Intelligenza Artificiale e Machine Learning

Appunti delle Lezioni di Intelligenza Artificiale e Machine Learning
Anno Accademico: 2024/25

Giacomo Sturm

Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche
Università degli Studi "Roma Tre"

Sorgente del file LaTeX disponibile al seguente link:
https://github.com/00Darxk/Intelligenza-Artificiale-e-Machine-Learning

Indice

1 Introduzione all'Intelligenza Artificiale

L'intelligenza artificiale è l'area di studio che analizza come permettere ai computer di compiere operazioni che eseguite da un essere umano richiederebbero intelligenza. Rappresenta uno dei campi di ricerca più interessati recentemente ed ha avuto un'enorme crescita dal punto di vista economico e tecnologico.

Un modo per poter riepilogare la storia dell'intelligenza artificiale consiste nell'elencare i vincitori del Turing Award, in questo campo. Questo premio rappresenta l'analogo del premio Nobel per l'informatica.

1.1 Storia

La prima fase di questa storia si può attribuire a personaggi come Alan Turing, che introdusse il concetto di "test di Turing" per determinare se una macchina si può considerare intelligente nel 1950. Una macchina secondo questo criterio si può considerare intelligente se un essere umano interagendo con essa, senza saperlo a priori, non sia in grado di sapere se si tratta di una macchina o di un altro essere umano.

Un altro di questi personaggi fondamentali è John McCarthy, ha coniato il termine "intelligenza artificiale", nel 1955 ed ha proposto la creazione di un gruppo di lavoro l'anno successivo al collage di Dartmouth. Al convegno di Dartmouth del '56 parteciparono tutti i personaggi che vengono chiamati i padri fondatori dell'IA. L'obiettivo del convegno era lo studio della costruzione di macchine in grado di formare astrazioni e concetti, per risolvere problemi, al tempo riservati agli umani. In questo convegno si teorizzò che queste macchine potranno svolgere funzioni umane, ritenute intelligenti.

In questo convegno Cliff Shaw, Allen Newell e Herbert Simon dimostrarono il primo programma di intelligenza artificiale, chiamato "Logic Theorist", in grado di dimostrare i teoremi dei Principia Mathematica. Proposero l'idea di "General Problem Solver" per risolvere una grande varietà di problemi simulando i processi mentali umani. Questo sistema riusciva a dimostrare teoremi, calcolare funzioni matematica e risolvere problemi logici. Da sistemi come questi si crearono grandi aspettative per l'intelligenza artificiale, in questa seconda fase tra il 1952 ed il 1969. In questo periodo Nathaniel Rochester all'IBM sviluppò tra i primi programmi di IA. Nel 1959 Herbert Gelernter scrisse il "Geometry Theorem Prover", capace di dimostrare teoremi matematici complessi. John McCarty nel 1958 all'MIT definì il linguaggio di lato livello Lisp, destinato a diventare il linguaggio di programmazione per eccellenza dell'IA per i seguenti trent'anni. Nel 1963 fondò il laboratorio dell'IA a Stanford, per costruire una versione definita dell'"Advice Taker".

Nonostante queste grandi aspettative, i sistemi di IA realizzati per problemi semplici non dimostravano un comportamento altrettanto soddisfacente per problemi di natura più complessa. Uno dei problemi per questo mancato successo dell'IA in questo periodo tra il '66 ed il '73 fu l'incapacità di comprendere l'intrattabilità di molti problemi, su cui l'IA stava cercando di operare. Nel 1973 venne stilato il rapporto Lighthill, dove venne criticata l'intelligenza artificiale per la mancata soluzione al problema dell'"esplosione combinatoria", ovvero sull'utilizzo dell'IA per problemi di natura reale.

Un altra critica giunse dal libro "Perceptrons" scritto nel 1969 da Marin Minsky e Seymour Papert che, come altri, discusse il limite fondamentale della struttura di base per generare un comportamento intelligente. In questo periodo il clima generale attorno all'IA era pessimista e libri come questo aiutarono a maturare questo clima di diffidenza nei confronti delle potenzialità dell'IA.

Negli anni successivi per risolvere il problema dell'esplosione combinatoria si realizzarono sistemi basati sulla conoscenza, riservati ad esperti umani del settore. Questi programmi erano estremamente utili su settori molto ristretti, costituiti da un motore di inferenza. Il collo di bottiglia per questi sistemi era la conoscenza, per cui esistevano intere posizioni dedicate a trovare un modo per rappresentare ed astrarre il dominio di interesse su cui opera l'IA. Queste persone venivano chiamate scienziati della conoscenza, ed erano tipicamente professionisti ed esperti di quel settore specifico.

Nel 1981 il governo giapponese avviò un progetto chiamato "Quinta Generazione" per realizzare computer intelligenti utilizzando il linguaggio Prolog. Analogamente sia gli Stati Uniti che l'Inghilterra finanziarono progetti simili.

Tuttavia questi progetti non riuscirono a mantenere le loro promesse sulle capacità dell'IA o di impatto commerciale. I sistemi erano comunque limitati da una conoscenza limitata ed una difficile capacità di apprendere dall'esperienza e verificare la correttezza. Erano sistemi poco flessibili e robusti rispetto agli obiettivi promessi da questi progetti.

Nonostante questo tra l'inizio e la fine degli anni '80 l'industria dell'IA conobbe un boom economico con centinaia di aziende costruttrici di sistemi esperti, di visione artificiale, di robot, di software ed hardware specializzati per questi scopi.

Molte di queste aziende fallirono per l'impossibilità di mantenere le loro promesse e seguì un periodo chiamato "inverno dell'IA'.

A metà degli anni '80 almeno quattro gruppi diversi reinventarono l'algoritmo di apprendimento basato sulla retropropagazione, sviluppato negli anni '60. Questi modelli furono considerati in diretta opposizione ai modelli simbolici di Newell e Simon e dell'approccio logicista di McCarthy. Geoff Hinton è una delle figure in primo piano nel risorgere delle reti neurali durante gli anni '80 e 2010, descrisse i simboli come l'"etere luminifero dell'IA".

Dai limiti dei sistemi esperti si introdusse un nuovo approccio basato sulla probabilità invece che sulla logica booleana, sul "Machine Learning" più che sulla programmazione manuale, su risultati sperimentali più che su affermazioni filosofiche. Questo permette a questi sistemi con logiche non cablate di poter apprendere dall'esperienza e migliorarsi nel tempo.

Nei primi anni del 2000 con lo sviluppo e l'espansione del "World Wide Web" ed il progresso sulla potenza di calcolo ha permesso di realizzare data set molto grandi, fenomeno indicato con il termine "big data". Questo ha portato allo sviluppo di algoritmi di machine learning progettati per trarre vantaggio da questi grandi insiemi di dati. La loro disponibilità ed il cambiamento di approccio verso il ML ha permesso all'IA di recuperare appetibilità commerciale.

Nel 2006 Geoffrey Hinton, introduce un algoritmo di apprendimento veloce per reti neurali, dando il via alla rivoluzione del "Deep Learning". Utilizzando molteplici livelli di elementi computazionali il ML diventa Deep Learning. Questo sistema ha ottenuto successi in molti domini applicativi, aumentando l'interesse verso l'IA.

Attualmente l'IA è in grado di lavorare in molti settori:

- Può guidare veicoli robotizzati come automobili o droni autonomi. Nel 2004 DARPA introduce una sfida per la guida autonoma di veicoli;
- Permette di eseguire locomozione su arti con robot umanoidi;
- Pianificazione e scheduling autonomo, per la navigazione autonoma ed il sistema GPS globale;
- Traduzione automatica;
- Riconoscimento vocale;
- Raccomandazioni su social network per gli utenti. Nel 2006 Netflix ha lanciato una borsa per aumentare la precisione del loro sistema di raccomandazione tramite ML, vinta nel 2009;
- Può battere i migliori giocatori umani in molti giochi. Nel 1997 Deep Blue batte il campione del mondo di scacchi Garry Kasparov, e nel 2011 IBM Watson batte i campioni del gioco "Jeopardy!", ora questo sistema è utilizzato in molti sistemi. Tra il 2015 ed il 2017 AlphaGo di DeepMind batte i campioni del mondo nel gioco di Go, lo stesso team ha sviluppato AlphaZero in grado di giocare sia scacchi, shogi e Go;
- Può interpretare e riconoscere immagini. Nel 2021 ImageNet comincia il suo concorso annuale per rilevare e classificare correttamente oggetti in un set di immagini ampio e ben accurato, con un incremento nella precisione dovuto ai progressi nelle "deep convolutional neural networks". Nel 2014 Facebook pubblica un lavoro sul DeepFace in grado di identificare volti con un'accuratezza del 97%;
- Permette di diagnosticare malattie, raggiungendo o superando la diagnosi di medici esperti;
- Recentemente ha accelerato la ricerca per il vaccino anti-Covid, nell'individuazione delle proteine utilizzate;
- Permette di rilevare informazioni dettagliate su eventi climatici estremi.

Nel 2014 il consumo di energia per raffreddare i Data Center è stato ridotto del 40% con un modello di ML, analogamente nel 2017 il software per analizzare le immagini di galassie sotto lenti gravitazionali è stato velocizzato di un fattore di 10^7.

Nel 2022 venne rilasciato ChatGPT, "Generative Pre-trained Transformer", il chatbot di OpenAI progettato per rendere efficace la comunicazione con un utente umano.

2 Risoluzione dei Problemi e Ricerca

Si definisce un agente risolutore di problemi un agente con uno specifico obiettivo da raggiungere e che deve identificare una sequenza di azioni per raggiungerlo.

Bisogna determinare l'obiettivo, un insieme degli stati del mondo dove ci si trova, che si vuole raggiungere. Inoltre bisogna formulare il problema, ovvero le azioni e gli stati considerati dall'agente.

Un agente che ha a disposizioni diverse opzioni immediate di valore sconosciuto, può decidere quale scegliere la sua azione analizzando le diverse possibili sequenze di azioni, che portano a stati di valore conosciuto, per scegliere la sequenza di costo migliore.

Questo processo di selezione viene definito ricerca, un algoritmo di ricerca quindi prende un problema e restituisce una soluzione costituita da una sequenza di azioni.

Nella formulazione si definisce uno stato iniziale e dell'obiettivo, come un insieme di stati e si definiscono le azioni come transizioni tra stati. Dopo aver trovato la sequenza di azioni corrispondente alla soluzione, la esegue.

Si possono distinguere due tipi di problemi, i problemi giocattolo o "toy problems", sono ideati come illustrazione o esercitazione dei metodi risolutivi. Rappresentano delle astrazioni, anche semplificate, dei problemi del mondo reale in generale più difficili a cui si è effettivamente interessati.

I problemi del mondo reale possono essere la configurazione VLSI, la navigazione dei robot, la sequenza di montaggio, la ricerca dell'itinerario e generali problemi di viaggio come il commesso viaggiatore.

Si possono utilizzare due tipi diversi di formulazione di un problema. Si può partire da uno stato vuoto, ed utilizzando operatori si può estendere progressivamente la descrizione dello stato. Invece nella formulazione a stato completo consiste nel partire da uno stato iniziale completo, ed ogni operatore altera questo stato per cercare una soluzione.

Lo spazio degli stati è un grafo che rappresenta tutti i possibili stati, come nodi, collegati tra archi che rappresentano le possibili azioni. Il problema consiste nel trovare un percorso in questo spazio degli stati, dallo stato iniziale ad uno dei possibili stati soluzione. L'algoritmo deve decidere ad ogni stato quale azione prendere e quindi a quale nodo del grafo spostarsi. Per definire il costo della soluzione, si considera il costo del cammino delle azioni intraprese dall'agente.

Per definire formalmente un problema, sono necessarie quattro componenti. Uno stato iniziale in cui si trova l'agente. Una descrizione delle azioni possibili, questa può utilizzare una funzione successore che dato uno stato restituisce i suoi possibili successori. Tramite la funzione successore e lo stato iniziale si può costruire lo spazio degli stati. Oppure si può utilizzare un insieme di operatori. Un test obiettivo per determinare se un particolare stato è uno stato obiettivo. Come ultimo componente è necessaria una funzione di costo del cammino per determinare il costo di una data soluzione, assegnando un valore numerico ad ogni cammino.

Il tipo di dato problema è rappresentato da questi quattro componenti, le istanze di questo tipo di dato rappresentano gli input degli algoritmi di ricerca.

Il mondo reale è estremamente complesso, e lo spazio degli stati deve essere creato mediante un processo di astrazione. In questo processo, lo stato astratto rappresenta un insieme di stati reali più complessi. Analogamente per le azioni astratte, queste rappresentano combinazioni di azioni reali. Nei problemi giocattolo non è necessario effettuare questo processo di astrazione, poiché rappresenta un problema semplice.

Per individuare queste possibili sequenze di azioni l'algoritmo si può costruire un albero di ricerca, dove il nodo radice corrisponde allo stato iniziale, ed i rami rappresentano le azioni possibili, ed i nodi figli rappresentano gli stati successori di un certo stato. Tuttavia i nodi dell'albero di ricerca e gli stati dello spazio degli stati sono differenti, poiché è possibile che più nodi condividano lo stesso stato, mentre ogni stato nello spazio è univoco. Generalmente si vuole evitare di ripetere stati all'interno di un cammino, questo rappresenta il problema degli stati ripetuti, e sarà analizzato successivamente.

Ad ogni nodo si possono inserire altre informazioni utili, oltre allo stato, un riferimento al genitore, l'operatore che ha generato lo stato, la profondità, il costo del cammino parziale fino a questo stato, etc.

```
NODO = <stato, genitore, operatore, profondità, costo parziale, ...>
```

Il processo di ricerca comporta la stessa sequenza di azioni. Deve scegliere tra le foglie dell'albero corrente un nodo da "espandere", secondo un certo criterio o strategia. In seguito bisogna determinare se questo nodo rappresenta un'obiettivo del problema, altrimenti vengono generati i suoi nodi figli, ed i corrispondenti stati successori, e tutte le componenti dei nodi figli. La collezione dei nodi in attesa di essere espansi viene chiamata in vari modi: confine, frontiera, frangia o lista aperta.

2.1 Algoritmo di Ricerca Generale: Tree Search

Un algoritmo generale di ricerca può essere chiamato `TREE-SEARCH`. Quest'algoritmo prende un problema ed una strategia come input e restituisce una soluzione oppure un fallimento. Viene realizzato semplicemente ad un ciclo che ripete le operazioni precedentemente descritte, fino a quando non identifica una soluzione o viene sollevato un problema, e quindi restituisce un fallimento. Il primo passo è la generazione dell'albero di ricerca del problema, se la frontiera è vuota, ovvero non esistono nodi candidati per l'espansione viene riportato un fallimento, poiché non è stato ancora trovato uno stato obiettivo. Se si arriva ad un nodo corrispondente ad uno stato obiettivo viene restituita rappresenta la sequenza di nodi ottenuta come soluzione.

La frontiera può contenere un nodo relativo ad uno stato obiettivo, ma l'algoritmo non termina fino a quando non viene scelto per essere espanso.

Una strategia di ricerca rappresenta un criterio per decidere quale nodo da espandere. Può essere definita come una funzione per la scelta di un elemento tra un insieme di nodi, la frontiera. Oppure si può considerare come una funzione di inserimento di un elemento in una sequenza. Se già nella fase di inserimento si analizza tramite una metrica il valore di ognuno di questi stati, allora l'elemento in prima posizione in questa struttura dati rappresenta il nodo migliore.

2.1.1 Operazioni su Frontiera, Nodi e Problemi

La frontiera viene implementata con una struttura dati chiamata coda, ma non necessariamente segue la disciplina FIFO. Su questa coda sono definite una serie di operazioni:

- `MAKE-QUEUE`: prende come input un nodo `n` e restituisce una coda `q`, realizza una coda contenente solo il nodo `n`;

- `EMPTY`: prende come input una coda `q` e restituisce un booleano, verifica se la coda è vuota;
- `REMOVE-FRONT`: prende come input una coda `q` e restituisce il primo nodo `n` della lista;
- `QUEUING-FN`: prende come input una coda `q` ed una lista di nodi `n`, e restituisce la coda con aggiunti tutti questi nodi.

L'ultima funzione si utilizza quando si producono tutti i nodi successori e si vogliono aggiungere alla lista. Questa funzione non inserisce generalmente in coda, ma dipende dalla strategia di ricerca utilizzata.

Assumendo che esistano le operazioni sul tipo di dato problema, e sul tipo di dato nodo. Le operazioni sul tipo di dato problema sono:

- `INITIAL-STATE`: prende come input un problema `p` e restituisce lo stato iniziale del problema `n`;
- `GOAL-TEST`: prende come input un problema `p` ed uno stato `n` e verifica se questo rappresenta una soluzione, restituendo un booleano;
- `OPERATORS`: prende come input un problema `p` e restituisce una lista con tutti gli operatori del problema `ops`. Ogni operatore `op` applicato ad uno stato `n` restituisce una lista di stati `ss`.

Le operazioni sul tipo di dato nodo sono:

- `MAKE-NODE`: prende come input uno stato `s` e costruisce un nodo su di esso `n`;
- `STATE`: prende come input un nodo `n` e ne restituisce lo stato contenuto `s`;
- `EXPAND`: prende come input un nodo `n` ed una lista di operatori `ops` e restituisce una lista di nodi successori `ns`.

2.1.2 Implementazione

Date queste operazioni, si può rappresentare in pseudocodice l'algoritmo di `TREE-SEARCH` in modo più semplice:

```
function TREE-SEARCH(problem) returns a solution or failure

fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop do
    if EMPTY(fringe) then return failure
    node <- REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    fringe <- QUEUING-FN(fringe, EXPAND(node, OPERATOR(problem)))
end
```

In questa variazione non viene conservato l'intero albero, ma solamente la coda con i nodi della frontiera.

2.2 Criteri di Valutazione

Per valutare questi algoritmi oltre alla complessità temporale e spaziale, si utilizzano altre due criteri, la completezza e l'ottimalità. Un'algoritmo si definisce completo, se quando esiste una soluzione è garantito sia in grado di trovarla. Un algoritmo si dice ottimo se dato un problema con diverse soluzioni, individua sempre la migliore, quella a costo minimo. La complessità dell'algoritmo dipende dal fattore di ramificazione b dello spazio degli stati e dalla profondità d della soluzione più superficiale. Il fattore di ramificazione b rappresenta il massimo numero di figli che un nodo può avere. Mentre la profondità d è la minima lunghezza di un cammino dal nodo iniziale alla radice.

2.3 Ricerca non Informata o Cieca

Questi algoritmi non sono molto efficienti in generale, ma sono utili per comprendere il comportamento gli algoritmi di ricerca informata o di euristica, che si avvalgono della conoscenza sul domino dello spazio degli stati e dalla creazione dell'albero di ricerca per scegliere il percorso più promettente. Nel caso medio quest'ultimi sono certamente più efficienti degli algoritmi trattati in questa sezione.

2.3.1 Algotimo di Ricerca in Ampiezza: Breadth First Search (BFS)

Nella ricerca in ampiezza si espande il nodo radice, e si espandono i nodi generati dalla radice, e si ripete per ogni nodo successore. Per implementare un algoritmo che utilizza una strategia di ricerca non informata in ampiezza, la "Queuing Function" inserisce i nodi appena generati in coda. Questa funzione quindi rappresenta sempre un inserimento in coda e si può chiamare "Enqueue at the End": `ENQUEUE-AT-END`.

Un algoritmo che utilizza questo tipo di strategia viene chiamato "Breadth First Search" o in ampiezza, e si può implementare in modo analogo all'algoritmo di ricerca generale trattato precedentemente:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution or failure

fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop do
    if EMPTY(fringe) then return failure
    node <- REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    fringe <- ENQUEUE-AT-END(fringe, EXPAND(node, OPERATOR(problem)))
end
```

In questo approccio, tutti i nodi di profondità d vengono espansi prima dei nodi di profondità $d+1$. Rappresenta una strategia sistematica, ma permette di individuare solamente i nodi obiettivi più superficiali, non è garantito che questo rappresenta la soluzione ottima. Questo algoritmo è quindi completo, ma non è ottimo. Invece è ottimale se il costo del cammino $g(n)$ è una funzione monotona non decrescente della profondità del nodo $p(n)$:

$$p(n) < p(n) \implies g(n) \leq g(m)$$

$$p(n) = p(m) \implies g(n) = g(m)$$

Ovvero se due nodi n ed m sono a profondità diverse, dove il nodo m è a profondità maggiore, il costo del cammino dalla radice al nodo n è al massimo uguale al costo del cammino dalla radice al nodo m. Inoltre se i nodi sono alla stessa profondità, allora i costi dei loro cammini dalla radice sono uguali.

Utilizzando questo algoritmo, bisogna generare un numero di nodi, prima di trovare una soluzione, almeno pari a tutti i nodi precedenti al nodo soluzione. Nel caso peggiore questo nodo è l'ultimo nodo espanso alla profondità d, e per ogni nodo vengono generati esattamente b figli, quindi bisogna espandere al massimo un numero di nodi N pari a:

$$N = \left(\sum_{i=1}^{d+1} b^i \right) - b$$

Supponendo che ogni generazione rappresenta un'operazione semplice allora la complessità temporale di questa ricerca è $O(N) = O(b^d)$. La complessità temporale è analogamente $O(b^d)$ poiché bisogna memorizzare tutte le foglie generate.

2.3.2 Algoritmo di Ricerca Guidata dal Costo: Dijkstra

Modificando la ricerca in ampiezza espandendo il nodo della frontiera di costo più basso, si può aumentare l'efficienza dell'algoritmo precedente.

Si definisce con $g(n)$ il costo del cammino dalla radice al nodo n. Questo valore viene salvato nella struttura dati nodo, e viene scelto il nodo di costo $g(n)$ minore per essere espanso. Se il costo del cammino corrisponde alla funzione di profondità, si ha la ricerca in ampiezza. Questo algoritmo è completo e ottimale quando il costo di ogni step è sempre maggiore o uguale ad una costante positiva ε. Per cui è garantito che non attraversi costantemente lo stesso cammino, senza espandere altri nodi di profondità minore.

I costi di ogni nodo vengono salvati in un campo etichetta nella struttura dati nodo. Dalla frontiera si estrae sempre il nodo a costo minore, questa collezione viene quindi ordinata in base al costo delle etichette di ogni nodo.

2.3.3 Algoritmo di Ricerca in Profondità: Depth First Search (DFS)

La ricerca in profondità consiste nell'espansione del nodo più profondo, dopo aver espanso la radice. Per implementare questa funzione, si utilizza una queuing function che inserisce i nodi appena espansi all'inizio della lista, con una "Enqueue at the Front": `ENQUEUE-AT-FRONT`:

```
function DEPTH-FIRST-SEARCH(problem) returns a solution or failure

fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop do
    if EMPTY(fringe) then return failure
    node <- REMOVE-FRONT(fringe)
```

```
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    fringe <- ENQUEUE-AT-FRONT(fringe, EXPAND(node, OPERATOR(problem)))
end
```

Questa funzione si può implementare mediante una funzione ricorsiva, dove viene passata una versione del problema, dove i nodi appena generati rappresentano i nuovi nodi radice. Quindi per ogni espansione vengono generati al massimo b sotto-problemi, risolti dallo stesso algoritmo. La lista dei nodi da visitare viene conservata implicitamente nello stack dei record di attivazione delle varie chiamate ricorsive. Quando si raggiunge un nodo foglia non obiettivo, si effettua il backtracking, ovvero si risale l'albero fino a trovare il nodo a profondità maggiore non ancora espanso su cui è possibile effettuare una scelta.

Questa ricerca non è né completa né ottimale, ha una complessità temporale di $O(b^m)$, dove m rappresenta la profondità massima dell'albero di ricerca. Se una soluzione è presente a profondità minore nel sotto-albero di destra, non verrà mai individuata se non è stato già espanso tutto il sotto-albero di sinistra, senza aver trovato una soluzione. Quindi se individua una soluzione la restituisce indipendentemente dalla sua ottimalità.

Si guadagna rispetto alla ricerca in ampiezza nella complessità spaziale. Infatti non bisogna memorizzare l'intero albero, ma solamente il cammino dalla radice alla foglia, ed i fratelli non espansi di ciascun nodo del cammino di profondità m: $O(b \cdot m)$. Se l'albero ha rami infiniti, allora la ricerca non termina.

2.3.4 Algoritmo di Ricerca in Profondità Limitata

Nella ricerca in profondità limitata si impone un limite alla profondità massima, per impedire di proseguire all'infinito su uno stesso cammino. Un nodo viene espanso solo se la lunghezza del cammino dalla radice al nodo è minore del massimo stabilito. Se non viene trovata alcuna soluzione restituisce il valore speciale taglio se alcuni nodi non sono stati espansi, altrimenti fallisce.

Si possono utilizzare conoscenze specifiche al problema per fissare questo limite. Se si lavora su di un grafo si potrebbe utilizzare il diametro del grafo come la profondità. Se non si sceglie un valore adeguato per questo limite allora l'algoritmo non funzionerà correttamente. L'algoritmo è completo, se la soluzione è ad una profondità minore della lunghezza l imposta, mentre non è ottimale. La complessità temporale e spaziale è rispettivamente $O(b^l)$ e $O(b \cdot l)$. Risolve il problema della completezza, ma non risolve l'ottimale.

2.3.5 Algoritmo di Ricerca Iterative-Deepening Search

Questo algoritmo risolve il problema dell'ottimalità sugli algoritmo di ricerca in profondità senza conoscere un limite adeguato. Evita il problema della scelta del limite provando iterativamente tutti i limiti possibili fino a quando non individua una soluzione:

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns solution or failure

for depth = 0 to ∞   do
    if DEPTH-LIMITED-SEARCH(problem, depth) succeeds
```

```
        then return ist result
    end
```

Questo approccio combina i benefici di una ricerca in ampiezza con i benefici di una ricerca in profondità, poiché per ogni profondità vengono analizzati tutti i nodi.

Quindi questo algoritmo è ottimale e completo per le condizioni della ricerca in ampiezza. La complessità spaziale è $O(b \cdot d)$, quindi non è esponenziale. Mentre la complessità temporale è simile a quella a quella della ricerca in ampiezza. L'algoritmo viene richiamato ogni volta che si aumenta il limite, quindi i nodi a profondità minore vengono generati ogni volta che viene eseguito nuovamente l'algoritmo. Quindi vengono generati in totale N nodi:

$$N = \sum_{i=1}^{d} b^i \cdot (d + 1 - i)$$

I primi b nodi a profondità 1 vengono generati d volte, fino ai nodi al livello d generati una sola volta. Questo algoritmo è quindi circa l'11% meno efficiente rispetto alla ricerca in ampiezza. Ma non rappresenta un incremento considerevole rispetto alla ricerca in ampiezza, quindi è accettabile.

2.4 Problema della Ripetizione degli Stati: Graph-Search

Il problema della ripetizioni degli stati può provocare gravi complicazioni nel processo di ricerca. Questo problema sorge sopratutto quando sono possibili azioni bidirezionali ed in questo caso è possibile che gli alberi di ricerca siano infiniti. Si vuole quindi evitare quando è possibile di ripetere gli stessi stati in più nodi dell'albero di ricerca.

Questi stati ripetuti possono in certi casi rendere il problema irrisolvibile, è conveniente controllare se uno stato è replicato. Se un algoritmo arriva ad uno stesso stato attraverso due cammini differenti, allora ha individuato uno stato ripetuto e deve scartare uno di questi due cammini, per determinare quale scartare si sceglie generalmente l'ultimo cammino ottenuto. Si scarta anche se questo cammino è migliore del cammino precedente. Si utilizza questo approccio poiché negli algoritmi di ricerca euristica, sotto alcune condizioni, quando trova un percorso questo è ottimo, quindi non sorgono problemi nello scartare cammini che portano allo stesso stato.

In altre implementazioni dove non è garantito che il primo percorso trovato sia il migliore, bisogna controllare quale dei due cammini presenti il costo migliore. Per evitare la ripetizione bisogna contenere gli stati già visitati in memoria, tramite un'altra struttura dati chiamata insieme esplorato o lista chiusa, contenente ogni nodo espanso.

Si modifica l'algoritmo Tree Search nell'aggiunta alla frontiera per verificare la ripetizione degli stati:

```
function GRAPH-SEARCH(problem) returns a solution or failure

close  <- empty set
fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop do
    if EMPTY(fringe) then return failure
```

```
    node <- REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    if STATE(node) not in close
        then ADD(close, node)
        child_list <- EXPAND(node, OPERATOR(problem))
        for child_node in child_list
            if STATE(child_node) not in close then
                fringe <- QUEUING-FN(fringe, child_node)
end
```

Questo approccio si chiama Graph Search, dove prima di aggiungere un nodo alla frontiera, si controlla se il suo stato è già stato avvistato. Si suppone che il primo cammino che raggiunge nuo stato s è il più conveniente. Questo algoritmo realizza un albero direttamente sul grafo dello spazio degli stati, poiché è presente al massimo una singola copia di ogni stato. La frontiera separa nel grafo dello spazio degli stati in due regioni, una esplorata, ed una da esplorare. In questo modo ogni cammino dallo stato iniziale ad uno stato inesplorato deve passare attraverso uno stato sulla frontiera.

L'algoritmo scarta sempre il cammino appena trovato, se lo stato raggiunto è ripetuto, quindi potrebbe scartare un cammino corrispondente ad una soluzione migliore. Potrebbe quindi non essere un algoritmo ottimale.

Inoltre l'uso della lista chiusa significa che la ricerca in profondità e quella ad approfondimento iterativo non richiedono requisiti spaziali lineari.

2.5 Algoritmo di Ricerca Informata o Euristica: Best First Search

Quando si parla di ricerca euristica, l'algoritmo può sfruttare conoscenze specifiche sul problema in questione, aiutandolo nella scoperta della soluzione. Questa modifica si inserisce nella funzione di inserimento in coda QUEUING-FN. Questa conoscenza sul dominio del problema viene implementata tramite una funzione di valutazione f applicata ai nodi dell'albero di ricerca. Questa funzione stima quanto un nodo sia più o meno "promettente", ovvero stima della desiderabilità di espandere il nodo associato. Generalmente la funzione f è una funzione di stima del costo della soluzione, per si considera il nodo n appartenente alla frontiera con $f(n)$ minore.

L'algoritmo "Best First Search" ordina i nodi inseriti nella frontiera dal migliore al peggiore secondo una data funzione di valutazione f. Il nodo scelto da espandere è quello valutazione migliore, in genere con $f(n)$ minore, per cui a differenza di funzioni di valutazione f si hanno diverse versioni di questo algoritmo di ricerca.

La ricerca guidata dal costo (2.3.2)si può considerare un caso particolare della ricerca best first dove la funzione di valutazione $f(n)$ coincide con la funzione di costo del cammino $g(n)$ dallo stato iniziale al nodo n. Queste rappresenta un'informazione di euristica nulla.

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution or failure

QUEUING-FN <- function that orders nodes by EVAL-FN
fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
```

```
loop do
    if EMPTY(fringe) then return failure
    node <- REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    fringe <- QUEUING-FN(fringe, EXPAND(node, OPERATOR(problem)))
end
```

Dove `EVAL-FN` rappresenta la funzione di valutazione.
Per favorire la comprensione si utilizza una serie di notazioni:

- $s, s_0, s_1, \cdots, s_i$: stati del problema;
- s_0: stato iniziale;
- $k^*(s_i, s_j)$: costo del cammino minimo da s_i a s_j, se esiste;
- $g^*(s_i) = k^*(s_0, s_i)$: costo del cammino minimo dallo stato iniziale a s_i;
- $h^*(s_i)$: costo effettivo, non una stima, di un cammino da uno stato s_i ad uno stato obiettivo;
- $f^*(s_i) = g^*(s_i) + h^*(s_i)$: il costo minimo di una soluzione vincolata a passare per s_i.

Si introducono ulteriori notazioni per l'albero di ricerca:

- $n, n_1, n_2, \cdots, n_i$: nodi dell'albero di ricerca;
- $g*(n)$: costo del cammino dallo stato iniziale allo stato associato al nodo n;
- $h^*(n)$: costo effettivo di un cammino dallo stato associato al nodo n ad uno stato obiettivo;
- $g(n)$: costo del cammino dallo stato iniziale allo stato di n, $g(n) \leq g^*(n)$;
- $h(n)$: stima di $h^*(n)$.

Le funzioni senza asterisco all'apice si riferiscono a funzioni di stima, altrimenti sono funzioni di costi effettivi. La funzione h si dice funzione euristica.

2.5.1 Algoritmo di Ricerca Greedy

Questo algoritmo "goloso" utilizza la funzione h come funzione di valutazione, si espande quindi il primo nodo che si ritiene sia vicino all'obiettivo. Se n corrisponde allo stato obiettivo, deve essere $h(n) = 0$. Minimizza il costo stimato per raggiungere l'obiettivo.

Una possibile funzione euristica è la distanza a linea d'aria " Straight Line Distance" SLD: $h_{\text{SLD}}(n)$, per problemi di viaggio. Questo algoritmo non espande nodi inutilmente, ma la soluzione trovata dall'algoritmo potrebbe non essere la soluzione ottima del problema. In generale la ricerca golosa non è ottimale. Inoltre non è neanche completa, poiché se una soluzione richiedesse allontanarsi dall'obiettivo e quindi andare verso stati con una valutazione peggiore, non sarebbe mai scoperta da questo algoritmo.

Nel caso peggiore è anche esponenziale spazialmente e temporalmente con una complessità asintotica di $O(b^m)$. Nonostante questi risultati con una buona euristica è possibile ottenere buoni risultati anche con un algoritmo greedy. Uno dei difetti è che la scelta del nuovo stato è effettuata interamente dalla stima della distanza, senza considerare il percorso parziale.

2.5.2 Algoritmo A*

L'algoritmo A* risolve ip roble dell'algoritmo greedy, considerando anche il percorso parziale nella sua funzione di valutazione: $f(n) = g(n) + h(n)$, fornisce quindi una stima del costo del cammino dallo stato iniziale ad uno stato obiettivo, vincolato a passare per il nodo n.

Con questa semplice aggiunta è possibile migliorare notevolmente l'algoritmo precedente, inoltre sotto certe condizioni è possibile recuperare l'ottimalità e la completezza. Date queste due ipotesi sul grafo di partenza:

- Ogni nodo del grafo abbia un numero finito di successori;
- Tutti i costi abbiano costi maggiori di una quantità positiva δ.

Se h è un'euristica ammissibile, ovvero se per ogni nodo n del grafo vale la condizione $h(n) <\leq h^*(n)$, allora l'algoritmo A* è ottimale e completo:

$$\forall n \text{ t.c. } h(n) \leq h^*(n) \implies \text{A*: completo ed ottimale} \tag{2.5.1}$$

In generale per un dato problema è possibile identificare diverse funzioni di euristica che soddisfano la condizione di limite inferiore. Nei problemi della ricerca di itinerari, e non solo, un'altra possibile alternativa è la funzione di Manhattan che considera ogni mossa, o spostamento, sicuramente ammissibile. La ricerca guidata dal costo è un approccio molto particolare della ricerca A* $\lceil f(n) = g(n) \rceil$, dove la funzione di euristica h è nulla per ogni nodo, è quindi uno stimatore super ottimistico.

Date due versioni dell'algoritmo A* con due euristiche diverse $h_1 < h_2$, per tutti i nodi non obiettivo, si dice che l'algoritmo A*$_2$ è più informato dell'algoritmo A*$_1$. Vale allora il teorema secondo cui al termine delle loro ricerche su un qualsiasi grafo con un percorso dal nodo iniziale n_o al nodo obiettivo, allora ogni nodo espando da A*$_2$ sarà anche espando da A*$_1$. Quindi il primo algoritmo espande almeno tanti nodi quanto il secondo algoritmo, quindi l'algoritmo più informato A*$_2$ è più efficiente di A*$_1$.

In generale è più conveniente scegliere un'euristica per cui l'algoritmo è più informato. Per determinare una funzione euristica, un buon approccio consiste nel determinare la funzione di costo effettivo per un problema simile all'originale con minori restrizioni sugli operatori. Infatti il costo effettivo di una soluzione in questi "Relaxed Problems" è una buona euristica per il problema originale. Ma il calcolo della funzione euristica potrebbe avere un calcolo computazionale elevato, quindi bisogna scegliere l'euristica considerando anche la loro complessità.

L'algoritmo A* può essere eseguito sia in modalità Tree Search che Graph Search, ma in queste due modalità potrebbe trovare due soluzioni differenti allo stesso problema.

Si introduce quindi la condizione di consistenza, e si dice che una certa funzione euristica h obbedisce a questa condizione se per tutte le coppie di nodi n_j, successore di n_i nel grafo di ricerca

si ha:

$$h(n_i) \leq c(n_i, n_j) + h(n_j) \tag{2.5.2}$$

Dove $c(n_i, n_j)$ è il costo dell'arco che congiunge i due nodi. Analogamente:

$$h(n_j) \geq h(n_i) - c(n_i, n_j)$$

Ovvero su un qualsiasi percorso, la stima del costo ottimo non pul diminuire più del costo di un arco lungo quel percorso. Si può considerare anche come un tipo di diseguaglianza triangolare:

$$h(n_i) \leq c(n_i, n_j) + h_(n_j)$$

La condizione di consistenza impone che i valori della funzione di valutazione f dei nodi nell'albero di ricerca siano strettamente non-decrescenti all'allontanarsi dal nodo di partenza. Dati due nodi n_j, successore di n_i, se è soddisfatta si ha:

$$f(n_j) \geq f(n_i)$$

Dalla condizione di consistenza si aggiunge ad entrambi i membri $g(n_j) = g(n_i) + c(n_i, n_j)$:

$$\begin{gathered} h(n_j) \geq h(n_i) - c(n_i, n_j) \\ h(n_j) + g(n_j) \geq h(n_i) - c(n_i, n_j) + g(n_j) = h(n_i) - \cancel{c(n_i, n_j)} + g(n_i) + \cancel{c(n_i, n_j)} \\ f(n_j) \geq f(n_i) \end{gathered} \tag{2.5.3}$$

Per cui spesso la condizione di consistenza sulla funzione euristica h viene spesso chiamata condizione monotona su f.

Se la condizione di consistenza è soddisfatta su h, allora quando l'algoritmo A* espande un nodo n ha già trovato il percorso ottimo per n. In questo modo la ricerca su grafo non è differente dalla ricerca su un albero per quanto riguarda l'ottimalità della soluzione.

2.6 Algoritmo di Ricerca Locale: Hill-Climbing

Nei problemi precedenti quando l'algoritmo risolutivo raggiungeva uno stato obiettivo, il cammino verso quello stato rappresenta una soluzione del problema. Tuttavia in alcuni problemi lo stato obiettivo contiene tutte le informazioni rilevanti per la soluzione, dove il cammino è irrilevante. Come esempio si consideri il problema dell otto regine, è indifferente il cammino attraverso gli stadi intermedi, solamente lo stato finale, la disposizione delle regine nello stato finale.

Algoritmi di ricerca locale si utilizzano per risolvere questo tipi di problemi. In questi problemi è sempre presente uno spazio degli stati ed uno spazio degli stati aventi ciascuno una sua valutazione. Si può immaginare questi stadi su una superficie del territorio, uno spazio dove l'altezza di questo stato rappresenta la sua valutazione. L'algoritmo quindi itera su ognuno di questi stati per cercare quello di altezza maggiore, o minore, identificando quindi la soluzione al problema, indipendentemente dal cammino preso per raggiungerla. Questi punti di massimo rappresentano dei picchi, i cui punti adiacenti sono strettamente minori dello stato di massimo. Quindi l'algoritmo che parte da uno stato iniziale deve cercare un massimo globale in questo spazio, determinando quale sia tra

i vari massimi locali ed i massimi locali piatto, e le "spalle" massimi locali "piatti", prima di un massimo globale.

Questi algoritmi chiamati anche di miglioramento iterativo, si muovono sulla superficie cercando questi picchi, senza tenere traccia del cammino effettuato, tenendo solamente traccia dello stato attuale e dei suoi vicini o successori, gli stati immediatamente adiacenti. Bisogna formulare il problema in modo che l'algoritmo non rimanga bloccato tra due massimi locali.

Questo algoritmo segue sempre le colline più ripide, si muove sempre verso l'alto nella direzione dei valori crescenti, e termina quando raggiunge uno stato per il quale si ha un picco che non ha vicino stati di valore maggiore. Tuttavia questo algoritmo può rimanere intrappolato su massimi globali.

Non viene memorizzato lo stato corrente, solamente il valore attraverso nodi che contengono lo stato ed il suo valore.

Esistono diversi tipi di algoritmi di questo genere per evitare di rimanere bloccati su picchi locali, utilizzando diverse tecniche.

- Steepest Ascent Hill-Climbing;
- First-Choice Hill-Climbing;
- Random-Restart Hill-Climbing (Iterated Hill-Climbing);
- Stochastic Hill-Climbing.

2.6.1 Algoritmo Steepest Ascent Hill-Climbing

Si considera il seguente pseudocodice dell'algoritmo Hill-Climbing di tipo "Steepest Ascent":

```
function HILL-CLIMBING(problem) returns a state local max

current <- MAKE-NODE(INITIAL-STATE(problem))
loop do
    next <- MAX(EXPAND(current))
    if VALUE(next) < VALUE(current) then return STATE(current)
    current <- next
end
```

Dallo stato corrente si ricava il suo valore, in seguito comincia un ciclo che prende in considerazione tutti i successori e si sceglie come `next` il nodo di valore più alto. Se tutti i nodi adiacenti hanno un valore minore di `next` allora questo rappresenta la soluzione dell'algoritmo e l'algoritmo termina, tuttavia questo stato potrebbe corrispondere ad un massimo locale, invece se esiste uno stato adiacente di valore maggiore, questo diventa `next` e si passa alla nuova iterazione.

2.6.2 Algoritmo Random-Restart-Hill Climbing

L'algoritmo contiene una componente di ripartenza casuale, questo infatti conduce una serie di ricerche di Hill-Climbing partendo da stati generati casualmente. Questo algoritmo da un punto di

vista teorico è completo, poiché con una serie infinita di ripartenze, sicuramente l'algoritmo visita tutti gli stati del sistema, trovando sicuramente la soluzione ottima del problema.

```
function RANDOM-RESTART-HILL-CLIMBING(problem) returns a state solution

t <- 0
best <- MAKE-NODE(NULL)

repeat
    local <- false
    current <- RANDOM(problem)
    repeat
        next <- MAX(EXPAND(current))
        if VALUE(next) > VALUE(current)
            then current <- next
        else local <- true
    until local
    t <- t + 1
    if VALUE(current) > VALUE(best)
        then best <- current
until t = MAX
return STATE(best)
end
```

Il ciclo interno ad ogni iterazione genera un ottimo locale, e prova ad evitare ottimi locali effettuando una nuova ricerca da un nuovo stato scelto casualmente. L'algoritmo è completo con probabilità tendente ad uno, poiché è possibile generi come stato iniziale uno stato obiettivo.

2.6.3 Algoritmo Stochastic Hill-Climbing

L'algoritmo Stochastic Hill-Climbing si ottiene modificando la procedura normale dell'algoritmo. Invece di valutare tutti i vicini dallo stato corrente, l'algoritmo sceglie casualmente uno solo dei suoi successori da valutare per determinare se si tratta il successore, ed in caso diventa il nuovo stato corrente `next`, questo viene accetta con una probabilità che dipende dalla differenza della valutazione tra i due punti: $\Delta E =$`VALUE(current)-VALUE(next)`.

```
function STOCHASTIC-HILL-CLIMBING(problem) returns a state solution

t <- 0
current <- RANDOM(problem)
best <- MAKE-NODE(NULL)

repeat
    next <- RANDOM(EXPAND(current))
```

```
    if p = 1/(1 + e^(ΔE/T))
        then current <- next
        if VALUE(current) > VALUE(best)
            then best <- current
    t <- t + 1
until t = MAX
return STATE(best)
end
```

Il nuovo stato viene scelto con una probabilità p, calcolata come:

$$p = \frac{1}{1 + e^{\Delta E/T}} \tag{2.6.1}$$

In seguito dopo una serie di iterazioni l'algoritmo restituisce uno stato ottimo. L'algoritmo ha quindi un solo ciclo, e può scegliere un nuovo punto con una probabilità p, quindi anche di valore minore. Questa probabilità dipende da un parametro T costante durante l'esecuzione dell'algoritmo. Se vale 1, la probabilità di accettazione è sostanzialmente pari al 100%.

All'aumentare del valore i T la probabilità di accettazione tende al 50%, diventa quindi sempre meno importante la differenza della valutazione tra i due punti, effettivamente comporta una ricerca casuale, mentre al diminuire di T, la procedura rappresenta un semplice algoritmo Hill-Climbing.

In caso di stati di valore uguale, la probabilità è del 50%, se il valore dello stato `next` è minore, la probabilità diminuisce, mentre se il valore di `next` è maggiore dello stato corrente, la probabilità aumenta.

Bisogna trovare una "link function" tra l'intervallo $\Delta E/T$ e la probabilità p.

La caratteristica di poter scegliere come passo uno stato peggiore questo algoritmo potrebbe evitare massimi locali.

2.6.4 Algoritmo di Simulated Annealing

L'algoritmo di Simulated Annealing, introdotto nel 1983 da S. Kirkpatrick, C.D. Gelatt, Jr. e M.P. Vecchi nella rivista Science, è un algoritmo che migliora considerevolmente l'approccio dell'algoritmo precedente. Ha causato una vera e propria rivoluzione in termini di ottimizzazione, ebbe un enorme successo in ogni settore dell'informatica come l'algoritmo migliore per problemi di ricerca locale, anche solo nella sua versione base. Questo algoritmo è talmente importante che ogni anno vengono riuniti congressi annuali internazionali per discutere possibili ottimizzazioni.

Questo algoritmo prende il nome dall'analogia con il processo di metallurgia per temprare un materiale, questo processo infatti raggiungere uno stato di struttura cristallina ad energia minima. La differenza principale con l'algoritmo stocastico, è la possibilità di variare il valore di T diminuisce gradualmente durante l'esecuzione dell'algoritmo. Il valore di T parte da un valore elevato, per poi diminuire nel tempo, come se fosse la temperatura durante un processo di temperatura, per cui l'algoritmo si comporta in modo molto simile ad un normale hill-climber. Inoltre sceglie sempre uno stato se è migliore del punto corrente.

```
function SIMULATED-ANNEALING(problem) returns a state solution

t <- 0
current <- RANDOM(problem)
best <- MAKE-NODE(NULL)

repeat
    repeat
        next <- RANDOM(EXPAND(current))
        if VALUE(next) > VALUE(current)
            then current <- next
            if VALUE(current) > VALUE(best)
                then best <- current
            else if RANDOM[0,1) < e^(-ΔE/T)
                then current <- next
    until termination-condition
    T = g(T, t)
    t <- t + 1
until halting-condition
return STATE(best)
end
```

La probabilità di accettazione è leggermente diversa rispetto all'algoritmo precedente, poiché nel simulated annealing si considera solamente un semiasse dell'ascissa, dato che in caso `next` sia migliore non viene calcolata la probabilità di accettazione. Si accetta sempre uno stato di valore migliore, mentre l'accettazione di uno stato peggiore dipende da una probabilità. Quindi il mapping della link function non viene realizzata sull'intero intervallo $(-\infty, +\infty)$, ma solo su metà dell'ascissa, su $[0, +\infty)$, per cui è sufficiente utilizzare la funzione $e^{-\Delta E/T}$ avente come dominio il semiasse $[0, +\infty)$ e come codominio $(0, 1]$.

Questo ciclo interno viene effettuato un certo numero di volte fino ad una condizione di terminazione, che verrà trattata nelle implementazioni future. Finito questo ciclo si abbassa leggermente la temperatura tramite una funzione g e si incrementa il contatore delle iterazioni `t`. Anche la condizione di terminazione dell'algoritmo dipende dallo specifico problema e verranno trattate in futuro.

Molte implementazioni dell'algoritmo seguono la stessa sequenza di passi. Si assegna la variabile `T` alla temperatura massima, e si sceglie uno stato corrente casuale al primo passo. Si determina un successore assegnandolo direttamente se è migliore oppure tramite la funzione di probabilità e si ripete per un certo numero di cicli, e come passo finale si diminuisce la temperatura e si ripete dal secondo passo se la temperatura non ha raggiunto la temperatura minima. Quando la temperatura ha raggiunto la temperatura minima, si può scegliere se terminare l'algoritmo o ripeterlo un certo numero di volte, ripartendo dal primo passo.

3 Introduzione a Python

Python è un linguaggio di programmazione vastamente utilizzato nell'area dell'intelligenza artificiale e nel machine learning. Recentemente è diventato il linguaggio di programmazione più diffuso al mondo. Python è un linguaggio general-purpose, ideato da Guido van Rossum nel 1989, a più alto livello del C, poiché gestisce automaticamente le più fondamentali operazioni. Per cui è molto semplice e spesso utilizzato a fine didattico tra i primi linguaggi di programmazione insegnati.

La versione di Python utilizzata nel corso è la versione 3, nell'ambiente Anaconda. Può essere avviato tramite un interprete. Alla creazione di una variabile non è necessario definirne il tipo, il nome identificativo è arbitrario e può contenere numeri, ma non cominciare con un numero, viene consigliato di utilizzare un carattere minuscolo come primo carattere del nome. Esistono 33 parole chiave, non utilizzabili come nomi di variabili. Si può assegnare un valore ad una variabile tramite l'operatore `=`, senza specificarne il tipo.

3.1 Operatori

Esistono una serie di operatori aritmetici come `+`, `-`, `*`, \, `**`, per l'elevamento a potenza, `%` per il modulo. Una differenza tra Python 2 consiste nella gestione della divisione, infatti in Python 2 viene considerata solo la parte intera dell'operazione. Per ottenere lo stesso risultato esiste l'operatore \\, chiamato "floor division".

Gli operatori seguono un ordine di precedenza naturale, come la sintassi moderna matematica:

1. Parentesi;
2. Elevamento a potenza;
3. Moltiplicazione e divisione;
4. Addizione e sottrazione;
5. Operatori con lo stesso ordine valutati da sinistra verso destra.

In Python sono presenti tutti gli operatori booleani del C come `==` ed operatori di confronto come `<, <=, >, >=`, in aggiunta sono presenti altri operatori `is` ed `is not`. Inoltre sono presenti due versione degli operatori logici `&&` e `and`, —— e `or` e `!=` e `not`. Come in C un qualsiasi valore diverso da zero corrisponde al booleano `true`.

Si possono inserire dati dall'utente tramite la funzione `input()` e la funzione `raw_input()` per lo stesso comportamento di Python 2, e si può convertire in un tipo specifico con `tipo(var)`. I commenti vengono realizzati tramite il carattere #.

3.2 Istruzioni Condizionali

In Python per identificare funzioni o istruzioni condizionali non si usano parentesi, ma si indenta di quattro posizioni. Dopo la condizione dell' istruzione condizionale vanno inserite dei due punti `::`

```
if condizione:
    # corpo dell'if
else:
    # corpo dell'else
```

Si possono gestire le eccezioni con il costrutto `try` ed `except`:

```
try:
    # corpo del try
except:
    # corpo dell'except
```

3.3 Funzioni Built-In, Moduli e Definizione di Funzioni

In Python sono integrate tantissime funzioni utili, per svolgere attività comuni, utilizzabili senza doverle definire, queste sono funzioni "built-in". Per invocare funzioni presenti in un certo modulo e non built-in si utilizzata la notazione puntata `nomeModulo.nomeFunzione()`. Alcune tra le funzioni built-in più utili sono `max()` e `min()` che restituiscono il carattere più grande e più piccolo in una stringa; la funzione `len()` che restituisce la lunghezza della stringa. Per convertire variabili in certi tipi è già stata mostrata la funzione `tipo()`, dove al posto di `tipo` si inerisce il tipo specifico, si usa `str` per convertire in una stringa.

Per importare moduli contenenti altre funzioni si utilizza `importa` seguito dal nome del modulo da scaricare, che crea un object module con quel nome, si può rinominare seguendo questa istruzione con `as` seguito dall'alias del modulo. Si utilizza un alias per semplificare la notazione puntata.

Dati gli algoritmi analizzati precedentemente, si nota la necessita di introdurre generatori di numero casuali. La maggior parte di generatori casuali, sono deterministici, ovvero dato lo stesso input, generano la stessa sequenza di numeri casuali. Si utilizzano quindi numeri pseudo-casuali, generati da un calcolo deterministico, ma non è quasi possibile distinguerli da numeri generati casualmente. In Python esiste il modulo `random` contenente funzioni pertinenti alla generazione di numeri casuali. La funzione `random()` genera un numero casuale tra 0.0, compreso e 1.0, non compreso. Un altra funzione `randint()` accetta due parametri, estremi dell'intervallo, inclusi, per generare un numero intero tra loro compreso. Con la funzione `choice()` si può scegliere un elemento casualmente da una sequenza passata come argomento.

Per definire nuove funzioni si utilizza la parola chiave `def`, specificando il nome, tra parentesi tonde gli argomenti ed i due punti, indentando di quattro posizioni per scrivere il corpo della funzione:

```
def nomeFunzione(listaArgomenti):
    # corpo della funzione
    # resto del codice
```

Dopo aver passato degli argomenti ad una funzione, questi vengono assegnati a delle variabili locali. Si può utilizzare anche una variabile come argomento. Inoltre tutte le aggiunte possibili alle funzioni built-in, si possono effettuare sulle funzioni definite dall'utente.

Si dividono le funzioni in due tipi "fruitful function", funzioni produttive, e "void function", funzioni vuote, le prime restituiscono un valore, le seconde non restituiscono valore. Le prime quindi vengono usate per assegnare o inizializzare variabili. Se si tenta di assegnare il risultato di una void function ad una variabile, viene ottenuto un valore chiamato `None`. Questo valore ha un suo proprio tipo. Per definire una funzione produttiva, nel corpo si inserisce la parola chiave `return` seguita dai parametri da restituire come risultato della funzione.

3.4 Cicli

Si possono realizzare cicli tramite il costrutto `while` o `for`, seguito da una condizione booleana e dai due punti `:`:

```
while condizione:
    # corpo del ciclo
```

Si può interrompere il ciclo con `break`, e si può saltare l'iterazione corrente con `continue`. Quando bisogna iterare su una collezione, un insieme di elementi, si può realizzare un ciclo "for-each":

```
for elemento in collezione:
    # corpo del ciclo
```

3.5 Stringhe

Le stringhe sono sequenze di caratteri, indicizzati come fosse un array:

```
stringa[i] # (i+1)-esimo carattere
```

La funzione già discussa `len()` restituisce il numero di caratteri di una stringa, anche se può essere utilizzata per altri tipi di dati come dizionari. Poiché è strutturata come un array è possibile scandire ogni carattere della stringa individualmente con un ciclo for-each.

Talvolta è comodo accedere ad una sottostringa, o "slice", della stringa di partenza. La selezione di una sottostringa è simile alla selezione di un carattere, utilizzando due indici divisi da due punti per indicare l'inizio e la fine della sottostringa, il primo estremo è compreso, mentre il secondo no:

```
stringa[i:j] # slice contenente i caratteri da i a j-1
```

A volte si ha la necessità si realizzare una sottostringa che parte dall'inizio o la fine della stringa originaria, per effettuarlo si può omettere l'estremo corrispondente:

```
stringa[:j] # slice dall'inizio della stringa
stringa[i:] # slice fino alla fine della stringa
```

I valori di una stringa sono immutabili una volta definiti, per cui non è possibile modificarne il valore accedendo tramite indice, verrà sollevato un messaggio di errore. Si può modificare una stringa realizzando una nuova stringa, come variante, tramite l'operatore di concatenazione `+`:

```
stringa = stringa_1 + stringa_2
```

L'operatore `*` applicato su una lista, la replica un certo numero di volte specificato.

L'operatore `in` è estremamente importante, permette di individuare se una stringa è sottostringa di un'altra, restituisce un valore booleano vero o falso:

```
stringa_1 in stringa_2
```

Si possono inoltre utilizzare operatori di confronto tra stringhe all'uguaglianza con `==`, oppure con `<, >`, per confrontarle in ordine alfabetico. In Python le maiuscole vengono prima delle minuscole. Le stringhe sono degli oggetti che oltre alla sequenza di caratteri contengono oltre i dati anche i metodi disponibili per ogni istanza dell'oggetto stringa.

Con la funzione `type` si ha la possibilità di identificare il tipo dell'oggetto su cui viene operata e `dir` mostra i metodi disponibili. Per utilizzare un metodo si utilizza la notazione puntata con il nome dell'istanza dell'oggetto. Uno dei metodi sulle stringhe è `find()` che prende come argomento una sottostringa, ed un indice opzionale da cui cercare il carattere, e restituisce la prima posizione dell'occorrenza della sottostringa specificata.

Il metodo `strip()` rimuove lo spazio bianco prima e dopo la stringa. Il metodo `startswith()` restituisce un valore booleano se la stringa comincia con la sottostringa passata come argomento. Il metodo `capitalize()` consente di impostare a maiuscolo il primo carattere, per mettere tutti i caratteri in maiuscolo si utilizza il metodo `upper()`.

L'operatore "format" `%` permette di costruire stringhe formattando la stringa rispetto a dati contenuti in altre variabili. All'interno di una stringa chiamata "format string" si può inserire questo operatore seguito da una lettera per specificare il tipo di dato associato, chiamate "format sequences". Questa viene utilizzata come il primo argomento, mentre il secondo operando è la variabile da formattare, questo produce una stringa:

```
>>> x = 1
>>> 'il numero è %d' % x
'il numero è 1'
```

Se in una stringa compare una sequenza di format sequences si specifica l'operando come una sequenza, una tupla, tra parentesi tonde, separando gli elementi con virgole:

```
>>> x = 1
>>> 'il %s è %d' % ('numero', x)
'il numero è 1'
```

La tupla deve corrispondere in numero, ordine e tipo alle format sequences nella stringa. Si utilizza `%d` per numeri interi `%g` per floating point e `%s` per stringhe.

Esiste un nuovo operatore format, scrivendo tra parentesi graffe `:` seguito dal carattere corrispondente al tipo, dopo la stringa si invoca il metodo `format()` specificando nell'argomento il valore da inserire:

```
>>> 'il {:s} è {:d}' .format('numero', 1)
'il numero è 1'
```

Questa implementa tutte le caratteristiche già trattate per il format `%`.

3.6 Liste

Una lista è una sequenza di item di qualsiasi tipo, per cui a differenza di una stringa il tipo non è omogeneo. All'interno di una lista è possible inserire una lista, creando una lista annidata. Una lista si realizza specificando gli elementi tra parentesi quadre:

```
[item_1, item_2, [item_3, item_4]]
```

Una lista che non contiene elementi si chiama lista vuota mediante `[]`, oppure con il comando `list()`. Si può assegnare ad una variabile una lista allo stesso modo di un valore:

```
>>> lista = [item_1, item_2, [item_3, item_4]]
```

Le liste al contrario delle stringhe sono modificabili, con la stessa sintassi per le stringhe, specificando tra parentesi quadre l'indice corrispondente, ed è possibile aggiornare il valore contenuto in questo indice. Se ad un certo indice è presente una lista, questo elemento viene trattato come lista e quindi si possono utilizzare due coppie di parentesi quadre per indicizzare questi elementi annidati:

```
>>> lista[2][2]
item_4
```

Si può considerare una lista come un mapping tra indici ed elementi, come nelle stringhe possono essere variabili ed espressioni, se si prova a leggere o scrivere ad un indice non esistente si solleva un `IndexError`. Mentre se si utilizza un indice negativo, si conta all'indietro partendo dalla fine della stringa. L'operatore `in` ha un funzionamento analogo a quello per le stringhe.

Un modo per scandire la lista è con un ciclo for-each, come per le stringhe, oppure iterando manualmente su ogni elemento. Questo si può effettuare tramite la funzione `range()` che permette di iterare su un insieme di valori, fornito l'ultimo come argomento:

```
for i in range(len(lista)):
    # corpo del for
```

L'operatore di concatenazione vale anche per le liste, analogamente per l'operatore `*` che replica una lista un certo numero di volte. L'operatore slice può essere utilizzato anche sulle liste per ottenere una sottolista:

```
lista[i:j] # sottolista da i a j-1
lista[:j]  # sottolista da 0 a j-1
lista[i:]  # sottolista da i a len(lista)
lista[:]   # copia della lista
```

L'operatore slice può essere utilizzato per aggiornare una sequenza di elementi in una lista. La sottolista da aggiornare deve avere lo stesso numero dei valori della lista originaria.

Come per le stringhe, le liste sono oggetti e contengono metodi built-in. Il metodo `append()`. aggiunge un nuovo elemento alla fine di una lista, il metodo `extend()` prende come argomento una lista e la concatena alla lista su cui si è operato. Se si passa una lista al metodo `append()` si genera

una lista annidata. Si può ordinare gli elementi di una lista dal minore al maggiore con `sort()` in ordine alfabetico. Molti dei metodi su liste, come questi, sono metodi void.

Per estrarre un elemento dalla lista si utilizza il metodo `pop()` che estrae dalla lista l'ultimo elemento, oppure l'elemento di indice specificato, lo restituisce rimuovendolo dalla lista:

```
>>> lista.pop(1)
item_2
>>> print(lista)
[item_1, [item_3, item_4]]
```

L'operatore `del` rimuove un elemento dalla lista:

```
>>> del lista[1]
>>> print(lista)
[item_1, [item_3, item_4]]
```

Utilizzando lo slice si può rimuovere una sottostringa allo stesso modo:

```
>>> del lista[:1]
>>> print(lista)
[[item_3, item_4]]
```

Se non si conosce l'indice dell'elemento da cancellare si può utilizzare il metodo `remove()`, ma non restituisce alcun valore:

```
>>> lista.remove(item_2)
>>> print(lista)
[item_1, [item_3, item_4]]
```

Alcuni metodi come `sum()` possono essere invocati solamente se la lista contiene solo numeri, altri metodi possono essere invocati se la lista contiene dati confrontabili tra di loro. Si può convertire una stringa in una lista tramite la funzione `list()`, suddividendo la stringa in caratteri singoli, mentre si può suddividere in parole singole, separate da spazi con la funzione `split()`. Si può specificare un delimitatore in questa funzione `split()` da utilizzare al posto del delimitatore di default spazio. La funzione inversa della `split()` è il metodo `join()`, permette di realizzare una stringa, applicato su una stringa contenente il delimitatore da utilizzare per separare gli elementi. Si può concatenare senza spazi con la stringa vuota `[]`.

In Python si possono confrontare due oggetti con l'operatore `is`, se si creano due stringhe di contenuto uguale, Python realizza un unico oggetto stringa, mentre se vengono realizzare due liste, contenenti gli stessi elementi, sono due oggetti distinti. Le due liste sono equivalenti poiché hanno lo stesso valore, ma non sono identiche, poiché non sono lo stesso oggetto. Due oggetti identici sono anche equivalenti.

Si può assegnare ad una variabile ad un'altra in modo che entrambe si riferiscano allo stesso oggetto, e quindi anche se si tratta di liste il confronto `is` sarà verificato. Un oggetto che ha più di riferimenti ha un "alias", più di un nome, e si dice un oggetto "aliased". Se gli oggetti sono mutabili questo può essere fonte di errore. Si consiglia di evitare di utilizzare alias quando si lavora con liste

ed oggetti mutabili. Si distingue tra operazioni che creano nuove liste ed operazioni che modificano lista, come l'operatore di concatenazione `+` ed il metodo `append()`, quest'ultimo modifica la lista su cui è invocato analogamente di `extend()`, mentre l'operatore di concatenazione genera una nuova lista.

C'è uno speciale costrutto per accedere a tutti gi elementi di una lista effettuando la stessa operazione su di loro e memorizzare i nuovi elementi in un'altra lista:

```
>>> lista_1 = [1, 2, 3]
>>> lista_2 = [item * 2 for item in lista_1]
>>> print(lista_2)
[2, 4, 6]
```

Quando bisogna accedere contemporaneamente a più liste si può utilizzare la sintassi che utilizza la parola chiave `zip` per trattare la sequenza di liste passate contemporaneamente:

```
for item_1, item_2, ... in zip(lista_1, lista_2, ...):
    # corpo del ciclo, per ogni iterazione si ha
    # item_1 = lista_1[i], item_2 = lista_2[i], ...
```

In questo modo si può iterare su tutte le liste insieme, senza dover specificare per ognuna il loro indice. Questa funziona associa le liste e termina alla lista più corta.

3.7 Dizionari