Andrea Augello
Department of Engineering, University of Palermo, Italy

La ricerca e il problem solving

Università degli Studi di Palermo

Ricerca su grafi

Ricerca su grafi

Molti problemi possono essere modellati come dei problemi di ricerca su grafi.

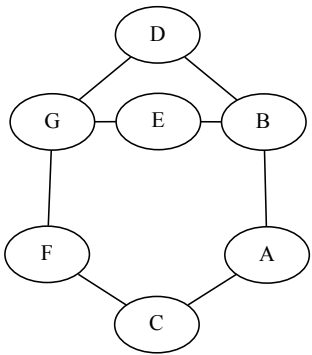
- **Stato iniziale**: stato da cui parte la ricerca
- **Stato finale**: stato che si vuole raggiungere
- **Stato corrente**: stato in cui si trova l'algoritmo
- **Operatore**: azione che modifica lo stato corrente
- **Cammino**: sequenza di stati che porta dallo stato iniziale a quello corrente
- **Costo del cammino**: costo totale per raggiungere lo stato corrente

Non tutte queste informazioni sono necessarie e/o disponibili per ogni problema.

Ricerca su grafi

- Iniziamo modellando un grafo in modo esplicito.
- Il grafo è rappresentabile con un dizionario di liste (di adiacenza).
- Le chiavi del dizionario esterno sono i nodi del grafo.
- Se il grafo non è orientato ogni arco è rappresentato due volte.

```
graph = {'A': ['B', 'C'],
         'B': ['A', 'D', 'E'],
         'C': ['A', 'F'],
         'D': ['B', 'G'],
         'E': ['B', 'G'],
         'F': ['C', 'G'],
         'G': ['D', 'E', 'F']}
```



Ricerca su grafi

In questo primo esempio, il problema è quello di trovare un cammino che porti dal nodo `A` al nodo `D` (se esiste) attraversando il minor numero di archi.

- **Stato iniziale**: `A`
- **Stato finale**: `D`
- **Stato corrente**: Il nodo in cui si trova l'algoritmo ad ogni passo
- **Operatore**: Spostarsi su un nodo adiacente
- **Costo del cammino**: Numero di archi attraversati

Volendo attraversare il minor numero di archi, possiamo utilizzare una ricerca in ampiezza.

Ricerca su grafi

Implementiamo un agente che esegue una ricerca in ampiezza.

```
class Agent():
    def __init__(self, graph, start, goal):
        self.graph = graph
        self.goal  = goal
        self.frontier = [[start]]

    def next_states(self, path):
        pass

    def is_goal(self, state):
        pass

    def bfs(self):
        #...
        yield from self.bfs()
```

L’algoritmo BFS

1. Se la frontiera è vuota, restituisci `None`
2. Prendi il primo cammino dalla frontiera
3. Se l’ultimo stato del cammino è uno stato goal, restituisci il cammino
4. Altrimenti, genera tutti i cammini ottenibili aggiungendo un nuovo stato alla fine del cammino (evitando cicli)
5. Aggiungi i nuovi cammini alla frontiera
6. Ripeti dal passo `1`

Ricerca su grafi

Implementazioni alternative:

- Riusciamo ad implementare lo stesso comportamento rappresentando il grafo come lista di archi?
- La ricerca in ampiezza può essere implementata in modo iterativo?

Ricerca su grafi

Non sempre è possibile modellare un grafo in modo esplicito.

- Il numero di stati può essere infinito.
- Il grafo può essere troppo grande per essere memorizzato.
- Il grafo può essere dinamico.
- L'adiacenza può essere determinata da una funzione.

La funzione `next_states` può quindi dover fare valutazioni meno banali.

- Facciamo quindi uso di operatori di trasformazione di stato
- Dato uno stato ed una azione, un operatore di trasformazione di stato restituisce un nuovo stato.

Cannibali e missionari

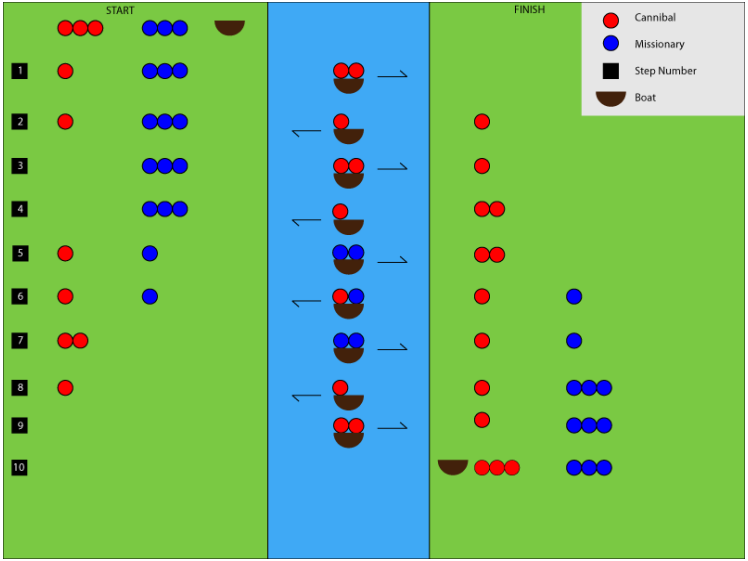
Cannibali e missionari

Tre missionari e tre cannibali devono attraversare un fiume.

- Missionari e cannibali sono inizialmente sulla sponda sinistra del fiume e devono raggiungere la sponda destra.
- Per farlo, hanno a disposizione una barca che può contenere al massimo due persone.
- La barca non può attraversare il fiume senza nessuno a bordo.
- Il numero di cannibali non può mai essere superiore al numero di missionari su una delle due sponde.

L'obbiettivo è quello di trovare una sequenza minima di azioni che permetta a tutti di attraversare il fiume.

Cannibali e missionari



Cannibali e missionari

Guardiamo lo scheletro di un agente che risolve il problema.

```
class Agent():
    def __init__(self):
        pass

    def is_valid(self, state):
        pass # Controlla se lo stato rispetta le regole del problema

    def apply_move(self, state, move):
        pass # Applica una mossa ad uno stato

    def next_states(self, state):
        pass # Genera tutti gli stati raggungibili da uno stato

    def is_goal(self, state):
        pass # Verifica se uno stato sia lo stato finale

    def bfs(self):
        pass # Implementa la ricerca in ampiezza
```

Rappresentazione dello stato

Il primo problema che bisogna porsi in un problema di ricerca è come rappresentare lo stato.

Rappresentazione dello stato

Il primo problema che bisogna porsi in un problema di ricerca è come rappresentare lo stato.

In questo caso, rappresentiamo lo stato come un dizionario:

```
start = {'L': {'m': 3, 'c': 3},
         'R': {'m': 0, 'c': 0},
         'boat':  'L'}
```

```
goal = {'L': {'missionaries': 0, 'cannibals': 0},
        'R': {'missionaries': 3, 'cannibals': 3},
        'boat':  'R'}
```

Movimenti possibili

Rappresentare esplicitamente tutti gli stati possibili e le loro transizioni sarebbe troppo dispendioso.

Definiamo quindi una lista di azioni possibili che, applicate ad uno stato, generano un nuovo stato.

Movimenti possibili

Rappresentare esplicitamente tutti gli stati possibili e le loro transizioni sarebbe troppo dispendioso.

Definiamo quindi una lista di azioni possibili che, applicate ad uno stato, generano un nuovo stato.

Data la posizione della barca, le azioni possibili sono:

- `{'missionaries': 1, 'cannibals': 0}`: spostare un missionario
- `{'missionaries': 0, 'cannibals': 1}`: spostare un cannibale
- `{'missionaries': 1, 'cannibals': 1}`: spostare un missionario e un cannibale
- `{'missionaries': 2, 'cannibals': 0}`: spostare due missionari
- `{'missionaries': 0, 'cannibals': 2}`: spostare due cannibali

Non tutte le azioni sono però sempre possibili o risultano in uno stato valido.

Controllare la validità di uno stato

Necessitiamo di una funzione che ci permetta di verificare se uno stato è valido secondo le regole del problema per rimuovere gli stati non validi dalla frontiera.

Uno stato è valido se in ogni sponda il numero di missionari è maggiore o uguale al numero di cannibali, oppure se non ci sono missionari su quella sponda (e non ci sono quantità negative di persone).

Controllare la validità di uno stato

Necessitiamo di una funzione che ci permetta di verificare se uno stato è valido secondo le regole del problema per rimuovere gli stati non validi dalla frontiera.

Uno stato è valido se in ogni sponda il numero di missionari è maggiore o uguale al numero di cannibali, oppure se non ci sono missionari su quella sponda (e non ci sono quantità negative di persone).

```
def is_valid(self, state):
  for side in ['L', 'R']:
      if state[side]['m'] < 0 or state[side]['c'] < 0:
          return False
      if state[side]['m'] > 0 and state[side]['c'] > state[side]['m']:
          return False
  return True
```

Applicare una mossa

Dato uno stato e una mossa, dobbiamo generare un nuovo stato. Per prima cosa copiamo lo stato corrente. Cambiamo la posizione della barca, e rimuoviamo le persone dalla sponda di partenza e le aggiungiamo alla sponda di arrivo.

Applicare una mossa

Dato uno stato e una mossa, dobbiamo generare un nuovo stato. Per prima cosa copiamo lo stato corrente. Cambiamo la posizione della barca, e rimuoviamo le persone dalla sponda di partenza e le aggiungiamo alla sponda di arrivo.

```
def apply_move(self, state, move):
    new_state = copy.deepcopy(state)
    new_state['boat'] = 'L' if state['boat'] == 'R' else 'R'
    for person in ['m', 'c']:
        new_state[state['boat']][person] -= move[person]
        new_state[new_state['boat']][person] += move[person]
    return new_state
```

Generazione degli stati successivi

Creiamo nuovi stati prendendo in considerazione tutte le azioni possibili.
Solamente le azioni che non violano le regole del problema verranno restituite.

Generazione degli stati successivi

Creiamo nuovi stati prendendo in considerazione tutte le azioni possibili.
Solamente le azioni che non violano le regole del problema verranno restituite.

```
def next_states(self, state):
    next_states = []
    for move in self.moves:
        new_state = self.apply_move(state, move)
        if self.is_valid(new_state):
            next_states.append(new_state)
    return next_states
```

Generazione degli stati successivi

Creiamo nuovi stati prendendo in considerazione tutte le azioni possibili.
Solamente le azioni che non violano le regole del problema verranno restituite.

```
def next_states(self, state):
    next_states = []
    for move in self.moves:
        new_state = self.apply_move(state, move)
        if self.is_valid(new_state):
            next_states.append(new_state)
    return next_states
```

Implementazione alternativa:

```
def next_states(self, state):
    return [new_state for move in self.moves if self.is_valid(new_state :=
        self.apply_move(state, move))]
```

Sequenza di stati

A questo punto, possiamo eseguire una ricerca in ampiezza per trovare una sequenza di stati che porti dallo stato iniziale a quello finale.

Questo codice è identico a quello che abbiamo visto per il grafo esplicito perché abbiamo raggiunto un livello di astrazione sufficientemente elevato.

Problema

La sequenza di stati è molto scomoda da leggere.

```
[
{'L': {'missionaries': 3, 'cannibals': 3}, 'R': {'missionaries': 0, 'cannibals': 0}, 'boat': 'L'},
{'L': {'missionaries': 3, 'cannibals': 1}, 'R': {'missionaries': 0, 'cannibals': 2}, 'boat': 'R'},
{'L': {'missionaries': 3, 'cannibals': 2}, 'R': {'missionaries': 0, 'cannibals': 1}, 'boat': 'L'},
{'L': {'missionaries': 3, 'cannibals': 0}, 'R': {'missionaries': 0, 'cannibals': 3}, 'boat': 'R'},
{'L': {'missionaries': 3, 'cannibals': 1}, 'R': {'missionaries': 0, 'cannibals': 2}, 'boat': 'L'},
{'L': {'missionaries': 1, 'cannibals': 1}, 'R': {'missionaries': 2, 'cannibals': 2}, 'boat': 'R'},
{'L': {'missionaries': 2, 'cannibals': 2}, 'R': {'missionaries': 1, 'cannibals': 1}, 'boat': 'L'},
{'L': {'missionaries': 0, 'cannibals': 2}, 'R': {'missionaries': 3, 'cannibals': 1}, 'boat': 'R'},
{'L': {'missionaries': 0, 'cannibals': 3}, 'R': {'missionaries': 3, 'cannibals': 0}, 'boat': 'L'},
{'L': {'missionaries': 0, 'cannibals': 1}, 'R': {'missionaries': 3, 'cannibals': 2}, 'boat': 'R'},
{'L': {'missionaries': 0, 'cannibals': 2}, 'R': {'missionaries': 3, 'cannibals': 1}, 'boat': 'L'},
{'L': {'missionaries': 0, 'cannibals': 0}, 'R': {'missionaries': 3, 'cannibals': 3}, 'boat': 'R'}
]
```

Occorre una funzione che consenta di ottenere una sequenza di azioni.

Sequenza di azioni

```
def states_to_moves(self, path):
    moves = []
    for i in range(len(path) - 1):
        for move in self.moves:
            if self.apply_move(path[i], move) == path[i + 1]:
                moves.append(str(move['m']) + ' missionaries and ' +
                    str(move['c']) + ' cannibals from ' +
                    path[i]['boat'] + ' to ' + path[i + 1]['boat'])
                break
    return moves

def solve(self):
    path = next(self.bfs())
    if path:
        print(*path, sep = '\n')
        print(*self.states_to_moves(path), sep='\n')
```

Sequenza di azioni

Implementazione alternativa utilizzando `pairwise` di `itertools`:

```
from itertools import pairwise

def states_to_moves(self, path):
    for i,j in pairwise(path):
        for move in self.moves:
            if self.apply_move(i, move) == j:
                yield str(move['m']) + ' missionaries and ' +
                    str(move['c']) + ' cannibals from ' + i['boat'] +
                    ' to ' + j['boat']
                break
```

Cannibali e missionari

- 0 missionaries and 2 cannibals from L to R
- 0 missionaries and 1 cannibals from R to L
- 0 missionaries and 2 cannibals from L to R
- 0 missionaries and 1 cannibals from R to L
- 2 missionaries and 0 cannibals from L to R
- 1 missionaries and 1 cannibals from R to L
- 2 missionaries and 0 cannibals from L to R
- 0 missionaries and 1 cannibals from R to L
- 0 missionaries and 2 cannibals from L to R
- 0 missionaries and 1 cannibals from R to L
- 0 missionaries and 2 cannibals from L to R

Bonus

Bonus

Nella repo GitHub troverete anche una funzione `plot_states` che vi permetterà di visualizzare graficamente la sequenza di stati.

Il codice richiede l'uso di `matplotlib` e `numpy`, che fino ad ora non abbiamo utilizzato. Questo codice è relativamente complicato, non è necessario imparare a fare questo tipo di grafici per l'esame.