Contents

(a te malcapitato, un avviso prima di leggere, non so la punteggiatura, non so il LaTeX, e non so scrivere, auguri)

1 Albero di Ricerca (Search Tree)

non è una struttura dati che si vuole mettere in memoria tende a essere CHONKER

- la **radice** è lo **stato iniziale**
- i **figli** di uno stato sono gli **stati raggiungibili** da quello stato

è possibile che ci siano cammini multipli che portano a uno stesso stato

la struttura dati, essendo tutti i possibili cammini fattibili, è infinita, visto che un cammino può andare avanti all'infinito

ogni nodo rappresenta un cammino, se si arriva a uno stato con due cammini diversi, non è lo stesso nodo, visto che i due nodi rappresentano due cammini diversi quindi sì, è effettivametne un albero, non abbiamo cicli

1.1 Espansione

l'espansione di un nodo consiste nella generazione di tutti i suoi possibili successori la creazione dell'insieme dei possibili successori

l'espansione è un'operazione che espande il singolo nodo espandere un nodo nell'albero di ricerca o espandere lo stato del nodo boh (non ho capito cosa cazzo ha detto)

1.2 Frontiera

ma tu non lo udisti e il tempo passava
con le stagioni, a passo di Java
ed arrivasti a passar la frontiera
in un bel giorno di primavera
La Guerra Di Piero – Il Cazzo che Me Frega

la frontiera è l'inseme dei nodi dell'albero che sono stati espansi e corrispondono alle foglie dell'albero parziale generato esplorando l'albero di ricerca la frontiera è un'insieme dinamico

1.2.1 Interfaccia della frontiera

la frontiera è una struttura astratta, questa è definita dall'implementare le seguenti funzioni

```
public interface Frontier<T> {
    public boolean empty();
    public T pop();
    public void insert(T element);
}
```

(che sia chiaro, non mi ricordo il java)

comunque si farà una coda con priorità, pop prende quello con priorità maggiore

1.3 Ok ma nodi già visti

ai fini di non fare loop infiniti si fa un set?

per fare tipo dijkstra però c'è bisogno di indicizzare, proprio parte dell'algoritmo

1.3.1 Nomi del cazzo

open list frontiera

closed list "set" di nodi visitati

1.4 RTFS

Read The Fucking Source
DETTO POPOLARE

puoi andare su github a cercare AIMA[1] Code

1.4.1 Implementatio

1. Node como cazzo è fatto il nodo

```
struct Node {
    struct Node parent; // da dove sono venuto
    Action action; // come ci sono arrivato
    State state; // in che stato è
    double path_cost; // quanto mi è costato arrivarci
    unsigned int depth;
};
typedef struct Node Node;
```

2. Problem serve una classe `Problem`, per descrivere il problema questa descrive il problema, quindi

   - lo stato iniziale
   - le actions
   - lo stato goal...

```
abstract class Problem {
    public State getInitialState();
    public boolean isGoalState(State s);
    public List<Actions> actionsFrom(State s);
}
```

3. Frontiera ricorda che la frontiera è una priority queue ordinata secondo una funzione `f`

[1] Artificial Intelligence, a Modern Approach

1.4.2 Pythonus

```
# yields rende, bene o male, un iteratore
class Problem:
    def __init__(self):
        self.initial_state = cazzo_ne_so

    def expand_node(self, node):
        yield qualcosa

    def is_goal_state(self, state):


def best_first(problem, sorting_function):
    node = Node(problem.initial_state)
    frontier = Queue(sorting_function=sorting_function)
    frontier.insert(node)
    costs = {problem.initial_state : node}
    # questo dovrebbe essere un dictionary in python
    while not q.empty():
        n = q.pop()
        if problem.is_goal_state(n.state):
            return n # tutto il nodo, ergo tutto il cammino
        for child in problem.expand_node(n):
            s = child.state
            if not s in costs or child.path_cost < costs[s].path_cost:
                costs[s] = child
                q.insert(child)

    # arrivati qui abbiamo esaurito la frontiera
    # ma se siamo usiciti così dal loop vuol dire che
    # non abbiamo comunque trovato un cazzo
    # quindi
    return None
```

per il

```
s = child.state
if not s in costs or child.path_cost < costs[s].path_cost:
    costs[s] = child
    q.insert(child)
```

`or child.path_cost < costs[s].path_cost` vuol dire che se l'ho già visto, ma l'ho già visto con un costo più alto, aggiorno il costo visto che posso arrivarci con un costo più basso. Ho una soluzoine migliore, quindi usiamola

1.4.3 Esempii al variare di `sorting_function`

questo è un'algoritmo generico, il comportamento di questo dipende da `sorting_function`, al variare di questa l'algoritmo agisce in modi diversi

alcune `sorting_function` possibili sono

- `f(n) = -n.depth`
- `f(n) = n.depth`
- `f(n) = n.path_cost`

i comportamenti risultanti sono

`-n.depth` più è profondo, meno pesa, quindi vado sempre a fondo, ne esce una DFS

`n.depth` più è profondo, più pesa, quindi vado sempre in largo, ne esce una BFS

`n.path_cost` più mi costa, meno lo voglio, ne esce Dijkstra

il Dijkstra che ne esce è noto anche come `uniform_cost`

quste bellezze sono note anche come *blind search algorithms*, questo blind vuol dire che non so un cazzo del problema, quindi sto andando in giro alla cazzo più totale se gli dico di trovare la strada per roma, proverà anche a vedere se passare per via bolognese, e ci perderà tempo

2 Paragone di algoritmi boh

	complete	optimal	time	space
DFS	NOPE	NOPE	$O(b^m))$	$O(bm)$
BFS	YES	DIPENDE	$O(b^d))$	$O(b^d))$
UC	YES	YES	$O(b^{1+\lfloor \frac{c}{\varepsilon} \rfloor})$	idem $O(b^{1+\lfloor \frac{c}{\varepsilon} \rfloor})$

- con **d** si indica la **profondità** del goal **più superficiale** che si trova nell'albero di ricerca il goal più superficiale in generale non nececcariamente quello ottimo, solo quello con meno archi presente nell'albero

- **b** è il **branching factor**, più o meno quanti nodi generi quando fai `expand`
- **m** è la **massima lunghezza** di un camminio semplice[2] nel **GRAFO** di ricerca (non l'albero, il coso che ho visitato, se "togli la direzionalità" degli archi)

completo vuol dire che un algoritmo riesce a trovare la soluzione ogni volta che parte da un punto da cui esiste una soluzione

2.1 DFS

2.1.1 Incompleta

una dfs brutale potrebbe anche girare all'infinito da un cammino inutile e infinto con un set di `trovati` la dfs, anche brutale, funziona quando il set degli stati è finito, se il goal è raggiungibile se il set degli stati è infinito la DFS se la prende in culo, incompleta

tutti sono completi con set di stati finiti ma abbiamo tanti di quei set infiniti che diocane

2.1.2 Non Ottima

anche su alberi finiti. facciamo finta che la dfs prenda sempre prima il sottoalbero "più a sinistra" si può garantire che trova il goal più a sinistra, ma un goal più a destra potrebbe tranquillamente essere stato più vicino

2.1.3 Tempo

il caso peggiore è aver visto TUTTO quindi vedi prima a sinistra, ma il risultato era il duce

2.2 BFS

la bfs è ottima solo nel caso di path cost unifore in quanto trova sempre il cammino con meno archi

[2] senza cicli e senza tornare sugli stessi nodi

2.3 UC/Dijkstra

2.3.1 Ottimo

La frontiera separa ciò che è stato visto da ciò che non è stato visto. Qualsiasi nodo che non è stato visto, per essere visto, peffoza deve passare per la frontiera, questa è nota come la *proprietà di separazione* della frontiera qualsiasi cammino che ... deve tagliare la frontiera, questo è quello che il coromen non ti dice, scopri questo semplice trucco per la dimostrazione.

2.3.2 Tempo

ecco la bestia di satana

$$O(b^{1+\lfloor \frac{c}{\varepsilon} \rfloor})$$

- la ε è un tecnicismo matematico, per garantire che ogni soluzione ha un camminino di lunghezza finita è un tecnicismo matematico per dire **nessuno step ha costo inferiore a** ε, quindi non abbiamo cammini che sono cammini infiniti di passi infinitesimali

2.4 Analisi in spazio

quando lo spazio ha complessità esponenziale so' cazzi il tempo è un limite di pazienza la ram COL CAZZO che è un limite pazienza