Andrea Augello
Department of Engineering, University of Palermo, Italy

Ricerca non informata (continuazione)

Università degli Studi di Palermo

Ricerca non informata (continuazione)

La scorsa esercitazione abbiamo visto come sia possibile rappresentare un problema di intelligenza artificiale come una ricerca in un grafo, e come sia possibile risolvere questo problema utilizzando una ricerca in ampiezza.

La ricerca in ampiezza è un algoritmo completo, ovvero è garantito che, se esiste una soluzione, questa verrà trovata. Tuttavia, la ricerca in ampiezza non è sempre l'algoritmo migliore per risolvere un problema.

In questa esercitazione vedremo come sia possibile risolvere un problema utilizzando la ricerca in profondità.

Problemi di soddisfacimento di vincoli, ricerca in profondità

Problemi di soddisfacimento di vincoli, ricerca in profondità

I problemi di soddisfacimento di vincoli (CSP) sono un tipo di problema di ricerca.

Sebbene esistano algoritmi più sofisticati, è possibile risolvere un problema CSP utilizzando una ricerca in profondità con backtracking.

Problemi di soddisfacimento di vincoli, ricerca in profondità

I problemi di soddisfacimento di vincoli (CSP) sono un tipo di problema di ricerca.

Sebbene esistano algoritmi più sofisticati, è possibile risolvere un problema CSP utilizzando una ricerca in profondità con backtracking.

Nella maggior parte dei casi, in un problema CSP, lo stato corrente è una assegnazione parziale delle variabili, ovvero una assegnazione che non è ancora completa.

Inoltre, lo stato finale è ciò che ci interessa trovare, non è quindi noto a priori e non importa tenere traccia del cammino.

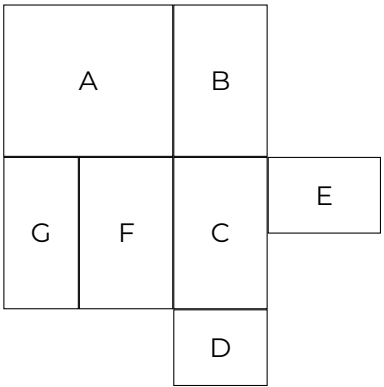
Esempio: il teorema dei quattro colori

Il teorema afferma che, data una superficie piana divisa in regioni connesse, come ad esempio una carta geografica politica, sono sufficienti quattro colori per colorare ogni regione facendo in modo che regioni adiacenti non abbiano lo stesso colore.

Questo teorema è stato uno dei primi ad essere dimostrato utilizzando un computer.

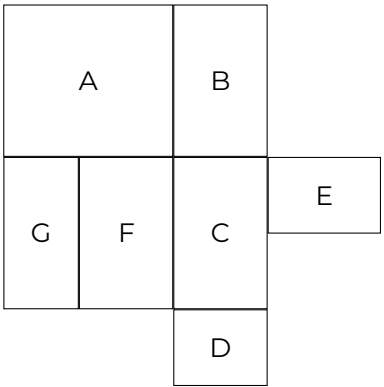
Esempio: il teorema dei quattro colori

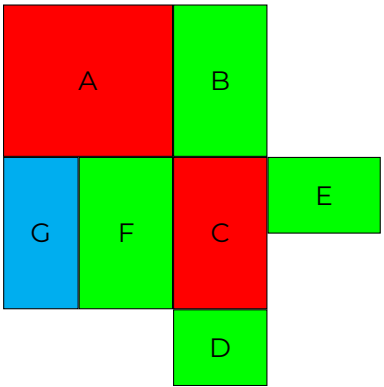
Possibile colorazione di una data mappa con quattro colori.



Esempio: il teorema dei quattro colori

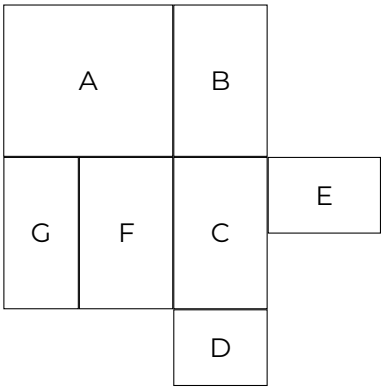
Possibile colorazione di una data mappa con quattro colori.

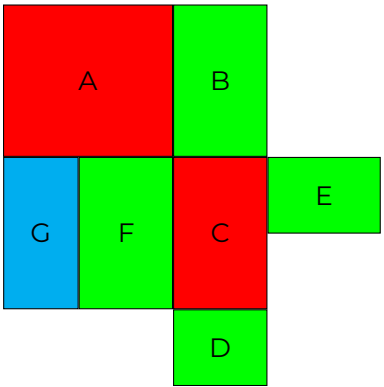




Esempio: il teorema dei quattro colori

Possibile colorazione di una data mappa con quattro colori.






```
{'A': 'red', 'B': 'green', 'C': 'red', 'D': 'green', 'E': 'green', 'F': 'green', 'G': 'blue'}
```

Modellazione del problema

Memorizziamo con un dizionario di liste di adiacenza il grafo che rappresenta le regioni e le loro adiacenze.
Inoltre, memorizziamo in una lista i colori disponibili.

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

colors = ['red','green','blue','yellow']
```

Problemi di soddisfacimento di vincoli, ricerca in profondità

- **Stato iniziale**: nessuna variabile assegnata
- **Stato finale**: assegnazione completa delle variabili
- **Stato corrente**: assegnazione parziale delle variabili
- **Operatore**: assegnare un valore ad una variabile
- **Cammino**: sequenza di assegnazioni che porta allo stato corrente, ininfluente
- **Costo del cammino**: costante

Problemi di soddisfacimento di vincoli, ricerca in profondità

```
class DFSAgent:
    def __init__(self, colors, adjacency):
        self.colors = colors
        self.adjacency = adjacency

    def valid_coloring(self, state):
        pass #verifica che l'assegnamento parziale non violi i vincoli

    def next_moves(self, state):
        pass #genera le possibili assegnazioni successive

    def dfs(self, state={}):
        pass #implementa la ricerca in profondità
```

Controllo dei vincoli e generazione degli stati successivi

Controllo dei vincoli:

```
    def valid_coloring(self, state):
        for node in state:
            for neighbor in self.adjacency[node]:
                if neighbor in state and state[node] ==
                    state[neighbor]:
                    return False
        return True
```

Controllo dei vincoli e generazione degli stati successivi

Controllo dei vincoli:

```
    def valid_coloring(self, state):
        for node in state:
            for neighbor in self.adjacency[node]:
                if neighbor in state and state[node] ==
                    state[neighbor]:
                    return False
        return True
```

Nuove assegnazioni possibili:

```
    def next_moves(self, state):
        for node in self.adjacency:
            if node not in state:
                for color in self.colors:
                    yield (node,color)
```

Ricerca in profondità

```
    def dfs(self, state={}):
        if len(state) == len(self.adjacency):
            yield state
        for node, color in self.next_moves(state):
            new_state = state.copy()
            new_state[node] = color
            if self.valid_coloring(new_state):
                yield from self.dfs(state=new_state)
        return None
```

```
{'A': 'red', 'B': 'green', 'C': 'red', 'D': 'green', 'E': 'green',
    'F': 'green', 'G': 'blue'}
```

Ricerca a costo uniforme

Ricerca a costo uniforme

La ricerca a costo uniforme è un algoritmo completo che, a differenza della ricerca in ampiezza, non assume che tutti i costi dei cammini siano uguali e può quindi risultare più versatile.

Il funzionamento è simile a quello della ricerca in ampiezza, ma invece di ordinare la frontiera in ordine di profondità, la frontiera viene ordinata in ordine di costo del cammino.

Problema dello zaino modificato

Problema dello zaino modificato

Il problema dello zaino modificato è un problema di ottimizzazione che consiste nel riempire uno zaino con oggetti di diverso peso e valore, in modo da minimizzare il peso totale e garantire che il valore totale sia maggiore di una soglia prefissata.

Problema dello zaino modificato

Il problema dello zaino modificato è un problema di ottimizzazione che consiste nel riempire uno zaino con oggetti di diverso peso e valore, in modo da minimizzare il peso totale e garantire che il valore totale sia maggiore di una soglia prefissata.

Più formalmente: dato un insieme di oggetti O, con $o_i \in O$ avente peso w_i e valore v_i, e una soglia V, il problema consiste nel trovare un sottoinsieme $S \subseteq O$ che risolva il seguente problema di ottimizzazione:

$$\begin{aligned} \underset{S}{\operatorname{argmin}} \quad & \sum_{o_i \in S} w_i \\ \text{s.t.} \quad & \sum_{o_i \in S} v_i \geq V \end{aligned}$$

Problema dello zaino modificato

Istanza del problema: Vogliamo andare a fare una escursione in montagna e vogliamo portare con noi il minor peso possibile, ma vogliamo essere sicuri di avere con noi il necessario per sopravvivere.

Materiale a disposizione codificato come dizionario di tuple (peso, valore):

```
items = {
            'bottle': (1, 600),
            'binoculars': (2, 900),
            'map': (2, 150),
            'compass': (1, 200)
        }
```

Problema dello zaino modificato

```
class UCSAgent():
    def __init__(self, minimum_value, items):
        self.minimum_value = minimum_value
        self.items = items
        self.frontier = [set()]

    def state_value(self, state):
        pass

    def state_weight(self, state):
        pass

    def search(self):
        pass
```

Algoritmo di ricerca

1. Inizializziamo la frontiera con l'insieme vuoto
2. Estraiamo lo stato S con costo minimo dalla frontiera
3. Se S soddisfa il vincolo, restituisci S
4. Altrimenti, generiamo tutti gli stati successivi a S e li aggiungiamo alla frontiera
5. Torniamo al passo 2

Prossimi passi

Prossimi passi

Come potremmo modificare l'algoritmo di ricerca a costo uniforme se nel problema dello zaino non ci interessasse minimizzare il peso totale, ma massimizzare il valore totale restando entro una soglia di peso?

$$\begin{aligned} \underset{S}{\text{argmax}} \quad & \sum_{o_i \in S} v_i \\ \text{s.t.} \quad & \sum_{o_i \in S} w_i \leq W \end{aligned}$$

Indizio

1. Inizializziamo la frontiera con l'insieme vuoto
2. Estraiamo lo stato S con valore minimo dalla frontiera
3. Generiamo tutti gli stati successivi a S e li aggiungiamo alla frontiera se rispettano il vincolo
4. Se la frontiera è vuota, restituiamo S
5. Torniamo al passo 2

L'ultimo stato rimanente nella frontiera è quello con valore massimo tra tutti quegli stati che rispettano il vincolo, e non è più possibile generare stati successivi che rispettino il vincolo.