Uninformed Search

Esposito
Marcello
N46006315

19/07/2024

—

Elementi di IA

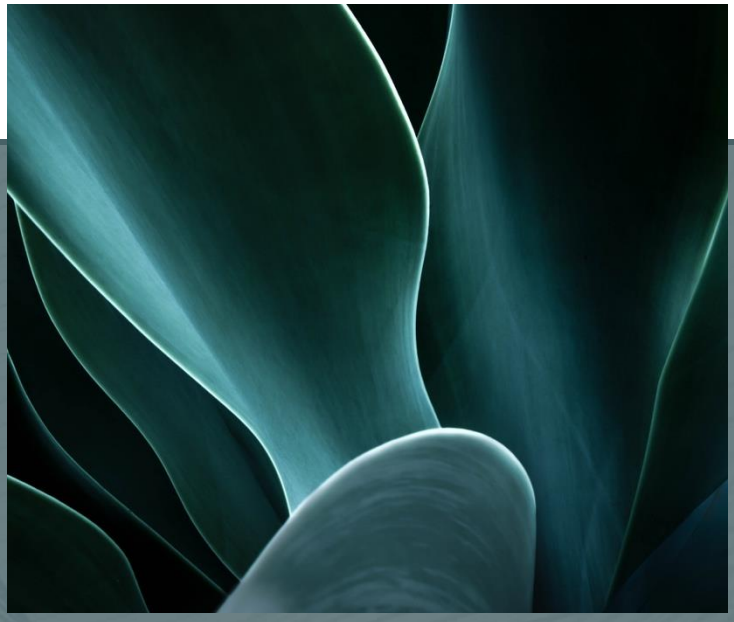
—

Giancarlo Sperlì

OBIETTIVI

Implementare gli algoritmi di ricerca non informata,
in maniera indipendente dagli specifici dataset per poterli utilizzare anche in contesti reali.

Verificare sperimentalmente le prestazioni teoriche attese di tali algoritmi



RICERCA RAMIFICATA

PROBLEM SOLVING & SEARCH

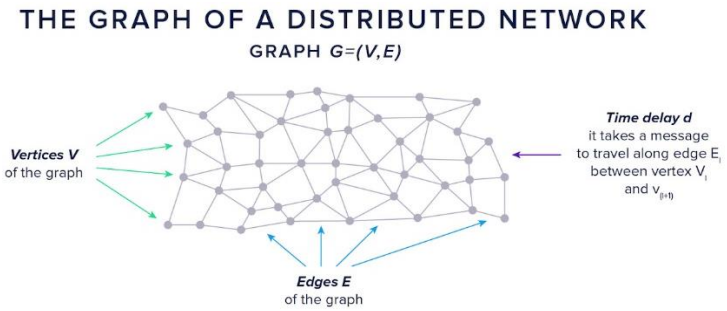
Per problem solving & search si intende l'insieme di algoritmi ed approcci atti a far sì che un agente possa raggiungere il suo obiettivo.

In questo caso andremo ad affrontare una particolare tipologia di problemi, detti single state problem dove questi ultimi risultano essere deterministici e completamente osservabili;

tale tipologia di problemi richiede di ricercare una sequenza di azioni che permetta all'agente di passare da un determinato stato iniziale ad uno stato finale desiderato

RAPPRESENTAZIONE

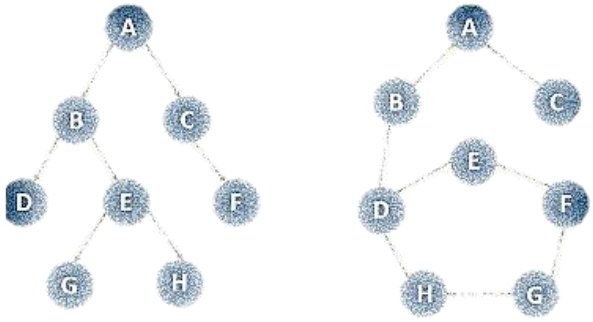
Uno degli approcci più diffusi per affrontare tali problemi è quello di rappresentare lo spazio degli stati attraverso un grafo, dove ogni nodo rappresenta uno stato e gli archi le azioni che possiamo intraprendere per passare ad un altro stato;
tale struttura a grafo ci permette di rappresentare chiaramente l'evoluzione degli stati ed è facilmente scalabile per rappresentare contesti più complessi, come la rappresentazione di azioni con costo non omogeneo



1 i grafi vengono utilizzati in numerosi contesti, rendendo gli algoritmi di ricerca particolarmente importanti

ALBERO

A partire da un grafo, è possibile mappare un'apposita struttura ad albero, che risulta essere particolarmente vantaggiosa per il nostro scopo;
Ogni nodo dell'albero incapsula il concetto di stato, ma possiede inoltre diverse informazioni utili agli algoritmi di ricerca, come i nodi figli, il nodo padre, la profondità…



STRATEGIA

Il meccanismo alla base della ricerca sugli alberi è il concetto di frontiera, che è l'insieme dei nodi che separa i nodi esplorati da quelli che dobbiamo ancora esplorare;
tale frontiera può essere esplorata utilizzando delle vere e proprie **strategie** differenti, che possono portare a risultati anche molto differenti tra di loro.
In generale avremo che gli algoritmi utilizzano il seguente pattern:

Inizializza **frontiera**
Loop:
If (**frontiera** vuota) FAIL

Seleziona **Nodo** dalla frontiera secondo la **strategia**

If (**Nodo** è l'obiettivo) return **Sequenza** azioni
Else espandi **Nodo** ed estendi **frontiera** secondo la **strategia**

Quindi come si può notare la scelta della strategia risulta essere di fondamentale importanza, e per poter effettuare tale scelta è necessario utilizzare delle apposite metriche di performance, che sono le seguenti:

Completezza	se viene sempre trovata una soluzione
Ottimità	la soluzione trovata è sempre ottimale
Time complexity	numero di nodi espansi
Space complexity	numero di nodi max da mantenere in memoria contemporaneamente

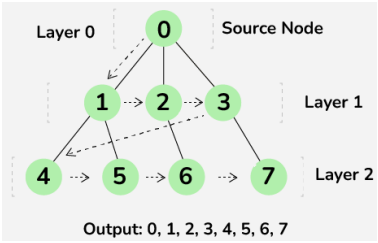
le ultime 2 utilizzano a loro volta: max branch factor ***b***, optimal depth ***d***, max three depth ***m***

ALGORITMI

Date le ipotesi effettuate, discusse brevemente nel paragrafo successivo, è stato di mio interesse andare a valutare, con le apposite metriche, i seguenti algoritmi:

- **Breadth First Search (BFS)**

La strategia utilizzata da tale algoritmo è quella di espandere l'albero in orizzontale, ovvero espandendo prima i nodi con uno stesso livello di profondità per poi proseguire con i nodi di livello inferiore

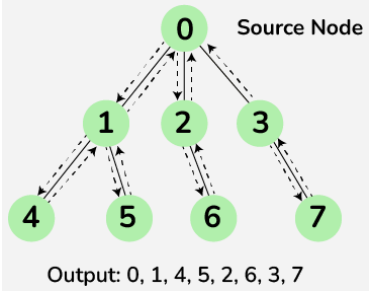


Completezza	sì, **iff** b non è ∞
Ottimità	sì, **iff** costo degli step è uniforme
Time complexity	O (b^{d+1})
Space complexity	O (b^{d+1})

Il problema principale di tale algoritmo è la space complexity, in caso d sia elevato, anche se con il progressivo abbassamento dei costi delle memorie diventa sempre meno importante

- **Depth First Search (DFS)**

La strategia utilizzata da tale algoritmo è quella di espandere l'albero in verticale, ovvero espandendo prima i nodi con una profondità maggiore per poi passare ad i rami adiacenti



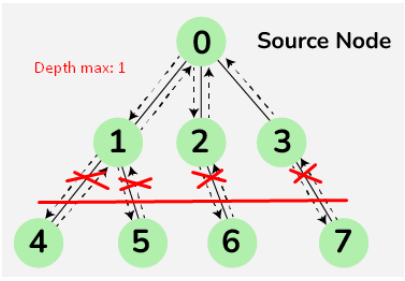
Completezza	sì, **iff** m è finito e si applicano meccanismi anti loop
Ottimità	no
Time complexity	O (b^{m})
Space complexity	O $(b * m)$

Adesso la space complexity ha un'equazione lineare, al costo però di avere una time complexity decisamente peggiore

Il vero vantaggio dell'algoritmo DFS, è che quest'ultimo in realtà può fungere da base per degli algoritmi più avanzati che ci permettono di andare ad attenuare le problematiche principali in termini di time complexity, completezza e ottimità

- **Limited Depth First Search (LDFS)**

La strategia utilizzata da tale algoritmo è quella di sfruttare l'algoritmo DFS, ma adesso verrà imposto un livello di profondità massima esplorabile ***l***, oltre il quale l'albero verrà tagliato (cutoff)

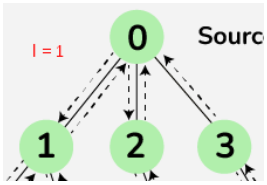


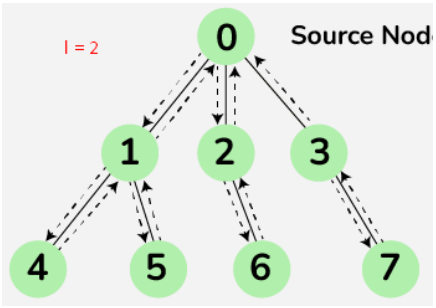
Completezza	sì, **iff** $l \geq d$
Ottimità	no
Time complexity	O (b^l)
Space complexity	O $(b * l)$

Ora la time complexity sarà relativa alla profondità massima esplorabile ed inoltre non abbiamo più problemi in caso m sia ∞, ma c'è il rischio concreto di non trovare una soluzione con una scelta errata di l, dato che in genere non siamo a conoscenza di d

- **Iterative Deepening Search (IDS)**

La strategia utilizzata da tale algoritmo è quella di combinare un approccio iterativo con la limited depth first search, dove ad ogni iterazione verrà aumentata la profondità massima esplorabile fin quando non verrà trovata la soluzione o non verrà esplorato l'intero albero





Completezza	sì
Ottimità	sì, **iff** costo degli step è uniforme
Time complexity	O (b^d)
Space complexity	O $(b * d)$

Ora siamo riusciti a combinare i vantaggi della breadth first search con quelli della depth first search, senza dover conoscere d fin dall'inizio, a patto di poter iterare d volte l'algoritmo

IMPLEMENTAZIONE

IPOTESI UTILIZZATE

Per evitare di innescare loop infiniti e semplificare l'implementazione degli algoritmi, ho supposto che i dataset siano composti da grafi direzionati con costo degli step omogenei, e che i nodi del grafo siano univoci (nel caso in cui ci siano duplicati viene preso il primo che appare nel dataset)

GRAFO

Per la creazione del grafo è stata utilizzata una versione rivisitata della classe Nodo vista durante l'esercitazione

```python
class Node:
    def __init__(self, value=None):
        self.__depth: int = 0
        self.__children = []
        self.__value = value
        self.__parent: Node = None

    def add_parent(self, parent):
        self.__parent = parent

    def add_child(self, child):
        # we need this check due to duplicate nodes,
        if child.__depth == 0:
            child.__depth = self.__depth + 1
            self.__children.append(child)

    def get_value(self):
       return self.__value

    def get_children(self):
       return self.__children

    def get_sequence(self):

        if self.__parent is None:
            return f"{self.__value} -> "

        path = self.__parent.get_sequence()
        return  path + f"{self.__value} -> "

    def get_depth(self) -> int:
        return self.__depth
```

```python
def make_graph(filename) -> Node:
    nodes = {}

    with open(filename, 'r') as file:
        for line in file:
            if line.startswith('#'):
                continue  # Ignore comment line

            parts = line.split()
            from_node = int(parts[0])
            to_node = int(parts[1])

            if from_node not in nodes:
                nodes[from_node] = Node(from_node)

            if to_node not in nodes:
                nodes[to_node] = Node(to_node)
                nodes[to_node].add_parent( nodes[from_node] )

            nodes[from_node].add_child(nodes[to_node])


    return nodes[0] # return only the first node
```

ALGORITMI

Gli algoritmi sono implementati cercando di seguire lo schema generale della ricerca negli alberi,
gestendo la frontiera tramite una deque, che permette di realizzare una coda FIFO per la BFS e LIFO per la DFS.

(inoltre sono stati aggiunti dei meccanismi che ci permettono di valutare le prestazioni dei vari algoritmi)

Mentre il LDFS e l'IDS sono realizzate utilizzando la DFS 'in loop'

```python
def breadth_first_search(first_node: Node, value_to_find):
    node_viewed = set()
    fringe = deque([first_node])
    space_complexity = 0

    while fringe:
        space_complexity = max(space_complexity, len(fringe))

        to_check_node = fringe.popleft() # use as FIFO queue
        node_viewed.add(to_check_node)

        if to_check_node.get_value() == value_to_find:
            print_result(to_check_node, node_viewed, space_complexity)
            return res_found


        for child in __expand(to_check_node):
            if child not in node_viewed:
                fringe.append(child)

    print_fail(value_to_find)
    return res_fail


def __expand(node: Node) -> list[Node]:
    return node.get_children()
```

RISULTATI

DATASET

I risultati mostrati sono basati sul dataset dei prodotti Amazon acquistati insieme ad altri prodotti, ma ovviamente si possono ottenere risultati analoghi in termini di prestazioni anche su altri dataset

BFS

```
Found node: 189182
Sequence found: 0 -> 4 -> 17 -> 31 -> 85 -> 170
Nodes expanded: 91713
Max nodes in memory: 13277
```

DFS

```
Found node: 189182
Node's depth: 24
Sequence found: 0 -> 4 -> 17 -> 31 -
Nodes expanded: 88394
Max nodes in memory: 57
```

LDFS

```
search with l >= d:
Found node: 189182
Node's depth: 24
Sequence found: 0 -> 4 -> 17 -> 31 -> 85 -> 170 -> 217 -
Nodes expanded: 72295
Max nodes in memory: 47

search with l < d
Node 189182 not found in the graph.
Nodes expanded: 3286
fatal cutoff
```

IDS

```
Node 189182 not found in the graph using max depth: 23
Found node: 189182
Node's depth: 24
Sequence found: 0 -> 4 -> 17 -> 31 -> 85 -> 170 -> 217 -
Nodes expanded: 47571
Max nodes in memory: 41
```

ANALISI (dettagli da notare)

- Differenza space complexity tra BFS e algoritmi basati su DFS notevole
- Time complexity tra BFS e DFS simile, dato che $d \approx m$
- LDFS con l non corretta non trova una soluzione
- Netto miglioramento time complexity tra DFS e le sue versioni avanzate, in particolar modo la IDS ci permette di non dover conoscere la d, e trovare la l ottimale

MATERIALE

Repository GitHub: https://github.com/Moriarty2002/elementi-di-IA