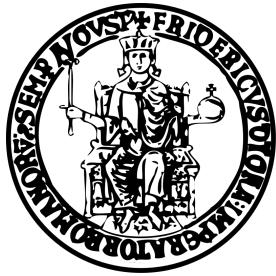


Università degli Studi di Napoli

Federico II



Progetto Intelligenza Artificiale

Docente Filippo Neri

Salvatore Giugliano - N46003673
10 giugno 2021

Esercitazione 1: DT Learning	3
Esercizio 1 : Creazione e Uso di un Decision Tree	4
<i>Punto 1: Modeling with Decision Trees</i>	4
<i>Punto 2: Applicare DT Learning ad un Dataset</i>	7
<i>Punto 3: Visualizzare variazione accuratezza</i>	8
Esercizio 2: Agente e DT learning	9
Esercizio 3: A* Search	10
Esercizio 4: Utilizzo Albero Decisione	12
Esercizio 5: Apprendimento di un Decision Tree	13
Esercizio 6: Spazio degli stati	16
Esercitazione 2: Algoritmi di ricerca locale	19
Esercizio 1: Principali algoritmi di ricerca Locale	20
<i>Punto 1: Evolutionary Optimization</i>	20
<i>Punto 2: Ottimizzare una funzione</i>	24
Esercizio 2: Map coloring	26
<i>Punto 1: Colore la mappa della regione Campania con 2 colori</i>	26
<i>Punto 2: Colore la mappa della regione Campania con 3 colori</i>	29
<i>Punto 3: Colore la mappa della regione Campania con 4 colori</i>	31
Esercizio 3: Algoritmo Genetico e Map Coloring	33
Esercizio 4: Logica preposizionale	35
Esercitazione 3: Reti Bayesiane e Reti Neurali	39
Esercizio 1: Rete Bayesiana	40
<i>Punto 1: Concetti Principali</i>	40
<i>Punto 2: Esempio di Rete Bayesiana</i>	42
Esercizio 2: Reti Neurali	43
<i>Punto 1: Concetti principali</i>	43
<i>Punto 2: Esempio di Rete neurale</i>	48
Esercizio opzionale A: Riconoscimento Numeri	49

Esercitazione 1: DT Learning

Esercizio 1 : Creazione e Uso di un Decision Tree

Punto 1: Modeling with Decision Trees

Dopo aver scaricato il codice della cartella del corso sulla piattaforma Google Drive ed averlo eseguito tramite l'editor 'Spyder', ho effettuato delle correzioni generali per adattarlo alla versione di Python installata sul mio elaboratore. Dopodiché ho eseguito tutte le istruzioni presenti nel Capitolo 7 del libro 'Programming Collective Intelligence' per capire il funzionamento delle varie funzioni presenti nel codice.

I dati usati sono :

```
[['slashdot','USA','yes',18,'None'], ['google','France','yes',23,'Premium'],
['digg','USA','yes',24,'Basic'], ['kiwitobes','France','yes',23,'Basic'],
['google','UK','no',21,'Premium'], [ '(direct)','New Zealand','no',12,'None'],
[ '(direct)','UK','no',21,'Basic'], ['google','USA','no',24,'Premium'],
['slashdot','France','yes',19,'None'], ['digg','USA','no',18,'None'],
['google','UK','no',18,'None'], ['kiwitobes','UK','no',19,'None'], ['digg','New
Zealand','yes',12,'Basic'], ['slashdot','UK','no',21,'None'], ['google','UK','yes',18,'Basic'],
['kiwitobes','France','yes',19,'Basic']].
```

Dagli esempi riportati sul libro si evince che l'entropia è circa 1,505 mentre la geni impurity è 0,632. Stampando l'albero con la funzione treepredict.drawtree(tree,jpeg='treeview.jpg') si ottiene la Figura 1.1

Divedendo in due il dataset con la funzione

treepredict.divideset(treepredict.my_data,2,'yes') si ottengono:

- Set1 = ['slashdot', 'USA', 'yes', 18, 'None'], ['google', 'France', 'yes', 23, 'Premium'], ['digg', 'USA', 'yes', 24, 'Basic'], ['kiwitobes', 'France', 'yes', 23, 'Basic'], ['slashdot', 'France', 'yes', 19, 'None'], ['digg', 'New Zealand', 'yes', 12, 'Basic'], ['slashdot', 'UK', 'no', 21, 'None'], ['google', 'UK', 'yes', 18, 'Basic'], ['kiwitobes', 'France', 'yes', 19, 'Basic']]

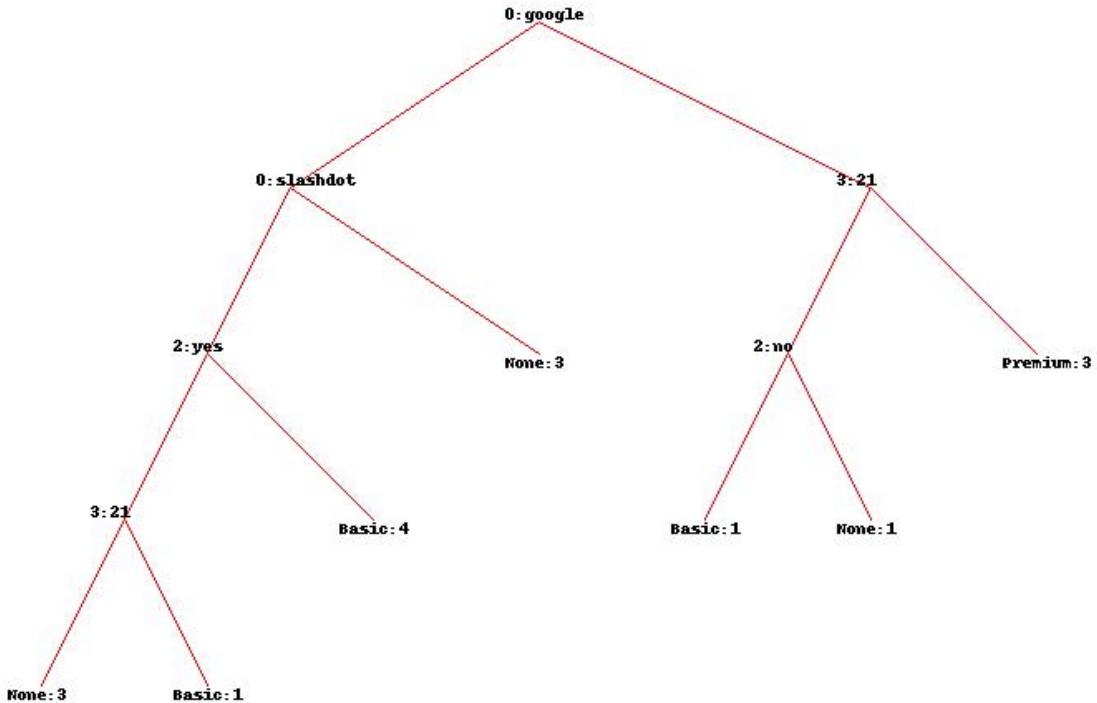


Figura 1.1.1

- Set2 = ['google', 'UK', 'no', 21, 'Premium'], ['(direct)', 'New Zealand', 'no', 12, 'None'], ['(direct)', 'UK', 'no', 21, 'Basic'], ['google', 'USA', 'no', 24, 'Premium'], ['digg', 'USA', 'no', 18, 'None'], ['google', 'UK', 'no', 18, 'None'], ['kiwitobes', 'UK', 'no', 19, 'None'], ['slashdot', 'UK', 'no', 21, 'None'].

L'entropia del Set1 è di 1,29 mentre la Geni impurity è di 0,53; il Set2 ha gli stessi valori del Set1.

Come si nota, dividendo il Data set in due sottoinsiemi si ottengono dei valori di entropia e Geni Impurity minori, questo vuol dire che dividere i dati sulla condizione 'L'utente legge la FAQ?' è una buona scelta; però modificando la condizione su cui eseguire lo 'split' dei dati ve ne sono altre migliori. La migliore che trovata è:

`treepredict.divideset(treepredict.my_data,3,22);` i cui sottoinsiemi risultanti sono:

- Set1:
 - A. Entropia = 1;
 - B. Gini Impurity = 0.5
- Set2:
 - A. Entropia = 1.28;
 - B. Geni Impurity = 0.54

Provando ora a classificare nuovi dati con la funzione:

"treepredict.classify(observation, tree)" e modificando le osservazioni, come
['google','USA','yes',21], si evince che l'algoritmo riesce sempre a trovare la
soluzione giusta.

Punto 2: Applicare DT Learning ad un Dataset

Dal sito UC Irvine ML Repository abbiamo scaricato il Dataset

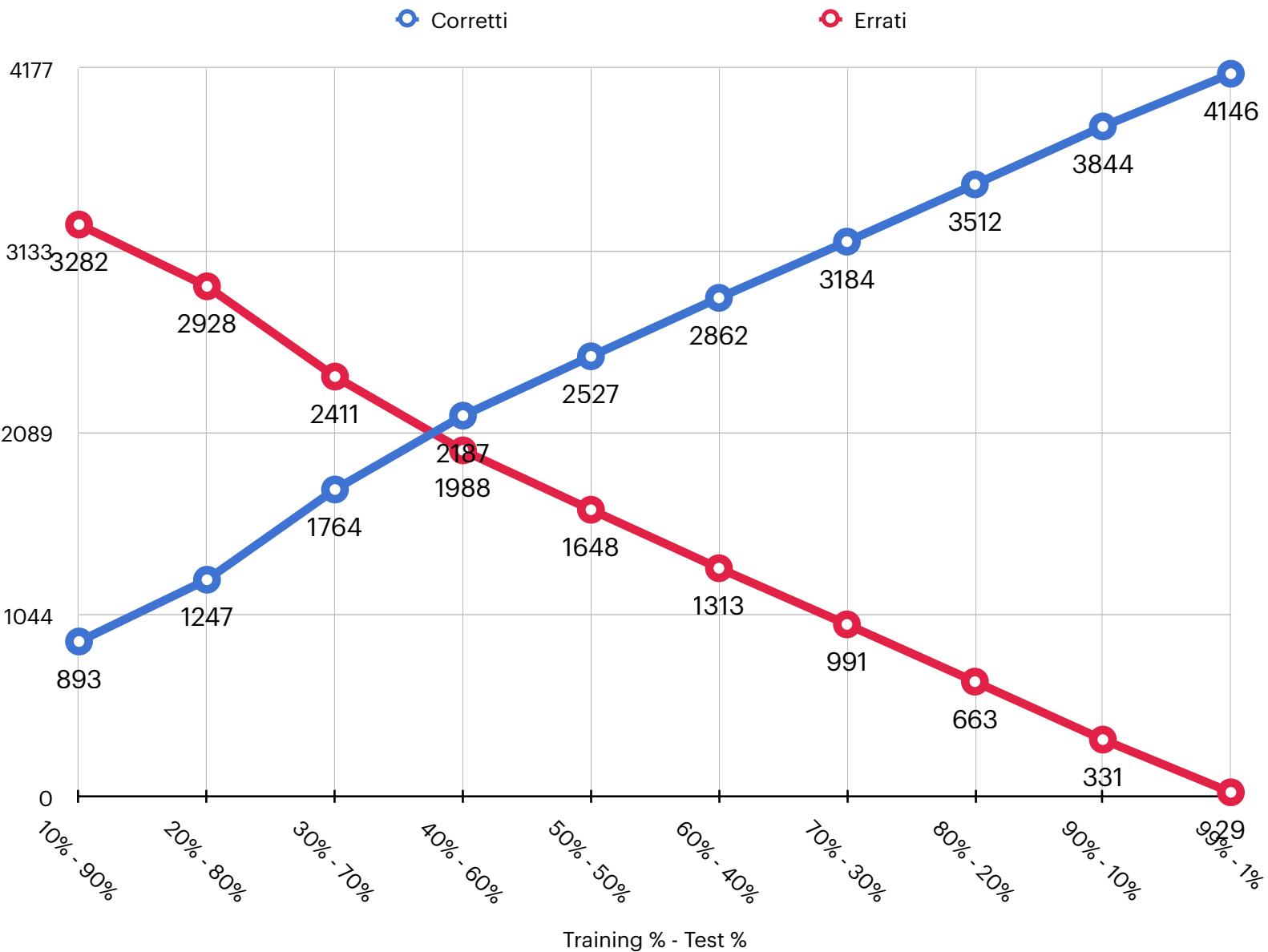
Abalone.data, il quale contiene 4177 istanze sugli abalone (un specie di conchiglie). La struttura dati è composta da otto attributi che danno le seguenti informazioni:

- Genere (Maschio , Femmina, Infante),
- Lunghezza (in millimetri),
- Diametro (in millimetri),
- Altezza (in millimetri),
- Peso intero (in grammi),
- Peso senza guscio (in grammi),
- Peso degli organi (in grammi),
- Peso della conchiglia (in grammi),
- Numero di anelli (i quali, aggiungendo 1,5, ci indica gli anni).

Un volta usato l'algoritmo per costruire l'albero delle decisioni si ottiene l'immagine a questo [link](#) (per una questione di dimensioni si usa solo il 10% di dati)

Punto 3: Visualizzare variazione accuratezza

Dividendo il DataSet in Training Set e Testing Set e cambiando le percentuali dei dati che ne fanno parte, si ottiene il seguente grafico il quale ci indica il numero di errori e di successi quando l'algoritmo di classificazione usato nel punto 1 viene usato.



Il codice e i dati usati per questa esercitazioni si trovano su questa [repository](#)

Esercizio 2: Agente e DT learning

Perché un agente, in grado di apprendere Alberi di Decisione, e' considerato Intelligente?

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

Nel mondo dell' intelligenza artificiale l' apprendimento permette di adattarsi a nuove circostanze, individuare e estrapolare schemi.

Un agente in grado di apprendere alberi di decisione, è in grado di elaborare una decisione.

Un albero di decisione rappresenta una funzione che prende in ingresso una coppia di attributi-valore e ritorna in uscita una soluzione (o decisione). Un albero di decisioni trova la soluzione andando ad effettuare una sequenza di test, ogni nodo dell'albero corrisponde ad un controllo su un valore di uno degli attributi in input.

Un albero di decisione è logicamente equivalente ad un'asserzione logica ,in cui l'obiettivo è quello di classificare l'esempio nella classe che si ritiene corretta se e solo se gli attributi in input soddisfano uno dei percorsi che permettono di raggiungere una foglia con il relativo valore di decisione. Questo implica che ogni forma di proposizione logica può essere espressa attraverso un albero di decisione.

Esercizio 3: A* Search

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.

L'algoritmo A* valuta i nodi da espandere combinando:

- La funzione $g(n)$, che indica il costo del nodo da raggiungere,
- La funzione $h(n)$, che indica il costo per andare dal nodo all'obiettivo e si basa su un'euristica, ovvero su delle informazioni note a priori dall'agente.

L'algoritmo è sia completo che ottimale, anche se la proprietà dell'essere ottimale dipende dall'ammissibilità dell'euristica. Un'euristica si dice ammissibile se il valore che essa stima è minore o uguale del costo reale per raggiungere l'obiettivo a partire dal nodo che stiamo considerando.

Un esempio di applicazione dell'algoritmo A* è il calcolo del miglior percorso per andare da Arad a Bucarest, la funzione $h(n)$ è la distanza in linea d'aria per andare dalla città in cui ci troviamo all'obiettivo. Una mappa delle distanze è mostrata in Figura 1.3.

Avendo come punto iniziale Arad, la sequenza delle azioni da eseguire è:

1. Arad: $g(n) = 0$; $h(n) = 366$ quindi $f(n) = 366$
2. Quindi espandiamo Arad ed otteniamo Sibiu, Timisoara e Zenind
 1. Sibiu: $140 + 254 = 294$
 2. Timisoara: $118 + 329 = 447$
 3. Zenind: $75 + 347 = 422$
3. Espandiamo quindi Sibiu, poiché ha il costo minore, otteniamo:
 1. Arad: $280 + 366 = 646$
 2. Fagaras: $239 + 176 = 415$
 3. Oraea: $291 + 380 = 671$
 4. Rimnicu Vilcea: $220 + 193 = 413$
4. Espandiamo Rimnicu Vilcea ed otteniamo:

1. Cracovia: $366 + 160 = 526$
2. Pitesti: $317 + 100 = 417$
3. Sibiu: $300 + 253 = 553$
5. Espandiamo quindi Fagaras, poiché il costo è di 415, mentre Pitesti è di 417, quindi otteniamo:
 1. Sibiu: $338 + 253 = 591$
 2. Bucarest: $450 + 0 = 450$
6. Espandiamo Pitesti che ha costo di 415 ed otteniamo:
 1. Bucarest: $418 + 0 = 418$
 2. Cracovia: $455 + 160 = 615$
 3. Rimnicu Vilcea: $414 + 193 = 607$

La scelta migliore è quindi: ARAD - SIBIU - RIMNICU VILCEA - PITESTI - BUCAREST il cui costo è di 418

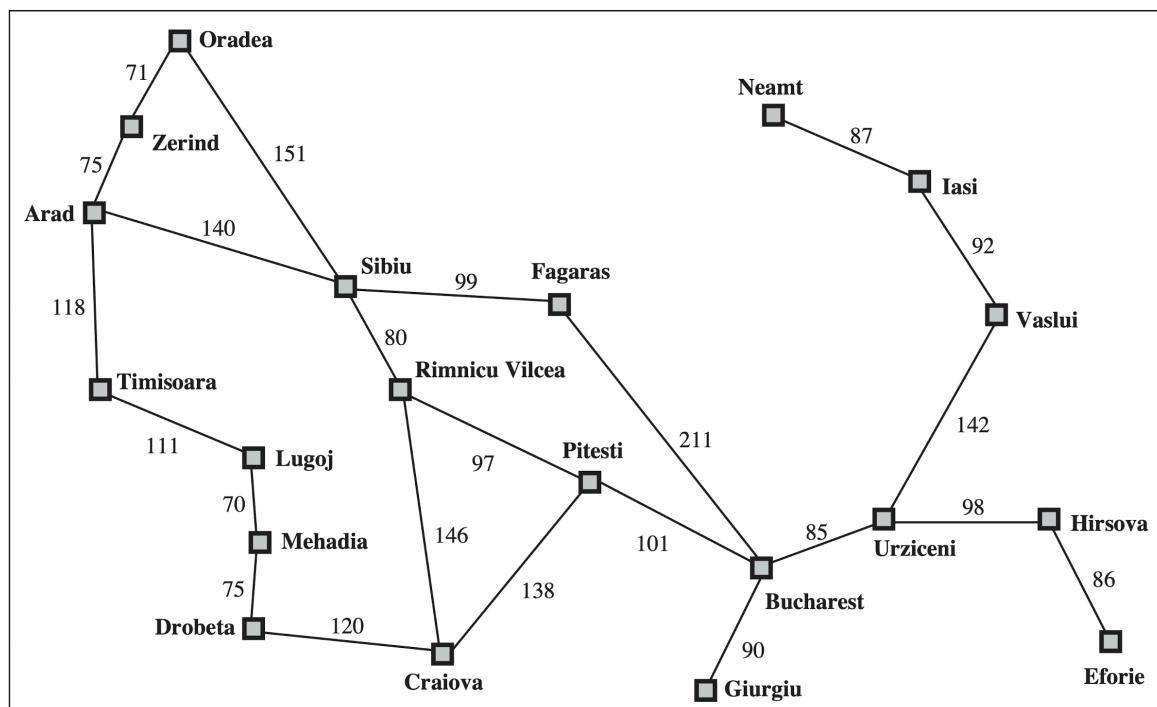


Figura 1.3

Esercizio 4: Utilizzo Albero Decisione

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

Un esempio di utilizzo dell'albero di decisione è quello visto nell'[Esercizio 1](#) in cui un sito web che offre un servizio a pagamento offre dei giorni di prova gratuita e collaziona i dati degli utenti che vi si iscrivono quali: Sito da cui sono stati indirizzati al servizio, la posizione, se leggono le FAQ, le pagine viste, che tipo di iscrizione hanno scelto.

Creando quest'albero è possibile, per l'amministratore del sito, focalizzare la propria offerta nelle regioni dove il numero di utenti che si iscrive è maggiore, oppure modifiche quella nelle regioni dove gli utenti non si abbonano al servizio.

Esercizio 5: Apprendimento di un Decision Tree

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

Il processo di apprendimento di un albero di decisione può essere effettuato tramite la funzione DTL (Decision Tree Learning), un'operazione ricorsiva, che prende in input gli esempi, gli attributi e una classificazione:

- Gli esempi sono i dati iniziali, nell'Esercizio 1 essi sono i dati delle varie conchiglie analizzate
- Gli attributi sono la descrizione dei dati, nell'Esercizio 1 sono per esempio Altezza, Larghezza etc.

La funzione viene eseguita secondo i seguenti passaggi:

1. La funzione controlla se l'insieme degli esempi è vuoto;
 1. se lo è, allora restituisce la classificazione creata;
 2. altrimenti se gli esempi hanno la stessa classificazione, allora ritorna quest'ultima.
2. Se l'insieme degli attributi è vuoto allora restituisce la MODA degli esempi, ovvero la classe che ha più occorrenze tra gli esempi;
3. Altrimenti:
 1. sceglie l'attributo tra quelle ricevuti in input
 2. Crea un nuovo albero con nodo radice l'attributo scelto
 3. Per ogni valore dell'attributo:
 1. Crea un sottoinsieme di esempi contenente solo quelli il cui attributo ha il valore in questione
 2. Crea un sotto-albero richiamando se stessa, con parametri: il sottoinsieme di esempi, l'insieme degli attributi (escluso quello scelto) e come classificazione, la MODA degli esempi
 3. Aggiunge il sotto-albero come ramo dell'albero principale
 4. Restituisce l'albero creato

Il codice è il seguente:

```

function DECISION-TREE-LEARNING(examples , attributes ,
parent examples ) returns atree
  if examples is empty then
    return PLURALITY-VALUE(parent examples )
  else if all examples have the same classification then
    return the classification
  else if attributes is empty then
    return PLURALITY-VALUE(examples )
  else
    A ← argmaxa ∈ attributes I MPORTANCE(a, examples )
    tree ← a new decision tree with root test A
    for each value vk of A do
      exs ←{e : e ∈ examples and e.A = vk}
      subtree ← DECISION-TREE-LEARNING(exs ,
attributes - A, examples )
      add a branch to tree with label (A = vk ) and subtree
    subtree
  return tree

```

Se per esempio considerassimo un sistema che ci dica se è conveniente aspettare in coda per entrare ad un ristorante con la tabella degli esempi, in Figura 1.5.1, si otterrebbe l'albero, ancora non completo, in Figura 1.5.2.

Example	Input Attributes										Goal <i>WillWait</i>
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
x ₁	Yes	No	No	Yes	Some	\$\$\$	No	Yes	French	0–10	y ₁ = Yes
x ₂	Yes	No	No	Yes	Full	\$	No	No	Thai	30–60	y ₂ = No
x ₃	No	Yes	No	No	Some	\$	No	No	Burger	0–10	y ₃ = Yes
x ₄	Yes	No	Yes	Yes	Full	\$	Yes	No	Thai	10–30	y ₄ = Yes
x ₅	Yes	No	Yes	No	Full	\$\$\$	No	Yes	French	>60	y ₅ = No
x ₆	No	Yes	No	Yes	Some	\$\$	Yes	Yes	Italian	0–10	y ₆ = Yes
x ₇	No	Yes	No	No	None	\$	Yes	No	Burger	0–10	y ₇ = No
x ₈	No	No	No	Yes	Some	\$\$	Yes	Yes	Thai	0–10	y ₈ = Yes
x ₉	No	Yes	Yes	No	Full	\$	Yes	No	Burger	>60	y ₉ = No
x ₁₀	Yes	Yes	Yes	Yes	Full	\$\$\$	No	Yes	Italian	10–30	y ₁₀ = No
x ₁₁	No	No	No	No	None	\$	No	No	Thai	0–10	y ₁₁ = No
x ₁₂	Yes	Yes	Yes	Yes	Full	\$	No	No	Burger	30–60	y ₁₂ = Yes

Figura 1.5.1

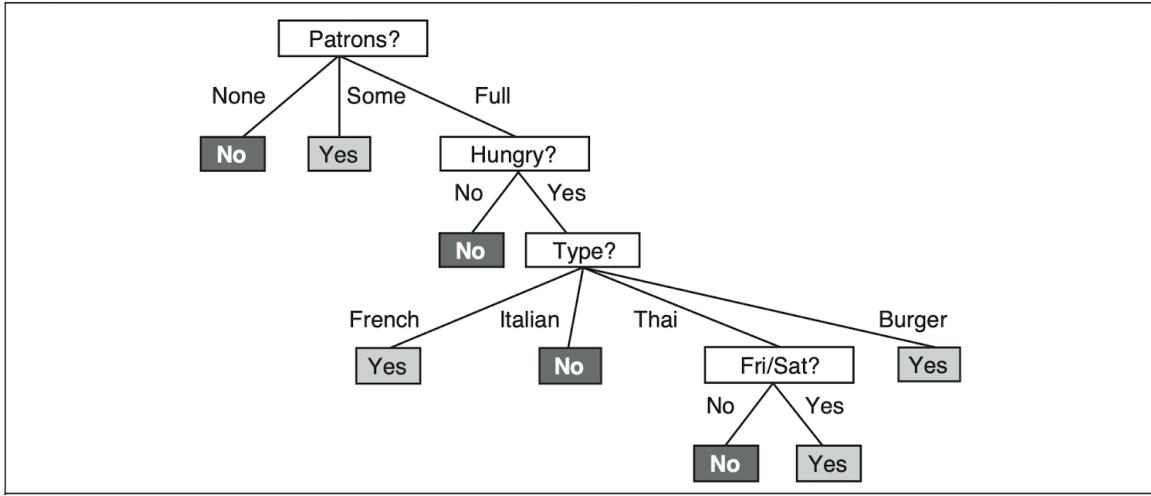


Figura 1.5.2

Esercizio 6: Spazio degli stati

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.

Prima di definire lo stato degli spazi dobbiamo prima definire le componenti da cui è composto :

- Stato iniziale: il punto dal cui l'agente fa partire la sua elaborazione,
- Azioni: le varie azioni che un agente può intraprendere,
- Il modello delle transazioni: indica il risultato che ogni azione ha quando l'agente si trova in un determinato stato,

Lo spazio degli stati è, quindi, il set di tutti gli stati raggiungibili dello stato iniziali attraverso una qualsiasi sequenza di azioni. Lo stato degli spazi forma il grafo degli stati, ovvero una rete composta da:

- Nodi, che rappresentano gli stati,
- Link, che indicano le azioni per andare da uno stato ad un altro.

L'albero di ricerca è un possibile sequenza di azioni che porta dallo stato iniziale alla soluzione. L'albero è composto da:

- Il nodo radice: indica lo stato iniziale
- Il nodo foglia: che indica un possibile stato finale
- La frontiera: l'insieme di tutti i nodi foglia

Ogni nodo (tranne i nodi foglia) sono nodi genitori: possono essere espansi per raggiungere altri nodi (i figli) muovendosi così verso la soluzione.

L'algoritmo generale dell'albero delle decisioni è:

```
function TREE-SEARCH(problema) return a soluzione, or  
fallimento
```

inizializza la frontiera usando lo stato iniziale
del *problema*

```

loop do
    if la frontiera è vuota then return fallimento
    sceglie un nodo foglia e lo rimuove della
    frontiera
    if il nodo contiene l'obiettivo then return la
    soluzione corrispondente
    espandi il nodo scelto, aggiungi i nodi trovati
    alla frontiera

```

Tutti gli algoritmi di ricerca si basano su questa struttura, in cui:

1. all'inizio si inizializza la frontiera aggiungendo il nodo radice.
2. Dopodiché inizia il loop che scorre l'albero
 1. si controlla se la frontiera è vuota (all'inizio non lo è poiché vi è la radice), se lo è, allora, è fallita la ricerca
 2. altrimenti sceglie uno dei nodi della frontiera e poi lo rimuove da essa (il modo in cui si sceglie il nodo varia a secondo dell'algoritmo e può essere casuale o basato su euristiche)
 3. Se il nodo scelto è l'obiettivo, allora ritorna la soluzione (dove la soluzione è la sequenza di azioni intraprese per trovare la soluzione)
 4. Altrimenti, espandi il nodo scelto e aggiungi i nuovi nodi alla frontiera e re-inizia il loop

Gli algoritmi principali per la ricerca in un albero si dividono in due famiglie: ricerca non informata e ricerca informata

Gli algoritmi di ricerca non informata non hanno nessuna informazione sul numero di passi o sul costo del cammino dallo stato attuale alla soluzione. I principali sono:

- Breadth-first search: la frontiera viene memorizzata all'interno di una coda con politica FIFO, ovvero si espando i nodi aggiunti prima, quindi, partendo dalla radice, si espandono prima i nodi figli di un determinato livello e poi si scende al livello successivo fino a trovare la soluzioni, oppure, fino all'ultima foglia dell'albero. Questa è una strategia **completa**,

ovvero troverà sicuramente la soluzione (se presente), però ha un **problema di spazio**, poiché la frontiera cresce in modo esponenziale mano che scendiamo nell'albero. È **non ottimale** poiché non trova la soluzione migliore, ma la prima ottenuta;

- Depth-first search: la frontiera viene memorizzata all'interno di una coda con politica LIFO, ovvero espando prima gli ultimi nodi aggiunti, quindi partendo dalla radice espando in nodi mano mano che li incontro. Se arrivo ad un foglia, ed essa non è la soluzione, devo risalire nell'albero e seguire un percorso differente. È una soluzione **non completa** poiché, se sono presenti dei cicli l'albero, avrebbe profondità infinita. È **non ottimale** poiché non trova la soluzione migliore, ma la prima ottenuta.

Gli algoritmi di ricerca informata si basano sul concetto di euristiche, ovvero un conoscenza nota dall'algoritmo a priori, quindi l'efficienza dell'algoritmo dipende anche dal tipo di euristica che si sceglie. I principali algoritmi sono:

- Greedy search: l'euristica è una funzione $h(n)$ che stima il costo per andare dal nodo che scelgo all'obiettivo. L'algoritmo ha bisogno di una struttura dati (chiamata *closed*) che serve ad indicare i nodi esplorati in modo da evitare i cicli. In questo caso l'algoritmo è **completo** ma la soluzione è **non ottimale** poiché sceglio la prima soluzione, che potrebbe non essere la migliore.
- A* search: la funzione che sceglie il nuovo nodo non si basa solo sull'euristica, ma anche sul costo necessario per raggiungere il nodo dallo stato iniziale. L'algoritmo A* è **completo**, la soluzione è **ottimale**, solo se l'euristica è ammissibile. Un'euristica si dice **ammissibile** se il valore che essa stima è minore o uguale al costo per raggiungere l'obiettivo dal nodo in cui ci troviamo. Un esempio di applicazione di A* si trova all'Esercizio 3

Esercitazione 2: Algoritmi di ricerca locale

Esercizio 1: Principali algoritmi di ricerca Locale

Punto 1: Evolutionary Optimization

Dopo aver scaricato il codice della cartella del corso sulla piattaforma Google Drive ed averlo eseguito tramite l'editor 'Spyder', ho effettuato delle correzioni generali per adattarlo alla versione di Python installata sul mio elaboratore. Dopodiché ho eseguito tutte istruzioni presenti nel Capitolo 5 del libro 'Programming Collective Intelligence' per capire il funzionamento delle varie funzioni presenti nel codice.

I dati usati sono :

- `people = [('Seymour','BOS'), ('Franny','DAL'), ('Zooey','CAK'), ('Walt','MIA'), ('Buddy','ORD'), ('Les','OMA')]`
- `destination='LGA'`
- *Una lista dei possibili voli contenuti nel file 'schedule.txt'*
- *L'array s = [1,4,3,2,7,3,6,3,2,4,5,3], che rappresenta il volo che ogni persona sceglie (0 indica il primo volo della giornata, 1 il secondo e cos' via)*

Con l'array di voli indicato, lo schedule, ottenuto attraverso la funzione `printschedule(s)` sarebbe il seguente:

Seymour	BOS	8:04-10:11	\$ 95	12:08-14:05	\$ 142
Franny	DAL	10:30-14:57	\$ 290	9:49-13:51	\$ 229
Zooey	CAK	17:08-19:08	\$ 262	10:32-13:16	\$ 139
Walt	MIA	15:34-18:11	\$ 326	11:08-14:38	\$ 262
Buddy	ORD	9:42-11:32	\$ 169	12:08-14:47	\$ 231
Les	OMA	13:37-15:08	\$ 250	11:07-13:24	\$ 171

Con un costo, ottenuto con la funzione `schedulecost(s)`, di 4635; questo poiché la funzione aggiunge un costo per ogni minuto sull'aereo (preferendo i voli diretti), un costo per ogni minuto di attesa in aeroporto e aggiunge un penalità se l'auto in affitto per raggiungere l'aeroporto viene consegnata oltre il tempo limite.

Dopodiché si possono provare le varie funzioni di ottimizzazione:

A. Il primo è il **Random Searching** in cui si cerca in modo casuale la soluzione, questa strategia non è efficiente in quanto prendendo valori casuali, non è detto che troverà i migliori, anzi, come nell'esecuzione effettuata, può trovare anche valori peggiori. La funzione usata è: *randomoptimize(domain, costf)*; dove:

- Domain è una array composto da tuple di due valori (il massimo e il minimo di ogni variabile) e la lunghezza è il numero totale di voli come per 's'
- Costf è la funzione "schedulecost" che calcola il costo di ogni valore scelto nel dominio

Dopo aver eseguito l'algoritmo lo schedule ha un costo di 6467 ed è:

1. Seymour BOS 18:34-19:36 \$136 13:39-15:30 \$ 74
2. Franny DAL 12:19-15:25 \$342 15:49-20:10 \$497
3. Zooey CAK 12:08-14:59 \$149 6:58- 9:01 \$238
4. Walt MIA 9:15-12:29 \$225 8:23-11:07 \$143
5. Buddy ORD 9:42-11:32 \$169 17:06-20:00 \$ 95
6. Les OMA 11:08-13:07 \$175 16:35-18:56 \$144

B. Il secondo è l'**Hill Climbing** che parte da un valore casuale e poi si muove nei valori vicini, fino a trovare quello più basso, che, però, è un minimo locale, ovvero la funzione troverà il valore con costo minore vicino al punto di inizio, ignorando la presenza di valori con il costo più basso. La funzione eseguita è *hillclimb(domain, costf)* e dopo la sua esecuzione lo schedule ha un costo di 4353 ed è:

5. Seymour BOS 13:40-15:37 \$138 18:24-20:49 \$124
6. Franny DAL 10:30-14:57 \$290 9:49-13:51 \$229
7. Zooey CAK 13:40-15:38 \$137 18:17-21:04 \$259
8. Walt MIA 11:28-14:40 \$248 6:33- 9:14 \$172
9. Buddy ORD 14:22-16:32 \$126 7:50-10:08 \$164

C. La terza è il **Simulated Annealing** che è simile all'hill climb. Esso si ripete un numero T di volte, partendo da un valore casuale. Dopodiché ad ogni iterazione sceglie un valore vicino e calcola il costo prima e dopo il cambiamento, se il nuovo costo è minore, allora questo valore è il migliore, altrimenti accetta questo valore con probabilità

$p = e^{((-highcost-lowcost)/(numeroIterazioni))}$, e si sposta. La funzione eseguita è annealingoptimize(domain,costf) e dopo la sua esecuzione lo schedule ha un costo di 3076 ed è il seguente:

1. Seymour BOS 12:34-15:02 \$109 12:08-14:05 \$142
2. Franny DAL 10:30-14:57 \$290 17:14-20:59 \$277
3. Zooey CAK 12:08-14:59 \$149 13:37-15:33 \$142
4. Walt MIA 11:28-14:40 \$248 12:37-15:05 \$170
5. Buddy ORD 9:42-11:32 \$169 12:08-14:47 \$231
6. Les OMA 12:18-14:56 \$172 12:31-14:02 \$234

D. L'ultimo è il **Genetic Algorithms**, esso crea un set di valori casuali e calcola il costo di ognuno di essi; dopodiché combina i risultati con il costo minore. La combinazione avviene attraverso due strategie:

- Crossover: nella quale si prendono dei valori delle primi due soluzioni e li si mescola;
- Mutazione: che cambia alcuni valori casuali all'interno delle soluzioni ottenute.

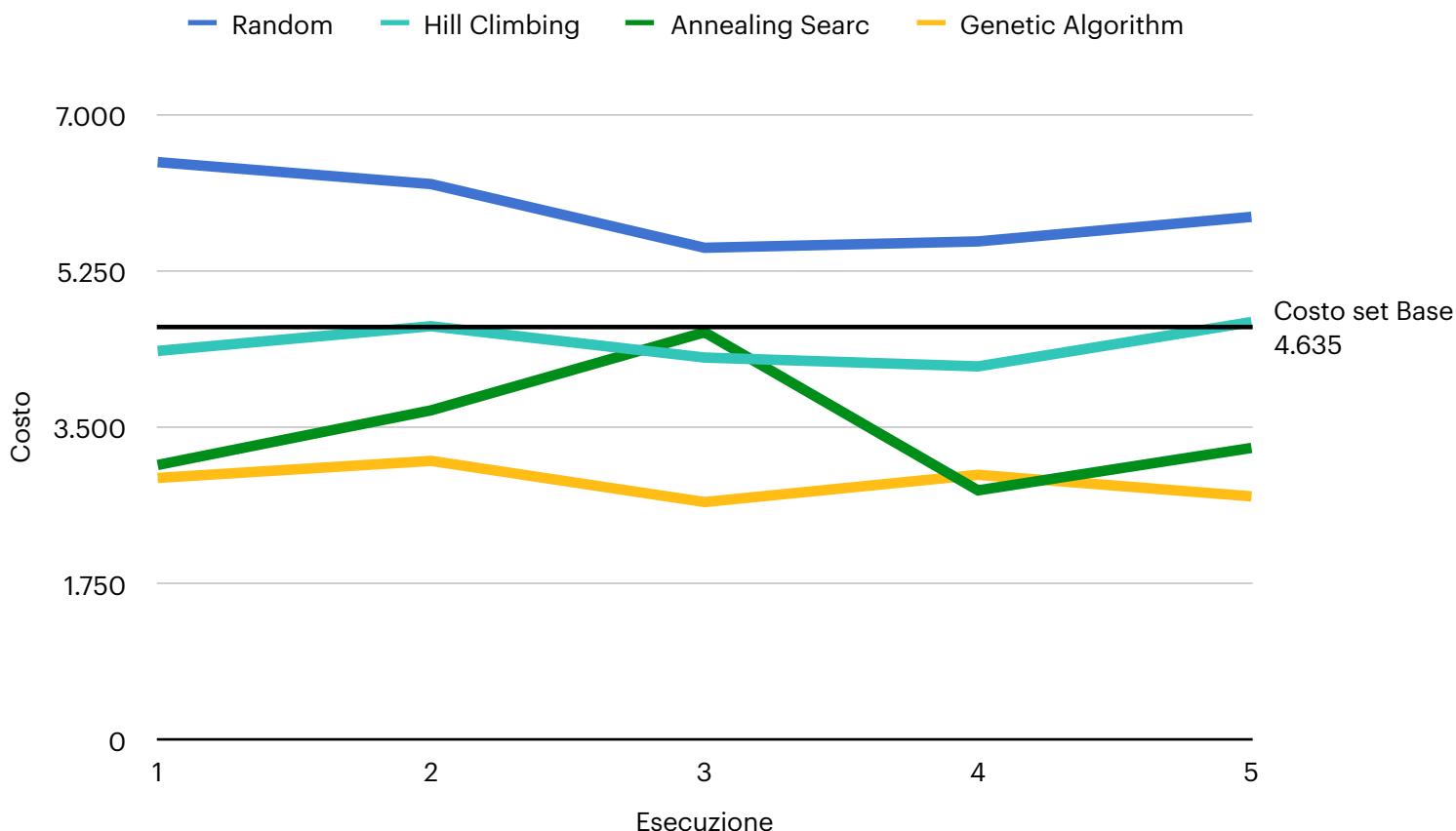
La funzione eseguita è geneticoptimize(domain,costf) e dopo la sua esecuzione lo schedule ha un costo di 2931 ed è il seguente:

1. Seymour BOS 13:40-15:37 \$138 10:33-12:03 \$ 74
2. Franny DAL 6:12-10:22 \$230 9:49-13:51 \$229
3. Zooey CAK 13:40-15:38 \$137 8:19-11:16 \$122

4. Walt MIA 11:28-14:40 \$248 8:23-11:07 \$143
5. Buddy ORD 14:22-16:32 \$126 7:50-10:08 \$164
6. Les OMA 15:03-16:42 \$135 8:04-10:59 \$136

Ognuno di questi algoritmi, eseguito più volte, restituisce risultati diversi.

Eseguendoli 5 volte si ottiene il seguente grafico:



Dal grafico si nota come l'Hill climbing, l'Annealing search e il Genetic Algorithm sono i migliori, mentre il Random Search ottiene valori non ottimi.

Punto 2: Ottimizzare una funzione

Applicare le strategie del punto precedente per trovare il valore minore della funzione:

$$f(x) = \begin{cases} 10 & \text{if } x < 5.2 \\ x^2 & \text{if } 5.2 \leq x \leq 20 \\ \cos(x) + 160x & \text{if } x > 20 \end{cases} \quad \forall x \in \{-100 \leq x \leq 100\}$$

Il cui grafico è mostrato in Figura 2.1.2.1

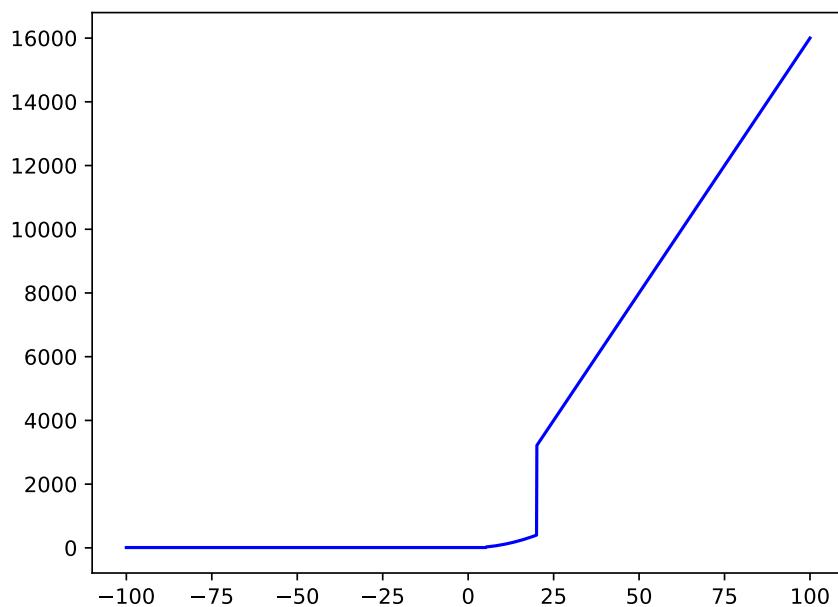


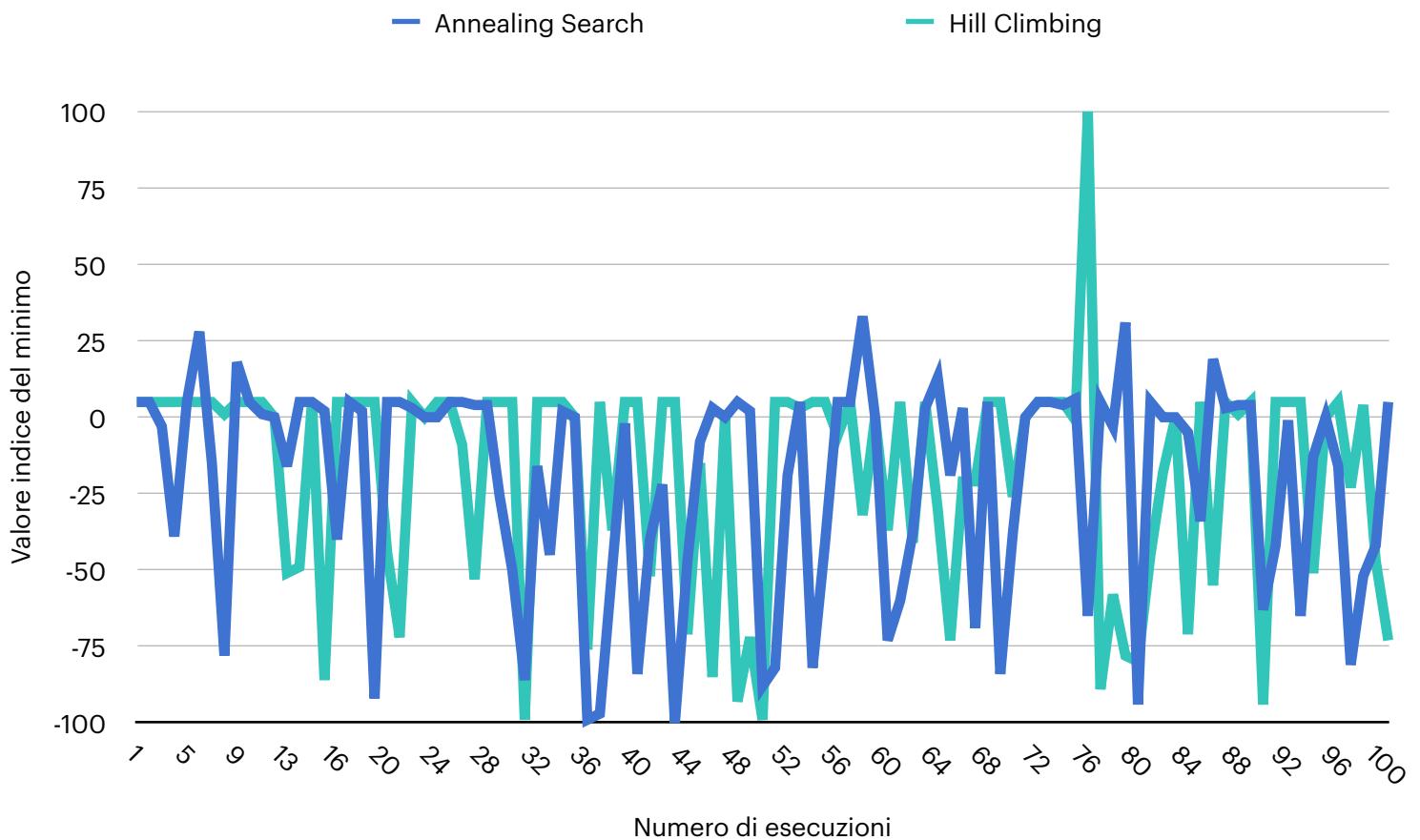
Figura 2.1.2.1

Per fare ciò si definisce la seguente funzione per calcolare il costo:

```
def function(x):
    if x < 5.2:
        return 10
    elif x >= 5.2 and x <= 20:
        return x*x
    elif x > 20:
        return math.cos(x) + 160*x
    return 0

def costf(x):
    return function(x[0])
```

Dopodiché definiamo il domain = (-100,100) e si possono così eseguire le funzioni precedenti. È stato deciso di effettuare un confronto tra l’Hill climbing e l’Annealing Search. Eseguendo 100 volte questi algoritmi si ottiene il seguente grafico:



Da questo si evince che in media, l’Annealing Search, riesce a trovare gli indice dei minimi ovvero le ascisse che hanno ordinata minima (che sono tutte le $x < 5.2$, come mostrato in [Figura 2.1](#)), più volte rispetto all’Hill climbing

La repository che contiene il codice per questo esercizio si trova al seguente link: <https://www.icloud.com/iclouddrive/0kXpu6vF5F33T9OAFdwpuVRFg#Codice>.

Esercizio 2: Map coloring

Punto 1: Colore la mappa della regione Campania con 2 colori

Per questo esercizio è richiesto di colorare la mappa delle province della regione Campania con solo 2 colori (Rosso e Blu)



Per fare ciò si usa la strategia della propagazione dei vincoli la quale prevede la divisione del problema in:

- Variabili: gli oggetti da elaborare,
- Dominio: i valori che ogni variabile può assumere,
- Vincoli: le varie regole per l'assegnazione dei valori.

Essa può essere effettuato in modo semplice con l'algoritmo di **Backtracking-Search** che prima effettua l'assegnazione dei colori ad una variabile e dopo controlla se essa rispetta i vincoli, questa soluzione però non è efficiente.

Per creare un algoritmo ottimale, allora, si possono aggiungere delle funzioni euristiche; le due migliori sono:

- **Degree Heuristic** : si sceglie la variabile con il numero più alto di vincoli e gli si assegna un valore, rispettando i vincoli con altre variabili già assegnate,
- **Least Constraining Value**: si sceglie il valore che rimuove meno vincoli possibile alle variabili adiacenti a quella selezionata.

Per colorare la mappa delle provincie della regione Campania si evidenziano:

- Variabili: $X = \{NA, SA, AV, BN, CE\}$

- Valori: $D_i = \{\text{rosso}, \text{blu}\}$

- Vincoli:

$$C = \{NA \neq CE, CE \neq BN, BN \neq NA, BN \neq AV, AV \neq NA, AV \neq SA, SA \neq NA\}$$

Applicando la **Degree Heuristic** i passi che si seguono (Figura 2.1.1) sono:

1. Si seleziona la provincia di Napoli, poiché è quella con più vincoli (4), per la quale scegliamo il rosso, e rimuovo il rosso come possibile valore da tutte le province che hanno un vincolo con Napoli,
2. Dopodiché si sceglie la prossima provincia, dato che hanno tutti lo stesso numero di vincoli (tranne Salerno), scegliamo Caserta e, abbiamo un solo valore possibile, ovvero il blu, e lo rimuoviamo da tutte le province che hanno vincoli con Caserta
3. Dopodiché si sceglie la prossima provincia, dato che hanno tutti lo stesso numero di vincoli, scegliamo Benevento e l'algoritmo fallisce poiché non ho più colori disponibili

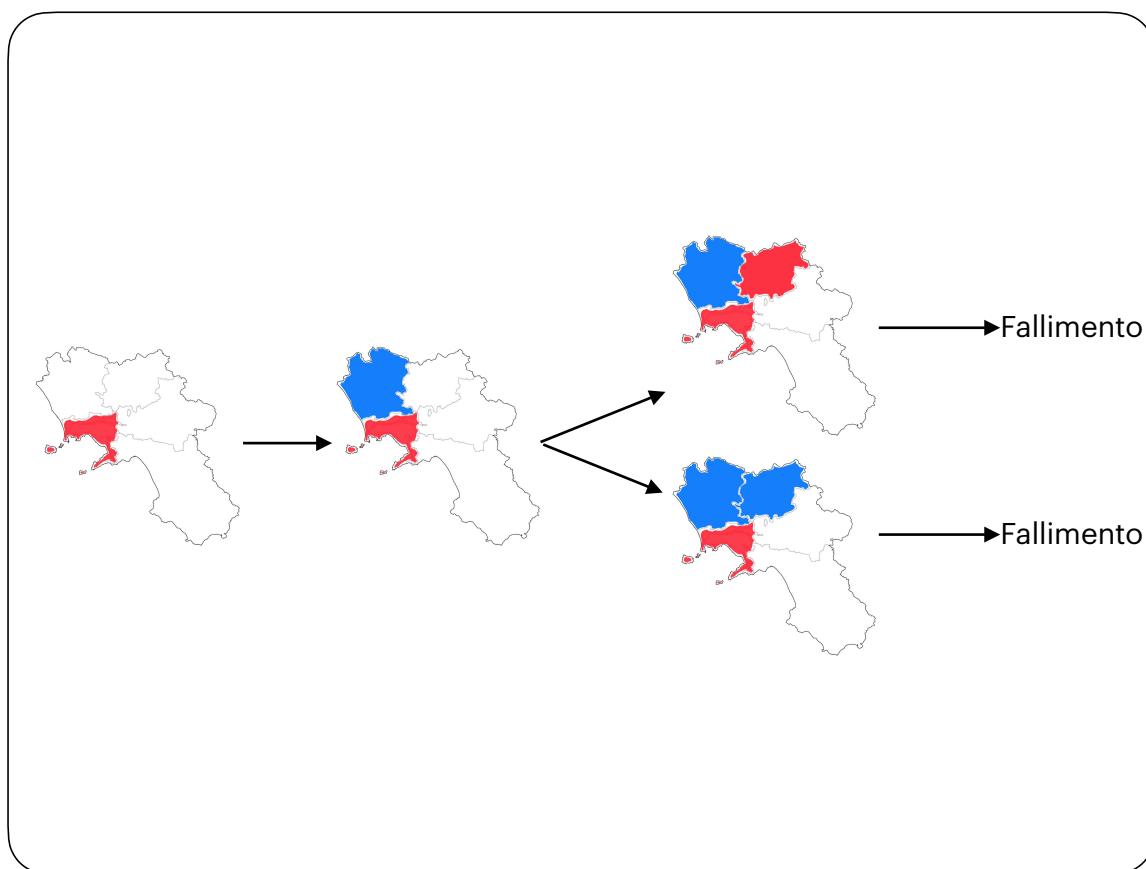


Figura 2.2.1.1

Applicando la **Least Constraining Value** i passi che si seguono (Figura 2.1.2) sono:

1. Si seleziona una provincia di Salerno, poiché è quella con meno vincoli (2), per la quale scegliamo il blu, lo rimuovo come valore dalle variabili con cui Salerno è vincolata
2. Dopodiché si sceglie la prossima provincia, dato che hanno tutti lo stesso numero di vincoli (tranne Napoli) scegliamo Avellino e, abbiamo un solo valore disponibile ovvero il rosso
3. L'algoritmo fallisce poiché la provincia di Napoli non ha più colori possibili

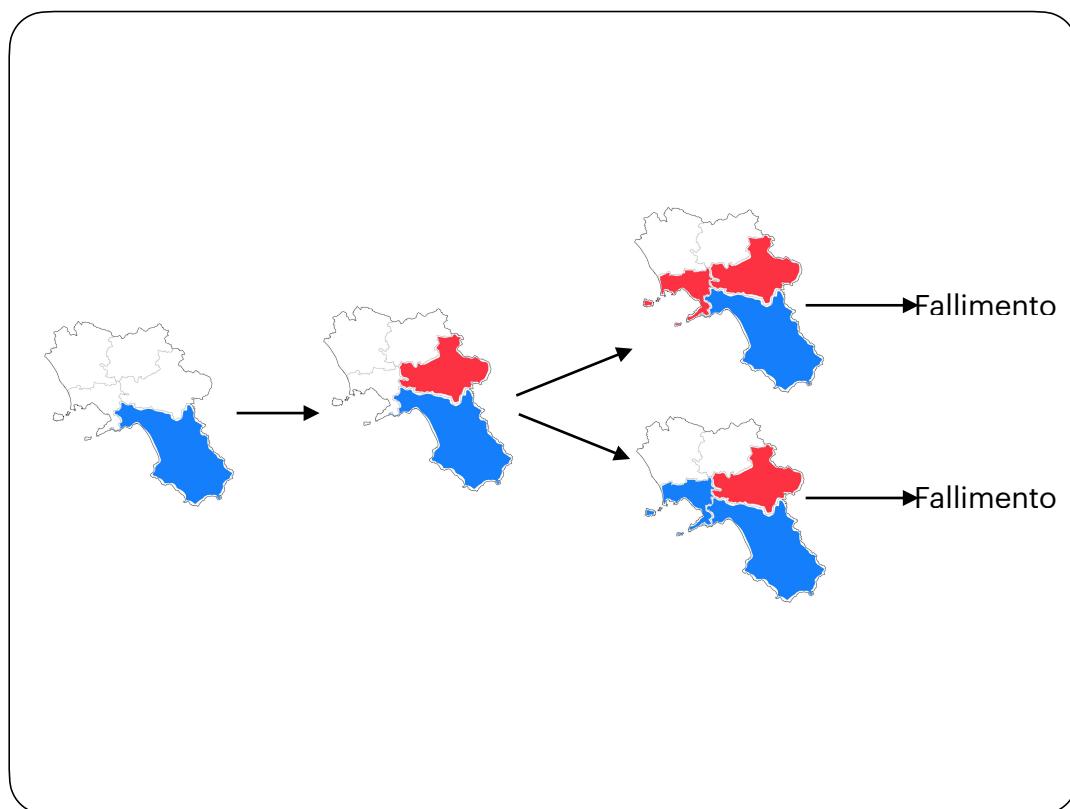
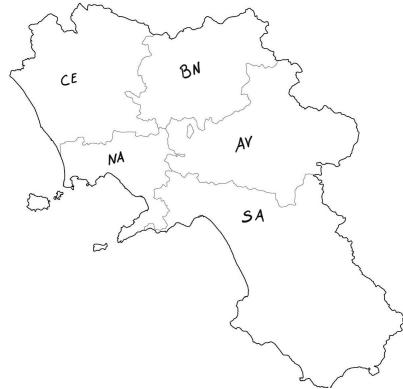


Figura 2.2.1.2

Da come si vede, è impossibile colorare la mappa delle province con solo due colori

Punto 2: Colore la mappa della regione Campania con 3 colori

Per questo esercizio si deve colorare la mappa delle province della regione Campania con solo 3 colori (Rosso, Blu e Giallo)



Per colorare la mappa delle provincie della regione Campania si evidenziano:

- Variabili: $X = \{NA, SA, AV, BN, CE\}$
- Valori: $D_i = \{\text{rosso, blu, giallo}\}$
- Vincoli:

$$C = \{NA \neq CE, CE \neq BN, BN \neq NA, BN \neq AV, AV \neq NA, AV \neq SA, SA \neq NA\}$$

Applicando la **Degree Heuristic** i passi che si seguono (Figura 2.2.1) sono:

1. Si seleziona la provincia di Napoli, poiché è quella con più vincoli (4), per la quale scegliamo il rosso, e rimuovo il rosso come possibile valore da tutte le province vincolate con Napoli
2. Dopodiché si sceglie la prossima provincia, dato che hanno tutti lo stesso numero di vincoli (tranne Salerno), scegliamo Caserta, e abbiamo due valori possibili, ovvero il blu e il giallo; scelgo il blu e lo rimuoviamo da tutte le province che hanno vincoli con Caserta
3. Dopodiché si sceglie la prossima provincia, dato che hanno tutti lo stesso numero di vincoli (tranne Salerno) scegliamo Benevento e ho un colore disponibile, ovvero il giallo, e lo rimuovo da tutte le province vincolate con Benevento

4. Scelgo come prossima provincia, ed Avellino è quella con più vincoli, ed ho un solo colore disponibile, ovvero il blu, e lo rimuovo da tutte le province vincolate ad Avellino
5. Scelgo l'ultima provincia, ovvero Salerno, ed ha un solo colore disponibile, ovvero il giallo, la mappa allora è colorata.

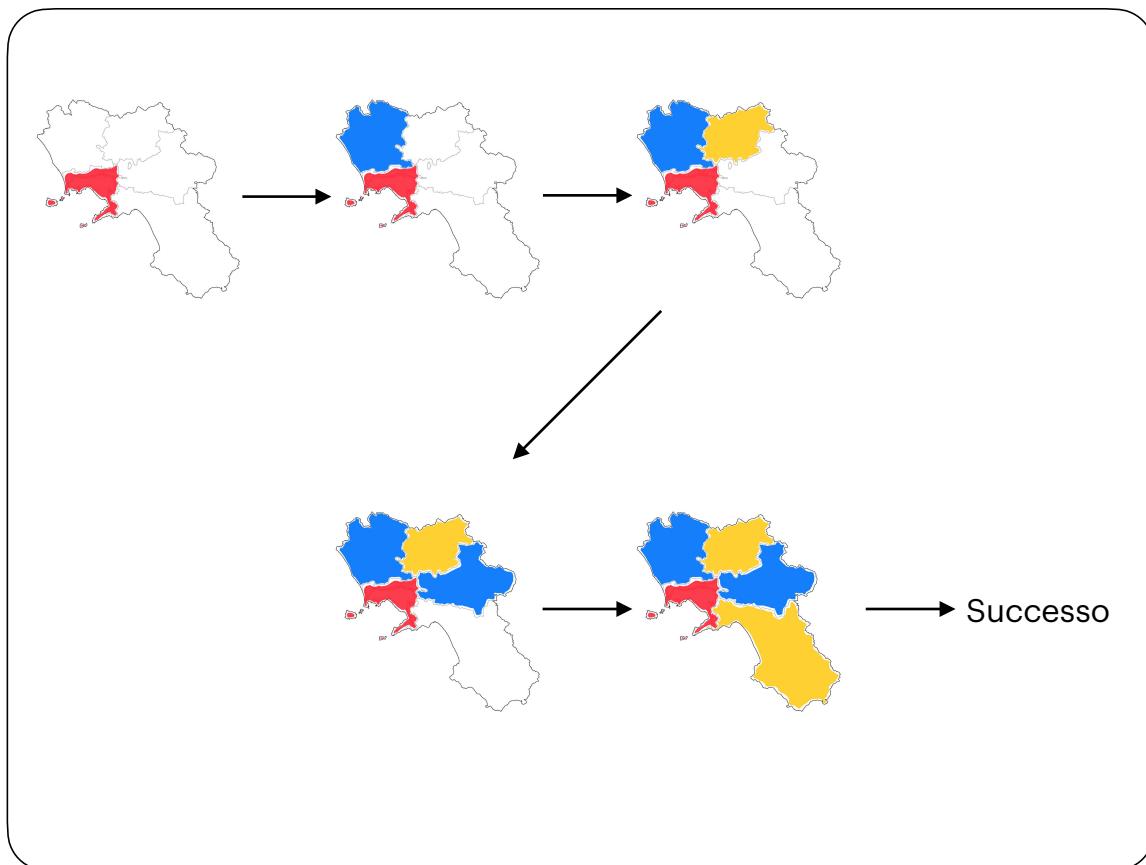
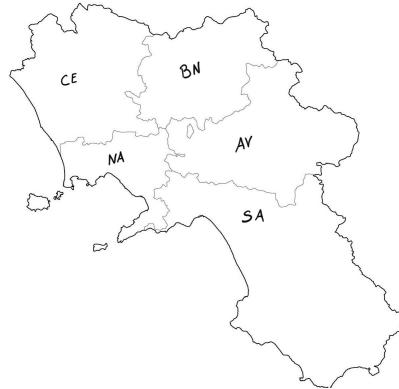


Figura 2.2.2.1

Punto 3: Colore la mappa della regione Campania con 4 colori

Per questo esercizio si deve colorare la mappa delle province della regione Campania con solo 4 colori (Rosso, Blu, Giallo e Verde)



Per colorare la mappa delle provincie della regione Campania si evidenziano:

- Variabili: $X = \{NA, SA, AV, BN, CE\}$
- Valori: $D_i = \{\text{rosso, blu, giallo, verde}\}$
- Vincoli:

$$C = \{NA \neq CE, CE \neq BN, BN \neq NA, BN \neq AV, AV \neq NA, AV \neq SA, SA \neq NA\}$$

Applicando la **Degree Heuristic** i passi che si seguono (Figura 2.3.1) sono:

1. Si seleziona la provincia di Napoli, poiché è quella con più vincoli (4), per la quale scegliamo il rosso, e rimuovo il rosso come possibile valore da tutte le province vincolate con Napoli
2. Dopodiché si sceglie la prossima provincia, dato che hanno tutti lo stesso numero di vincoli scegliamo Caserta e, abbiamo tre valori possibili, ovvero il blu, il giallo e il verde; scelgo il blu e lo rimuoviamo da tutte le province che hanno vincoli con Caserta
3. Dopodiché si sceglie la prossima provincia, dato che hanno tutti lo stesso numero di vincoli scegliamo Benevento e ho due colori disponibili, ovvero il giallo e il verde, scelgo il giallo e lo rimuovo da tutte le province vincolate con Benevento

4. Scelgo come prossima provincia Avellino ed ho due colori disponibili, ovvero il blu e il verde, scelgo il verde e lo rimuovo da tutte le province vincolate ad Avellino
5. Scelgo l'ultima provincia, ovvero Salerno, ed ha due colori disponibili, ovvero il giallo ed il blu, scelgo il blu e la mappa allora è colorata.

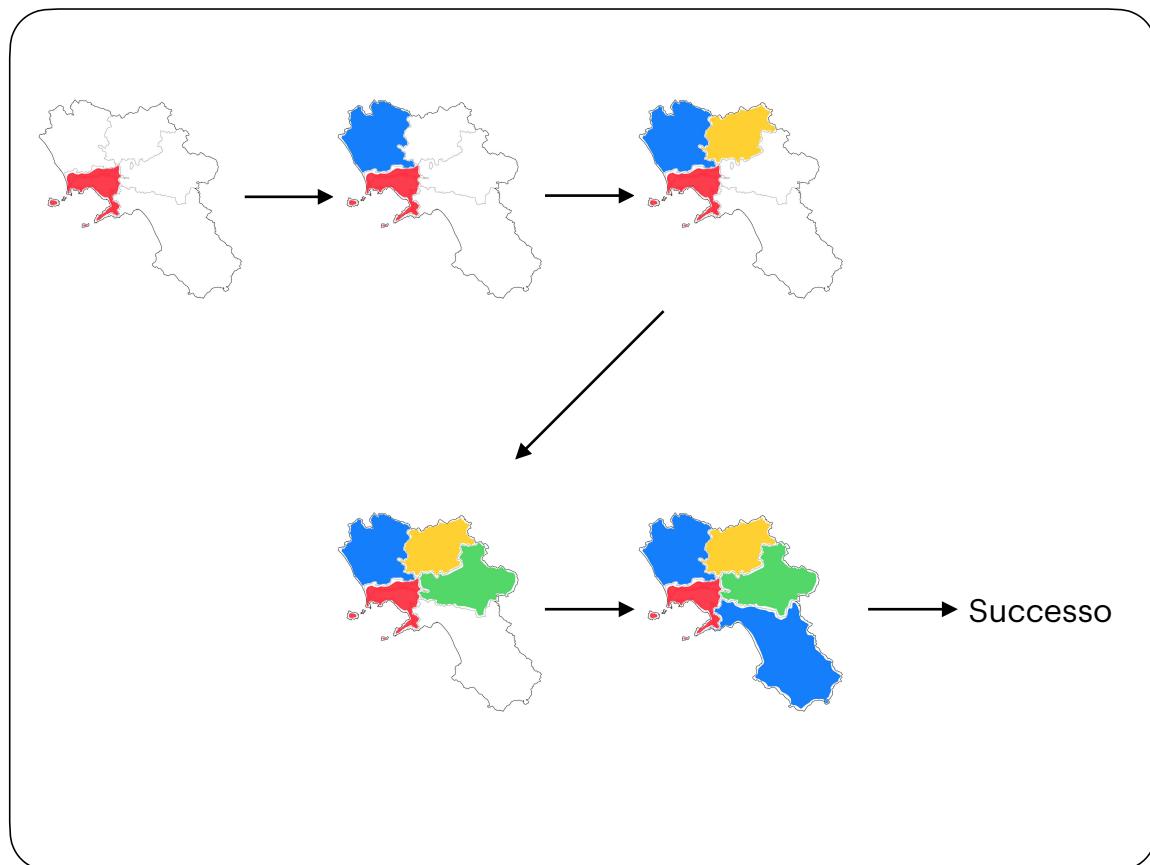


Figura 2.2.2.1

Da qui si evince che per colorare la mappa della regione Campania bastano solo tre colori.

Esercizio 3: Algoritmo Genetico e Map Coloring

Per utilizzare l'algoritmo genetico, dai vincoli generiamo il grafo (Figura 2.3.1) che verrà utilizzato dall'algoritmo:

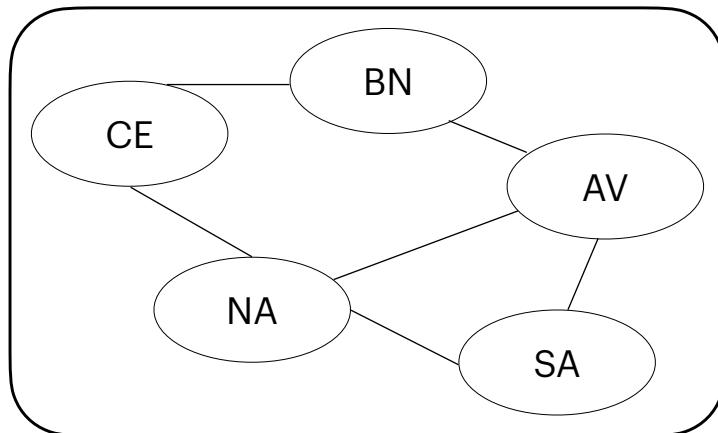


Figura 2.3.1

Il codice completo si trova al seguente link: <https://colab.research.google.com/drive/1Hr8U3TDnUSRlils-7UtVjhKq6zSpypxZ?usp=sharing>

L'**algoritmo genetico** è fatto in questo modo:

```
define: popolazione, genitori, figli
Popolazione = generazione casuale di valori;
while (!termiato) {
    gaRun();
}
function gaRun() {
    parents = getParents();
    child = crossover(parents);
    child = mutate(child);
    population.add(child);
}
```

In quest'algoritmo si genera prima una array di **popolazione**, ovvero un insieme di grafi colorati in modo casuale. Si esegue la funzione dell'algoritmo

in continuo finché non raggiunge un certo valore di **terminazione** che può essere ottenuto in due modi:

- Un numero massimo di volte in cui eseguire la funzione
- Quando avrà raggiunto uno stato in cui la mappa è colorata e tutti i vincoli sono rispettati.

La funzione che viene eseguita in *loop* è **gaRun()** che esegue tre subroutine:

1. **getParents()**: restituisce due elementi dell'array di Popolazione, che, a seconda di quanto si è vicini alla soluzione, può essere eseguita in due modi diversi:

1. Se siamo lontani dalla soluzione, prende 4 valori casuali dalla popolazione e ne sceglie i 2 migliori,
2. Se siamo vicini alla soluzione, prende 2 valori dalla popolazione che sono i migliori.

2. **crossover(parents)**: crea un nuovo elemento di popolazione partendo da due genitori; sceglie un indice chiamo *Crosspoint* dell'array di un genitori e costruisce un nuovo figlio prendendo tutti i valori da 0 a *Crosspoint* dal primo genitore ed il restante dal secondo genitore

3. **mutate(child)**: cambia leggermente il nuovo figlio in modo da portarlo quanto più vicine alla soluzione e, a seconda di quanto siamo vicini alla soluzione, si divide in due tipi:

1. Per ogni vertice (*Provincia*) in *popolazione*; se il vertice ha colori adiacenti uguali, allora, rimuove dai colori validi quelli in comune; sceglie un nuovo colore, in modo casuale tra quelli validi e modifica il figlio con il nuovo colore per il vertice selezionato,
 2. Per ogni vertice in *popolazione*; se il vertice ha colori adiacenti uguali, allora, sceglie un nuovo colore, in modo casuale tra tutti i colori e modifica il figlio con il colore selezionato.
4. Aggiunge il figlio alla popolazione

Esercizio 4: Logica preposizionale

La logica preposizionale si basa sull'algebra booleana; la sintassi prevede elementi base detti **proposizioni**, che sono delle proprietà che posso essere o vere o *false*. Le proposizioni possono essere combinate tramite gli **operatori**: *and*, *not*, *or*, *implicazione* ed *equivalenza*.

Si introduce il concetto di **conseguenza logica** con il quale si intende che, una determinata proposizione β ne segue logicamente un'altra α (si indica con la seguente notazione $\alpha \models \beta$) espressa in questo modo:

$\alpha \models \beta \Leftrightarrow \beta = \text{True}$ if $\alpha = \text{True}$, ovvero α determina β se e solo se, nel caso in cui α è vera, allora lo è anche β .

La conseguenza logica, quindi, può essere usata per ottenere delle **inferenze logiche**, ovvero delle conclusioni. La differenza tra conseguenza logica e inferenza logica è che: la conseguenza rappresenta il soggetto da trovare (la soluzione), mentre l'inferenza è il processo attraverso il quale trovo la soluzione. Il formalismo matematico che indico ciò è : $KB \models i(\alpha)$, ovvero α è derivato da KB attraverso i.

Un algoritmo per trovare l'inferenza è il **model checking**, che è un implementazione diretta della conseguenza logica, ovvero si deve enumerare esplicitamente tutti i modelli e verificare che α si vera in ogni modello in cui KB è vera.

Per dimostrare ciò usiamo un esempio ridotto del *Wumpus World*, dove consideriamo la presenza di sole due proposizioni :

- $B_{[i,j]}$: rappresenta la presenza di brezza nella casella $[i,j]$
- $P_{[i,j]}$: rappresenta la presenza di un baratro nella casella $[i,j]$

E abbiamo già esplorato le caselle (1,1) - (1,2) - (2,1), come mostrato nella Figura 2.4.1.

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK			
1,1	2,1 B	3,1	4,1
V	OK		

Figura 2.4.1

Si vuole determinare la presenza di un baratro nella cella (2,2); per fare ciò si costruisce la tabella che dovrebbe avere 2^7 (128), possibili modelli

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	true	true	true	true	false	false						
false	false	false	false	false	false	true	true	true	false	true	false	false
:	:	:	:	:	:	:	:	:	:	:	:	:
false	true	false	false	false	false	false	true	true	false	true	true	false
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>
:	:	:	:	:	:	:	:	:	:	:	:	:
true	false	true	true	false	true	false						

Figura 2.4.2

come nella figura 4.2.

Vediamo che $P_{[2,2]}$ è vera in due modelli su tre, quindi non possiamo dire se c'è un pozzo o meno.

Una **Regola di inferenza** è rappresentata dalla risoluzione, che, unita ad un algoritmo di ricerca completo, genera un algoritmo di inferenza completo. La regola di inferenza si applica solo alle clausole, ovvero alle disgiunzioni di letterali, quindi dovrebbe essere applicabile a KB solo se composta da

clausole. Però, dato che ogni formula nella logica preposizionale è equivalente ad una congiunzione di clausole, la regola di inferenza può essere applicata sull'intera logica preposizionale. Questo poiché ogni formula può essere convertita in **CNF** (forma normale congiuntiva) seguendo la seguente procedura:

1. Ho la formula $R_2 : B_{[1,1]} \Leftrightarrow (P_{[1,2]} \vee P_{[2,1]})$

2. Si elimina \Leftrightarrow , sostituendola con la regola

$\alpha \Leftrightarrow \beta \equiv (\alpha \Rightarrow \beta) \wedge (\alpha \Leftarrow \beta)$, quindi la nostra formula diventa

$$(B_{[1,1]} \Rightarrow (P_{[1,2]} \vee P_{[2,1]})) \wedge (B_{[1,1]} \Leftarrow (P_{[1,2]} \vee P_{[2,1]}))$$

3. Si elimina \Leftarrow e \Rightarrow , con la regola $\alpha \Rightarrow \beta \equiv \neg\alpha \vee \beta$, quindi la formula diventa $(\neg B_{[1,1]} \vee P_{[1,2]} \vee P_{[2,1]}) \wedge (\neg(P_{[1,2]} \vee P_{[2,1]}) \vee B_{[1,1]})$

4. Il \neg deve essere applicato solo ai letterari quindi applicando le regole:

1. $\neg(\neg\alpha) \equiv \alpha$ (eliminazione doppia negazione)

2. $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ (De Morgan)

3. $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$ (De Morgan)

Applicando la 3 si ottiene

$$(\neg B_{[1,1]} \vee P_{[1,2]} \vee P_{[2,1]}) \wedge ((\neg P_{[1,2]} \wedge \neg P_{[2,1]}) \vee B_{[1,1]})$$

La formula in CNF quindi è:

$$(\neg B_{[1,1]} \vee P_{[1,2]} \vee P_{[2,1]}) \wedge (\neg P_{[1,2]} \vee B_{[1,1]}) \wedge (\neg P_{[2,1]} \vee B_{[1,1]}).$$

Le procedure di inferenze sfruttano, per la risoluzione, il principio dell'assurdo. Un algoritmo di risoluzione si comporta in questo modo:

1. Convertire $(KB \wedge \neg\alpha)$ in CNF

2. Applicare la regola di risoluzione alle clausole, ogni coppia che contiene letterali complementari viene risolta e poi aggiunta alle altre

3. Il processo continua finché:

1. Non posso aggiungere nuove clausole, allora α NON è conseguenza logica di KB
2. Oppure, la risoluzione di due clausole genera una clausola vuota, allora α è conseguenza logica di KB

Un algoritmo di risoluzione è il seguente:

```

function PL-RESOLUTION(KB ,  $\alpha$ ) returns true or false
    inputs: KB , the knowledge base, a sentence in
    propositional logic  $\alpha$ , the query, a sentence in
    propositional logic

    clauses  $\leftarrow$  the set of clauses in the CNF
        representation of KB  $\wedge \neg \alpha$ 

    new  $\leftarrow \{ \}$ 
    loop do
        for each pair of clauses  $C_i, C_j$  in clauses do
            resolvents  $\leftarrow$  PL-RESOLVE( $C_i, C_j$ )
            if resolvents contains the empty clause
                then return true
            new  $\leftarrow$  new  $\cup$  resolvents
            if new  $\subseteq$  clauses then return false
            clauses  $\leftarrow$  clauses  $\cup$  new

```

Esercitazione 3: Reti Bayesiane e Reti Neurali

Esercizio 1: Rete Bayesiana

Punto 1: Concetti Principali

Una **rete Bayesiana** è un grafo orientato che viene usato per le asserzioni di indipendenza condizionata. Essa è composta da:

- **Nodi:** sono costituiti da variabili aleatorie, sia discrete che continue,
- **Archi:** collegano i nodi in modo orientato e non ciclico (DAG), ovvero un arco può andare da X a Y ma non viceversa; X è detto genitore di Y,
- Ogni nodo ha una sua **distribuzione di probabilità** che qualifica gli effetti che il suo genitore ha su di esso.

La **topologia** della rete specifica le condizioni di indipendenza. La combinazione tra topologia e delle distribuzioni è sufficiente a definire la distribuzione completa di tutte le variabili.

La rete Bayesiana fornisce, quindi, una descrizione completa del domino, ovvero, ogni elemento della distribuzione di probabilità può essere calcolato a partire dell'informazione contenuta nella rete

- La **semantica** della rete Bayesiana può essere interpretata in due modi:
- La rete è una rappresentazione di una distribuzione congiunta di probabilità, che si rivela utile per capire come costruire le reti,
 - La rete è una codifica di una collezione di asserzioni di indipendenza condizionale, che aiuta a progettare le procedure di inferenza.

Da qui si ottiene la **semantica topologica** che è composta dalle due specifiche seguenti, equivalenti tra di loro:

- Un nodo è dipendente solo dai suoi genitori e non da altri predecessori
- Sul nodo vale la **copertura di Markov**, ovvero un nodo è condizionalmente indipendente da tutti i nodi della rete, dati i suoi genitori, i suoi figli e i genitori dei figli

Per calcolare l'**Inferenza** usiamo la seguente notazione:

- X: indica la variabile di query (ovvero della probabilità dell'evento che si sta osservando),

- E: l'insieme di variabili di evento, dove e indica l'evento che si sta osservando

- Y: variabili nascoste

Ogni probabilità può essere calcolata moltiplicando i termini relativi della distribuzione congiunta. Ovvero per risolvere una query $P(X | e)$ si applica la seguente equazione: $P(X | e) = \prod_{i=1}^n P(x_i | \text{genitori}(X_i))$; ciò mostra che i termini $P(x, e, y)$ si possono scrivere come prodotti di probabilità condizionate.

Punto 2: Esempio di Rete Bayesiana

Una esempio di rete Bayesiana è contenuta nella Figura 3.2.1

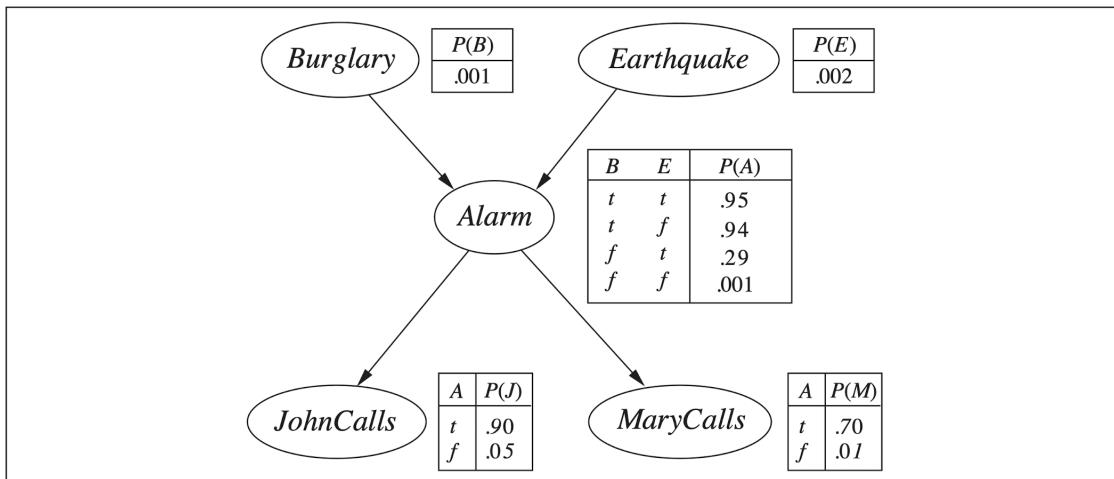


Figura 3.2.2.1

Essa rappresenta la probabilità che un allarme suoni e i vicini chiamino il proprietario di casa. L'allarme suona in caso di intrusione da parte di un ladro oppure in caso di terremoto. Se l'allarme suona i vicini, John e Mary, chiamano il proprietario con le probabilità indicate.

Dalla **Copertura di Markov** si evince che il nodo *Burglary* dipende dal figlio *Alarm* e dal genitore del figlio *Earthquake*. Mentre *JohnCalls* è indipendente da *Burglary* e *Earthquake*, ma dipende da *Alarm*.

Se scegliamo di osservare l'evento in cui l'allarme suona e non vi sono né un'intrusione né un terremoto, la cui query è

$P(Alarm | Earthquake = False, Burglary = False)$, o più sinteticamente $P(A | b, e)$. Applichiamo la regola dell'inferenza definita nel punto precedente; si ha che le variabili nascoste sono $j = JohnCalls$ e $m = MaryCalls$, per la copertura di Markov e si ottiene $P(A | b, e) = P(j, m, a, \neg b, \neg e)$

Applichiamo le proprietà di distribuzione e otteniamo:

$$P(A | b, e) = P(j | a)P(m | a)P(a | \neg b, \neg e)P(\neg b)P(\neg e)$$

Sostituendo i valori numerici si ottiene:

$$P(A | b, e) = 0.9 \times 0.7 \times 0.999 \times 0.001 \times (1 - 0.001) \times (1 - 0.002) \approx 0.00063$$

Esercizio 2: Reti Neurali

Punto 1: Concetti principali

Le **Reti Neurali** sono chiamate così poiché ispirate al cervello umano.

Esse sono composte da unità chiamate **neuroni** e da **link** che le collegano; ad ogni link è associato un peso numerico $W_{[i,j]}$, che determina la forza e il segno della connessione.

Un neurone, mostrato in Figura 3.2.1.1, è composto da:

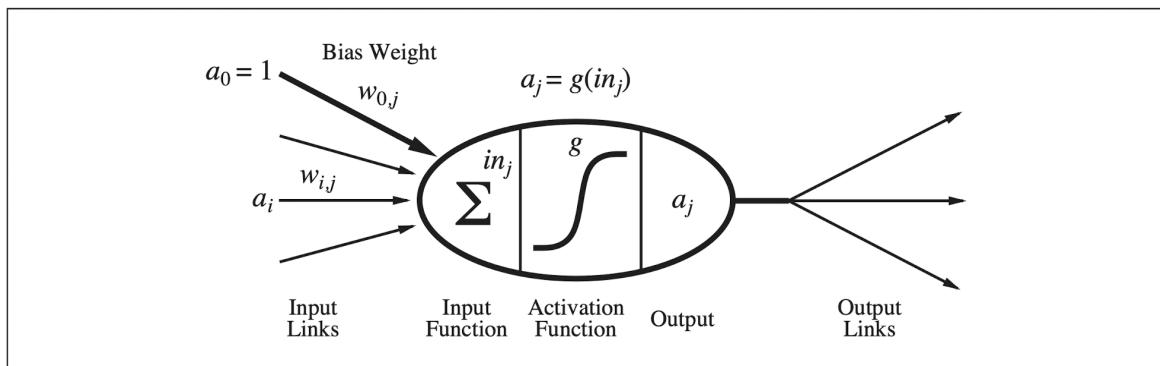


Figura 3.2.1.1

- **Link di ingresso**, attraverso i quali il neurone ottiene degli input, e il neurone ne esegue un somma pesata: $in_i = \sum_{j=0}^n W_{[i,j]} a_j$,
- Gli **input** pesati, che vengono elaborati dalla la **funzione di attivazione** $g(x)$ per ottenere gli **output** : $a_i = g(in_i) = g(\sum_{j=0}^n W_{[i,j]} a_j)$.

La funzione di attivazione $g(x)$ deve soddisfare due requisiti:

1. Il neurone è attivo ($g(x) \rightarrow 1$, $g(x)$ tende a 1) quando sono dati in input i valori “giusti”; mente è inattivo ($g(x) \rightarrow 0$) quando gli input sono “sbagliati”
2. L’attivazione deve essere non lineare, altrimenti la rete collasserebbe nel caso si usasse una funzione non lineare per gli ingressi

Due possibili funzioni di attivazione sono:

- Il gradino (Figura 3.2.1.2 (a)), in qual caso il neurone è chiamato **perceptron**,
- Il sigmoide (Figura 3.2.1.2 (b)), in qual caso il neurone è chiamato **sigmoid perceptron**, la funzione sigmoide aggiunge la caratteristica di essere differenziabile.

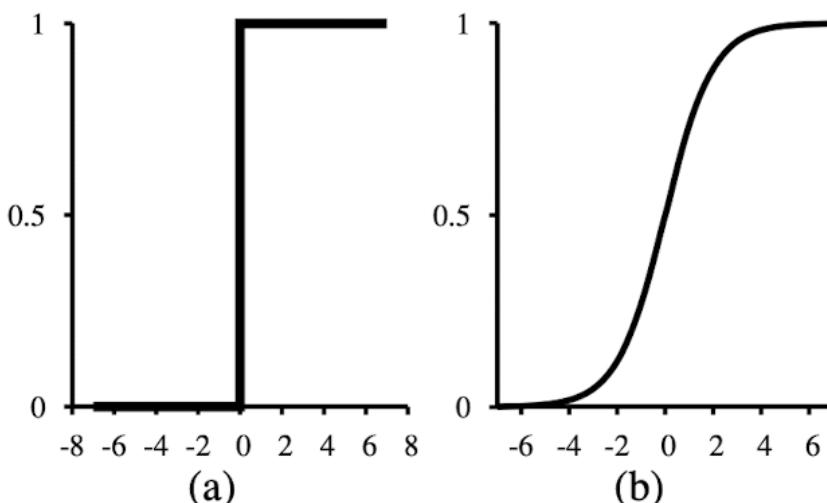


Figura 3.2.1.2

La reti neurali sono divise in due categorie principali, in base al tipo di connessione presente tra i neuroni, e sono:

- **Acicliche** (o feed-forward): in cui i neuroni sono collegati in un'unica direzione, ovvero ogni nodo riceve gli input dai nodi precedenti e invia gli output a nodi successivi,
- **Ricorrenti**: in cui l'output di un neurone può essere anche l'input di se stesso. In questo senso le reti possiedono una sorta memoria breve termine, rendendole più simili al nostro cervello.

Le prime reti neurali erano composte da un unico livello (layer), con un solo neurone; ciò aveva limiti di calcolo elevati, infatti esse non potevano eseguire funzioni come la XOR. Aggiungendo però più livelli, si incrementò lo spazio delle ipotesi rappresentabili, Figura 3.2.1.3.

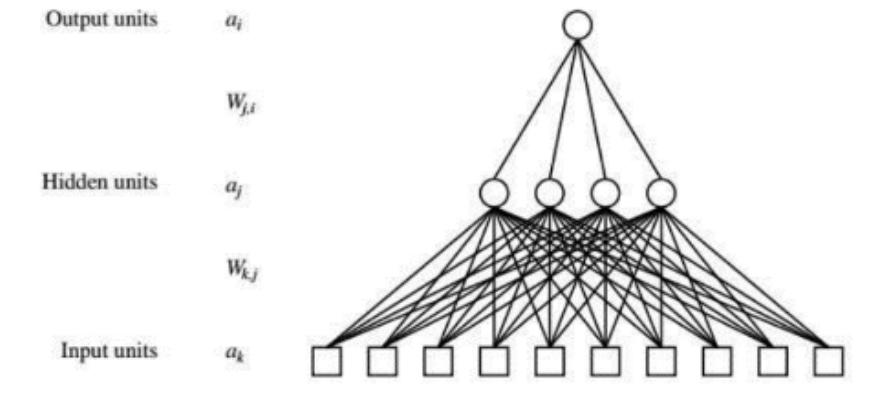


Figura 3.2.1.3

Gli algoritmi di apprendimento per le reti multi-layer sono simili a quelli per i percettori, con la differenza che le reti possono avere più uscite, racchiuse in un vettore $h_w(x)$ ed ogni esempio, usato per addestrare la rete, ha un vettore di output y .

La differenza principale è che, definito l'errore $e = y - h_w$, esso risulta visibile nel layer più esterno, ovvero quello di output; mentre risulta essere difficile da capire per i layer più interni (o nascosti).

Per ridurre questo errore una strategia applicabile è quella del **backward propagation**; quest'algoritmo retropropaga l'errore di livello output ai livelli nascosti.

A livello di output, la regola di aggiornamento dei pesi (usata per minimizzare l'errore) è: $w_{[j,k]} \leftarrow w_{j,k} + (\alpha \times a_j \times \Delta_k)$.

Per aggiornare le connessioni tra le unità di input e i layer nascosti, occorre definire una quantità analoga all'errore definito precedentemente, grazie alla retropropagazione. Essa si basa sull'idea che il nodo j è responsabile per una parte di errore Δ_i in ogni nodo di output a cui è collegato. La regola della retropropagazione ha la seguente struttura

$$\Delta_j = g'(in_j) \sum_i W_{[i,j]} \Delta_i , \text{ ottenendo così un regola di aggiornamento dei pesi}$$

simile a quello per l'output: $w_{[j,k]} \leftarrow w_{j,k} + (\alpha \times a_k \times \Delta_j)$.

L'algoritmo è il seguente:

```
function BACK-PROP-LEARNING(examples, network) return a neural network
  inputs: examples , a set of examples, each with input vector x and
output vector y; network , a multilayer network with L layers, weights
w[i,j], activation function g
  local variables: Δ, a vector of errors, indexed by network node
repeat
  for each weight w[i,j] in network do
    w[i,j] ← a small random number
  for each example (x, y) in examples do
    /* Propagate the inputs forward to compute the outputs */
    for each node i in the input layer do
      ai ← xi
    for l=2 to L do
      for each node j in layer l do
        inj ← ∑i w[i,j] ai
        aj ← g(inj)
    /* Propagate deltas backward from output layer to input layer */
    for each node j in the output layer do
      Δ[j] ← g'(inj) × (yj − aj)
    for l = L − 1 to 1 do
      for each node i in layer do
        Δ[i] ← g'(inj) ∑j w[i,j] Δ[j]
    /* Update every weight in network using deltas */
    for each weight w[i,j] in network do
      w[i,j] ← w[i,j] + (α × ai × Δ[j])
until some stopping criterion is satisfied
return network
```

Ecco un esempio di come si comporta una rete con un solo strato nascosto, alle prese con il problema della coda al ristorante (Esercitazione 1 - Esercizio 5)

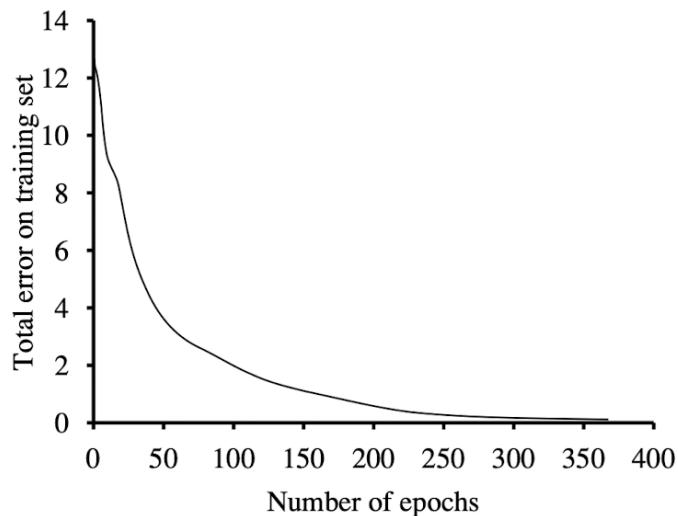


Figura 3.2.1.4

In Figura 3.2.4 si mostra come la rete converga fino a ridurre a zero l'errore, man mano che l'algoritmo viene addestrato.

Punto 2: Esempio di Rete neurale

Per evidenziare la presenza di alberi all'interno di un'immagine, si può procedere in questo modo:

1. Si estraggono i bordi dell'immagine con un algoritmo di Canny per semplificare la classificazione
2. Si crea una rete neurale multi-layer in cui ad ogni percettore è dato un pixel dell'immagine e la sua soglia di attivazione
3. Ogni neurone avrà un peso numerico che influenza il risultato
4. Si addestra la rete con un set di immagini ed applico la **error back propagation** per aggiornare i pesi
5. Si può applicare tale modello su un altro set di immagini per testarne l'accuratezza

Esercizio opzionale A: Riconoscimento Numeri

Per creare una rete neurale che riconosca i numeri scritti a mano utilizziamo il Dataset **MNIST** che contiene circa 70000 immagini di numeri scritti a mano già etichettati. Per il codice completo usare il seguente link:
<https://colab.research.google.com/drive/1RgW5DCERpLq-a2iQU7PUyjQVfDD4e8Xz>

Per svolgere l'esercizio si fa uso dell'ambiente "Google Colaboratory".

1. Si importa il Dataset all'interno dell'ambiente con le seguenti linee di codice:

```
import mxnet as mx  
mnist = mx.test_utils.get_mnist()
```

2. Fatto questo si divide il Dataset in trainingSet e testSet con le seguenti linee di codice:

```
batch_size = 100  
train_iter = mx.io.NDArrayIter(mnist['train_data'],  
mnist['train_label'], batch_size, shuffle=True)  
val_iter = mx.io.NDArrayIter(mnist['test_data'],  
mnist['test_label'], batch_size)
```

3. Si convertono le immagini in 2D:

```
data = mx.sym.var('data')  
# Flatten the data from 4-D shape into 2-D (batch_size,  
num_channel*width*height)  
data = mx.sym.flatten(data=data)
```

4. Si creano i primi Hidden Layer

```
# The first fully-connected layer and the corresponding  
activation function  
fc1 = mx.sym.FullyConnected(data=data, num_hidden=128)  
act1 = mx.sym.Activation(data=fc1, act_type="relu")
```

```

# The second fully-connected layer and the corresponding
activation function

fc2 = mx.sym.FullyConnected(data=act1, num_hidden = 64)
act2 = mx.sym.Activation(data=fc2, act_type="relu")

```

5. Si crea l'output layer:

```

# MNIST has 10 classes

fc3 = mx.sym.FullyConnected(data=act2, num_hidden=10)
# Softmax with cross entropy loss

mlp = mx.sym.SoftmaxOutput(data=fc3, name='softmax')

```

6. Si addestra la rete con il seguente codice:

```

import logging

logging.getLogger().setLevel(logging.DEBUG) # logging to stdout

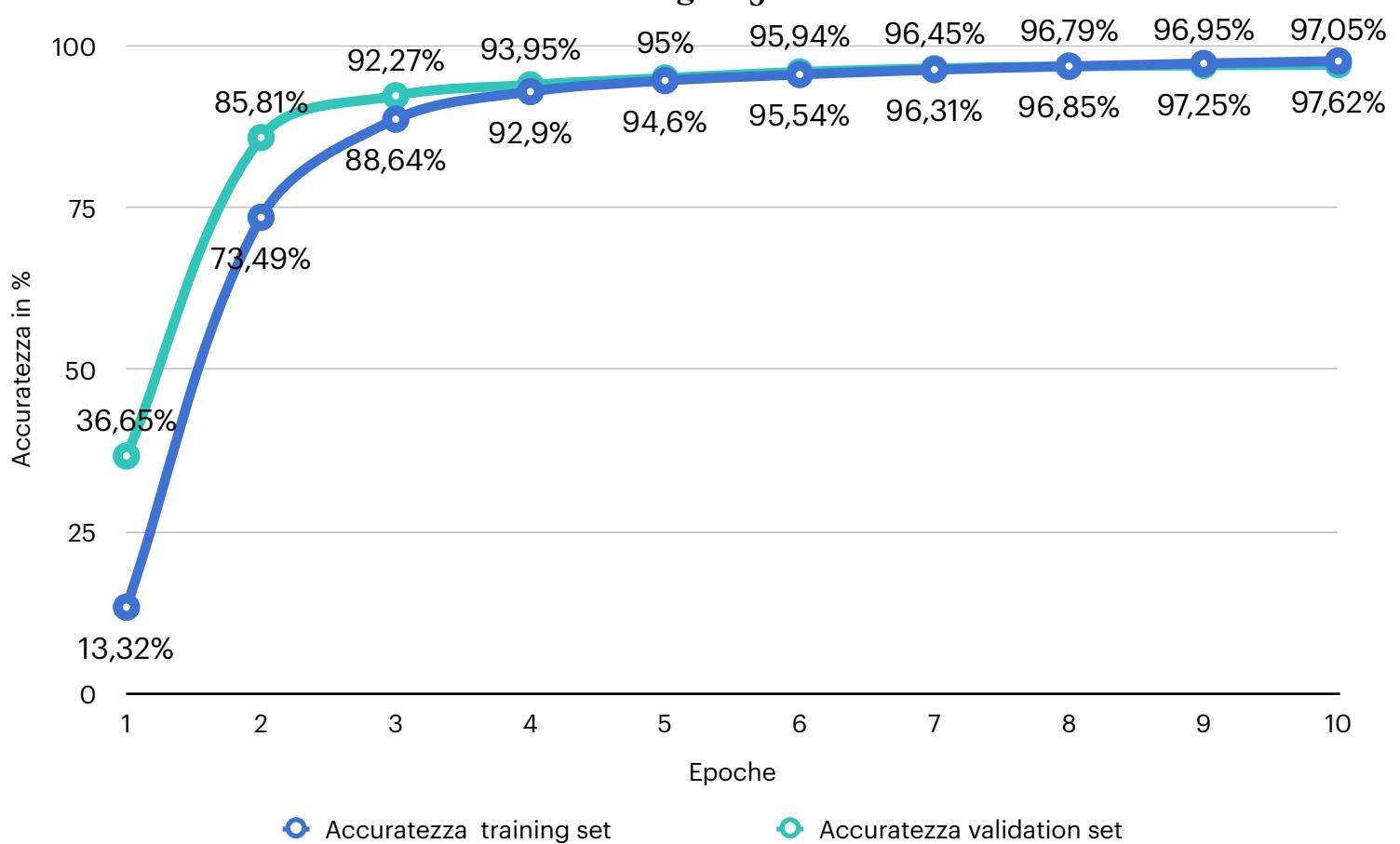
mlp_model = mx.mod.Module(symbol=mlp, context=ctx)

mlp_model.fit(train_iter, # train data
              eval_data=val_iter, # validation data
              optimizer='sgd', # use SGD to train
              optimizer_params={'learning_rate':0.1},
              eval_metric='acc', # report accuracy during training
              batch_end_callback =
mx.callback.Speedometer(batch_size, 100),
              num_epoch=10) # train for at most 10 dataset passes

```

Da questo addestramento si ottiene in grafico in Figura 3.A.1, che ci mostra come, al susseguirsi delle epoche, varia l'accuratezza.

Figura 3.A.1



Per calcolare l'accuratezza del testSet si usa il seguente codice

```
test_iter = mx.io.NDArrayIter(mnist['test_data'],
mnist['test_label'], batch_size)
# predict accuracy of mlp
acc = mx.metric.Accuracy()
mlp_model.score(test_iter, acc)
print(acc)
assert acc.get()[1] > 0.96, "Achieved accuracy (%f) is
lower than expected (0.96)" % acc.get()[1]
```

L'accuratezza del testSet, si approssima al 97,05%