Backtrack

Giuseppe Persiano

Università di Salerno

Ottobre, 2020

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili
     - ★ marca $(\texttt{r}, \texttt{c})$ come *esaurita*

▸ Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili
     - ★ marca $(\texttt{r}, \texttt{c})$ come *esaurita*
     - ★ continue

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili
     - ★ marca $(\texttt{r}, \texttt{c})$ come *esaurita*
     - ★ continue
   - sia `nmove` prossima mossa ammissibile

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili
     - ★ marca $(\texttt{r}, \texttt{c})$ come *esaurita*
     - ★ continue
   - sia `nmove` prossima mossa ammissibile
   - marca $(\texttt{r}, \texttt{c})$ come *visitata*

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili
     - ★ marca $(\texttt{r}, \texttt{c})$ come *esaurita*
     - ★ continue
   - sia `nmove` prossima mossa ammissibile
   - marca $(\texttt{r}, \texttt{c})$ come *visitata*
   - fai push di $(\texttt{r}, \texttt{c}, \texttt{nmove})$

▸ Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili
     - ★ marca $(\texttt{r}, \texttt{c})$ come *esaurita*
     - ★ continue
   - sia `nmove` prossima mossa ammissibile
   - marca $(\texttt{r}, \texttt{c})$ come *visitata*
   - fai push di $(\texttt{r}, \texttt{c}, \texttt{nmove})$
   - fai mossa `nmove` e calcola la nuova posizione $(\texttt{newr}, \texttt{newc})$

▸ Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili
     - ★ marca $(\texttt{r}, \texttt{c})$ come *esaurita*
     - ★ continue
   - sia `nmove` prossima mossa ammissibile
   - marca $(\texttt{r}, \texttt{c})$ come *visitata*
   - fai push di $(\texttt{r}, \texttt{c}, \texttt{nmove})$
   - fai mossa `nmove` e calcola la nuova posizione $(\texttt{newr}, \texttt{newc})$
   - if $(\texttt{newr}, \texttt{newc}) == E$, return TRUE

Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili
     - ★ marca $(\texttt{r}, \texttt{c})$ come *esaurita*
     - ★ continue
   - sia `nmove` prossima mossa ammissibile
   - marca $(\texttt{r}, \texttt{c})$ come *visitata*
   - fai push di $(\texttt{r}, \texttt{c}, \texttt{nmove})$
   - fai mossa `nmove` e calcola la nuova posizione $(\texttt{newr}, \texttt{newc})$
   - if $(\texttt{newr}, \texttt{newc}) == E$, return TRUE
   - fai push di $(\texttt{newr}, \texttt{newc}, 0)$

▸ Backtrack generale

L'Algoritmo di BackTrack per il Maze

1. inizia dallo stato in cui tutte le caselle sono *libere* o *bloccate*
2. poni nello stack $(S = (sr, sc), 0)$ ad indicare che l'ultima casella visitata è S e che nessuna mossa è stata effettuata.
3. segna S come visitata
4. finché lo stack è non vuoto:
   - fai pop dallo stack $(\texttt{r}, \texttt{c}, \texttt{lmove})$,
   - se non ci sono altre mosse ammissibili
     - ★ marca $(\texttt{r}, \texttt{c})$ come *esaurita*
     - ★ continue
   - sia `nmove` prossima mossa ammissibile
   - marca $(\texttt{r}, \texttt{c})$ come *visitata*
   - fai push di $(\texttt{r}, \texttt{c}, \texttt{nmove})$
   - fai mossa `nmove` e calcola la nuova posizione $(\texttt{newr}, \texttt{newc})$
   - if $(\texttt{newr}, \texttt{newc}) == E$, return TRUE
   - fai push di $(\texttt{newr}, \texttt{newc}, 0)$
5. return FALSE

▸ Backtrack generale

Backtrack

1. Si parte dallo *stato iniziale* `statIn`

Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto

Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto
   - `(state, move)` $\leftarrow$ `Stack.Pop`

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto
   - `(state, move)` $\leftarrow$ `Stack.Pop`
   - Calcola `newMove`

Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto
   - `(state, move)` $\leftarrow$ `Stack.Pop`
   - Calcola `newMove`
     - ★ mossa dopo `move` ammissibile per `state`

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto
   - `(state, move)` $\leftarrow$ `Stack.Pop`
   - Calcola `newMove`
     - ⋆ mossa dopo `move` ammissibile per `state`
   - if `newMove` $\neq$ `None`

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto
   - `(state, move)` $\leftarrow$ `Stack.Pop`
   - Calcola `newMove`
     - ★ mossa dopo `move` ammissibile per `state`
   - if `newMove` $\neq$ `None`
     - ★ `Stack.Push(state, newMove)`

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto
   - `(state, move)` $\leftarrow$ `Stack.Pop`
   - Calcola `newMove`
     - ★ mossa dopo `move` ammissibile per `state`
   - if `newMove` $\neq$ `None`
     - ★ `Stack.Push(state, newMove)`
     - ★ Calcola lo stato `newState` ottenuta dall'eseguire `newMove` in `state`

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto
   - `(state, move)` $\leftarrow$ `Stack.Pop`
   - Calcola `newMove`
     - ★ mossa dopo `move` ammissibile per `state`
   - if `newMove` $\neq$ `None`
     - ★ `Stack.Push(state, newMove)`
     - ★ Calcola lo stato `newState` ottenuta dall'eseguire `newMove` in `state`
     - ★ If `newState` è uno stato **finale**, return `True`

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto
   - `(state, move)` $\leftarrow$ `Stack.Pop`
   - Calcola `newMove`
     - ★ mossa dopo `move` ammissibile per `state`
   - if `newMove` $\neq$ `None`
     - ★ `Stack.Push(state, newMove)`
     - ★ Calcola lo stato `newState` ottenuta dall'eseguire `newMove` in `state`
     - ★ If `newState` è uno stato **finale**, return `True`
     - ★ `Stack.Push(newState, None)`

▸ Backtrack per Maze

Backtrack

1. Si parte dallo *stato iniziale* `statIn`
2. Segna `(statIn, None)` come **visitata**
3. `Stack.Push(statIn, None)`
   - `None` indica che nessuna mossa è stata effettuata
   - lo stack contiene coppie `(state, move)`
4. Finché lo stack non è vuoto
   - `(state, move)` $\leftarrow$ `Stack.Pop`
   - Calcola `newMove`
     - ★ mossa dopo `move` ammissibile per `state`
   - if `newMove` $\neq$ `None`
     - ★ `Stack.Push(state, newMove)`
     - ★ Calcola lo stato `newState` ottenuta dall'eseguire `newMove` in `state`
     - ★ If `newState` è uno stato **finale**, return `True`
     - ★ `Stack.Push(newState, None)`
5. return `False`

▸ Backtrack per Maze

Backtrack

1. Definire lo stato
2. Come calcolare lo stato iniziale
3. Come marcare uno stato $(M, (r, c))$ come già visitato
4. Generare la prossima mossa `nmove` dopo `lmove` quando siamo in `state`
5. Eseguire una mossa `nmove` per andare da `state` a `newState`
6. Controllare se uno stato $(M, (r, c))$ è finale

Backtrack

1. Definire lo stato
   - Griglia M in cui ogni casella è
     - ★ **libera**
     - ★ **bloccata**
     - ★ **visitata**
     - ★ **finale**
   - Casella (r, c) attualmente occupata
2. Come calcolare lo stato iniziale
3. Come marcare uno stato $(M, (r, c))$ come già visitato
4. Generare la prossima mossa `nmove` dopo `lmove` quando siamo in `state`
5. Eseguire una mossa `nmove` per andare da `state` a `newState`
6. Controllare se uno stato $(M, (r, c))$ è finale

Backtrack

1. Definire lo stato
2. Come calcolare lo stato iniziale
   - Nessuna casella visitata (quindi solo **libere**, **bloccate**, o **finale**)
   - Posizione iniziale S
3. Come marcare uno stato $(M, (r, c))$ come già visitato
4. Generare la prossima mossa `nmove` dopo `lmove` quando siamo in `state`
5. Eseguire una mossa `nmove` per andare da `state` a `newState`
6. Controllare se uno stato $(M, (r, c))$ è finale

Backtrack

1. Definire lo stato
2. Come calcolare lo stato iniziale
3. Come marcare uno stato $(M, (r, c))$ come già visitato
   - poni $M[r][c] = visitata$
4. Generare la prossima mossa `nmove` dopo `lmove` quando siamo in `state`
5. Eseguire una mossa `nmove` per andare da `state` a `newState`
6. Controllare se uno stato $(M, (r, c))$ è finale

Backtrack

1. Definire lo stato
2. Come calcolare lo stato iniziale
3. Come marcare uno stato $(M, (r, c))$ come già visitato
4. Generare la prossima mossa `nmove` dopo `lmove` quando siamo in `state`
   - $N \rightarrow E \rightarrow S \rightarrow O$
5. Eseguire una mossa `nmove` per andare da `state` a `newState`
6. Controllare se uno stato $(M, (r, c))$ è finale

Backtrack

1. Definire lo stato
2. Come calcolare lo stato iniziale
3. Come marcare uno stato $(M, (r, c))$ come già visitato
4. Generare la prossima mossa `nmove` dopo `lmove` quando siamo in `state`
5. Eseguire una mossa `nmove` per andare da `state` a `newState`
   - calcola nuovi valori di r e c
6. Controllare se uno stato $(M, (r, c))$ è finale

Backtrack

1. Definire lo stato
2. Come calcolare lo stato iniziale
3. Come marcare uno stato $(M, (r, c))$ come già visitato
4. Generare la prossima mossa `nmove` dopo `lmove` quando siamo in `state`
5. Eseguire una mossa `nmove` per andare da `state` a `newState`
6. Controllare se uno stato $(M, (r, c))$ è finale
   - controlla se $M[r][c] = finale$

Backtrack

- Definiamo una classe Backtrack che ha solo

Backtrack

- Definiamo una classe Backtrack che ha solo
  - Costruttore

Backtrack

- Definiamo una classe Backtrack che ha solo
  - Costruttore
  - Metodo `Solve`

Backtrack

- Definiamo una classe Backtrack che ha solo
  - Costruttore
  - Metodo `Solve`

- Definiamo una classe Maze derivata da Backtrack che fornisce

Backtrack

- Definiamo una classe Backtrack che ha solo
  - Costruttore
  - Metodo `Solve`

- Definiamo una classe Maze derivata da Backtrack che fornisce
  - Costruttore

Backtrack

- Definiamo una classe Backtrack che ha solo
  - Costruttore
  - Metodo `Solve`

- Definiamo una classe Maze derivata da Backtrack che fornisce
  - Costruttore
  - initState

Backtrack

- Definiamo una classe Backtrack che ha solo
  - Costruttore
  - Metodo `Solve`

- Definiamo una classe Maze derivata da Backtrack che fornisce
  - Costruttore
  - initState
  - setVisited

Backtrack

- Definiamo una classe Backtrack che ha solo
  - Costruttore
  - Metodo `Solve`

- Definiamo una classe Maze derivata da Backtrack che fornisce
  - Costruttore
  - initState
  - setVisited
  - nextAdmMove

Backtrack

- Definiamo una classe Backtrack che ha solo
  - Costruttore
  - Metodo `Solve`

- Definiamo una classe Maze derivata da Backtrack che fornisce
  - Costruttore
  - initState
  - setVisited
  - nextAdmMove
  - makeMove

Backtrack

- Definiamo una classe Backtrack che ha solo
  - Costruttore
  - Metodo `Solve`

- Definiamo una classe Maze derivata da Backtrack che fornisce
  - Costruttore
  - initState
  - setVisited
  - nextAdmMove
  - makeMove
  - isFinal

Cosa ci guadagno?

- Non devo riscrivere il metodo `Solve`
  - Posso usare la classe Backtrack

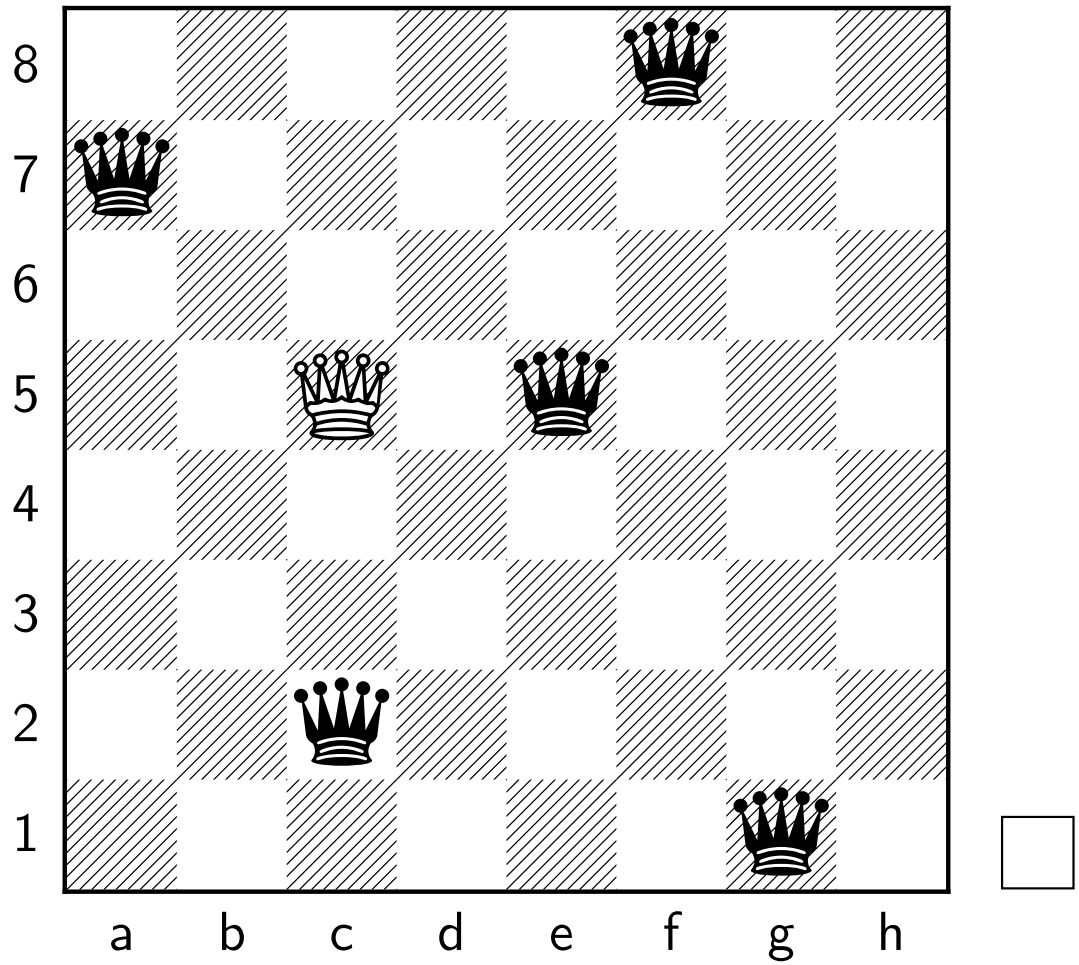
Cosa ci guadagno?

- Non devo riscrivere il metodo `Solve`
  - Posso usare la classe Backtrack

- Devo definire solo
  - Come è fatto lo stato
  - Costruttore
  - initState
  - setVisited
  - nextAdmMove
  - makeMove
  - isFinal

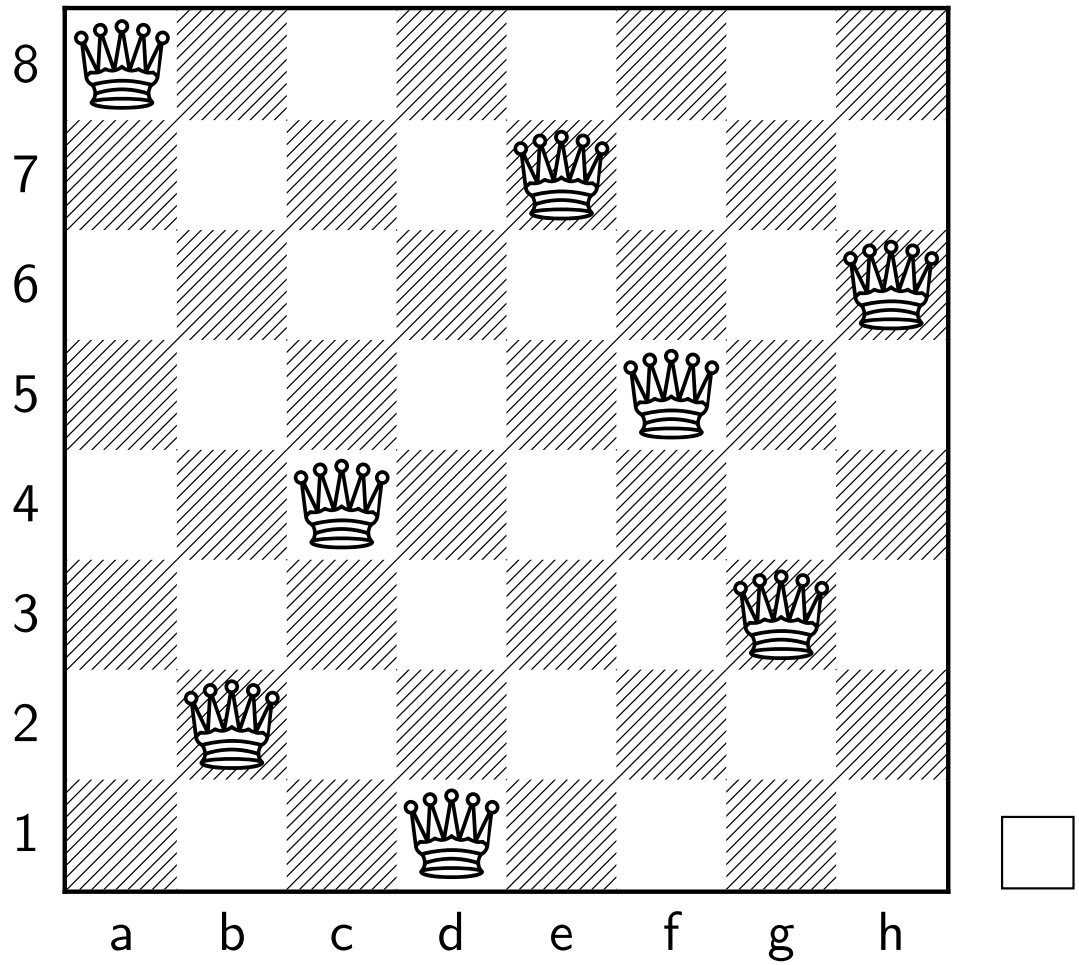
N regine

Piazzare 8 regine su una scacchiera in modo tale che nessuna regina attacchi un'altra regina.



N regine

Piazzare 8 regine su una scacchiera in modo tale che nessuna regina attacchi un'altra regina.



Cosa ci guadagno?

- Non devo riscrivere il metodo `Solve`
  - Posso usare la classe Backtrack

- Devo definire solo
  - Come è fatto lo stato
  - Costruttore
  - initState
  - setVisited
  - nextAdmMove
  - makeMove
  - isFinal

N regine

Lo stato

Osservazione: una riga può contenere una sola regina

- uno stato consiste di $(q, \texttt{nr})$
  - lista q di N interi $q[0], \ldots, q[N-1]$
    - $q[i]$ è la colonna in cui si trova la regina della riga i
  - indice `nr` della prossima riga senza regina

N regine – I Metodi

- Stato iniziale

$$q = [\texttt{None}, \ldots, \texttt{None}], \texttt{nr} = 0$$

N regine – I Metodi

- Stato iniziale

$$q = [\texttt{None}, \ldots, \texttt{None}], \texttt{nr} = 0$$

- makeMove(c) : $q[\texttt{nr}] = c; \texttt{nr}++$

N regine – I Metodi

- Stato iniziale

$$q = [\texttt{None}, \ldots, \texttt{None}], \texttt{nr} = 0$$

- makeMove(c) : $q[\texttt{nr}] = c; \texttt{nr}++$

- isFinal : $\texttt{nr} == N$

N regine – I Metodi

nextAdmMove

Siamo in stato $(q, \texttt{nr})$

- prova tutte le colonne in riga $\texttt{nr}$:

$$q[\texttt{nr}] + 1, \ldots, N - 1$$

e restituisce la prima di queste colonne che permette di piazzare una regina senza attaccare quelle che abbiamo messo nelle righe $0, \ldots, \texttt{nr} - 1$

N regine – I Metodi

nextAdmMove

Siamo in stato $(q, \texttt{nr})$

- prova tutte le colonne in riga $\texttt{nr}$:

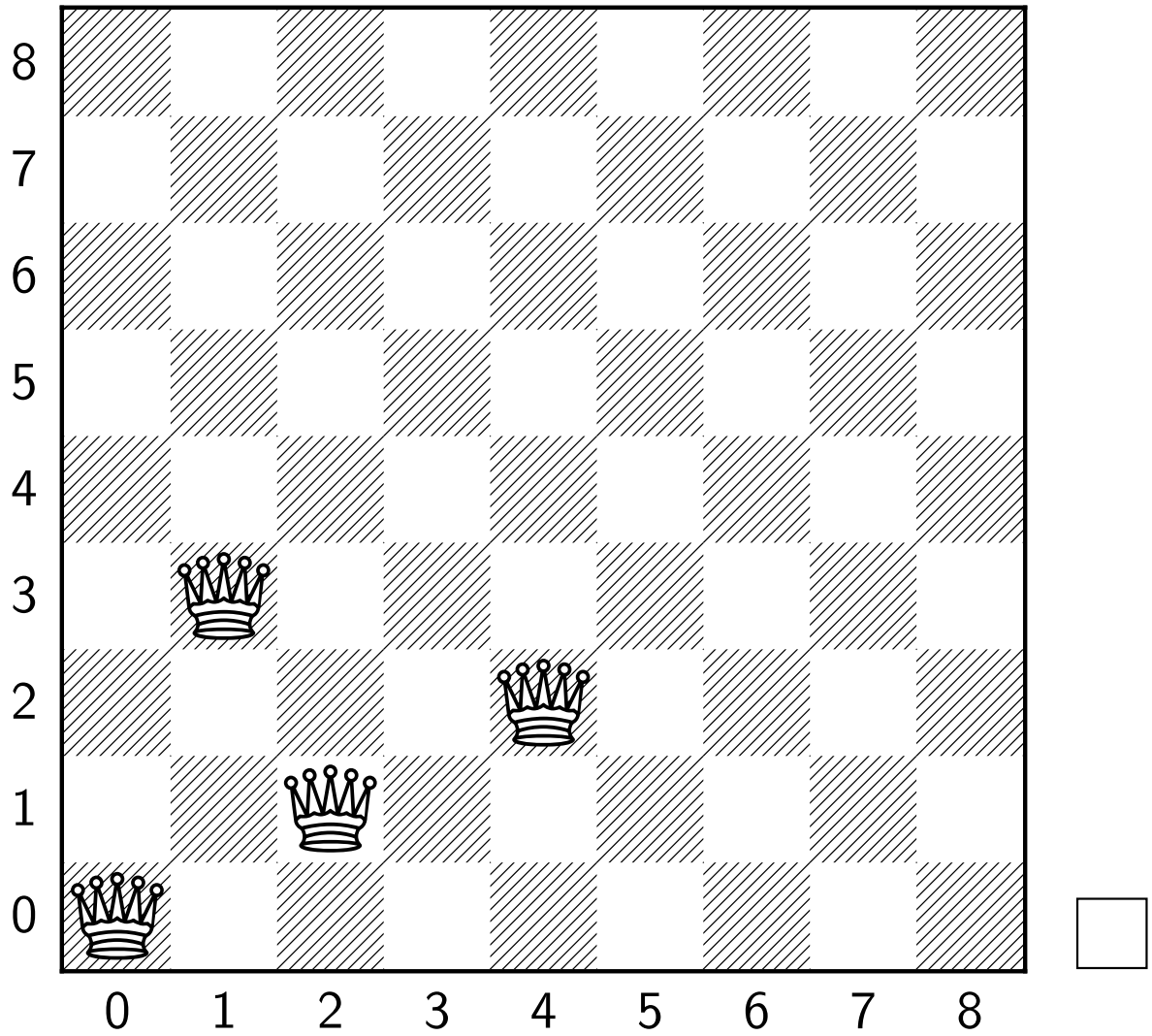
$$q[\texttt{nr}] + 1, \ldots, N - 1$$

e restituisce la prima di queste colonne che permette di piazzare una regina senza attaccare quelle che abbiamo messo nelle righe $0, \ldots, \texttt{nr} - 1$

  - le regine sono a

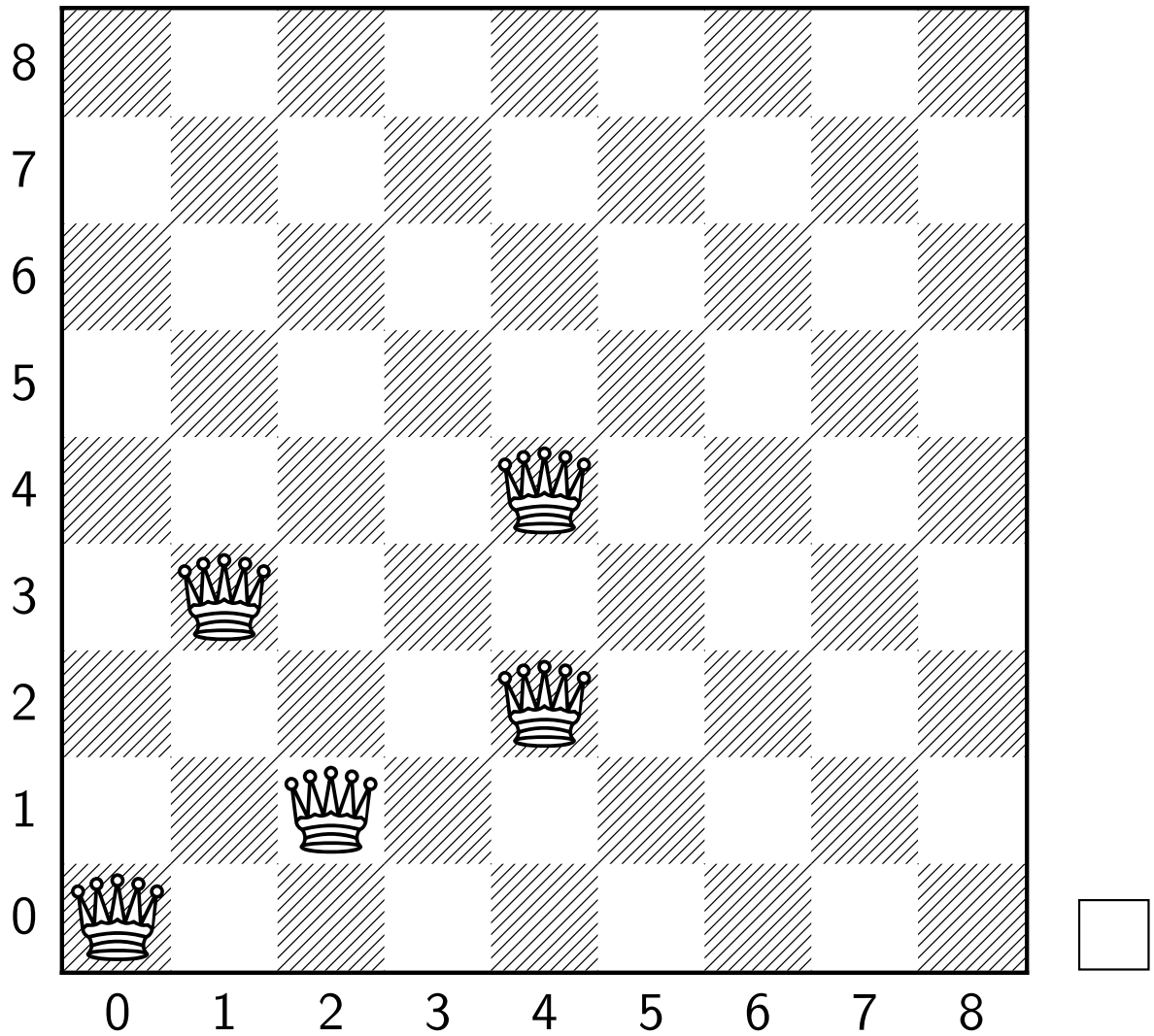
$$(0, q[0]), (1, q[1]), \ldots, (\texttt{nr} - 1, q[\texttt{nr} - 1])$$

Stato: $q = [0, 2, 4, 1, 3, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{None}], \qquad \texttt{nr} = 4$



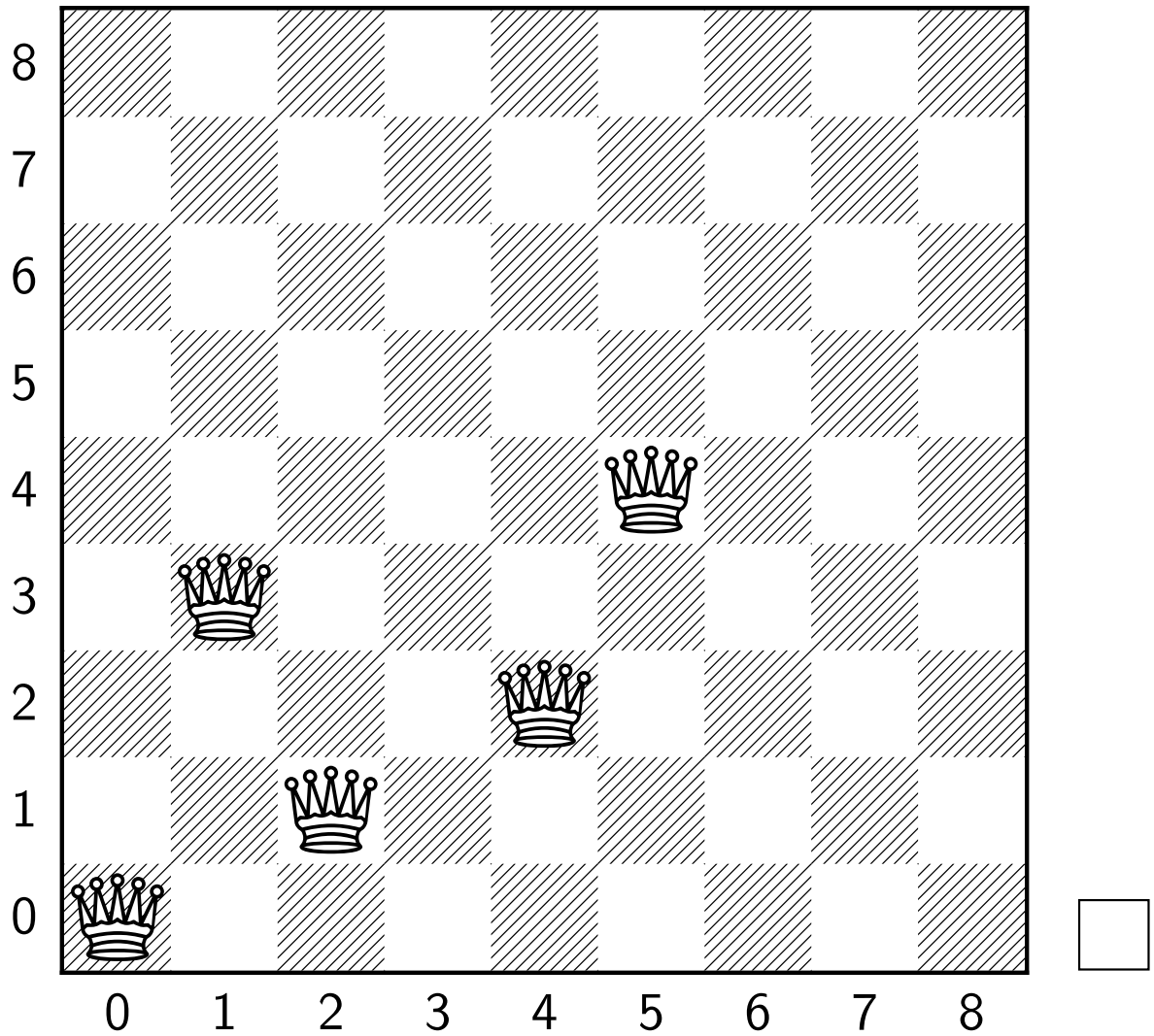
Stato: $q = [0, 2, 4, 1, 3, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{None}], \quad \texttt{nr} = 4$

Prova colonna 4 per riga 4 – Attacca regina in riga 2



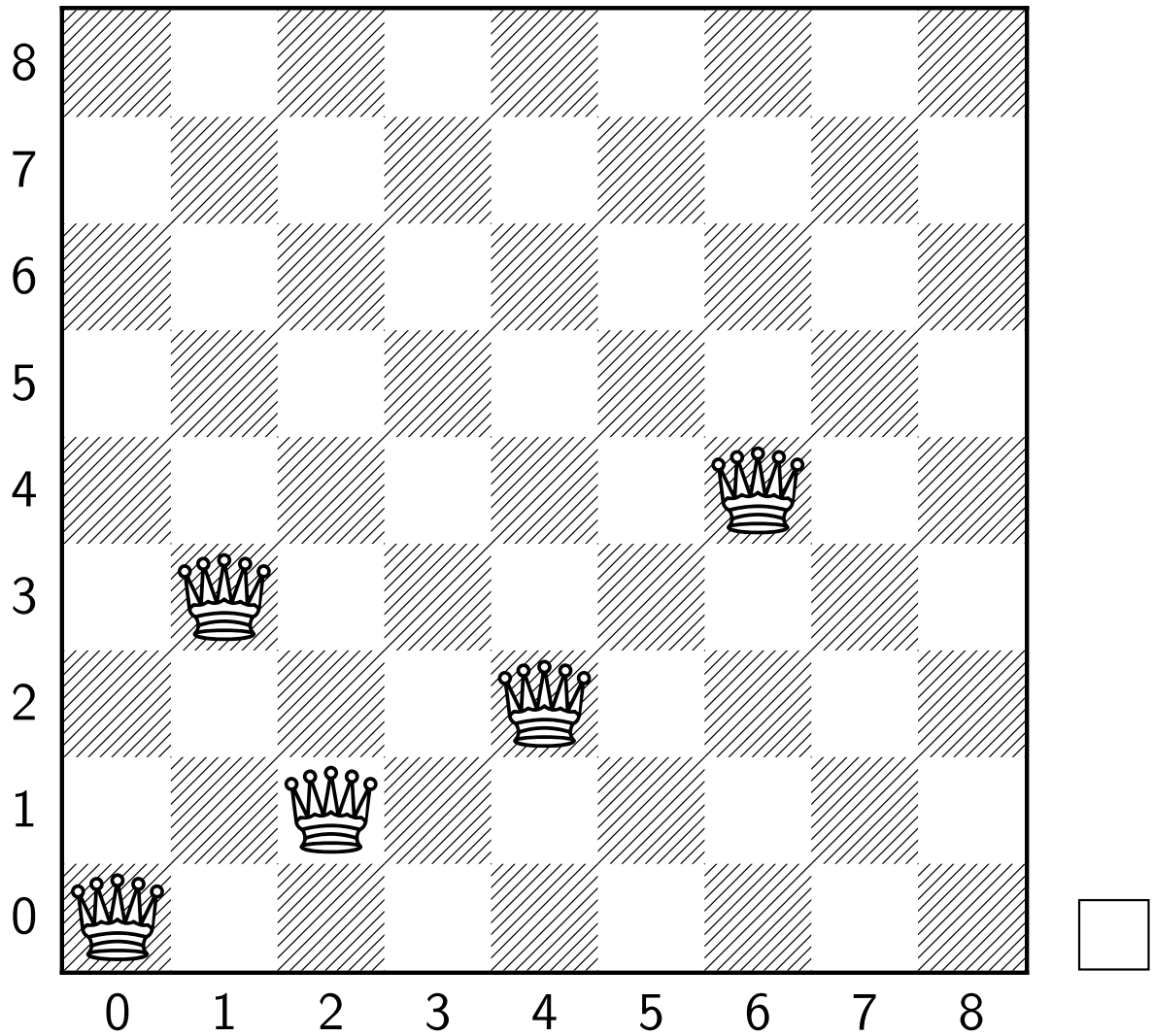
Stato: $q = [0, 2, 4, 1, 3, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{None}], \quad \texttt{nr} = 4$

Prova colonna 5 per riga 4 – Attacca regina in riga 1



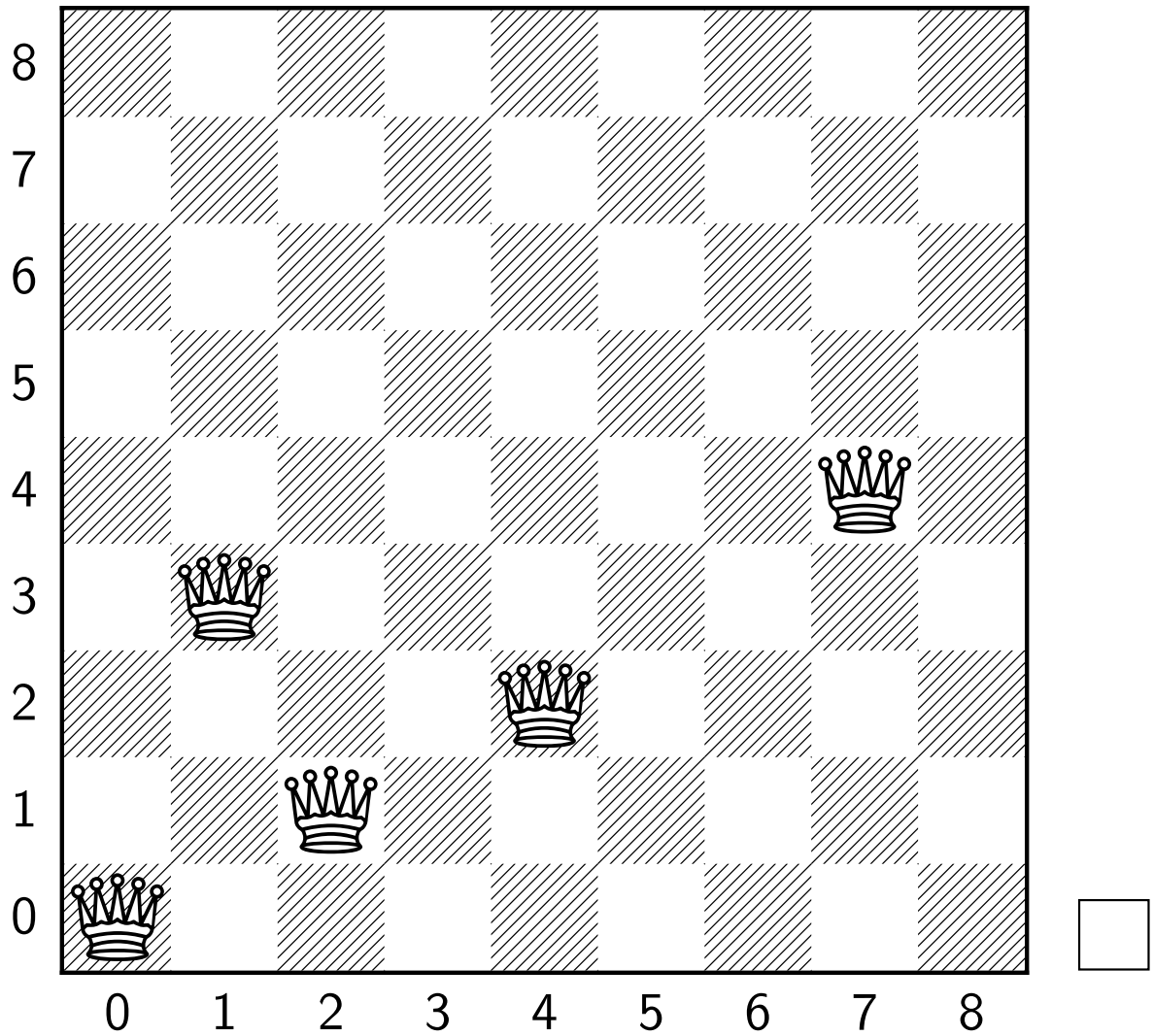
Stato: $q = [0, 2, 4, 1, 3, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{None}], \quad \texttt{nr} = 4$

Prova colonna 6 per riga 4 – Attacca regina in riga 2

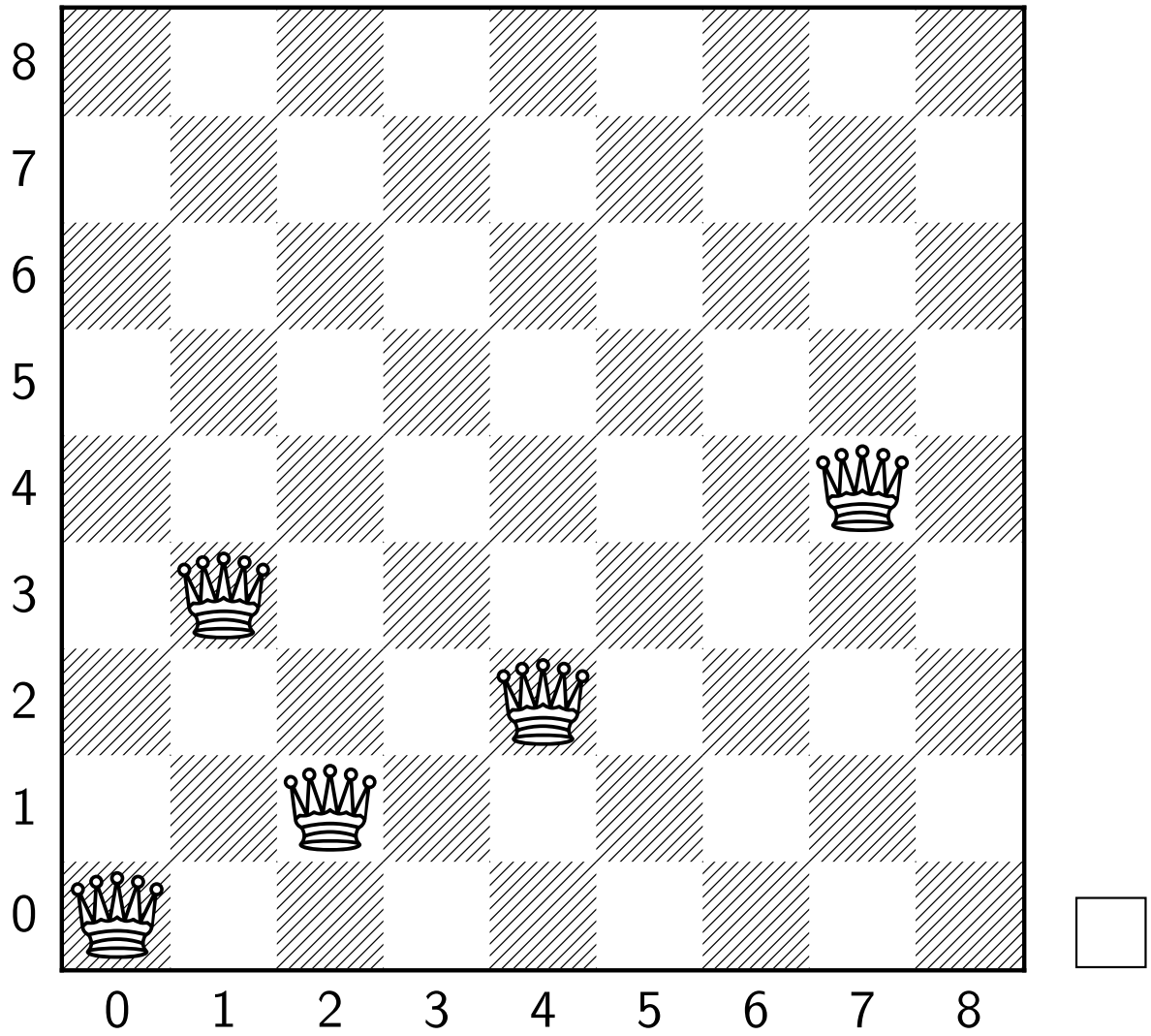


Stato: $q = [0, 2, 4, 1, 3, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{None}], \quad \texttt{nr} = 4$

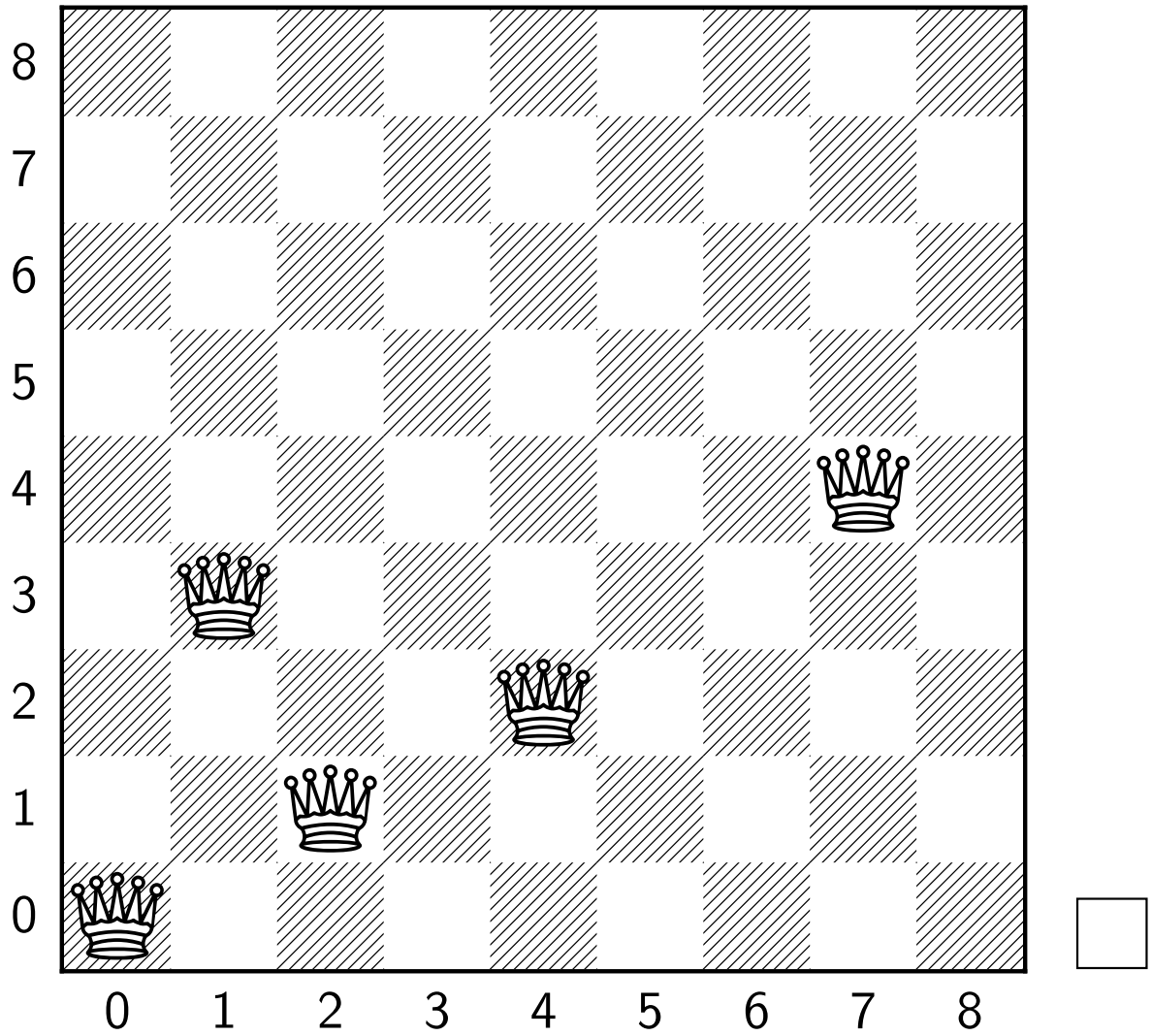
Prova colonna 7 per riga 4 – Nessun attacco



Nuovo Stato=$q = [0, 2, 4, 1, 7, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{None}]$, $\texttt{nr} = 5$



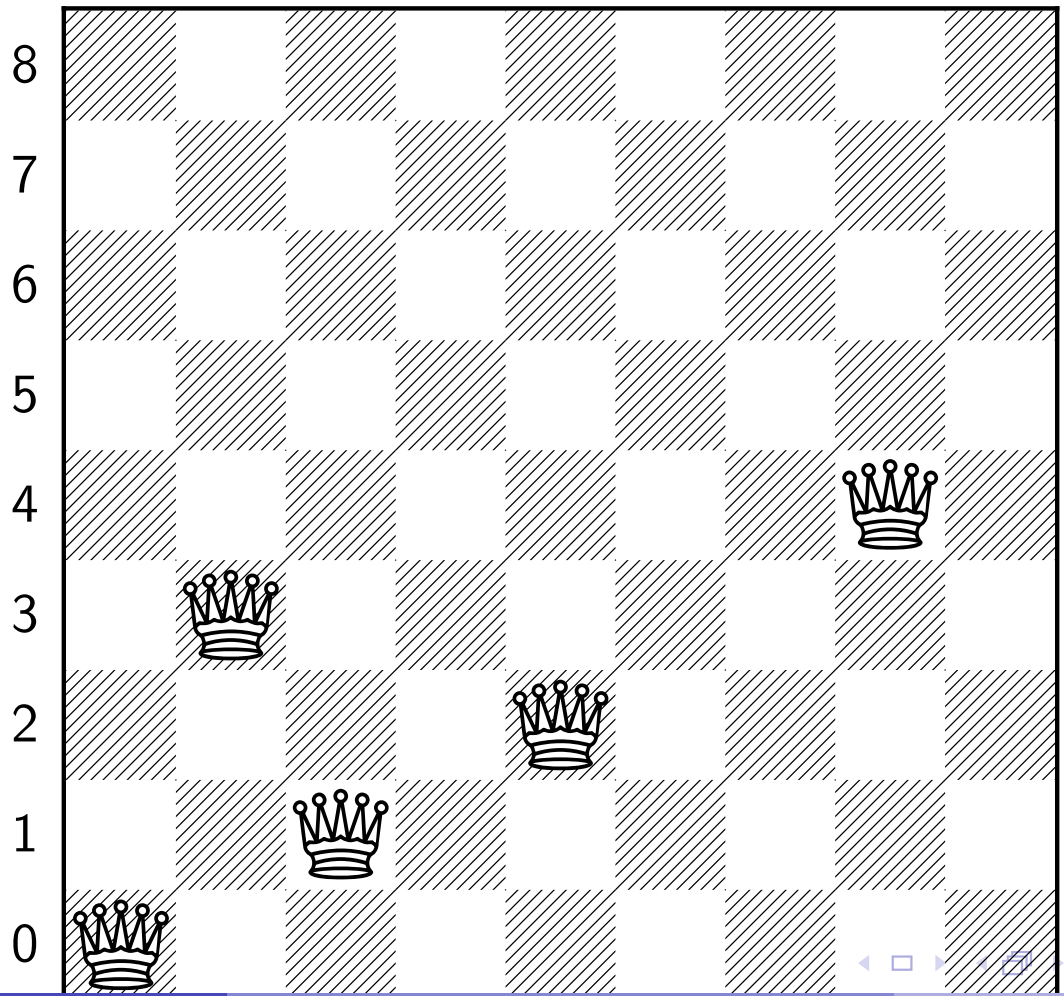
$\text{nextAdmMove}(q = [0, 2, 4, 1, 3, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{None}], \texttt{nr} = 4) = 7$



makeMove$((q = [0, 2, 4, 1, 3, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{None}], \texttt{nr} = 4), 7)$

$\rightarrow$

$(q = [0, 2, 4, 1, 7, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{None}], \texttt{nr} = 5)$



Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
  - $r_1 == r_2$

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
  - $r_1 == r_2$
- controlla se sono sulla stessa colonna:

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
  - $r_1 == r_2$
- controlla se sono sulla stessa colonna:
  - $c_1 == c_2$

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
  - $r_1 == r_2$
- controlla se sono sulla stessa colonna:
  - $c_1 == c_2$
- controlla se sono sulla stessa diagonale maggiore:

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
  - $r_1 == r_2$
- controlla se sono sulla stessa colonna:
  - $c_1 == c_2$
- controlla se sono sulla stessa diagonale maggiore:
  - $c_1 - r_1 == c_2 - r_2$

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
  - $r_1 == r_2$
- controlla se sono sulla stessa colonna:
  - $c_1 == c_2$
- controlla se sono sulla stessa diagonale maggiore:
  - $c_1 - r_1 == c_2 - r_2$
- controlla se sono sulla stessa diagonale minore:

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
  - $r_1 == r_2$
- controlla se sono sulla stessa colonna:
  - $c_1 == c_2$
- controlla se sono sulla stessa diagonale maggiore:
  - $c_1 - r_1 == c_2 - r_2$
- controlla se sono sulla stessa diagonale minore:
  - $c_1 + r_1 == c_2 + r_2$

Subset Sum

Subset Sum

Input:

- Una lista L di N interi positivi
- Un intero positivo target T

Output:

- Un sottoinsieme S degli interi di L la cui somma è uguale a T

Subset Sum

Subset Sum

Input:

- Una lista L di N interi positivi
- Un intero positivo target T

Output:

- Un sottoinsieme S degli interi di L la cui somma è uguale a T

Example

Subset Sum Input:

- $L = [3, 2, 11, 4, 17]$
- $T = 22$

Output:

- $S = \{3, 2, 17\}$

Subset Sum

Subset Sum

Input:

- Una lista L di N interi positivi
- Un intero positivo target T

Output:

- Un sottoinsieme S degli interi di L la cui somma è uguale a T

Example

Subset Sum Input:

- $L = [3, 2, 11, 4, 17]$
- $T = 36$

Output:

- Nessuna soluzione
- $36 = 2 + 2 + 11 + 4 + 17$ non è una soluzione perché 2 è usato due volte

Usiamo Backtrack

Definiamo gli stati

- consideriamo gli interi $L[i]$ della lista L uno per volta, $i = 0, \ldots, N-1$
- i è il prossimo intero da considerare
- abbiamo due mosse possibili per ogni intero $L[i]$
  - Mossa 0: non aggiungere $L[i]$ a S
  - Mossa 1: aggiungere $L[i]$ a S

Usiamo Backtrack

Definiamo gli stati

- consideriamo gli interi $L[i]$ della lista L uno per volta, $i = 0, \ldots, N-1$
- i è il prossimo intero da considerare
- abbiamo due mosse possibili per ogni intero $L[i]$
  - Mossa 0: non aggiungere $L[i]$ a S
  - Mossa 1: aggiungere $L[i]$ a S

Example

$(2, [0, 1, \texttt{None}, \texttt{None}, \texttt{None}])$

- abbiamo scartato $L[0] = 3$ ed aggiunto $L[1] = 2$
- dobbiamo ancora decidere su $L[2] = 11, L[3] = 4$ e $L[4] = 17$
- prossima mossa:
  - scartare $L[2]$
- prossimo stato

$$(3, [0, 1, 0, \texttt{None}, \texttt{None}])$$

Usiamo Backtrack

Definiamo gli stati

- consideriamo gli interi $L[i]$ della lista L uno per volta, $i = 0, \ldots, N-1$
- i è il prossimo intero da considerare
- abbiamo due mosse possibili per ogni intero $L[i]$
  - Mossa 0: non aggiungere $L[i]$ a S
  - Mossa 1: aggiungere $L[i]$ a S

Example

$(2, [0, 1, 0, \texttt{None}, \texttt{None}])$

- abbiamo scartato $L[0] = 3$, aggiunto $L[1] = 2$ e scartato $L[2] = 11$
- dobbiamo ancora decidere su $L[3] = 4$ e $L[4] = 17$
- prossima mossa:
  - aggiungere $L[2]$
- prossimo stato

$$(3, [0, 1, 1, \texttt{None}, \texttt{None}])$$

Usiamo Backtrack

Definiamo gli stati

- consideriamo gli interi $L[i]$ della lista L uno per volta, $i = 0, \ldots, N-1$
- i è il prossimo intero da considerare
- abbiamo due mosse possibili per ogni intero $L[i]$
  - Mossa 0: non aggiungere $L[i]$ a S
  - Mossa 1: aggiungere $L[i]$ a S

Example

$(2, [0, 1, 1, \texttt{None}, \texttt{None}])$

- abbiamo scartato $L[0] = 3$, aggiunto $L[1] = 2$ e $L[2] = 11$
- dobbiamo ancora decidere su $L[3] = 4$ e $L[4] = 17$
- Non esiste una prossima mossa

Esercizio

Definire una classe SubsetSum derivata da Backtrack che fornisce

- Costruttore
- initState
- setVisited
- nextAdmMove
- makeMove
- isFinal