UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Corso di Intelligenza Artificiale

Prof. Filippo Neri

# Work Projects

Nicola della Volpe

Anno 2019-20

# Work Project 1

Questo primo work project si incentra sulla comprensione empirica dei Decision Trees, esempio di strumento ampiamente diffuso nel Machine Learning, e sul Problem Solving, metodo applicato dall'I.A. per la risoluzione dei problemi.

## 1.1 Esercizio 1 : Modelling with Decision Trees

### 1.1.1 Costruzione di un albero di decisone

Ripercorrendo l'excursus fatto dal libro, viene innanzitutto chiarito il concetto di albero di decisione.
Un albero di decisione è un modello di rappresentazione compatto delle decisioni e delle loro possibili conseguenze, costruito al fine di supportare l'azione decisionale. La sua struttura discende da un insieme di esempi dati e determina le regole di condizione-azione atte alla classificazione di esempi futuri. Dunque l'agente è in grado di apprendere da una serie di dati il comportamento da assumere in situazioni non specificate.
Successivamente viene trattata l'implementate di una serie di funzioni e strutture dati al fine di costruire un albero di decisione.
La funzione principe è :

```
def buildtree(rows, scoref){…}
```

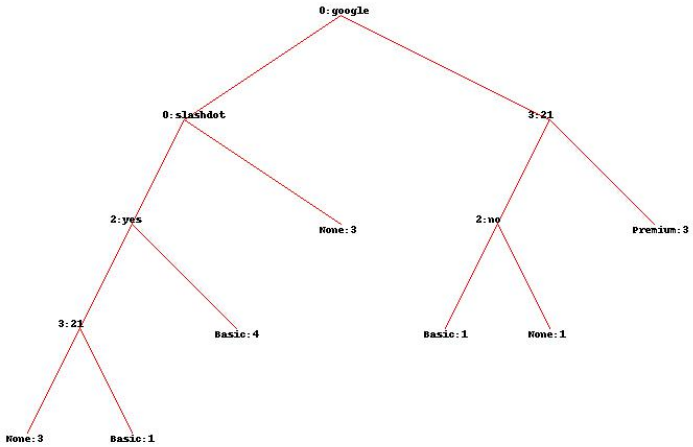
Quest'ultima prende in ingresso un dataset e una funzione di score utile per determinare gli attributi tra quelli del dataset che rendano il più omogeneo possibile il sottoinsieme che andiamo a trovare.
Per `scoref` vengono mostrate le implementazioni della *Gini Impurity* e dell' *Entronpy.*

Nel codice in esame viene analizzato il comportamento di un utente che naviga su un sito web e la sua decisione finale riguardante un acquisto su tale sito.

Un esempio del dataset si presenta nella forma seguente:

| Example | Referrer | Location | FAQ | Pages Viewed | Target |
|---|---|---|---|---|---|
| **E1** | Google | France | Yes | 18 | Premium |

L'esecuzione del codice porta alla costruzione dell'albero di decisione e alla creazione di un immagine dell'albero ( grazie alla funzione di utilità `drawtree(tree, jpeg)` )




La condizione radice è 'Google in colonna 0'. Se verificata, si trova che tutti gli utenti con più di 21 visite alla pagina diventano utenti Premium, altrimenti si passa a verificare 'Slashdot in colonna 0' e cosi via finché non si raggiunge un branch con un risultato.

Infine, viene fornita la funzione `classify(observation, tree)` che accetta una nuova osservazione e la classifica in base all'albero decisionale.

Sulla base dell'albero costruito proviamo a classificare

| | | | |
|---|---|---|---|
| Google | USA | no | 23 |

Otteniamo come risultato che l'utente sceglierà un servizio Premium.
D'altronde seguendo la logica del nostro albero, partendo da nodo radice abbiamo: Google? (True)—> >21? (True)—>Premium.

### 1.1.2 Mushrooms Dataset

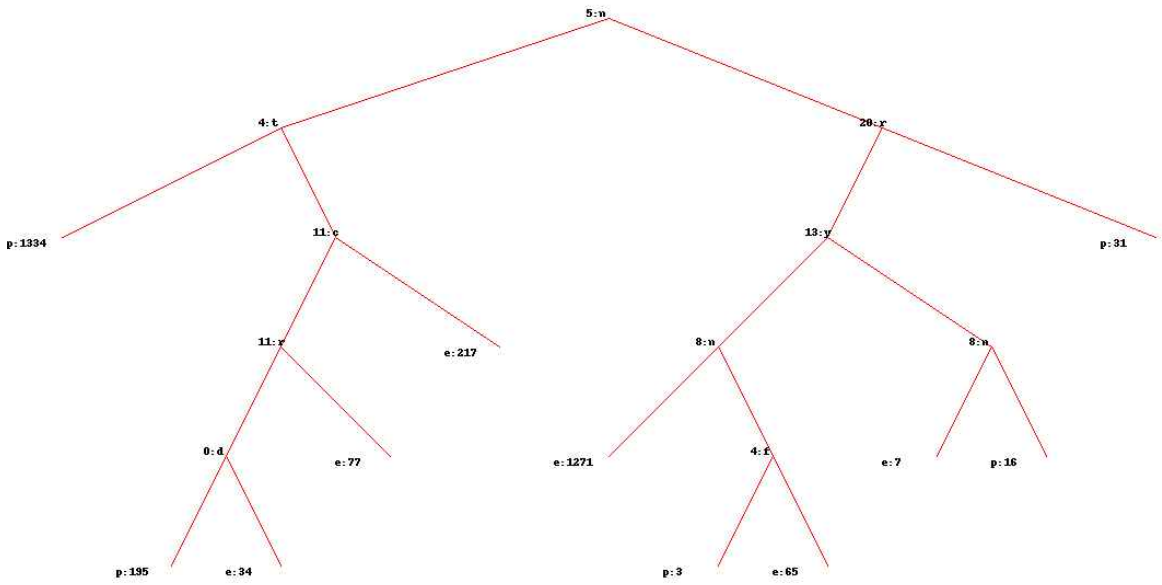
Analizziamo il dataset Mushrooms, il fine è quello di verificare se un fungo è commestibile o velenoso.
Il dataset è caratterizzato da ben 21 attributi e 8124 istanze.
Per il caricamento del dataset si è implementata la funzione

```
def loadDataset(file = 'nomefile.txt'){…}
```

Mentre per la determinazione di training-set e test-set, viene implementata un'altra funzione che seleziona un insieme casuale di esempi dal dataset di partenza con cardinalità determinata dai parametri di ingresso in base alla percentuale desiderata

```
def createDataset(data, numdati){…}
```

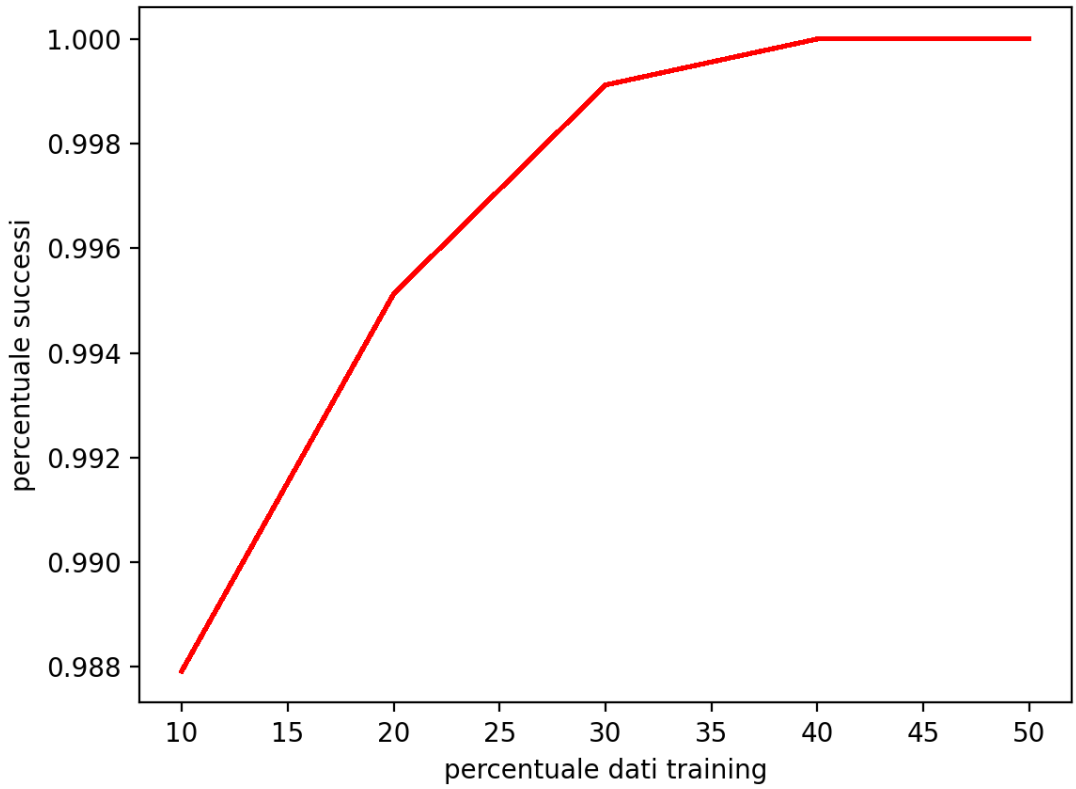
Si decide quindi di usare il 40% dei dati forniti dal dataset come training-set per effettuare l'apprendimento, ed il restante 60 % per il test-set. Ciò al fine di "conservarci" alcuni esempi per misurare le performance dell'albero.
Con le funzioni già prima citate `builtree()` e `drawtree()` otteniamo l'albero di decisione :



Esso, come si può osservare dalla figura, classifica tutti i 3250 esempi del training-set distintamente senza utilizzare tutti gli attributi forniti dal dataset. Si presenta, quindi, in una forma compatta e generalizzante.
Ipotizziamo, allora, che l'albero sia stato costruito facendo uso di attributi rappresentativi con alto guadagno informativo, in grado di effettuare un chiaro split degli esempi sulla base dei valori da essi assunti e quindi con un alto valore predittivo.

### 1.1.3 Learning Curve

Per valutare l'approssimazione dell'ipotesi alla funzione ideale (Problem of Induction) si utilizzano delle misure di performance.
La misura di performance adoperata fa uso di un test-set per stimare l'accuratezza del modello di apprendimento. Essa può essere descritta mediante la Learning Curve, che rappresenta la percentuale di correttezza sul test-set rispetto alle dimensioni del training-set.
Si è scelto, al fine di preservare un test-set consistente, di attribuire al training-set dal 10% fino al 50% dell'intero dataset.
La curva è stata prodotta facendo uso della funzione `fperformance()`, che si avvale, a sua volta, di una funzione `performance()` per il calcolo dell'*accuracy*, e della libreria *matplotlib* per il tracciamento del grafico in questione.
Al seguito di una fase iniziale, prevedente l'inizializzazione delle variabili (test-set massimo) e la determinazione del numero di esempi corrispondete al 10% del dataset, si procede con un ciclo di 5 iterazioni che implementa l'intera logica. Ad ogni ciclo, `createdataset()` drena il 10% del test-set nel training-set.
Segue la creazione dell'albero, il calcolo della sua accuratezza, di cui vengono memorizzati i differenti valori da porre sull'asse delle ordinate, e l'aggiornamento della lista, che memorizza i valori dell'asse delle ascisse. Al termine del ciclo ci si avvale delle funzioni offerte dalla libreria per la rappresentazione grafica della curva
Come si evince dal grafico, la nostra accuratezza migliora fino al 40%, dopodiché rimane costante all'aumentare dei dati forniti al training-set.
Osservando la curva di apprendimento, si evince inoltre che l'aumento delle performance all'aumentare dei dati è minimo e si raggiunge un accuratezza del 100%.




Ciò mostra come tale modello ha un altro grado di predicibilità, nonché come il Decision Tree risulti uno strumento ottimale per la modellazione di tale tipo di problema.

## 1.2 Esercizio 2

### 1.2.1 Agente intelligente

Un agente è un'entità in grado di percepire l'ambiente che lo circonda tramite dei sensori e di eseguire della azioni tramite attuatori.
Un agente è detto intelligente o razionale se, data una sequenza di percezioni, sceglie come azione quella che massimizza il valore atteso della misura di performance.

Un agente razionale è un agente in grado di esplorare, apprendere e prendere decisioni in maniera autonoma.
Ciò implica che l'apprendimento di un un DT (Decision Tree) rende l'agente a tutti gli effetti intelligente sin quanto ha appreso l'albero e attraverso questo è in grado di prendere delle decisioni al fine di risolvere un problema.

### 1.2.2 DT Learning

Per DT learning si intende la costruzione di un albero di decisione (cosa diversa dalla sua rappresentazione).

Il DT Learning può essere espresso tramite una funzione ricorsiva siffatta:

```
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
```

Tale funzione prende in ingresso esempi, attributi e una classificazione di default e restituisce un albero di decisione. In particolare:

- Se l'insieme degli esempi è vuoto, allora restituisce la classificazione di default.
- Se tutti gli esempi hanno la stessa classificazione, allora restituisce la classificazione stessa.

- Se l'insieme degli attributi è vuoto, allora restituisce la moda degli esempi, che conta la classe degli esempi e ritorna la classe che occorre maggiormente.
- Negli altri casi, sceglie un attributo, crea un nuovo albero di decisone avente come radice l'attributo scelto e per ogni valore che l'attributo può assumere riduce l'insieme degli esempi, creando un sottoinsieme che contenga solo gli esempi per cui l'attributo vale l' i-esimo valore. In questo modo sto partizionando l'insieme degli esempi sulla base del valore assunto dall'attributo. Si crea un sottoalbero, passando alla funzione DTL, chiamata per ricorsione, il sottoinsieme degli esempi precedentemente determinato, l'insieme degli attributi eccetto quello scelto, e la moda degli esempi come classificazione di default. Infine il sottoalbero viene collegato all'albero principale.

Ci chiediamo ora in base a cosa viene scelto l'attributo nodo dell'albero.
Idealmente un buon attributo splitta gli esempi in sottoinsiemi interamente omogenei.
Per il calcolo effettivo della sua bontà si può utilizzare una funzione quale l'entropia.
In particolare preso un attributo, per ogni valore che quest'ultimo può assumere, si creano diversi sottoinsiemi ognuno dei quali contiene solo gli esempi per cui l'attributo vale l'iesino valore.
L'entropia associata all'attributo è data dalla somma delle entropie dei diversi sottoinsiemi opportunamente pesate.

## 1.3 Esercizio 3: A*

### 1.3.1 Descrizione ed euristica ammissibile

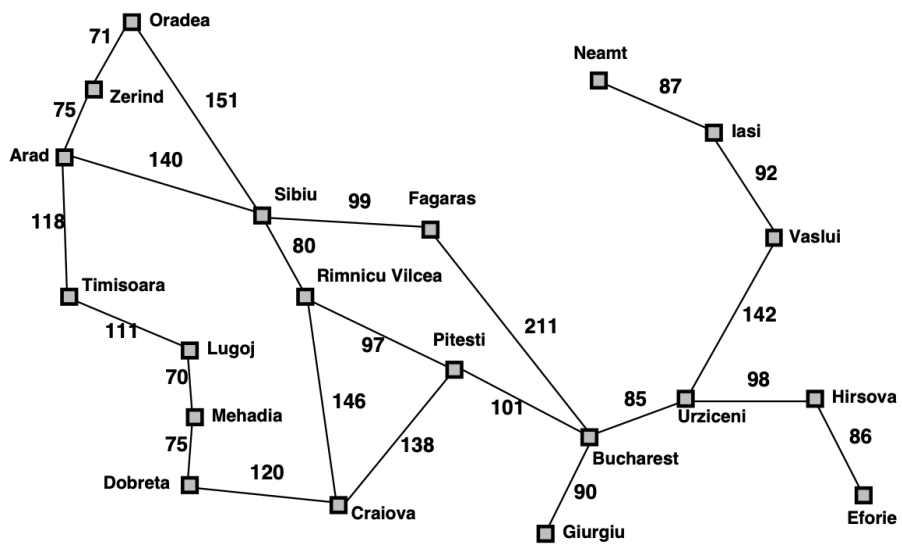
A* è un algoritmo di ricerca  informata.

Esso sceglie i nodi da espandere utilizzando come criterio di desiderabilità la funzione $f(n) = g(n) + h(n)$.

*g(n)* è il costo passato per raggiungere *n*, mentre *h(n)* è la stima di quanto lo stato sia distante dallo stato obiettivo.
*g(n)* è certa , *h(n)* è basata su un'euristica.
Un'euristica si dice ammissibile se il valore stimato da essa è minore o uguale al costo reale per raggiungere l'obiettivo a partire dal nodo che stiamo considerando.
Se *h(n)* è ammissibile in A*, quest'ultimo mi consente, nel caso esistesse, di trovare la soluzione ottimale al problema, ossia quella che ha path-cost minore.

## 1.3.2 Navigazione stradale

Nel caso ad esempio di una navigazione stradale, consideriamo la cartina della Romania e di voler partire dalla città di Arad ed arrivare a Bucharest.



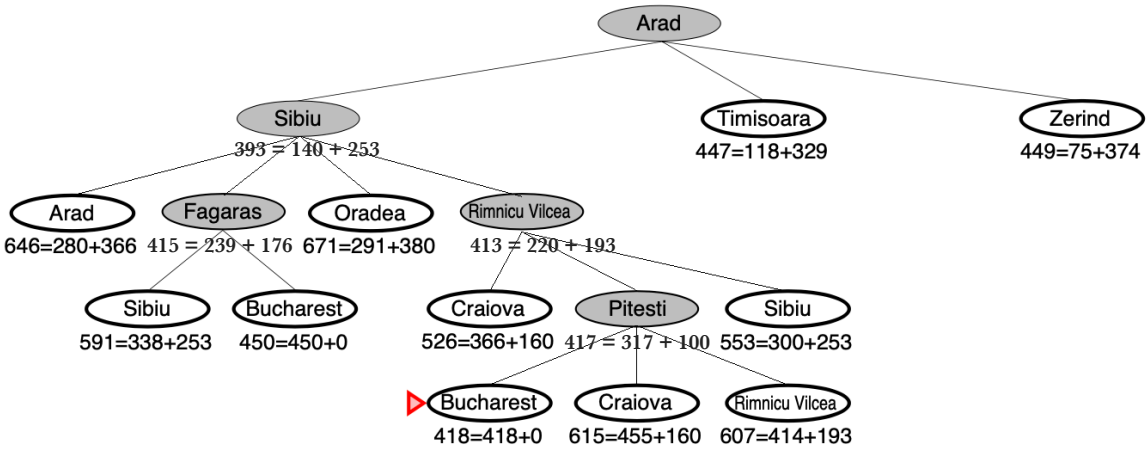

| Straight–line distance to Bucharest | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 178 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234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

Utilizzo come euristica *h* la distanza in linea retta tra le varie città, mentre *g* è la distanza stradale.
Secondo A* espando i nodi con $f = g + h$ minore.
Espandendo la radice Arad, trovo come nodi Sibiu, Timisoara e Zerind.
Calcolo $f$ per tutti i nodi, e trovo che:

- Sibiu ha una $f$ = 140(distanza stradale da Arad) + 253(distanza in linea retta da Bucharest) = 393
- Timisoara ha una $f$ = 118(distanza stradale da Arad) + 329(distanza in linea retta da Bucharest) = 447

- Zerind ha una $f$ = 75(distanza stradale da Arad) + 374(distanza in linea retta da Bucharest) =449

Quindi per A* espando Sibiu che ha *f* minore di tutte.
Espandendo Sibiu trovo allo stesso modo le f per i nodi Arad, Fagaras, Oradea e Rimnicu Vilcea.
Quest'ultima ha *f* minore di tutte pari a 413, la espando e trovo che i suoi nodi figli hanno *f* tutte maggiori della *f* di Fagaras. Vado quindi ad espandere tale nodo.
Confrontato le *f* ricavate, vado quindi ad espandere Pitesti e finalmente arrivo a Bucharest.




## 1.5 Esercizio 5: Tree Search

### 1.5.1 Problemi ben definiti e soluzioni

Un problema può essere definito formalmente da cinque componenti:

- Lo stato iniziale in cui si trova l'agente.
- Una descrizione delle possibili azioni disponibili per l'agente.

- Una descrizione di ciò che ogni azione fa, formalmente detto *modello di transizione*. Usiamo anche il termine funzione successore per fare riferimento a qualsiasi stato raggiungibile a partire da un dato stato con una singola azione.

Insieme, lo stato iniziale, le azioni e il modello di transizione definiscono implicitamente lo *spazio degli stati* del problema: l'insieme di tutti gli stati raggiungibili dallo stato iniziale a partire da una qualsiasi sequenza di azioni. Lo spazio degli stati forma una rete o un *grafo* diretto in cui i nodi sono gli stati e i collegamenti tra i nodi sono le azioni. (La mappa della Romania mostrata sopra può essere interpretata come un grafo degli stati).

Un percorso nello spazio degli stati è una sequenza di stati collegati da una sequenza di azioni.

- Il test obiettivo, che determina se un determinato stato è uno stato obiettivo. A volte esiste un insieme esplicito di possibili stati obiettivo e il test controlla semplicemente se lo stato dato è uno di questi.
- Una funzione di costo del percorso che assegna un costo numerico a ciascun percorso. L'agente di risoluzione dei problemi sceglie una funzione di costo che riflette la propria misura delle prestazioni.

Gli elementi precedenti definiscono un problema e possono essere raccolti in una singola struttura di dati che viene fornita come input per un algoritmo di risoluzione dei problemi. Una soluzione a un problema è una sequenza di azioni che porta dallo stato iniziale a uno stato obiettivo. La qualità della soluzione viene misurata dalla funzione costo percorso e una soluzione ottimale ha il costo percorso più basso tra tutte le soluzioni.

### 1.5.2 Tree Search

Dopo aver formulato alcuni problemi, ora dobbiamo risolverli. Una soluzione è una sequenza di azioni, quindi gli algoritmi di ricerca funzionano considerando varie possibili sequenze di azioni. Le possibili sequenze di azioni che partono dallo stato iniziale formano un albero di ricerca con lo stato iniziale alla radice; i rami sono azioni e i nodi corrispondono agli stati nello spazio degli stati del problema.

L'algoritmo generale per il *Tree-Search* è il seguente:

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

Il nodo radice dell'albero corrisponde allo stato iniziale.
Quindi dobbiamo considerare di intraprendere varie azioni.
Lo facciamo espandendo lo stato attuale; vale a dire, applicare ogni azione legale allo stato attuale, generando così un nuovo insieme di stati.
Aggiungiamo dei rami dal nodo padre che porta alla creazione di un pari numero di nodi figlio.
Ora dobbiamo scegliere quale di questi espandere.
L'essenza della ricerca è proprio quella di seguire inizialmente una scelta e mettere le altre da parte nel caso in cui la prima non porti a una soluzione.
Ogni nodo senza figli è detto nodo foglia. L'insieme di tutti i nodi foglia disponibili per l'espansione in un determinato punto è chiamato frontiera.
Il processo di espansione dei nodi alla frontiera continua fino a quando non viene trovata una soluzione o non ci sono più stati da espandere
Gli algoritmi di ricerca condividono tutti questa struttura di base; variano principalmente in base al modo in cui scelgono quale stato espandere, la cosiddetta *strategia di ricerca*.

### 1.5.3 Uninformed search strategies

Dette anche strategie di ricerca "alla cieca", il termine sta ad indicare il fatto che tali strategie non prevedono ulteriori informazioni sugli stati oltre a quelle fornite nella definizione del problema.
Tutto quello che possono fare è generare successori e distinguere uno stato obiettivo da uno stato non obiettivo.
Tutte le strategie di ricerca si distinguono per l'ordine in cui i nodi vengono espansi.

### Breadth-first search

E’ una strategia per la quale tutti i nodi vengono espansi a una determinata profondità nella struttura di ricerca prima che vengano espansi tutti i nodi al livello successivo.
Tale strategia è completa in quanto se esiste una soluzione la trova perché vengono esplorati tutti i percorsi possibili.
Il problema principale è la dimensione dell’albero: c’è una complessità in termini di spazio che cresce in modo esponenziale.

### Deep-first search

Tale strategia espande sempre il nodo più profondo nella frontiera corrente dell'albero di ricerca.
Non è completo, fallisce in spazi di profondità infinita, ossia in spazi con dei cicli.
Non è ottimale, perché non trova la soluzione migliore, ossia quella a profondità minima, bensì la prima.

### Graph-first search

Finora, abbiamo trascurato la possibilità di “perdere tempo” ad espandere stati che abbiamo già incontrato ed espanso prima in qualche altro cammino. Si determinano quindi cicli sulla mappa.
Per ovviare a questo problema, si introduce dal punto di vista astratto una variabile che memorizza gli stati già visitati.
In particolare, se ci troviamo in uno stato già visitato in passato non lo reinseriamo nella coda(caso Breadth-first) o nello stack(caso Deep-first).
L’algoritmo di ricerca su un albero nel momento in cui introduciamo questa gestione si dice *Graph-search*.

## 1.5.4 Informed (Heuristic) search strategies

Le strategie di ricerca informata utilizzano una conoscenza specifica del problema oltre la definizione del problema stesso.
L'approccio generale che consideriamo è chiamato *Best-first search*.

Best-first search è un'istanza dell'algoritmo generale tree-search o graph-search in cui un nodo è selezionato per l'espansione in base a una funzione di valutazione, *f(n)*. La funzione di valutazione viene interpretata come una stima dei costi, quindi il nodo con la valutazione più bassa viene espanso per primo.
La scelta di *f* determina la strategia di ricerca. La maggior parte dei migliori algoritmi include come componente di *f*, un'euristica, indicata con *h (n)*:
*h(n)* = costo stimato del percorso più economico dallo stato al nodo n a uno stato obiettivo.
Le funzioni euristiche sono la forma più comune in cui viene impartita una conoscenza aggiuntiva del problema all'algoritmo di ricerca.
Se n è un nodo obiettivo, *h(n)* = 0.

## Greedy best-first search

Greedy best-first search tenta di espandere il nodo più vicino all'obiettivo, poiché è probabile che ciò porti rapidamente a una soluzione. Pertanto, valuta i nodi usando solo la funzione euristica, cioè *f(n) = h(n)*.
La soluzione che trovo con questo approccio non è detto sia la soluzione ottimale, tuttavia ha come vantaggio quello di trovare in maniera rapida una soluzione sub ottima.

## A* search

Descritto sopra (pagina 8-9).

# Work Project 2

Il secondo work project si basa sugli algoritmi di ricerca locale, i problemi di ricerca con vincoli e gli aspetti principali della logica proposizionale.

## 2.1 Esercizio 1 : Optimization

### 2.1.1 Local search algorithms

Il capitolo del libro introduce le tecniche di ottimizzazione stocastica, le quali sono in genere utilizzate in problemi che hanno molte soluzioni possibili in molte variabili e che producono risultati che possono cambiare notevolmente a seconda delle combinazioni di queste ultime.
L'ottimizzazione cerca di trovare la soluzione migliore a un problema provando molte soluzioni diverse e assegnando loro un punteggio per determinarne la qualità.

Viene mostrato un esempio riguardate la pianificazione di un viaggio per un gruppo di persone (la famiglia Glass in questo caso) che partono da diverse località e che vogliono arrivare tutte nello stesso posto.
La "sfida" sta nello stabilire quale volo dovrebbe prendere ogni persona della famiglia.
Un obiettivo è ovviamente mantenere basso il prezzo totale, ma ci sono molti altri possibili fattori che la soluzione ottimale prenderà in considerazione e proverà a minimizzare, come ad esempio il tempo di attesa totale in aeroporto o il tempo totale di volo.
Poiché le funzioni di ottimizzazione sono abbastanza generiche da essere riutilizzate in molti tipi diversi di problemi, è importante scegliere una rappresentazione semplice e generale.
La rappresentazione che viene utilizzata in questo caso è quelle di una lista di numeri.

Viene poi analizzata la funzione di costo (`def schedulecost(sol)`), che è la chiave per la risoluzione dei problemi di ottimizzazione ed è solitamente la cosa più difficile da determinare.
L'obiettivo di qualsiasi algoritmo di ottimizzazione è trovare un insieme di input - i voli, in questo caso - che riducano al minimo tale funzione.
I fattori che possono essere presi in considerazione e che possono essere misurati nell'esempio di un viaggio di gruppo sono molteplici, si può pensare al prezzo, al tempo di viaggio, al tempo di attesa, all'orario di partenza, al periodo di noleggio dell'auto etc…
La funzione di costo scelta in questo esempio tiene conto del costo totale del viaggio e del tempo totale trascorso in attesa negli aeroporti per i vari membri della famiglia.
Viene dato poi un peso a tali fattori e combinati in un unico numero.
L'obiettivo è ora ridurre al minimo i costi scegliendo l'insieme di numeri corretto. Testare ogni combinazione(in questo caso 9^16 combinazioni possibili) garantirebbe la risposta migliore, ma ciò richiederebbe troppo tempo per maggior parte dei calcolatori.
Vengono quindi utilizzati algoritmi di ricerca locale, i quali si basano sul principio dei miglioramenti successivi.
Si cerca in primo luogo una soluzione, anche non ottima, ed in seguito ci si concentra sull'ottimizzazione.
Spesso si riesce a determinare solo un massimo locale, ottenendo risultati più o meno soddisfacenti a seconda dell'applicazione.

Sono quindi implementati e testati diversi algoritmi per la risoluzione del problema in esempio.

## Random Searching

Questo algoritmo prevede semplicemente la determinazione casuale di un insieme di soluzioni e la comparazione dei relativi costi per l'identificazione di quella a costo minimo.
Effettuando diversi esperimenti variando il numero di soluzioni estratte casualmente si nota che: ad un eccessivo aumento delle soluzione estratte non si ottengono necessariamente risultati nettamente migliori da giustificare l'evidente incremento della complessità computazionale in termini temporali; d'altro canto, un numero esiguo di soluzioni produce risultati dalla qualità incostante.

*Costo medio riscontrato prevedendo 1000 soluzioni estratte : c.a 3328*

## Hill Climbing

L'Hill Climbing inizialmente determina una soluzione ed i suoi "vicini", se uno di essi presenta un costo migliore della soluzione corrente, verrà scelto questo come soluzione corrente alla prossima iterazione, attuando in tal modo il processo dei miglioramenti successivi; se la soluzione corrente ha costo minore dei suoi vicini, termina la serie di miglioramenti iterativi poiché si è in presenza di un massimo locale (minimo locale rispetto al costo a seconda della formulazione del problema).
Testandolo si possono notare risultati decisamente migliori del Random Searching, che sottolineano la maggior efficienza dell'Hill Climbing. La variabilità nella qualità della soluzione, al seguito di svariate esecuzioni, è dipendente dai massimi locali del problema specifico: partendo da situazioni iniziali diverse, arriviamo a risultati diversi (Random-Restart Hill Climbing) .
*Costo medio riscontrato : c.a 3063*

## Simulated Annealing

Versione rivisitata dell'Hill Climbing, questo metodo di ottimizzazione è ispirato dal riscaldamento termico di una lega e al suo successivo raffreddamento. Viene comparata la soluzione corrente ad una determinata casualmente (nel nostro caso tramite un cambiamento di quella corrente): l'algoritmo tende a spostarsi sempre verso una soluzione successiva migliore, solo con una certa probabilità verso quelle peggiori, che tenderà a diminuire col tempo. Questa probabilità dipende dalla qualità della soluzione successiva: mosse pessime avranno minore probabilità. Viene, inoltre, introdotto un concetto di "temperatura": essa determina la probabilità di spostarsi verso soluzioni non ottime. Inizialmente questo valore sarà alto (temperatura elevata), in seguito essa tenderà a calare (raffreddamento), consentendo, in ultima battuta, la scelta della soluzione ottimizzata. Tale probabilità è data dalla seguente espressione:
*e^−(costo successivo+costo attuale)/temperatura*
Osserviamo che con l'abbassamento della temperatura, ad ogni iterazione dell'algoritmo, tale probabilità decresce. Le ragioni alla base di questa logica risiedono nella maggiore possibilità di trovare un massimo globale, evitando che ci si possa assestare su un punto di massimo locale.
Dalle prove sperimentali si sono constatati risultati migliori o uguali all'Hill

Climbing, appurando la maggiore capacità di ottenere soluzioni con costo inferiore.
*Costo medio riscontrato: c.a 2278*

### Genetic Algorithms

Ulteriore algoritmo che tenta di superare i limiti dell'Hill Climbing è quello genetico. Esso è ispirato dalla natura, proponendo un'evoluzione delle soluzioni tipicamente darwiniana. L'algoritmo prende le mosse selezionando un insieme casuale di soluzioni, detto popolazione. Gli esemplari di questa vengono poi valutati ed ordinati attraverso una funzione di fitness (rappresentata nel nostro caso dal costo minimo). Nell'implementazione utilizzata, si procede alla creazione della successiva generazione a partire da un prestabilito numero di migliori soluzioni. Attraverso il processo di elitarismo quelle ottime sopravviveranno nella nuova generazione.
Difatti quest'ultima sarà costruita attraverso il crossover (combinazione di due soluzioni) dell'élite e attraverso la mutazione puntiforme (cambiamento casuale di alcune cifre al fine di evitare convergenza verso un unico picco).
Vari esperimenti di hanno portato a riscontrare risultati simili al Simulated Annealing, se non migliori in taluni casi, che presentano, però, una minore variabilità con misure ripetute.
*Costo medio riscontrato: c.a 2183.*

## 2.1.1 Ottimizzazione di una funzione

Vogliamo provare ad ottimizzare la funzione F(x), così definita:

$$F(x) = \begin{cases} 10 & \text{se } x < 5.2 \\ x^2 & \text{se } 5.2 \leq x \leq 20 \\ cos(x) + 160x & \text{se } x > 20 \end{cases} ; x \in [-100,100]$$

La ricerca del "valore di ottimizzazione" verrà effettuata tramite algoritmo genetico. Prima di tutto, però, dobbiamo definire la *funzione di costo*, così da poter determinare quando la nostra soluzione tende a migliorare o, viceversa, a peggiorare.

Definisco una funzione `costmax(sol)` tale che se il valore di cui si calcola il costo non è accettabile, poiché nel passaggio tra una generazione ad un'altra si è avuto un esemplare della popolazione non voluto, cioè non ammissibile per la soluzione (valore al di fuori dell'intervallo [−100, 100] ), gli si dà un costo pari a 0, in modo tale che per la creazione della prossima generazione non verrà preso in considerazione; altrimenti il costo è dato dall'applicazione della F sul valore in ingresso.
L' algoritmo prima di tutto genera una popolazione iniziale prendendo a caso dei valori all'interno del dominio, ne calcola i vari score e sceglie una élite di individui, quelli che in questo caso massimizzano lo score.
Successivamente, effettuiamo il crossover tra coppie di individui appartenenti all'élite e applichiamo ad essi un operatore di mutazione che, in base ad una probabilità, opera dei cambiamenti casuali sui "figli" nati dal crossover.
Per effettuare le due operazioni sopracitate, i numeri passati all'algoritmo vengono rappresentati in forma binaria.
Il crossover quindi è eseguito scegliendo un numero casuale di bit da una soluzione e il restante numero di bit da un'altra, mentre la mutazione risulta nel cambiamento di un bit da 1 a 0, o viceversa.
Tali processi sono eseguiti per un numero di generazioni volute, così facendo, ad ogni cambio generazionale, la progenie che si crea è combinazione di geni dei genitori migliori della generazione precedente, attuando in tal modo una selezione naturale.
La mutazione, invece, permette di introdurre nuovi tratti negli individui, donandogli in tal modo caratteristiche distintive e consentendo alla specie di attuare un processo evolutivo.
Eseguendo il codice risulta che il valore che ottimizza la F(x) tende a x=100. Partendo da una popolazione iniziale relativamente ristretta(circa 100 elementi), per un numero di iterazioni discreto (pari a 100 in questo caso) la soluzione non solo viene trovata, ma viene confermata più volte, dimostrando che, anche se si provano nuove combinazioni, l'esemplare che massimizza la fitness è stato già trovato precedentemente e le iterazioni successive non fanno altro che confermarlo.

## 2.2 Esercizio 2 : Constraint Satisfaction Problems

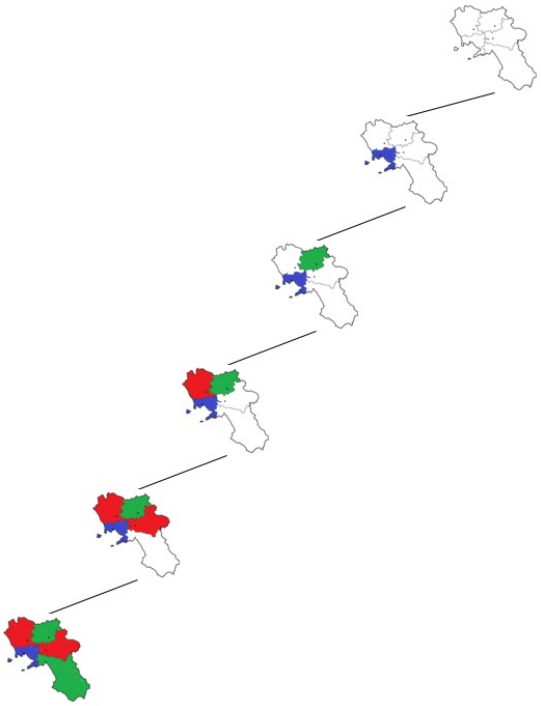
Il problema posto è il seguente:
"Un agente intelligente deve colorare le province della regione Campania evitando che le confinanti abbiano lo stesso colore".
Tale problema è tipico di quelli risolvibili con algoritmi che tengono conto del soddisfacimento dei vincoli.
Come prima cosa formalizziamolo :
Variabili: *Caserta , Benevento , Napoli , Avellino , Salerno*
Vincoli: province confinanti devono avere colori diversi
Possiamo, dunque, ricavare il grafo dei vincoli, i cui nodi sono le variabili ed i cui archi connettono coppie di attributi che partecipano ad un vincolo.


Analizziamo il caso in cui l'agente disponga di solo *2 colori*.
Dominio = {Colore1 , Colore2}
Da uno sguardo attento del grafo possiamo formalizzare il vincolo per cui le variabili Napoli, Benevento e Avellino devono essere necessariamente tutte diverse (non sono l'unica tripla soggetta a questo vincolo).
Un dominio formato da solo due colori non ci permetterà mai di poter soddisfare questo vincolo, pertanto il problema non è risolvibile.
Ma, oltre a questa formulazione informale, si perviene alla stessa conclusione anche applicando l'algoritmo di *Backtracking Search*.

Questo prevede l'assegnazione dei valori alle variabili senza preoccuparsi dei vincoli, dopo l'assegnazione, nel caso di vincolo non soddisfatto, si torna indietro di un passo(da ciò Backtracking) e si contrassegna il valore dato all'ultima variabile come "da non usare". L'algoritmo è ricorsivo e depth-first.
Andiamolo quindi ad applicare al problema in esame e sfruttiamo anche le euristiche per migliorare l'efficienza della ricerca.
Assegniamo a Napoli (*Degree Heuristic*: variabile con più vincoli sulle rimanenti variabili) il valore Colore1; se procediamo assegnando Colore2 a Caserta (o qualsiasi altra variabile dato che ora presentano tutte un solo valore ammissibile), si ottiene un fallimento, poiché tramite il *Forward Checking* si osserva che Benevento non può assumere nessun valore ammissibile. Questa risultato è comune anche agli altri path dell'albero di ricerca e tutto ciò è intuitivamente comprensibile poiché con soli 2 colori non siamo in grado di soddisfare i loop di 3 nodi presenti nel grafo dei vincoli.

Analizziamo il caso in cui l'agente disponga di *3 colori*.
Dominio = {Colore1 , Colore2 , Colore3}
Partiamo da Napoli assegnandole il Colore1 e notiamo che questa scelta riduce i valori ammissibili per le altre province confinanti a due. Coloriamo, ora, Benevento con il Colore2; di conseguenza Caserta sarà la nostra prossima scelta poiché le si può assegnare solo un colore, il Colore3. Tale scelta è dettata dall'euristica *Minimum Remaining Values*, che prevede la scelta della variabile con numero minimo di valori possibili. Ad Avellino, non confinante con Caserta, le possiamo assegnare il Colore3 senza produrre nessun conflitto. A Salerno, infine, assegniamo il Colore2. Osserviamo che ogni provincia ha un colore diverso dalle confinanti.
Possiamo, in conclusione, affermare che è possibile soddisfare i vincoli dato questo dominio.

Anche il caso con *quattro colori* sicuramente permetterà di soddisfare i vincoli, perché si è aumentato il numero di valori ammissibili lasciando invariate le informazioni iniziali. Verifichiamo che sia così.
Dominio = {Colore1 , Colore2 , Colore3 , Colore4}
La prima provincia che coloriamo è Napoli, assegnandole il Colore1: così facendo riduciamo tutti i colori possibili per le altre province, avendo Napoli il maggior numero di vincoli. Avellino sarà la nostra prossima scelta: assegnandole



il Colore2 diminuirà anche il numero di valori per Salerno e Benevento. Passiamo a Caserta: essa può assumere tutti i valori tranne Colore1. Osserviamo, però, che l'assegnazione dei valori Colore3 e Colore4 produrrebbe una diminuzione del dominio degli attributi che possiamo assegnare a Benevento. Applichiamo il *Least Constraining Value*, euristica che predilige il valore che lascia più libertà alle variabili adiacenti, e scegliamo il Colore2 per Caserta. Segue Benevento con il Colore3(o 4) e Salerno con il Colore3(o 4).

Come ci si poteva aspettare, l'aggiunta di un nuovo elemento nel Dominio, senza l'alterazione dei vincoli, permette ugualmente la risoluzione del problema. In quest'ultimo caso però, si può parlare di vincoli più rilassati, grazie ai maggiori gradi di libertà per la determinazione della soluzione (con un Dominio di tre elementi il numero di soluzioni possibili sicuramente sarà molto più limitato rispetto ad uno con quattro elementi, che permette più opportunità di scelta).

Possiamo, quindi, affermare che le soluzioni relative al Dominio esteso includono quelle del compatto.

## 2.3 Esercizio 3 : Colorazione mappa Campania con Genetic Algorithm

Si cerca in questo esercizio un possibile approccio per trovare la soluzione con l' ausilio di algoritmi genetici al problema vincolato sopracitato con Dominio = {Colore1 , Colore2 , Colore3 , Colore4}.
Prima di utilizzare gli algoritmi genetici per risolvere il problema, dobbiamo definire una funzione di costo. Tra le possibili scelte, si opta per la seguente:
*costo = numero di conflitti*
dove il conflitto è inteso come l'evento in cui due provincie confinanti hanno lo stesso colore.
Una soluzione sarà quindi valutata rispetto ad una funzione di fitness che prende in considerazione il numero di conflitti: quanto più una soluzione presenterà un costo minore, tanto più assumerà un valore di fitness maggiore. Ovviamente ottimizzare la soluzione significa minimizzare il valore assunto dalla funzione di costo. Poiché essa può assumere solo valori positivi, la migliore soluzione riscontrabile avrà costo nullo.
La popolazione iniziale sarà costituita da un insieme di mappe che presentano una colorazione casuale delle province.
Definiamo l'élite come l'insieme di mappe aventi il miglior valore di fitness.
Effettuiamo il crossover prendendo dall'élite due elementi (mappe colorate) e uniamo un certo numero di province di uno con le restanti dell'altro.
Definiamo poi una probabilità di mutazione che consente di variare casualmente colore ad una provincia tra i tre rimanenti (escludendo il colore attualmente assegnatole).
Questi processi consentono la creazione della nuova generazione, dopodiché il tutto si ripeterà ciclicamente fino a che non si arrivi alla soluzione ottimale, o ad una sub-ottima nel caso il problema sia troppo complesso.
Potrebbe, infatti, capitare che tutti gli elementi della popolazione convergano verso una soluzione che rappresenti solo un punto di massimo locale, e non globale, della funzione di fitness, quindi il risultato che potremmo ottenere utilizzando questo algoritmo non è detto sia ottimo e, conseguentemente, potrebbe non rappresentare una soluzione per il problema vincolato. Ciò non dovrebbe necessariamente sorprenderci considerando che il problema vincolato

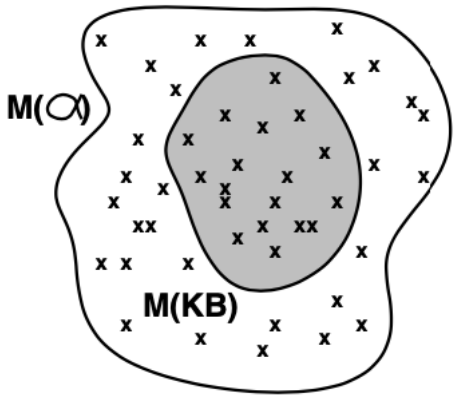
potrebbe anche non avere del tutto soluzione (come nel caso del problema della mappa con 2 colori).
Considerando il tipo di problema potremmo effettuare varie modifiche.
Una prima modifica potrebbe consistere nell'impedire all'algoritmo di terminare fin quando non si sia giunti alla soluzione con costo pari a zero (soluzione del problema vincolato). Ovviamente tale modifica potrebbe fare in modo che il nostro algoritmo non termini mai! Per ovviare a questo inconveniente, potremmo invece pensare di introdurre una modifica che faccia ripartire l'algoritmo se non si sia giunti alla soluzione dopo un certo numero di iterazioni.

## 2.4 Esercizio 4 : Logica Proposizionale

La logica proposizionale è un linguaggio formale utilizzato dagli agenti logici per la rappresentazione della conoscenza. Un agente intelligente che sfrutta tale approccio si ispira al processo umano di deduzione a partire da avvenimenti noti che ha precedentemente appreso.
Quindi l'agente avrà bisogno di una base di conoscenza (*KB: knowledge base*), in cui archiviare le informazioni utilizzate nella deduzione, e di un meccanismo di inferenza (inference engine) per effettuare le deduzioni.
L'agente ogni volta che innesca il processo deduttivo, enuncia alla base di conoscenza la percezione corrente, chiede ad essa l'azione da eseguire ed in base alle informazioni archiviate compie una scelta.
Comunica infine di aver eseguito l'operazione, così da garantire che l'evento scatenante il processo di ragionamento possa ampliare la base di conoscenza.
Inizialmente sarà necessario una fase di raccolta delle informazioni.
Possiamo fornire questa conoscenza al nostro agente in due modi: mediante un approccio dichiarativo vi è proprio una fase di raccolta di informazioni da parte dell'agente; nel caso di un approccio procedurale invece possiamo fornire noi una conoscenza iniziale all'agente. Solitamente i due approcci vengono usati insieme.
Le informazioni devono essere archiviate nella base di conoscenza dell'agente in un linguaggio tale da poter trarre delle conclusioni.
Per tale motivo le frasi, o formule, che costituiscono il KB (Knowledge base) , sono espresse in un linguaggio formale di rappresentazione della conoscenza.

Esso determina per le formule una sintassi per esprimerle in maniera corretta ed una semantica che ne definisce la verità rispetto ad ogni mondo possibile.
I mondi possibili sono gli ambienti, le condizioni in cui l'agente potrebbe venirsi a trovare.
Mondi strutturati formalmente, rispetto ai quali è possibile valutare la verità delle frasi, sono più propriamente detti modelli.
Diciamo che m è un modello di una frase *a* se *a* è vera in *m*.
Indichiamo, inoltre, come M(*a*) l'insieme di tutti i modelli di *a*.
L'agente, in verità non comprende cosa sta manipolando, ma può produrre nuove frasi e usarle per prendere decisioni.
E' invece l'essere umano a dare una semantica ad una manipolazione che è puramente sintattica e basata sulla rappresentazione.

Determinata la base di conoscenza, da essa si possono riuscire a dedurre ulteriori informazioni attraverso la relazione di *conseguenza logica* (*entailment*):
KB$\boxed{?}$ implica(entails) la frase *a* $\Longleftrightarrow$ *a* è vero in tutti i mondi in cui KB è vero, sintatticamente si scrive KB $\models a$,
nel linguaggi insiemistico KB$\boxed{?}$ implica $\alpha \Longleftrightarrow M(KB) \subseteq M(\alpha)$.




La conseguenza logica può essere applicata per derivare conclusioni, ovvero per eseguire *inferenze logiche.*
Con la notazione KB $\vdash_i \alpha$ si intende che $\alpha$ è derivato da KB attraverso l'algoritmo di inferenza i.
Una procedura di inferenza si dice consistente (*soundness*) se deriva solo formule che sono conseguenze logiche (KB $\vdash_i a \Rightarrow$ KB $\boxed{?} \models a$), completa (*completeness*) se può derivare ogni formula che è conseguenza logica (KB $\models \boxed{?} a \Rightarrow$ KB $\vdash_i a$).

La logica proposizionale è una logica semplice in cui la sintassi è composta da frasi atomiche, o simboli proposizionali, legate tra loro tramite connettivi logici per ottenere formule più complesse. Un simbolo proposizionale può assumere il valore vero o falso e le regole per determinare il valore di verità di una formula, rispetto ad un particolare modello, sono definite dalla semantica:

- $\neg P$ è vero se e solo se P è falso
- $P \wedge Q$ è vero se e solo se P e Q sono entrambi veri
- $P \vee Q$ è vero se e solo se P o Q è vero
- $P \Rightarrow Q$ è falso se e solo se P è vero e Q è falso
- $P \Leftrightarrow Q$ è vero se e solo se P e Q sono entrambi veri o entrambi falsi

Queste regole possono anche essere espresse con le tabelle di verità, che esprimono i valori di verità di una frase complessa per ogni possibile configurazione dei suoi simboli proposizionali.

Come procedura di inferenza possiamo ricorrere ad un approccio per enumerazione dei modelli (*model checking*): enunciamo tutti i possibili modelli, tramite la tabella di verità, e verifichiamo che α sia vera in ogni modello in cui KB è vera. I modelli sono rappresentati da un assegnamento di valori vero o falso ad ogni simbolo proposizionale. Questo algoritmo è consistente e completo, ma è molto oneroso in quanto i possibili mondi sono tanti (crescono esponenzialmente) ed enumerarli tutti comporterebbe una notevole complessità spaziale e temporale.

Un approccio differente è strettamente legato al concetto di soddisfacibilità. Una frase è soddisfacibile se è vera in qualche modello, viceversa insoddisfacibile se non è vera in nessun modello. La soddisfacibilità è connessa all'inferenza dalla:

$KB \boxed{?} \models \alpha \Longleftrightarrow KB \wedge \neg\alpha$ è *insoddisfacibile*

La procedura inferenziale tenterà, dunque, di provare *a* attraverso una *reductio ad absurdum*, ovvero dimostrando che $KB \wedge \neg a$ è falsa in ogni modello.

L'algoritmo di risoluzione che prova *a* per assurdo necessita che tutte le frasi siano espresse nella forma CNF (*Conjunctive Normal Form*), cioè come congiunzioni di clausole (disgiunzioni di letterali).

Dopo aver effettuato la conversione si cerca, poi, di dimostrare che la congiunzione della base di conoscenza con il negato della frase da dedurre sia insoddisfacibile, cioè non esista nessun modello in cui KB non implichi α. Se ciò fosse provato, significherebbe che in tutti i mondi in cui è vera la base di conoscenza, sarebbe vera anche la frase.

Si considerano, allora, coppie di clausole della frase complessa KB ∧ ¬α alle quali si applica la regola di risoluzione inferenziale:

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

where $\ell_i$ and $m_j$ are complementary literals.

si produce una nuova clausola che contiene tutti i loro letterali tranne quelli complementari. Si provvede ad applicare questo ragionamento ricorsivamente fino a che non si riescono più a generare nuove clausole, ottenendo esito negativo, oppure si arriva alla clausola vuota, rappresentante la deducibilità della frase dalla base di conoscenza. Anche questo algoritmo è corretto e completo per la logica proposizionale.

Mostriamo un esempio di risoluzione.

La nostra KB è data da :

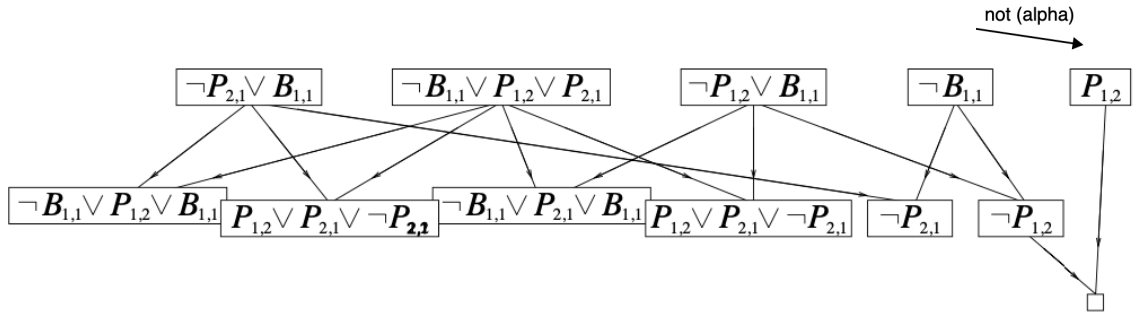
$$KB = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

Effettuando la conversione alla forma CNF otteniamo:

$$KB = (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) \wedge \neg B_{1,1}$$

Vogliamo provare KB ? |= α con α = not(P(1,2)).

Aggiungiamo quindi not(α) al set di clausole e applichiamo poi l'algoritmo di risoluzione per ogni coppia di queste.
Troviamo quindi la clausola vuota dunque perveniamo al fatto che KB |= α



# Work Project 3

Il work project 3 è incentrato sull'esposizione e l'utilizzo delle reti bayesiane e delle reti neurali

## 3.1 Esercizio 1 : Reti Bayesiane

Gli agenti potrebbero dover gestire situazioni di incertezza, sia a causa di osservabilità parziale del problema sia per non determinismo sia per una combinazione dei due.
Un agente potrebbe non sapere con certezza in quale stato si trova o dove finirà dopo una sequenza di azioni.
Un possibile approccio per gestire l'incertezza è tenere traccia di uno "stato di convinzione"(*belief state*), ossia una rappresentazione dell'insieme di tutti i possibili stati del mondo in cui l'agente potrebbe trovarsi, e prevedere un piano che gestisca ogni possibile eventualità che i suoi sensori possono segnalare durante l'esecuzione.
Questo approccio ha diverse problematiche: un piano che gestisca ogni eventualità può diventare incredibilmente grande e può portare a considerare situazioni altamente improbabili.
Una "decisione razionale" terrebbe quindi conto sia dell'importanza relativa dei vari obiettivi sia della probabilità con la quale possono essere raggiunti i diversi possibili mondi.
Per prendere una decisione del genere, ad una teoria dell'utilità viene aggiunta per l'agente una teoria della probabilità, ottenendo così una teoria della decisione.
Un modello di probabilità completamente specificato associa una probabilità numerica $P(\omega)$ a ciascun mondo possibile.
Un modello di probabilità è completamente specificato dalla *distribuzione di probabilità congiunta*, una matrice che rappresenta la probabilità di ciascun evento atomico.
Tale tabella incontra diversi problemi:
Innanzitutto cresce in modo combinatorio per quante sono le variabili.

Inoltre, dato che i valori di probabilità sono dati dall'uomo, risulta per quest'ultimo impossibile o quasi dare valori accurati per ogni cella della tabella.
Utilizziamo quindi per ridimensionare la tabella delle probabilità congiunte *l'indipendenza condizionata*, la quale consente di fattorizzare l'intera distribuzione congiunta in distribuzioni condizionali più piccole.
Il *naive Bayes model* assume l'indipendenza condizionale di tutte le variabili di effetto, data una variabile di causa, e quindi cresce linearmente con il numero di effetti.

Tale introduzione è strettamente propedeutica per enunciare le reti di Bayes.
Queste ultime infatti non sono altro che una notazione grafica per le asserzioni di indipendenza condizionata e quindi, per specificazioni compatte delle distribuzioni congiunte complete.

Una rete bayesiana è un grafo aciclico diretto in cui :
Ogni nodo corrisponde a una variabile aleatoria, che può essere discreta o continua.
I collegamenti diretti o frecce collegano coppie di nodi. Se esiste una freccia dal nodo X al nodo Y, si dice che X è un genitore di Y.
Ogni nodo Xi ha una distribuzione di probabilità condizionale P (Xi | Genitori (Xi)) che quantifica l'effetto dei genitori sul nodo.
La topologia della rete specifica le relazioni di indipendenza condizionale che ci sono nel dominio.
Il significato intuitivo di una freccia da X a Y è tipicamente che X influenza direttamente Y, il che suggerisce che i genitori sono le cause e i figli gli effetti.
Di solito è facile per un esperto di un determinato dominio specificare quali influenze dirette esistono all'interno di esso, molto più facile che specificare le probabilità stesse.
Una volta definita la topologia della rete bayesiana, dobbiamo solo specificare una distribuzione di probabilità condizionale per ogni variabile, dati i suoi genitori.
Nel caso più semplice, viene rappresentata come una tabella di probabilità condizionale (CPT) che fornisce la distribuzione su Xi per ogni combinazione dei valori del padre.

La combinazione della topologia e delle distribuzioni di probabilità condizionale è sufficiente per specificare (implicitamente) l'intera distribuzione di probabilità congiunta.

In particolare la semantica globale definisce la distribuzione congiunta completa come il prodotto delle distribuzioni condizionali locali. Quindi:

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

Per queste reti si definisce anche il concetto di semantica locale, cioè ogni nodo è indipendente dai suoi non discendenti dato il nodo padre (il che ci consente di calcolare la semantica globale, Semantica locale <=> Semantica globale), e della coperta di Markov, ovvero un nodo è indipendente da tutti gli altri dato il nodo padre, i figli e i padri dei figli (Semantica locale => coperta di Markov).

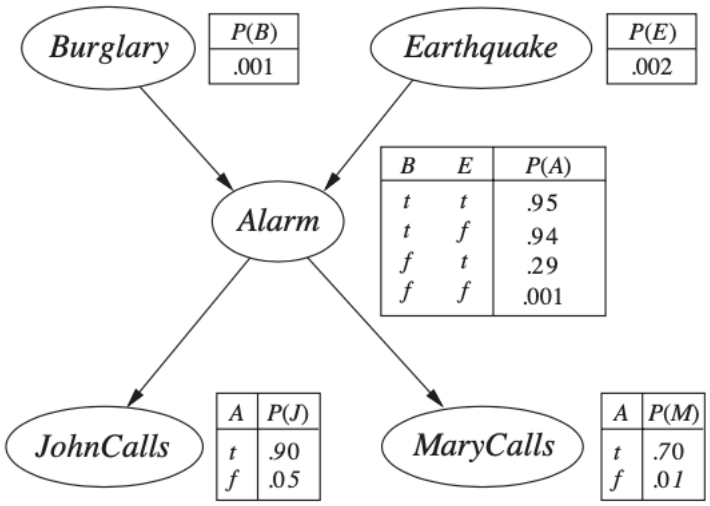
### 3.1.2 Inferenza nelle reti Bayesiane

La probabilità che un fenomeno si verifichi si può ottenere per ricombinazione successiva delle tabelle di probabilità condizionate degli elementi presenti sulla rete.
Tale tipo di inferenza è detta *inferenza per enumerazione*, l'algoritmo che effettua tale inferenza fa una ricerca depth-first sulla rete: bisogna esplorare la rete e ricalcolare le probabilità congiunte che ci servono per calcolare la probabilità condizionata o congiunta marginalizzata che il problema pone.
E' inefficiente in quanto rieffettua computazioni già fatte, e complesso nel caso in cui la rete sia grande, ossia il numero di possibili valori delle variabili sia elevato.
Se il mio obiettivo è quindi calcolare il valore di probabilità congiunta di un numero elevato di variabili l'onere computazionale potrebbe diventare non sostenibile.
Un altro approccio è quello frequentista (*inferenza stocastica*) che consiste nel generare un token che avrà una probabilità di ottenere un valore del nodo pari proprio all' incertezza legata ad esso, dopodiché si inizia a campionare la rete facendo scorrere il token su di essa, fissandone i valori con quelli risultanti dal campionamento. Le configurazioni ricavate, una volta raggiunta la quantità di risultati voluti, vengono divise per il numero dei campioni valutati, tenendo

conto della loro molteplicità. Tale metodo per un alto numero di eventi determinati deve tendere all'approccio di calcolare le probabilità a posteriori. Le reti bayesiane così costruite sono statiche perché non tengono conto di un riferimento temporale. Le reti bayesiane dinamiche tengono conto anche di un valore temporale, suddiviso in istanti.

### 3.1.3 Esempio di rappresentazione di una rete bayesiana

Consideriamo un esempio.
Supponiamo un nuovo antifurto installato a casa abbastanza affidabile nel rilevare un furto con scasso, ma che risponde anche a volte a piccoli terremoti. Supponiamo anche due vicini, John e Mary, che hanno promesso di chiamare al lavoro quando sentiranno l'allarme. John quasi sempre chiama quando sente l'allarme ma a volte lo confonde il telefono che squilla e chiama anche allora. Mary, d'altra parte, ama la musica piuttosto rumorosa e spesso non sente l'allarme. Date le prove di chi ha o non ha chiamato, vorremmo stimare la probabilità di un furto con scasso.
Una rete bayesiana per questo dominio appare in Figura.




La struttura della rete mostra che il furto con scasso e i terremoti influiscono direttamente sulla probabilità che l'allarme si attivi, ma che John e Mary chiamino dipende solo dall'allarme. La rete rappresenta quindi le nostre ipotesi che non percepiscono direttamente i furti con scasso, non notano piccoli terremoti e non si consultano prima di chiamare.

Le distribuzioni condizionali sono mostrate come una tabella di probabilità condizionale o CPT.
Ogni riga in un CPT contiene la probabilità condizionale di ciascun valore di nodo per un caso di condizionamento. Un caso di condizionamento è solo una possibile combinazione di valori per i nodi principali.
Ogni riga deve essere pari a 1, poiché le voci rappresentano un insieme esaustivo di casi per la variabile. Dunque, una volta noto che la probabilità di un valore vero è p, la probabilità di falso deve essere 1 - p, quindi spesso si omette tale numero. In generale, una tabella per una variabile booleana con k genitori contiene 2k probabilità specificabili indipendentemente. Un nodo senza genitori ha solo una riga, che rappresenta le probabilità a priori di ogni possibile valore della variabile.
Si noti che la rete non ha nodi corrispondenti a Mary che sta ascoltando musica ad alto volume o che il telefono squilla e confonde John. Questi fattori sono riassunti nell'incertezza associata ai collegamenti da Alarm a JohnCalls e MaryCalls. Le probabilità in realtà riassumono anche un insieme potenzialmente infinito di circostanze in cui l'allarme potrebbe non riuscire a suonare (alta umidità, mancanza di corrente, batteria scarica, fili tagliati ecc.).

Supponiamo ora, su questa rete, di voler calcolare la probabilità :

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

Utilizzando la semantica globale:

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

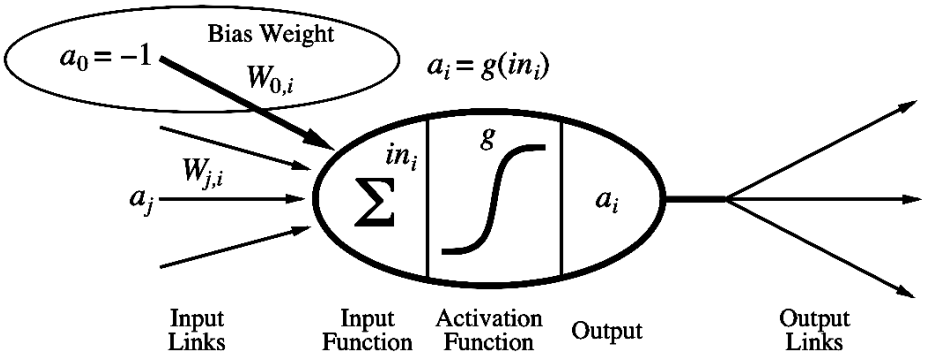
Otteniamo:

$$\begin{aligned} &= P(j|a)P(m|a)P(a|\neg b, \neg e)P(\neg b)P(\neg e) \\ &= 0.9 \times 0.7 \times 0.001 \times 0.999 \times 0.998 \\ &\approx 0.00063 \end{aligned}$$

## 3.2 Esercizio 2 : Reti Neurali

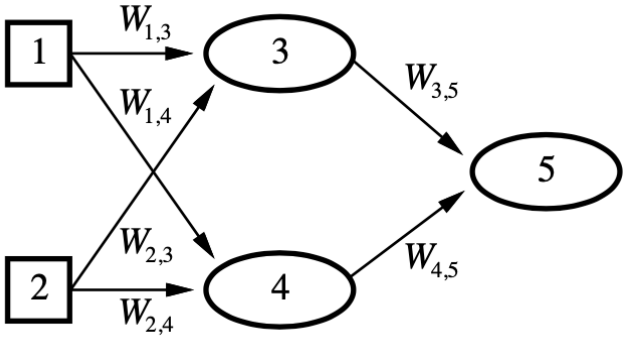
Le reti neurali sono un meccanismo computazionale che si ispira al funzionamento del cervello umano. Sono rappresentabili tramite un grafo orientato, i cui nodi costituiscono un modello matematico del neurone, e agli archi orientati è associato un peso. Ogni nodo *i* calcola la sua uscita *ai* applicando una funzione di attivazione *g*, chiamata *activation function*, alla somma pesata degli ingressi.

$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$




La funzione di attivazione storicamente utilizzata era quella di soglia; in seguito si è diffusa la funzione *sigmoide*, detta anche *funzione logistica*.

I singoli neuroni possono essere strutturalmente collegati in due modi.
Possiamo avere una rete *feed-forward* la quale ha connessioni solo in una direzione, ovvero forma un grafico aciclico diretto. Ogni nodo riceve input da nodi "upstream" e fornisce output a nodi "downstream"; non ci sono anelli.
Un esempio di rete feed-forward è il seguente:



abbiamo dei valori in ingresso rappresentati dai quadratini a sinistra, abbiamo quindi un vettore di valori.
Questi valori vengono propagati all'interno della rete secondo gli archi con cui la rete è costruita.
Il nodo 5 è quello di uscita in quanto produce il valore di output.
Tale valore viene calcolato tenendo conto del peso dei vari archi:

$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

Possiamo avere poi una *rete ricorrente*, la quale reinserisce invece le proprie uscite nei propri ingressi. Ciò significa che i livelli di attivazione della rete formano un sistema dinamico che può raggiungere uno stato stabile oppure mostrare oscillazioni o persino comportamenti caotici. Inoltre, la risposta della rete a un determinato input dipende dal suo stato iniziale, che può dipendere da input precedenti. Pertanto, le reti ricorrenti (diversamente dalle reti feed-forward) possono supportare la memoria a breve termine.

Le reti feed-forward sono generalmente disposte in strati, o *layer* , in modo tale che ogni unità riceva input solo dalle unità nel livello immediatamente precedente. Le reti neurali a singolo layer riescono a rappresentare solo operazioni linearmente separabili, a due layer tutte le funzioni continue, a tre layer anche le funzioni discontinue.
(Il termine *deep-learning* non è nient'altro che un alias per indicare reti neurali multilivello)

Una difficoltà non trascurabile risiede nella costruzione della struttura della rete neurale. Esse, infatti, sono sistemi black-box, il cui funzionamento interno è di difficile interpretazione logica.
Per determinare il peso degli archi si utilizza l'algoritmo di *Error Back Propagation*.
Inizialmente si costruisce una rete con pesi casuali o uniformi per abolire l'onere computazionale.

Si considera poi una coppia ingresso-uscita del dataset. L'errore tra l'uscita ottenuta e quella desiderata verrà propagato all'indietro per ogni livello, così da modificare i pesi ed ottenere una migliore approssimazione della funzione. Tale propagazione fa uso del *learning rate* per far si che un singolo esempio non influisca troppo pesantemente sui valori dei pesi, altrimenti questi tenderebbero ad oscillare di molto nel caso in cui ogni esempio fornisca un errore relativamente grande. La procedura verrà iterata per ogni esempio, ciascuno dei quali consentirà una correzione dei pesi, delineando una struttura della rete coerente con la funzione da approssimare.
Lo svantaggio di questo meccanismo è che è molto costoso in termini di tempo.
Per quanto riguarda, invece, la determinazione del numero di neuroni, si procede tipicamente con un approccio trial and error.
Si parte, quindi, da reti semplici: se queste dopo la Back Propagation funzionano bene, le si usa, altrimenti si iniziano a inserire nuovi neuroni o layer di neuroni e si continua a testare la validità delle nuove reti create.
In definitiva, le reti neurali si sono rivelate empiricamente un buon approssimatore di funzioni ed oggi il loro utilizzo ha ripreso vigore nella forma di Deep Learning. Tali tecniche ottengono un basso error rate nei problemi relativi al riconoscimento di caratteri e sono largamente diffuse nei sistemi multimediali per quanto concerne il riconoscimento di immagini, audio o nella Computer Vision in generale. Seppur presentando buoni risultati, bisogna ricordare che *non esiste un algoritmo di apprendimento migliore in assoluto, ma bisogna trovare quello più adatto al problema dato*.

### 3.2.2 Reti neurali per l'image recognition

Il riconoscimento delle immagini utilizza tecniche di intelligenza artificiale per identificare automaticamente oggetti, persone, luoghi e azioni nelle immagini. Esso viene utilizzato per eseguire attività come l'etichettatura delle immagini con tag descrittivi, la ricerca di contenuti all'interno della immagini stesse e la guida di robot e veicoli autonomi.
Il riconoscimento delle immagini è naturale per l'uomo e gli animali, ma è un compito estremamente difficile da eseguire per i computer.

Negli ultimi due decenni, è emerso il campo del Computer Vision e lo strumento più efficace trovato per l'attività di riconoscimento delle immagini è una rete neurale profonda.

L'occhio umano vede un'immagine come un insieme di segnali, interpretati dalla corteccia visiva del cervello. Il risultato è un'esperienza di una scena, legata a oggetti e concetti che vengono conservati nella memoria. Il riconoscimento delle immagini imita questo processo. I computer "vedono" un'immagine come un insieme di vettori(primitive geometriche alle quali possono essere attribuiti colori) o come raster (una tela di pixel con valori numerici discreti per i colori).
Nel processo di riconoscimento dell'immagine, la codifica vettoriale o raster dell'immagine viene trasformata in costrutti che descrivono oggetti e caratteristiche fisiche.
I sistemi di visione artificiale possono analizzare logicamente questi costrutti, prima semplificando le immagini ed estraendo le informazioni più importanti, quindi organizzando i dati attraverso l'estrazione e la classificazione delle caratteristiche.
Infine, i sistemi di visione artificiale utilizzano la classificazione o altri algoritmi per prendere una decisione sulle immagini o parte di esse (a quale categoria appartengono o come possono essere meglio descritte).
Un tipo di algoritmo di riconoscimento delle immagini è un classificatore di immagini. Prende un'immagine (o parte di un'immagine) come input e presagisce cosa l'immagine contiene.
L'output è un'etichetta di classe, come cane, gatto, albero etc.
L'algoritmo deve essere addestrato per apprendere e distinguere tra le classi.
Facciamo il caso di un algoritmo di classificazione in grado di identificare la presenza di alberi in un'immagine.
Bisogna allenare la rete neurale con migliaia di immagini di alberi e migliaia di immagini senza alberi. L'algoritmo imparerà a estrarre le caratteristiche che identificano l'oggetto "albero" e ad identificare tale oggetto all'interno dell'immagine.

E' importante sottolineare che gli algoritmi di riconoscimento delle immagini della rete neurale si basano sulla qualità del set di dati.

Alcuni parametri da tenere in conto sono ad esempio: la dimensione e il numero delle immagini, il numero di canali, le proporzioni, la deviazione standard e la normalizzazione dei dati di input e così via.
Una volta preparate le immagini di allenamento, vengono elaborate dalla rete neurale in modo da poter fare previsioni su immagini nuove e sconosciute.

Le reti neurali tradizionali usano un'architettura completamente connessa in cui ogni neurone in uno strato si collega a tutti i neuroni nello strato successivo.
Un'architettura completamente connessa è inefficiente quando si tratta di elaborare dati di immagine.
Per un'immagine media, quindi con centinaia di pixel e tre canali, una rete neurale tradizionale genererà milioni di parametri, il che può portare a un overfitting.
Il modello sarebbe molto difficile da trattare anche dal punto di vista computazionale.
Potrebbe inoltre essere difficile interpretare i risultati, eseguire il debug e ottimizzare il modello per migliorarne le prestazioni.
A differenza di una rete neurale completamente connessa, si può utilizzare in una rete neurale convoluzionale (CNN), in cui i neuroni in uno strato non si collegano a tutti i neuroni nello strato successivo.
Piuttosto, una rete neurale convoluzionale utilizza una struttura tridimensionale, in cui ogni serie di neuroni analizza una regione o "caratteristica" specifica dell'immagine.
Considerando sempre l'esempio dell'albero, un gruppo di neuroni potrebbe identificare il tronco, un altro le foglie, un altro i rami, ecc.
L'output finale è un vettore di probabilità, che prevede, per ciascun oggetto dell'immagine, la probabilità che appartenga a una classe o categoria.