

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*

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.

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

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:

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 $(r, c, lmove)$,

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 $(r, c, lmove)$,
 - ▶ se non ci sono altre mosse ammissibili

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 ($r, c, lmove$),
 - ▶ se non ci sono altre mosse ammissibili
 - ★ marca (r, c) come *esaurita*

L'Algoritmo di BackTrack per il Maze

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

L'Algoritmo di BackTrack per il Maze

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

L'Algoritmo di BackTrack per il Maze

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

L'Algoritmo di BackTrack per il Maze

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

L'Algoritmo di BackTrack per il Maze

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

L'Algoritmo di BackTrack per il Maze

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

L'Algoritmo di BackTrack per il Maze

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

L'Algoritmo di BackTrack per il Maze

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

Backtrack

- 1 Si parte dallo *stato iniziale* **statIn**

Backtrack

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

Backtrack

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

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

- 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

- 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

- 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) ← Stack.Pop`

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) ← Stack.Pop`
 - ▶ Calcola `newMove`

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) ← Stack.Pop`
 - ▶ Calcola `newMove`
 - ★ mossa dopo `move` ammissibile per `state`

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) ← Stack.Pop`
 - ▶ Calcola `newMove`
 - ★ mossa dopo `move` ammissibile per `state`
 - ▶ if `newMove ≠ None`

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) ← Stack.Pop`
 - ▶ Calcola `newMove`
 - ★ mossa dopo `move` ammissibile per `state`
 - ▶ if `newMove ≠ None`
 - ★ `Stack.Push(state, newMove)`

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) ← Stack.Pop`
 - ▶ Calcola `newMove`
 - ★ mossa dopo `move` ammissibile per `state`
 - ▶ if `newMove ≠ None`
 - ★ `Stack.Push(state, newMove)`
 - ★ Calcola lo stato `newState` ottenuta dall'eseguire `newMove` in `state`

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) ← Stack.Pop`
 - ▶ Calcola `newMove`
 - ★ mossa dopo `move` ammissibile per `state`
 - ▶ if `newMove ≠ None`
 - ★ `Stack.Push(state, newMove)`
 - ★ Calcola lo stato `newState` ottenuta dall'eseguire `newMove` in `state`
 - ★ If `newState` è uno stato **finale**, return `True`

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) ← Stack.Pop`
 - ▶ Calcola `newMove`
 - ★ mossa dopo `move` ammissibile per `state`
 - ▶ if `newMove ≠ 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

- 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) ← Stack.Pop`
 - ▶ Calcola `newMove`
 - ★ mossa dopo `move` ammissibile per `state`
 - ▶ if `newMove ≠ 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

- 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 $nmov$ dopo $lmov$ quando siamo in $state$
- 5 Eseguire una mossa $nmov$ 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 $M[r][c] = \textit{visitata}$
- 4 Generare la prossima mossa \textit{nmove} dopo \textit{lmove} quando siamo in \textit{state}
- 5 Eseguire una mossa \textit{nmove} per andare da \textit{state} a $\textit{newState}$
- 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 n e r
- 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 $M[r][c] = \text{finale}$

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?

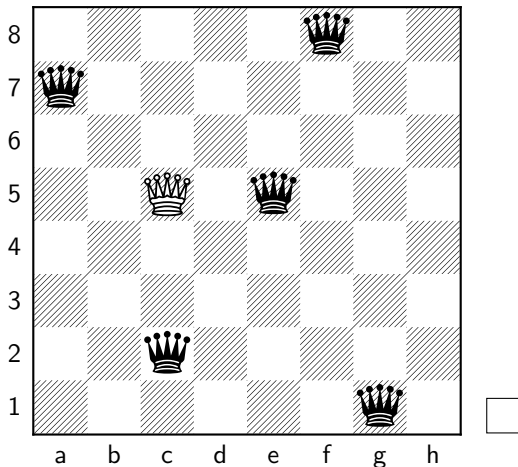
- Non devo riscrivere il metodo `Solve`
 - ▶ Posso usare la classe `Backtrack`

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`

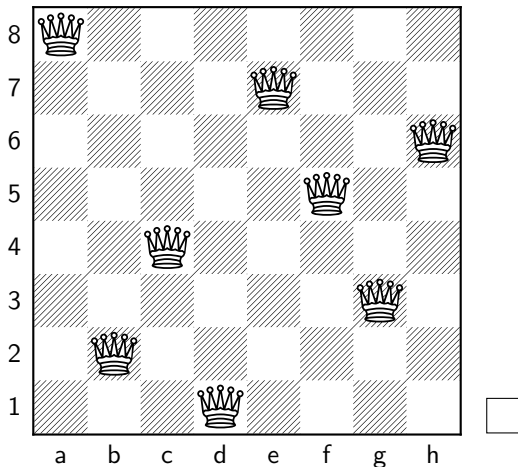
N regine

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, nr)
 - ▶ lista q di N interi $q[0], \dots, q[N-1]$
 - ★ $q[i]$ è la colonna in cui si trova la regina della riga i
 - ▶ indice nr della prossima riga senza regina