dalla permutazione $\sigma$ ergo il gioco ha soluzione. Un secondo metodo per calcolare la parità della permutazione consiste nel contare il numero di inversioni. Abbiamo già definito cosa si intende con inversione, nell'esempio rappresentato otteniamo un totale di 21 inversioni, cioè un valore dispari. Questo metodo si presta meglio a una soluzione algoritmica. Consideriamo la disposizione sul piano di gioco come una permutazione delle celle che dove la cella contenente la pedina 16 è in posizione $z$, otteniamo:

$$a_1 a_2 a_3 ... a_{z-1} a_z a_{z+1} a_{15}$$

Sia $N$ il numero di inversioni. Sia $K$ l'indice della riga in cui si trova l'elemento vuoto. Definiamo il calcolo di $K$ come:

$$K = \lfloor z/4 \rfloor + 1$$

la soluzione, quindi, esiste se $K + N$ è pari, in quanto un numero pari può essere ottenuto come somma di due numeri pari o di due numeri dispari. In entrambi i casi si tratta di due numeri la cui parità coincide, che è ciò che vogliamo ottenere.

### 3.1.5 Algoritmo per il calcolo dell'esistenza di una soluzione [3]

**Algoritmo 4** Funzione per verificare se la disposizione del gioco del 15 ha soluzione

**Input:** $a$: array di 16 valori che indicano le chiavi contenute nelle 16 posizioni del piano di gioco

```
1:  function esistenzaSoluzione(a)
2:      inv ← 0
3:      for i ← 0 to 15 do
4:          if a[i] ≠ 0 then
5:              for j ← 0 to i − 1 do
6:                  if a[j] > a[i] then
7:                      inv ← inv + 1
8:                  end if
9:              end for
10:         end if
11:     end for
12:     for i ← 0 to 15 do
13:         if a[i] == 0 then
14:             inv ← inv + (1 + ⌊i/4⌋)
15:         end if
16:     end for
17:     if inv mod 2 == 0 then
18:         return 1
19:     else
20:         return 0
21:     end if
22: end function
```

A causa del doppio ciclo for utilizzato per contare le inversioni, la complessità di tale algoritmo è di $O(n^2)$.

## 3.2 Un primo approccio alla soluzione del puzzle

Un primo approcio alla soluzione, dove con soluzione intendiamo un algoritmo capace di portare una disposizione del 15 alla disposizione finale, se la disposizione di partenza accetta soluzione, prevede di utilizzare gli algoritmi di ricerca.

Quello che possiamo fare è spostare le pedine, passando da una configurazione a un'altra, seguendo un percorso dettato da qualche metrica che ci consenta di capire quando siamo più vicini alla soluzione.

Per farlo possiamo sfruttare degli algoritmi di ricerca sui grafi, vedendo il piano di gioco come un grafo e ogni nodo come una cella del piano.

### 3.2.1 Distanza di Manhattan [1]

Siccome gli algoritmi di ricerca utilizzano una funzione di costo, introduciamo il concetto di distanza di Manhattan che useremo per calcolare i costi. La **distanza di Manhattan** misura la distanza tra due

punti in una griglia considerando soltanto gli spostamenti orizzontali e verticali. Formalmente, dati due punti $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$, la distanza di Manhattan è definita come:

$$d(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2|.$$

Questa metrica è ampiamente utilizzata, ad esempio, nei puzzle a scorrimento (come il 15-puzzle) e nella teoria dei grafi su reticoli.

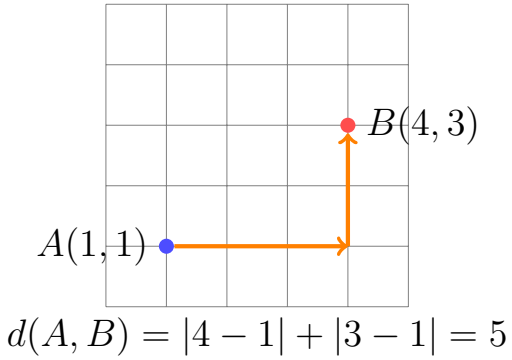



Figura 3.5: Esempio di distanza di Manhattan

### 3.2.2 Algoritmo A*

Gli algoritmi di ricerca su grafi come *Depth-First Search* (DFS), *Breadth-First Search* (BFS) e *best-first*, rappresentano approcci fondamentali alla risoluzione di problemi di esplorazione dello spazio degli stati. DFS esplora in profondità un cammino prima di tornare indietro, BFS visita i nodi per livelli, garantendo il ritrovamento del cammino più corto in grafi non pesati mentre best-first sceglie di espandere la ricerca verso il nodo più promettente, assegnando un punteggio a ciascun nodo [4]. Mentre i primi due metodi risultano inefficaci in presenza di costi associati agli archi o quando si desidera ridurre lo spazio di ricerca con l'ausilio di informazioni euristiche, il metodo best-first diventa comodo per la soluzione del problema.

In questo contesto si inserisce l'algoritmo A* (*A-star*) [4], che combina la ricerca best-first con un meccanismo euristico per stimare il costo verso il nodo obiettivo. A* utilizza una funzione, per valutare il costo di una mossa, $f(n) = g(n) + h(n)$, dove $g(n)$ rappresenta il costo effettivo dal nodo iniziale al nodo $n$, e $h(n)$ è una funzione euristica che stima il costo minimo residuo da $n$ al nodo obiettivo. L'algoritmo A* trova un percorso ottimo.

Nel nostro caso andiamo a implementare l'algoritmo considerando:

- $g(n)$: numero di mosse fatte finora;
- $h(n)$: distanza di Manhattan di una pedina da dove dovrebbe trovarsi;

### 3.2.3 Ottimizzazione iniziale (conflitti lineari)

Per ottimizzare l'algoritmo iniziale possiamo considerare il concetto di conflitto lineare.

Un *conflitto lineare* si verifica quando due tessere si trovano nella stessa riga (o colonna) della griglia, entrambe dovrebbero trovarsi in quella riga (o colonna) nella configurazione obiettivo, ma sono

nell'ordine sbagliato rispetto alla destinazione finale. In questi casi, almeno una delle due tessere dovrà necessariamente essere spostata fuori dalla riga (o colonna) per permettere all'altra di raggiungere la propria posizione, causando un aumento delle mosse minime necessarie.

Possiamo utilizzare l'identificazione dei conflitti lineari per migliorare la funzione euristica di A*, aumentando la precisione rispetto alla semplice distanza di Manhattan (3.2.1).

### 3.2.4 Algoritmo IDA* [12]

L'algoritmo IDA* (*Iterative Deepening A**) è una variante dell'algoritmo A* che ne conserva la funzione euristica ma utilizza meno memoria ([2]). A differenza di A*, che opera in modalità best-first, IDA* adotta un approccio basato su depth-first search (DFS), evitando così la memorizzazione dell'intero albero di ricerca. L'algoritmo esplora lo spazio degli stati iterativamente, imponendo a ogni iterazione un vincolo di soglia sul valore della funzione di valutazione $f(n) = g(n) + h(n)$, dove $g(n)$ rappresenta il costo del cammino dal nodo iniziale a $n$ e $h(n)$ una stima euristica del costo restante. Se durante una visita in profondità viene incontrato un nodo il cui valore di $f$ supera la soglia corrente, esso viene scartato ma contribuisce all'aggiornamento della soglia per l'iterazione successiva. Questo meccanismo consente a IDA* di evitare percorsi ridondanti.

### 3.2.5 Differenza tra A* e IDA*: esempio

Consideriamo un semplice grafo in cui il nodo iniziale è $S$ e l'obiettivo è $G$.

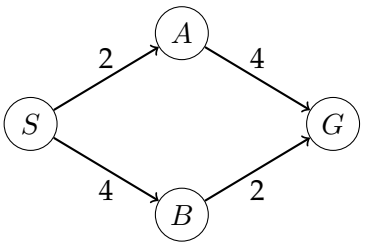



Figura 3.6: Grafo d'esempio con nodi $S$, $A$, $B$ e $G$ e costi sugli archi.

Definiamo inoltre come euristica il costo per passare da $S$ ai nodi vicini. Otteniamo come valori: $h(A) = 2$, $h(B) = 1$, $h(G) = 0$, l'ultimo in quanto $S$ non ha un arco diretto verso $G$.

**A*** calcola $f(n) = g(n) + h(n)$ e mantiene una frontiera ordinata per $f$. In questo esempio, dopo aver espanso $S$, otteniamo $f(A) = 2 + 2 = 4$ e $f(B) = 4 + 1 = 5$. Poiché $f(A) < f(B)$, A* espande prima $A$ e trova il percorso $S \to A \to G$ di costo totale 6, senza esplorare $B$.

**IDA***, invece, esegue una ricerca in profondità limitata da una soglia iniziale di $f$, che aumenta progressivamente se non viene trovata la soluzione. Nel nostro esempio, IDA* può dover esplorare più volte i cammini passando da $A$ e da $B$ con soglie crescenti, fino a raggiungere la soluzione ottimale con costo 6. Il vantaggio è l'uso di memoria proporzionale soltanto alla profondità della ricerca; lo svantaggio è un maggior numero di espansioni rispetto ad A*.

## 3.3 Un possibile approcio a spanning-tree

Considerando quando detto nell'introduzione, precisamente nel capitolo 2, possiamo modellare il gioco del 15 a una soluzione che sfrutta il concetto di spanning-tree.

Cerchiamo quindi di trovare un modo per modellare l'approccio a "sottogruppi" al nostro contesto di lavoro.

### 3.3.1 Una prima idea

| | | | |
|---|---|---|---|
| 1 | 3 | 7 | 4 |
| 6 | 2 | 16 | 8 |
| 5 | 9 | 11 | 12 |
| 13 | 10 | 14 | 15 |

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

| | | | |
|---|---|---|---|
| 1 | 16 | 3 | 4 |
| 6 | 2 | 7 | 8 |
| 5 | 9 | 11 | 12 |
| 13 | 10 | 14 | 15 |

Figura 3.7: Tre stati del gioco del 15: iniziale $\sigma_0$, finale ordinata, e dopo aver fissato la tessera 3.

L'idea è quella di trattare ogni pedina come una foglia dell'albero, portarla a destinazione effettuando solo scambi con la cella vuota e non muovere più tale pedina. Ad esempio, supponiamo di avere la disposizione iniziale $\sigma_0 = [1, 3, 7, 4, 6, 2, 16, 8, 5, 9, 11, 12, 13, 10, 14, 15]$, possiamo provare a portare la pedina etichettata dal numero 3 nella rispettiva posizione. Per farlo scambiamo la cella vuota con la cella contenente il 7 e poi effettuiamo un ulteriore scambio tra la cella vuota e il 3. (figura 3.7) Ora la pedina è nella sua posizione finale e non potrà essere più spostata.

Tuttavia, questo metodo si rivela inapplicabile a causa della forte interdipendenza tra le tessere. Ogni movimento della cella vuota coinvolge necessariamente lo spostamento di una tessera adiacente, il che implica che per collocare correttamente una pedina, è spesso necessario spostare temporaneamente altre tessere già posizionate.

Inoltre, vincolare una tessera a restare ferma una volta posizionata impedisce l'esplorazione di configurazioni intermedie che sarebbero indispensabili per il completamento del puzzle. Di conseguenza, il problema non è risolvibile come una semplice sequenza indipendente di sottoproblemi locali: la soluzione richiede una pianificazione globale che tenga conto di vincoli combinatori e configurazioni transitorie.

## 3.4 Impossibilità dell'approccio tramite spanning tree: dimostrazione

### 3.4.1 Modello algebrico del gioco

Ogni configurazione del gioco del 15 può essere rappresentata da una permutazione dell'insieme $\{1, 2, \ldots, 15\}$, che rappresenta le tessere numerate, più un simbolo aggiuntivo per la cella vuota (nel nostro caso il 16). I movimenti ammessi corrispondono a trasposizioni tra la cella vuota e una tessera adiacente. L'insieme delle configurazioni raggiungibili forma un sottogruppo $G$ del gruppo simmetrico

$S_{16}$, il gruppo di tutte le permutazioni su 16 elementi. Più precisamente, il gruppo $G$ è generato dalle trasposizioni locali che coinvolgono il vuoto e una delle tessere vicine.

Poiché ogni mossa ammissibile coinvolge una sola tessera alla volta e il vuoto, le permutazioni ottenibili sono composte da cicli di lunghezza pari e dunque il gruppo risultante è contenuto nel gruppo alternante $A_{16}$ (il sottogruppo di permutazioni pari di $S_{16}$).

### 3.4.2 Effetto della fissazione di tessere

Quando si decide di "fissare" una tessera $t$ in una posizione specifica, si sta imponendo una restrizione alle permutazioni ammissibili: si considera solo il sottogruppo delle permutazioni che lasciano $t$ invariato. Se $G$ è il gruppo originale delle permutazioni raggiungibili, allora si considera il sottogruppo detto stabilizzatore:

$$\text{Stab}_G(t) = \{\pi \in G \mid \pi(t) = t\}$$

Ogni nuova tessera fissata riduce ulteriormente lo spazio delle permutazioni consentite:

$$\text{Stab}_G(t_1, t_2, \ldots, t_k) = \{\pi \in G \mid \pi(t_i) = t_i \text{ per ogni } i = 1, \ldots, k\}$$

Questa progressiva riduzione dello spazio delle permutazioni ha come effetto che pur essendo la configurazione finale ancora teoricamente raggiungibile nel gruppo $G$, essa non lo sia più nel sottogruppo stabilizzatore ottenuto fissando alcune tessere. In altri termini, il percorso verso la soluzione potrebbe richiedere mosse che temporaneamente "sconvolgono" le tessere già posizionate, per poi riposizionarle correttamente in seguito. L'approccio che impedisce tali mosse per principio elimina queste possibilità.

## 3.5 Un possibile algoritmo risolutivo [13]

Un possibile algoritmo risolutivo del gioco del 15 ha bisogno di un compromesso. Risolvere il puzzle andando a considerare un processo che si divide in 2 fasi.

La prima fase consiste nel risolvere la prima riga da sinistra a destra:

- è necessario trovare la pedina che si vuole portare nella posizione corretta della prima riga;
- se non è l'ultima pedina allora è necessario spostarla fino a raggiungere la posizione desiderata tenendo conto di:
  - non è permesso spostare mai le pedine sistemate precedentemente;
  - per spostare la pedina in una determinata posizione è necessario spostare le altre intorno ad essa finché non si è portata la cella vuota accanto ad essa;
- se l'ultima pedina non è in posizione, è necessario portarla nella posizione sotto a quella in cui deve andare. Eseguendo le seguenti mosse { sotto, sotto, destra, sopra, sinistra, sopra, destra, sotto, sotto, sinistra, sopra}.

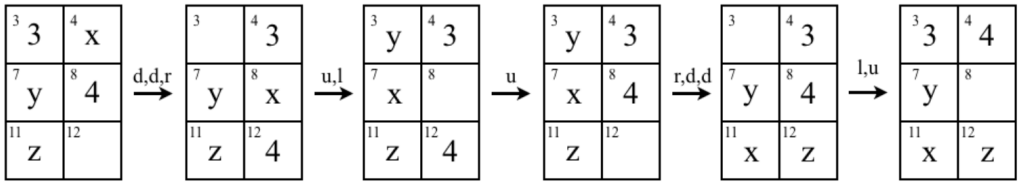



Figura 3.8: Mosse per portare il 4 in posizione

La seconda fase serve a concludere il gioco:

- ripeti la prima fase finché non sono rimaste solo due righe da sistemare;
- ruota il piano di gioco di 45 gradi a destra;
- usa la fase uno per risolvere le nuove prime righe finché non sono rimaste solo due righe, ovvero finché non rimane un piano 2x2;

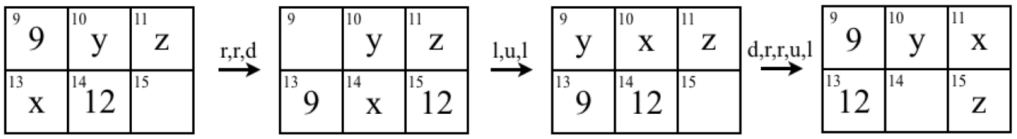



Figura 3.9: seconda fase

- sposta le caselle finché il vuoto e una delle altre 3 caselle non sono in posizione corretta, le altre due dovrebbero essere automaticamente sistemate;

# 4

# Codice per la risoluzione del puzzle

## 4.1 Introduzione

Definiamo ora le strutture generali necessarie allo sviluppo del codice, in Java, utile allo sviluppo di un programma per la risoluzione del puzzle utilizzando i metodi introdotti nel capitolo 3.

## 4.2 Strutture principali

- `board`: matrice intera $N \times M$ che rappresenta lo stato attuale.
- `pos`: mappa che associa ogni valore alla sua posizione obiettivo.
- `pebbles`: mappa che associa ogni valore alla posizione corrente.
- `empty`: valore che identifica la cella vuota.

### 4.2.1 Classe Board

La seguente classe ha lo scopo di fornire una rappresentazione astratta del piano di gioco. La struttura è stata astratta in maniera tale che possa descrivere il gioco del 15 ma anche qualsiasi altra rappresentazione di una tavola $N \times M$.

**`board`** Matrice intera $N \times M$ che rappresenta lo stato attuale del puzzle, dove ogni cella contiene il valore di una tessera oppure il valore speciale della casella vuota.

**`firstDimension`, `secondDimension`** Interi che rappresentano rispettivamente il numero di righe ($N$) e di colonne ($M$) della board.

**`empty`** Valore intero che indica la tessera vuota nel puzzle (tipicamente $0$, $-1$ oppure $16$).

**`pos`** Dizionario che associa ad ogni valore di tessera la sua posizione obiettivo (riga, colonna) nella configurazione risolta del puzzle.

**`pebbles`** Dizionario che associa ad ogni valore di tessera la sua posizione attuale (riga, colonna) nella configurazione corrente.

**`Board(int[][] tiles, int N, int M, int empty)`** Costruttore della classe. Inizializza la board copiando la matrice di input, imposta le dimensioni e il valore della casella vuota, e popola i dizionari `pos` e `pebbles` per la gestione delle posizioni delle tessere.

**`isSolvable()`** Metodo che determina se la configurazione attuale del puzzle è risolvibile, calcolando il numero di inversioni e la posizione della casella vuota secondo le regole matematiche del gioco del 15.

**`printBoard()`** Stampa a video la configurazione attuale della board, utile per il debug o la visualizzazione dello stato.

**`hamming()`** Calcola e restituisce il numero di tessere fuori posto rispetto alla configurazione risolta (distanza di Hamming).

**`manhattan()`** Calcola e restituisce la somma delle distanze di Manhattan di tutte le tessere dalla loro posizione obiettivo.

**`getPhase(int val)`** Restituisce la "fase" a cui appartiene una tessera, utile per strategie di risoluzione suddivise in fasi (ad esempio, risolvere prima le righe superiori, poi le colonne, ecc.).

**`pesoPerFase(int value, int fase)`** Restituisce un peso associato a una tessera in base alla fase attuale della risoluzione, per dare priorità a certe tessere durante la ricerca.

**`manhattanPesato(int fase)`** Calcola la distanza di Manhattan pesata, cioè la somma delle distanze di tutte le tessere moltiplicata per il loro peso relativo alla fase.

**`countLCrow(int row)`, `countLCcol(int col)`** Calcolano il numero di conflitti lineari rispettivamente su una riga o su una colonna. Un conflitto lineare si verifica quando due tessere sono nella stessa riga (o colonna) della loro posizione obiettivo ma sono invertite.

**`linearConflicts()`** Calcola il numero totale di conflitti lineari su tutta la board, utile per migliorare le euristiche di ricerca.

**`neighbors()`** Genera e restituisce tutte le configurazioni adiacenti ottenibili effettuando una singola mossa (spostamento della casella vuota in una delle quattro direzioni possibili).

**`toJSON()`** Serializza la configurazione attuale della board in formato JSON, utile per esportare o salvare lo stato del puzzle.

## 4.3 Possibili tecniche risolutive

### 4.3.1 Algoritmo A*

Descriviamo in questa sezione le strutture utili per implementare l'algoritmo A* introdotto nel capitolo 3.

In questa implementazione l'algoritmo si basa su una disposizione iniziale del piano di gioco $N \times M$ in cui la cella vuota viene identificata dal valore $N \times M$, nel nostro caso 16. L'algoritmo risolutivo è quello di A* in cui vengono usate:

- una coda con proprità: per contenere a ogni iterazione le disposizioni a cui si può arrivare da quella corrente ordinate in base a una priorità in modo da scegliere sempre quella con priorità maggiore;
- una lista: per contenere le disposizioni visitate finora in modo da non visitare di nuovo disposizioni visitate rischiando di rimanere bloccati in cicli infiniti;

A ogni iterazione, quindi, l'algoritmo genera le disposizioni a cui si può arrivare da quella corrente, assegna a loro una priorità, le inserisce in una coda con priorità e sceglie la mossa successiva prendendo quella con maggiore priorità.

La priorità viene generata sommando due valori:

- numero di mosse: numero di mosse eseguite finora;
- distanza di Manhattan e conflitti lineari;

I due valori rappresentano l'euristica $f(x) = g(x) + h(x)$ che guida l'algoritmo di A* nella ricerca della soluzione. Vengono quindi definite le seguenti strutture:

**`Peso`** Classe che contiene il parametro `peso`, usato per implementare una versione pesata dell'algoritmo A* (A* pesato). Il peso modifica l'importanza dell'euristica nella funzione di costo.

**`State`** Classe interna che rappresenta uno stato del puzzle durante la ricerca. Ogni oggetto `State` contiene:

- `config`: oggetto `Board` che rappresenta la configurazione attuale del puzzle.
- `moves`: numero di mosse effettuate per raggiungere questo stato dalla configurazione iniziale.
- `previous`: riferimento allo stato precedente, utile per ricostruire il percorso di soluzione.

Implementa l'interfaccia `Comparable` per essere inserita in una coda di priorità, confrontando gli stati in base alla funzione di priorità (tipicamente $f(n) = g(n) + h(n)$, dove $g(n)$ è il numero di mosse e $h(n)$ è una funzione euristica come la somma di Manhattan e conflitti lineari).

**`astar`** Classe principale che implementa l'algoritmo A* per la risoluzione del gioco del 15 (o di un puzzle $N \times M$ generico). Le sue componenti principali sono:

- `solution`: riferimento allo stato finale risolto, utile per ricostruire la soluzione;
- `isSolvable`: booleano che indica se la configurazione iniziale è risolvibile;
- `nodesExplored`: contatore dei nodi esplorati durante la ricerca;
- `finalConfigs`: mappa per memorizzare configurazioni finali già generate;

**`generateFinalConfig(int N, int M)`** Metodo che genera la configurazione finale (risolta) del puzzle, riempiendo la matrice con valori crescenti da 1 a $N \times M$.

**`astar(Board initial, int N, int M, int empty)`** Costruttore che esegue l'algoritmo A*, la sua implementazione è descritta dai seguenti passi:

- Genera la configurazione finale.
- Inizializza la coda di priorità (`openSet`) e l'insieme dei visitati (`visited`).
- Inserisce lo stato iniziale.
- Esegue il ciclo di ricerca: estrae lo stato con priorità minima, verifica se è la soluzione, esplora i vicini e aggiorna la coda.
- Salva la soluzione se trovata.

**`isSolvable()`** Restituisce `true` se la configurazione iniziale è risolvibile, altrimenti `false`.

**`nodesExplored()`** Restituisce il numero di nodi (stati) esplorati durante la ricerca.

**`moves()`** Restituisce il numero minimo di mosse necessarie per risolvere il puzzle, oppure $-1$ se non esiste soluzione.

**`solution()`** Restituisce un oggetto iterabile contenente la sequenza delle configurazioni (`Board`) dalla iniziale alla finale, ricostruita a ritroso tramite i riferimenti `previous` degli stati.

**`solutionToJson()`** Restituisce una stringa JSON che rappresenta la sequenza delle configurazioni della soluzione, utile per esportare o visualizzare la soluzione in formato strutturato.

**`writeSolutionToFile(String filename, String jsonString)`** Scrive la soluzione in formato JSON su un file, gestendo eventuali errori di I/O.

**`main(String[] args)`** Metodo di test che permette di eseguire l'algoritmo su una configurazione di esempio (o inserita da tastiera), stampa la soluzione, il numero di mosse e salva la soluzione in un file JSON.

Di seguito sono riportati alcuni degli algoritmi principali utili per l'implementazione del codice:

**Algoritmo 5** Calcolo della distanza di Manhattan nella classe `Board`

```
1:  function MANHATTAN
2:      dist ← 0
3:      for i ← 0 to firstDimension − 1 do
4:          for j ← 0 to secondDimension − 1 do
5:              value ← board[i][j]
6:              if value ≠ empty then
7:                  (goalRow, goalCol) ← pos[value]
8:                  dist ← dist + |i − goalRow| + |j − goalCol|
9:              end if
10:         end for
11:     end for
12:     return dist
13: end function
```

**Algoritmo 6** Calcolo della distanza di Manhattan pesata nella classe `Board`

```
1: function manhattanPesato(fase)
2:     totalCost ← 0
3:     for i ← 0 to firstDimension − 1 do
4:         for j ← 0 to secondDimension − 1 do
5:             value ← board[i][j]
6:             if value ≠ empty then
7:                 (goalRow, goalCol) ← pos[value]
8:                 peso ← pesoPerFase(value, fase)
9:                 totalCost ← totalCost + peso × (|i − goalRow| + |j − goalCol|)
10:            end if
11:        end for
12:    end for
13:    return totalCost
14: end function
```

### 4.3.2 Algoritmo risolutivo

**Algoritmo 7** Algoritmo A* nella classe `astar`

```
1: function astar(initial, N, M, empty)
2:     finalConfig ← generateFinalConfig(N, M)
3:     openSet ← nuova coda di priorità
4:     visited ← nuovo insieme vuoto
5:     startState ← nuovo State con initial, moves = 0, previous = null
6:     inserisci startState in openSet
7:     while openSet non è vuoto do
8:         current ← estrai stato con priorità minima da openSet
9:         if current.config = finalConfig then
10:            return current                                  ▷ Soluzione trovata
11:        end if
12:        aggiungi current.config a visited
13:        for ogni neighbor in current.config.neighbors() do
14:            if neighbor non in visited then
15:                newState ← nuovo State con neighbor, moves = current.moves + 1, previous = current
16:                inserisci newState in openSet
17:            end if
18:        end for
19:    end while
20:    return null                                             ▷ Nessuna soluzione trovata
21: end function
```

### 4.3.3 Algoritmo IDA*

In questa sezione descriviamo le strutture utili per implementare l'algoritmo IDA*.

A differenza di A*, non mantiene esplicitamente l'insieme dei nodi già visitati a livello globale, ma esplora ricorsivamente i cammini fino a una soglia di costo, detta *threshold*, evitando cicli tramite una lista di nodi visitati valida solo per la ricorsione corrente.

La funzione di costo è la stessa di A*:

$$f(x) = g(x) + h(x)$$

dove $g(x)$ rappresenta la distanza di Manhattan e $h(x)$ è una funzione euristica (ad esempio il numero di mosse o i conflitti lineari).

Il meccanismo iterativo è il seguente:

- si imposta come soglia iniziale il valore $f(x)$ della configurazione di partenza;
- si esegue una ricerca in profondità, visitando solo configurazioni con $f(x) \leq$ soglia;
- se la soluzione non viene trovata, la soglia viene aggiornata al valore minimo di $f(x)$ che ha superato la soglia corrente;
- si ripete il processo finché non viene trovata la soluzione o finché non esistono più nodi da esplorare.

Vengono quindi definite le seguenti strutture:

**`State`** Classe che rappresenta uno stato del puzzle. Contiene:

- `config`: oggetto `Board` che descrive la configurazione attuale del puzzle;
- `moves`: numero di mosse effettuate per raggiungere questo stato;
- `previous`: riferimento allo stato precedente, utile per ricostruire il percorso.

Definisce inoltre la funzione di priorità $f(n) = g(n) + h(n)$.

**`Result`** Classe interna che rappresenta l'esito di una ricerca ricorsiva. Contiene:

- `found`: indica se è stata trovata la soluzione;
- `nextThreshold`: la nuova soglia calcolata, da usare per l'iterazione successiva;
- `state`: lo stato finale se la soluzione è stata trovata.

**`idastar`** Classe principale che implementa l'algoritmo IDA*. Le sue componenti principali sono:

- `solution`: riferimento allo stato finale risolto;
- `threshold`: soglia corrente della ricerca;
- `finalConfig`: configurazione finale target del puzzle.

**`idastar(Board initial)`** Costruttore che avvia la ricerca IDA* a partire da una configurazione iniziale.

I passi principali sono:

- inizializza la soglia con il valore $f(n)$ della configurazione iniziale;

- esegue iterativamente la funzione `search` finché non viene trovata la soluzione o non ci sono più stati validi;
- aggiorna la soglia a ogni iterazione.

**`search(State current, double threshold, Set<Board> visited)`** Funzione ricorsiva che esplora la configurazione `current`:

- calcola $f(n)$ e confronta con la soglia;
- se $f(n)$ eccede la soglia, interrompe la ricerca e aggiorna la soglia;
- se viene raggiunta la configurazione finale, ritorna la soluzione;
- esplora i vicini della configurazione corrente, evitando cicli tramite `visited`;
- utilizza backtracking rimuovendo il nodo corrente da `visited` al termine della ricorsione.

**`moves()`** Restituisce il numero di mosse della soluzione, oppure $-1$ se non esiste soluzione.

**`solution()`** Restituisce la sequenza di configurazioni dalla iniziale alla finale ricostruita tramite i riferimenti `previous`.

**`main(String[] args)`** Metodo di test: inizializza una configurazione, verifica la risolvibilità e lancia l'algoritmo IDA*, stampando a video i passi della soluzione e il numero di mosse.

# 5

# Teoria della complessità

## 5.1 Alfabeti e linguaggi [6] [5]

**Definizione 8.** *Un simbolo è un'entità primitiva astratta; lettere e caratteri sono esempi di simboli.*

Per poter lavorare con i simboli in modo sistematico, raccogliamo un insieme finito di essi:

**Definizione 9.** *Un alfabeto $\Sigma$ è un insieme finito di simboli.*

A partire dall'alfabeto, possiamo costruire sequenze ordinate di simboli. Queste costituiscono il passo successivo:

**Definizione 10.** *Una stringa è una sequenza finita di simboli, generalmente indicata con $w$; la sua lunghezza è indicata invece con $|w|$. La stringa vuota si indica con $\epsilon$ ed è tale che $|\epsilon| = 0$.*

Infine, avendo definito cosa siano stringhe e alfabeti, possiamo introdurre il concetto centrale della teoria dei linguaggi formali:

**Definizione 11.** *Un linguaggio formale è un insieme di stringhe di simboli da un alfabeto $\Sigma$. L'insieme $\emptyset$ e l'insieme $\{\epsilon\}$ sono due linguaggi formali di qualunque alfabeto. Con $\Sigma^*$ si intende il linguaggio costituito da tutte le stringhe su un fissato alfabeto $\Sigma$. Ad esempio, se $\Sigma = \{0\}$ allora $\Sigma^* = \{\epsilon, 0, 00, 000, \dots\}$.*

## 5.2 Definizione di macchina di Turing deterministica [6] [5]

Una macchina di Turing M può essere definita come una quintupla di elementi:

$$M = (Q, \Sigma, q_0, \delta, F)$$

dove:

- $Q$: insieme finito di stati;
- $\Sigma$: insieme finito di simboli;
- $\delta$: funzione di transizione, dati uno stato $q$ e un simbolo di nastro $X$, $\delta(q, X)$ restituisce una tripla $(p, Y, D)$:

  - $p$: è lo stato successivo;
  - $Y$: è il simbolo di $\Sigma$ scritto nella cella visitata;
  - $D$: è una direzione, può essere $L$ o $R$;
- $q_0$: lo stato iniziale;
- $F$: è l'insieme degli stati finali;

$\delta$ può essere definito come:

$$\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$$

Definiamo a questo punto una descrizione istantanea (ID) come una quadrupla:

$$\langle q, v, s, w \rangle$$

dove:

- $q \in Q$: è lo stato in cui si trova la macchina;
- $v, w \in \Sigma^*$: sono i caratteri diversi da blank a sinistra e destra della testina;
- $s \in \Sigma$: è il simbolo letto dalla testina;

ad esempio:

$$X_1 X_2 ... X_{i-1} q X_i X_{i+1} ... X_n$$

in cui:

- $q$ è lo stato della macchina di Turing;
- la testina visita l'i-esimo simbolo da sinistra;
- $X_1 X_2 ... X_n$ è la porzione del nastro tra il simbolo diverso dal blank più a sinistra e quello più a destra;

Otteniamo quindi che le mosse di una macchina di Turing M sono definite mediante il simbolo $\vdash_M$.

Supponiamo che $\delta(q, X_i) = (p, Y, L)$ cioè la prossima mossa è verso sinistra, otteniamo:

$$X_1 X_2 ... X_{i-1} q X_i X_{i+1} ... X_n \vdash_M X_1 X_2 ... X_{i-2} p X_{i-1} Y X_{i+1} ... X_n$$

bisogna però tenere conto di due accorgimenti:

- se $i = 1$ allora la mossa a sinistra fa muovere M verso il blank, ottenendo:

$$q X_1 X_2 ... X_n \vdash_M pBY X_2 ... X_n$$

- se $i = n$ e $Y =$ \$ allora il simbolo \$ scritto su $X_n$ si unisce alla sequenza di blank e non compare nella ID:

$$X_1 X_2 ... X_{n-1} q X_n \vdash_M X_1 X_2 ... X_{n-2} p X_{n-1}$$

Ci comportiamo allo stesso modo se la testina si muove a destra. Anche in questo caso si presentano le due eccezioni precedentemente descritte cambia solo il verso nel quale si vanno a scrivere i simboli.

Per funzionare, la stringa di input viene posta sul nastro e la testina parte dal simbolo di input più a sinistra. Se la macchina di Turing entra in uno stato accettante allora l'input è accettato, altrimenti no. Si ha quindi che se $M = (Q, \Sigma, q_0, \delta)$ è una macchina di Turing, allora $L(M)$ è l'insieme di stringhe $w \in \Sigma^*$ tale che $q_0 w \vdash^* \alpha p \beta$ per uno stato $p$ finale e qualunque stringa di nastro $\alpha$ e $\beta$.

## 5.3 Linguaggi accettati da una macchina di Turing [6] [5]

Un linguaggio $L \subseteq \Sigma^*$ si dice accettato da una macchina di Turing $M$ se ogni stringa $w \in L$, quando fornita come input a $M$, porta la macchina a fermarsi in uno stato di accettazione. Mentre tutte le stringhe appartenenti a $L$ vengono necessariamente accettate con arresto, per le stringhe $w \notin L$ la macchina può comportarsi in due modi: fermarsi in uno stato di rifiuto, oppure non arrestarsi mai.

## 5.4 Arresto di una macchina di Turing [6] [5]

Una macchina di Turing MdT si arresta se entra in uno stato $q$ visitando un simbolo $X$ e non ci sono mosse possibili in questa situazioni, in tal caso otteniamo che $\delta(q, X)$ è indefinito, oppure se computa l'input. Possiamo sempre presumere che MdT si arresti se accetta, tuttavia, non possiamo sempre richiedere che MdT si arresti. I linguaggi per i quali esiste una macchina di Turing che prima o poi si arresta (indipendentemente dal fatto che accetti o no), sono detti ricorsivi.

Invece i linguaggi che possiamo accettare usando una macchina di Turing, ma che se non accettati non ci garantiscono che la macchina si fermi, sono denominati linguaggi ricorsivamente enumerabili.

## 5.5 Funzioni calcolabili da MdT [6] [5]

Ad ogni MdT può essere associata una funzione.

**Definizione 12.** *Assumiamo di trattare solo funzioni sui naturali e di codificare i numeri naturali in una codifica unaria in cui ogni simbolo $n \in \mathbb{N}$ è rappresentato da $n + 1$ zeri consecutivi. Una funzione $f : \mathbb{N}^n \to \mathbb{N}$ è Turing-calcolabile se esiste una MdT M tale che partendo dalla configurazione iniziale:*

$$...q_0\$x_1\$...\$x_n\$...$$

*se $f(x_1, ..., x_n)$ è definita allora M termina nella configurazione:*

$$...\$\$\$...q_f\$f(x_1, ..., x_n)\$...$$

*altrimenti M non termina, dove per ogni $(x_1, ..., x_n) \in \mathbb{N}^n$, $x_1, ..., x_n$, $f(x_1, ..., x_n)$ sono le rappresentazioni in unario di $x_1, ..., x_n$ e $f(x_1, ..., x_n)$ mentre $q_f \in Q$ è tale che per cui $\delta(q_f, \$)$ è indefinito.*

**Definizione 13.** *Una funzione totale è una funzione definita per ogni input.*

## 5.6 Macchine di Turing multinastro [6] [5]

Una MdT multinastro è formata da un controllo finito e un numero $k$ finito di nastri. Ogni nastro è diviso in celle e ogni cella può contenere un simbolo dell'alfabeto. L'insieme di simboli di nastro comprende un blank e ha un sottoinsieme definito di simboli di input. L'insieme degli stati ne comprende uno iniziale e un sottoinsieme di stati finali (accettanti). Una MdT multinastro inizia rispettando:

- l'input si trova sul primo nastro;
- tutte le altre celle contengono un blank;
- il controllo si trova nello stato iniziale;
- la testina del primo nastro è all'estremità sinistra dell'input;
- le altre testine si trovano in celle arbitrarie siccome sono tutte blank;

Una mossa su una MdT multinastro dipende dallo stato e dai simboli letti da ciascuna testina. In una mossa la macchina compie:

- il controllo entra in un nuovo stato;
- su ogni cella visitata da una testina viene scritto un nuovo simbolo di nastro;
- ogni testina si muove a sinistra o a destra oppure sta ferma, i movimenti sono indipendenti.

In questo caso la funzione di transazione cambia, ed è definita come:

$$\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{L, R\}^k$$

cioè è un estensione della definizione di $\delta$ per le MdT singolo nastro dove partendo da uno stato $q \in Q$ si guardano $k$ simboli (uno su ogni nastro) e restituisce una tripla formata da uno stato, $k$ simboli scritti nelle $k$ celle visitate e $k$ direzioni in cui si muovono i $k$ nastri.

**Teorema 5.** *Ogni linguaggio accettato da una MdT multinastro è ricorsivamente enumerabile.*

*Dimostrazione.* Sia $L$ un linguaggio accettato da una MdT con $k$ nastri, M. Simuliamo M con una MdT mononastro N il cui nastro è diviso in $2k$ tracce. Metà delle tracce ($k$) replicano i nastri di M, ognuna delle restanti indica la posizione corrente della testina del corrispondente nastro di M. □

## 5.7 Definizione di macchina di Turing nondeterministica [6] [5]

Una macchina di Turing nondeterministica si distingue da quella deterministica in quanto la funzione $\delta$ associa a ogni stato $q$ un insieme di triple, definito come:

$$\delta(q, X) = \{(q_1, Y_1, D_1), (q_2, Y_2, D_2), ..., (q_k, Y_k, D_k)\}$$

con $k$ intero finito.

A ogni passo la N-MdT sceglie una delle triple come mossa. Si ottiene che M accetta una stringa $w$ se c'è una sequenza di scelte che conduce dalla ID iniziale con $w$a una ID con stato accettante.

**Teorema 6.** *Se $M_N$ è una macchina di Turing nondeterministica, esiste una macchina di Turing deterministica $M_D$ tale che $L(M_N) = L(M_D)$.*

## 5.8 $k-$MdT [5] [6] [5]

Una $k-$Mdt è una macchina di Turing con $k-$nastri. Come ogni MdT è definita come $M = (Q, \Sigma, q_0, P)$ dove:

- $Q$: è l'insieme finito di stati e $q_0 \in Q$ è lo stato iniziale;
- $\Sigma$: è l'alfabeto in cui assumiamo siano presenti almeno i simboli \$ (blank) e $\triangleright$ (first);
- L'insieme delle istruzioni $P$ che rappresenta una funzione di transazione $\delta$ che assumiamo essere una funzione totale:

$$\delta : Q \times \Sigma^k \to (Q \cup \{h, yes, no\}) \times (\Sigma \times \{L, R, F\})^k$$

- $h, yes, no$ stati finali $\notin Q$;
- la configurazione iniziale è del tipo:

$$(q_0, \underbrace{\epsilon, \triangleright, x}_{nastro\ 1}, \underbrace{\epsilon, \triangleright, \epsilon,}_{nastro\ 2} ..., \underbrace{\epsilon, \triangleright, \epsilon}_{nastro\ 3})$$

**Definizione 14.** *Una $k-$MdT è di tipo decisionale se ogni qual volta termina, raggiunge uno degli stati finali $yes$ o $no$, in questo caso l'output è lo stato raggiunto. Una $k-$MdT invece calcola una funzione se ogni qual volta termina, raggiunge lo stato finale $h$, in questo caso l'output è contenuto nel nastro $h$.*

## 5.9 Classi di complessità, determinismo e nondeterminismo [5]

Per parlare di complessità di un determinato algoritmo, dobbiamo definire le classi di complessità, ovvero insiemi di linguaggi accumunati da una qualche proprietà. Informalmente, quando si calcola la complessità di un determinato algoritmo, quello che si fa è andare a calcolare il tempo con il quale una macchina di Turing accetta un determinato linguaggio.

**Definizione 15.** *Sia M una $k-$MdT deterministica a $k$ nastri.*
*Sia $x$ un input della macchina.*
*Il tempo richiesto da M per $x$ è il numero di passi di computazione necessari a $M$ con input $x$ per terminare.*

Il tempo però cambia nel caso in cui la macchina è una MdT deterministica oppure non.

**Definizione 16.** *Sia $f : \mathbb{N} \to \mathbb{N}$ una funzione totale.*
*Sia M una $k-$MdT deterministica a $k$ nastri.*
*Sia $x$ un input della macchina.*
*Una $k-$MdT opera in tempo $f(n)$ se per ogni input $x$ il tempo richiesto da $M$ per $x$ è minore o uguale a $f(|x|)$, supponendo che $|x| = n$ questo vuol dire minore o uguale a $f(n)$.*

Dopo aver definito il tempo di calcolo di una macchina, possiamo introdurre la nozione di linguaggio deciso da una MdT deterministica. Un linguaggio $L \subseteq \Sigma^*$ è deciso da una k-MdT M:

- se $x \in L$ allora $M(x) = yes$;
- se $x \notin L$ allora $M(x) = no$;

**Definizione 17.** *Se $L \subseteq \Sigma^*$ è deciso da una k-MdT M e M opera in tempo $f(n)$, allora $L \in TIME(f(n))$, quindi $TIME(f(n))$ è una classe di complessità in tempo.*

Il concetto può essere esteso anche al modello non deterministico. In questo caso la definizione di decisione richiede di considerare i rami di computazione.

**Definizione 18.** *Un linguaggio $L \subseteq \Sigma^*$ è deciso da una MdT non deterministica M se per ogni $x \in \Sigma^*$:*

- *se $x \in L$ allora esiste una computazione nondeterministica tale che:*

$$(q_0, \epsilon, \triangleright, x) \rightarrow_* (yes, u, s, v)$$

- *se $x \notin L$ allora non esiste una computazione non deterministica tale che:*

$$(q_0, \epsilon, \triangleright, x) \rightarrow_* (yes, u, s, v)$$

Analogamente a quanto fatto per le MdT deterministiche, possiamo definire la misura del tempo di esecuzione.

**Definizione 19.** *Una ND-MdT M opera in tempo $f(n)$ se per ogni $x \in \Sigma^*$, ogni computazione non deterministica sull'input $x$ ha al più lunghezza $f(|x|)$.*

Quindi, nel caso di ND-MdT, non è l'intera macchina a operare in tempo $f(n)$, ma ciascun ramo della computazione non deterministica.

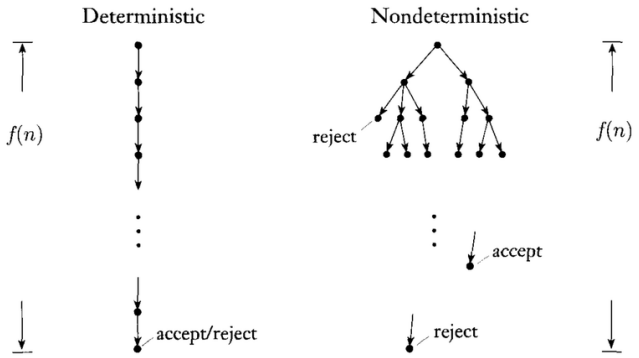



Figura 5.1: Differenza tra MdT deterministica e non

A questo punto possiamo introdurre la classe di complessità corrispondente.

**Definizione 20.** *Se un linguaggio $L \in \Sigma^*$ è deciso da una N-MdT M che opera in tempo $f(n)$ allora $L \in NTIME(f(n))$.*

Le due classi più importanti che derivano da queste nozioni sono $P$ e $NP$, che catturano la complessità polinomiale nei modelli deterministico e non deterministico.

**Definizione 21.** *La classe di complessità P è l'insieme di tutti i linguaggi che possono essere decisi in tempo polinomiale, ovvero:*

$$P = \bigcup_k TIME(n^k)$$

**Definizione 22.** *La classe di complessità NP è l'insieme di tutti i linguaggi che possono essere decisi in tempo polinomiale da una macchina non deterministica, ovvero:*

$$NP = \bigcup_k NTIME(n^k)$$

## 5.10 Riduzioni polinomiali

La tecnica per dimostrare che un problema $P_2$ non può essere risolto in tempo polinomiale è la riduzione di un problema $P_1$, che si sa non essere $\mathcal{P}$, a $P_2$. Bisogna, però, richiedere un ulteriore vincolo: la traduzione da $P_1$ a $P_2$ deve richiedere un tempo polinomiale nella lunghezza dell'input.

## 5.11 Problemi NP-completi e NP-hard [6] [5]

**Definizione 23.** *Diciamo che $L$ è $NP-$completo se:*

- *$L$ è in $NP$;*
- *per ogni linguaggio $L'$ in $NP$ esiste una riduzione polinomiale di $L'$ a $L$;*

Su alcuni problemi $L$, sebbene sia possibile dimostrare la seconda condizione non è possibile dimostrare la prima, ovvero che $L \in NP$, tali problemi prendono il nome di NP-hard.

## 5.12 Il problema SAT [6] [5]

Il problema di soddisfacibilità booleana, detto più comunemente SAT, è un esempio di problema NP-completo.

Il problema si basa sull'andare a decidere se un'espressione booleana è soddisfacibile.

Le espressioni booleane sono formate da:

- variabili a valori booleani ovvero 1 per indicare true e 0 per indicare false;
- gli operatori binari $\wedge$ e $\vee$ per indicare rispettivamente AND e OR;
- l'operatore unario $\neg$ che indica la negazione logica;
- parentesi per raggruppare il tutto;

**Esempio 1.** *Un esempio di espressione booleana è $x \wedge \neg(y \vee z)$.*

**Definizione 24.** *Un assegnamento di valori di verità per un espressione booleana E assegna i valori vero o falso a ognuna delle variabili presenti in E, di conseguenza il valore dell'espressione E a seguito dell'assegnamento T è indicato con $E(T)$. Se $E(T) = 1$ allora l'espressione è soddisfatta, senò no.*

Il problema della soddisfacibilità è definito come:

- un'espressione booleana assegnata è soddisfacibile?

Enunciato come linguaggio, SAT è l'insieme delle espressioni booleane soddisfacibili.

Bisogna però trovare un modo per codificare il SAT. In un espressione booleana possono esserci un numero infinito di simboli, bisogna ideare un codice con un alfabeto finito per rappresentare espressioni che possono essere infinitamente grandi. Per farlo:

- rappresentiamo gli operatori binari e unari da se stessi ($\neg, \wedge, \vee$);
- rappresentiamo la variabile $x_i$ come $x$ seguito dalla codifica binaria di $i$;

**Esempio 2.** *Ad esempio $\neg x_1 \wedge (x_2 \vee x_3)$ equivale a $\neg x1 \wedge (x10 \vee x11)$*

### 5.12.1 NP-completezza del problema SAT

Per dimostrare la NP-completezza di SAT andiamo a usare la riduzione per ridurre un qualsiasi linguaggio NP (accettato da un N-MdT in tempo polinomiale) a SAT. Per provare che SAT è NP-completo bisogna provare che:

1. SAT sta in NP;
2. qualsiasi altro linguaggio in NP è riducibile a SAT;

*Dimostrazione.* Dimostriamo che SAT è NP-completo:

1. Per provare che SAT è NP sfruttiamo una N-MdT. Supponiamo di avere un espressione booleana E e una macchina di Turing non deterministica M. Costruiamo M in modo tale che grazie al non determinismo provi tutti gli assegnamenti possibili di valori di verità in E. Ogni diramazione rappresenta quindi il tentativo di un diverso assegnamento di valori. Si possono raggiungere $2^n$ ID, ovvero configurazioni diverse e a questo punto basta valutare E per l'assegnamento di valori fatto (in ogni diramazione), se $E(T) = 1$ allora accettiamo. Una N-MdT multinastro effettua la valutazione in tempo $O(n^2)$ mentre una singolo nastro in tempo $O(n^4)$, che è comunque polinomiale.
2. Sia $L$ un linguaggio qualsiasi appartenente a NP. Per definizione, esiste una macchina di Turing non deterministica $M$ che decide $L$ in tempo polinomiale $p(n)$. L'idea della riduzione è costruire, a partire da un'istanza $x$ di $L$, una formula booleana $\varphi_x$ tale che:

$$\varphi_x \text{ è soddisfacibile} \iff x \in L.$$

La costruzione avviene codificando il calcolo di $M$ su input $x$ in una tabella *tempo* × *spazio*, dove ogni cella rappresenta il contenuto di un nastro, lo stato della macchina e la posizione della testina

in un determinato passo. Le condizioni di correttezza del calcolo (transizioni valide, stato iniziale corretto, stato finale accettante) vengono espresse tramite vincoli booleani. La formula risultante $\varphi_x$ ha dimensione polinomiale rispetto a $|x|$ e può essere costruita in tempo polinomiale. Pertanto, ogni problema in NP si riduce in tempo polinomiale a SAT. [3]

□

### 5.12.2 Variante di SAT: 3SAT

Una versione ristretta del problema SAT molto utilizzata in Informatica è la variante 3SAT. Anche questo è un problema di soddisfacibilità booleana ma dove le espressioni booleane sono congiunzioni di disgiunzioni di esattamente tre variabili. Formalmente il 3SAT prende in input istanze di SAT in cui ogni clausola consta di esattamente 3 elementi e si pone come problema quello di stabilire, come per SAT, se esiste un assegnamento che rende vera la formula.

**Esempio 3.** *Un esempio di espressione $E$ che rispetta le ipotesi di 3SAT è $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$.*

**Teorema 7.** *3SAT, come SAT, è NP-completo.*

*Dimostrazione.* L'appartenenza a NP si mostra in quanto ogni istanza di 3SAT è anche istanza di SAT. Abbiamo già dimostrato che ogni problema in NP si può ridurre a SAT, mostriamo che ogni problema NP si può ridurre a 3SAT mostrando che SAT si può ridurre a 3SAT (SAT $\preceq$ 3SAT). Per farlo ci basta tradurre ogni clausola $l_1 \vee ... \vee l_m$ in una clausola uguale di 3SAT:

- se m=1: la trasformo in $l_1 \vee l_1 \vee l_1$;
- se m=2: la trasformo in $l_1 \vee l_2 \vee l_2$;
- se m=3: la tengo così com'è;
- se m>3: introduco una nuova variabile X e restituisco $l_1 \vee l_2 \vee X$ e $\neg X \vee l_4 \vee ... \vee l_m$ e riapplico la regola;

□

## 5.13 3-Dimensional Matching (3DM) [7]

Dati 3 insiemi disgiunti X,Y,Z di uguale dimensione $n$. Dato un insieme di triple $T \subseteq X \times Y \times Z$. Esiste un sottoinsieme $S \subseteq T$ tale che ogni elemento $\in X \cup Y \cup Z$ è in esattamente una tripla $s \in S$.

**Teorema 8.** *3DM è NP-completo.*

*Dimostrazione.* Per dimostrare la NP-completezza di 3DM dobbiamo dimostrare sia che 3DM appartiene a NP, sia che ogni problema in NP è riducibile a 3DM. La seconda parte la dimostriamo dimostrando che 3SAT è riducibile a 3DM e siccome abbiamo già dimostrato che SAT è riducibile a 3SAT e SAT soddisfa l'ipotesi, allora anche 3DM la soddisfa.

- per dimostrare che 3DM appartiene a NP basta considerare un algoritmo non deterministico che crea un sottoinsieme di triple e in tempo polinomiale verifica che nessuna delle triple sia intersecata;

- riduciamo 3SAT a 3DM. Sia $U = \{u_1, u_2, ..., u_n\}$ l'insieme delle variabili e sia $C = \{c_1, c_2, ..., c_m\}$ l'insieme delle clausole. Cerchiamo di costruire 3 insiemi disgiunti $X, Y, Z$ e un insieme $M \subseteq X \times Y \times Z$ tale che M contenga una tripla se e solo se tutte le clausole di $C$ sono soddisfacibili. L'insieme delle triple M può essere visto come partizionato in 3 parti che possiamo definire variable gadget, clause gadget e garbage collection. La prima componente corrisponde alle singole variabili $u \in U$, per ognuna di esse vengono creati due elementi $y_{u_i} \in Y$ e $z_{u_i} \in Z$ che vanno a creare una tripla insieme a $u_i$. In questo caso si veranno a creare $2nu_i$ triple dove $nu_i$ rappresenta il numero di clausole in cui è presente la variabile $u_i$. Il tutto può essere rappresentato attraverso un grafo sul quale veranno mappate le variabili $u_i$ e le loro negazioni $\neg u_i$, si ottiene una situazione rappresentabile come:

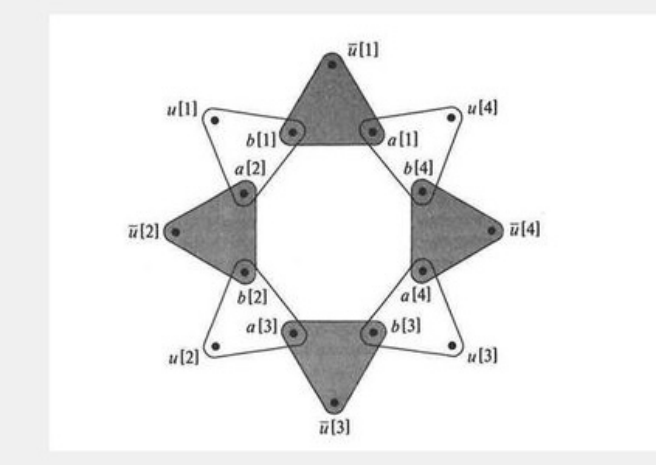



Figura 5.2: Rappresentazione grafica delle clausole del 3DM

Si ottengono in questo modo due insiemi:

$$T_i^t = \{(\neg u_i, y_{u_i}, z_{u_i})\} T_i^f = \{(u_i, y_{u_i}, z_{u_{i+1}})\}$$

$$T = T_i^f \cup T_i^t$$

Si nota quindi che in ogni caso, per non sovrapporre gli elementi, si dovranno scegliere o i triangoli che contengono il valore positivo $u_i$ o la sua negazione, questo obbliga a scegliere un valore di verità per una delle variabili. Allo stesso modo si crea la variable gadget, dove ogni variabile forma una tripla con due nuove variabili $y_c, z_c$ per ogni clausola $c_j \in C$ presente in $E$, questo per indicare quali variabili appaiono in quali clausole, si forma quindi:

$$C_j = \{(u_{ij}, y_{c_j}, z_{c_j}) : u_i \in c_j\} \cup \{(\neg u_{ij}, y_{c_j}, z_{c_j}) : u_i \in c_j\}$$

Per completare la costruzione viene usata la garbage collection che unisce ogni variabile ($u_i$) e la sua negazione ($\neg u_i$) a due componenti $y_{g_k} \in Y$ e $z_{g_k} \in Z$, in questo modo tutti gli elementi

che non appaiono in variable gadget o in clause gadget possono essere recuperate sfruttando la garbage collection:

$$G = \{(u_{ij}, y_{g_k}, z_{g_k}), (\neg u_{ij}, y_{g_k}, z_{g_k})\}$$

Infine, creando un insieme di triple in modo che ogni elemento di $X \times Y \times Z$ appaia esattamente in una sola tripla:

$$M = (\bigcup_{i=1}^{n} T_i) \cup (\bigcup_{j=1}^{m} C_j) \cup G$$

abbiamo ridotto il problema di SAT a 3DM.

□

## 5.14 3-Exact-Cover (X3C)

Sia $X$ un insieme finito con $|X| = 3q$ elementi e una collezione $C$ di sottoinsiemi di 3 elementi di $X$ ci chiediamo se $C$ contiene una exact cover per X, cioè, se esiste una collezione $C' \subseteq C$ tale che ogni elemento di $X$ appare esattamente una volta in $C'$. Si noti che 3DM non è altro che una versione ristretta di X3C. Infatti è possibile ridurre il problema del 3DM a 3XC.

Sia $C = \{\{x, y, z\} : (x, y, z) \in T\}$ e sia $k = |X| = |Y| = |Z| = n$. Otteniamo che $C$ è una collezione di sottoinsiemi di $T$ dove $T$ è l'insieme delle triple di valori e $k$ rappresenta il numero di sottoinsiemi di $C$, ovvero $|C'| = k$, cioè ogni tripla rappresenta un sottoinsieme di $C$, ciò che vogliamo è che ogni elemento di $C'$ sia una tripla disgiunta. Supponiamo $M$ sia la soluzione al 3DM. Sia $C' = \{\{x, y, z\} : (x, y, z) \in M\}$, abbiamo $|M| = n = |C'|$. Sappiamo che se $(x, y, z)$ e $(x', y', z')$ sono due elementi distinti di $M$ allora, siccome per definizione $X \cap Y \cap Z = \emptyset$ avremmo $\{x, y, z\} \cap \{x', y', z\} = \emptyset$. Cioè gli elementi in $C'$ sono a due a due disgiunti, e siccome coprono tutti gli elementi dell'insieme universo, otteniamo che $C'$ è una soluzione al 3XC.

Allo stesso modo, supponiamo $C' = \{C_1, C_2, ..., C_k\}$ sia una soluzione al 3XC. Sia $M = \{(x, y, z) : \{x, y, z\} = C_i \; per \; i = 1, 2, ...k\}$. Siccome gli $C'_i$ sono a due a due disgiunti, se $(x, y, z)$ e $(x', y', z')$ sono due elementi distinti di $M$ allora sono tutti distinti. Quindi $M$ è una soluzione al 3DM.

Avendo quindi dimostrato che 3DM è un problema NP-completo, si dimostra che anche X3C lo è.

# 6

# Dimostrazione della NP-completezza del gioco del 15

In principio il problema di calcolare la minor sequenza di mosse che servono per passare da una configurazione del gioco del 15 a un altra è stato dimostrato, da parte di Oded Goldreich, essere un problema NP-hard. Nella sua dimostrazione Goldreich dimostra tale assunto riducendo il problema di 3-Exact-Cover (3XC), descritto sopra, a una formalizzazione del problema del gioco del 15. [9]

La dimostrazione però può essere fatta anche in un secondo modo ovvero considerando i concetti di gruppo e generatore.

## 6.1 Definizione di gruppo

Un gruppo è una struttura algebrica $(G, \cdot)$ costituita da un insieme $G$ non vuoto munito di un'operazione binaria $\cdot : G \times G \to G$ che soddisfa i seguenti assiomi:

- Associatività: Per ogni $a, b, c \in G$, si ha $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- Elemento neutro: Esiste un elemento $e \in G$ tale che per ogni $a \in G$, $e \cdot a = a \cdot e = a$
- Elemento inverso: Per ogni $a \in G$, esiste un elemento $a^{-1} \in G$ tale che $a \cdot a^{-1} = a^{-1} \cdot a = e$

Nel contesto delle permutazioni, i gruppi emergono dal fatto che ogni permutazione di un insieme finito $X = 1, 2, \ldots, n$ può essere vista come una funzione biiettiva $\sigma : X \to X$. L'insieme di tutte le permutazioni di $X$, denotato $S_n$, forma un gruppo rispetto alla composizione di funzioni, chiamato gruppo simmetrico di grado $n$.

## 6.2 Generatori di un gruppo

Sia $(G, \cdot)$ un gruppo e sia $S \subseteq G$ un sottoinsieme. Il sottogruppo generato da $S$, denotato $\langle S \rangle$, è il più piccolo sottogruppo di $G$ che contiene tutti gli elementi di $S$. Equivalentemente, $\langle S \rangle$ è l'insieme di tutti gli elementi di $G$ che possono essere espressi come prodotto finito di elementi di $S$ e dei loro inversi.

Un sottoinsieme $S \subseteq G$ si dice insieme di generatori di $G$ se $\langle S \rangle = G$, ovvero se ogni elemento di $G$ può essere espresso come combinazione finita di elementi di $S$ e dei loro inversi.

## 6.3 Gruppi, permutazioni e gioco del 15

Nel nostro caso possiamo formalizzare il gioco del 15 con un gruppo. Consideriamo:

- un insieme di elementi: nel nostro caso i numeri da 1 a 16;
- i generatori come qualsiasi trasposizione di una pedina con la cella vuota (16);

## 6.4 Ricerca del numero minimo di mosse per passare da una permutazione a un altra e teoria dei gruppi

Il motivo per cui siamo interessati ai gruppi e alla teoria dei gruppi è perché la ricerca del numero minore di mosse che portano da una configurazione del piano di gioco alla soluzione corrisponde al problema di trovare la più breve sequenza generatrice che realizza una permutazione. [9]

Problema che è stato dimostrato essere NP-hard. [14] Essendo ogni generatore una trasposizione tra una pedina e la cella vuota, questo equivale a cercare il numero minimo di trasposizioni che portano alla soluzione. Ma essendo ogni trasposizione (mossa) rappresentabile con una permutazione questo equivale a cercare il numero minimo di permutazioni del piano di gioco che portano alla soluzione.

## 6.5 Grafi di Cayley e problema del calcolo del diametro

Il nostro problema può essere dimostrato essere NP-hard seguendo anche una seconda strada, ovvero sfruttando il concetto di grafo di Cayley.

Un grafo di Cayley è una struttura matematica che fornisce una rappresentazione geometrica di un gruppo attraverso la teoria dei grafi. Formalmente, dato un gruppo finito $G$ e un insieme di generatori $S \subseteq G$ tale che $S$ non contenga l'elemento neutro e sia chiuso rispetto all'inverso, il grafo di Cayley contiene un vertice che corrisponde a ogni elemento del gruppo $G$ e un arco tra due vertici $g$ e $h$ se esiste un generatore $s \in S$ tale che $h$ può essere ottenuto da $g$ moltiplicando per $s$.

Nel nostro caso ogni vertice del grafo corrisponde a una configurazione del puzzle. Ogni arco collega due configurazioni se è possibile passare dall'una all'altra tramite una mossa elementare. Così, il grafo di Cayley rappresenta tutte le possibili disposizioni e le transizioni tra di esse.

Il diametro di un grafo connesso $G$ è la massima distanza tra due vertici qualsiasi del grafo:

$$\text{diam}(G) = \max_{u,v \in V(G)} d(u,v)$$

dove $d(u,v)$ denota la lunghezza del cammino più breve tra $u$ e $v$.

Nel contesto dei grafi di Cayley, il diametro ha un'interpretazione algebrica particolarmente significativa: rappresenta il numero minimo di elementi dell'insieme generatore $S$ necessari per esprimere qualsiasi elemento del gruppo come prodotto di generatori. Nel puzzle del 15, il diametro del grafo di Cayley corrisponde al numero massimo di mosse minime necessarie per risolvere qualsiasi configurazione.

Calcolare il diametro di un grafo di Cayley è stato più volte dimostrato essere un problema NP-hard. [11]

# 7

# Calcolo delle complessità computazionli dei pseudocodici proposti come algoritmi implementativi per la soluzione del gioco del 15

Analizziamo ora dal punto di vista della complessità computazionale gli algoritmi che sono stati proposti per affrontare la risoluzione del gioco del 15. L'obiettivo è valutare, per ciascun approccio, l'ordine di grandezza del tempo di esecuzione e dello spazio in memoria richiesto.

## 7.1 Calcolo dell'esistenza del routing number

Come già descritto nel capitolo 2, l'algoritmo descritto ha una complessità temporale che dipende dalla struttura del grafo e dal numero di nodi $n$. In particolare:

- Il calcolo che involve una BFS per ogni nodo, ha una complessità di $O(n+m)$, dove $n$ è il numero di nodi e $m$ il numero di archi nel grafo.
- In generale, il numero massimo di passi necessari è di $O(n+m)$;

Pertanto, la complessità complessiva dell'algoritmo è $O(n+m)$.

## 7.2 Algoritmo di verifica di risolvibilità

Anche questo algoritmo è già stato analizzato all'interno del capitolo 3, ribadiamo che a causa del doppio ciclo for utilizzato per contare le inversioni l'algoritmo ha una complessità pari a $O(n^2)$.

## 7.3 Algoritmo risolutivo: A*

La complessità temporale di A* è espressa in funzione di due parametri fondamentali:

- $b$: fattore di branching (numero medio di successori per nodo);
- $d$: profondità della soluzione ottima;

Nel caso peggiore, A* presenta una complessità temporale di $\mathcal{O}(b^d)$, equivalente a quella di una ricerca in ampiezza. Tuttavia, questa rappresenta uno scenario che raramente si verifica.

In generale A* ha una complessità:

- **Caso migliore**: quando l'euristica è perfetta A* esplora solo i nodi lungo il percorso ottimo, risultando in una complessità temporale $\mathcal{O}(d)$.
- **Caso peggiore**: A* degenera in una ricerca con complessità $\mathcal{O}(b^d)$ equivalente a una in ampiezza;
- **Caso medio**: con euristiche ammissibili la complessità si riduce rispetto al caso peggiore nell'ordine di $\mathcal{O}(b^{d\cdot\varepsilon})$ dove $\varepsilon < 1$ dipende dalla qualità dell'euristica.

La complessità di A* dipende quindi fortemente dall'euristica che viene utilizzata.

### 7.3.1 Analisi della Complessità nell'Implementazione

Nel codice implementato per il gioco del 15 il fattore di branching è limitato a 4 (movimenti nelle quattro direzioni).

Nell'implementazione però vengono usati vari metodi per "filtrare" la ricerca. Il seguente frammento di codice illustra la generazione e il filtraggio dei nodi successori:

Esempio di codice 7.1: Generazione dei nodi successori

```
for (Board neighbor : neighborsList) {
    if (!visited.contains(neighbor)) {
        visited.add(neighbor);
        openSet.add(new State(neighbor, current.moves + 1, current));
    }
}
```

Il fattore di branching viene in questo modo ridotto in quanto:

1. la tessera vuota può avere 2, 3 o 4 movimenti possibili a seconda della posizione;
2. l'utilizzo del `Set<Board> visited` elimina la rivisitazione degli stati;

La funzione di priorità implementata segue la formulazione classica di A*:

$$f(n) = g(n) + h(n)$$

dove $g(n)$ rappresenta il costo del percorso dalla radice al nodo $n$, e $h(n)$ è la stima euristica del costo dal nodo $n$ alla soluzione.

Esempio di codice 7.2: Implementazione della funzione di priorità

```
public int priority() {
    return this.moves + config.manhattan();
}
```

L'euristica combinata utilizza:

- **Distanza di Manhattan**: $\mathcal{O}(1)$ calcola la distanza di ogni pedina dalla posizione "obiettivo";
- **Numero di mosse**: $\mathcal{O}(n)$ il numero esatto di mosse dalla configurazione iniziale

Esempio di codice 7.3: Calcolo delle distanze di Manhattan

```
    public int manhattan() {
    int dist = 0;
    for (int i = 0; i < this.firstDimension; i++) {
        for (int j = 0; j < this.secondDimension; j++) {
            int value = board[i][j];
            if (value != empty) {
                Pair targetPosition = pos.get(value);
                dist += Math.abs(i - targetPosition.first) + Math.abs(j -
                    targetPosition.second);

            }
        }
    }
    return dist;
}
```

La distanza di Manhattan ha complessità computazionale $O(N^2)$ per puzzle $NN$ in quanto, in questo caso, il calcolo viene svolto ogni volta, cioè non viene utilizzata alcuna tecnica di memoizzazione. Nel caso di utilizzo di memoizzazione tale complessità è riducibile al valore di $O(1)$.

Il codice include una verifica preliminare che previene calcoli inutili nel caso di disposizioni che non sono risolvibili. La complessità di questo codice è già stata trattata ed è nell'ordine di $O(n^2)$.

Questa ottimizzazione elimina a priori le configurazioni non risolvibili, evitando esplorazioni infinite dello spazio degli stati.

L'implementazione utilizza strutture dati ottimizzate per le operazioni critiche:

Esempio di codice 7.4: Inizializzazione delle strutture dati

```
PriorityQueue<State> openSet = new PriorityQueue<>();
Set<Board> visited = new HashSet<>();
```

Le complessità delle operazioni fondamentali sono:

- **PriorityQueue.poll()**: $O(logn)$ per l'estrazione del nodo con priorità minima;
- **HashSet.contains()**: $O(1)$ medio per la verifica di appartenenza;
- **PriorityQueue.add()**: $O(logn)$ per l'inserimento ordinato;

Il confronto tra stati è implementato direttamente nella classe `State`:

Esempio di codice 7.5: Implementazione del comparatore

```
@Override
```

```
public int compareTo(State y) {
if (this.priority() > y.priority()) {
return 1;
} else if (this.priority() < y.priority()) {
return -1;
} else {
return 0;
}
}
```

Questa implementazione garantisce un ordinamento corretto degli stati nella coda di priorità, fondamentale per il funzionamento ottimale di A*.

La distanza di Manhattan, pur essendo ammissibile, presenta alcune limitazioni:

- **Non considera i conflitti**: tessere che si bloccano reciprocamente non sono rilevate

Queste limitazioni possono portare a un aumento del numero di nodi esplorati rispetto a euristiche più informative come quella che combina Manhattan e conflitti lineari.

L'algoritmo A* implementato con la distanza di Manhattan come euristica presenta una complessità temporale teorica $O(b^d)$ con prestazioni che dipendono fortemente dalla qualità dell'euristica.

### 7.3.2 Algoritmo: A* con conflitti lineari

La versione dell'algoritmo A* implementata considera, oltre alla distanza di Manhattan, anche i *conflitti lineari* come parte dell'euristica. Questa scelta consente di rendere l'euristica più informativa e ridurre il numero di nodi esplorati.

#### Modifiche principali rispetto ad A* classico

- La funzione di priorità $f(n)$ viene calcolata come:

$$f(n) = g(n) + h(n) = \text{\#mosse dalla radice} + (\text{distanza di Manhattan} + \text{conflitti lineari})$$

  I conflitti lineari permettono di rilevare situazioni in cui due o più tessere si bloccano reciprocamente sulla stessa riga o colonna, aumentando il valore dell'euristica in modo da scoraggiare percorsi non ottimali.

- La classe `State` è stata aggiornata affinché il metodo `priority()` includa il calcolo dei conflitti lineari:

```
public int priority() {
    return config.linearConflicts() + config.manhattan();
}
```

- La struttura dei nodi successori e la gestione della coda di priorità rimangono identiche, ma la combinazione di Manhattan e conflitti lineari migliora la selezione del nodo successivo da esplorare.

**Effetti sull'efficienza**

- L'aggiunta dei conflitti lineari aumenta il costo computazionale per il calcolo dell'euristica di ciascun nodo. Tuttavia, poiché l'euristica è più informativa, il numero totale di nodi esplorati diminuisce sensibilmente rispetto all'uso della sola distanza di Manhattan.
- La complessità temporale rimane $\mathcal{O}(b^d)$ nel caso peggiore, ma nella pratica il fattore di branching effettivo viene ridotto grazie alla guida più precisa.
- L'implementazione continua a utilizzare un `HashSet` per i nodi visitati, evitando di riesplorare stati già considerati durante la ricerca.

**Funzionalità aggiuntive**

Questa versione dell'algoritmo permette di ottenere soluzioni più rapide e vicine al numero minimo di mosse rispetto all'A* basato solo sulla distanza di Manhattan, grazie alla maggiore precisione dell'euristica combinata.

### 7.3.3 Algoritmo risolutivo: IDA*

L'algoritmo IDA* (Iterative Deepening A*) è una variante dell'A* che combina la profondità limitata tipica della DFS con l'euristica di A*. A differenza di A*, IDA* utilizza una soglia (threshold) per guidare la ricerca in profondità, evitando di memorizzare tutti i nodi esplorati e riducendo significativamente l'uso della memoria.

- La funzione di costo $f(n) = g(n) + h(n)$ è calcolata esattamente come in A*, dove $g(n)$ è il numero di mosse dalla radice e $h(n)$ è l'euristica (somma di Manhattan e conflitti lineari, se implementata).
- La soglia iniziale viene impostata al costo del nodo iniziale $f(start)$.
- L'algoritmo esegue una ricerca in profondità limitata dalla soglia. Se un nodo supera la soglia, viene ignorato, ma viene registrato il minimo $f$ oltre la soglia per aggiornare la soglia nella prossima iterazione.
- La lista visited serve solo per evitare cicli all'interno della stessa iterazione ricorsiva, consentendo però di riesplorare gli stessi nodi in iterazioni successive.

L'implementazione utilizza:

- Una classe State che rappresenta ciascun stato del puzzle, memorizzando la configurazione, il numero di mosse e il nodo precedente per ricostruire il percorso.
- Un metodo ricorsivo search che esplora i vicini di ciascuno stato secondo la logica DFS e confronta $f(n)$ con la soglia corrente.
- La soglia viene aggiornata a ogni iterazione con il minimo $f(n)$ dei nodi che l'hanno superata, permettendo di approfondire gradualmente lo spazio degli stati.

Nel calcolo della complessità, IDA* si distingue principalmente per l'uso della memoria:

- Memoria: $O(d)$, dove $d$ è la profondità massima della soluzione. L'algoritmo memorizza solo il percorso corrente nella ricorsione, evitando di salvare tutti i nodi visitati come fa A*.
- Tempo: nel caso peggiore, IDA* può visitare ripetutamente gli stessi nodi in iterazioni successive. La complessità temporale è quindi $O(b^d)$, simile a quella di A*, ma con un fattore moltiplicativo dovuto alle ripetizioni causate dall'approccio iterativo.
- L'uso della lista visited locale riduce i cicli durante la DFS, ma non elimina completamente le riesplorazioni tra iterazioni successive.

IDA* richiede quindi molta meno memoria, in quanto non mantiene tutti i nodi esplorati in una coda di priorità globale.

## 7.4 Tabella riassuntiva

Tabella 7.1: Confronto tra A*, A* con conflitti lineari e IDA* per il gioco del 15

| **Caratteristica** | **A*** | **A* con conflitti lineari** | **IDA*** |
|---|---|---|---|
| Strategia di ricerca | Best-First (coda di priorità) | Best-First (coda di priorità) | Depth-First iterativa |
| Funzione costo | $f(n) = g(n) + h(n)$ | $f(n) = g(n) + h(n)$ (Manhattan + conflitti lineari) | $f(n) = g(n) + h(n)$ |
| Memoria richiesta | $O(b^d)$ | $O(b^d)$ | $O(d)$ |
| Tempo nel caso peggiore | $O(b^d)$ | $O(b^d)$ | $O(b^d)$ (riesplorazioni incluse) |
| Tempo nel caso migliore | $O(d)$ se l'euristica è perfetta | $O(d)$ se l'euristica è perfetta | $O(d)$ se l'euristica è perfetta |
| Tempo medio | Dipende dalla qualità dell'euristica | Migliore di A* grazie ai conflitti lineari | Dipende dal numero di soglie aggiornate |
| Uso dell'euristica | Manhattan | Manhattan + conflitti lineari | Manhattan (o altra euristica) |
| Gestione dei nodi già visitati | Evita rivisitazioni usando `visited` globale | Evita rivisitazioni usando `visited` globale | Evita cicli solo nella DFS corrente |

# 8

# Modellazione del gioco del 15 con Answer Set Programming (ASP)

## 8.1 Introduzione ad ASP [15]

L'Answer Set Programming (ASP) è un paradigma di programmazione dichiarativa basato sulla programmazione logica, adatto per la risoluzione di problemi di ricerca combinatoria e ragionamento automatico. A differenza della programmazione imperativa tradizionale, dove si specifica come risolvere un problema attraverso una sequenza di istruzioni, in ASP si descrive cosa costituisce una soluzione valida del problema. Il cuore di ASP risiede nel concetto di "answer set":
dato un programma logico, **un answer set è un insieme di atomi che soddisfa tutte le regole del programma in modo coerente e minimale**.
Il sistema ASP trova automaticamente tutti gli answer set possibili, che rappresentano le soluzioni del problema modellato.

## 8.2 Struttura e sintassi

In **ASP** (Answer Set Programming), i programmi sono rappresentati come un insieme finito di *regole* della forma: [10]
$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_k$
dove:

- $a$ è la *testa della regola*
- $b_i$, $c_j$ sono *atomi* della logica del primo ordine presenti nel *corpo della regola*;

In ASP questo viene rappresentato come: [15]

```
<testa> :- <corpo>.
```

dove:

- testa è un **atomo**, cioè una formula atomica ovvero un'espressione che non può essere scomposta ulteriormente usando connettivi logici;
- corpo è una **congiunzione** di letterali ovvero atomi positivi o negativi;
- $:-$ il simbolo indica un "if";

Le regole in ASP possono assumere diverse forme.

L'atomo nella testa risulta vero se tutti gli atomi positivi nel corpo possono essere derivati e nessuno degli atomi negati (*not* $c_j$) lo può essere, in questo caso la regola prende il nome di **fatto**.

Le *regole con testa vuota* rappresentano **vincoli di integrità**, utilizzati per esprimere condizioni di incompatibilità (ad esempio: due componenti che non possono coesistere).

Al contrario, una *regola con corpo vuoto* rappresenta un **fatto**, cioè un'informazione assunta come vera.

ASP permette anche la definizione di *vincoli di cardinalità*, che specificano scelte tra insiemi di atomi, come 'scegliere da 1 a 2 CPU da un insieme disponibile', queste regole prendono il nome di **regole di scelta**.

La risoluzione di un programma ASP avviene in due fasi: [8]

- prima il programma viene *groundato*, ovvero tutte le variabili vengono sostituite con costanti dell'universo di Herbrand;
- poi il programma senza variabili viene analizzato da un *risolutore ASP*, che calcola i *modelli stabili* (o *answer sets*), ossia le possibili soluzioni minimali che soddisfano tutte le regole.

## 8.3 Modellazione del contesto di lavoro a ASP

Per rappresentare il problema del gioco del 15 e trovare una sequenza minima di mosse per passare da una configurazione iniziale a una finale, utilizziamo *Answer Set Programming* (ASP).

Il modello ASP definisce un dominio temporale discreto, rappresentato dal predicato `tempo(0..t)`, dove $t$ è il massimo numero di mosse ammesse. La griglia $4 \times 4$ è modellata mediante il dominio delle coordinate `range(0..3)`.

Ogni stato del puzzle è descritto da fatti del tipo `cell(T,X,Y,V)`, che indicano che al tempo $T$ la tessera $V$ si trova nella cella con coordinate $(X, Y)$. La tessera vuota è rappresentata dal valore $0$.

Lo stato iniziale viene definito assegnando i fatti `cell(0,X,Y,V)` corrispondenti alla configurazione di partenza. Lo stato obiettivo è codificato come un vincolo che richiede che, al tempo finale $t$, la disposizione delle tessere corrisponda a quella della configurazione risolta.

Le mosse legali sono rappresentate da atomi `move(T,Dir)`, dove $Dir \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$ indica la direzione dello spostamento del buco nella mossa eseguita al tempo $T$. Il modello impone che per ogni $T$ venga eseguita al massimo una mossa e che questa sia valida, ovvero che lo spostamento non esca dalla griglia.

Le regole ASP definiscono la transizione di stato conseguente alla mossa, aggiornando la posizione del buco e delle tessere coinvolte, mentre le celle non modificate mantengono il loro valore per inerzia.

Infine, includiamo una direttiva di minimizzazione. Questo consente al solver ASP di trovare una soluzione con il numero minimo di mosse necessarie per risolvere il puzzle.

Qui vengono definiti i domini fondamentali del problema. La costante 't=50' stabilisce il numero massimo di mosse consentite. Il predicato 'tempo(0..t)' crea il dominio temporale, 'val(0..15)' definisce i possibili valori delle tessere (0-15, dove 0 rappresenta il buco), e 'range(0..3)' stabilisce le coordinate della griglia 4x4.

```
% Parametri
t=50.
tempo(0..t).
range(0..3).
```

Le seguenti righe di codice definiscono la struttura del predicato principale 'cell' che rappresenta lo stato completo del puzzle. Ogni fatto $cell(T, X, Y, V)$ indica che al tempo T, la cella nella riga X e colonna Y contiene il valore V. Di conseguenza viene definita una configurazione "iniziale" di partenza da cui il gioco inizia.

```
% Stato iniziale (esempio)
cell(0,0,0,0). cell(0,0,1,1). cell(0,0,2,7). cell(0,0,3,3).
cell(0,1,0,2). cell(0,1,1,6). cell(0,1,2,8). cell(0,1,3,4).
cell(0,2,0,5). cell(0,2,1,9). cell(0,2,2,11). cell(0,2,3,12).
cell(0,3,0,13). cell(0,3,1,10). cell(0,3,2,14). cell(0,3,3,15).
```

Allo stesso modo definiamo la configurazione finale del puzzle. Le prime due regole calcolano i valori corretti per ciascuna posizione usando la formula $4 * X + Y + 1$, che produce la sequenza $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$. La terza regola specifica che il buco (valore $0$) deve trovarsi nell'angolo in basso a destra. L'ultimo vincolo garantisce che tutte le posizioni soddisfino il goal.

```
% Stato finale desiderato
goal(X,Y) :- range(X), range(Y), X < 3, cell(t,X,Y,4*X+Y+1).
goal(X,Y) :- range(X), range(Y), X = 3, Y < 3, cell(t,X,Y,4*X+Y+1).
goal(3,3) :- cell(t,3,3,0).
:- range(X),range(Y), not goal(X,Y).
```

Questa regola di scelta garantisce che ad ogni istante temporale T venga selezionata esattamente una mossa tra le quattro possibili direzioni (su, giù, sinistra, destra).

```
% Una mossa per tempo
1 { move(T,up); move(T,down); move(T,left); move(T,right) } 1 :- tempo(T), T < t.
```

Questa regola identifica la posizione del buco (tessera vuota con valore 0) al tempo T, informazione necessaria per determinare quali mosse sono possibili.

```
hole(T,X,Y) :- cell(T,X,Y,0), tempo(T), range(X), range(Y).
```

Questi vincoli di integrità impediscono mosse illegali che porterebbero il buco fuori dai confini della griglia. Rispettivamente: non si può muovere su dalla riga 0, non si può muovere giù dalla riga 3, non si può muovere a sinistra dalla colonna 0, e non si può muovere a destra dalla colonna 3.

```
%mosse proibite
:- tempo(T), move(T,up),    range(Y), hole(T,0,Y).
:- tempo(T), move(T,down),  range(Y), hole(T,3,Y).
:- tempo(T), move(T,left),  range(X), hole(T,X,0).
:- tempo(T), move(T,right), range(X), hole(T,X,3).
```

Queste regole determinano quale tessera viene spostata in base alla direzione del movimento del buco. Quando il buco si muove in una direzione, la tessera nella direzione opposta viene spostata nella posizione del buco.

```
% Effetti
moved(T,X-1,Y) :-  hole(T,X,Y), move(T,up),    tempo(T),  range(X), range(Y).
moved(T,X+1,Y) :-  hole(T,X,Y), move(T,down),  tempo(T),  range(X), range(Y).
moved(T,X,Y-1) :-  hole(T,X,Y), move(T,left),  tempo(T),  range(X), range(Y).
moved(T,X,Y+1) :-  hole(T,X,Y), move(T,right), tempo(T),  range(X), range(Y).
```

Questo blocco implementa la transizione di stato. La prima regola stabilisce che la posizione della tessera spostata diventa il nuovo buco (valore 0). La seconda regola sposta il valore della tessera che era nella posizione spostata alla vecchia posizione del buco.

```
cell(T+1,X,Y,0)     :-
     moved(T,X,Y), tempo(T), tempo(T+1), range(X), range(Y).
cell(T+1,X1,Y1,V)   :-
    hole(T,X1,Y1), moved(T,X2,Y2), cell(T,X2,Y2,V),
    tempo(T), tempo(T+1), range(X1), range(X2), range(Y1), range(Y2), val(V).
```

Questo meccanismo implementa il principio di inerzia: le posizioni non coinvolte nella mossa (né il buco né la tessera spostata) mantengono il loro valore invariato nel tempo successivo.

```
% Inerzia
affected(T,X,Y) :- hole(T,X,Y),  tempo(T),  range(X), range(Y).
affected(T,X,Y) :- moved(T,X,Y), tempo(T),  range(X), range(Y).
```

sono le celle che non sono state spostate, in sostanza, perché valga cell(T+1,X,Y,V) devono valere le tre cose dopo.

```
cell(T+1,X,Y,V) :-  cell(T,X,Y,V),
       not affected(T,X,Y),
       tempo(T), tempo(T+1), range(X), range(Y), val(V).
```

| | | | |
|---|---|---|---|
| **16** | **1** | **7** | **3** |
| **2** | **6** | **8** | **4** |
| **5** | **9** | **11** | **12** |
| **13** | **10** | **14** | **15** |

| | | | |
|---|---|---|---|
| **5** | **1** | **3** | **4** |
| **2** | **7** | **8** | **12** |
| **9** | **6** | **11** | **15** |
| **13** | **10** | **14** | **16** |

| | | | |
|---|---|---|---|
| **1** | **2** | **3** | **4** |
| **5** | **6** | **7** | **8** |
| **9** | **10** | **11** | **12** |
| **14** | **13** | **15** | **16** |

Figura 8.1: Disposizioni esempio usate per misurare i tempo

La direttiva "show" specifica che nell'output devono essere mostrate solo le mosse eseguite. La direttiva 'minimize' ottimizza la soluzione cercando di minimizzare il numero totale di mosse necessarie per risolvere il puzzle.

```
#show move/2.
#minimize{1,T : move(T,_)}.
```

## 8.4 Risultati

Consideriamo una prova con 3 configurazioni del gioco del 15.

Di cui le prime due configurazioni hanno soluzione mentre la terza non è risolvibile.

Per confrontare il tutto calcoliamo il tempo impiegato da ASP con il tempo impiegato dal programma scritto in Java che risolve il problema con A*. Siccome in ASP dobbiamo specificare il numero di istanti temporali $t$ (numero di mosse) in cui vogliamo risolvere il puzzle, usiamo l'algotimo di A* per trovare una soluzione, ci facciamo comunicare il numero di mosse richieste e impostiamo t a tale valore.

Il programma A* trova una sequenze di mosse nei seguenti tempi:

- configurazione 1: 25ms e numero di mosse pari a 24;
- configurazione 2: 19ms e numero di mosse pari a 12;
- configurazione 3: 1ms, termina subito;

Il programma ASP trova una sequenza di mosse nei seguenti tempi:

- configurazione 1: 15.22s e numero di mosse pari a 24;
- configurazione 2: 156ms e numero di mosse pari a 12;
- configurazione 3: 150ms (t=10) termina senza trovare una soluzione e indicando che la configurazione fornita non è soddisfacibile;

# 9

# Conclusioni

In questo lavoro è stato affrontato il problema dell'instradamento di permutazioni su grafi tramite accoppiamenti, con particolare riferimento all'applicazione al gioco del 15. Partendo dal quadro teorico delineato da Alon, Chung e Graham, si è mostrato come il puzzle possa essere formalizzato in termini di permutazioni, gruppi e grafi, collegandolo direttamente al concetto di routing number.

Dopo aver fornito gli strumenti teorici di base, dalla definizione di permutazione e parità, fino ai richiami alla teoria della complessità e alla distinzione tra classi P e NP, il lavoro ha indagato diversi approcci algoritmici alla risoluzione del gioco come:

- algoritmi euristici come A* e IDA* con ottimizzazioni basate su distanza di Manhattan e conflitti lineari;
- una modellazione in Answer Set Programming (ASP), utile per evidenziare la flessibilità del paradigma dichiarativo;
- un tentativo di ridurre il problema al caso delle routing permutations su uno spanning tree, trattato da Alon, Chung e Graham nel paper da cui questo lavoro è iniziato, nella speranza di sfruttare la struttura gerarchica dell'albero per semplificare il processo di instradamento;

Quest'ultimo approccio, tuttavia, non ha portato ai risultati desiderati. La natura fortemente interdipendente delle mosse del gioco del 15 rende infatti impossibile trattare il problema come una semplice sequenza di sottoproblemi locali: fissare progressivamente le tessere, come avverrebbe in uno spanning tree, impedisce di considerare le configurazioni transitorie necessarie a raggiungere la soluzione. Ciò evidenzia un limite concreto nell'applicazione diretta del modello del routing number al puzzle e conferma che la sua risoluzione richiede metodi più flessibili e globali.

Dal punto di vista della complessità computazionale, è stato discusso come la ricerca della sequenza minima di mosse appartenga a problemi NP-completi. Dopo aver messo a confronto i due programmi su tre diverse disposizioni del gioco del 15 siamo giunti alla conclusione che gli algoritmi di ricerca, in particolare A*, hanno dimostrato di essere tra i più efficaci in termini di costi di calcolo, mentre ASP si è rivelato interessante come strumento di modellazione, pur risultando meno competitivo dal punto di vista prestazionale.

Il contributo principale di questo lavoro consiste dunque nell'aver mostrato come un classico problema combinatorio come il gioco del 15, possa essere reinterpretato attraverso il quadro teorico delle

permutazioni di instradamento su grafi tramite accoppiamenti. Il collegamento, pur non essendo pienamente risolutivo nel caso dello spanning tree, ha comunque permesso di chiarire i limiti del modello e di evidenziare la necessità di approcci ibridi.

Un aspetto che merita particolare attenzione riguarda la possibilità di sviluppare codici e algoritmi in grado di ridurre la complessità computazionale rispetto a quanto analizzato in questo lavoro. Gli approcci implementati, pur risultando efficaci, presentano ancora margini di miglioramento sia dal punto di vista teorico sia sul piano pratico.

In primo luogo, si potrebbero introdurre euristiche più sofisticate per l'algoritmo A* e per l'algoritmo IDA* volte a ridurre le riesplorazioni ridondanti.

Infine, sul piano teorico, resta aperta la possibilità di individuare sottoclassi di configurazioni del gioco del 15 per le quali il routing number possa essere calcolato in modo più efficiente. L'identificazione di queste strutture particolari permetterebbe di restringere il campo dei casi NP-completi e di delineare strategie ibride capaci di adattarsi alla natura specifica del problema.

# Bibliografia


[1] Noga Alon, F. R. K. Chung e R. L. Graham. "Routing Permutations on Graphs via Matchings". In: *SIAM Journal on Discrete Mathematics* (1994).

[2] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Pearson Education, 2001.

[3] Algorithms for Competitive Programming. *15 Puzzle Game: Existence Of The Solution*. 2022. URL: https://cp-algorithms.com/others/15-puzzle.html.

[4] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th. MIT Press, 2022.

[5] Agostino Dovier e Roberto Giacobazzi. *Fondamenti dell'informatica*. Bollati Boringhieri, 2020.

[6] John E.Hopcroft, Rajeev Motwani e Jeffrey D. Ullman. *Automi, Linguaggi e Calcolalibilità*. Pearson, 2009.

[7] Michael R. Garey e David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman e Company, 1979.

[8] Martin Gebser et al. *Chapter 2 - Basic modeling*. Springer Cham, 2013.

[9] Oded Goldreich. "Finding the Shortest Move-Sequence in the Graph-Generalized 15-Puzzle Is NP-Hard". In: *Lecture Notes in Computer Science* (1993).

[10] Lothar Hotz et al. *Chapter 6 - Configuration Knowledge Representation and Reasoning*. A cura di Alexander Felfernig et al. Morgan Kaufmann, 2014.

[11] Elena Konstantinova. "Some problems on Cayley graphs". In: *Elsevier* (2008).

[12] Richard E. Korf. "Depth-first iterative-deepening: An optimal admissible tree search". In: *Artificial Intelligence* 27.1 (1985), pp. 97–109. DOI: 10.1016/0004-3702(85)90084-0.

[13] Jamie Mulholland. *Permutation Puzzles, A Mathematical Perspective*. 2022. URL: https://courses.mai.liu.se/GU/TATA55/LECTURES/lecture5.pdf.

[14] S.Even e O.Goldreich. "The minimum-length generator sequence problem is NP-hard". In: *Journal of algorithms 2* (1981).

[15] Wikipedia contributors. *Answer Set Programming — Wikipedia, The Free Encyclopedia*. Accessed: 2025-08-04. 2025. URL: https://en.wikipedia.org/wiki/Answer_set_programming.