Andrea Augello
Department of Engineering, University of Palermo, Italy

Ricerca in profondità e problemi di soddisfacimento dei vincoli

Università degli Studi di Palermo

Ricerca in profondità e problemi di soddisfacimento dei vincoli

Le scorse 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 vari algoritmi di ricerca.

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.

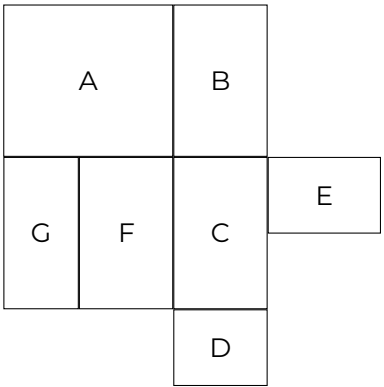
Esempio: il teorema dei quattro colori

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.

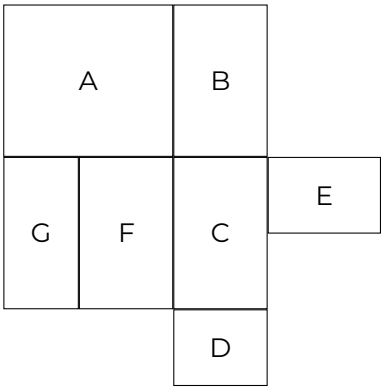
Esempio: il teorema dei quattro colori

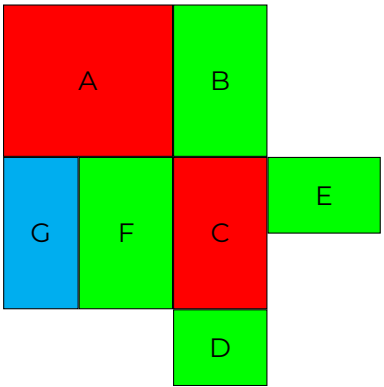
Possibile colorazione di una data mappa con quattro colori.



Esempio: il teorema dei quattro colori

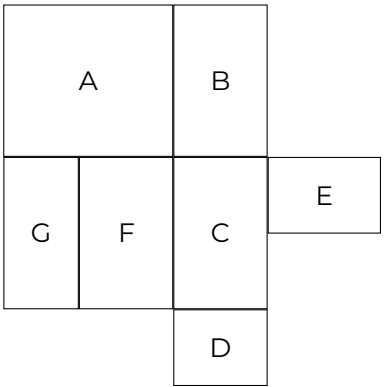
Possibile colorazione di una data mappa con quattro colori.

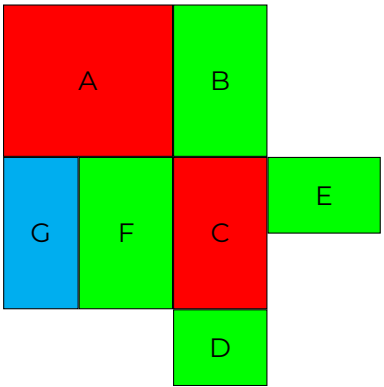




Esempio: il teorema dei quattro colori

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
        self.state = {}

    def valid_coloring(self, state, new_node=None):
        pass #verifica che l'assegnamento parziale non violi i
            vincoli. Specificando new_node, verifica solo i vincoli
            del nodo new_node

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

    def dfs(self):
        pass #implementa la ricerca in profondità
```

Controllo dei vincoli e generazione degli stati successivi

Controllo dei vincoli:

```
    def valid_coloring(self, state, new_node=None):
        for node in state if new_node is None else [new_node]:
            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, new_node=None):
        for node in state if new_node is None else [new_node]:
            for neighbor in self.adjacency[node]:
                if neighbor in state and state[node] ==
                    state[neighbor]:
                    return False
        return True
```

Il controllo dei vincoli può essere limitato ai soli vicini del nodo appena assegnato assumendo che tutti i nodi precedenti siano già stati assegnati correttamente. In questo modo evitiamo controlli ridondanti velocizzando l'esecuzione.

Controllo dei vincoli e generazione degli stati successivi

Nuove assegnazioni possibili:

```
    def next_states(self, state):
        for node in self.adjacency:
            if node not in state:
                for color in self.colors:
                    new_state = state.copy()
                    new_state[node] = color
                    if self.valid_coloring(new_state):
                        yield (node,color)
```

Controllo dei vincoli e generazione degli stati successivi

Alternativa equivalente

```
def next_states(self, state):
    return (
        new_state
        for node in self.adjacency if node not in state
        for color in self.colors
        if self.valid_coloring(new_state := {**state, node: color},
            new_node=node)
    )
```

Ricerca in profondità

```
    def dfs(self):
        if len(self.state) == len(self.adjacency):
            yield self.state
        for new_state in self.next_states(self.state):
            self.state = new_state
            yield from self.dfs()
```

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

Perché l’impelementazione ricorsiva?

L’implementazione ricorsiva ci permette di non dovere gestire manualmente la pila degli stati da esplorare, delegando questa responsabilità allo stack delle chiamate ricorsive.

La profondità massima della ricorsione è limitata dalla profondità dell’albero di ricerca, e non dallo spazio disponibile in memoria perché la memoria utilizzata per le chiamata che espande un nodo viene liberata appena si torna al nodo padre.

Inoltre, l’implementazione ricorsiva è più compatta e leggibile rispetto a quella iterativa.

Provare per credere: implementazione iterativa

```
from itertools import chain
class DFSAgentIterative(DFSAgent):
    def __init__(self, colors, adjacency):
        super().__init__(colors, adjacency)
        self.frontier = (s for s in [{}])

    def dfs(self):
        while (state := next(self.frontier),None)!= None:
            if len(state) == len(self.adjacency):
                yield state
            self.frontier =
                chain.from_iterable([self.next_states(state),
                self.frontier])
```

Le chiamate successive a `chain.from_iterable()` consentono di concatenare più generatori in un'unica sequenza, tuttavia occupano spazio aggiuntivo nello stack delle chiamate per ogni nodo esplorato (potenziale segmentation fault per alberi grandi).

Leggermente più leggibile, rinunciando a yield

```
class DFSAgentIterative2(DFSAgent):
    def __init__(self, colors, adjacency):
        super().__init__(colors, adjacency)
        self.frontier = [{}, None]

    def dfs(self):
        while (state := self.frontier.pop(0))!= None:
            if len(state) == len(self.adjacency):
                yield state
            self.frontier[0:0] = list(self.next_states(state))
```

Per evitare di saturare lo stack siamo costretti ad espandere i nodi da mettere nella frontiera, non possiamo fare lazy evaluation.

Soddisfacimento di vincoli meno triviali

L’approccio della ricerca in profondità con backtracking è molto semplice da implementare, però è quasi equivalente ad una ricerca esaustiva. Possiamo fare di meglio?

Con un grafo più grande (14 nodi), la ricerca in profondità con backtracking impiega circa 11 secondi per trovare una soluzione (sul mio pc), con 30 nodi il tempo di esecuzione è oltre i 5 minuti.

Soddisfacimento di vincoli meno triviali

Le ottimizzazioni che vedremo sono:

- Non ha senso testare ordini diversi di assegnazione delle variabili, se un ordine fallisce, falliranno anche tutte le permutazioni, meglio testare solo un ordine.
- Forward checking: se qualche variabile non ha più valori assegnabili, è inutile continuare ad assegnare le variabili rimanenti.
- Most constrained variable: scegliere come prossima variabile da assegnare quella con il minor numero di valori assegnabili. Questa è la variabile che ha più probabilità di causare un fallimento, meglio scoprire subito se c'è un problema.
- Least constraining value: scegliere come prossimo valore da assegnare quello che limita meno le scelte future. Questo permette di evitare di bloccarsi in vicoli ciechi.

Scheletro della classe

```
class DFSAgentOptim(DFSAgent):
    def __init__(self, colors, adjacency, mrw=False, lcv=False, fwc=False):
        super().__init__(colors, adjacency)
        self.fwc, self.lcv, self.mrw = fwc, lcv, mrw

    def forward_checking(self, state):
        return True

    def MCV_sort(self, state):
        return [n for n in self.adjacency if n not in state][0]

    def LCV_sort(self, state, node):
        return self.colors

    def next_states(self, state):
        node = self.MCV_sort(state) #Prima ottimizzazione, ordine fisso
        return ( new_state
                 for color in self.LCV_sort(state, node :=
                     self.MCV_sort(state)[0])
                 if self.valid_coloring(new_state := {**state, node: color},
                     new_node=node) and self.forward_checking(new_state))
```

Forward checking: come controllare i valori rimanenti?

Step 1: funzione di utilità che, dato uno stato ed una variabile, restituisce il numero di valori rimanenti assegnabili.

Forward checking: come controllare i valori rimanenti?

Step 1: funzione di utilità che, dato uno stato ed una variabile, restituisce il numero di valori rimanenti assegnabili.

```
def legal_values(self, state, node):
    return len(self.colors) - len({state[neighbor] for neighbor in
        self.adjacency[node] if neighbor in state})
```

Forward checking: come controllare i valori rimanenti?

Step 1: funzione di utilità che, dato uno stato ed una variabile, restituisce il numero di valori rimanenti assegnabili.

```
def legal_values(self, state, node):
    return len(self.colors) - len({state[neighbor] for neighbor in
        self.adjacency[node] if neighbor in state})
```

Step 2: restituire `True` solo se tutte le variabili non assegnate hanno almeno un valore assegnabile.

Forward checking: come controllare i valori rimanenti?

Step 1: funzione di utilità che, dato uno stato ed una variabile, restituisce il numero di valori rimanenti assegnabili.

```
def legal_values(self, state, node):
    return len(self.colors) - len({state[neighbor] for neighbor in
        self.adjacency[node] if neighbor in state})
```

Step 2: restituire `True` solo se tutte le variabili non assegnate hanno almeno un valore assegnabile.

```
def forward_checking(self, state):
    return all(self.legal_values(state, node) > 0 for node in
        self.adjacency if node not in state) if self.fwc else True
```

Most Constrained Variable

La stessa funzione di utilità `legal_values` può essere utilizzata come criterio per trovare la variabile non assegnata con il numero minimo di valori assegnabili. Possiamo usare il metodo `min` di Python con il parametro opzionale `key` per trovare il valore che minimizza un criterio arbitrario.

Most Constrained Variable

La stessa funzione di utilità `legal_values` può essere utilizzata come criterio per trovare la variabile non assegnata con il numero minimo di valori assegnabili. Possiamo usare il metodo `min` di Python con il parametro opzionale `key` per trovare il valore che minimizza un criterio arbitrario.

```
def MCV_sort(self, state):
    return min([n for n in self.adjacency if n not in state],
        key=lambda n: self.legal_values(state, n) if self.mrw else 0)
```

Least Constraining Value

Per implementare la scelta del valore che limita meno le scelte future, per prima cosa ci serve una funzione di utilità che conta quante opzioni rimuove l'assegnazione di un valore ad una variabile.

Least Constraining Value

Per implementare la scelta del valore che limita meno le scelte future, per prima cosa ci serve una funzione di utilità che conta quante opzioni rimuove l'assegnazione di un valore ad una variabile.

```
def removed_neighbors_colors(self, state, node, color):
    neighbors = [ n for n in self.adjacency[node] if n not in state]
    return sum(self.legal_values(state, n) for n in neighbors) -
        sum(self.legal_values({**state, node: color}, n) for n in
        neighbors)
```

Least Constraining Value

Per implementare la scelta del valore che limita meno le scelte future, per prima cosa ci serve una funzione di utilità che conta quante opzioni rimuove l'assegnazione di un valore ad una variabile.

```
def removed_neighbors_colors(self, state, node, color):
    neighbors = [ n for n in self.adjacency[node] if n not in state]
    return sum(self.legal_values(state, n) for n in neighbors) -
        sum(self.legal_values({**state, node: color}, n) for n in
        neighbors)
```

Fatto ciò, possiamo utilizzare il metodo `sorted` di Python per ordinare i colori in ordine crescente di quanti valori rimuovono se assegnati al nodo dato.

```
def LCV_sort(self, state, node):
        return sorted(self.colors, key=lambda c:
            self.removed_neighbors_colors(state, node, c) if self.lcv
            else 0)
```

Soddisfacimento di vincoli meno triviali

Queste ottimizzazioni sono delle ottime euristiche per ridurre il numero di assegnazioni fallimentari. A seconda della complessità delle verifiche da fare i vantaggi non sempre potrebbero giustificare l'overhead computazionale aggiuntivo.

Esistono anche altre tecniche come la propagazione dei vincoli, ma queste sono più complesse da implementare e non le vedremo dal punto di vista pratico.