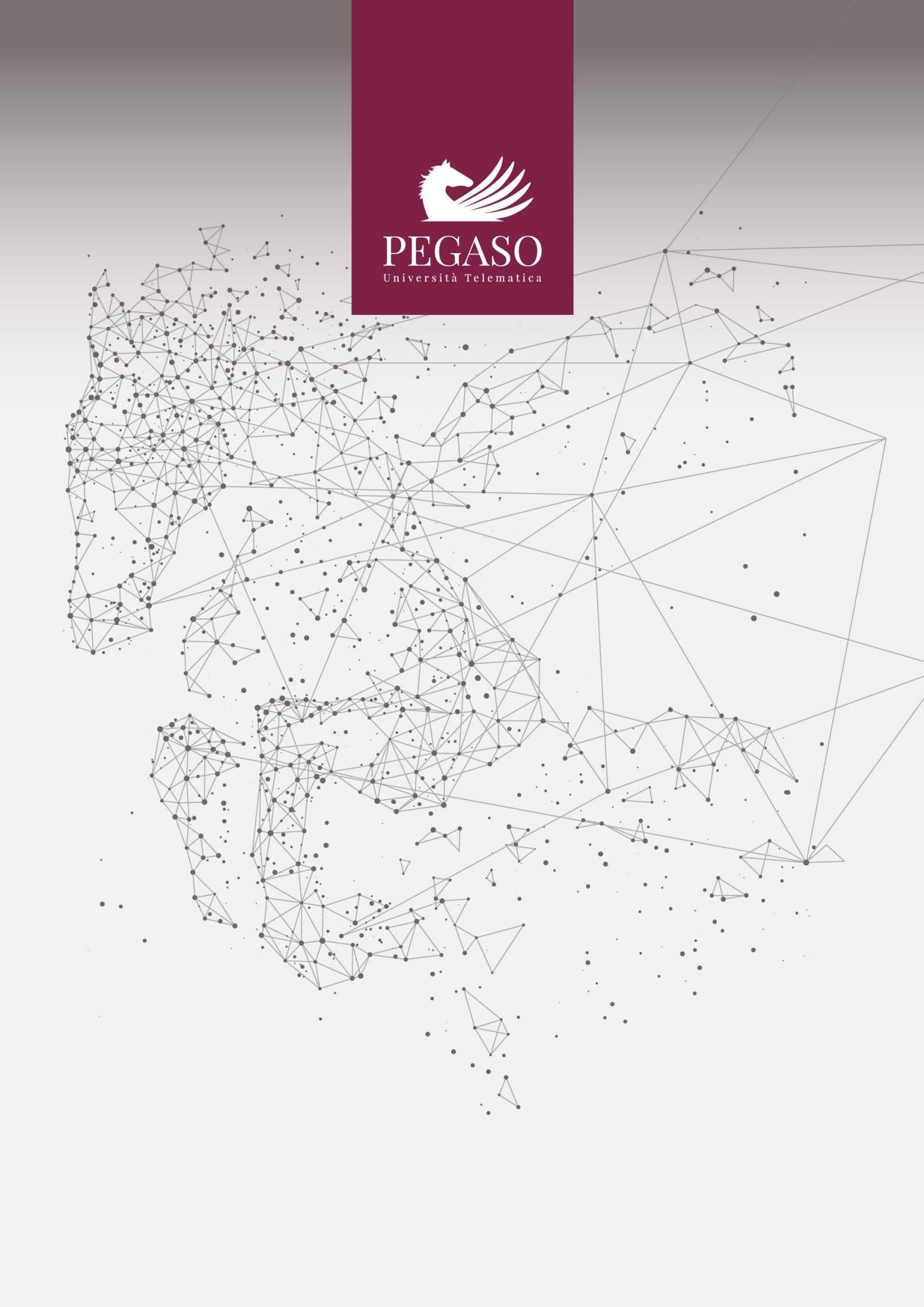
PEGASO
Università Telematica

Indice

1. Backtracking

La tecnica di risoluzione dei problemi chiamata "backtracking" è un approccio basato sull'elaborazione di alberi decisionali. Viene spesso utilizzato in ambito di programmazione e intelligenza artificiale per risolvere problemi che richiedono una sequenza di decisioni, come ad esempio problemi di ottimizzazione, di ricerca e di soddisfacimento di vincoli.

Il backtracking funziona esplorando le possibili soluzioni del problema attraverso un processo ricorsivo e incrementale: quando si incontra un ramo dell'albero decisionale che non porta a una soluzione valida, il backtracking "torna indietro" e prova una decisione alternativa, continuando a esplorare l'albero fino a trovare una soluzione o esaurire tutte le possibilità. In tal senso, il backtracking è una soluzione efficiente perché riesce ad evitare di esplorare rami dell'albero decisionale che sono sicuramente sbagliati o che non soddisfano i vincoli del problema.

Qualora invece si procedesse con l'analisi di tutte le soluzioni possibili, senza applicare alcuna strategia o euristica di ottimizzazione, si starebbe agendo in termini di **brute force attack**. Nella risoluzione di problemi computazionali, un approccio di brute force attack consiste nell'enumerare tutte le soluzioni possibili e valutarle una ad una: ad esempio, nel caso di elencare tutte le permutazioni di un insieme, si potrebbero generare tutte le combinazioni possibili e valutarle separatamente. Tuttavia, questo approccio può essere molto inefficiente, soprattutto per problemi con uno spazio delle soluzioni molto ampio, come trovare una sequenza di mosse per il gioco del 15.

Il backtracking cerca di evitare l'inefficienza del brute force attack esplorando lo spazio delle soluzioni in modo più mirato, eliminando rapidamente le soluzioni parziali che non soddisfano i vincoli del problema; ciò consente di ridurre notevolmente il numero di soluzioni che devono essere analizzate, risparmiando tempo e risorse computazionali.

Inoltre, il backtracking può essere adattato per fermarsi non appena viene trovata una soluzione valida, rendendolo ideale per trovare una soluzione ammissibile in uno spazio delle soluzioni molto grande.

Teniamo presente che lo spazio delle possibili soluzioni può avere dimensione superpolinomiale ed a volte è l'unica strada possibile; la potenza dei computer moderni rende "affrontabili" problemi di dimensioni medio-piccole:

- $10! = 3.63 \cdot 10^6$ (permutazione di 10 elementi)
- $2^{20} = 1.05 \cdot 10^6$ (sottoinsieme di 20 elementi)

A nostro vantaggio, a volte lo spazio delle soluzioni non deve essere analizzato interamente.

Qual è la filosofia del backtracking:

"Prova a fare qualcosa; se non va bene, disfalo e prova qualcos'altro"

"Ritenta, e sarai più fortunato"

L'implementazione del backtracking può avvenire sia in maniera iterativa che ricorsiva; entrambi gli approcci, iterativo e ricorsivo, possono essere efficaci per risolvere problemi di backtracking. La scelta tra i due dipende spesso dalle preferenze del programmatore, dalla comprensione del problema e dai requisiti di efficienza del programma.

Ecco i passi chiave del backtracking:

- Inizia con una soluzione parziale (inizialmente vuota).
- Espandi la soluzione parziale aggiungendo un nuovo elemento. Controlla se l'elemento corrente soddisfa i vincoli del problema.
- Se l'elemento corrente soddisfa i vincoli, procedi in profondità per esplorare ulteriori soluzioni parziali.
- Se si raggiunge una soluzione completa, registrare la soluzione e, se necessario, continuare a cercare altre soluzioni.
- Se l'elemento corrente non soddisfa i vincoli, esegui il "backtracking" e rimuovi l'elemento dalla soluzione parziale, tornando al passo precedente per provare un elemento alternativo.
- Continua il processo fino a quando tutte le soluzioni possibili sono state esplorate o si è trovata una soluzione ottimale.

Il backtracking può essere molto efficiente in alcuni problemi, soprattutto se si riesce a identificare rapidamente i rami dell'albero decisionale che non portano a soluzioni valide e a eliminarli dalla ricerca. Tuttavia, in problemi complessi o con molti vincoli, il backtracking può richiedere molto tempo e risorse computazionali. In questi casi, si possono utilizzare tecniche euristico o metaeuristiche per accelerare la ricerca delle soluzioni.

2. Gioco del 15

Il gioco del 15 è un rompicapo che consiste in una griglia 4x4 con 15 piastrelle numerate e uno spazio vuoto: l'obiettivo è riordinare le piastrelle in ordine numerico, spostandole nell'unico spazio vuoto disponibile, utilizzando il minor numero di mosse possibile.

Nel brute force attack, si generano tutte le possibili sequenze di mosse senza considerare i vincoli del gioco o la struttura dello spazio delle soluzioni: questo implica calcolare tutte le possibili configurazioni del puzzle e verificare se queste conducono alla soluzione desiderata.

Il gioco del 15 ha 16 caselle (15 piastrelle numerate e uno spazio vuoto), quindi, ci sono 16! (16 fattoriale) modi per disporre le piastrelle. Tuttavia, non tutte le permutazioni sono raggiungibili a partire da una configurazione valida: in realtà, solo la metà delle permutazioni è raggiungibile.

Per capire il motivo, dobbiamo considerare la "parità" di una permutazione, che è il numero di inversioni in essa. Un'inversione in una sequenza di numeri si verifica quando un numero più grande precede un numero più piccolo. Per esempio, consideriamo la seguente sequenza di numeri: [4, 1, 3, 2]. Ci sono 4 inversioni in questa sequenza:

- (4, 1)
- (4, 3)
- (4, 2)
- (3, 2)

La parità di un numero è il suo stato di essere pari o dispari. Nel contesto delle inversioni, possiamo parlare della parità del numero di inversioni presenti in una sequenza. Se il numero di inversioni è pari, la parità delle inversioni è pari; se il numero di inversioni è dispari, la parità delle inversioni è dispari.

Nel gioco del 15, la parità delle inversioni è importante per determinare se una configurazione del puzzle è risolvibile o meno. Una configurazione del gioco del 15 è risolvibile se la somma delle inversioni e la distanza della riga dello spazio vuoto dalla riga inferiore è pari.

Per esempio, consideriamo la seguente configurazione del gioco del 15:

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

In questa configurazione, lo spazio vuoto è nella riga inferiore, quindi la distanza della riga dello spazio vuoto dalla riga inferiore è 0; poiché la configurazione è già risolta, il numero di inversioni è anche 0. La somma delle inversioni e la distanza della riga dello spazio vuoto dalla riga inferiore è 0 + 0 = 0, che è un numero pari, il che indica che questa configurazione è risolvibile.

La parità di una configurazione di gioco del 15 indica se è risolvibile o meno; pertanto per determinare se una configurazione è risolvibile, si somma il numero di inversioni e la distanza della riga dello spazio vuoto dalla riga inferiore. Se questa somma è pari, allora la configurazione è risolvibile, altrimenti no.

Poiché solo la metà delle permutazioni ha la stessa parità delle inversioni, solo metà delle 16! configurazioni possibili sono risolvibili. Pertanto, ci sono 16! / 2 = 10,461,394,944,000 configurazioni raggiungibili nel gioco del 15. Questo numero è ancora molto grande, il che rende l'approccio di forza bruta estremamente inefficiente per risolvere il problema. Per questo motivo, si preferiscono tecniche più efficienti come il backtracking o gli algoritmi euristici come A* o IDA*.

Nel backtracking, si esplora lo spazio delle soluzioni in modo più mirato, cercando di costruire una sequenza di mosse valide che porti alla soluzione desiderata. A ogni passo, si valuta se la mossa corrente è valida e se migliora la soluzione parziale. Se ciò accade, si procede con la mossa successiva. Se invece si raggiunge un punto morto o si rileva che la soluzione parziale non può portare a una soluzione valida, si "torna indietro" e si prova un'altra mossa. In questo modo, si evita di esplorare rami dell'albero decisionale che non conducono a una soluzione.

In conclusione, il brute force attack per il gioco del 15 è molto inefficiente a causa del vasto spazio delle soluzioni da esplorare. Il backtracking offre un'alternativa più efficiente, poiché esplora lo spazio delle soluzioni in modo mirato, eliminando le soluzioni parziali che non portano a una soluzione valida. Tuttavia, per problemi complessi come il gioco del 15, tecniche euristiche come A* o IDA* possono offrire prestazioni ancora migliori.

Nel codice "brute force", utilizziamo la ricerca in ampiezza (Breadth-First Search, BFS) per esplorare tutte le possibili configurazioni del gioco del 15. La BFS esplora le configurazioni senza alcuna euristica, il che lo rende un approccio di forza bruta.

```
from collections import deque
from itertools import chain


def find_zero(puzzle):
    for i, row in enumerate(puzzle):
        for j, tile in enumerate(row):
            if tile == 0:
```

```
                return i, j


def get_neighbors(i, j):
    neighbors = [(i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1)]
    return [(x, y) for x, y in neighbors if 0 <= x < 4 and 0 <= y < 4]


def swap(puzzle, pos1, pos2):
    i1, j1 = pos1
    i2, j2 = pos2
    new_puzzle = [list(row) for row in puzzle]
    new_puzzle[i1][j1],    new_puzzle[i2][j2]    =    new_puzzle[i2][j2],
new_puzzle[i1][j1]
    return tuple(tuple(row) for row in new_puzzle)


def brute_force_solver(puzzle):
    goal = tuple(tuple(range(1 + 4 * i, 1 + 4 * (i + 1)) for i in
range(4)))
    goal = tuple(chain(*goal[:-1], (goal[-1][:-1] + (0,))))
    visited = set()
    queue = deque([(puzzle, [])])

    while queue:
        current, path = queue.popleft()

        if current == goal:
            return path

        if current in visited:
            continue

        visited.add(current)
        i, j = find_zero(current)

        for neighbor in get_neighbors(i, j):
            new_puzzle = swap(current, (i, j), neighbor)
            new_path = path + [neighbor]
            queue.append((new_puzzle, new_path))


# Esempio di utilizzo:
puzzle = (
    (1, 2, 3, 4),
    (5, 6, 7, 8),
    (9, 10, 11, 12),
```

```
    (13, 14, 0, 15)
)
solution = brute_force_solver(puzzle)
print("Brute force solution:", solution)
```

- find_zero(puzzle): questa funzione prende in input la griglia del puzzle e restituisce la posizione dello spazio vuoto (0)
- get_neighbors(i, j): questa funzione prende in input le coordinate (i, j) di una cella e restituisce una lista delle celle adiacenti valide all'interno della griglia 4x4
- swap(puzzle, pos1, pos2): questa funzione prende in input la griglia del puzzle e due posizioni (pos1 e pos2) e restituisce una nuova griglia con le celle in pos1 e pos2 scambiate
- brute_force_solver(puzzle): questa è la funzione principale che implementa l'approccio di forza bruta. Prende in input la configurazione iniziale del puzzle e restituisce la sequenza di mosse per risolverlo
    - Definiamo la configurazione obiettivo goal come una sequenza ordinata dei numeri da 1 a 15, con lo spazio vuoto (0) alla fine
    - Utilizziamo un set visited per memorizzare le configurazioni che abbiamo già esaminato, in modo da non ripetere lo stesso lavoro
    - Utilizziamo una coda queue per memorizzare le configurazioni da esaminare. Ogni elemento nella coda è una coppia (configurazione, percorso), dove "configurazione" è l'attuale disposizione delle piastrelle e "percorso" è la sequenza di mosse eseguite finora.
    - Entriamo in un ciclo while finché la coda non è vuota. In ogni iterazione:
        - Estraiamo il primo elemento dalla coda e lo assegniamo alle variabili current (configurazione corrente) e path (percorso corrente)
        - Se la configurazione corrente è uguale alla configurazione obiettivo, restituiamo il percorso corrente come soluzione
        - Se abbiamo già visitato la configurazione corrente, la saltiamo e passiamo all'iterazione successiva del ciclo
        - Altrimenti, aggiungiamo la configurazione corrente al set visited e troviamo la posizione dello spazio vuoto (0) nella griglia.
        - Per ogni cella adiacente valida allo spazio vuoto, scambiamo lo spazio vuoto con la cella adiacente per ottenere una nuova configurazione del puzzle. Aggiungiamo la nuova configurazione e il percorso aggiornato alla coda.

- Alla fine, nella sezione "Esempio di utilizzo", definiamo una configurazione iniziale del puzzle e chiamiamo la funzione brute_force_solver(puzzle) per trovare la soluzione

Soluzione mediante backtracking:

```
def is_goal(puzzle):
    n = len(puzzle)
    expected = list(range(1, n * n)) + [0]
    return puzzle == tuple(chain(*[tuple(row) for row in expected]))

def backtracking_solver(puzzle, path=[]):
    if is_goal(puzzle):
        return path

    i, j = find_zero(puzzle)
    neighbors = get_neighbors(i, j)

    for neighbor in neighbors:
        new_puzzle = swap(puzzle, (i, j), neighbor)
        if new_puzzle not in path:
            result = backtracking_solver(new_puzzle, path + [new_puzzle])
            if result is not None:
                return result

    return None

# Esempio di utilizzo:
puzzle = (
    (1, 2, 3, 4),
    (5, 6, 7, 8),
    (9, 10, 11, 12),
    (13, 14, 0, 15)
)
solution = backtracking_solver(puzzle)
print("Backtracking solution:", solution)
```

3. Il problema delle otto regine

Un esempio classico di problema risolvibile con il backtracking è il problema delle otto regine. Il problema consiste nel posizionare otto regine su una scacchiera 8x8 in modo che nessuna regina sia in grado di catturarne un'altra. In altre parole, le regine non devono trovarsi sulla stessa riga, colonna o diagonale.

Ecco come è possibile risolvere il problema delle otto regine usando il backtracking:

- Iniziamo con una scacchiera vuota e piazziamo la prima regina nella prima colonna della prima riga
- Proseguiamo aggiungendo una nuova regina alla colonna successiva, cercando di posizionarla in modo che non sia sotto attacco dalle regine già posizionate.
- Se riusciamo a posizionare la regina in modo valido, continuiamo ad esplorare questa soluzione parziale, passando alla colonna successiva e cercando di posizionare un'altra regina.
- Se posizioniamo tutte le otto regine in modo valido, abbiamo trovato una soluzione al problema.
- Se, invece, non riusciamo a trovare una posizione valida per la regina nella colonna corrente, eseguiamo il backtracking: rimuoviamo l'ultima regina posizionata e prova a spostarla in una posizione diversa nella colonna precedente.
- Continuiamo il processo di posizionamento e backtracking fino a quando non hai trovato tutte le soluzioni possibili o una soluzione specifica che soddisfi i criteri desiderati.

Il backtracking è particolarmente adatto per risolvere il problema delle otto regine, poiché consente di esplorare sistematicamente lo spazio delle soluzioni possibili, eliminando rapidamente le soluzioni parziali che violano i vincoli. Tuttavia, questo approccio può richiedere più tempo per problemi simili con scacchiere di dimensioni maggiori o con vincoli più complessi. In tali casi, si possono utilizzare tecniche euristiche per accelerare la ricerca delle soluzioni.

Implementazione in Python:

```
def is_safe(board, row, col, n):
     # Controlla la riga sul lato sinistro
     for i in range(col):
          if board[row][i] == 1:
               return False

     # Controlla la diagonale superiore sul lato sinistro
     for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
          if board[i][j] == 1:
```

```
                return False

     # Controlla la diagonale inferiore sul lato sinistro
     for i, j in zip(range(row, n, 1), range(col, -1, -1)):
          if board[i][j] == 1:
               return False

     return True


def solve_n_queens_util(board, col, n):
     if col >= n:
          return True

     for i in range(n):
          if is_safe(board, i, col, n):
               board[i][col] = 1

           if solve_n_queens_util(board, col + 1, n):
               return True

           # Esegui il backtracking
               board[i][col] = 0

     return False


def solve_n_queens(n):
     board = [[0 for _ in range(n)] for _ in range(n)]

     if not solve_n_queens_util(board, 0, n):
          print("Soluzione non trovata")
          return

     for row in board:
          print(row)


if __name__ == "__main__":
    n = 8
    solve_n_queens(n)
```

Questo codice definisce una funzione solve_n_queens che prende come input la dimensione della scacchiera n e risolve il problema delle n regine.

La funzione is_safe controlla se è possibile posizionare una regina in una posizione specifica senza violare i vincoli, mentre la funzione solve_n_queens_util implementa il backtracking per trovare una soluzione.

Nell'esempio sopra, il problema viene risolto per una scacchiera $8x8$ (problema delle otto regine), ma puoi cambiare il valore di n per risolvere il problema delle n regine su una scacchiera nxn.

4. Sudoku

Il backtracking può essere utilizzato per risolvere il Sudoku. Il Sudoku è un rompicapo che si basa su una griglia $9x9$ divisa in $3x3$ sottogriglie, chiamate "regioni".

L'obiettivo è riempire la griglia con numeri da 1 a 9 in modo tale che ogni riga, colonna e regione contenga ogni numero una sola volta.

Ecco un'implementazione semplice di un risolutore di Sudoku in Python utilizzando il backtracking:

```
def is_safe(board, row, col, num):
    # Controlla la riga
    if num in board[row]:
        return False

    # Controlla la colonna
    if num in [board[i][col] for i in range(9)]:
        return False

    # Controlla la regione
    start_row, start_col = row - row % 3, col - col % 3
    for i in range(3):
        for j in range(3):
            if board[i + start_row][j + start_col] == num:
                return False

    return True


def find_empty_cell(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i, j
    return None


def solve_sudoku(board):
    empty_cell = find_empty_cell(board)

    if not empty_cell:
        return True

    row, col = empty_cell

    for num in range(1, 10):
```

```
        if is_safe(board, row, col, num):
            board[row][col] = num

            if solve_sudoku(board):
                return True

            # Esegui il backtracking
            board[row][col] = 0

    return False


def print_sudoku(board):
    for row in board:
        print(row)


if __name__ == "__main__":
    sudoku_board = [
        [5, 3, 0, 0, 7, 0, 0, 0, 0],
        [6, 0, 0, 1, 9, 5, 0, 0, 0],
        [0, 9, 8, 0, 0, 0, 0, 6, 0],
        [8, 0, 0, 0, 6, 0, 0, 0, 3],
        [4, 0, 0, 8, 0, 3, 0, 0, 1],
        [7, 0, 0, 0, 2, 0, 0, 0, 6],
        [0, 6, 0, 0, 0, 0, 2, 8, 0],
        [0, 0, 0, 4, 1, 9, 0, 0, 5],
        [0, 0, 0, 0, 8, 0, 0, 7, 9],
    ]

    if solve_sudoku(sudoku_board):
        print_sudoku(sudoku_board)
    else:
        print("Soluzione non trovata")
```

Ecco una spiegazione dettagliata della strategia utilizzata per risolvere il Sudoku con il backtracking:

- **Trova una cella vuota**: cerca una cella vuota (contenente 0) nella griglia del Sudoku; se non ci sono celle vuote, significa che il Sudoku è stato risolto correttamente. La funzione find_empty_cell si occupa di trovare una cella vuota nella griglia.
- **Prova i numeri da 1 a 9**: per la cella vuota trovata, prova a inserire i numeri da 1 a 9 uno alla volta.

- **Verifica i vincoli**: controlla se il numero inserito soddisfa i vincoli del Sudoku, ovvero se il numero non è già presente nella stessa riga, colonna o regione $3x3$. La funzione is_safe si occupa di verificare questi vincoli.
- **Ricorsione**: Se il numero inserito soddisfa i vincoli, procedi in profondità nella ricorsione e passa alla cella vuota successiva. Utilizza la funzione solve_sudoku per continuare a riempire le celle vuote.
- **Backtracking**: Se il numero inserito non soddisfa i vincoli o se la ricorsione successiva fallisce nel trovare una soluzione, esegui il backtracking rimuovendo il numero inserito nella cella corrente e provando il numero successivo. Se tutti i numeri da 1 a 9 sono stati provati senza successo, torna indietro alla cella precedente e riprova con un numero diverso.
- **Ripeti**: Continua ad applicare la strategia di backtracking fino a quando tutte le celle vuote sono riempite correttamente o si esauriscono tutte le possibilità senza trovare una soluzione valida.

Il backtracking è una strategia efficace per risolvere il Sudoku perché esplora sistematicamente tutte le possibili combinazioni di numeri nelle celle vuote. Tuttavia, in alcuni casi, può richiedere molto tempo per esplorare tutte le combinazioni, soprattutto se il rompicapo è complesso o ha molte soluzioni possibili. In tali situazioni, si possono utilizzare tecniche euristiche o di pruning per accelerare la ricerca delle soluzioni.

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.