UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

INTELLIGENZA ARTIFICIALE

ELABORATO DI:

GIULIANO DI GIUSEPPE N46004374

IN COLLABORAZIONE CON:

SOSSIO CIRILLO

MATTEO BARTIROMO

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

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.

Eseguendo il codice riportato nel testo 'Modeling with Decision Trees' sull'ambiente di sviluppo Visual Studio Code, otteniamo i seguenti risultati:

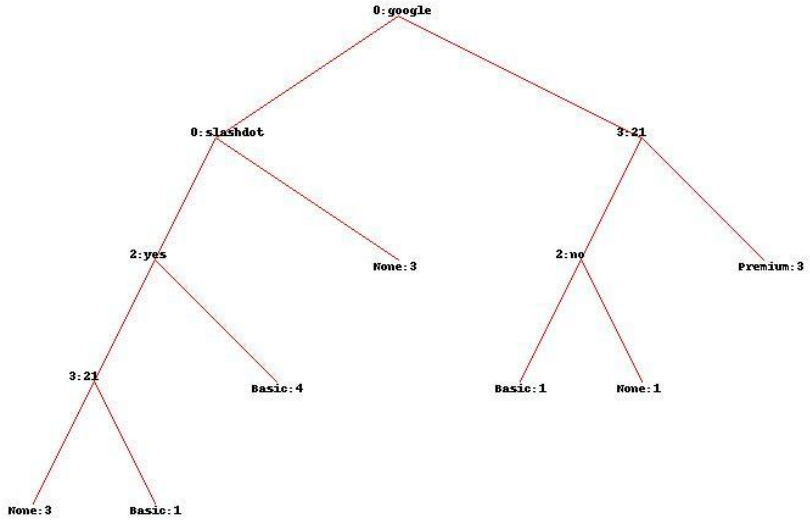
```
0:google?
T-> 3:21?
  T-> {'Premium': 3}
  F-> 2:yes?
    T-> {'Basic': 1}
    F-> {'None': 1}
F-> 0:slashdot?
  T-> {'None': 3}
  F-> 2:yes?
    T-> {'Basic': 4}
    F-> 3:21?
      T-> {'Basic': 1}
      F-> {'None': 3}
```

Questo rappresenta l'albero di decisione costruito sulla base della seguente tabella:

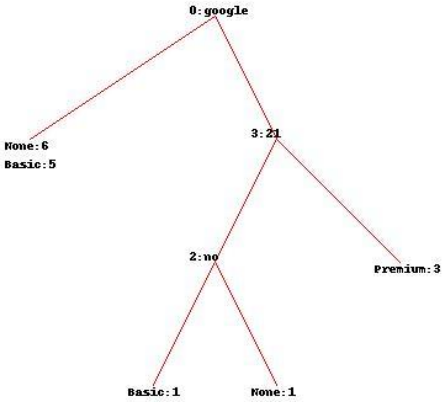
Referrer	Location	Read FAQ	Pages viewed	Service chosen
Slashdot	USA	Yes	18	None
Google	France	Yes	23	Premium
Digg	USA	Yes	24	Basic
Kiwitobes	France	Yes	23	Basic
Google	UK	No	21	Premium
(direct)	New Zealand	No	12	None
(direct)	UK	No	21	Basic
Google	USA	No	24	Premium
Slashdot	France	Yes	19	None
Digg	USA	No	18	None
Google	UK	No	18	None
Kiwitobes	UK	No	19	None
Digg	New Zealand	Yes	12	Basic
Google	UK	Yes	18	Basic
Kiwitobes	France	Yes	19	Basic

Essa rappresenta un servizio online che fornisce un periodo di prova gratuito e che vuole predire, sulla base di alcune informazioni quali la localizzazione geografica, le pagine visitate dall'utente precedentemente, se vengano lette o meno le FAQ ecc., se l'utente sottoscriverà, al termine del periodo, un account Basic,

Premium o None (nessuno dei due). Il numero che precede la domanda corrisponde alla colonna relativa all'attributo mentre il numero accanto al valore del servizio scelto, che si trova negli endpoint, indica la "chance" che l'utente scelga quel tipo di servizio.
L'istruzione treepredict.drawtree(tree,jpeg='treeview.jpg') una volta eseguita consente di creare un file .jpg che consente di rappresentare graficamente l'albero di decisione. Esso risulta essere strutturato in questo modo:



Il codice utilizzato fornisce anche le funzioni per la classificazione degli attributi in base ad un criterio (la "buildtree" utilizza l'Entropia Media Ponderata) ed eventualmente funzioni per eseguire un'operazione di pruning dell'albero. Quando la funzione prune(tree,mingain) viene chiamata sul nodo radice, crea un elenco combinato di risultati da entrambe le foglie e valuta l'entropia. Se la variazione di entropia è inferiore al parametro minigain, le foglie verranno eliminate e tutti i loro risultati verranno spostati al loro nodo genitore. Il nodo combinato diventa un possibile candidato per l'eliminazione e fusione con un altro nodo. In particolare, eseguendo il codice ed applicando la funzione di pruning otteniamo un albero di queto tipo:



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.

Utilizzando il dataset Mushrooms è possibile determinare , sulla base degli esempi scelti per il learning set, se un fungo sia commestibile (edible) o velenoso (poisonus). Gli attributi contenuti nel learning set sono i seguenti:

Attribute Information:

1. cap-shape: bell=b,conical=c,convex=x,flat=f, knobbed=k,sunken=s
2. cap-surface: fibrous=f,grooves=g,scaly=y,smooth=s
3. cap-color: brown=n,buff=b,cinnamon=c,gray=g,green=r, pink=p,purple=u,red=e,white=w,yellow=y
4. bruises?: bruises=t,no=f
5. odor: almond=a,anise=l,creosote=c,fishy=y,foul=f, musty=m,none=n,pungent=p,spicy=s
6. gill-attachment: attached=a,descending=d,free=f,notched=n
7. gill-spacing: close=c,crowded=w,distant=d
8. gill-size: broad=b,narrow=n
9. gill-color: black=k,brown=n,buff=b,chocolate=h,gray=g, green=r,orange=o,pink=p,purple=u,red=e, white=w,yellow=y
10. stalk-shape: enlarging=e,tapering=t
11. stalk-root: bulbous=b,club=c,cup=u,equal=e, rhizomorphs=z,rooted=r,missing=?
12. stalk-surface-above-ring: fibrous=f,scaly=y,silky=k,smooth=s
13. stalk-surface-below-ring: fibrous=f,scaly=y,silky=k,smooth=s
14. stalk-color-above-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p,red=e,white=w,yellow=y
15. stalk-color-below-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p,red=e,white=w,yellow=y
16. veil-type: partial=p,universal=u
17. veil-color: brown=n,orange=o,white=w,yellow=y
18. ring-number: none=n,one=o,two=t
19. ring-type: cobwebby=c,evanescent=e,flaring=f,large=l, none=n,pendant=p,sheathing=s,zone=z
20. spore-print-color: black=k,brown=n,buff=b,chocolate=h,green=r, orange=o,purple=u,white=w,yellow=y
21. population: abundant=a,clustered=c,numerous=n, scattered=s,several=v,solitary=y
22. habitat: grasses=g,leaves=l,meadows=m,paths=p, urban=u,waste=w,woods=d

Il dataset nel formato .csv (**mushrooms_csv.csv**) risulta il seguente:

	A	B	C	D	E	F	G	H
1	x,s,n,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,k,s,u,p							
2	x,s,y,t,a,f,c,b,k,e,c,s,s,w,w,p,w,o,p,n,n,g,e							
3	b,s,w,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,n,m,e							
4	x,y,w,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,k,s,u,p							
5	x,s,g,f,n,f,w,b,k,t,e,s,s,w,w,p,w,o,e,n,a,g,e							
6	x,y,y,t,a,f,c,b,n,e,c,s,s,w,w,p,w,o,p,k,n,g,e							
7	b,s,w,t,a,f,c,b,g,e,c,s,s,w,w,p,w,o,p,k,n,m,e							
8	b,y,w,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,s,m,e							
9	x,y,w,t,p,f,c,n,p,e,e,s,s,w,w,p,w,o,p,k,v,g,p							
10	b,s,y,t,a,f,c,b,g,e,c,s,s,w,w,p,w,o,p,k,s,m,e							
11	x,y,y,t,l,f,c,b,g,e,c,s,s,w,w,p,w,o,p,n,n,g,e							
12	x,y,y,t,a,f,c,b,n,e,c,s,s,w,w,p,w,o,p,k,s,m,e							
13	b,s,y,t,a,f,c,b,w,e,c,s,s,w,w,p,w,o,p,n,s,g,e							
14	x,y,w,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,n,v,u,p							
15	x,f,n,f,n,f,w,b,n,t,e,s,f,w,w,p,w,o,e,k,a,g,e							
16	s,f,g,f,n,f,c,n,k,e,e,s,s,w,w,p,w,o,p,n,y,u,e							
17	f,f,w,f,n,f,w,b,k,t,e,s,s,w,w,p,w,o,e,n,a,g,e							
18	x,s,n,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,k,s,g,p							
19	x,y,w,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,n,s,u,p							
20	x,s,n,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,n,s,u,p							
21	b,s,y,t,a,f,c,b,k,e,c,s,s,w,w,p,w,o,p,n,s,m,e							
22	x,y,n,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,n,v,g,p							
23	b,y,y,t,l,f,c,b,k,e,c,s,s,w,w,p,w,o,p,n,s,m,e							
24	b,y,w,t,a,f,c,b,w,e,c,s,s,w,w,p,w,o,p,n,n,m,e							
25	b,s,w,t,l,f,c,b,g,e,c,s,s,w,w,p,w,o,p,k,s,m,e							
26	f,s,w,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,n,v,g,p							
27	x,y,y,t,a,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,n,m,e							
28	x,y,w,t,l,f,c,b,w,e,c,s,s,w,w,p,w,o,p,n,n,m,e							
29	f,f,n,f,n,f,c,n,k,e,e,s,s,w,w,p,w,o,p,k,y,u,e							
30	x,s,y,t,a,f,w,n,n,t,b,s,s,w,w,p,w,o,p,n,v,d,e							
31	b,s,y,t,l,f,c,b,g,e,c,s,s,w,w,p,w,o,p,n,n,m,e							
32	x,y,w,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,n,s,u,p							
33	x,y,y,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,n,m,e							
34	x,y,n,t,l,f,c,b,p,e,r,s,y,w,w,p,w,o,p,n,y,p,e							
35	b,y,y,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,s,m,e							

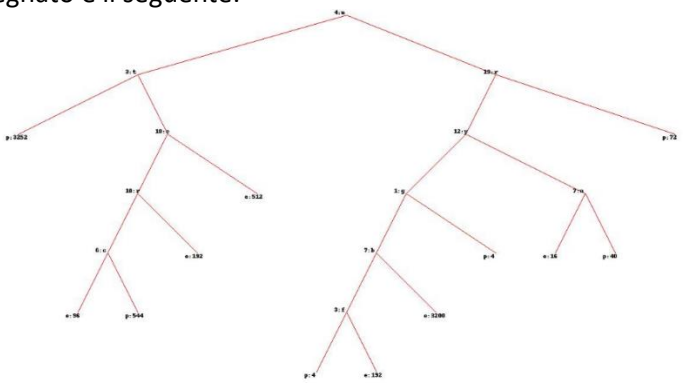
Nel file **treepredict.py** viene aggiunto il seguente frammento di codice per leggere il dataset e memorizzarlo in una lista chiamata “my_data”, che rappresenterà il learning set:

```
import csv
with open('mushroom_csv.csv', 'r') as file:
    reader = csv.reader(file)
    count = 0
    my_data = []
    for row in reader:
      my_data.append(row)
      if count > 8123:
          break
      count = count + 1
      print(row)
```

Il numero 8123 è il numero di righe del dataset che si vuole che rappresentino il *learning set*.
Usiamo il seguente codice per costruire l’albero e disegnarlo utilizzando tutte le righe del dataset:

```
import treepredict as tp
def run():
    tree = tp.buildtree(tp.my_data)
    tp.printtree(tree)
    tp.drawtree(tree,jpeg='mush.jpg')
```

L'output, cioè l'albero disegnato è il seguente:



L'albero è stato salvato in un'immagine chiamata **mush.jpg** e rappresenta l'albero utilizzando tutto il dataset come *learning set*.

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

Ora il nostro scopo è distinguere, nel dataset, un insieme di variabili per il *learning set* e uno per il *training set*, con lo scopo poi di verificare con quale percentuale dei due set si ha il risultato migliore.
Inseriamo le seguenti variabili per determinare il numero di righe e la percentuale attribuite al dataset per il testing e al dataset per il learning.

```
total_set=tp.my_data.__len__()
learn_rows = 812*5
test_rows = 8123-learn_rows
learn_percent = (learn_rows)*100/(rows)
test_percent = (test_rows)*100/(rows)
error = tp.giniimpurity(tp.my_data[0:learn_rows])
```

Per creare il dataset necessario per il testing viene rimossa l'ultima colonna, cioè la colonna dei risultati, dal dataset completo.
Importiamo nel file mush_test.csv cioè il data set per il testing , successivamente classifichiamo attraverso la funzione mdclassify il possibile valore che puo assumere il singolo fungo

```
with open('mush_test.csv') as file:
    reader = csv.reader(file)
    print(reader)
    count = 0
    end = test_rows
    for row in reader:
        if count > 0:
            classificazione = tp.mdclassify(row,tree)
            print(classificazione)
        if count > end:
              break
        count = count + 1
```

Calcolo ERROR RATE attraverso la funzione giniimpurity e stampo a video l ERROR RATE e il SUCCESS RATE e la percentuale di righe del LEARN e TEST SET

```
error = tp.giniimpurity(tp.my_data[0:learn_rows])
print('ERROR RATE', error)
print('SUCCESS RATE',1-error)
print('% learn set : ',learn_percent,'%')
print('%  test set : ',test_percent,'%')
```

Successivamente attraverso il seguente codice stampo il grafico che mostra l'andamento del SUCCESS rate in funzione dei dati di apprendimento.
Nei seguenti screen viene mostrato l'output del codice precedente:
LEARNING SET ≃90% TESTING SET ≃10%

```
ERROR RATE 0.49861344246413436
SUCCESS RATE 0.5013865575358656
% LEARNING SET :  89.99261447562778 %
% TESTING SET :  9.99507631708518 %
```

LEARNING SET ≃80% TESTING SET ≃20%

```
ERROR RATE 0.49034721949127025
SUCCESS RATE 0.5096527805087298
% LEARNING SET :  79.9975381585426 %
% TESTING SET :  19.99015263417036 %
```

LEARNING SET ≃50% TESTING SET ≃50%

```
ERROR RATE 0.2966654468517599
SUCCESS RATE 0.7033345531482401
% LEARNING SET :  50.01230920728705 %
% TESTING SET :  49.9753815854259 %
```

LEARNING SET ≃70% TESTING SET ≃30%

```
ERROR RATE 0.1917635740523047
SUCCESS RATE 0.8082364259476953
% LEARNING SET :  30.022156573116693 %
% TESTING SET :  69.96553421959626 %
```

LEARNING SET ≃10% TESTING SET ≃90%

```
ERROR RATE 0.18877338251345555
SUCCESS RATE 0.8112266174865445
% LEARNING SET :  10.032003938946332 %
% TESTING SET :  89.95568685376662 %
```

Per semplificare le stampe possiamo usare questo set di comandi

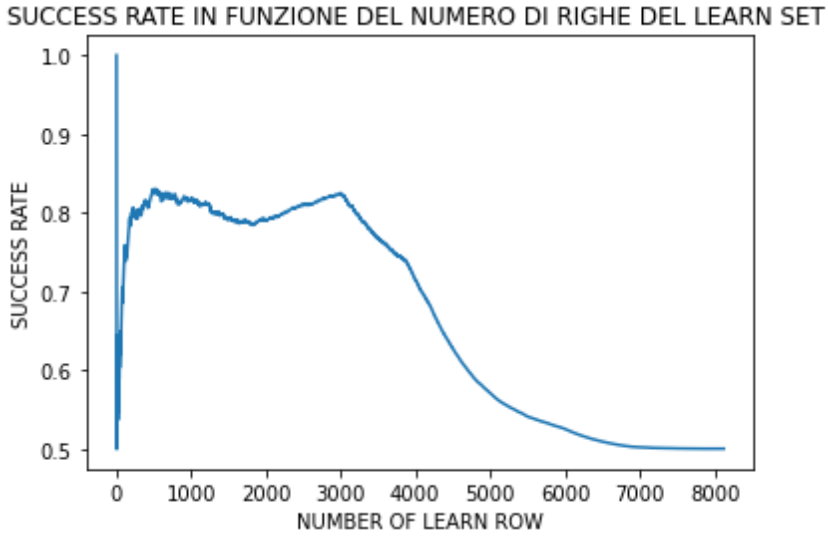
```
learn_rows = #numero di righe learning set
test_rows = 8123-learn_rows
learn_percent = (learn_rows)*100/(rows)
test_percent = (test_rows)*100/(rows)
error = tp.giniimpurity(tp.my_data[0:learn_rows])
print('ERROR RATE', error)
print('SUCCESS RATE',1-error)
print('% LEARNING SET : ',learn_percent,'%')
print('% TESTING SET : ',test_percent,'%')
```

Andiamo a controllare e a stampare a video il grafico che descrive il success rate in base al numero di righe del LEARN SET

```
from pylab import plot,show,title,xlabel,ylabel
def grafico ():
    with open('mushroom_csv.csv') as file:
        reader = csv.reader(file)
        count=0
        x=[]
        y=[]
        dati = []
        for row in reader:
            dati.append(row)
            error=tp.giniimpurity(dati)
            y.append(1-error)
            x.append(count)
            if count > 8123:
              break
            count = count + 1
        plot(x,y)
        title('SUCCESS RATE IN FUNZIONE DEL NUMERO DI RIGHE DEL LEARN SET ')
        xlabel('NUMBER OF LEARN ROW')
        ylabel('SUCCESS RATE')
        show()
```

E se mandiamo nella shell il seguente comando abbiamo la stampa a video del grafico

```
grafico()
```



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?

L'albero di decisione è un esempio di Machine Learning. Esso si basa sull'apprendimento induttivo che è una semplificazione dell'apprendimento umano. Questa forma di apprendimento è molto semplice:

- **ignora la conoscenza a priori.**
- **assume che l'ambiente sia deterministico e osservabile.**
- **assume che gli esempi siano dati.**

Di conseguenza un agente in grado di apprendere alberi di decisione è considerato intelligente perché è in grado di distinguere quale sia un buon o cattivo comportamento classificando correttamente gli esempi valutati sulla base di una conoscenza rappresentata dalla funzione obiettivo *f.* La funzione *f* è comunque una funzione target e non è raggiungibile, infatti lo scopo dell'agente è quello di trovare una funzione *h* che riesca ad approssimare quanto meglio possibile la funzione *f.* Dunque, un agente in grado di apprendere alberi di decisione dimostra di essere intelligente perché è in grado di fare una ricerca nello spazio delle ipotesi al fine di costruire un albero di decisione sulla base degli attributi a disposizione.

Un agente apprende se può fare delle riflessioni sulle azioni compiute e sullo stato del mondo, se può modificare il suo comportamento sulla base delle sue riflessioni ed esibire un comportamento che migliora nel corso del tempo.
L'apprendimento è la capacità di adattarsi ad un ambiente sconosciuto, esso modifica le decisioni dell'agente per migliorare le performance. Con il termine apprendimento nell'ambito degli alberi di decisione, si intende determinare tra tutti i possibili alberi di decisione quello che più approssima l'albero di decisione perfetto. L'apprendimento non termina con la selezione dell'albero, il passaggio successivo è quello di verificare che esso si comporti bene quando riceve degli esempi non visti.

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.

La forma più conosciuta di *best-first search* è **A* search**. L'algoritmo A* è un tipo di algoritmo informato, e il suo schema è identico a quello del *tree search*. La differenza sostanziale sta nella funzione con cui si decide il nodo da espandere. L'idea di A* è quella di scegliere il nodo che minimizza la funzione di stima *f(n)* data da:

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

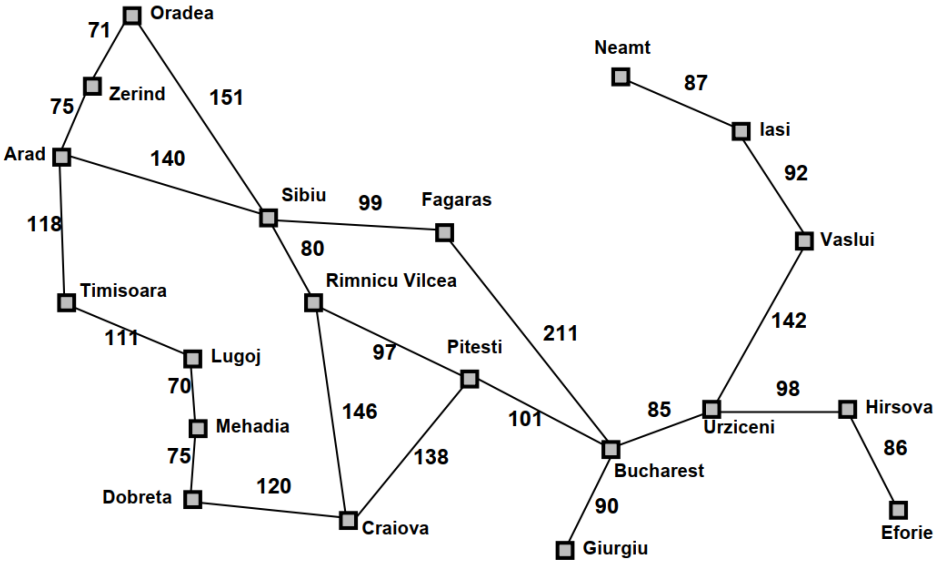
dove:

- ***g(n)*** = rappresenta la componente storica, ed è il costo impiegato per raggiungere il nodo n dalla radice.
- ***h(n)*** = rappresenta la componente euristica, cioè quella stimata, ed è il costo stimato per raggiungere il nodo obbiettivo partendo dal nodo n.
- ***f(n)*** = la funzione che calcola il costo di un nodo, ed è il costo totale stimato del percorso che, partendo dalla radice, porta fino al nodo obbiettivo passando attraverso il nodo n.

Quindi, se stiamo cercando la soluzione con costo minore, la cosa più ragionevole da fare è provare ad espandere il nodo col valore minore di ***f(n)***. Questa strategia risulta poi molto più che ragionevole, dal momento che, posto che la funzione euristica ***h(n)*** soddisfi certe condizioni, la ricerca A* risulta sia completa che ottimale.
L'algoritmo A* viene utilizzato quando si hanno a disposizione delle informazioni aggiuntive rispetto a quelle di definizione del problema; queste informazioni sono necessarie per creare la funzione di euristica.
Di fondamentale importanza è questa definizione:

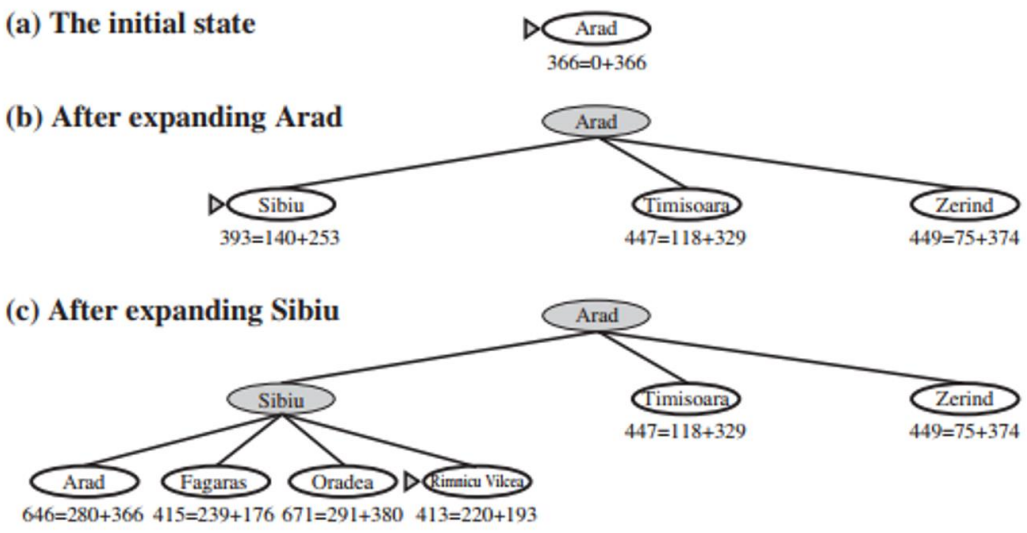
- la funzione ***h(n)*** si dice **ammissibile** per il *tree search* se non sovrastima mai il costo reale per raggiungere il nodo obbiettivo, cioè il suo valore è sempre minore o uguale del costo reale per raggiungere il nodo obbiettivo.
- la funzione ***h(n)*** si dice **consistente** per il *graph search* se non sovrastima mai il costo reale per raggiungere il nodo obbiettivo, cioè il suo valore è sempre minore o uguale del costo reale per raggiungere il nodo obbiettivo.

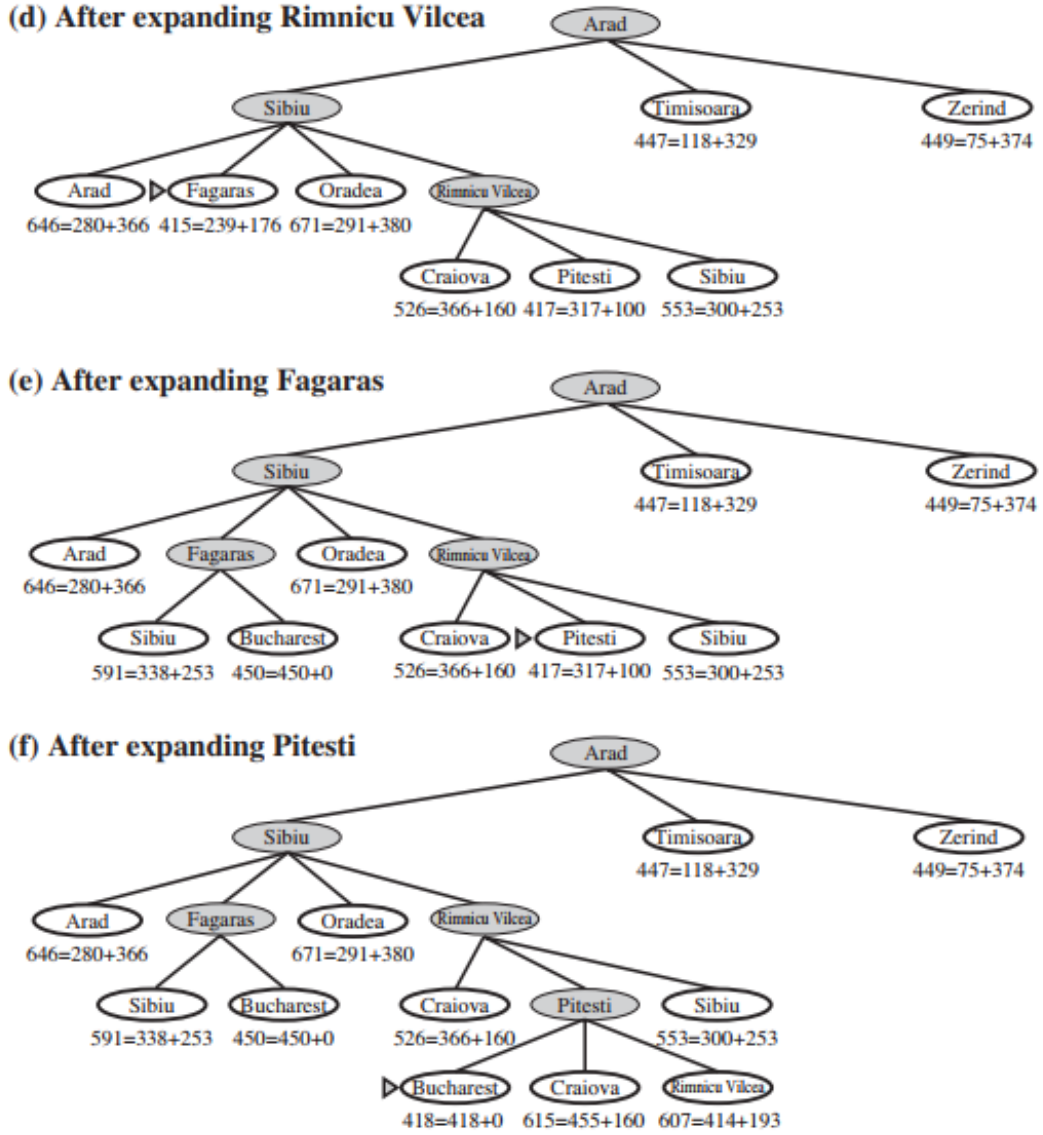
Si può dimostrare che se ***h(n)*** è ammissibile per il ***tree search*** oppure consistente per il ***graph search***, allora l'algoritmo A* è **ottimo**, cioè trova sempre la soluzione a costo minimo.
Spieghiamo il funzionamento dell'algoritmo A* su un problema di navigazione stradale. In particolare consideriamo una versione semplificata della mappa della Romania, ovvero una rappresentazione astratta in cui si ha una perfetta conoscenza del mondo. Nella realtà l'essere umano non ha una conoscenza perfetta del mondo.
Il nostro scopo è quello di andare da **Arad**, punto di partenza, fino a **Bucharest**, facendo il percorso più breve possibile. A destra della mappa, sono presenti le distanze in linea d'aria tra le varie città e la città obbiettivo, cioè **Bucharest**.



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

IL nostro punto di partenza è **Arad**, che ha un costo ***f(n)=366*** che corrisponde solamente alla componente stimata ***h(n)*** dal momento che è il nostro stato iniziale **(a)**. Espandendo **Arad (b)**, ci sono 3 nodi figli, cioè 3 città, tra le quali scegliere, e si sceglie quella che minimizza la funzione ***f(n),*** che in questo caso è **Sibiu**. Si va avanti espandendo **Sibiu (c).** Tra i nodi figli di Sibiu, si ripresenta di nuovo Arad, ma l'algoritmo A* sicuramente non sceglierà Arad, perché come si vede ha un costo nettamente grande dato che ci allontanerebbe dal nodo obbiettivo. Tra gli altri nodi figli di Sibiu, scegliamo di nuovo quello che minimizza la ***f(n),*** ovvero **Rimnicu Vilcea**. Espandendo **Rimnicu Vilcea (d)**, ci ritroviamo 3 nodi figli. Notiamo però che il nodo non ancora espanso che presenta il valore di *f(n)* minore non è un nodo figlio di Rimnicu Vilcea, ma il nodo **Faragas**, figlio di Sibiu. Espandiamo **Faragas (e)**. Tra i figli di Faragas, c'è il nodo obbiettivo, ma esso non verrà scelto per l'espansione, dato che tra i nodi non ancora espansi, quello con il valore di ***f(n)*** minore è **Pitesti**. Si espande **Pitesti (f)**, e tra i suoi nodi figli notiamo che c'è proprio il nodo obbiettivo e questa volta è il nodo con costo minore, quindi viene scelto. Questa è anche una dimostrazione pratica della ottimalità di A*, in quanto, nonostante il nodo obbiettivo Bucharest fosse comparso due volte, l'algoritmo ha scelto il percorso con costo complessivo minore.





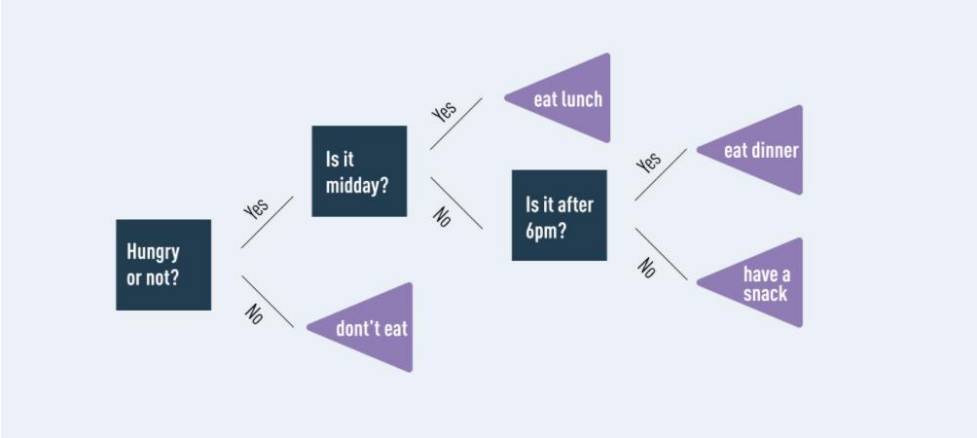
Esercizio 4

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

Una possibile rappresentazione dello spazio delle ipotesi è l'albero delle decisioni attraverso il quale è possibile rappresentare una qualsiasi funzione booleana: questa proprietà si può esprimere dicendo che l'albero di decisione ha l'espressività massima. Ad ogni riga di una tabella booleana corrisponde un percorso sull'albero delle decisioni.

L'albero di decisione è un particolare albero che, data una specifica situazione con specifici attributi, ognuno dei quali può assumere valori diversi, in base al valore che assumono gli attributi, arriva ad una conclusione. Nell'albero, i nodi intermedi e il nodo radice rappresentano tutti gli attributi; i nodi foglia rappresentano le decisioni finali; gli archi di collegamento tra i nodi rappresentano i diversi valori che gli attributi possono assumere.

Riportiamo l'esempio di un albero di decisione che consiste in una coppia *(x,y)* in cui *x* è il vettore dei valori degli attributi di input, *y* rappresenta il singolo valore di output. L'obbiettivo è *WillEat,* i cui possibili valori sono: *don't eat, eat lauch, eat, dinner, have a snack.* Il valore finale è determinato dai valori degli attributi, in questo caso Booleani.



Di seguito è presente una tabella in cui sono riportati i vari esempi in cui può essere utilizzato l'albero di decisione.

Example	Input attributes			Goals WillEat
	Hungry or not?	Is it midday?	Is it after 6pm?	
x1	NO	-	-	y1 = don't eat
x2	YES	YES	-	y2 = eat lunch
x3	YES	NO	YES	y3 = eat dinner
x4	YES	NO	NO	y4 = have a snack

Prendiamo in considerazione l'esempio x_3. Partendo dalla radice, il valore assunto dall'attributo radice *Hungry or not* è positivo, quindi si continua esplorando l'albero. Il valore del secondo attributo *Is it midday?* è negativo, così si continua e si arriva al nodo *Is it after 6pm?* è positivo, si espande quindi il nodo obbiettivo *eat dinner,* arrivando quindi alla decisione finale.

Esercizio 5

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

Per la costruzione dell'albero di decisione, dobbiamo trovare l'attributo da inserire nella radice (o nella sotto-radice) dell'albero, cioè l'attributo corrispondente ad una colonna della tabella, scelto in base all'entropia. Scelto l'attributo, disegniamo tanti rami quanti sono i valori di quell'attributo. Ad ognuno dei rami si applica ricorsivamente la procedura per la costruzione dell'albero.
La funzione dell'algoritmo presenta in ingresso:

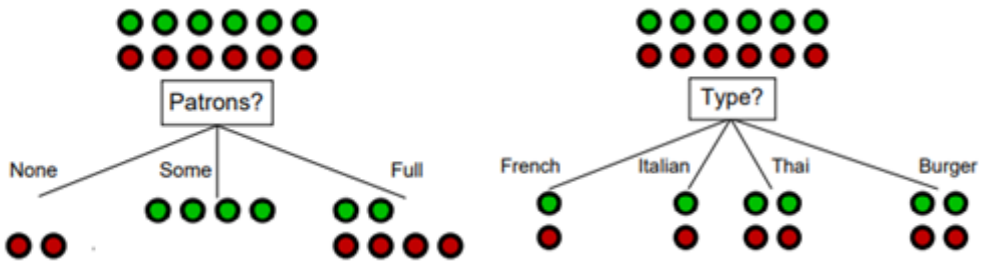
- l'insieme delle situazioni viste (gli esempi, ovvero le righe della tabella)
- gli attributi (ovvero i nomi delle colonne),
- default (classificazione di default che l'albero deve dare come risposta se non riesce a proseguire). È la classe che calcolo sulla base del valore corrente dell'insieme degli esempi. Serve perché quando arrivo al nodo che non si può espandere devo stabilire se (Vero o Falso): la soluzione che viene adottata è la classe di default.

L'algoritmo restituisce un albero di decisione costruito.

```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same classification then return the classification
    else if attributes is empty then return MODE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value v_i of best do
            examples_i ← {elements of examples with best = v_i}
            subtree ← DTL(examples_i, attributes − best, MODE(examples_i))
            add a branch to tree with label v_i and subtree subtree
        return tree
```

La funzione *MODE* è una funzione statistica che produce come risultato la classificazione maggiormente rappresentata dagli esempi. Gli esempi più numerosi determinano la classificazione associata all'albero.
La funzione *CHOOSE-ATTRIBUTE* consente tra tutti gli attributi, di sceglierne uno considerando l'insieme degli esempi scelto, che diventa *best*.
Inizialmente, l'algoritmo verifica se l'insieme *examples* è vuoto; nel caso lo fosse ritorna il valore *default.* Poi controlla i valori presenti all'interno di *examples,* nel caso in cui hanno tutti la stessa classificazione, allora ritorna quella stessa classificazione. Poi fa un altro controllo e verifica se l'insieme *attributes* è vuoto e nel caso, restituisce la *MODE* di *examples.*
Tutti questi controlli servono a dare delle condizioni di uscita all'algoritmo, altrimenti esso chiamerebbe per sempre sé stesso essendo un algoritmo ricorsivo.
Se non si è verificata nessuna delle precedenti condizioni, allora esegue effettivamente l'algoritmo. Viene assegnato un valore alla variabile *best* scelto dalla funzione *CHOOSE-ATTRIBUTE*. Per ogni valore assunto dalla variabile *best* nell'insieme degli esempi di partenza, vengono costruiti tanti sottoalberi, in cui viene creato un sottoinsieme di esempi $examples_i$ dell'insieme di partenza *examples* composto solo dagli esempi che hanno per valore dell'attributo *best* il valore v_i che sto considerando ($best = v_i$). Man mano che va avanti con la costruzione dell'albero delle decisioni gli esempi vengono ridotti.
Una volta determinato l'insieme di esempi che rispetta la condizione $best = v_i$, vado a costruire (con *DTL chiamata ricorsiva*) un nuovo albero di decisioni cioè il sottoalbero che avrà come parametri di ingresso il sottoinsieme $examples_i$ che ho calcolato, l'insieme degli attributi di partenza tolti l'attributo che ho assegnato alla variabile *best*, e come classe di default la *MODE* (ovvero la classe) degli $examples_i$. Sul cammino radice-foglia posso trovare un attributo una sola volta. Nella funzione *DTL* non posso utilizzare 2 volte lo stesso attributo dato che ogni volta l'insieme degli attributi diventerà sempre più piccolo.
Poi aggiunge il ramo all'albero con l'etichetta v_i (il valore dell'attributo best) e il sottoalbero iniziale che sto costruendo.

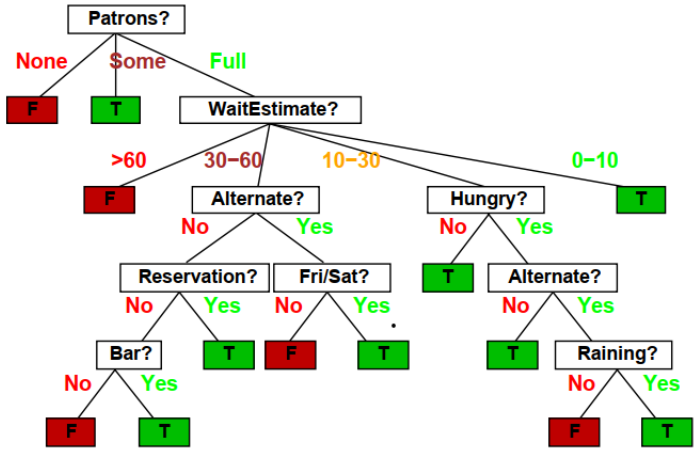
Vediamo come funziona la funzione *CHOOSE-ATTRIBUTE.*
L'idea è che un buon attributo da scegliere come radice dell'albero (o del sottoalbero) è un attributo che divide gli *examples* in sottoinsiemi che sono idealmente tutti con esito positivo o con esito negativo.



Supponiamo di voler scegliere quale tra gli attributi *Patrons* e *Type* è il più adatto come radice dell'albero. In base ai valori degli attributi, analizzo come vengono ripartiti gli esempi sulla base di questi valori. Mi porto non solo il numero degli esempi ma anche la loro classificazione (Rossi → Negativi, Verdi → Positivi). La situazione che si presenta è quella mostrata nella figura di sopra.
Quale dei 2 è migliore?
Evidentemente è meglio quello di sinistra perché con Patrons il 1° nodo è formato solo da negativi, il 2° solo da positivi e sul 3° siamo incerti. Ho 2 nodi con massima certezza.
L'albero con *Type* invece ha tutti nodi incerti. Di conseguenza l'albero di sinistra mi dà maggiore certezza a livello di classificazione è quello di sinistra.
Per riprodurre questo ragionamento in un calcolatore è necessario formularlo in termini matematici o algoritmici. In particolare, possiamo usare la funzione entropia per valutare l'incertezza a livello dei singoli nodi. La funzione entropia ci dà l'informazione dell'incertezza sul singolo nodo, ma a me non interessa valutare il singolo nodo ma tutti i nodi figli. Facciamo una somma pesata sulla base del numero di esempi, così abbiamo l'entropia del sottoalbero che può essere utilizzata per dare un valore ad ogni sottoalbero. Poi vado a selezionare il sottoalbero che ha il valore di entropia più basso, cioè che ha il minor grado di incertezza in termini di somma pesata.
Vediamo un esempio di applicazione del *decision tree,* su un esempio in cui, sulla base di alcune caratteristiche e parametri, bisogna decidere se aspettare per sedersi ad un ristorante oppure andarsene. Il punto di partenza è una tabella in cui viene esposto il risultato della decisione sulla base delle varie combinazioni di valori assunti dalle variabili. La tabella di riferimento è la seguente

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

Sulla base delle precedenti considerazioni, l'albero di ricerca che scaturisce è il seguente.



Prendiamo ad esempio in considerazione l'esempio X_{10}. Si inizia a vedere il valore di *Patrons* nell'esempio considerato e vediamo che è *Full,* quindi andiamo avanti ad esplorare l'albero e passiamo al prossimo

attributo *WaitEstimate*, ovvero il tempo stimato di attesa, e per X_{10} è compreso tra 10 e 30, quindi si passa al prossimo attributo, cioè *Hungry* che per il nostro esempio è *T,* cioè vero e quindi continuiamo. Il prossimo attributo da verificare è *Raining,* cioè se sta piovendo oppure no e nel nostro caso *F,* quindi falsa e si arriva quindi alla decisione, cioè quella di andarsene e non di aspettare.

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.

1. **Spazio degli stati** = lo stato iniziale, le azioni e il modello delle transizioni implicitamente definiscono lo spazio degli stati del problema, cioè l'insieme di tutti gli stati raggiungibili dallo stato iniziale da una sequenza di azioni.

2. **Grafo degli stati** = il grafo degli stati è un grafico formato dallo spazio degli stati. Nel grafo degli stati i nodi rappresentano gli stati e i collegamenti tra i nodi rappresentano le azioni.

3. **Albero di ricerca** = È una struttura dati che consente di esplorare il grafo degli stati al fine di individuare una soluzione che è uno dei possibili percorsi dalla radice allo stato obiettivo**.** Nell'albero di ricerca, la radice rappresenta lo stato iniziale; gli archi rappresentano le azioni e i nodi rappresentano gli stati dello spazio degli stati del problema.

Ecco il meta-algoritmo di costruzione di un *tree search*.

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

L'algoritmo ha bisogno come dati di input: il problema da risolvere e la strategia (l'euristica o anche criterio di esplorazione) che si vuole usare. Inizializzo l'albero con lo stato iniziale del problema (quindi lo stato inziale diventa la radice dell'albero). Fatto ciò, faccio un loop infinito in cui espando i nodi dell'albero uno ad uno. Faccio un controllo: se non riesco più ad espandere nessun nodo, allora vuol dire che la soluzione non c'è sul grafo, perché non esiste oppure perché non è stato usato un modello corretto. Se posso invece espandere qualche nodo, decido quale nodo espandere in base alla strategia prefissata. Una volta espanso il nodo faccio un altro controllo: il nuovo stato è il mio stato obbiettivo? Se si, allora ho trovato la soluzione, cioè il percorso dalla radice fino a quel nodo; altrimenti, aggiungo il nuovo passo al mio percorso e ripeto l'algoritmo.

A seconda di come realizziamo l'albero, andiamo a coprire parti differenti del grafo dello spazio degli stati. Le strategie vengono valutate lungo le seguenti caratteristiche:

1. **completezza**: trova sempre una soluzione se ne esiste una?
2. **complessità temporale**: numero di nodi generati / espansi (il tempo impiegato per costruire l'albero di ricerca

3. **complessità dello spazio**: numero massimo di nodi in memoria (memoria necessaria per memorizzare l'albero di ricerca)
4. **ottimale**: trova sempre una soluzione a costo minimo?

La complessità temporale e spaziale si misura in termini di

- **b**: fattore di ramificazione massimo dell'albero di ricerca (massimo n. di figli di un nodo dell'albero)
- **d** — profondità della soluzione più economica (distanza tra la radice e il nodo considerato)
- **m**: profondità massima dello spazio degli stati (può essere ∞)

Le strategie di base, dette non informate perché espandono i nodi utilizzando solo le informazioni disponibili nella definizione del problema, sono 2:

- **Breadth-first search**: strategia che utilizza una politica FIFO. Il modo di costruzione dell'albero di ricerca si sviluppa in ampiezza.
- **Depth-first search**: strategia che utilizza una politica LIFO. Il modo di costruzione dell'albero di ricerca si sviluppa in profondità.

Oltre a queste, se ne aggiunge un'altra, che è una strategia informata perché sfrutta le informazioni legate al problema:

- **Uniform-cost search**: strategia che utilizza coda ordinata in base al costo di raggiungimento del nodo. Il nodo che ha il costo minore sta in testa alla coda e sarà quello che verrà selezionato per l'espansione. L'algoritmo infatti espande sempre il nodo che ha il costo minore.

Non ci sono delle regole per trovare la strategia migliore, solo tramite l'esperienza è possibile creare delle regole empiriche.

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) = (se x < 5.2, F(x) = 10; se 5.2 <=x<= 20, F(x) = x^2; se 20 < x, F(x) = cos(x) + 160*x)
F(x) è definita sull'intervallo [-100, 100] e non definita altrove.

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.**

Con il codice riportato si vuole ottimizzare un problema di organizzazione di un viaggio di un gruppo di persone appartenenti alla stessa famiglia (la famiglia Glass) che partono da città dell'America diverse e vogliono raggiungere tutti l'aeroporto di Laguardia a New York. Nel file **optimization.py** viene definita la lista di persone associate al proprio aeroporto di partenza. Inoltre, viene fornito un file **schedule.txt** contenente una serie di voli con luogo di partenza, destinazione, orario di partenza, orario di arrivo previsto e costo totale del volo.
Nel codice sono presenti diverse funzioni che consentono di valutare la funzione di costo per algoritmi differenti:

- `def getminutes(t):` : funzione che restituisce i minuti relativi ad un singolo viaggio. Essa verrà utilizzata dalla funzione di costo, considerando il tempo trascorso come un contributo utile per la valutazione del costo totale.
- `def printschedule(r):` : funzione che stampa a video i voli presi dalle singole persone all'andata e al ritorno con i relativi prezzi dei singoli voli, i nomi dei voli e gli orari di partenza e arrivo.
- `def schedulecost(sol):` : funzione che restituisce il costo del problema, l'obiettivo è quello di minimizzare tale costo. Essa prende in considerazione per il costo il prezzo totale del viaggio e il tempo totale speso per attendere agli aeroporti i vari membri della famiglia. Inoltre, aggiunge 50$ di penalità se la macchina ritorna in ritardo rispetto all'orario del giorno in cui è stata affittata.
- `def randomoptimize(domain,costf):` : **algoritmo di ottimizzazione randomico**, esso prende in ingresso un dominio e la funzione di costo definita da **schedulcost**. Essa genera 1000 ipotesi e chiama la funzione **costf** su di essi valutando la migliore ipotesi tra quelle disponibili, ovvero quella con il costo minore e la restituisce.
- `def hillclimb(domain,costf):` : **algoritmo di Hill climbing**, genera una soluzione casuale e poi si sposta trai quelle vicine in direzione di quella con il costo minore. In particolare, considera i voli che sono in orari prossimi a quello considerato (leggermente precedente o successivo) e valuta la funzione di costo per ogni vicino. Quello con il costo minore diventa una possibile soluzione e il processo viene iterato per tutti i vicini fino a quando vi è una soluzione più promettente.

- `def annealingoptimize(domain,costf,T=10000.0,cool=0.95,step=1):` : **algoritmo di Simulated Annealing**, genera una soluzione casuale e considera una variabile che rappresenta una temperatura, che viene progressivamente decrementata fino a 0. Ad ogni iterazione la soluzione viene fatta cambiare in una certa direzione modificando qualcosa (nel nostro esempio, uno dei membri potrebbe prendere il 3° volo al posto del 2°). Il costo viene calcolato prima e dopo il cambiamento e i costi vengono confrontati: se il nuovo costo è inferiore, ci si sposta verso quella soluzione, se invece il costo è maggiore si fa un passo peggiorativo sulla base di una certa probabilità:

$$p=e^{(-highcost-lowcost)temperature}$$

che va a decrescere man mano che la variabile "temperature" decresce.

- `def geneticoptimize(domain,costf,popsize=50,step=1,mutprob=0.2,elite=0.2,maxiter=100):` : algoritmo genetico, prende in considerazione un set di soluzioni casuali, dette popolazione. Ad ogni fase dell'ottimizzazione viene calcolata la funzione di costo per l'intera popolazione per ottenere un elenco ordinato di soluzioni. In seguito, le funzioni vengono modificate in due modi:
    - il 1° consiste, attraverso la **mutazione**, nella modifica di un elemento nella popolazione in maniera random, che potrebbe portare ad una diminuzione del costo della funzione;
    - il 2° metodo è il **crossover** e consiste nello scegliere un determinato punto nelle varie popolazioni e nel combinarle fra loro per cercare di ottenere dei figli con un costo minore.

Applicando le varie funzioni di ottimizzazione descritte attraverso i seguenti codici:

```
import optimization
s=[1,4,3,2,7,3,6,3,2,4,5,3]
optimization.printschedule(s)


optimization.schedulecost(s)

#Algoritmo randomico
reload(optimization)
domain=[(0,8)]*(len(optimization.people)*2)
s=optimization.randomoptimize(domain,optimization.schedulecost)
optimization.schedulecost(s)
optimization.printschedule(s)

#Algoritmo Hill Climbing
s=optimization.hillclimb(domain,optimization.schedulecost)
optimization.schedulecost(s)

optimization.printschedule(s)

#Algoritmo Simulated Annealing
reload(optimization)
s=optimization.annealingoptimize(domain,optimization.schedulecost)
optimization.schedulecost(s)

optimization.printschedule(s)

#Algoritmo genetico
s=optimization.geneticoptimize(domain,optimization.schedulecost)

optimization.printschedule(s)
```

Otterremo i seguenti OUTPUT:

1) **OUTPUT senza ottimizzazione**

```
   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

4635
```

2) **OUTPUT con ottimizzazione mediante algoritmo randomico**

```
   Seymour       BOS  8:04-10:11 $ 95  6:39- 8:09 $ 86
    Franny       DAL 18:26-21:29 $464 12:20-16:34 $500
     Zooey       CAK 18:35-20:28 $204 18:17-21:04 $259
      Walt       MIA  7:34- 9:40 $324 12:37-15:05 $170
     Buddy       ORD  8:25-10:34 $157  6:03- 8:43 $219
       Les       OMA 16:51-19:09 $147 16:35-18:56 $144

7235
```

3) **OUTPUT con ottimizzazione mediante algoritmo Hill Climbing**

```
   Seymour       BOS  6:17- 8:26 $ 89  6:39- 8:09 $ 86
    Franny       DAL 10:30-14:57 $290 17:14-20:59 $277
     Zooey       CAK 12:08-14:59 $149  8:19-11:16 $122
      Walt       MIA 11:28-14:40 $248 12:37-15:05 $170
     Buddy       ORD  9:42-11:32 $169  7:50-10:08 $164
       Les       OMA 12:18-14:56 $172  8:04-10:59 $136

3995
```

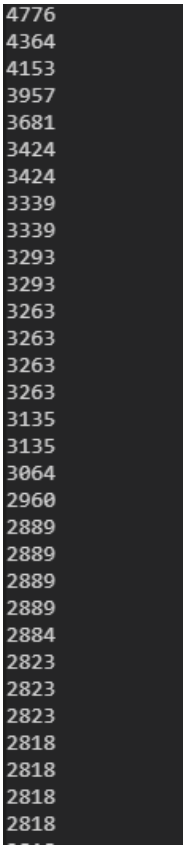
4) **OUTPUT con ottimizzazione mediante algoritmo Simulated Annealing**

```
   Seymour       BOS 15:27-17:18 $151 10:33-12:03 $ 74
    Franny       DAL 10:30-14:57 $290 14:20-17:32 $332
     Zooey       CAK  8:27-10:45 $139 13:37-15:33 $142
      Walt       MIA 14:01-17:24 $338  9:25-12:46 $295
     Buddy       ORD  9:42-11:32 $169 10:33-13:11 $132
       Les       OMA  9:15-12:03 $ 99  9:31-11:43 $210

4335
```

5) **OUTPUT con ottimizzazione mediante algoritmo Genetico**

```
   Seymour       BOS 13:40-15:37 $138 10:33-12:03 $ 74
    Franny       DAL 10:30-14:57 $290  9:49-13:51 $229
     Zooey       CAK 13:40-15:38 $137  8:19-11:16 $122
      Walt       MIA 11:28-14:40 $248  8:23-11:07 $143
     Buddy       ORD  9:42-11:32 $169  7:50-10:08 $164
       Les       OMA 12:18-14:56 $172  8:04-10:59 $136
```



Continua così stampando sempre il valore 2818.
Se si rieseguissero le funzioni di ottimizzazione, si otterrebbero valori differenti poiché tutti gli algoritmi partono da valori casuali definiti all'interno della variabile **domain**.

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

F(x) = (se x < 5.2, F(x) = 10; se 5.2 <=x<= 20, F(x) = x^2; se 20 < x, F(x) = cos(x) + 160*x)
F(x) è definita sull'intervallo [-100, 100] e non definita altrove.

Nel 2° punto dell'esercizio si vuole ottimizzare una funzione matematica, ovvero ricercare il massimo globale della funzione. Per farlo applichiamo l'**algoritmo di Hill Climbing.** La funzione matematica che si vuole ottimizzare è la seguente:

F(x) = (se x < 5.2, F(x) = 10; se 5.2 <=x<= 20, F(x) = x^2; se 20 < x, F(x) = cos(x) + 160*x)
F(x) è definita sull'intervallo [-100, 100] e non definita altrove.

Innanzitutto, creiamo un file **funzione_matematica.py** in cui si andrà a definire:
1) la funzione, che corrisponderà alla funzione di costo (in quanto il costo di una soluzione x corrisponde proprio a $f(x) = y$)
2) un'altra funzione che consentirà di tracciare il grafico per poter osservare l'andamento della funzione stessa definita dalla traccia.
Per definire la funzione all'interno dell'ambiente Python dobbiamo utilizzare delle liste di valori perché gli algoritmi di ottimizzazione lavorano solo con liste. Pertanto, indichiamo con x = [ ] la lista dei valori in ingresso alla funzione e con y = [ ] la lista dei valori di uscita che la funzione deve fornire.
Il codice rispettando le specifiche sopra definite è il seguente:

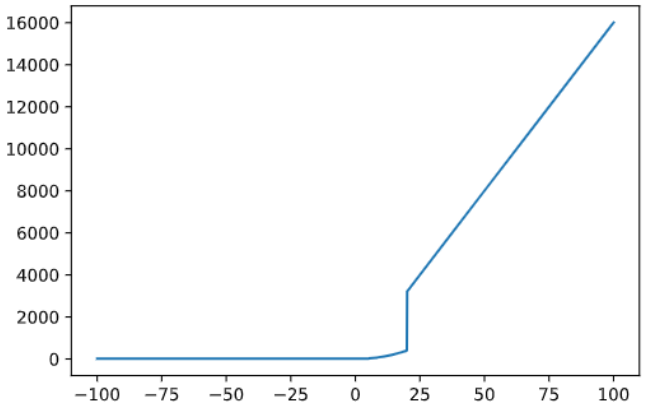
```python
import math
import matplotlib.pylab as p
import optimization as op
import numpy as n
#si vuole ottimizzare la funzione F(x) definita nell'intervallo [-100,100]

def traccia_grafico():
    ascissa = n.linspace(-100, 100, 2000)
    limit = ascissa.size
    count_1 = 0
    x = []
    while count_1 < limit:
        x.append(ascissa[count_1])
        count_1 += 1
    count_2 = 0
    y = []
    while count_2 < limit:
        if ascissa[count_2] < 5.2:
            y.append(10)
        else:
            if 5.2 < ascissa[count_2] < 20:
                y.append(ascissa[count_2]*ascissa[count_2])
            else:
                y.append(math.cos(ascissa[count_2]) + 160*ascissa[count_2])
        count_2 += 1

    p.plot(x,y)
    p.show()

def funzione(x):
    y=[]
    for i in range(len(x)):
        if x[i] < 5.2:
            y.append(18)
        else:
            if 5.2 <= x[i]<= 20:
                y.append(x[i]*x[i])
            else:
                y.append(math.cos(x[i]) + 160*x[i])
    return y
```

La funzione funzione(x) definisce la funzione sopra riportata. Essa prende in ingresso una lista x di valori. Questi vengono poi utilizzata dalla funzione traccia_grafico() per stampare a video il grafico che descrive l'andamento della funzione F(x) sulla base del numero di campioni generati (di base 2000).
Se eseguiamo il codice otteniamo il seguente andamento:



Dal grafico si nota che la funzione è strettamente crescente nell'intervallo, al crescere del numero di campioni generati. Dunque, il massimo globale si troverà nell'estremo superiore dell'intervallo di definizione della funzione.
Per trovare il massimo globale utilizziamo l'algoritmo di ottimizzazione Hill Climbing applicato alla funzione F(x):

```
def run():
    domain =[(-100,100)]
    count = 0
    count2 = 0
    count3 = 0
    max_trovati = []
    tentativi=2000
    while count<tentativi:
        max_locale = op.hillclimb(domain,funzione)
        max_trovati.append(max_locale[0])
        count = count + 1
    while count2 < max_trovati.__len__():
        if max_trovati[count2]==100:
            count3 = count3 + 1
        count2 = count2 + 1
    print("MAX_globale trovato per ", count3 ," volte dopo: ",count2,"tentativi")
    print(" percentuale di successo è :", count3*100/2000,"%")
```

In particolare, definiamo il dominio di definizione con la variabile domain e creata una lista dei massimi trovati tra i quali sarà poi necessario individuare quello globale.

Tuttavia, data la casualità dell'algoritmo, saranno trovati vari massimi ma pochi corrisponderanno al massimo globale. Ciò avviene anche perché per un certo intervallo [-100 ≤ x < 5,2] la funzione è costante e l'algoritmo non troverà punti migliori, dunque si fermerà sul punto trovato. Applicando l'algoritmo N volte, è possibile fare una valutazione statistica e vedere quante volte l'Hill climbing trova il massimo globale. Le ultime 2 righe di codice generano il seguente output:

```
MAX_globale trovato per  14  volte dopo:  2000 tentativi
 percentuale di successo è : 0.7 %
```

Esercizio 2

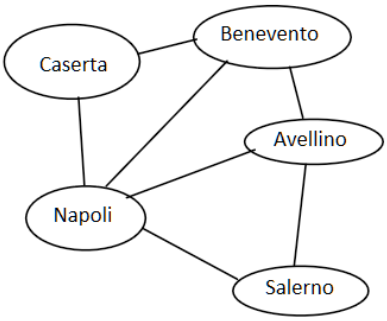
1) Un agente Intelligente deve colorare le provincie della Regione Campania evitando che due provincie confinanti abbiano lo stesso colore, l'Agente ha a disposizione solo 2 colori. Se il compito non è possibile spiegare perché non si riesce utilizzando gli algoritmi visti a lezione.
2) Ripetere il punto 1 con 3 colori a disposizione.
3) Ripetere il punto 1 con 4 colori a disposizione.

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.

Consideriamo la seguente mappa della Regione Campania

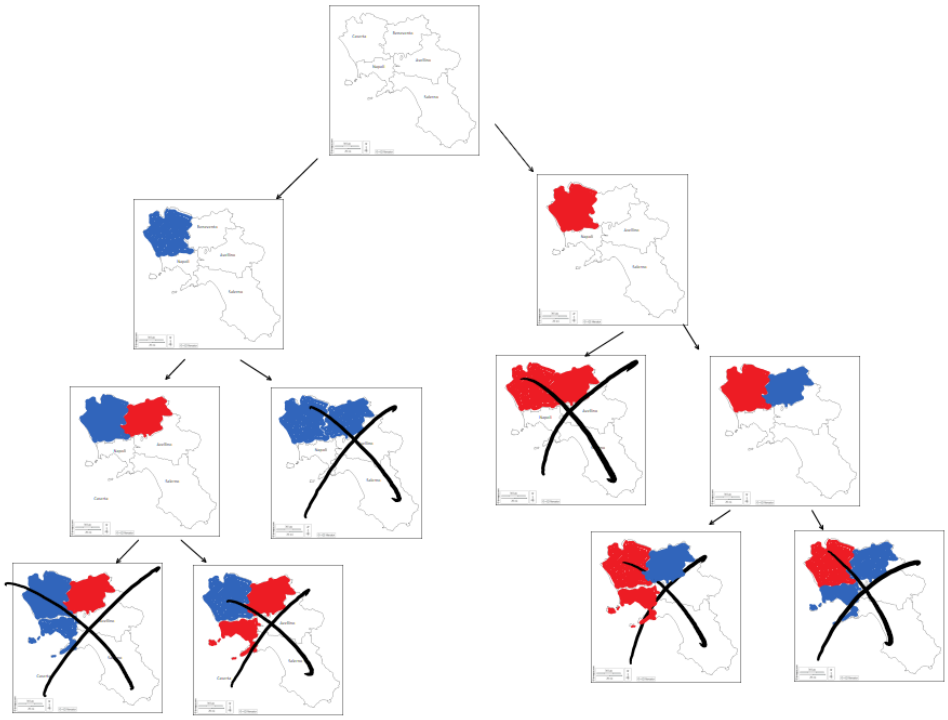
Fonte: https://d-maps.com/pays.php?num_pay=399&lang=it

Il grafo dei vincoli è il seguente:



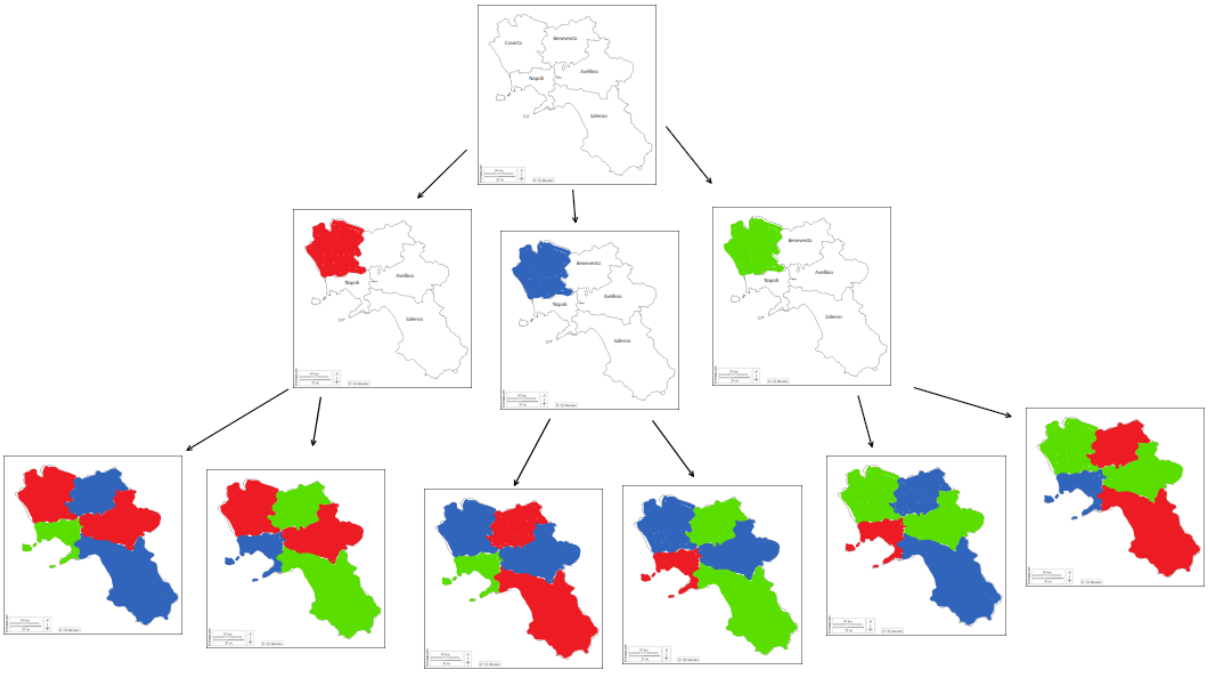
Se andiamo ad utilizzare solo 2 colori non è possibile colorare le province in modo che le province adiacenti abbiano colori diversi, questo perché semplicemente ogni provincia confina con almeno altre due.
Usando l'algoritmo di *Backtracking search* ci si accorge che tale compito non è fattibile.
L'algoritmo *Backtracking search* assegna alle variabili, in questo caso le province, dei valori, in questo caso i colori, inizialmente provvisori e continua fin quando non si accorge eventualmente che un vincolo non è più rispettato; in quel caso torna indietro nelle assegnazioni dei valori e cambia assegnazione. Se non ci sono assegnazioni che rendono tutti i vincoli rispettati (come in questo caso), allora ritorna un fallimento, cioè che il compito con quei vincoli non è possibile.

Se assegniamo a una provincia X il colore 1 e successivamente viene assegnato a una provincia adiacente Y il colore 2, esisterà sempre una provincia Z adiacente alla provincia X e Y tale per cui qualsiasi colore assegnato provocherà la violazione dei vincoli.



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

Applicando l'algoritmo di *Backtracking search* e considerando 3 colori (rosso, verde, blu) si ottengono 6 possibili soluzioni.



Svolgimento assegnazione esatta:
1) Assegniamo il colore X alla provincia di Caserta
2) Assegniamo un colore Y alla provincia di Napoli e un colore Z alla regione di Benevento
3) Alla provincia di Avellino viene assegnato il colore X (lo stesso colore della provincia di Caserta) alla provincia di Salerno viene assegnato il colore Z (lo stesso colore della provincia di Benevento)

N.B. nel caso in cui viene assegnato dopo il passo 1 a una provincia un colore che non soddisfa i vincoli l'algoritmo annulla la scelta del colore e assegna un nuovo colore.

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

Si effettua il procedimento precedente ma usando 4 colori
1) Si assegna a una provincia un colore X
2) si assegna a una provincia adiacente un colore Y che soddisfa i vincoli se non soddisfa i vincoli viene scelto un nuovo colore
3) si ripete il passaggio 2) fino a quando tutte le province sono state colorate rispettando i vincoli

La differenza sostanziale sta nel fatto che c'è più libertà di scelta tra i colori, dato che le province a cui sono stati assegnati i valori per ultime, non sono vincolate ad assumere un solo colore per soddisfare i vincoli, ma possono scegliere tra più colori. Di conseguenza il risultato è che ci saranno molti più risultati corretti.

Esercizio 3

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

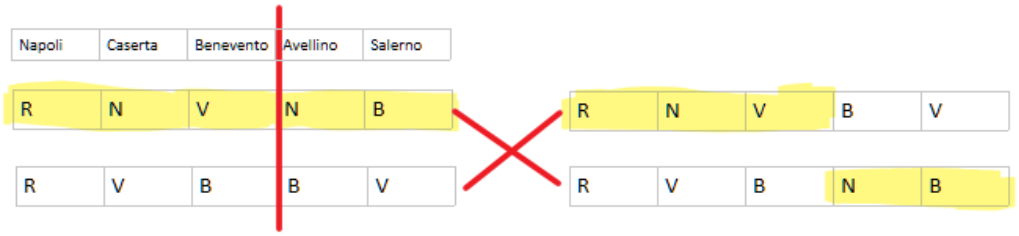
È necessario codificare gli stati in stringhe, nel nostro problema si procede assegnando a ogni stato una stringa di 5 elementi (ogni elemento rappresenta una provincia). A ogni elemento può essere assegnato uno dei seguenti colori: B=blu, R=rosso, V=verde, N=nero.

Le province adiacenti sono rappresentate come celle adiacenti eccetto per Napoli.
Napoli è un caso particolare dato che è adiacente con tutte le altre e lo considereremo come primo elemento della stringa.

L'assegnazione dei colori sarà casuale e attribuiremo un costo a ogni soluzione che sarà maggiore nel caso province adiacenti abbiamo lo stesso colore, in modo tale che la funzione fitness avrà come obiettivo di minimizzare il costo cioè ottenere tutte le province adiacenti con colori diversi.
Le condizioni sono:

- la cella da 1 a 4 devono avere valori adiacenti diversi
- la cella da 1 a 4 devono avere valori diverso dalla cella 0

Esempio:



Andando a combinare 2 stati, 1 stato che soddisfa i vincoli e 1 che non li soddisfa otteniamo 2 stati che soddisfano i vincoli.

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

1) cosa si intende per conseguenza logica
Data una base di conoscenza KB, l'agente logico deve essere in grado di dedurre informazioni aggiuntive date quelle presenti nella KB. Per **conseguenza logica** (o **entailment**) si intende quando una informazione, in questo caso una proposizione α, proviene, quindi è deducibile, da altre informazioni, in questo caso presenti in una KB. Il simbolo che indica la conseguenza logica è il seguente: KB |= α, cioè α è conseguenza logica di KB. abbiamo che KB |= α è vera se e solo se tutti i modelli in cui è vera KB sono inclusi nell'insieme dei modelli in cui è vera α, cioè che i modelli in cui è vera α sono un sovra-insieme dei modelli in cui è vera KB. Il metodo che permette di dedurre α da KB si chiama **inferenza**, infatti KB $|-_i$ α vuol dire che la proposizione α è deducibile da KB tramite l'inferenza i.

2) dimostrazione tramite model checking
L'obbiettivo è verificare se, data una certa KB, allora KB |= α.
Il model checking è un modo per poter verificare se tutti i modelli in cui KB è vera sono anche modelli in cui α è vera. Per arrivare all'inferenza posso utilizzare la **tabella di verità per l'inferenza**, cioè una tabella
i ncui sono elencati i modelli in cui KB è vera e in cui è falsa. Prendiamo ad esempio la seguente tabella:

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*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*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Per alcune assegnazioni di modelli, la KB è vera e per altri modelli è falsa. Se consideriamo una α qualsiasi, ad esempio *not* $P_{1,2}$, allora è possibile dire che KB implica α andando a controllare le righe in cui KB è vera e i modelli in cui α è vera. Deve verificarsi che tutti i modelli in cui α è vera siano un sovra-insieme (oppure lo stesso insieme) dell'insieme di modelli in cui è vera KB. Sulla tabella possiamo notare che se KB è vera, allora α deve essere necessariamente vera, quindi si può dire che α è deducibile da KB.

3) cosa si intende per deducibilità logica/inferenza

L'inferenza è un procedimento che permette di ottenere da alcune proposizioni, quindi da una KB, delle nuove proposizioni. Un algoritmo di inferenza che deriva solo proposizioni che sono conseguenza logica viene definito corretto. In particolare, sono due le caratteristiche di una inferenza:

- **Correttezza:** l'inferenza i e detta corretta se ogniqualvolta KB $|-_i \alpha$, è anche vero che KB $|= \alpha$.
- **Completezza:** l'inferenza i è completa se ogniqualvolta che KB $|= \alpha$, allora è anche vero che KB $|-_i \alpha$.

4) dimostrazione tramite inferenza logica (regola di risoluzione)

L'algoritmo di risoluzione permette di verificare che una data preposizione α sia conseguenza logica di una KB. I particolar modo l'algoritmo va a verificare in realtà che KB ^ NOT α in realtà porta ad una *clausola vuota,* cioè ad una proposizione sempre falsa, cioè che è insoddisfacibile, ovvero non è vera in nessun modello. Così facendo si può dimostrare che KB $|= \alpha$. l'algoritmo è il seguente:

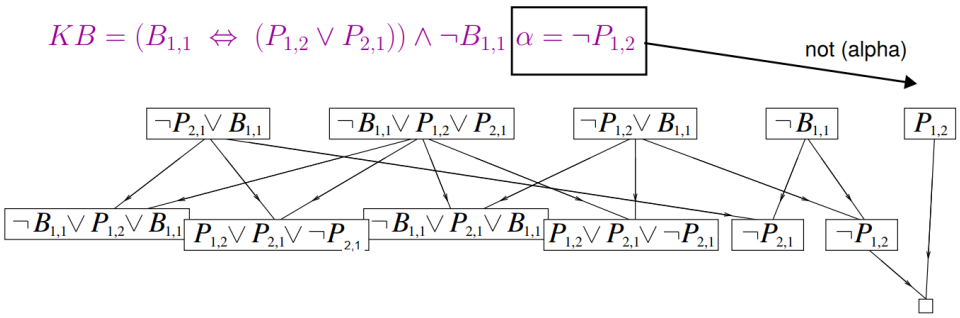
```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic
  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
    for each C_i, C_j in clauses do
      resolvents ← PL-RESOLVE(C_i, C_j)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

L'algoritmo riceve in ingresso tutte le clausole della KB e una proposizione α, della quale si deve dimostrare se è conseguenza logica della KB data. L'algoritmo prova tutte le combinazioni possibili tra clausole per fare

delle inferenze, in particolar modo, definisce un insieme di clausole *clauses,* scritto in modo da rispettare la CNF (Conjuntive Normal Form) dell'insieme KB ^ NOT α. Successivamente definisce una clausola vuota *new.* Poi fa un loop e per ogni coppia C_i, C_j di clausole, esegue il PL_RESOLVE, applicando la regola di inferenza e i risultati (cioè le nuove eventuali proposizioni) le inserisce in una variabile *resolvents.* Poi fa un controllo, se *revolvents* contiene la variabile vuota, cioè non è stata prodotta nessuna nuova proposizione, allora ritorna il valore true: questa situazione è la situazione di successo, dove si è dimostrato che KB ^ NOT α è insoddisfacibile; altrimenti aggiunge alla variabile *new* il risultato ottenuto, e se *new* è un sottoinsieme di *clauses* allora ritorna false, cioè condizione di fallimento perché α non è conseguenza logica di KB. Poi aggiunge a *clauses* i valori presenti nella variabile *new.*

5) un esempio di inferenza logica (risoluzione) espressa nella logica proposizionale

Come esempio possiamo prendere in considerazione quello del Wumpus World.
Con le proposizioni P si indica la presenza di un pozzo, invece con le proposizioni B si indica la presenza della brezza, coi pedici si fa riferimento alla posizione nella scacchiera del pozzo o della brezza. La KB risulta vera solo quando sono vere entrambe le seguenti proposizioni: "c'è brezza in 1,1 se e solo se c'è un pozzo in 1,1 oppure un pozzo in 2,1" e "non c'è brezza in 1,1". Si vuole verificare se la proposizione α = "non c'è un pozzo in 1,2" è una conseguenza logica della KB, e, come detto, può essere verificato andando ad aggiungere la proposizione NOT α alla KB e vedendo che il risultato non è altro che la clausola vuota, quindi è insoddisfacibile. Graficamente si avrà:



ESERCITAZIONE 3

Esercizio 1

1) Riassumere i concetti principali delle reti Bayesiane (rappresentazione e inferenza).
2) Descrivere e spiegare con un esempio come il formalismo delle reti Bayesiane può essere applicato.

1) Riassumere i concetti principali delle reti Bayesiane (rappresentazione e inferenza).

La rete Bayesiana è un grafo per l'indipendenza condizionale delle asserzioni, in cui le variabili aleatorie, con cui rappresentiamo i concetti del dominio, costituiscono i nodi del grafo e i cui archi sono le relazioni causali definite dall'essere umano. Ad ogni nodo è associata una tabella di probabilità.
In particolare:

- i nodi che non hanno archi entranti sono detti **cause iniziali** e rappresentano i fenomeni principali del dominio. Per questi nodi viene definita una probabilità a priori stabilita dall'ingegnere della conoscenza.
- I nodi che hanno archi entranti sono detti **nodi interni**. Se una variabile aleatoria si trova su un nodo interno vuol dire il suo valore è condizionato dalle sole sue cause dirette, ovvero quelle direttamente collegate ad essa. Per questi nodi viene definita una tabella della probabilità condizionata che indica come le cause dirette di una variabile aleatoria influenzino la variabile stessa.

In una rete Bayesiana infatti, ad ogni nodo, dati i suoi genitori, è associata una **distribuzione condizionale**

$$P(X_i|Parents(X_i)P(XiParents(Xi)$$

: la probabilità di un nodo dipende solo dai suoi genitori, ovvero è condizionata solo dalle sue cause dirette.
Il grafo è diretto aciclico ovvero privo di cicli, ciò significa che la variabile aleatoria di partenza non influenza in alcun modo la variabile aleatoria che si trova nel nodo terminale.
Una rete Bayesiana si costruisce seguendo i seguenti passi:

1. Scegliere un ordine tra tutte le variabili aleatorie $X_1, \ldots, X_n$. L'ordine da scegliere deve essere un ordine causale, cioè di collegamento causa-effetto.
2. *For i=1 to n*
   aggiungi X_i alla rete
   seleziona parenti da $X_1, \ldots, X_{i-1}$ in modo che

$$P(X_i|Parents(X_i)=P(X_i|X_1, \ldots, X_{i-1})P(XiParentsXi=P(XiX1, \ldots, Xi-1)$$

Ciò vuol dire che ogni volta che si aggiunge un variabile aleatoria X_i alla rete, esso dipende fa tutte le variabili già presenti, ma per una rete Bayesiana corretta, deve valere questa relazione, cioè si possono ignorare tutte le variabili aleatorie, tranne i genitori diretti di X_i. Questo è possibile perché le variabili aleatorie vengono ordinate in ordine causale, altrimenti ciò non sarebbe stato vero.

L'inferenza nelle reti Bayesiane può essere fatta in 2 modi:
1) Inferenza per enumerazione
Si ottiene mediante una ricombinazione delle tabelle di probabilità condizionata degli elementi presenti sulla rete, facendo alcune semplificazioni e mediante l'applicazione del **meccanismo di marginalizzazione delle variabili** applicato alla rete bayesiana: ovvero applicando la marginalizzazione sulla sommatoria della produttoria delle probabilità condizionate valutate sulla rete.
Supponiamo ad esempio di voler valutare la probabilità che ci sia un furto dato che entrambi i vicini (John e Mary) chiamino.

$$P(B|j,m) = \frac{P(B,j,m)}{P(j,m)} = \alpha P(B,j,m) = \alpha \sum_e \sum_a P(B,e,a,j,m)$$

Applicando la caratteristica delle reti Bayesiane precedentemente descritta è possibile riscrivere la probabilità congiunta come produttoria delle probabilità condizionate:

$$P(b \mid j, m) = \alpha \sum_{e} \sum_{a} P(b)P(e)P(a \mid b, e)P(j \mid a)P(m \mid a)$$

Per calcolare questa espressione, nel peggiore dei casi, dove dobbiamo riassumere quasi tutte le variabili, la complessità dell'algoritmo per una rete con n variabili booleane è O ($n2^n$). Un miglioramento può essere ottenuto dalle seguenti osservazioni: il termine *P(b)* è una costante e può essere spostato al di fuori della somma su *a* ed *e*, e il termine *P(e)* può essere spostato al di fuori della somma su a. Pertanto, si ottiene:

$$P(b \mid j, m) = \alpha\, P(b) \sum_{e} P(e) \sum_{a} P(a \mid b, e)P(j \mid a)P(m \mid a)$$

In questo modo, è possibile calcolare la probabilità richiesta, calcolando le sole probabilità condizionate. Questo calcolo può essere però ulteriormente semplificato applicando la marginalizzazione che può essere automatizzata mediante l'utilizzo di un algoritmo.
Alla fine, si ottiene che la probabilità richiesta è:

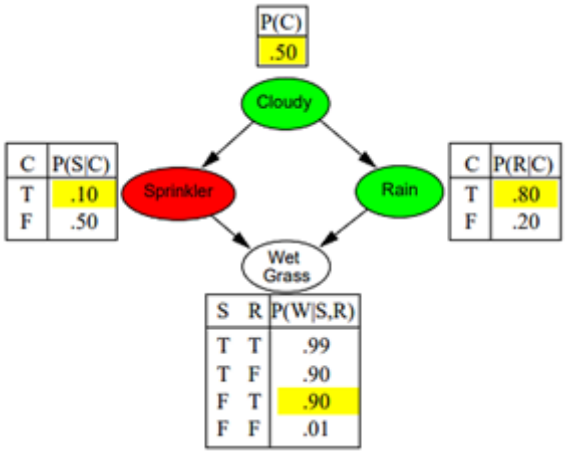
$$\mathbf{P}(B \mid j, m) = \alpha \langle 0.00059224, 0.0014919 \rangle \approx \langle 0.284, 0.716 \rangle$$

2) Inferenza per simulazione stocastica

Si effettua in casi in cui si ha un insieme di variabili aleatorie la cui distribuzione di probabilità non è nota.
Si estraggono N campioni da una distribuzione di probabilità nota facendo cadere un gettone astratto lungo la rete Bayesiana. Ogni gettone rappresenta un **campionamento statistico**, e per ogni gettone si memorizza l'evento che esso ha prodotto.
In particolare, supponendo di far cadere il 1° gettone: esso avrà probabilità di 0.5 che esca testa o croce, in base al risultato si assumerà vero o falso il valore della prima variabile. Scendendo lungo la rete esso determinerà con una certa probabilità i valori assunti dalle restanti variabili aleatorie e memorizzo il campione ottenuto che sarà dato da tutti i valori assunti dalle variabili all'attraversamento del gettone lungo la rete.
Facendo cadere tanti gettoni in modo tale da avere un campionamento significativo è possibile calcolare la frequenza con cui un certo evento si verifica nella sequenza di eventi considerati. In particolare, si dimostra che se chiamiamo $\mathbf{N_{PS}(x_1 \ldots\ldots x_n)}$ il numero di campioni generati per l'evento $\mathbf{x_1 \ldots\ldots x_n}$ (cioè quante volte occorre quel dato campione) ed effettuiamo il limite per N che tende a infinito di questo numero diviso il numero di campioni totali esso convergerà alla probabilità vera di quell'evento:

$$\lim_{N\to\infty} \hat{P}(x_1, \ldots, x_n) = \lim_{N\to\infty} \frac{N_{PS}(x_1, \ldots, x_n)}{N} = S_{PS}(x_1, \ldots, x_n) = P(x_1, \ldots, x_n)$$

Per esempio, possiamo modellare una rete che descrive la probabilità che l'erba del giardino sia bagnata dato che gli annaffiatoi si siano accessi e/o abbia piovuto, eventi legati a loro volta dal fatto che il tempo sia o meno nuvoloso.

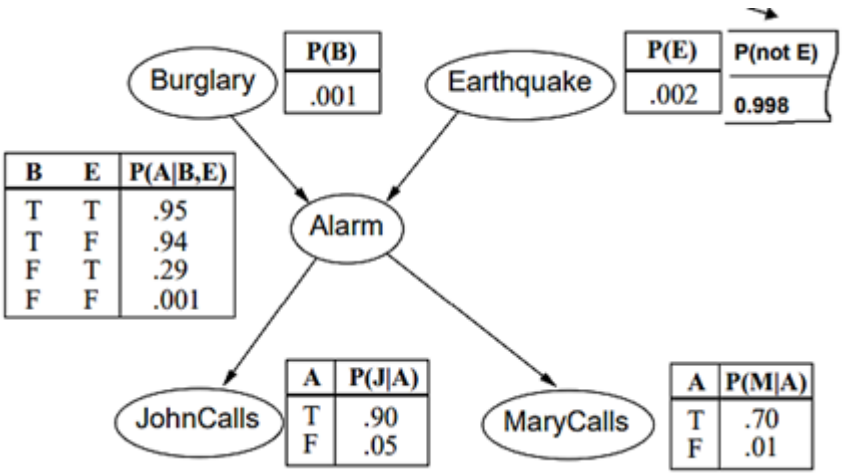


Nell'esempio si è ipotizzato che il gettone lanciato sia quello che fornisce il valore **V** alla variabile **Cloudy**. Allo stesso modo, la probabilità che gli annaffiatoi (**Sprinkler**) si attivino dato che il tempo è nuvoloso è del 10%, rispetto al 90% di probabilità che non si attivino, quindi si assume che la variabile sia **F**. Si applica lo stesso ragionamento per la pioggia e si assume che la variabile (**Rain**) sia **V**. Infine, in base ai valori di **Sprinkler** e **Rain**, si deduce che la probabilità che l'erba sia bagnata dato che abbia piovuto e gli annaffiatoi non si siano attivati è del 90%, pertanto si può attribuire **V** alla variabile.
Come si nota il valore finale è dato da un campionamento statistico ovvero dai valori assunti dalle singole variabili aleatorie al passaggio del gettone all'interno della rete. In questo caso la probabilità con cui è generato l'evento (Erba bagnata) è dato dalla formula:

$$S_{PS}(t,f,t,t) = 0.5 * 0.9 * 0.8 * 0.9 = 0.324 = P(t,f,t,t)$$

2) Descrivere e spiegare con un esempio come il formalismo delle reti Bayesiane può essere applicato.

Per spiegare come il formalismo delle reti Bayesiane possa essere applicato, consideriamo la seguente rete Bayesiana:



Questa rete si basa su un esempio pratico: una persona è a lavoro ed in casa ha installato un antifurto. L'allarme di casa suona. Potrebbe aver suonato perché c'è un ladro o perché c'è stato un terremoto (a causa della sensibilità dell'allarme). Se l'allarme suona, i suoi vicini, John e Mary, potrebbero chiamarlo. Però per un motivo o per un altro potrebbero anche non sentire l'allarme o confonderlo con un altro suono. Per la costruzione della rete si utilizzano le seguenti variabili: **Burglary, Earthquake, Alarm, JohnCalls, MaryCalls**.
In questo esempio:

- **Burglary, Earthquake** rappresentano le **cause iniziali** alle quali sono associate delle probabilità a priori scaturite ad esempio, dal luogo in cui si trova il soggetto.
- **Alarm**, **JohnCalls, MaryCalls** rappresentano i **nodi interni** influenzati solo dalle variabili presenti nei nodi che puntano ad essi:
    - **Alarm** è una variabile aleatoria condizionata dalle sue cause dirette **Burglary, Earthquake,** e pertanto avrà associata una tabella di probabilità condizionata che tiene conto di tutte le possibili combinazioni di valori da essi assunti.
    - **JohnCalls, MaryCalls** sono variabili aleatorie condizionate dalla sola causa diretta **Alarm**, pertanto anch'esse avranno associate 2 tabelle di probabilità condizionate che tengono conto di tutti i possibili valori assunti da **Alarm**.

Per Burglary si definisce come probabilità a priori P(B) = 0.001 mentre per Earthquake si definisce come probabilità a priori P(E) = 0.002. Le probabilità condizionate invece vengono valutate sulla base dei possibili valori assunti dalle cause dirette per ciascuna variabile aleatoria.
L'obiettivo principale in una rete Bayesiana è quello di determinare la distribuzione di probabilità congiunta delle variabili del dominio. Per farlo possiamo sfruttare una caratteristica delle reti Bayesiane che presa una semantica locale, è possibile ricostruire la probabilità congiunta come il prodotto delle distribuzioni di probabilità condizionate locali:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | parents(X_i))$$

e.g., $P(j \wedge m \wedge a \wedge \neg b \wedge \neg e)$

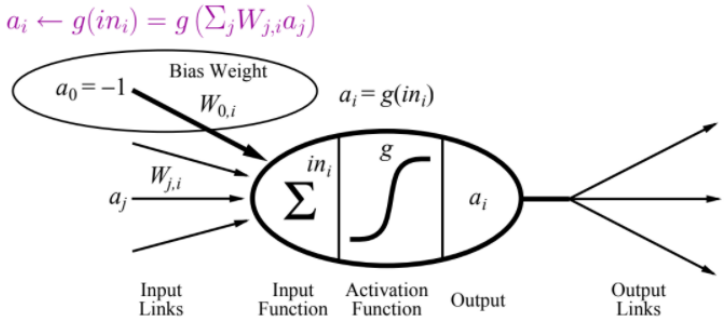
$$= P(j|a)P(m|a)P(a|\neg b, \neg e)P(\neg b)P(\neg e)$$
$$= 0.9 \times 0.7 \times 0.001 \times 0.999 \times 0.998$$
$$\approx 0.00063$$

Esercizio 2

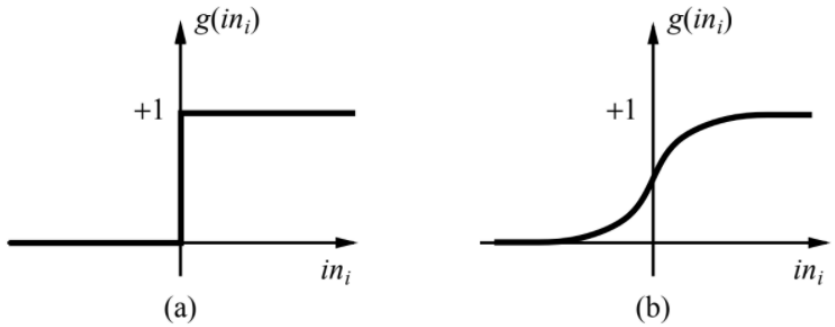
a) Riassumere i concetti principali delle reti neurali (spiegare come sono implementati i neuroni, la loro architettura e l'algoritmo di error back propagation)
b) come utilizzereste una rete neurale per riconoscere la presenza o meno di alberi in un'immagine digitale

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

Con il termine reti neurali si intende semplicemente una struttura a grafo. Il **neurone** è l'unità complementare di computazione delle reti neurali. Essi sono delle unità che prendono in ingresso determinati input di attivazione e producono, in funzione di questi, una serie di output. Gli input sono provenienti da vari archi, ognuno pesato in maniera differente in base all'intensità dell'ingresso. Una funzione di input effettua una combinazione lineare degli ingressi con i rispettivi pesi $W_{j,i}$ associati agli archi. Il risultato va in ingresso ad una funzione di attivazione che si attiva quando si oltrepassa una certa soglia e viene così prodotto l'output, che viene poi sparato sui vari archi di uscita del neurone.



La **funzione di attivazione** è una funzione che "fa passare" solamente gli ingressi con un peso sufficientemente grande.



Inizialmente si usava una funzione gradino (detta **step**, figura (a)) in cui se il peso dell'input era inferiore ad 1, non veniva considerato. Il problema principale di questa soluzione è che la funzione non risultava derivabile. Si passò successivamente ad una funzione detta **sigmoide** (o **funzione logistica**) che utilizza come soglia la funzione:

$$11+e-x11+e-x$$

.

L'elemento base delle reti neurali è l'output della funzione di attivazione.

Con il termine **reti** si intende un insieme di neuroni che vengono fatti "scattare", cioè si fa transitare l'input da una direzione verso l'altra e si produce l'output.

Con il termine **rete feed forward** si intende un grafo in cui ogni input è dato in pasto ad ogni funzione in base ad un peso. Possono essere:

- Single-layer perception: a singolo livello, cioè c'è un solo livello di nodi dopo gli ingressi
- Multi-layer perception: quando ci sono 2 o più livelli di neuroni, cioè di unità di controllo definite sulla base della funzione sigmoide.

È evidente che una rete neurale con un unico layer (livello) non può rappresentare un vasto insieme di funzioni, infatti, ad esempio, essa è capace di rappresentare le funzioni booleane AND, OR e NOT, ma non è capace di rappresentare la XOR.
Per tale motivazione si cominciò a realizzare reti con più layers (livelli): utilizzando l'uscita di un neurone come ingresso per un altro neurone, appartenente ad uno strato successivo, si può arrivare a realizzare funzioni di attivazione con più "espressività". Richiamando il caso precedente, con una rete neurale a tre livelli si riesce a modellare la XOR.

I valori dei pesi sugli input possono essere automaticamente da un algoritmo in modo che vengano diminuiti gli errori di predizione per la rete neurale. L'algoritmo di chiama **Error-Back-Propagation**, chiamato così perché "porta indietro" l'errore. L'idea è la seguente: inizialmente si assegnano i pesi a caso, si dà un input e si osservano i risultati errati, cioè che sono diversi da quelli attesi, allora l'errore viene portato indietro, cioè:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$

Cioè viene modificato il peso in seguito all'errore e si nota che la modifica avviene piano piano, cioè con piccole modifiche, infatti il fattore α viene chiamato **learning rate**, cioè tasso di apprendimento ed è un valore molto piccolo che va a smorzare l'errore.
Ora bisogna back-propagare l'errore e il nuovo valore sugli archi entranti ai nodi interni. L'equazione per fare ciò è la seguente:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \ .$$

Δ_j sarebbe la componente di errore del nodo j, cioè quanto il nodo j-esimo ha contribuito all'errore visto in uscita. Successivamente si va a fare la correzione dei pesi sui nodi entranti al nodo j, con lo stesso meccanismo visto prima:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

Questo procedimento di ripete per finchè non si arriva al primo nodo, e cos' si ha l'Error-Back-Propagation sul singolo esempio. Chiaramente lo stesso procedimento dovrà essere fatto su ogni esempio del training set. Considerando che questo algoritmo viene eseguito per ogni errore di predizione di ogni esempio, allora si capisce bene il perché c'è bisogno del fattore α, perché ogni esempio ha un contributo minimo sul risultato finale.
Questo algoritmo è molto costoso per quanto riguarda la necessità di potenza di calcolo.
Applicando sempre questo algoritmo, si può arrivare ad un errore molto piccolo, quasi nullo.

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

Una rete neurale prende solo numeri come input. Per un computer, l'immagine non è altro che una griglia di numeri che rappresentano il grado di oscurità di ogni pixel:
Per alimentare un'immagine nella nostra rete neurale, dobbiamo trasformare l'immagine X x Y pixel in una matrice di X*Y numeri
Per gestire N ingressi, sarà sufficiente espandere il Neural Network per avere N nodi di ingresso.
Invece di alimentare intere immagini nella nostra rete neurale andremo a scomporre l'immagine in piccole piastrelle semi-sovrapposte e successivamente inseriremo ogni piastrella all'interno di una piccola rete neurale per determinare le caratteristiche specifiche ricercate all'interno dell'immagine.

Per riconoscere la presenza di un albero si parte dalla creazione di un training set composto da diversi alberi, per poi andare a porre in ingresso un testing set di immagini contenenti degli alberi , per poi confrontare in output i risultate ed eventualmente effettuare un error back propagation.
La nostra rete neurale imparerà a riconoscere gli alberi ed evidenziarli utilizzando particolari filtri .