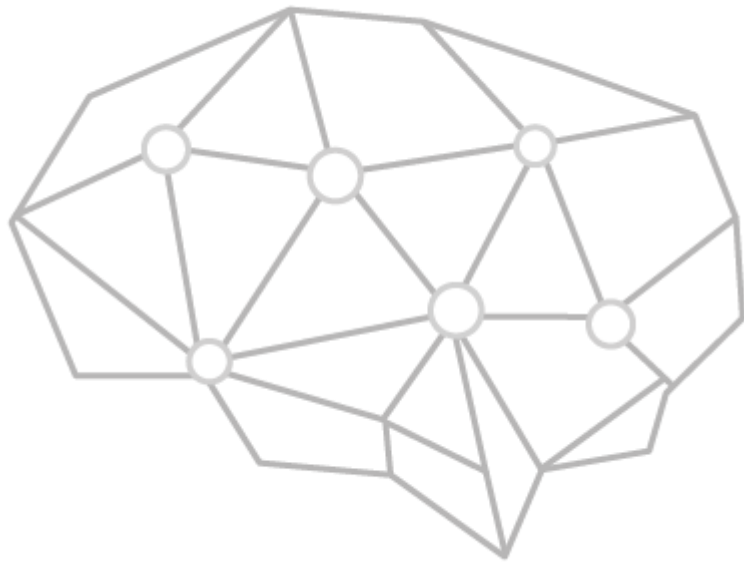


Corso di Intelligenza Artificiale. Prof. Filippo Neri

Anno Accademico 2020-2021

EXERCISES PROJECTS

- MODELING WITH DECISION TREES –
- OPTIMIZATION –
- BAYESIAN NETWORK, NEURAL NETWORK -



Antonio Romano N46004321
Giuseppe Riccio N46004297
Salvatore Pernice N46004341
Salvatore Ruggiero N46004239



UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

INDICE

PREFAZIONE	4
ESERCITAZIONE 1: MODELING WITH DECISION TREES	6
ESERCIZIO 1	6
1.1 Introduzione alla costruzione di un albero di decisione.	6
1.2 Dataset Echocardiogram.....	7
1.3 Dataset Mushrooms	10
ESERCIZIO 2	12
2.1 Perché' un agente in grado di apprendere Alberi di Decisione è considerato Intelligente?	12
2.2 Cosa si intende con il termine apprendimento quando si utilizza il DT learning?	13
ESERCIZIO 3	14
3.1 Spiegare l'algoritmo A*.....	14
3.2 Descrivere i casi in cui si applica e cosa si intende per euristica ammissibile in A*.....	14
3.3 Mostrare un esempio di applicazione di A* su un problema di navigazione stradale	15
ESERCIZIO 4	18
4.1 Descrivere con un esempio come si utilizza un albero di decisione	18
ESERCIZIO 5	20
5.1 Descrivere con un esempio il processo di apprendimento di un albero di decisione	20
ESERCIZIO 6	25
6.1 Cosa sono: lo spazio degli stati, il grafo degli stati e l'albero di ricerca (Tree Search) ...	25
6.2 Descrivere in modo sintetico il meta algoritmo generale di costruzione di un Tree Search	25
6.3 Spiegare in modo sintetico le caratteristiche degli algoritmi Tree Search visti a lezione	25
ESERCITAZIONE 2: OPTIMIZATION	28
ESERCIZIO 1	28
1.1 Introduzione agli algoritmi di ricerca locale.....	28
1.2 Ottimizzazione di una funzione	29
ESERCIZIO 2	33
2.1 Un agente Intelligente deve colorare le provincie della Regione Campania evitando che due provincie confinanti abbiano lo stesso colore, l'Agente ha a disposizione solo 2 colori. Se il compito non è possibile spiegare perché non si riesce utilizzando gli algoritmi visti a lezione.....	33
2.2 Ripetere il punto 1 con 3 colori a disposizione.....	34
2.3 Ripetere il punto 1 con 4 colori a disposizione.....	35
ESERCIZIO 3	36
3.1 Come utilizzereste un Algoritmo Genetico per risolvere il punto 3) dell'Esercizio 2?	36
ESERCIZIO 4	37
4.3 Cosa si intende per deducibilità logica/inferenza.....	39
4.4 Dimostrazione tramite inferenza logica (regola di risoluzione).....	39

4.5 Un esempio di inferenza logica (risoluzione) espressa nella logica proposizionale	40
ESERCITAZIONE 3: BAYESIAN NETWORK, NEURAL NETWORK	41
ESERCIZIO 1	41
1.1 Riassumere i concetti principali delle reti bayesiane (rappresentazione e inferenza)	41
1.2 Descrivere e spiegare con un esempio come il formalismo delle reti bayesiane può essere applicato - 3 pagine max	44
ESERCIZIO 2	46
2.1 Riassumere i concetti principali delle reti neurali (spiegare come sono implementati i neuroni, la loro architettura e l'algoritmo di error back propagation)	46
2.2 Come utilizzereste una rete neurale per riconoscere la presenza o meno di alberi in un'immagine digitale - 3 pagine max.....	49
ESERCIZIO A (opzionale).....	52
ESERCIZIO B (opzionale).....	54

PREFAZIONE

La stesura della seguente trattazione è mirata alla documentazione di realizzazione di esercitazioni assegnate. Le tracce delle esercitazioni:

Esercitazione 1

Esercizio 1

1) Eseguire sul proprio elaboratore (non riscrivere) il codice riportato nel Capitolo 7 'Modeling with Decision Trees' [wb1], pp 142 - 165, riflettere sui risultati ottenuti dall'esecuzione del codice e su quanto discusso in classe. Avete quindi ora un ambiente software disponibile per fare qualche esperienza pratica su DT learning

2) Scaricare un dataset da UC Irvine ML Repository a cui applicare DT Learning (codice del punto 1) e commentare per iscritto i risultati ottenuti. Un possibile dataset da utilizzare è chiamato Mushrooms.

3) Cambiare la percentuale di dati nell'insieme di training e di test (10%-90%, 20%-80%, ...) e creare un grafico con le performance di apprendimento

Scrivere un report di max 6 pagine sulla vostra esperienza per i punti su indicati

Esercizio 2

Perché' un agente in grado di apprendere Alberi di Decisione è considerato Intelligente? Cosa si intende con il termine apprendimento quando si utilizza il DT learning?

Esercizio 3

Spiegare l'algoritmo A*, descrivere i casi in cui si applica, cosa si intende per euristica ammissibile in A*, mostrare un esempio di applicazione di A* su un problema di navigazione stradale.

Esercizio 4

Descrivere con un esempio come si utilizza un albero di decisione.

Esercizio 5

Descrivere con un esempio il processo di apprendimento di un albero di decisione.

Esercizio 6

Cosa sono: lo spazio degli stati, il grafo degli stati e l'albero di ricerca (Tree Search). Descrivere in modo sintetico il meta algoritmo generale di costruzione di un Tree Search. Spiegare in modo sintetico le caratteristiche degli algoritmi Tree Search visti a lezione.

Esercitazione 2

Esercizio 1

1) Eseguire sul proprio elaboratore (non riscrivere) il codice riportato nel Capitolo 5 'Evolutionary Optimization' [wb1], pp 86-94, riflettere sui risultati ottenuti dall'esecuzione del codice e su quanto discusso in classe. Avete quindi ora un ambiente software disponibile per fare qualche esperienza pratica su ottimizzazione di funzioni e metodi greedy / evolutivi.

2) provare ad ottimizzare la funzione seguente con una delle tecniche di ottimizzazione del punto 1) che preferite:

$F(x) = (\text{se } x < 5.2, F(x) = 10; \text{ se } 5.2 \leq x \leq 20, F(x) = x^2; \text{ se } 20 < x, F(x) = \cos(x) + 160 \cdot x)$

$F(x)$ è definita sull'intervallo $[-100, 100]$ e non definita altrove

Esercizio 2

1) Un agente Intelligente deve colorare le provincie della Regione Campania evitando che due provincie confinanti abbiano lo stesso colore, l'Agente ha a disposizione solo 2 colori. Se il compito

non è possibile spiegare perché' non si riesce utilizzando gli algoritmi visti a lezione.

2) Ripetere il punto 1 con 3 colori a disposizione.

3) Ripetere il punto 1 con 4 colori a disposizione.

Esercizio 3

Come utilizzereste un Algoritmo Genetico per risolvere il punto 3) dell'Esercizio 2?

Esercizio 4

Riassumere i concetti principali della logica proposizionale inclusi:

1) cosa si intende per conseguenza logica

2) dimostrazione tramite model checking

3) cosa si intende per deducibilità logica/inferenza

4) dimostrazione tramite inferenza logica (regola di risoluzione)

5) un esempio di inferenza logica (risoluzione) espressa nella logica proposizionale - 4 pagine max

Esercitazione 3

Esercizio 1

a) Riassumere i concetti principali delle reti bayesiane (rappresentazione e inferenza)

b) descrivere e spiegare con un esempio come il formalismo delle reti bayesiane può essere applicato - 3 pagine max

Esercizio 2

a) Riassumere i concetti principali delle reti neurali (spiegare come sono implementati i neuroni, la loro architettura e l'algoritmo di error back propagation)

b) come utilizzereste una rete neurale per riconoscere la presenza o meno di alberi in un'immagine digitale - 3 pagine max

Gli esercizi seguenti sono opzionali. Possono essere liberamente svolti o meno senza che questo influenzi il voto d'esame. Il codice relativo è disponibile su Google drive o all'indirizzo web relativo all'esercizio. Non dedicate più di 2 ore di tempo per esercizio sia che riusciate o meno a svolgerlo. Se decidete di svolgerli, riassumere l'esperienza positiva o negativa in max 2 pagine per esercizio.

Esercizio A (opzionale)

Studiare il codice che mostra come una rete neurale MLP possa essere applicata al problema di handwritten character recognition

<https://mxnet.incubator.apache.org/tutorials/python/mnist.html>

Esercizio B (opzionale)

Eseguire e studiare la soluzione del problema di Cart Pole Balancing utilizzando il Q-Learning

<https://github.com/YuriyGuts/cartpole-q-learning/blob/master/cartpole.py>

“

Lavoro svolto con dedizione e pazienza
ai tempi del **CoronaVirus**.

Gli autori

ESERCITAZIONE 1:

MODELING WITH DECISION TREES

ESERCIZIO 1

1.1 Introduzione alla costruzione di un albero di decisione.

Gli alberi di decisione (o alberi decisionali) sono una rappresentazione dell'apprendimento nel **machine learning**. Un albero di decisione è un sistema con n variabili in input e m variabili in output. Le variabili in input (**attributi**) sono derivate dall'osservazione dell'ambiente. Le variabili in output, invece, identificano la decisione/azione da intraprendere. (Cfr. **Es.1.4** Successivamente).

Grazie al supporto del codice riportato nel Capitolo 7 'Modeling with Decision Trees' si è riflettuto sui risultati ottenuti dall'esecuzione del codice con l'utilizzo di due dataset prelevati dall'UC Irvine ML Repository. Partendo dal dataset iniziale costruiamo l'albero di decisione su una percentuale di dati desiderata (ad esempio il 60%) questo insieme di dati prende il nome di **training set**, mentre i dati rimanenti saranno utilizzati per la fase di test dell'albero decisionale ottenuto e per questo l'insieme viene chiamato **test set**.

Con il codice fornito nel Capitolo 7 e quello implementato dal nostro gruppo, dunque, i passi logici da seguire nell'esecuzione sarà il seguente:

```
#Funzione che apre il file con i dati presenti nel dataset
mydata=tp.aprifile("dataset.txt")

#Funzione che divide il dataset in due sottinsiemi train_data
#e test_data sulla base della percentuale di dati fornita in input
train_data, test_data = tp.splitdataset2(mydata, percentuale_dati, [])

#Funzione del Capitolo 7 che costruisce l'albero con il training set
decision_tree = tp.buildtree(train_data)

#Funzione del Capitolo 7 che disegna su un file JPEG l'albero decisionale
tp.drawtree(decision_tree, "decision_tree.jpeg")
```

Figura 1: Costruzione dell'albero di decisione

Quindi, dopo la costruzione dell'albero si passa alla fase di test per valutarne la sua performance inserendo i dati del test set all'interno della funzione (**learningcurve()** sviluppata da noi) e utilizzando percentuali via via crescenti (in maniera del tutto automatica all'interno della funzione stessa, il testing set viene incrementato dal 10% fino al 90%) dell'insieme di dati di test, la funzione restituisce un grafico che ci permetterà di capire per ogni percentuale di dati test utilizzati quanti casi vengono trattati correttamente nell'albero decisionale e quindi, **capire se l'albero costruito generalizza correttamente il problema**. Proseguendo i passi di esecuzione del codice della Figura 1, avremo che per costruire il grafico di cui abbiamo parlato precedentemente occorrerà aggiungere la seguente riga di codice:

```
#Funzione che disegna il grafico di performance dell'albero
tp.learningcurve(test_data)
```

Figura 2: Funzione che disegna il grafico delle performance

L'informazione che fornisce il grafico è particolarmente indicativa e viene spesso espressa in questo modo:

$$\text{Accuratezza} = \% \text{ casi correttamente trattati}$$

Da cui può essere ricavato anche il seguente parametro di bontà dell'albero:

$$\text{ErrorRate} = 1 - \text{Accuratezza}$$

La caratteristica *Training Set Size - % Correct On Test Set* a seconda del suo andamento assume diversi significati:

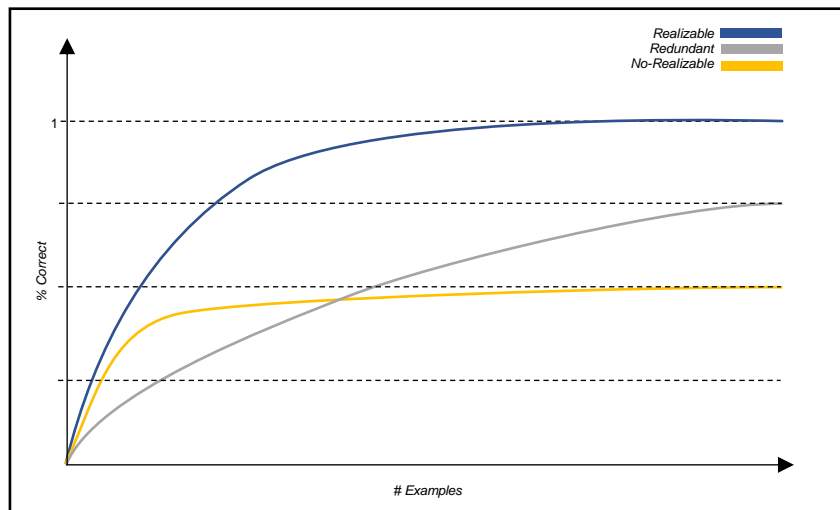


Figura 3: Tipici andamenti della Learning Curve

- Se la caratteristica si avvicina a quella **realizzabile** allora si è appreso il modello perfetto.
 - Se la caratteristica si avvicina a quella **non realizzabile** non migliora più anche aumentando i dati. In questo caso o il problema non è deterministico oppure abbiamo rappresentato il problema con una tecnica non adeguata.
 - Se la caratteristica si avvicina a quella **ridondante** vuol dire che abbiamo creato uno spazio delle possibili ipotesi così grande che la tecnica ci mette tempo per trovare l'albero adatto.
- Per il nostro caso di studio abbiamo selezionato dall'UCI i seguenti dataset:

- **Mushrooms** <https://archive.ics.uci.edu/ml/datasets/Mushroom>
- **Echocardiogram** <https://archive.ics.uci.edu/ml/datasets/Echocardiogram>

NB: I dataset sono stati scelti sulla base della diversificazione nella loro "correttezza".

1.2 Dataset Echocardiogram

Scopo: Nel dataset sono presenti i dati di tutti i pazienti che hanno sofferto di attacchi di cuore in passato. Alcuni sono ancora vivi e altri no. Le variabili di sopravvivenza e ancora in vita, se prese insieme, indicano se un paziente è sopravvissuto per almeno un anno dopo l'attacco di cuore. Il problema da affrontare è prevedere correttamente se il paziente sopravvivrà o meno. Il dataset è caratterizzato da ben 12 attributi e 132 istanze.

Echocardiogram training_set=40%, test_set=60%

Usando il 40% dati per il training (ovvero 53 istanze), abbiamo che l'albero (**Figura 5**) ha profondità massima 6, ed un grafico delle performance tendenzialmente crescente fino ad un massimo di circa 0.75 con alcuni punti in corrispondenza del 40%-60%, in cui la performance si riduce. (**Figura 4**)

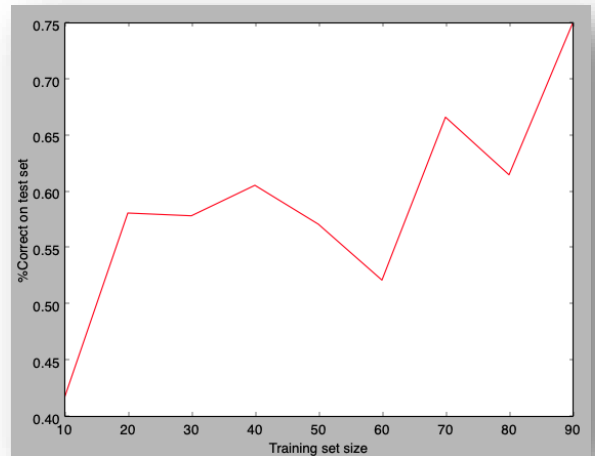


Figura 4: Performance training_set = 40% e test_set = 60%

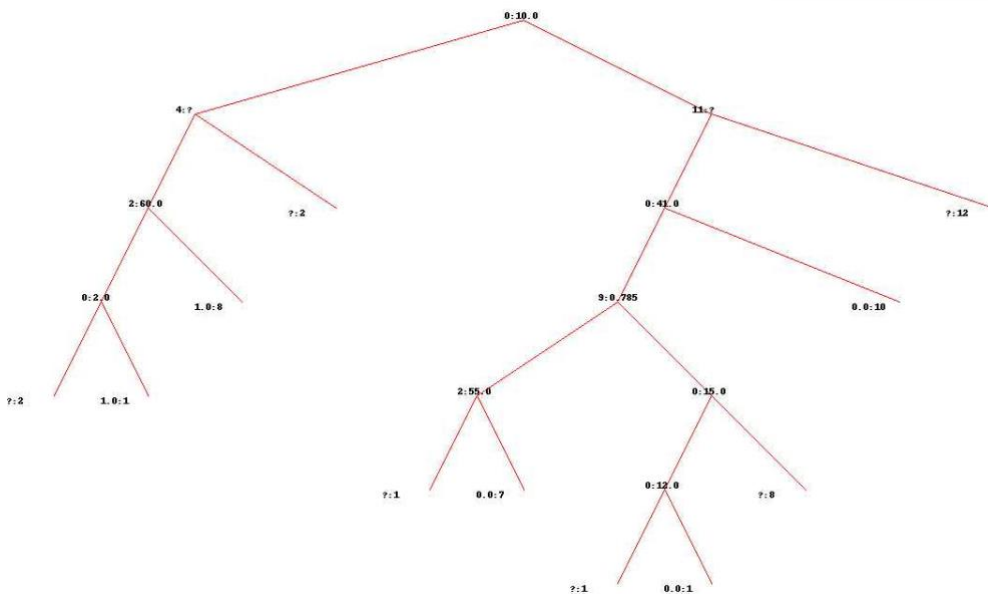


Figura 5: Albero training_set = 40% e test_set = 60%

Echocardiogram training_set=60%, test_set=40%

Usando il 60% dati per il training (ovvero 79 istanze), abbiamo che l'albero (**Figura 7**) ha profondità massima 8, quindi incomincia a diventare meno generale, ed un grafico delle performance tendenzialmente crescente con un massimo di 0.89, ma con alcune fasi di decrescita nel passaggio tra il 30%-40% e 60%-80% del training set size. (**Figura 6**)

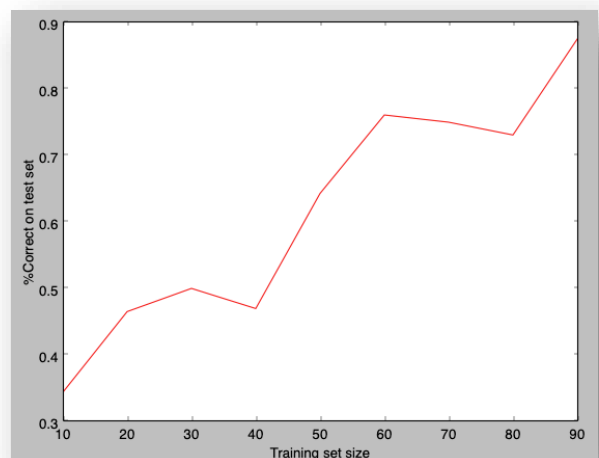


Figura 6: Performance training_set = 60% e test_set = 40%

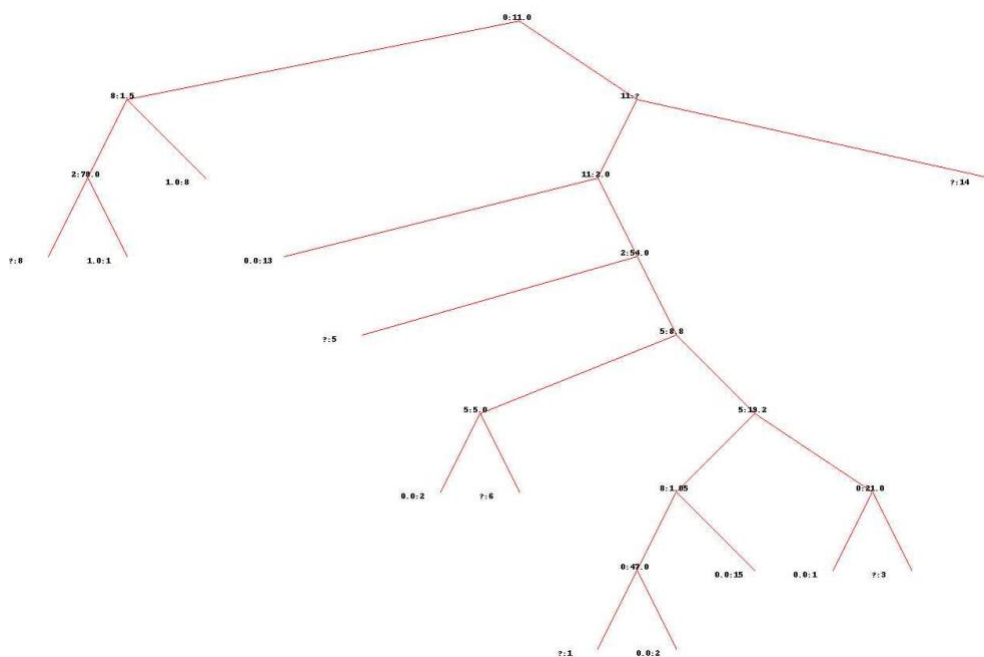


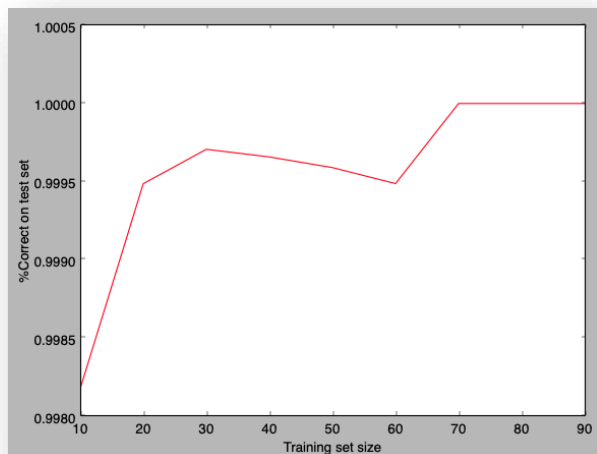
Figura 7: Albero training_set = 60% e test_set = 40%

Si nota dai grafici delle performance che la learning curve sale molto lentamente, questo perché abbiamo usato una rappresentazione del mondo ridondante ed è possibile migliorare la stima eliminando dalla tabella della rappresentazione del mondo di Echocardiogram degli attributi.

1.3 Dataset Mushrooms

Scopo: Nel dataset sono presenti i campioni di 23 specie di funghi con branchie nella famiglia Agaricus e Lepiota. Ogni specie è identificata come sicuramente commestibile, decisamente velenosa, o di commestibilità sconosciuta e sconsigliata. Quest'ultima classe è stata combinata con quella velenosa. Quindi l'albero decisionale deve identificare quali funghi sono commestibili o meno. Il dataset è caratterizzato da ben 21 attributi e 8124 istanze.

Mushrooms training_set=40%, test_set=60%



Usando il 40% dati per il training (ovvero 3250 istanze), abbiamo che l'albero (**Figura 8**) ha profondità massima 6, ed un grafico delle performance tendenzialmente crescente fino ad un massimo di circa 1.0 con alcuni punti in corrispondenza del 28%-60%, in cui la performance si riduce. (**Figura 9**)

Figura 8: Performance training_set = 40% e test_set = 60%

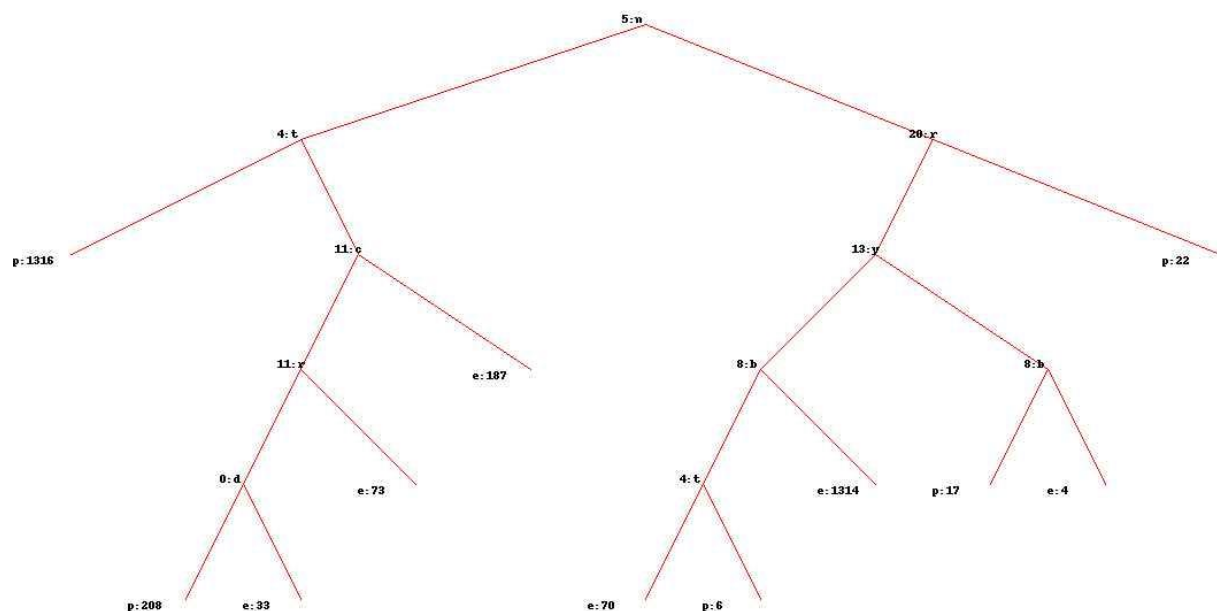
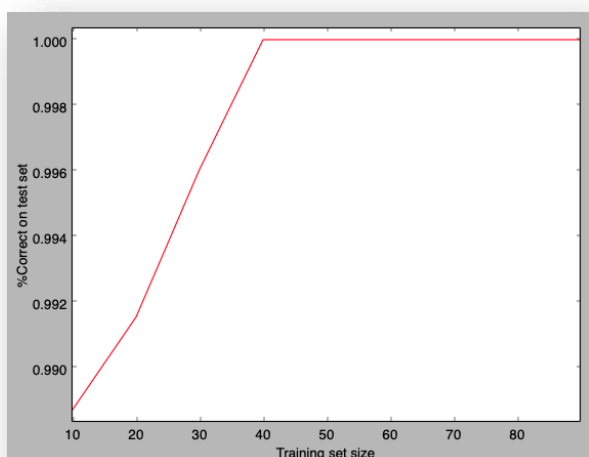


Figura 9: Albero training_set = 40% e test_set = 60%

Mushrooms training_set= 60%, test_set=40%



Usando il 60% dati per il training (ovvero 4874 istanze), abbiamo che l'albero (**Figura 11**) ha profondità massima 6, ed un grafico delle performance tendenzialmente crescente fino ad un massimo di circa 1.0. (**Figura 10**)

Figura 10: Performance training_set = 60% e test_set = 40%

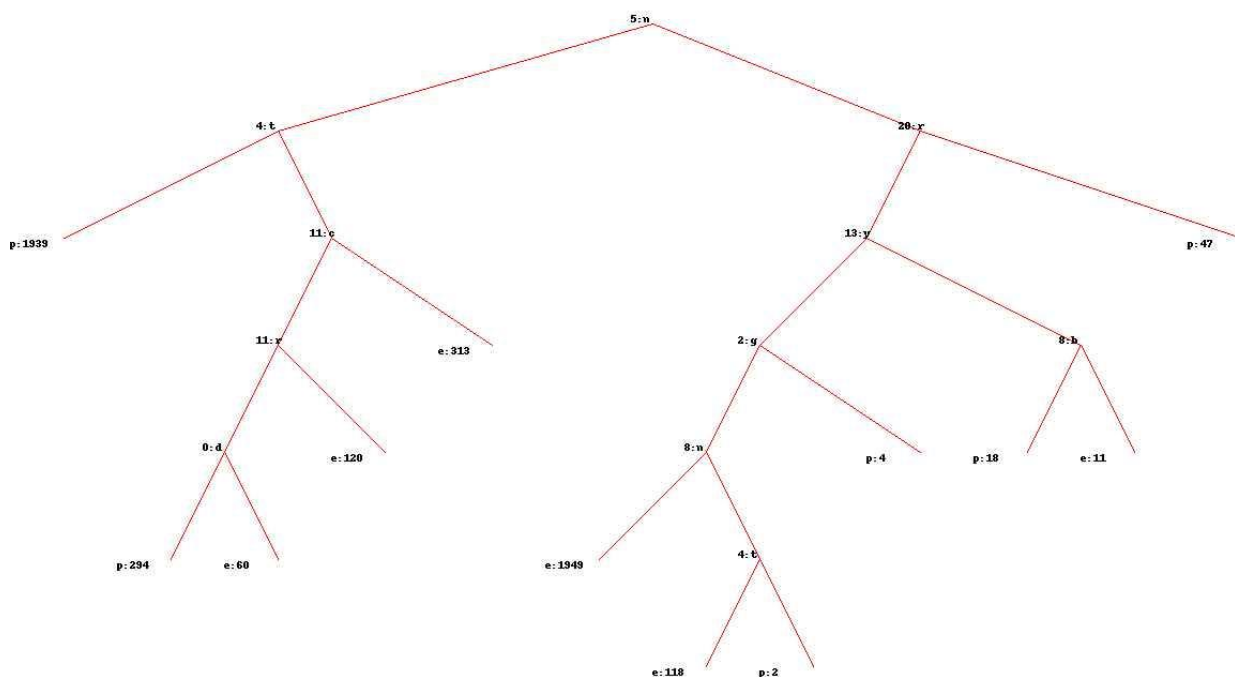


Figura 11: Albero training_set = 60% e test_set = 40%

Si nota dai grafici delle performance che la learning curve si avvicina molto alla curva realizzabile (dalla **Figura 3**) dunque abbiamo appreso il modello perfetto.

ESERCIZIO 2

2.1 Perché' un agente in grado di apprendere Alberi di Decisione è considerato Intelligente?

Un agente in grado di apprendere un albero di decisione è definito intelligente perché esso attraverso un processo **induttivo**, cioè a partire dalla descrizione di esempi di comportamento, di approssimare la **funzione target f** , che rappresenta la funzione che l'agente deve apprendere.

In particolare, l'agente classifica gli esempi (**positivi e negativi**) in base a delle euristiche proprie del problema di interesse, maggiori saranno gli esempi e più l'*ipotesi h* approssimerà in maniera migliore la f e quindi, spiegherà con maggior correttezza gli esempi dati. Tuttavia, non si limita a trovare una qualsiasi h , ma cerca una h che in base al "rasoio di Ockham" deve essere la più semplice possibile per generalizzare meglio il problema senza generare il fenomeno dell'**overfitting**, cioè quel fenomeno in cui il modello spiega così bene gli esempi forniti in fase di test che non riesce a spiegare gli esempi futuri.

Esempio: Curve fitting

Per comprendere meglio il fenomeno dell'**overfitting**, consideriamo gli esempi rappresentati nella **Figura 12** dalle **X** del grafico. Se si cerca una funzione che spieghi quei punti (passa per tali punti), si può notare che presa una qualsiasi funzione coerente con i dati (retta, esponenziale, sinusoidale, spezzata) all'aumentare degli esempi spiegati aumenta la complessità della funzione da studiare, e dunque, guardando la **Figura 13** risulta più conveniente scegliere la retta di regressione (1) piuttosto che la funzione esponenziale (2), quella (3) o (4), dato che la (1) è più semplice da modellare.

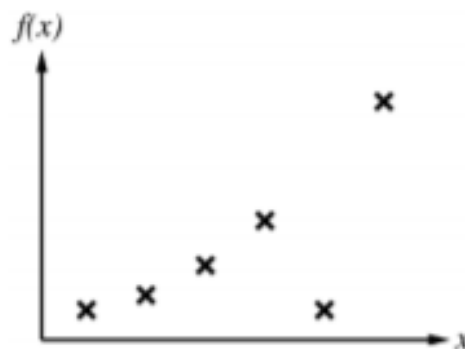


Figura 12: Rappresentazione grafica di un insieme di esempi

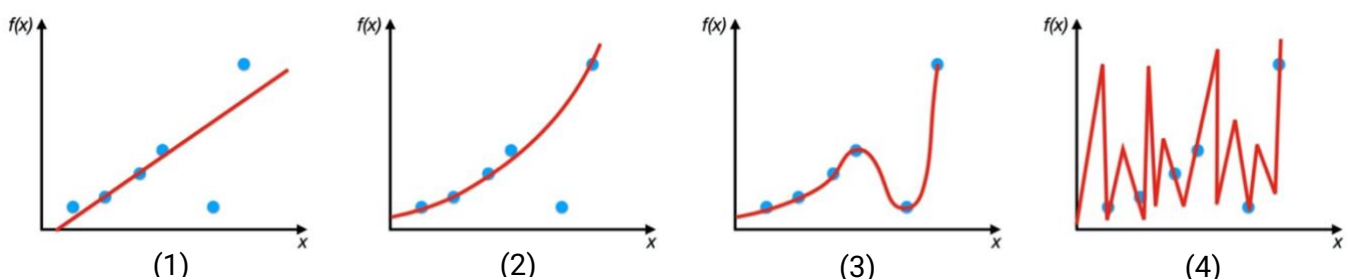


Figura 13: Diverse funzioni che interpolano i punti

2.2 Cosa si intende con il termine apprendimento quando si utilizza il DT learning?

Con il termine apprendimento nel contesto del DT learning intendiamo quel processo attraverso il quale, data la **"Tabella dei sistemi di apprendimenti"** composta sulle righe dagli esempi dati all'agente, ogni esempio avrà un insieme di colonne che caratterizzano le proprietà del sistema ed infine, una colonna target che caratterizza la classificazione (**vero o falsa**) dell'esempio che si vuole apprendere.

A partire da tale tabella si costruisce attraverso un algoritmo di **DTL (Decision Tree Learning)** un albero che permette ad esempio, di rappresentare in maniera automatica il comportamento di un essere umano intelligente, che si trova dinanzi al dover effettuare una scelta.

L'algoritmo elabora questi dati in un modo non previsto dal programmatore. L'essere umano non sa a priori come rappresentare il mondo e quali attributi mettere nella tabella. Si procede per raffinamenti successivi, vedendo se il risultato è soddisfacente applicando la tecnica di intelligenza artificiale.

Un albero è un **direct acyclic graph (DAG)**. Dalla radice si diramano i rami dell'albero. Scendiamo fino ad arrivare alle foglie. L'albero di decisione è fatto in questo modo: dentro alle scatole intermedie ci sono delle condizioni. Quando si arriva alla foglia, si giunge alla classificazione. Le condizioni sono i nomi degli attributi e da ogni nodo escono tanti archi quanti sono i valori che l'attributo può prendere.

La differenza tra l'informatica tradizionale e l'informatica che si dedica all'intelligenza artificiale è proprio il concetto di learning in cui l'agente apprende e non esegue un codice statico, scritto precedentemente, ma impara dalle sue interazioni con l'ambiente. Tramite uno score è possibile parametrare le performance e far capire all'agente se si sta comportando in modo positivo aumentando lo score o se si sta comportando in modo negativo penalizzando lo score.

ESERCIZIO 3

3.1 Spiegare l'algoritmo A*

L'algoritmo A* fa parte della famiglia degli algoritmi Best-First Search, cioè degli algoritmi di ricerca informata (basati su grafi) che si basano sulle informazioni date sul problema per scegliere la soluzione nello spazio delle soluzioni candidate; ovvero la sequenza di passi che porta da uno stato iniziale ad uno stato obiettivo. E quindi, tale algoritmo si adopera quando siamo interessati non solo alla sequenza di passi, ma anche al costo associato al cammino, in particolare si associa ad ogni nodo un grado di desiderabilità (in genere, quello più desiderabile è quello che si avvicina maggiormente alla soluzione). Il criterio usato è dato dalla funzione:

$$f(n) = g(n) + h(n)$$

in cui:

- **$g(n)$** è il costo storico per arrivare a partire dal nodo radice al nodo n -esimo
- **$h(n)$** è un'euristica che ci permette di stimare il costo che impieghiamo per arrivare al nodo obiettivo a partire dal nodo n .

3.2 Descrivere i casi in cui si applica e cosa si intende per euristica ammissibile in A*

L'algoritmo A* risulta efficace nelle situazioni in cui cerchiamo all'interno di un albero di ricerca la soluzione ottimale e completa. Ciò avviene solo e soltanto se ci ritroviamo in uno spazio di stati limitato e se utilizziamo un'euristica $h(n)$ ammissibile. L'euristica influenza fortemente i risultati conseguiti da A*.

Esso, in particolare, ne determina il tempo complessivo di esecuzione e la qualità della soluzione finale.

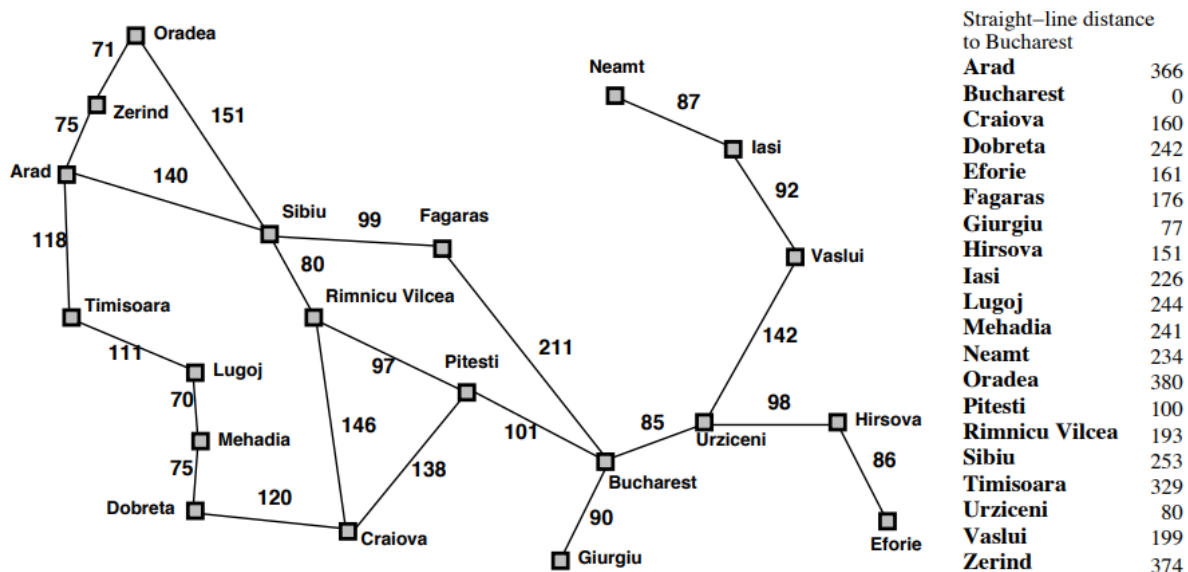
La funzione euristica si dice ammissibile se il valore da essa stimato è minore o uguale al costo reale per raggiungere la soluzione a partire dal nodo che stiamo considerando.

$$h(n) \leq \text{costo reale per raggiungere l'obiettivo da } n$$

Dunque, un'euristica è ammissibile quando l'errore di stima non è mai in eccesso.

3.3 Mostrare un esempio di applicazione di A* su un problema di navigazione stradale

Si utilizza la mappa della Romania per applicare l'algoritmo A* nella risoluzione del problema di navigazione stradale tra Oradea e Bucharest.



Si rappresenta tramite un grafo la mappa della Romania

Si rappresenta il problema creando un'astrazione della mappa della Romania con l'utilizzo di un grafo per far sì che sia facilmente rappresentabile tramite l'algoritmo A*. La rappresentazione a grafo sarà una sequenza di archi che porta dallo stato iniziale ad uno stato obiettivo. L'insieme delle possibili soluzioni comprende tutti i possibili percorsi, giusti o sbagliati sul grafo (senza vincoli sul punto di partenza e sul punto di arrivo).

La scelta dell'euristica ammissibile

Si tratta di un criterio approssimativo per trovare una soluzione "accettabile" per il nostro problema. Nel nostro caso utilizziamo l'euristica della **distanza a linea d'aria** di ogni città della Romania verso Bucharest, indicato nella tabella a destra della mappa. Con una buona funzione euristica la complessità può essere ridotta considerevolmente: l'entità precisa dipende dalla natura del problema e dalla quantità dell'euristica.

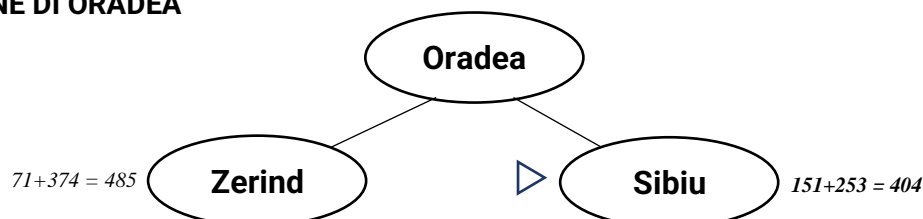
Sviluppo dell'albero di ricerca tramite algoritmo A*

STATO INIZIALE



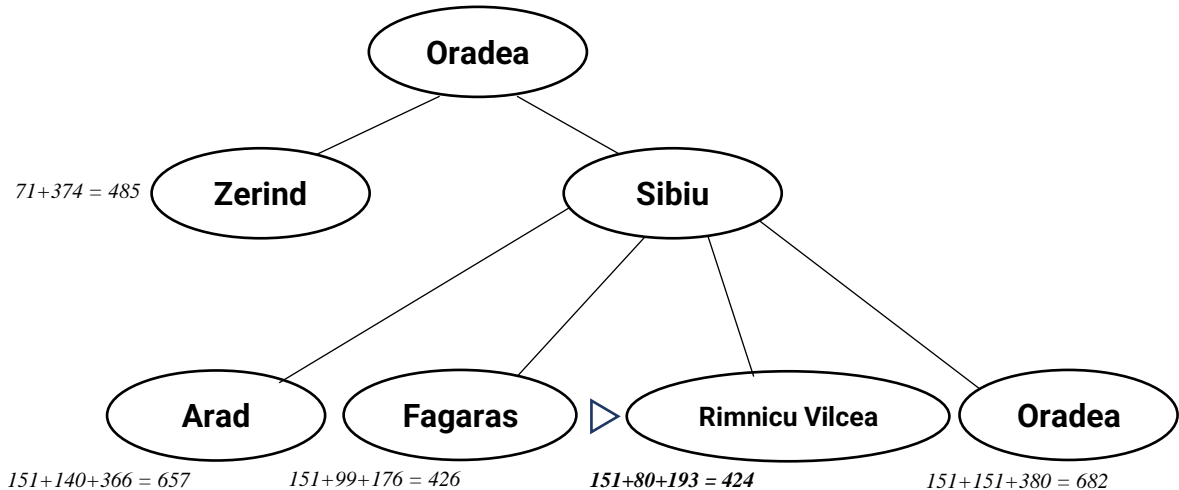
Lo stato iniziale rappresentato dalla radice del grafo è **Oradea**, il nostro punto di partenza. La funzione $f(n)$ sarà uguale a 380 dato che è presente solo la componente $h(n)$ indicante la distanza in linea d'aria da Bucharest, mentre il costo storico $g(n)$ è zero essendo il nodo radice.

ESPANSIONE DI ORADEA



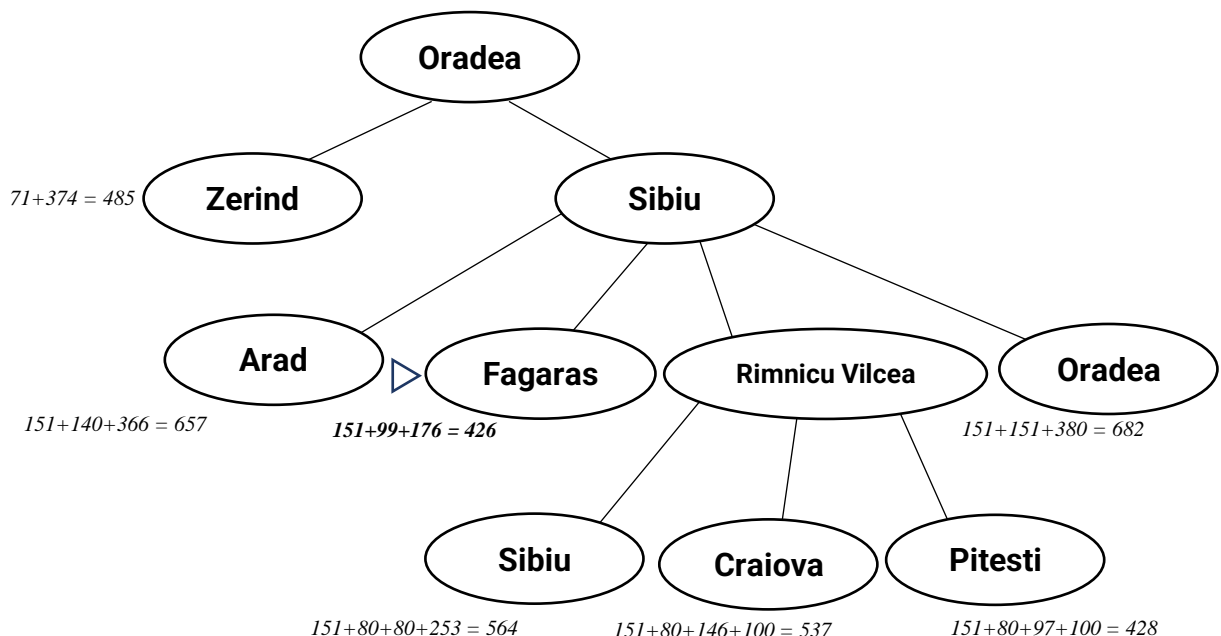
All'espansione di Oradea, la funzione $f(n)$ sarà costruita per i corrispettivi sottoalberi. Il prossimo nodo da espandere sarà **Sibiu** dato che ha $f(n)$ minore rispetto a quella di Zerind. Infatti, avremo che la $f(n)$ sarà data dalla somma di 151 che rappresenta la distanza Oradea-Sibiu percorsa e quindi il costo storico, mentre 253 è la distanza in linea d'aria Sibiu-Bucharest.

DOPO ESPANSIONE DI SIBIU



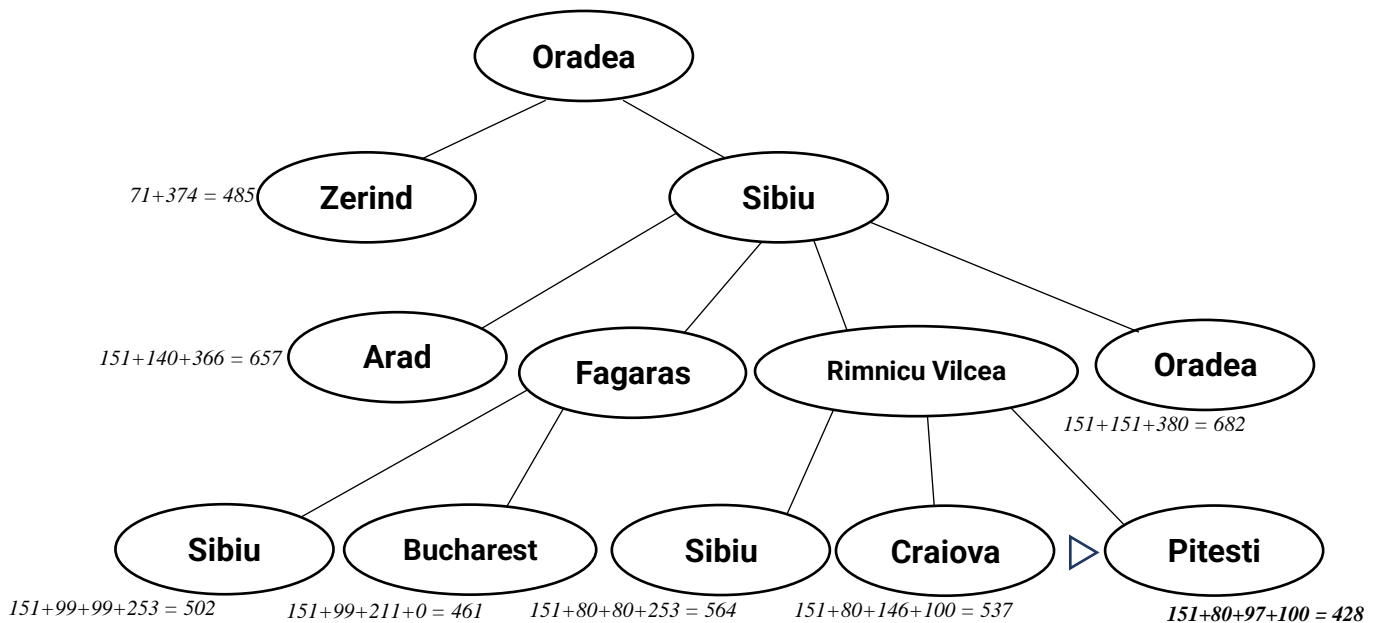
All'espansione di Sibiu si procede in modo analogo al passo precedente costruendo la funzione $f(n)$ per i rispettivi sottoalberi. La scelta questa volta ricade su **Rimnicu Vilcea** dato che ha $f(n)$ minore.

DOPO ESPANSIONE DI RIMNICU VILCEA



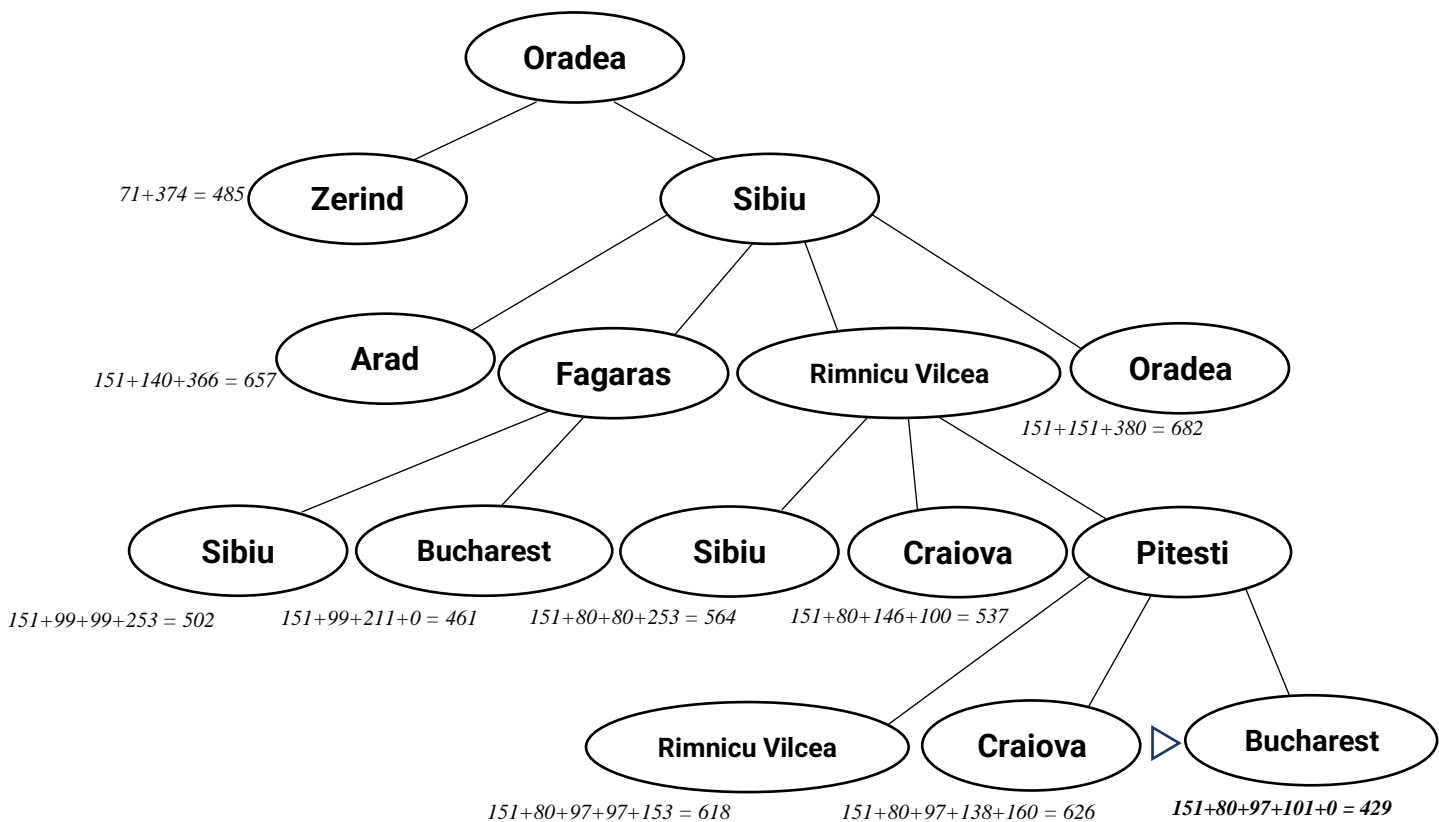
Con l'espansione di Rimnicu Vilcea, saremmo portati ad espandere Pitesti, questo però si rivela un vicolo cieco. Infatti, applicando l'algoritmo A* esso ci dice che il nodo precedente Fagaras (appartenente alla frontiera), ha $f(n)$ minore rispetto a questo. L'algoritmo allora decide di tornare indietro e espandere **Fagaras**.

DOPO ESPANSIONE DI FAGARAS



A questo punto con l'espansione di Fagaras e avendo ottenuto la funzione costruita per i sottoalberi, otteniamo il nodo Bucharest ma l'algoritmo A* ci dice che la soluzione non è ottimale, perché il nodo **Pitesti** ha una $f(n)$ minore, allora decidiamo di espandere Pitesti nell'altro sottoalbero.

DOPO ESPANSIONE DI PITESTI



Arrivati qui, all'espansione di Pitesti otteniamo nuovamente Bucharest, questa volta l'A* ci dice che questa è una soluzione ottimale, perché il nodo **Bucharest** trovato ha la $f(n)$ minore tra tutti i nodi nella frontiera.

ESERCIZIO 4

4.1 Descrivere con un esempio come si utilizza un albero di decisione

All'interno del Machine Learning la rappresentazione della conoscenza avviene con l'utilizzo degli alberi di decisione (o alberi decisionali). Un albero di decisione è un sistema con n variabili in input e m variabili in output. Le variabili in input, gli **attributi**, rappresentano le proprietà dell'ambiente che si ottengono dalla sua osservazione. Le variabili in output, invece, identificano la **decisione/azione** da intraprendere in relazione alla combinazione di valori degli attributi.

Il processo decisionale è rappresentato con un albero logico rovesciato dove ogni nodo è una funzione condizionale.

Ogni nodo verifica una condizione (**test**) su una particolare proprietà dell'ambiente (**variabile**) e ha due o più diramazioni verso il basso in funzione della condizione. Il processo decisionale consiste in una sequenza di test sui nodi dell'albero. Si comincia sempre dal nodo radice, ovvero il nodo genitore situato più in alto nella struttura, e procede verso il basso.

A seconda dei valori rilevati in ciascun nodo, il flusso prende una direzione oppure un'altra e procede progressivamente verso il basso, anche in base all'euristica adottata nella costruzione dell'albero.

La decisione finale si trova nei nodi foglia terminali, quelli più in basso. In questo modo, dopo aver analizzato le varie condizioni, l'agente giunge alla decisione finale.

Per la realizzazione dell'albero segue il concetto di scelta del migliore attributo.

Meta-Algoritmo DTL

```

funzione DTL (esempi, attributi, default) ritorna un albero
  SE non ci sono esempi allora ritorna default
  SE tutti gli esempi hanno la stessa classificazione ritorna la classificazione
  SE non ci sono attributi ritorna la Moda(esempi)
  ALTRIMENTI
    Sia best ← Choose-Attribute (attributi, esempi)
    Sia albero ← un nuovo albero di decisione con radice di test best
    PER OGNI valore  $v_i$  di best FAI
      esempii ← {elementi di esempi con best =  $v_i$ }
      sotto-albero ← DTL (esempii, attributi-best, Mode(esempi))
    AGGIUNGI un ramo ad albero con etichetta (best=  $v_i$ ) e sottoalbero sotto-albero
  RITORNA albero
  
```

Questa funzione è ricorsiva, perché l'albero viene costruito chiamando per ogni sotto-albero la funzione DTL. Questo algoritmo riceve in ingresso gli esempi, gli attributi e una classificazione di default. Gli esempi sono le righe della tabella dei sistemi di apprendimento mentre gli attributi sono le colonne.

- Se l'insieme degli esempi è vuoto, allora restituisce la classificazione di default.
- Se tutti gli esempi hanno la stessa classificazione, allora restituisce la classificazione stessa.
- Se l'insieme degli attributi è vuoto, allora restituisce la moda degli esempi, che conta la classe degli esempi e ritorna la classe che occorre maggiormente.
- Negli altri casi, scegli un attributo, crea un nuovo albero di decisione avente come radice l'attributo scelto, e per ogni valore che l'attributo può assumere, riduci l'insieme degli esempi, creando un sottoinsieme che contenga solo gli esempi per cui l'attributo vale l' i -esimo valore.

In questo modo, sto partizionando l'insieme degli esempi sulla base del valore assunto dall'attributo. Si crea un sottoalbero, passando alla funzione DTL, chiamata per ricorsione, il sottoinsieme degli esempi precedentemente determinato, l'insieme degli attributi eccetto quello scelto, e la moda degli esempi come classificazione di default. Infine, il sottoalbero viene collegato all'albero principale.

Questo risulta essere un meta-algoritmo, in quanto l'algoritmo non è scritto in un linguaggio di programmazione specifico, ma viene solo definita la struttura dell'algoritmo.

Analizziamo il problema di dover decidere se attendere al di fuori di un ristorante oppure cambiare locale, per risolvere questo problema utilizzeremo la tabella degli esempi e realizzeremo l'albero. (Sarà spiegato al meglio nel prossimo esercizio).

Con l'albero costruito seguendo la funzione DTL, otteniamo una schematizzazione ripartitiva che ci permette di stabilire una regola generale che permette all'agente intelligente di prendere una decisione, come quella dell'attesa al ristorante.

ESERCIZIO 5

5.1 Descrivere con un esempio il processo di apprendimento di un albero di decisione

Si prova a costruire un albero per decidere se aspettare che si liberi un tavolo al ristorante oppure no.

Per fare ciò come detto precedentemente abbiamo bisogno di una "Tabella dei sistemi di apprendimenti" dove andremo ad inserire i vari esempi su cui ci baseremo per la realizzazione dell'albero.

Nel nostro caso il target della tabella sarà l'attendere o meno un tavolo.

ATTRIBUTI		TARGET
Type: Qual è il genere di cucina?	<i>French, Italian, Thai, Burger</i>	WillWait
Friday/Sat: È il fine settimana?	<i>Yes, No</i>	
Hungry: Si è affamati?	<i>Yes, No</i>	
Price: Il ristorante è costoso?	<i>\$\$\$, \$\$, \$</i>	

Un albero di decisione booleano è una coppia (X, y) dove:

- X è un vettore di valori per gli attributi
- $y \in \{\text{True}, \text{false}\}$

EX	Type	Friday/Sat	Hun	Price	WillWait
x1	French	False	True	\$\$\$	y1=True
x2	Thai	False	True	\$	y2=False
x3	Burger	False	False	\$	y3=True
x4	Thai	True	True	\$	y4=True
x5	French	True	False	\$\$\$	y5=False
x6	Italian	False	True	\$\$	y6=True

Quale attributo utilizziamo prima per effettuare la decisione?

Per trovare l'attributo più discriminante veniamo in contro alla funzione **entropia**, che ci permette di calcolare il grado di disordine all'interno di un sistema e quindi il migliore attributo da espandere.

La funzione di entropia è una rappresentazione binaria e può assumere valori da 0 a 1 ed ha una forma:

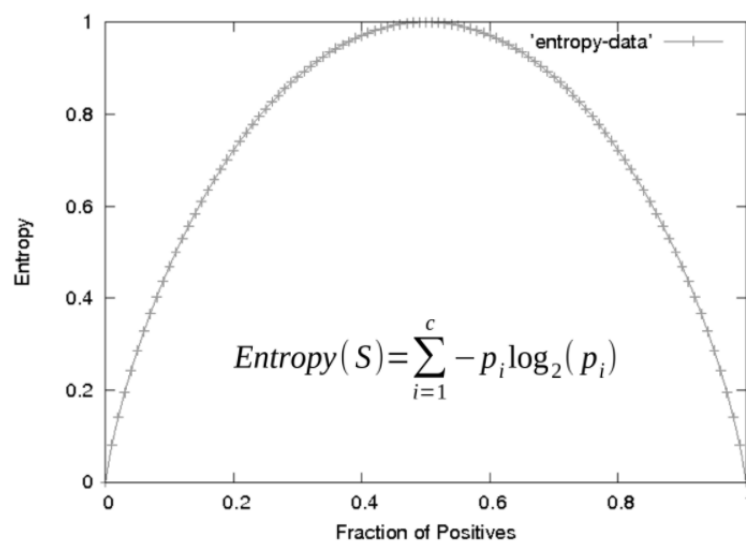


Figura 14: Funzione di entropia per classificazione binaria

CALCOLO DELL'ENTROPIA DEI SINGOLI ATTRIBUTI

Il calcolo si suddivide in due logaritmi in base due; il primo individua i **false**, il secondo individua i **true**.

Type (French, Thai, Burger, Italian)

$$\rightarrow \text{French: } \frac{2}{6} \left(-\frac{1}{2} \log_2 \left(\frac{1}{2} \right) - \frac{1}{2} \log_2 \left(\frac{1}{2} \right) \right) = 0.33(-0.5(-1) - 0.5(-1)) \cong 0.33$$

$$\rightarrow \text{Thai: } \frac{2}{6} \left(-\frac{1}{2} \log_2 \left(\frac{1}{2} \right) - \frac{1}{2} \log_2 \left(\frac{1}{2} \right) \right) = 0.33(-0.5(-1) - 0.5(-1)) \cong 0.33$$

$$\rightarrow \text{Burger: } \frac{1}{6} \left(-\frac{1}{1} \log_2 \left(\frac{1}{1} \right) \right) = 0 \text{ True}$$

$$\rightarrow \text{Italian: } \frac{1}{6} \left(-\frac{1}{1} \log_2 \left(\frac{1}{1} \right) \right) = 0 \text{ True}$$

$$\text{Entropy(Type)} = 0.33 + 0.33 = 0.66$$

Friday/Sat (False, True)

$$\rightarrow \text{False: } \frac{4}{6} \left(-\frac{1}{4} \log_2 \left(\frac{1}{4} \right) - \frac{3}{4} \log_2 \left(\frac{3}{4} \right) \right) = 0.67(-0.25(-2) - 0.75(-0.41)) \cong 0.54$$

$$\rightarrow \text{True: } \frac{2}{6} \left(-\frac{1}{2} \log_2 \left(\frac{1}{2} \right) - \frac{1}{2} \log_2 \left(\frac{1}{2} \right) \right) = 0.33(-0.5(-1) - 0.5(-1)) \cong 0.33$$

$$\text{Entropy(Type)} = 0.47 + 0.33 = 0.87$$

Hun (False, True)

$$\rightarrow \text{True: } \frac{4}{6} \left(-\frac{1}{4} \log_2 \left(\frac{1}{4} \right) - \frac{3}{4} \log_2 \left(\frac{3}{4} \right) \right) = 0.67(-0.25(-2) - 0.75(-0.41)) \cong 0.54$$

$$\rightarrow \text{False: } \frac{2}{6} \left(-\frac{1}{2} \log_2 \left(\frac{1}{2} \right) - \frac{1}{2} \log_2 \left(\frac{1}{2} \right) \right) = 0.33(-0.5(-1) - 0.5(-1)) \cong 0.33$$

$$\text{Entropy(Type)} = 0.47 + 0.33 = 0.87$$

Price (\$\$\$, \$\$, \$)

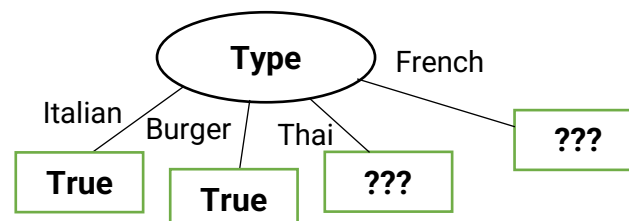
$$\rightarrow \text{$$$: } \frac{2}{6} \left(-\frac{1}{2} \log_2 \left(\frac{1}{2} \right) - \frac{1}{2} \log_2 \left(\frac{1}{2} \right) \right) = 0.33(-0.5(-1) - 0.5(-1)) \cong 0.33$$

$$\rightarrow \text{\$: } \frac{3}{6} \left(-\frac{1}{3} \log_2 \left(\frac{1}{3} \right) - \frac{2}{3} \log_2 \left(\frac{2}{3} \right) \right) = 0.5(-0.33(-1.59) - 0.67(-0.58)) \cong 0.44$$

$$\rightarrow \text{\$: } \frac{1}{6} \left(-\frac{1}{1} \log_2 \left(\frac{1}{1} \right) \right) = 0$$

$$\text{Entropy(Type)} = 0.46 + 0.33 = 0.77$$

Dunque, il nodo radice sarà:



Si continuano i calcoli estrapolando il sottoinsieme **TYPE = THAI** degli attributi:

EX	Friday/Sat	Hun	Price	WillWait
x2	False	True	\$	y2=False
x4	True	True	\$	y4=True

I calcoli saranno:

Friday/Sat (False, True)

$$\rightarrow \text{False: } \frac{1}{2}(-1\log_2(1) - 0\log_2(0)) = 0$$

$$\rightarrow \text{True: } \frac{1}{2}(-0\log_2(0) - 1\log_2(1)) = 0$$

$$\text{Entropy}(\text{Friday/Sat}) = 0$$

Hun (True)

$$\rightarrow \text{True: } 1\left(-\frac{1}{2}\log_2\left(\frac{1}{2}\right) - \frac{1}{2}\log_2\left(\frac{1}{2}\right)\right) = 1$$

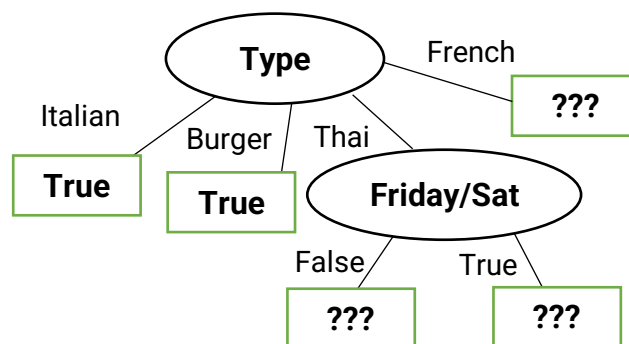
$$\text{Entropy}(\text{Hun}) = 1$$

Price (\$)

$$\rightarrow \$: 1\left(-\frac{1}{2}\log_2\left(\frac{1}{2}\right) - \frac{1}{2}\log_2\left(\frac{1}{2}\right)\right) = 1$$

$$\text{Entropy}(\text{Price}) = 1$$

Dunque, l'albero al passo successivo sarà il seguente:



Si continuano i calcoli estrapolando il sottoinsieme **TYPE = FRENCH** degli attributi:

EX	Friday/Sat	Hun	Price	WillWait
x1	False	True	\$\$\$	T
x5	True	False	\$\$\$	F

Friday/Sat (False, True)

$$\rightarrow \text{False: } \frac{1}{2}(-0\log_2(0) - 1\log_2(1)) = 0$$

$$\rightarrow \text{True: } \frac{1}{2}(-1\log_2(1) - 0\log_2(0)) = 0$$

$$\text{Entropy}(\text{Friday/Sat}) = 0$$

Hun (False, True)

$$\rightarrow \text{False: } \frac{1}{2}(-0\log_2(0) - 1\log_2(1)) = 0$$

$$\rightarrow \text{True: } \frac{1}{2}(-1\log_2(1) - 0\log_2(0)) = 0$$

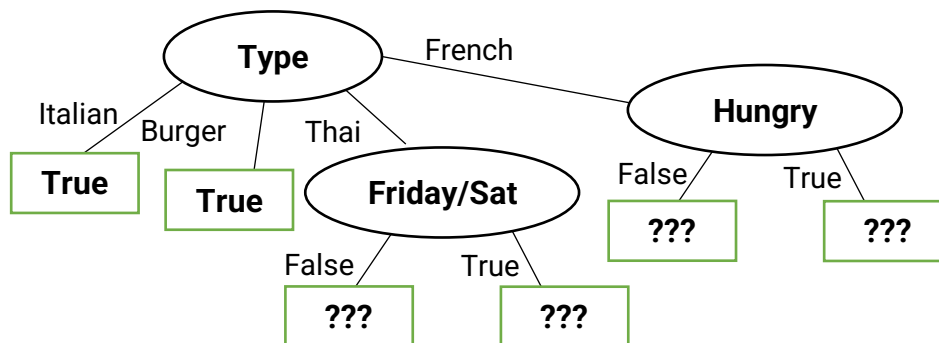
$$\text{Entropy}(\text{Hun}) = 0$$

Price (\$)

$$\rightarrow \$: 1\left(-\frac{1}{2}\log_2\left(\frac{1}{2}\right) - \frac{1}{2}\log_2\left(\frac{1}{2}\right)\right) = 1$$

$$\text{Entropy}(\text{Price}) = 1$$

Dunque, l'albero al passo successivo sarà il seguente:



Seguono gli altri sottoinsiemi:

Si estrapola il sottoinsieme **FRI/SAT=FALSE** e restituisce **FALSE**

EX	Hun	Price	WillWait
x2	True	\$	y2=False

Si estrapola il sottoinsieme **FRI/SAT=TRUE** e restituisce **TRUE**

EX	Hun	Price	WillWait
x4	True	\$	y4=True

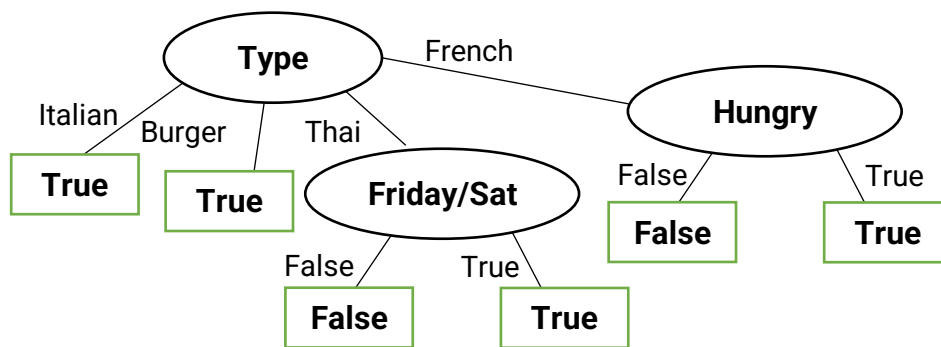
Si estrapola il sottoinsieme **HUN=TRUE** e restituisce **TRUE**

EX	Friday/Sat	Price	WillWait
x1	False	\$\$\$	y1=True

Si estrapola il sottoinsieme **HUN=FALSE** e restituisce **FALSE**

EX	Friday/Sat	Price	WillWait
X5	True	\$\$\$	y5=False

In definitiva l'albero di decisione per la tabella di apprendimento fornita è questo:



ESERCIZIO 6

6.1 Cosa sono: lo spazio degli stati, il grafo degli stati e l'albero di ricerca (Tree Search)

Un primo esempio di astrazione del problema da risolvere è un **grafo degli stati**, ovvero un tipo di diagramma orientato che descrive il comportamento di un problema attraverso nodi (stato) e frecce (collegamenti).

L'obiettivo del Problem Solving è la ricerca dello spazio degli stati il cui risultato è una sequenza di azioni che intercorre tra lo stato iniziale a quello finale (**soluzione del problema**).

Lo spazio degli stati è l'insieme di tutti gli stati che sono raggiungibili dallo stato iniziale attraverso sequenza di azioni (cammino) determinati dall'algoritmo di ricerca (**albero di ricerca**).

Il **Tree Search** è la rappresentazione delle sequenze di azioni che un determinato agente effettua a partire da uno stato iniziale (il nodo padre). Dal nodo padre sono collegati gli stati conseguenti per ogni azione possibile. Ogni azione possibile viene rappresentata tramite un collegamento che unisce il nodo padre ed il nodo figlio (i nodi dell'albero di ricerca sono gli stessi dello spazio degli stati). Dal nodo figlio possono espandersi nuovi nodi sulla base dell'esigenza del problema (espansione) ed un nodo senza ulteriori azioni possibili è detto nodo foglia e contribuisce la frontiera degli stati.

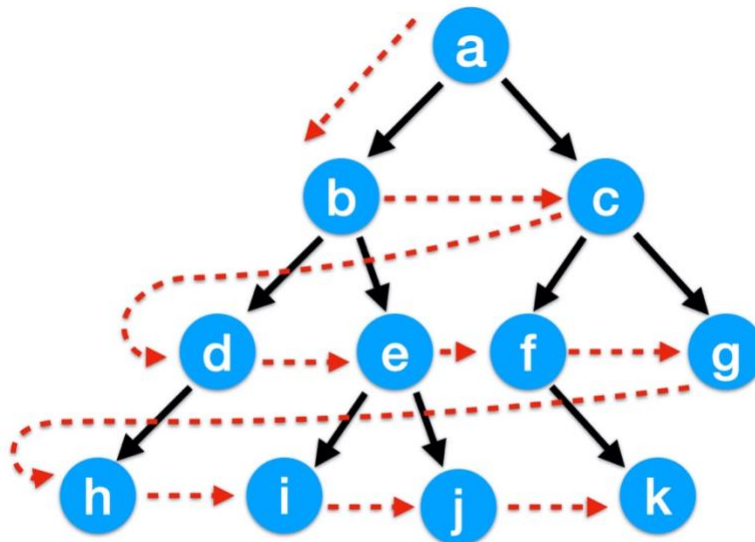
6.2 Descrivere in modo sintetico il meta algoritmo generale di costruzione di un Tree Search

Per costruire un Tree Search, si segue un algoritmo al quale vengono forniti in input un **problema** ed una **frontiera** (ovvero l'insieme dei nodi da espandere) e che restituisce in output l'albero di ricerca. Si inizia inserendo nella frontiera il nodo iniziale del problema, poi si esegue un ciclo infinito in cui si verifica se la frontiera è vuota, se lo è ritorna un fallimento. Altrimenti preleva un nodo dalla frontiera e testa se è un nodo obiettivo del problema e quindi, se lo è ritorna il nodo in questione. Se il nodo non è obiettivo, si espande tale nodo con una funzione **EXPAND()** e lo si inserisce nella frontiera, il ciclo può così iniziare da capo. La funzione **EXPAND()** ritorna un insieme di nodi, si inizializza un insieme vuoto in cui andranno inseriti i **nodi "successori"** di quello fornito in input, inoltre, viene fornito anche il problema alla funzione. Si esegue un ciclo per ciascun stato raggiungibile da quello attuale determinato dalla **funzione successore $S(x)$** , si crea un nodo per quello stato, si aggiungono a tale nodo tutte le informazioni che lo caratterizzano (nodo padre, azione, ovviamente lo stato che individua, il costo di cammino e la profondità). Si aggiunge tale nodo all'insieme dei nodi "successori", si verifica se ci sono altri stati raggiungibili e in caso affermativo si ripete il ciclo, altrimenti si ritorna alla funzione di costruzione del Tree Search l'insieme dei nodi espansi.

6.3 Spiegare in modo sintetico le caratteristiche degli algoritmi Tree Search visti a lezione

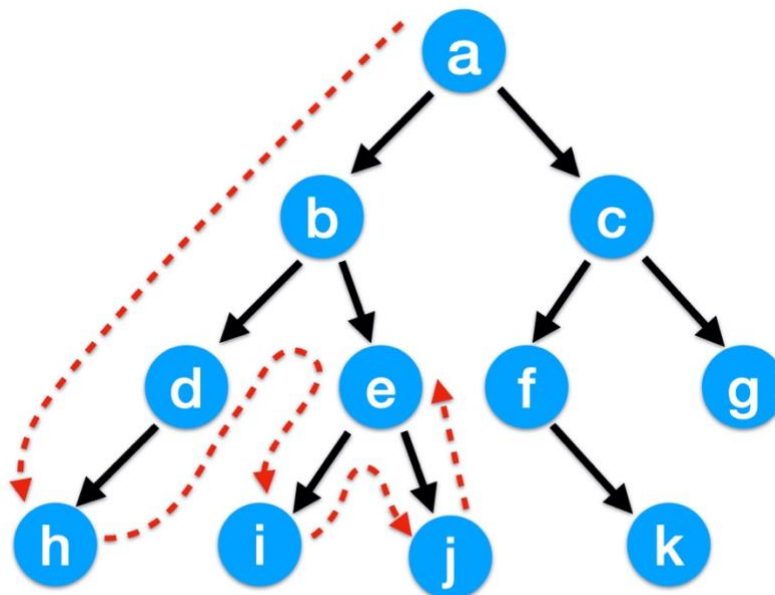
I problemi di intelligenza artificiale che riguardano il problem solving vengono sempre ricondotti a problemi di ricerca su alberi. Gli algoritmi di ricerca si dividono in due categorie, quelli **non informati** (Breadth-first search, Depth-first Search e Uniform-cost search) e quelli **informati** (Best-first search, che si specializza in due varianti Greedy Search e A* Search).

La prima categoria di algoritmi, come dice il nome, non sfruttano nessuna informazione sul numero di passi o sul costo di cammino dallo stato attuale a quello obiettivo, iniziamo con il **Breadth-first search** che effettua una ricerca in ampiezza, i nodi della frontiera vengono inseriti in una **coda gestita tramite politica FSEO**. Prendendo come esempio l'albero in figura, all'inizio nella coda viene inserito il nodo a, poi questo nodo viene prelevato dalla coda ed espanso, poi vengono aggiunti i nodi b e c, essendo che b viene inserito per primo nella coda sarà anche il primo ad essere espanso tra i due, e così via. Quest'algoritmo è completo (perché trova una soluzione se esiste), è ottimale se il costo di cammino è 1 per ogni step perché la soluzione trovata è quella a profondità minore, tuttavia in generale non è ottimale.



Breadth-first search

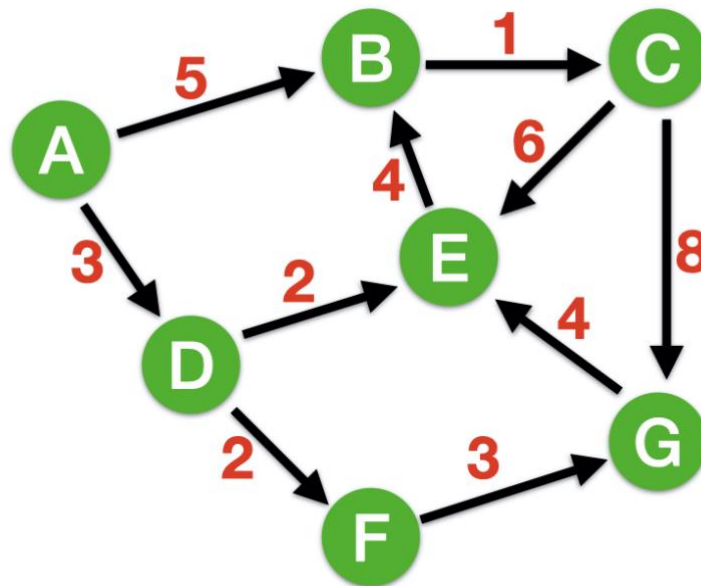
Il secondo algoritmo della categoria è il **Depth-first search**, esso memorizza i nodi della frontiera in uno **stack gestito con politica LSEO**, questa ricerca viene anche detta in profondità, usando sempre l'esempio in figura abbiamo che il nodo a viene espanso allora nello stack vengono inseriti nell'ordine c e b, a questo punto l'ultimo nodo inserito è b. Viene prelevato b, quindi viene espanso e vengono inseriti nello stack i nodi e ed d, e così via. Quest'algoritmo non è né completo né ottimale, perché se ci sono dei cicli nell'albero esso lo percorre all'infinito, inoltre non trova la soluzione migliore, ma la prima soluzione che riesce a calcolare.



Depth-first search

L'ultimo algoritmo tra quelli non informati è il **Uniform-cost search**, che dispone i nodi della frontiera all'interno di una **coda ordinata in maniera crescente sulla base del costo di raggiungimento di un nodo**, anche se appartiene agli algoritmi non informati, in realtà porta con sé un'informazione

maggiore rispetto ai precedenti rappresentata dal costo di cammino. Un esempio di applicazione è mostrato in figura, dove si nota che il percorso di ricerca su quest'albero segue i percorsi con costo minore.



SOLUTION

Explored : A D B E F C

Path : A to D to F to G

Cost = 8

Passiamo all'altra categoria di algoritmi di ricerca, ovvero quelli informati, per capire questi algoritmi dobbiamo introdurre il concetto di euristica, ovvero una regola empirica che ci permettono di trovare una buona soluzione al problema senza dimostrarlo analiticamente. Tra gli algoritmi informati, quello più importante è il **Best-first search**, i nodi della frontiera vengono ordinati secondo un **criterio di desiderabilità** (quello più desiderabile è quello che più si avvicina alla soluzione), questa è l'euristica su cui si basa l'algoritmo, che poi può essere implementata con diverse strategie, tra cui il Greedy Search e A* Search.

Per implementare il **Greedy Search (ricerca "avida")** è necessaria una funzione di desiderabilità, che può essere ad esempio, la distanza in linea d'aria con una città obiettivo nel caso della navigazione stradale. Quindi la scelta del nodo della frontiera da espandere viene fatta sulla base del valore di questa euristica $h(n)$ prendendo quella con il valore minore. Quest'algoritmo è completo, se gestisco i cicli all'interno dell'albero, ma non è ottimale, tuttavia, trova una soluzione sub-ottimale (massimo locale).

L'algoritmo **A* Search** si basa non solo su un'euristica ma anche su un costo storico, infatti la funzione di desiderabilità è ottenuta in questo modo: $f(n) = g(n) + h(n)$, in cui $g(n)$ è il costo passato per raggiungere il nodo n a partire dal nodo iniziale, mentre $h(n)$ è la stima di quanto sia distante n dallo stato obiettivo. Quest'algoritmo è ottimale se $h(n)$ è ammissibile cioè il costo stimato è minore o uguale a quello reale per raggiungere l'obiettivo a partire da n , mentre è completo se ci sono un numero finito di stati.

ESERCITAZIONE 2: OPTIMIZATION

ESERCIZIO 1

1.1 Introduzione agli algoritmi di ricerca locale

Gli **algoritmi di ricerca locale** ci permettono di costruire una soluzione senza interessarsi della sequenza dei passi (azione) a differenza di come avviene negli algoritmi di ricerca informata (Best first search). Sono in grado di iniziare a risolvere il problema con una soluzione casuale approssimata e la modifichiamo mentre eseguiamo l'algoritmo. Infatti, riescono a trovare soluzioni "ragionevoli" anche in spazi molto grandi e infiniti (spazio continuo). Offrono un'ottima gestione di efficienza di memoria grazie alla mancanza di tracciamento dei cammini. Algoritmo utilizzato quando abbiamo uno stato goal come soluzione del problema. Esistono diversi approcci di ricerca locale, uno di questi è l'**Hill Climbing**.

L'idea dell'Hill Climbing è quella di trovare una soluzione migliore "locale", ovvero supponendo il caso di una funzione continua (per il Th. Di Weierstrass, ammette massimi e minimi) e di cui vogliamo calcolare il massimo di questa funziona, calcoliamo la derivata e la poniamo uguale a 0. Se ci sono molti massimi locali, troviamo uno dei massimi. Lo spazio di ricerca dell'algoritmo Hill Climbing è limitato solo ai nodi vicini a quello corrente.

L'algoritmo **Simulated Annealing** sopperisce alla problematica dell'Hill Climbing (è una variazione dell'Hill Climbing), ovvero non si blocca al massimo locale, ma cerca di trovare il massimo globale (la migliore soluzione globale per risolvere il problema).

Il procedimento descritto dal meta algoritmo presenta le diverse differenza con l'Hill Climbing, infatti, viene aggiunto una ulteriore variabile chiamata "Temperature" che controllerà la distribuzione di probabilità e la variabile "next" che sostituirà "neighbor" dell'HC. Inizializzo il nodo con una possibile soluzione e assegno un valore di score, entro nel for (loop che parte da 1 e andrà all'infinito), assegno un valore alla Temperatura che decresce al progredire dell'algoritmo fino a quando $T = 0$ (solo in quel caso l'algoritmo termina). Genero dei successori allo stato corrente (creo ulteriori soluzioni successori alla soluzione precedente) e assegno anche ad esso degli score, se queste soluzioni hanno score maggiore ($\Delta E = \text{score}(\text{successore}) - \text{score}(\text{current}) \rightarrow \Delta E > 0$) allora passo a considerare quello con score maggiore, se invece ha uno score minore ($\Delta E < 0$) passo a considerare quello con lo score peggiore solo se la probabilità di $e^{(\Delta E/T)}$. Poiché lo spostamento da uno stato buono ad un qualsiasi stato che sia migliore e che sia il peggiore (Cosa non possibile con l'Hill Climbing) con una certa probabilità, se si fa girare il Simulated Annealing per un tempo lungo è possibile esplorare tutte le possibili soluzioni, questo meccanismo fa sì che l'algoritmo possa essere considerato **COMPLETO**, ma avere tempi lunghi di ricerca può essere uno svantaggio.

Esiste inoltre un particolare **algoritmo di ricerca locale** chiamato **Local Beam Search** in grado di effettuare una ricerca nello spazio delle soluzioni attraverso le **esecuzioni di più algoritmi in parallelo** in modo tale che ognuno di essi siano in grado di dare soluzioni differenti, in modo tale da confrontare le soluzioni trovate per fornire la soluzione migliore.

L'algoritmo di ricerca locale **Genetic Algorithm** è una variante del Local Beam Search e si basa sull'idea dell'evoluzione naturale di generazione dei microbi ad ognuno dei quali è associato un **valore di fitness**. Se il valore di fitness è alto, il microbo sopravvive, se il valore di fitness è basso, il microbo muore. Se sopravvive, si riprodurrà, incrociandosi con un altro microbo scambiandosi il

materiale genetico creando un microbo figlio. I figli, infatti, avranno una parte del DNA che proviene da uno e una parte dall'altro. I microbi figli continueranno la loro evoluzione proprio come quella dei genitori che verranno uccisi e il processo ripartirà da capo. (Cfr. **Es. 3.1**)

1.2 Ottimizzazione di una funzione

Si prova ad ottimizzare una funzione assegnata con una delle tecniche di ottimizzazione (Genetic Algorithm e Hill Climbing) utilizzando il codice riportato nel Capitolo 5 "Evolutionary Optimization".

La funzione che andremo ad analizzare sarà:

$$F(x) = \begin{cases} 10 & \text{se } x < 5.2 \\ x^2 & \text{se } 5.2 \leq x \leq 20 \\ \cos(x) + 160x & \text{se } x > 20 \end{cases}$$

con $x \in [-100, 100]$

Nel codice assegnato aggiungiamo la seguente funzione $F(x)$ che deve essere ottimizzata:

```
#Procedura che definisce la funzione da ottimizzare
# con gli algoritmi di ricerca locale
import math
def F(x):
    if(x<=5.2):
        return 10
    if((x>=5.2)and(x<=20)):
        return x*x
    if(x>20):
        return 160 * x + math.cos(x*180/math.pi)
```

Per poter determinare quando la soluzione migliora usiamo la funzione:

```
#Procedura che calcola il costo dell'algoritmo
# di ricerca per trovare il massimo locale
def cost_fun(sol):
    cost = minvalues
    for i in range(0, len(sol)):
        current=maxvalue-F(sol[i]);
        if current>cost:
            if current < 0:
                current=current*(-1)+100
            cost = current
    return cost
```

Inoltre, si definisce la funzione cost_max(sol) tale che se il valore del costo generato da una generazione all'altra sia non ammissibile per la soluzione ovvero un valore al di fuori dell'intervallo

$[-100,100]$, si dà un costo pari a 0, in modo tale che per la creazione della prossima generazione non verrà preso in considerazione.

```
#Procedura che definisce la funzione di costo
# che permette all'algorithm di trovare un massimo locale
def cost_max(sol):
    cost=[]
    for i in range(0, len(sol)):
        if sol[i]>100 or sol[i]<-100:
            cost.append(0)
        else:
            cost.append(F(sol[i]))
    return cost
```

In breve, l'algorithm inizialmente genera una popolazione iniziale, nel nostro caso si scelgono dei possibili valori che soddisfano le condizioni della $F(x)$, ne calcola i vari score e sceglie una élite di individui che massimizzano lo score.

Successivamente si effettua il crossover tra coppie di individui appartenenti all'élite. Entra in gioco la mutazione che in base ad una certa probabilità opera dei cambiamenti casuali sui "figli" nati dal crossover.

Si crea una funzione main per eseguire il codice:

```
if __name__ == '__main__':

    #Popolazione iniziale dei valori da assegnare alla funzione
    popolazione = [-60, 0, 5, 10, 15, 80, 100, 110]

    print('Popolazione iniziale: ' + str(popolazione))

    #Valori di ottimizzazione della funzione
    f = cost_max(popolazione)

    print('Optimization: ' + str(f))

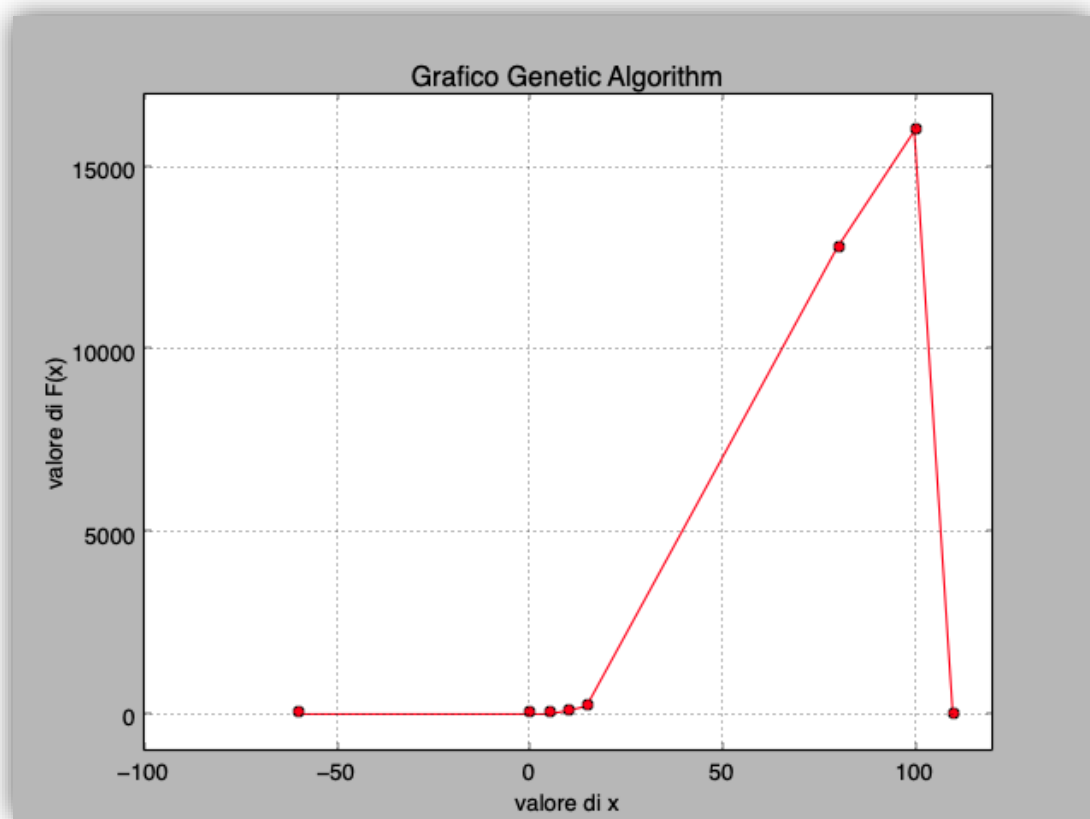
    #Calcoliamo il costo dell'algorithm data la popolazione
    costo_max = cost_fun(popolazione)

    print('Costo dell'algorithm: ' + str(costo_max))

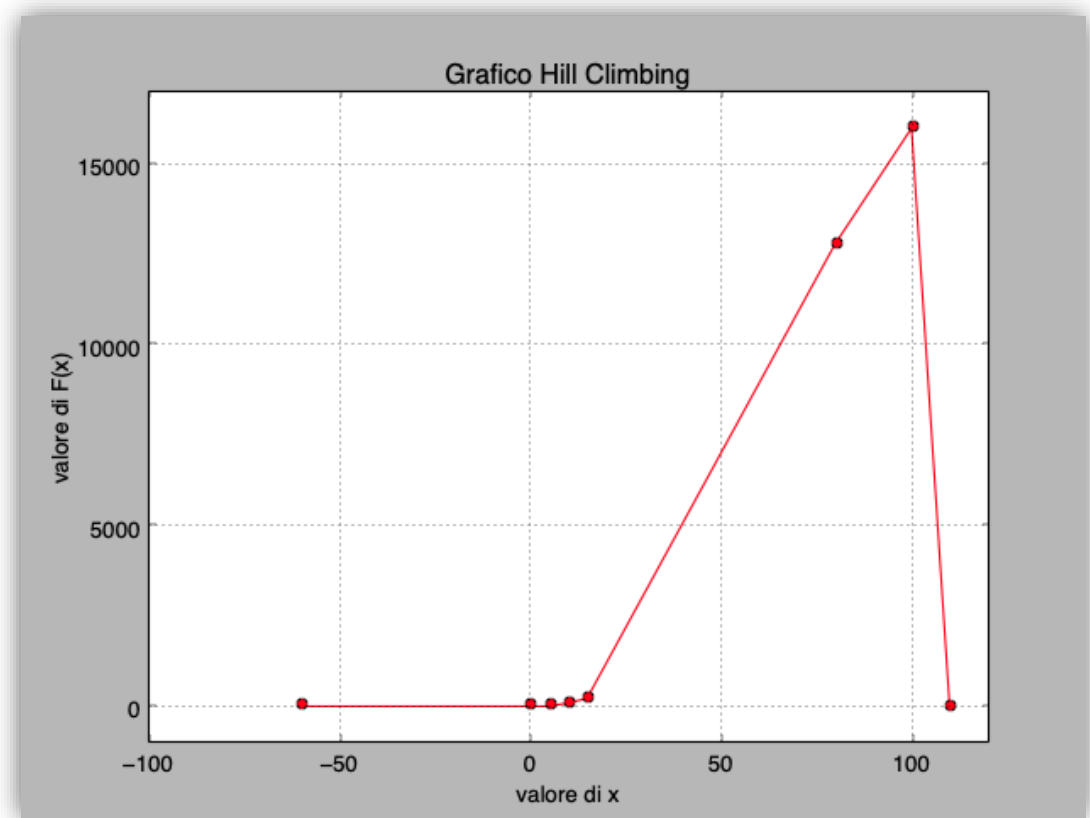
    #Funzione del Capitolo 5 che implementa l'algorithm genetico
    geneticalgorithm(popolazione, f)

    #Funzione del Capitolo 5 che implementa l'algorithm Hill Climbing
    hillclimb(popolazione, f)
```

Eseguendo il codice risulta che il max della funzione (ottimizzazione della $F(x)$) tende per $x = 100$ nel caso del Genetic Algorithm:



D'altra parte, provando ad utilizzare la funzione Hill Climbing si noterà che i risultati saranno gli stessi.



Il costo dell'algoritmo sarà:

```
Popolazione iniziale: [-60, 0, 5, 10, 15, 80, 100, 110]  
Optimization: [10, 10, 10, 100, 225, 12799.00309363223, 16000.773121177104, 0]  
Costo dello algoritmo: 90
```


ESERCIZIO 2

2.1 Un agente Intelligente deve colorare le provincie della Regione Campania evitando che due provincie confinanti abbiano lo stesso colore, l'Agente ha a disposizione solo 2 colori. Se il compito non è possibile spiegare perché non si riesce utilizzando gli algoritmi visti a lezione.

Per quanto riguarda il secondo esercizio di questa esercitazione abbiamo per le mani un classico problema di soddisfacimento dei vincoli (**CSP: Constraint Satisfaction Problems**), ovvero, un problema che viene risolto quando ogni variabile ha un valore che soddisfa tutti i vincoli.

Nel nostro caso le **variabili** sono le provincie della Regione Campania (Avellino, Benevento, Caserta, Napoli, Salerno).



Variabili = {AV, BE, CA, NA, SA}

I valori che queste variabili possono assumere provengono dal **dominio**. Nel nostro caso vedremo lo stesso esempio con tre domini diversi, domini che usano rispettivamente 2 valori, 3 valori e 4 valori (Per quanto riguarda l'esercizio assegnato i valori che le variabili possono assumere sono i colori).

Il vincolo da rispettare è che due provincie confinanti abbiano colori differenti.

Utili per la risoluzione di questo tipo di problemi sono i grafi dei vincoli (Constraint Graph) che permettono di analizzare più facilmente i vincoli tra le variabili di un problema. I nodi di questo grafo rappresentano le variabili e gli archi i vincoli tra di esse. Da quanto detto in precedenza discende il seguente grafo dei vincoli:



Analizziamo adesso il caso in cui viene utilizzato, come dominio dei valori, il dominio 1 (**D1**) che possiede solamente due valori:

D1 = {rosso, verde}

Dal grafo dei vincoli si nota subito come tutte le province confinano con almeno altre due province, questo ci suggerisce che con due singoli valori non riusciamo a risolvere il problema poiché un dominio formato da due soli valori non ci permetterà mai di soddisfare un vincolo tra tre variabili, di conseguenza il problema è irrisolvibile. Alla stessa conclusione si arriva anche sfruttando l'algoritmo studiato durante il corso il "Backtracking Search".

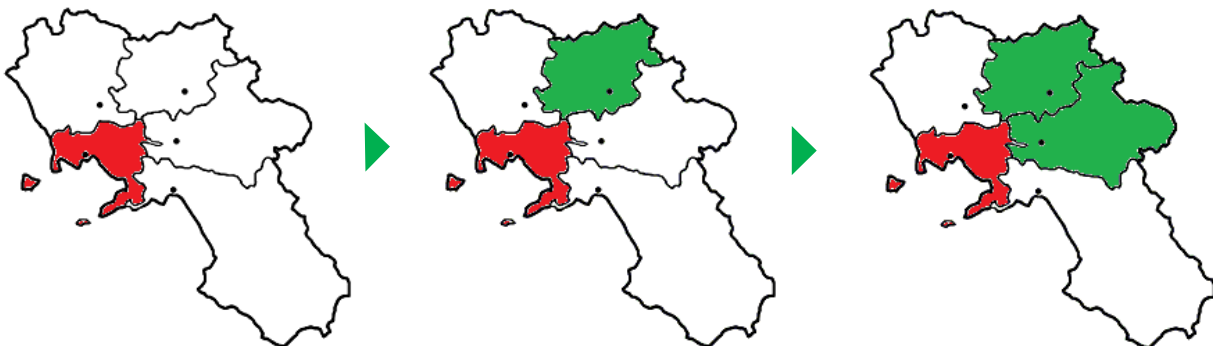
Il Backtracking Search è un algoritmo ricorsivo depth-first (che si espande in profondità).

Questo algoritmo seleziona una variabile a cui non è stata ancora assegnato uno stato e prova ad assegnargli ogni possibile valore presente all'interno del dominio dei valori senza preoccuparsi inizialmente dei vincoli. Se il valore è consistente per il problema dato i vincoli allora lo aggiungo alla variabile se invece viene individuata un'inconsistenza allora la funzione restituisce un fallimento facendo sì che la chiamata precedente della funzione provi un altro valore.

Per migliorare l'efficienza nella scelta di quale variabile deve essere la prossima a cui assegnare uno stato si possono usare diverse euristiche. In questo esercizio è stata usata la **Degree Heuristic** (Andiamo a scegliere la variabile che impatta maggiormente sulle restanti variabili).

Nel nostro caso scegliamo la variabile **NA**(Napoli) e gli assegniamo il valore **rosso**, successivamente se procediamo assegnando il valore **verde** alla variabile **BE**(Benevento) andiamo a ridurre ulteriormente il numero di valori legali per le province confinanti, in particolare la variabile **CA**(Caserta) e la variabile **AV**(Avellino) non avranno più a disposizione nessun valore ammissibile, quindi, procedendo con l'algoritmo otterremo un fallimento.

(Questo potrebbe essere uno dei risultati ottenuti non rispettando i vincoli).



Quanto notato analizzando il grafo dei vincoli coincide con il risultato ottenuto sfruttando l'algoritmo visto a lezione possiamo quindi concludere che il problema non è risolvibile utilizzando i valori del dominio D1.

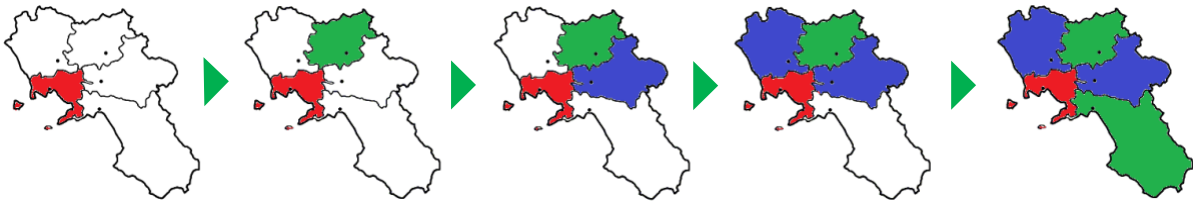
2.2 Ripetere il punto 1 con 3 colori a disposizione.

Passiamo adesso al caso in cui si usa il dominio 2 (**D2**) che possiede tre valori.

D2 = {rosso, verde, blu}

Usiamo ancora la **Degree Heuristic** quindi partiamo dalla variabile **NA**(Napoli) assegnando il valore **rosso**. Tra le restanti province scegliamo quella con il maggior numero di vincoli ovvero Benevento. La variabile **BE** (Benevento) ha solamente due valori ammissibili (**verde** e **blu**), andiamo quindi a colorare Benevento di **verde**. La prossima variabile da colorare è **AV** (Avellino) che ha un singolo

valore legale: il **blu**. Infine, abbiamo le due restanti variabili CA (Caserta) e SA (Salerno) che hanno entrambe disponibili solo un valore ammissibile rispettivamente Caserta **blu** e Salerno **verde**.



Osserviamo che ogni provincia ha un colore diverso dalle confinanti. Possiamo quindi affermare che è possibile risolvere il problema sfruttando il dominio **D2**.

2.3 Ripetere il punto 1 con 4 colori a disposizione.

Per ultimo abbiamo il caso in cui si utilizza il dominio 3 (D3) che possiede quattro valori.

D3 = {rosso, verde, blu, giallo}

Questo ultimo esempio sarà sicuramente risolvibile poiché si è aumentato il numero di valori possibili lasciando invariati i vincoli.

Procediamo andando subito a colorare la variabile NA (Napoli) di **rosso** riducendo i valori ammissibili per le altre province confinanti a tre. La successiva variabile scelta è BE (Benevento) che coloriamo di **verde**. La prossima variabile è AV (Avelino) che ha due valori ammissibili (**blu** e **giallo**) in questo caso scegliamo **blue** come valore. Le restanti Variabili CA (Caserta) e SA (Salerno) hanno entrambe a disposizione due valori legali a differenza del caso precedente (Quello in cui è stato usato il dominio D2). Concludiamo assegnando il colore **verde** a Salerno e il colore **viola** per Caserta. (Usiamo ancora una volta la **Degree Heuristic**)



Notiamo ancora una volta che ogni provincia assume un colore diverso dalle province limitrofe. Affermiamo quindi con sicurezza che è possibile risolvere il problema anche usando i valori del dominio D3. La differenza principale sta nel numero possibile di soluzioni, infatti, con il dominio D3 questo valore sarà sicuramente maggiore rispetto al valore ottenuto considerando il dominio D2. Abbiamo quindi che lo spazio delle soluzioni ottenuto usando il dominio D2 è incluso dello spazio delle soluzioni ottenuto sfruttando il dominio D3.

ESERCIZIO 3

3.1 Come utilizzereste un Algoritmo Genetico per risolvere il punto 3) dell'Esercizio 2?

La tecnica utilizzata dagli algoritmi genetici è una tra le migliori tecniche di ottimizzazioni delle funzioni ed in particolare, essa è anche **robusta ed efficiente** in quanto permette di trovare soluzioni che convergano verso buoni massimi locali, ciò avviene attraverso l'esplorazione in parallelo di differenti percorsi. Per applicarlo al nostro caso di studio occorre procedere per passi successivi, innanzitutto si definiscono le sequenze di variabili che formeranno un individuo (ovvero un fascio di esplorazione dell'algoritmo) come:

$$Province = \{NA, CA, BE, AV, SA\}$$

$$Colori = \{R, V, B, G\}$$

All'inizio abbiamo dunque, un insieme di individui che formano la popolazione iniziale, ogni individuo avrà un valore di fitness. Questo valore è assegnato tramite la scelta di una funzione di costo, che nel nostro caso sarà **score_fitness = numero di vincoli soddisfatti**, dove per vincolo si intende la condizione che due province adiacenti non abbiano lo stesso colore, un vincolo può essere inoltre espresso in questa maniera:

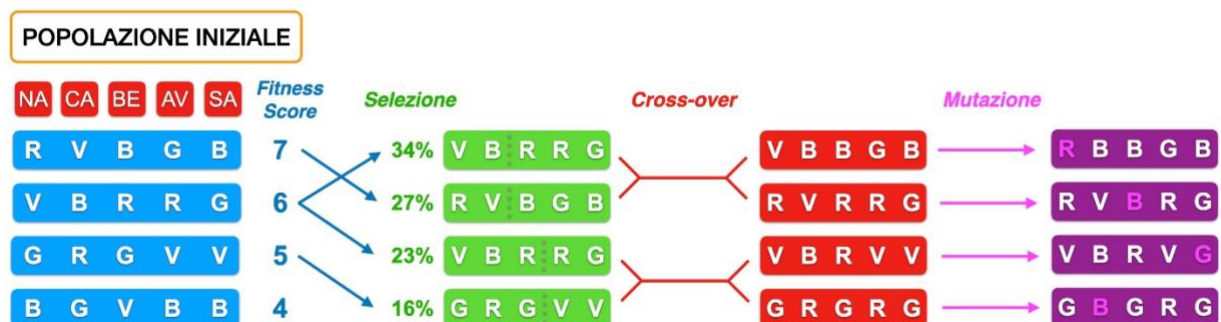
$$NA \neq CA \text{ oppure } (NA, CA) \in \{(R, V), (R, B), (R, G), (V, R), (V, B), (V, G), \dots\}$$

Verranno selezionati quindi, gli individui con un valore di fitness elevato e con una certa probabilità essi verranno accoppiati con altri individui, si parla in questo caso di **meeting pool**. Avviene poi lo scambio del materiale genetico (**cross-over**) in modo tale da ottenere i figli che hanno una parte proveniente da un genitore e una parte proveniente dall'altro genitore. Verifichiamo se i figli ottenuti soddisfano i vincoli assegnati sul problema, per poi passare all'operazione di mutazione che introduce dei cambiamenti casuali all'interno dell'individuo (in questo caso, si varia casualmente colore ad una provincia scegliendo tra i tre colori rimanenti, cioè escludendo il colore attualmente assegnatole).

Questo processo consente la creazione della nuova generazione, dopodiché il tutto si ripeterà ciclicamente fino a che non si trovi una soluzione ottimale, o una sub-ottima nel caso il problema sia troppo complesso.

Si mostra adesso, un esempio di applicazione dell'algoritmo genetico partendo da una popolazione iniziale del tutto casuale e scegliendo gli individui tra le $2^5 = 32$ possibili combinazioni con la funzione di fitness calcolata sui seguenti vincoli:

$$NA \neq CA, NA \neq BE, NA \neq AV, NA \neq SA, CA \neq BE, BE \neq AV, AV \neq SA$$



ESERCIZIO 4

Riassumere i concetti principali della logica proposizionale inclusi:

4.1 Cosa si intende per conseguenza logica

La conseguenza logica è un connettivo logico di implicazione logica. Considerata una base della conoscenza KB, si dice che KB implica la frase/formula α se e soltanto se α è vera in tutti i mondi (tutte le possibili situazioni in cui l'agente intelligente può venirsi a trovare) nel quale la KB è vera. Si scrive infatti $KB \models \alpha$.

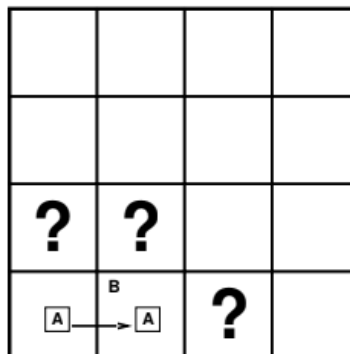
La conseguenza logica è alla base del ragionamento logico che permette agli agenti razionali di ottenere una rappresentazione della realtà a partire da un modello di riferimento (M) che sono mondi strutturati in maniera formale.

Se definiamo m come un modello di una frase/formula α , se α è vera nel modello m (definiamo $M(\alpha)$ il set di tutti i modelli di α). Allora possiamo dire che $KB \models \alpha \Leftrightarrow M(KB) \subseteq M(\alpha)$. Se vale questa inclusione vale anche l'implicazione logica dimostrabile attraverso il Model Checking.

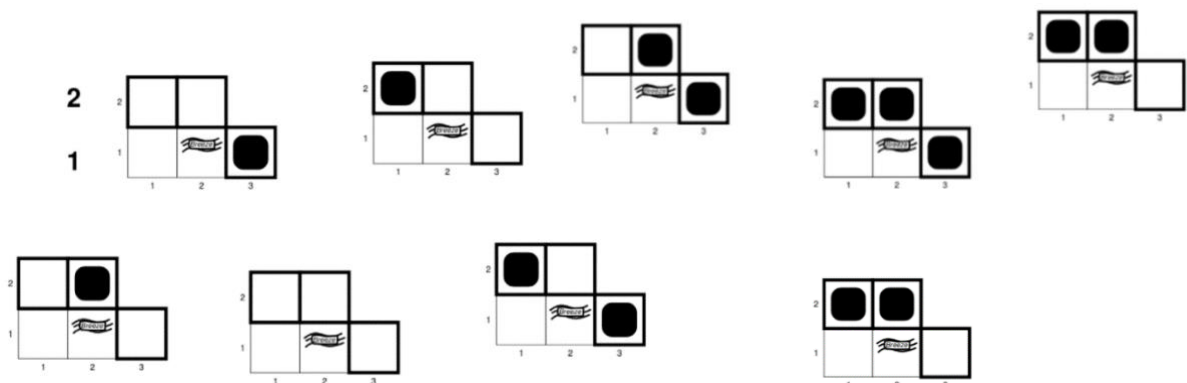
4.2 Dimostrazione tramite model checking



Per dimostrare $KB \models \alpha \Leftrightarrow M(KB) \subseteq M(\alpha)$ utilizziamo il Model Checking che enumera tutti i possibili modelli per verificare che α sia vera in tutti quelli in cui è vera la KB.

La dimostrazione sarà fatta seguendo l'esempio del **Wumpus World** e della sua "esplorazione". Supponiamo la seguente scacchiera:



Supponiamo che l'esploratore si trovi nella cella [1,1] e successivamente si sposti nella [2,1]. In questa posizione **sente del vento (Percezione)** e si domanda cosa c'è attorno. Rappresentiamo il mondo possibile in cui l'esploratore si possa trovare:



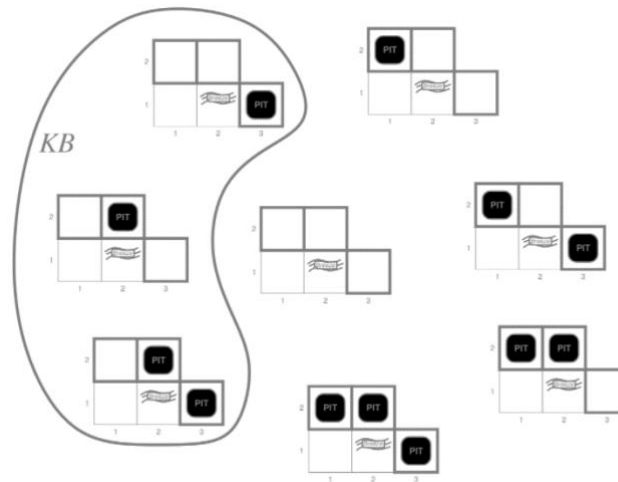
Si nota che viene indicato  il vento che percepisce l'esploratore e  il pozzo da evitare. Dall'insieme di tutte le possibili assegnazioni a cui facciamo riferimento consideriamo il caso in cui nelle celle possa esserci o non esserci un pozzo. Abbiamo infatti 2^3 mondi.

Noi sappiamo che in posizione [1,1] non c'è nulla.
Nella posizione [2,1] c'è una sensazione di vento.

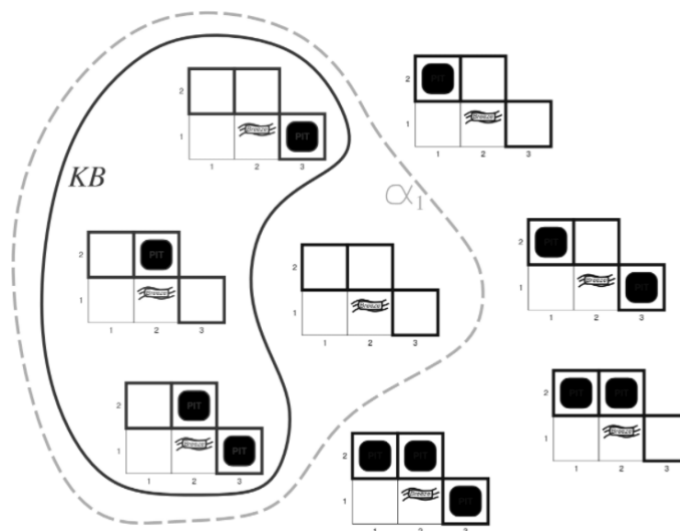
Dalla definizione di KB le osservazioni devono essere sommate alla base della conoscenza. Infatti:

$$KB = \text{regole del gioco} + \text{osservazioni}$$

Quindi dall'insieme precedente si va ad evidenziare l'insieme delle KB:



Supponiamo un certo α la deduzione che nella posizione [1,2] sia vuota (sicura). Rappresentando l'insieme α noteremo:



Si nota infatti che KB è contenuto nell'insieme α (vale l'inclusione), quindi per definizione diremo che $KB \models \alpha$.

Lo svantaggio di questa soluzione è che considerando un vettore di 10, 100, 1000 variabili (dove ogni variabile può assumere x valori, ci troveremo un numero di mondi pari a x^{10} (complessità esponenziale che rende la soluzione poco pratica).

4.3 Cosa si intende per deducibilità logica/inferenza

È un ragionamento logico che consente di giungere a una conclusione a partire da un insieme di premesse **deducendo** se l'implicazione da KB verso α è vera (vale). La deduzione è un processo di inferenza e si cerca una procedura che sia coerente e completa all'implicazione basata sui modelli.

Un algoritmo di inferenza i può derivare α da KB e si rappresenta con $(KB \vdash_i \alpha)$. Questa regola è corretta se deriva solo formule che sono conseguenze logiche. È completa se può derivare ogni formula che è conseguenza logica.

4.4 Dimostrazione tramite inferenza logica (regola di risoluzione)

Per sopperire alla possibile complessità del Model Checking, si segue un approccio differente legato al concetto di soddisfacibilità.

Si dice infatti che la frase/formula α è **soddisfacibile** se è vera in qualche modello.

Si dice frase/formula α **insoddisfacibile** se non è vera in nessun modello ovvero ed è possibile dimostrare la conseguenza logica:

(1) $KB \models \alpha$ se e solo se $KB \wedge \neg \alpha$ è **insoddisfacibile**

È possibile seguire l'algoritmo PL-RESOLUTION per dimostrare la (1).

Funzione **PL-Resolution(KB, α)** ritorna vero o falso

input: KB, α

Clausole \leftarrow il set di clausole ottenute nella rappresentazione CNF di $KB \wedge \neg \alpha$

new $\leftarrow \{\}$

Fai il ciclo:

per ogni C_i, C_j nella Clausola fai:

risolventi \leftarrow PL-Resolve(C_i, C_j)

Se risolventi contiene clausole vuote **ritorna vero**

Altrimenti new \leftarrow new \cup risolventi

Se new \subseteq clausole **ritorna false**

clausole \leftarrow clausole \cup new

Quindi, dato una KB e un α , si verifica la (1) raggiungendo una contraddizione, convertendo $KB \wedge \neg \alpha$ in CNF (**Conjunctive Normal Form**), se presenta "clausole complementari" produco una nuova clausola che viene aggiunta all'insieme delle clausole.

Le clausole complementari sono disgiunzioni letterali che si presentano in positivo da una parte e in negativa dall'altra. Sarà dunque possibile riscrivere queste proposizioni complementari in un'unica proposizione mettendo insieme tutti i letterali applicando la seguente regola:

$$\frac{l_1 \vee \dots \vee l_k, \quad m_1 \vee \dots \vee m_n}{l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n}$$

where l_i and m_j are complementary literals.

Se i risolventi contengono clausole vuote allora ritorna vero e dunque α è conseguenza logica di KB.

4.5 Un esempio di inferenza logica (risoluzione) espressa nella logica proposizionale

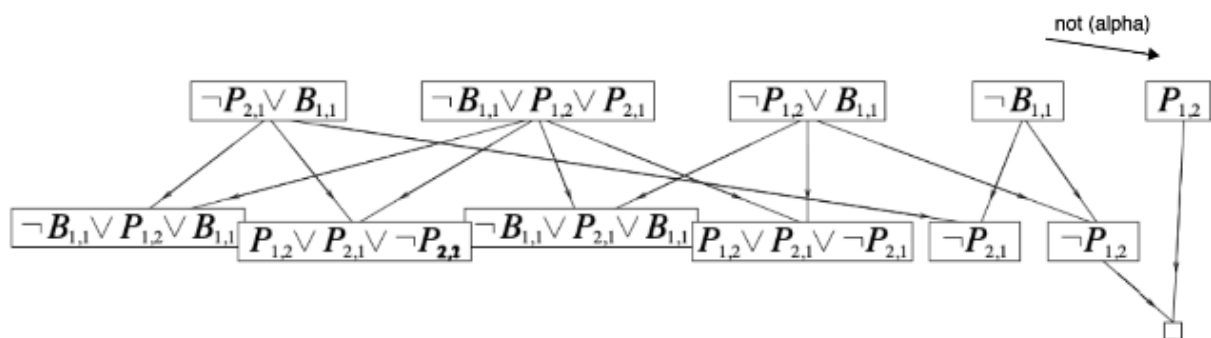
È possibile esprimere nella logica proposizionale un'inferenza logica applicando le diverse regole di inferenza. Si propongono alcune regole che seguiremo nell'esempio di rappresentazione CNF:

$$\begin{aligned}
 (\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
 (\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
 \neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
 \neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
 (\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
 (\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
 \end{aligned}$$

Supponendo l'esempio $KB = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$ convertito in CNF:

1. Eliminazione del \Leftrightarrow usando la regola $(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$
 $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) \wedge \neg B_{1,1}$
2. Eliminazione \Rightarrow usando la regola $(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$ implication elimination
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) \wedge \neg B_{1,1}$
3. Uso de Morgan $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) \wedge \neg B_{1,1}$
4. Applico la proprietà distributiva $(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) \wedge \neg B_{1,1}$

Avendo questa conoscenza di base KB si vuole verificare se $\alpha = \neg P_{1,2}$ aggiungendolo al set di clausole e applicando l'algoritmo di risoluzione per ogni coppia:



Trovando quindi la clausola vuota si giunge al fatto che $KB \models \alpha$

ESERCITAZIONE 3:

BAYESIAN NETWORK, NEURAL NETWORK

ESERCIZIO 1

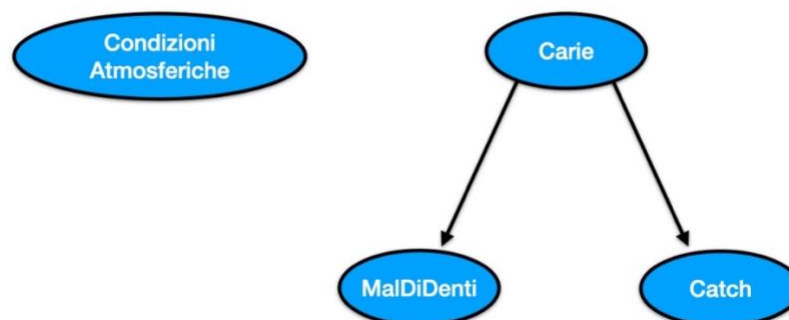
1.1 Riassumere i concetti principali delle reti bayesiane (rappresentazione e inferenza)

Abbiamo visto che la distribuzione di probabilità congiunta completa può rispondere a qualsiasi domanda riguardante il dominio, ma diventa intrattabilmente grande al crescere del numero di variabili. Le relazioni di dipendenza assoluta e condizionale tra le variabili possono ridurre drasticamente il numero di probabilità che devono essere specificate esplicitamente per definire la distribuzione congiunta completa. Ora introduciamo il concetto di una struttura dati, chiamata **rete bayesiana**, per rappresentare le dipendenze tra le variabili e fornire una specifica concisa di qualsiasi distribuzione di probabilità congiunta completa. Una **rete bayesiana** è un grafo orientato in cui ogni **nodo** è etichettato con informazione probabilistica quantitativa. La specifica completa è la seguente:

- 1) I nodi della rete sono costituiti da un insieme di variabili casuali, che possono essere discrete o continue.
- 2) Un insieme di archi orientati (freccie) collega coppie di nodi. Se c'è un arco tra il nodo X e il nodo Y, si dice che **X è genitore di Y**.
- 3) Ogni nodo X_i ha una distribuzione di probabilità condizionata $P(X_i | \text{Genitori}(X_i))$ che quantifica gli effetti dei genitori su un nodo.
- 4) Il grafo non ha cicli orientati (si dice che è un **DAG**, dall'inglese **Directed Acyclic Graph**, ovvero *Grafo Diretto Aciclico*).

La **topologia della rete**, ovvero l'insieme dei nodi e quello degli archi, specifica le condizioni di indipendenza. Solitamente il significato *intuitivo* di una freccia, in una rete costruita correttamente, è che X ha *un'influenza diretta* su Y.

Se prendiamo in considerazione l'esempio preso a lezione, vediamo che esso consisteva nelle sole variabili *MalDiDenti*, *Catch*, *Carie* e *CondizioniAtmosferiche*. Deduciamo che *CondizioniAtmosferiche* è indipendente dalle altre variabili, inoltre, *MalDiDenti* e *Catch* sono condizionalmente indipendenti, data *Carie*. Queste relazioni sono rappresentate dalla seguente struttura della rete bayesiana:



Una volta delineata la topologia di una rete bayesiana, rimane da specificare la **distribuzione di probabilità condizionata** di ogni variabile, dati i suoi genitori. La combinazione della topologia e delle

distribuzioni condizionali è sufficiente a definire implicitamente la distribuzione congiunta completa di tutte le variabili.

Ogni distribuzione, in figura, ha la forma di una tabella delle probabilità condizionate, o in acronimo CPT. Questo tipo di tabella può essere usato per le variabili discrete e contiene le probabilità condizionate di tutti i valori del nodo per un singolo caso condizionante. Con quest'ultimo termine si indica semplicemente una possibile combinazione dei valori dei nodi genitori: una specie di piccolo evento atomico. Ogni riga deve avere somma 1, perché i suoi elementi rappresentano un insieme esaustivo di casi. In generale, la tabella di una variabile booleana con k genitori booleani contiene 2^k probabilità indipendenti. Un nodo senza genitori ha una sola riga che riporta le probabilità a priori di ogni suo possibile valore.

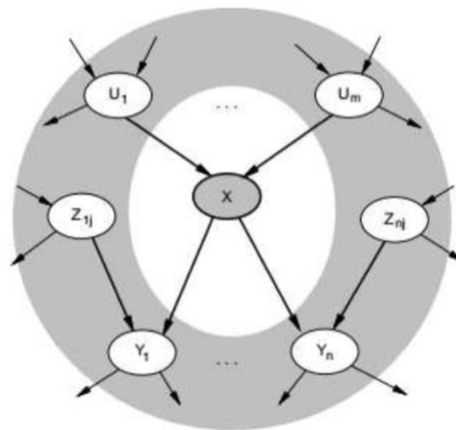
CONCETTO DI SEMANTICA GLOBALE:

A partire da distribuzioni di probabilità condizionata locale è possibile passare alla distribuzione di probabilità congiunta totale, attraverso la relazione:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | \text{genitori}(X_i))$$

CONCETTO DI SEMANTICA LOCALE:

In una rete bayesiana, ogni nodo, dati i suoi genitori è indipendente dagli altri nodi. In particolari condizioni, ovvero, dati i genitori, i figli e i padri dei figli; per un nodo, si viene a formare, la cosiddetta coperta di Markov. Il nodo in questo caso risulterà condizionalmente indipendente da tutti gli altri.



INFERENZA NELLE RETI BAYESIANE:

Per quando riguarda l'**inferenza per enumerazione**, ogni probabilità condizionata può essere calcolata semplicemente sommando i termini relativi della distribuzione congiunta completa. Specificatamente, a una query $P(X|e)$ si può rispondere applicando la seguente equazione:

$$P(X|e) = \alpha \sum_y P(X, e, y)$$

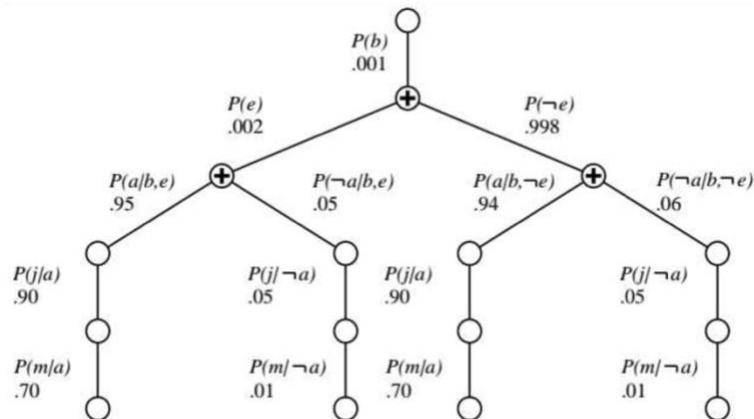
Ora una rete bayesiana fornisce una rappresentazione della distribuzione congiunta completa; più precisamente l'equazione:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | \text{genitori}(X_i))$$

Mostra che i termini $P(x, e, y)$ della distribuzione congiunta si possono scrivere come i prodotti di probabilità condizionate prese dalla rete. Quindi si può dare risposta ad una query attraverso una rete bayesiana, sommando i prodotti delle probabilità condizionate prese dalla rete stessa.

Quindi si applica alla rete bayesiana il meccanismo di marginalizzazione delle variabili che consiste nel sommare tutte le combinazioni possibili delle variabili che voglio eliminare e questo passaggio è automatizzabile.

La struttura che si viene a creare è di questo tipo:



Il **processo di valutazione** procede dall'alto verso il basso, moltiplicando i valori lungo ogni cammino e sommando nei nodi "+".

Quando non si ha accesso a tutte le tabelle di probabilità condizionate si procede per **simulazione stocastica** ovvero facciamo degli esperimenti stimando i valori possibili, attraversando l'albero, molteplici volte rispettando la distribuzione di probabilità di ognuno dei nodi considerati sulla rete; in modo tale da determinare dei campionamenti statistici basati sulla rete bayesiana stessa.

1.2 Descrivere e spiegare con un esempio come il formalismo delle reti bayesiane può essere applicato - 3 pagine max

Supponiamo di aver installato un allarme anti-intrusione all'interno della casa di Salvatore che si attiva quando rileva del movimento nelle stanze. Quando l'allarme suona viene inviato un sms di avvertimento sul telefono di Salvatore. Se Salvatore lascia la finestra aperta e si leva il vento, le foglie secche del giardino entrano in casa e possono fare scattare l'allarme.

Quello che vogliamo stimare è dato che oggi c'è vento qual è la probabilità che Salvatore riceva un sms di allarme causato dalle foglie secche entrate in casa. (Le distribuzioni di probabilità associate ai nodi della rete sono state scelte dagli studenti)

Da quanto scritto ne discende la seguente rete bayesiana che si compone di:

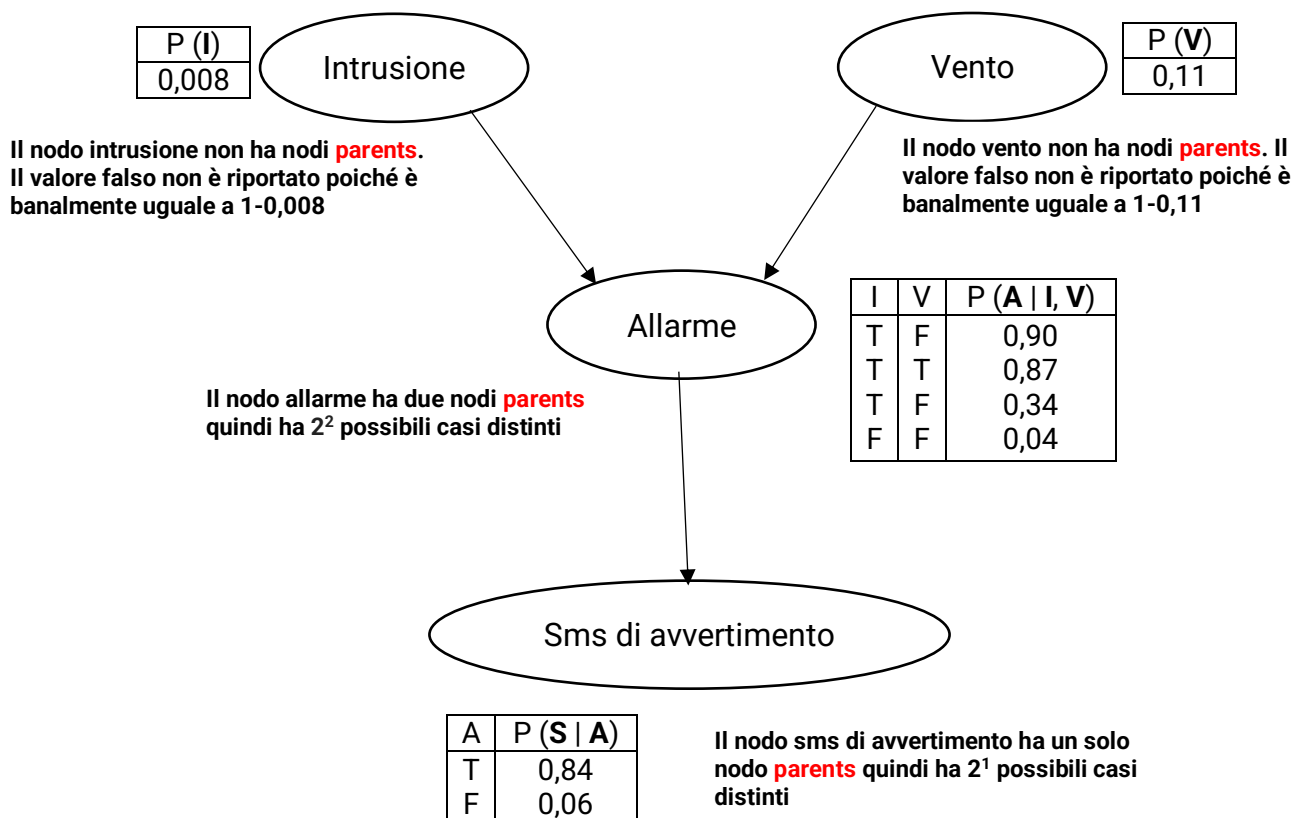
- **Nodes**: rappresentano l'insieme di variabili richieste per modellare il dominio di interesse $\{X_1, X_2, \dots, X_n\}$.

Per una rete più compatta sarebbe meglio ordinare le variabili in modo che la causa preceda l'effetto.

- **Links**: rappresentano i collegamenti tra le varie variabili. Dato un nodo X_i si scelgono un valore minimo di nodi **parents** (relazione di dipendenza). Ogni nodo **parents** è connesso al nodo X_i . Il nodo X_i avrà, infine, una **CPT** (tabella di probabilità condizionale) in cui avremo la $P(X_i | \text{Parents}(X_i))$.

Intuitivamente i parenti del nodo X_i sono tutti quei nodi che influenzano direttamente X_i .

Nel nostro caso, ad esempio, abbiamo che l'attivazione dell'allarme dipende dal nodo intrusione e vento, mentre l'invio di un sms di avvertimento dipende direttamente solo dall'attivazione dell'allarme.



Ricapitolando abbiamo che i **parents** del nodo **Allarme** sono **Intrusione** e **Vento** mentre il **parent** di **Sms di avvertimento** è il nodo **Allarme**.

All'interno delle CPT abbiamo le probabilità condizionali [es. $P(\text{Sms di avvertimento} \mid \text{Allarme})$]

Ogni riga di un CPT contiene la probabilità condizionale di ciascun valore di un nodo per caso di condizionamento, dove un caso di condizionamento non è altro che una combinazione di valori per i nodi **parents**.

Ogni riga della tabella deve essere, sommata, uguale a 1, poiché le voci rappresentano una serie esaustiva di casi per la variabile. Per le variabili booleane, una volta che si sa che la probabilità di un valore vero è p , la probabilità di un valore falso deve essere $1 - p$, quindi spesso omettiamo il secondo valore perché ricavabile dal primo.

In generale, una tabella per una variabile booleana con k **parents** booleani contiene 2^k probabilità specificabili indipendentemente. Un nodo senza **parents** sarà composto da una singola riga che rappresenta le probabilità a priori di ogni possibile valore della variabile.

Torniamo adesso allo scopo di questo esempio, ovvero, calcolare la probabilità che Salvatore riceva un sms di allarme dato la presenza di vento. In poche parole, vogliamo sapere il valore di $P(\text{Sms di avvertimento} \mid \text{Vento})$.

Facendo uso della semantica globale (riportata nell'esercizio precedente) otteniamo che:

$$\begin{aligned} P(S \mid V) &= P(S \mid A) P(A \mid \neg I, V) P(V) P(\neg I) \\ &= 0,84 \times 0,34 \times 0,11 \times (1 - 0,008) \\ &\approx \mathbf{0,03} \end{aligned}$$

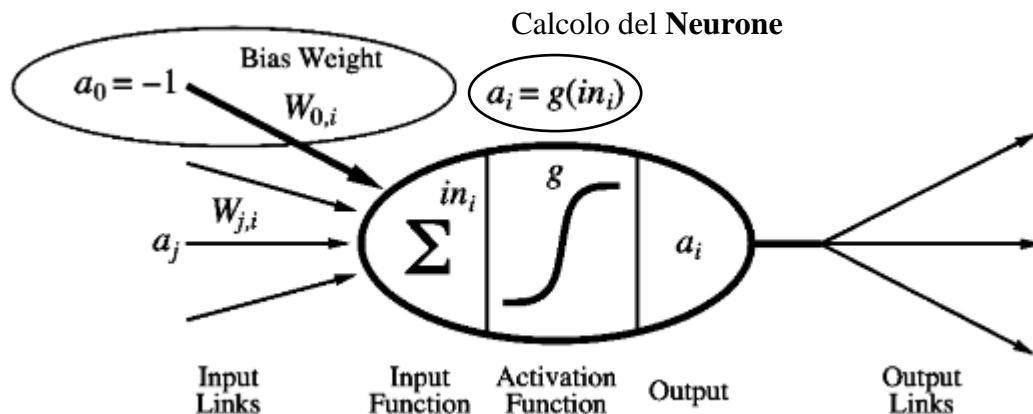
Dove S = Sms di attivazione, A = Allarme, I = Intrusione, V = Vento.

ESERCIZIO 2

2.1 Riassumere i concetti principali delle reti neurali (spiegare come sono implementati i neuroni, la loro architettura e l'algoritmo di error back propagation)

Le reti neurali sono diventate ormai il fiore all'occhiello dell'Intelligenza artificiale. Inizialmente affiancato al concetto di neurone umano che sulla base di percezioni sensoriali stimolano il neurone a scambiare messaggi con l'ambiente esterno in "modo elettrico".

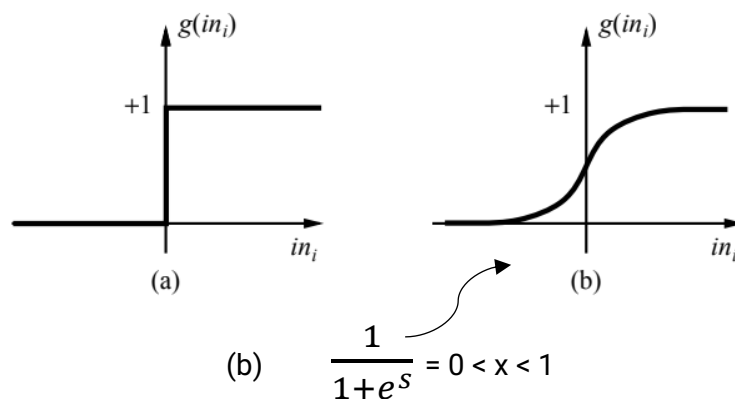
Certamente, il neurone umano non può essere simulato e adottato in un sistema di calcolo di elaborazione. Infatti, i ricercatori hanno trovato una soluzione che si avvicina ad avere un'idea del neurone attraverso una rappresentazione grafica come quella riportata in figura:



Dalla figura si notano:

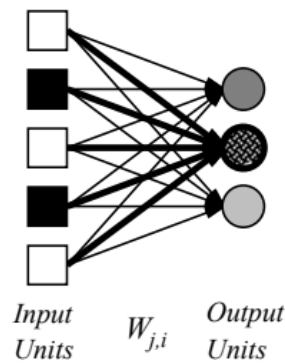
a_j = Input Links (archi entranti), in_i = Input Function, $W_{j,i}$ = Peso dell'arco, $g(in_i)$ = Funzione di Attivazione*, a_i = Output (archi uscenti).

*Inizialmente la funzione di Attivazione del Neurone era un gradino (a), mentre ora si utilizza la funzione di Sigmoide (b) detta anche funzione logistica.

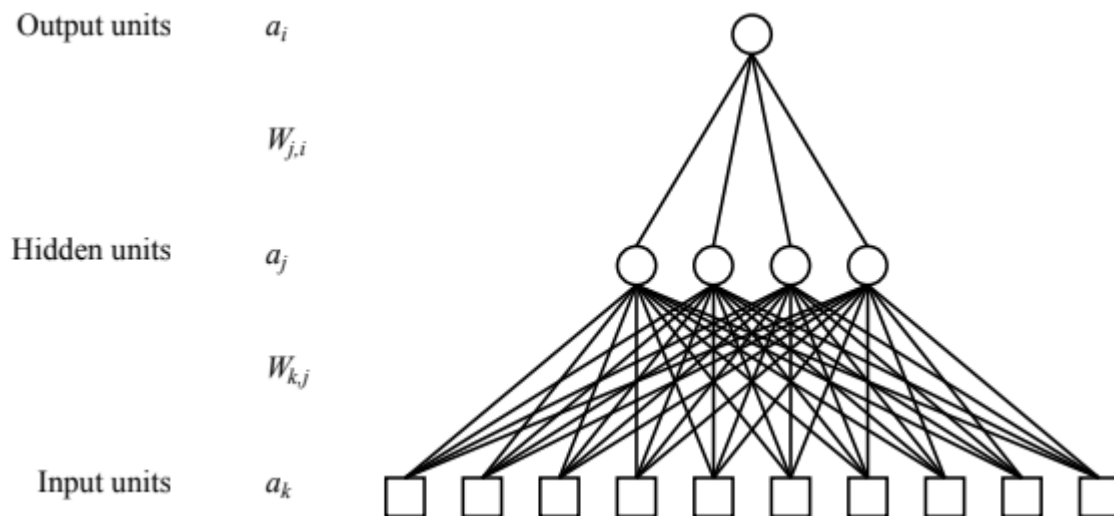


Esistono diverse classificazioni di RN, tra cui troviamo il **Feed Forward Network** (connessioni solo in una sola direzione) suddiviso in Single Layer Perceptron e in Multilayer Perceptron.

In breve, il Single Layer Perceptron è una rete che presenta tanti input collegati direttamente a tutte le uscite, come è rappresentato nella figura successiva. Queste reti funzionano solo per problemi linearmente separabili. Considerando in ingresso una funzione gradino (perceptron) posso rappresentare la AND, OR, NOT e il MAJOR ma non è possibile calcolare la XOR. Lo si capisce dalla rappresentazione cartesiana (input space) degli ingressi ($x = [0,1]$) e calcolando la linear separator con la $\sum_j W_j x_j > 0$.

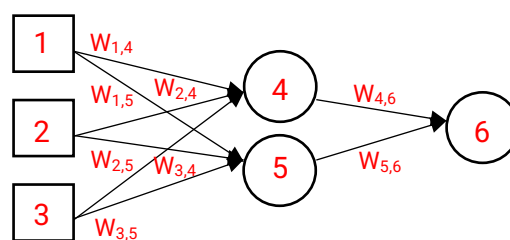


Il Multilayer Perceptron invece presenta ulteriori layer come rappresentato in figura:



Grazie alla suddivisione in più livelli, la rete è in grado di rappresentare funzioni continue e funzioni discontinue. In questo caso la funzione XOR è possibile rappresentarla. La MLP è utilizzata nel deep learning.

Un esempio di calcolo della rete neurale di MLP:



Dall'esempio precedente si nota come i valori vengono propagati all'interno della rete secondo gli archi con cui la rete è costruita. I nodi 1,2,3 sono i nodi di ingresso, i nodi 4,5 sono Hidden mentre il nodo 6 è di uscita ed è calcolabile:

$$a_6 = g(W_{4,6}a_4 + W_{5,6}a_5) =$$

$$= g(W_{4,6} * g(W_{1,4}a_1 + W_{2,4}a_2 + W_{3,4}a_3) + W_{5,6} * g(W_{1,5}a_1 + W_{2,5}a_2 + W_{3,5}a_3))$$

Da come si evince dal calcolo precedente, a_6 dipende dai pesi presenti nella rete. La scelta dei pesi, dunque, è di fondamentale importanza per determinare l'uscita della rete neurale. Esiste infatti l'algoritmo di **ERROR BACK PROPAGATION LEARNING** che dà la possibilità di apprendere in maniera automatica i pesi, dati i dati.

Partendo da una rete con una struttura a caso, come quella nella figura precedente, abbiamo una

coppia di valori di ingresso e di uscita. Supponiamo di ottenere in uscita un valore diverso da 6. C'è dunque un errore che deve essere corretto. Si utilizza il procedimento del back propagation dell'errore. Quando si prova ad effettuare la modifica dell'errore la si fa con calma, passo passo senza correggere direttamente l'errore in modo tale da tarare i pesi presenti nella rete senza esagerare. Se applichiamo questo procedimento su tutti gli esempi, ogni esempio andrà a modificare leggermente il peso degli archi. Ogni esempio darà un contributo alla correzione dell'errore che determinerà uno spostamento della rete verso una soluzione consistente con gli esempi.

Un possibile metalgoritmo del Back Propagation Error:

Funzione ERROR_BACK_PROPAGATION (rete, esempi, alfa) **Ritorna** rete con pesi modificati

Input: Rete MLP, Esempi, Alfa (tasso di apprendimento)

loop:

for each esempio

→ Calcolo l'uscita per questo esempio $a_i = \text{EseguiRete}(\text{rete}, a_j)$

→ Calcolo l'errore $\text{Err}_i = a_i^{\text{True}} - a_i$

→ Si aggiornano i pesi $W_{j,i} = W_{j,i} + \text{alfa} * a_j * \text{Err}_i * g'(in_i)$

for each strato successivo della rete

→ Calcolo l'errore in corrispondenza di ciascun nodo con $\Delta j = g'(in_i) \sum_i W_{j,i} * \text{Err}_i * g'(in_i)$

→ Si aggiornano i pesi che conducono all'uscita: $W_{k,j} = W_{k,j} + \text{alfa} * a_j * \Delta j$

Lo svantaggio di questo algoritmo che è molto costoso in termini di tempo seppur presentando buoni risultati.

In definitiva, le reti neurali sono applicate in diversi contesti di ricerca scientifica e di utilizzo ormai quotidiano nel mondo della tecnologia, alcuni esempi possono essere la guida autonoma, l'handwritten recognition e soprattutto l'elaborazione e ricerca di oggetti nelle immagini che nel prossimo paragrafo trattiamo.

2.2 Come utilizzereste una rete neurale per riconoscere la presenza o meno di alberi in un'immagine digitale - 3 pagine max

Un esempio di applicazione di rete neurale è l'**object recognition**, ovvero la capacità di riconoscere, trovare un determinato oggetto, luogo, persone e azioni in un'immagine o in una sequenza di immagini (video).

Nell'uomo e negli animali, riconoscere oggetti è naturale, l'occhio umano vede un'immagine come un insieme di segnali, l'interpreta attraverso la corteccia visiva del cervello e il risultato è una scena legata a concetti che vengono conservati nella nostra memoria interpretando le forme, i confini di un oggetto, i colori e distinguerli.

Per un sistema di elaborazione il concetto è sicuramente più complesso e diverso dato che l'immagine è una sequenza di bit o una tela di pixel con valori numerici discreti per i colori. I sistemi di elaborazione atti ad analizzare l'immagine devono essere opportunamente addestrati attraverso un particolare tipo di rete neurale chiamata **Rete Neurale Convoluzionale (CNN)**.

La Rete Neurale Convoluzionale è una tipica rete neurale di tipo **feed-forward (simile a quelli analizzati nel paragrafo precedente)**. Essa, infatti, è un esempio di Multilayer Perceptron ma a differenza della classica MLP ci sono una serie di livelli di convoluzione con funzione di attivazione (ReLU).

Lo **strato di convoluzione** estrae attraverso l'uso di **filtri** le caratteristiche delle immagini del contenuto che si vuole analizzare. A seconda del tipo di filtro utilizzato è possibile identificare sull'immagine di riferimento i contorni delle figure, le linee verticali, le curve, le linee orizzontali. I filtri applicabili sono e possono essere più di uno; Maggiore è il loro numero e maggiore è la complessità delle caratteristiche che si potranno individuare.

Dalla consegna dell'esercizio, supponiamo di riconoscere da un'immagine la presenza di un albero. La rete neurale inizialmente deve essere allenata con migliaia di immagini di alberi e senza alberi imparando a riconoscere i confini e le caratteristiche dello stesso (quindi anche il tipo che andrà ad individuare). Nel nostro esempio, per semplicità, usiamo un'immagine con un grande albero.

Nello strato di input si semplificherà l'immagine convertendola da colori in bianco e nero e sezionandola in forma matriciale.

Sempre per semplicità, consideriamo un sezionamento matriciale 8x8 per poter svolgere un semplice esempio di riconoscimento (come mostrato nella figura).



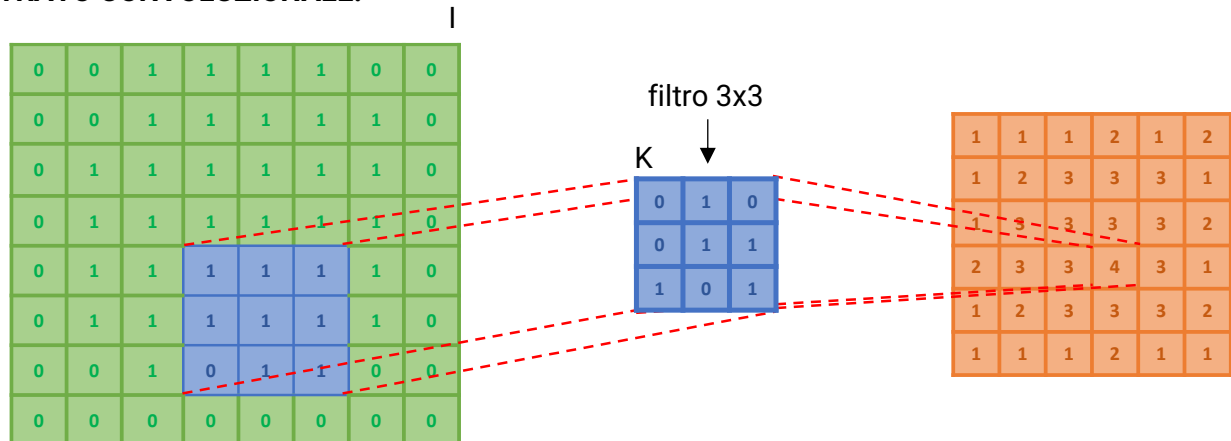
Nello strato di convoluzione si analizza ogni sezione dell'immagine individuando curve, angoli e circonferenze appartenenti all'albero che stiamo riconoscendo. Verrà opportunamente applicato un filtro (può essere più di uno, maggiore è il loro numero e maggiore è la complessità delle caratteristiche che si potranno individuare). Il filtro conosce tutti gli alberi che gli sono stati insegnati e dovrà essere in grado di capire se quel "pezzo" sezionato appartenga ad un albero e di che tipo di albero.

Questo filtro viene fatto scorrere sulle diverse posizioni dell'immagine sezionata e per ogni posizione viene generato un valore di output eseguendo il prodotto scalare tra la maschera di sezione e i valori del filtro.

Nella figura successiva il filtro è rappresentato da una matrice 3x3, pertanto, il pennello di scansione prenderà in esame solo quella porzione dell'immagine e fare il prodotto di convoluzione. Verrà scansionata tutta la matrice ottenendo in uscita una matrice dei valori definiti dal prodotto scalare:

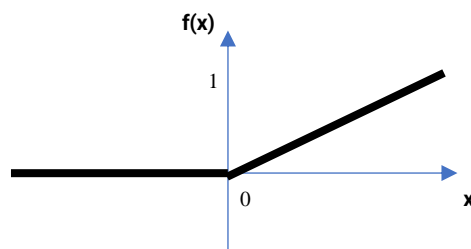
$$I \times K \text{ dove } I = [0(\text{non c'è albero}), 1(\text{c'è albero})]$$

STRATO CONVOLUZIONALE:



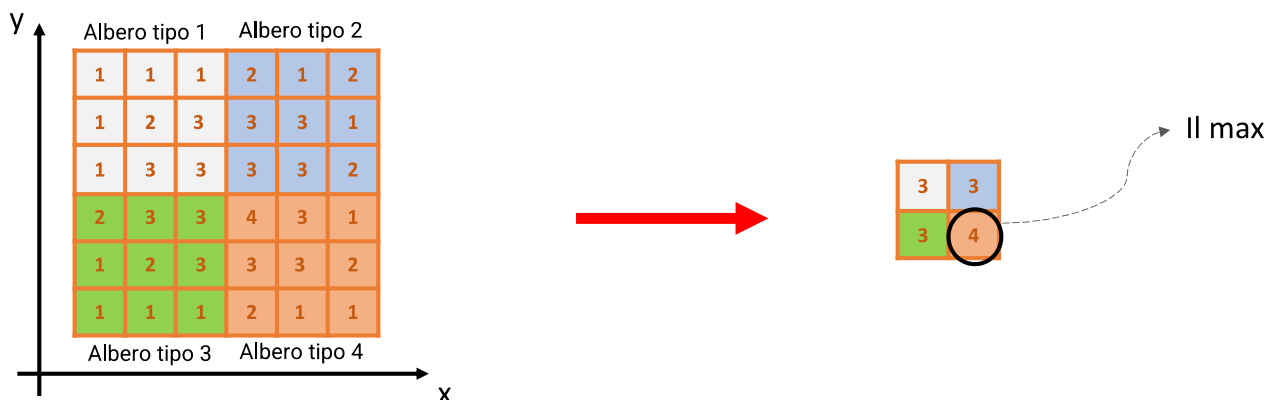
STRATO DI ATTIVAZIONE:

La funzione di attivazione più utilizzata è la funzione Rectified Linear Unit (ReLu) che restituisce zero per input negativi mentre preserva gli input positivi, cioè $f(x) = \max(0, x)$



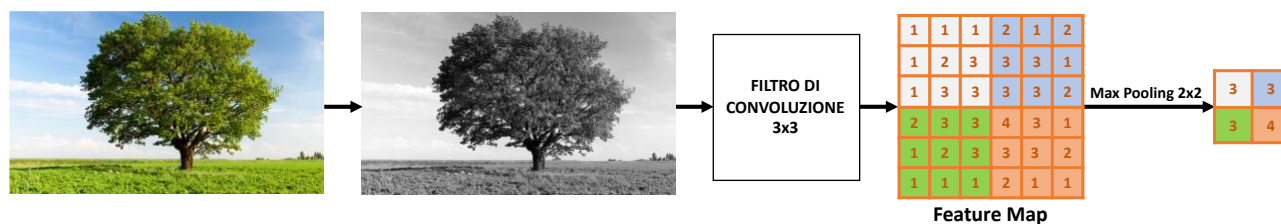
STRATO DI POOLING:

Il pooling può essere max-pooling o avg-pooling. Nel nostro caso utilizziamo il max-pooling. La forma più comune di pooling è il max-pooling 2×2 , che dimezza le due dimensioni prendendo il massimo calcolato su regioni 2×2 , spostandosi di due posizioni ad ogni iterazione.



Dalla figura notiamo che il massimo della matrice 2x2 appartiene alla classificazione dell'albero di tipo 4 e pertanto possiamo concludere che c'è un albero presente nell'immagine ed è del tipo 4.

RIEPILOGO:



Nel procedimento illustrato sono stati aggiunti dei nostri punti di vista e possibili nuove soluzioni per il riconoscimento di un oggetto nell'immagine.

Sitografia della ricerca: <https://www.domsoria.com/2019/10/rilevamento-di-oggetti-con-tensorflow/> ,
https://amslaurea.unibo.it/15607/1/tesi_gatto.pdf

ESERCIZIO A (opzionale)

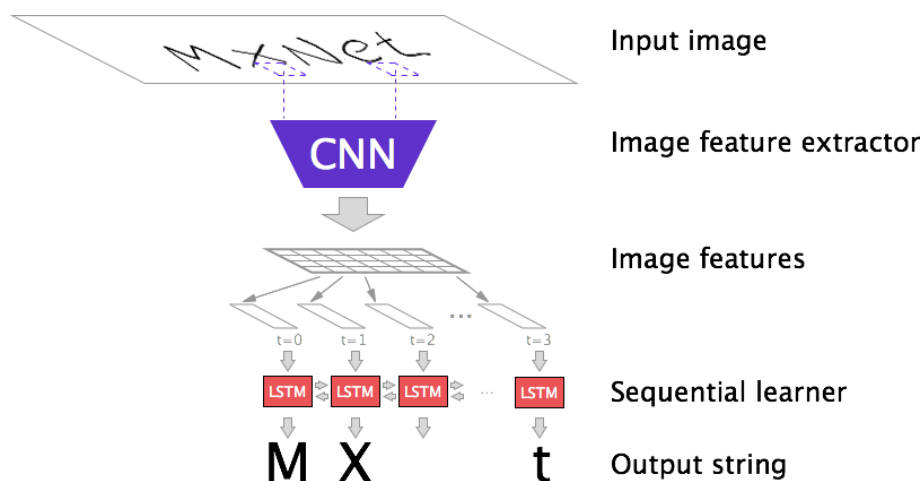
Studiare il codice che mostra come una rete neurale MLP possa essere applicata al problema di handwritten character recognition

Una delle applicazioni delle reti neurali (come già detto nei paragrafi precedenti) è l'Handwritten Character Recognition ovvero l'abilità del sistema di elaborazione ad interpretare l'handwritten input dai documenti, fotografie, tavola grafica etc. Nel caso di interpretazione di testo su carta può avvenire attraverso una scansione ottica oppure nel caso dei dispositivi touch o con la tavoletta grafica che viene opportunamente trattato attraverso un procedimento di riconoscimento intelligente delle parole con una rete neurale MLP. Per testare il codice si usa il dataset **MNIST** (Modified National Institute of Standards and Technology dataset) che è un set di dati di cifre scritte a mano. Consiste in 70.000 immagini in formato 28x28 pixel in scala di grigi, che si presentano come nell'immagine successiva:

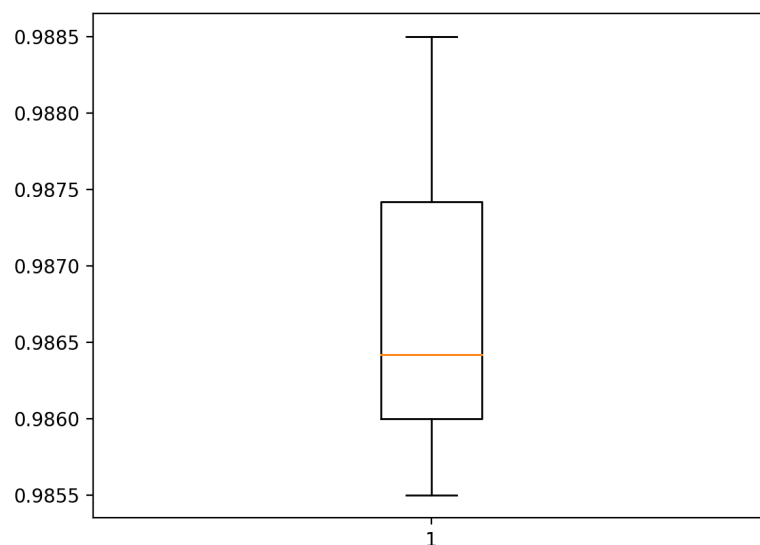


Leggendo il codice, si nota che l'approccio utilizzato è il Convolutional Neural Network, come visto già in precedenza.

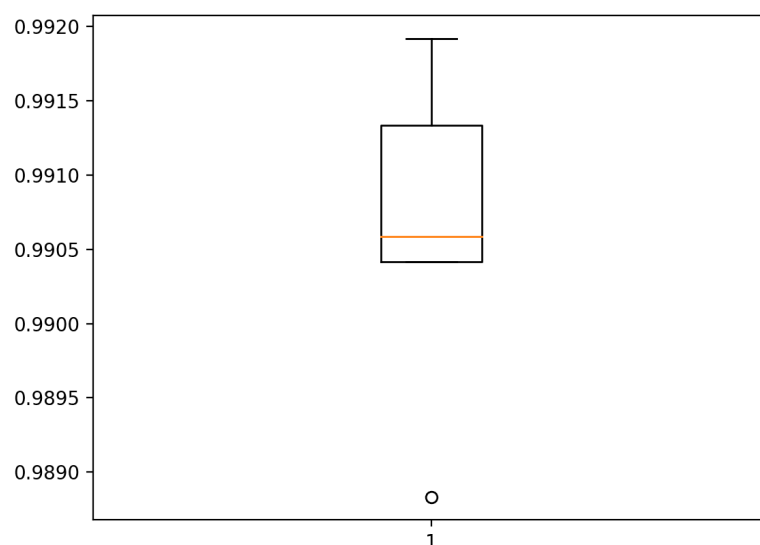
Innanzitutto, viene caricato il dataset utilizzando la libreria MXnet, si suddivide dunque, il set in due parti una per il train con 60.000 immagini ed un'altra con il test con 10.000 immagini. Vengono poste in ingresso le immagini del **trainset** con testo da riconoscere, e viene applicato nella Convolutional Neural Network al fine di creare un Model Evaluation. Un esempio semplificato graficamente:



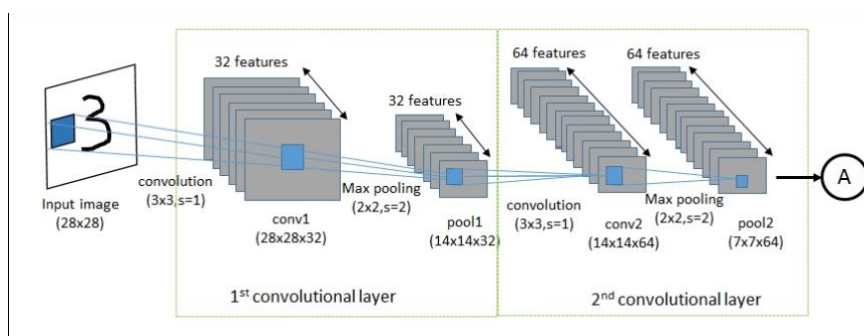
Grazie al modello trovato con il train della CNN, dovremmo essere in grado ponendo in ingresso un'immagine del **testset** di ottenere una predizione del numero scritto, con un valore di precisione che dovrebbe aggirarsi intorno a 0,96, il che significa che siamo in grado di prevedere accuratamente la cifra nel 96% delle immagini di prova. Purtroppo, durante la nostra esperienza non siamo riusciti ad eseguire perfettamente il codice riportato nel link, tuttavia abbiamo riscontrato attraverso alcuni riferimenti online dei risultati analoghi in particolare con un modello con profondità di estrazione della feature pari a 32, abbiamo ottenuto un valore di predizione pari a 0,9867 con un valore di deviazione standard di 0.0007, graficamente:



Anche se i risultati ottenuti sembrano già buoni, tuttavia è possibile migliorarli aumentando la profondità del modello con l'applicazione di due filtri, invece che uno, in cui il primo estrae 32 feature, mentre il secondo estrae 64 feature. Questa volta si nota che il valore di predizione del modello ha un valor medio dello 0.9909 ed una deviazione standard di 0.0004, graficamente:



In definitiva, il processo di costruzione della CNN usato può riassumersi nel seguente schema:



ESERCIZIO B (opzionale)

Eseguire e studiare la soluzione del problema di Cart Pole Balancing utilizzando il Q-Learning

Il caso di studio in esame ha come obiettivo quello di controllare un carrello con pendolo capovolto che si muove in due soli direzioni, per raggiungere questo obiettivo viene usato l'algoritmo del Q-Learning. Tale approccio fa parte dei metodi "temporal difference off-policy" per il reinforcement learning in cui ad ogni step nella sequenza di azioni viene aggiornata la funzione di utilità $Q(s,a)$, ciò ci permette di raggiungere un ottimo massimo locale.

Per utilizzare questo algoritmo va implementata la seguente formula:

$$Q^{new}(s_t, a_t) = \underbrace{Q(s_t, a_t)}_{\text{Old value}} + \underbrace{\alpha}_{\text{Learning Rate}} \cdot \underbrace{(\underbrace{R(s_t, a_t)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount Factor}} \cdot \underbrace{\max_a(Q(s_{t+1}, a_t))}_{\text{Estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{Old value}})}_{\text{New value (temporal difference target)}}$$

temporal difference

In cui i parametri γ e α , sono scelti da noi e forniti in input alla funzione del Q-Learning dell'agente, in particolare, avremo che il fattore α (**learning rate**) si decrementa all'aumentare del time step t . Il learning rate, o tasso di apprendimento, determina con quale estensione le nuove informazioni acquisite sovrascriveranno le vecchie informazioni. Il valore che può assumere varia tra 0 e 1, quando il valore è 0 ciò impedisce all'agente di apprendere, al contrario un fattore pari ad 1 fa sì che l'agente si interessi solo delle informazioni recenti.

L'unico fattore fisso è il **discount rate** γ . Il fattore γ determina l'importanza delle ricompense future. Un fattore pari a 0 renderà l'agente "opportunista" facendo sì che consideri solo le ricompense attuali, mentre un fattore tendente ad 1 renderà l'agente attento anche alle ricompense che riceverà in un futuro a lungo termine. In letteratura è comune utilizzare discount rate prossimo ad 1 perciò è stato scelto $\gamma = 0.99$.

All'interno dell'algoritmo è presente anche il seguente parametro **exploration rate** ϵ che rappresenta il fattore di esplorazione che si spinge l'algoritmo verso una scelta esplorativa (stocastica) piuttosto che di apprendimento, ovvero seguendo la policy assegnata π_i . Con l'avanzare degli episodi l'exploration rate viene decrementato per dare più peso alla policy di apprendimento. Nella pratica quindi si cerca di mantenere una politica maggiormente esplorativa all'inizio del training per poi spostare l'attenzione sul fattore di apprendimento. All'interno del codice questo meccanismo viene implementato in questo modo:

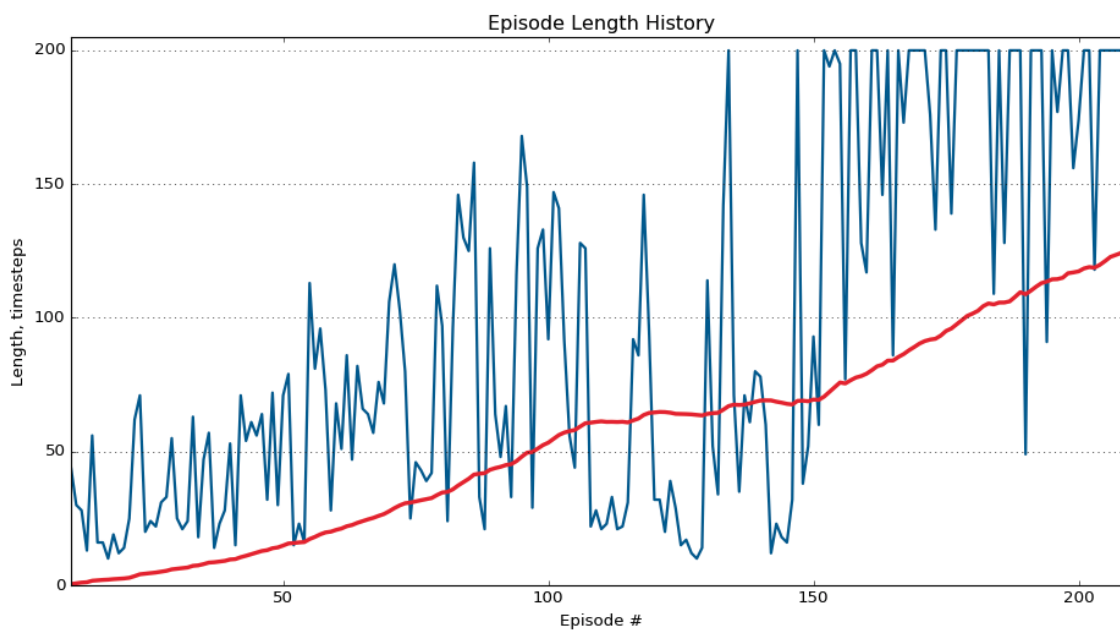
```
# Exploration/exploitation: choose a random action or select the best one.
enable_exploration = (1 - self.exploration_rate) <= np.random.uniform(0, 1)
if enable_exploration:
    next_action = np.random.randint(0, self._num_actions)
else:
    next_action = np.argmax(self.q[next_state])
```

I valori dei parametri del nostro codice sono i seguenti:

QLearning

Parametro	Valore
Learning rate α	0.2
Discount factor γ	1.0
Exploration Rate ϵ	0.5
Exploration Decay Rate	0.99

Eseguendo il codice, i risultati saranno:



Dopo un certo tempo il pendolo resterà in equilibrio:

