Intelligenza artificiale

Programma

Parte 1: Ricerca e agenti

- Introduzione
- **Agenti intelligenti**: [RN cap 2.1-2.3]
  - agenti intelligenti,
  - proprietà degli ambienti,
  - definizione di agente razionale,
  - rappresentazione degli stati,
  - spazio degli stati,
  - grafo dello spazio degli stati
- **Ricerca non informata**: [RN cap 3.1 - 3.4]
  - definizione del problema di ricerca,
  - strategie di ricerca non informata (ampiezza, profondità),
  - costo uniforme,
  - complessità, completezza e ottimalità
  - complessità di BFS, DFS, UCS
  - depth limited search
  - iterative deepening DFS
- **Ricerca informata**:
  - euristiche A*

    - IDA*, RBFS, SMA*

Parte 2: Machine learning

- **Introduzione al Machine Learning**
- **Decision Trees**
- **Model over-fitting**
- **Evaluation metrics**
- **Artificial neural networks**
- **K nearest neighbours**
- **Naive Bayes**

Introduzione

Definizione di intelligenza:

	pensare	comportarsi
umanamente	1	2
razionalmente	3	4

1. pensare umanamente: modello cognitivo
   Vogliamo capire se il programma pensa come un umano. Vengono comparate le sequenze e le tempistiche degli step di ragionamento del programma con quelli umani.
2. comportarsi umanamente: test di turing
   Viene testato che il programma possa comportarsi come un umano.
   Per fare ciò ha bisogno di:
   - natural language processing, per comunicare in un linguaggio umano
   - rappresentazione dell'informazione, per mantenere quello che conosce
   - ragionamento automatico, per rispondere a domande e per arrivare a nuove conclusioni
   - machine learning, per adattarsi a nuovi scenari
3. pensare razionalmente: le leggi del pensiero
   La logica studia le leggi del ragionamento e della dimostrazione.
   Si tenta di costruire un programma che possa risolvere un problema utilizzando il ragionamento logico.

Visto che le informazioni che abbiamo su molti fenomeni sono parziali (non conosciamo le leggi della politica), abbiamo bisogno della teoria della probabilità per arrivare a conclusioni vere.

4. agire razionalmente: l'approccio dell'agente razionale
   Un agente è qualcosa che agisce; dai computer agents ci si aspetta che operino autonomamente, percepiscano il loro ambiente, si adattino al cambiamento.
   Un
   agente razionale agisce per ottenere il migliore risultato o il risultato atteso.
   Questo approccio ha due vantaggi: è più generale di 3 perché l'inferenza è solo uno dei possibili modi per ottenere la razionalità, è più aperto allo sviluppo scientifico di 1 e 2 perché la razionalità è verificabile.
   Il campo dell'AI si è concentrato sullo studio e sullo sviluppo di agenti che facciano la cosa giusta.

Il modello dell'agente razionale assume che venga fornito alla macchina un obiettivo completamente specificato mentre molto spesso quello che si deve fare è bilanciare l'obiettivo e i suoi possibili effetti collaterali.

Il problema è detto: **problema dell'allineamento dei valori** i valori o gli obiettivi forniti alla macchina devono essere allineati a quelli degli umani.

Agenti intelligenti

> **Obiettivo sezione: discussione sulla natura degli agenti, la diversità degli ambienti e le tipologie di agenti risultanti.**

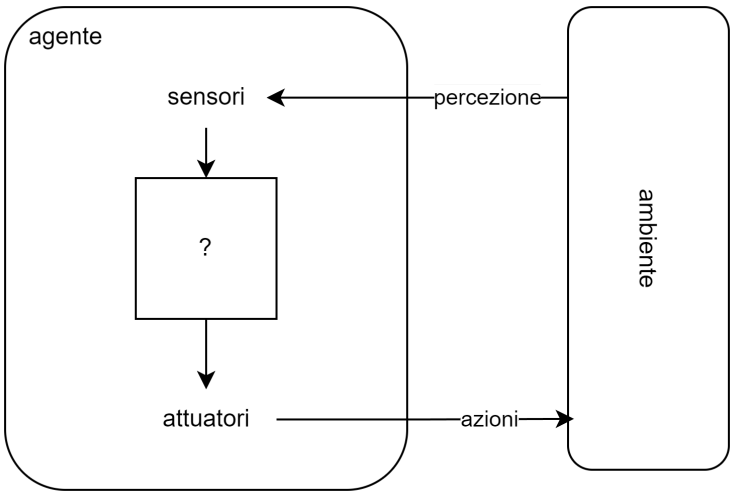
> Un **agente** è qualisiasi cosa che può percepire il suo ambiente tramite dei **sensori** e che può agire sul suo ambiente tramite degli **attuatori**.

Un agente elabora le percezioni che ha sull'ambiente e ne deriva le azioni da eseguire per ottenere lo stato desiderato.

La storia completa di tutto ciò che l'agente ha percepito è detta **sequenza di percezione.**

L'azione di un agente in un determinato momento dipende dalla sua conoscenza e dalla sua sequenza di percezione osservata fino a quel momento.

Matematicamente possiamo dire che il comportamento di un agente è descritto dalla **funzione agente** che mappa ogni possibile sequenza percettiva a un'azione.
Potremmo pensare di realizzare una tabella che associa a ogni possibile sequenza percettiva l'azione che l'utente deve compiere e essa rappresenterebbe un'implemetazione della funzione agente.
La funzione agente è solo una rappresentazione matematica astratta, internamente essa sarà implementata come
programma agente.



Concetto di razionalità

> Un **agente razionale** è un agente che fa la cosa giusta.

Per valutare cosa significa fare la cosa giusta utilizziamo la nozione del **consequenzialismo**: valutiamo il comportamento di un agente dalle sue conseguenze.

Nel momento in cui un agente è a contatto con un'ambiente, genererà una serie di azioni che faranno passare l'ambiente in una sequenza di stati. Se tale sequenza è **desiderabile** allora l'agente si è comportato bene.

La desiderabilità di una sequenza di stati è valutata dalla **misurazione delle performance**.

Razionalità

Ciò che è razionale in un dato istante dipende da:

- la misura delle performance che definisce il criterio di successo
- la conoscenza pregressa che l'agente ha dell'ambiente
- le azioni che l'agente può svolgere
- la sequenza percettiva fino a quel momento

> **Agente razionale**: per ogni possibile sequenza percettiva, un agente razionale deve selezionare un'azione che ci si apetta massimizzi la misurazione delle performance, data l'evidenza fornita dalla sequenza percettiva e dalla conoscenza dell'agente.

Natura degli ambienti

Prima di poter pensare a come costruire un agente dobbiamo studiare gli **ambienti di lavoro** che sono i problemi per i quali gli agenti sono la soluzione. La natura dell'ambiente di lavoro influenza direttamente il design appropriato per l'agente.

Specificare l'ambiente di lavoro

L'ambiente di lavoro è composto da: la misurazione delle performance, l'ambiente, i sensori e gli attuatori dell'agente → PEAS = performance, environment, attuatori, sensori

Il primo step per il design di un agente è quello di specificare l'ambiente di lavoro meglio possibile.

Es: tassista automatizzato

tipo di agente	misurazione delle performance	ambiente	attuatori	sensori
taxi driver	sicuro, veloce, legale, confortevole, massimizzazione del profitto, minimizza la possibilità di impatto con altri utenti della strada	strade, traffico, polizia, pedoni, clienti, meteo	sterzo, acceleratore, freno, clacson, display, discorso parlato	videocamere, radar, tachimetro, GPS, sensori motore, accelerometro, microfono, touchscreen

Le proprietà dell'ambiente di lavoro

Possiamo classificare gli ambienti di lavoro per determinare quale design è migliore per l'implementazione di un agente.

- **totalmente osservabili** - **parzialmente osservabili**: se i sensori di un agente gli danno accesso allo stato completo dell'ambiente in ogni istante di tempo, allora diciamo che l'ambiente di lavoro è totalmente osservabile. In pratica, un ambiente di lavoro è totalmente osservabile quando l'agente ha accesso a tutti gli aspetti dell'ambiente che sono rilevanti per le sue azioni.
  Se un agente non ha sensori, l'ambiente di lavoro non è osservabile.
- **single-agent** - **multi-agent**: un ambiente di lavoro è multi-agent quando ci sono due agenti A e B ognuno dei quali agisce per massimizzare delle misurazioni di performance che dipendono dal comportamento dell'altro.
- **competitivo** - **cooperativo**: un ambiente di lavoro è competitivo quando l'agente A massimizza la sua misurazione delle performance minimizzando quella di B.
- **deterministico** - **non deterministico**: quando lo stato successivo dell'ambiente è completamente determinato dal suo stato attuale e dall'azione dell'agente allora l'ambiente di lavoro è deterministico. Quando un ambiente è complesso possono esserci aspetti non osservabili che fanno in modo che esso sia percepito come non deterministico.
- **episodico** - **sequenziale**: in un ambiente di lavoro episodico, l'esperienza dell'agente è divisa in episodi atomici. In ogni episodio l'agente riceve una

percezione e effettua una singola azione. Gli episodi sono tra loro indipendenti. Negli ambienti sequenziali la decisione corrente può influenzare quelle successive (es: scacchi, taxi driver)

- **statico** - **dinamico**: se l'ambiente cambia mentre l'agente sta decidendo sull'azione da compiere allora l'ambiente di lavoro è dinamico (es: taxi driver). Se l'ambiente non cambia nel tempo mentre l'agente decide ma cambia il suo punteggio (es: scacchi con clessidra) allora l'ambiente è semidinamico. Se l'ambiente non cambia allora l'ambiente di lavoro è statico (es: puzzle)
- **discreto** - **continuo**: la distizione tra discreto e continuo viene applicata al tempo, agli stati dell'ambiente, alle percezioni e alle azioni dell'agente.
  Es: scacchi = tempo continuo, numero di stati discreto, percezioni e azioni discrete
- **conosciuto** - **sconosciuto**: un ambiente di lavoro è conosciuto quando si conoscono le leggi che lo regolano.

Struttura degli agenti

Lo scopo dell'AI è progettare un **programma agente** che implementi la funzione agente. Assumiamo che il programma verrà eseguito su un dispositivo di computazione con sensori e attuatori fisici (**architettura agente**).

```
agente = architettura + programma
```

Il programma scelto deve essere adeguato all'architettura.

Programma agente

I programmi agente avranno tutti la stessa struttura: prendono in ingresso la percezione dai sensori e restituiscono un'azione agli attuatori.

Il programma agente, a differenza della funzione agente, prende come input solo la percezione corrente, se c'è bisgono che l'azione dipenda dall'intera sequenza di percezione è necessario implementare una memoria.

4 tipi base di programma agente:

- simple reflex

- model-based reflex
- goal-based
- utility-based

Agenti simple reflex

Il più semplice tipo di agente è il **simple reflex**. Questi agenti selezionano le azioni da eseguire in base alla percezione corrente, ignorando la sequenza.

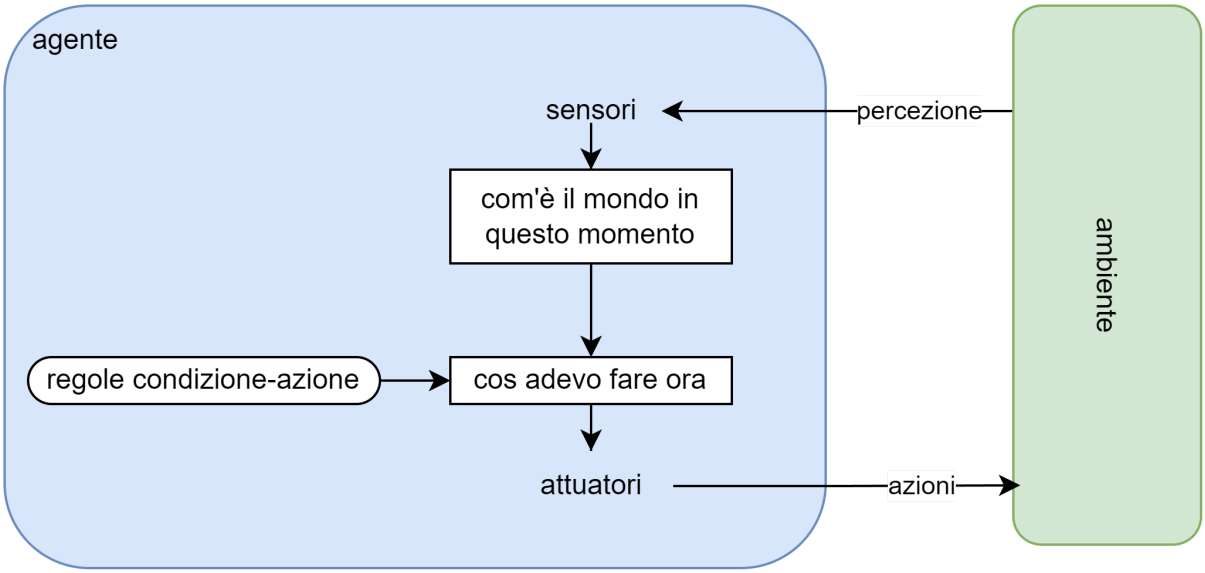
Es: aspirapolvere in una stanza costituita da due mattonelle A e B

```
function REFLEX-VACUUM-AGENT([location, status]) returns an action
    if status = Dirty then return Clean
    else if location = A then return Right
    else if location = B then return Left
```

Il programma è scritto sotto forma di **regole condizione-azione** del tipo: *if condition then action*

L'agente funziona tramite delle regole condizione-azione che gli permettono di effettuare la connessione tra la percezione e l'azione.

Questo tipo di agenti funziona bene solo le l'azione giusta può essere scelta in base alla percezione corrente, nel momento in cui l'ambiente non è totalmente osservabile si hanno dei fallimenti.




```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition-action rules

  state = INTERPRET-INPUT(percept)
  rule = RULE-MATCH(state, rules)
  action = rule.ACTION
  return action
```

Model-based reflex agent

La maniera più efficiente che l'agente ha per gestire la parziale osservabilità dell'ambiente è quella di tenere traccia della porzione di mondo che non può vedere in questo istante.

L'agente deve mantenere uno stato interno che dipende sulla storia delle percezioni e riflette almeno uno degli aspetti non osservabili dello stato attuale.

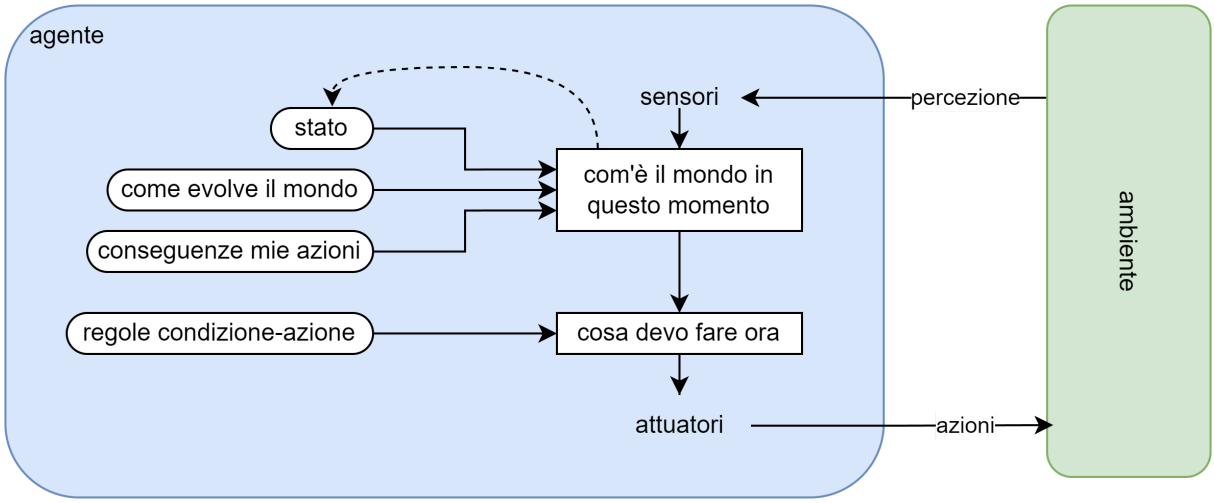
Per aggiornare lo stato interno dell'agente al passare del tempo abbiamo bisogno di inserire due tipi di conoscenza nel programma agente:

1. informazione su come il mondo cambia nel tempo, che può essere suddivisa in:
    a. gli effetti delle azioni dell'agente

b. come il mondo evolve indipendentemente dall'agente → **modello di transizione**

2. informazione su come lo stato dell'ambiente si riflette sulle percezioni dell'agente → **modello del sensore**

Modello di transizione e modello del sensore permettono all'agente di tenere traccia dello stato del mondo. Un agente che usa questi modelli è detto **model-based agent.**



Nella figura vediamo che la percezione attuale viene combinato con lo stato interno precedente per generare una descrizione aggiornata dello stato corrente, basato sul modello del funzionamento del mondo (1b).

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              transition_model, description of how the next state depends on
                and action
              sensor_model, description of how the current world state is refle
                agent's percepts
              rules, a set of condition-action rules
              action, the most recent action
  state = UPDATE-STATE(state, action, percept, transition_model, sensor_model
  rule = RULE-MATCH(state, rules)
```

```
    action = rule.ACTION
    return action
```

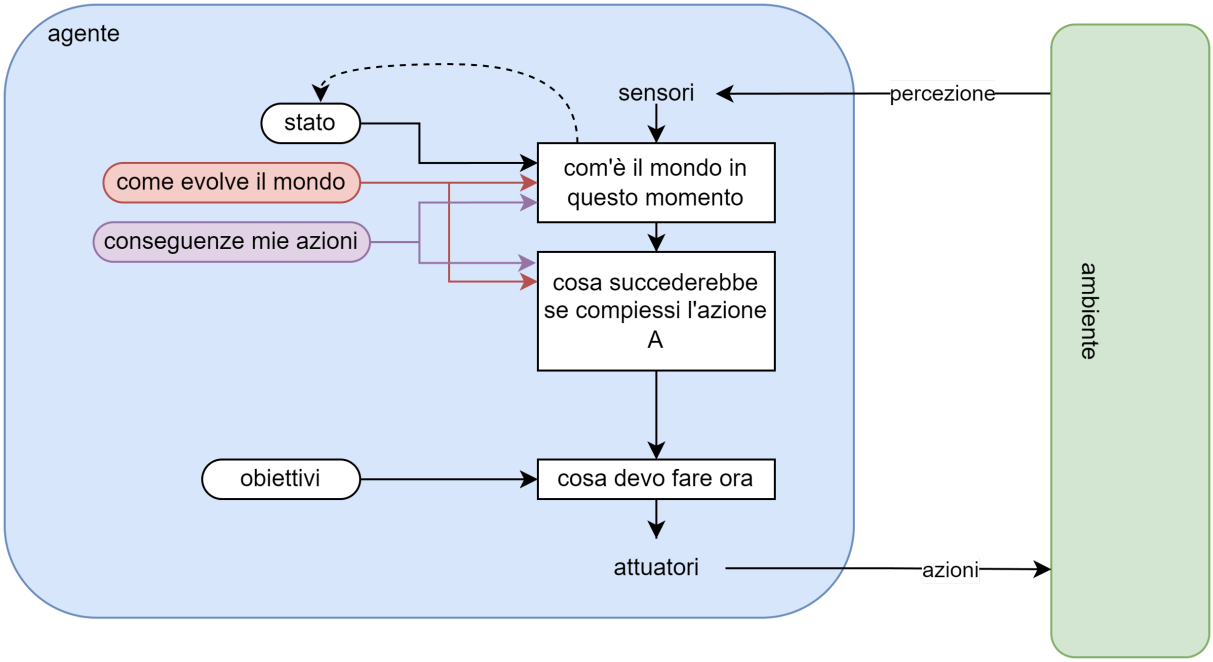
la funzione `UPDATE-STATE` è reponsabile della creazione del nuovo stato interno.

Raramente un agente riesce a determinare lo stato corrente di un mondo parzialmente osservabile, può lavorare solo sulla migliore approssimazione.

Goal-based agents

Insieme alla descrizione dello stato corrente, l'agente ha bisogno di qualche tipo di informazione sul suo obiettivo che descriva la situazione desiderabile.

Questo tipo di agente è più flessibile perché la conoscenza che supporta le sue decisioni è rappresentata esplicitamente e può essere modificata.



Utility-based agents

Gli obiettivi da soli non sono sufficienti a generare un comportamento di alta qualità nella maggior parte degli ambienti.

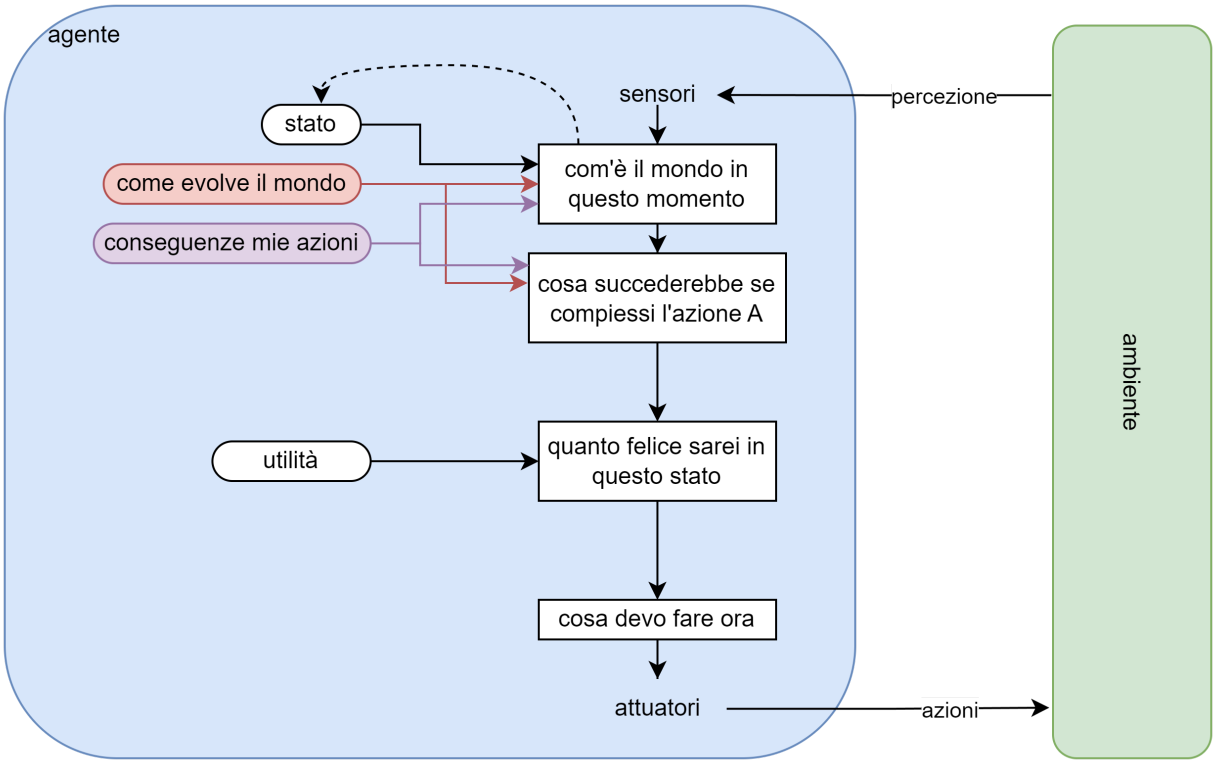
Gli obiettivi forniscono solo una differenziazione tra "**utile**" e "non utile". Ma una misurazione delle performance più accurata prende in considerazione anche la quantità di utile ottenuto.

La misurazione delle performance assegna un punteggio a ogni sequenza di stati dell'ambiente, quindi possiamo distinguere tra i modi più o meno desiderabili di raggiungere l'obiettivo.

La **funzione di utilità** dell'agente è un'interiorizzazione della misurazione delle performance.
Dal momento che la funzione di utilità e la misurazione delle performance sono allineate, un agente che sceglie azioni che massimizzano la sua utilità sarà giudicato razionale in accordo con la misurazione delle performance.

La funzione di utilità aiuta anche a scegliere quale compromesso adottare, si sceglie quello con maggiore utilità. Nel momento in cui ci sono più obiettivi che l'agente può completare, la funzione di utilità fornisce un mezzo per pesarne la possibilità di successo.

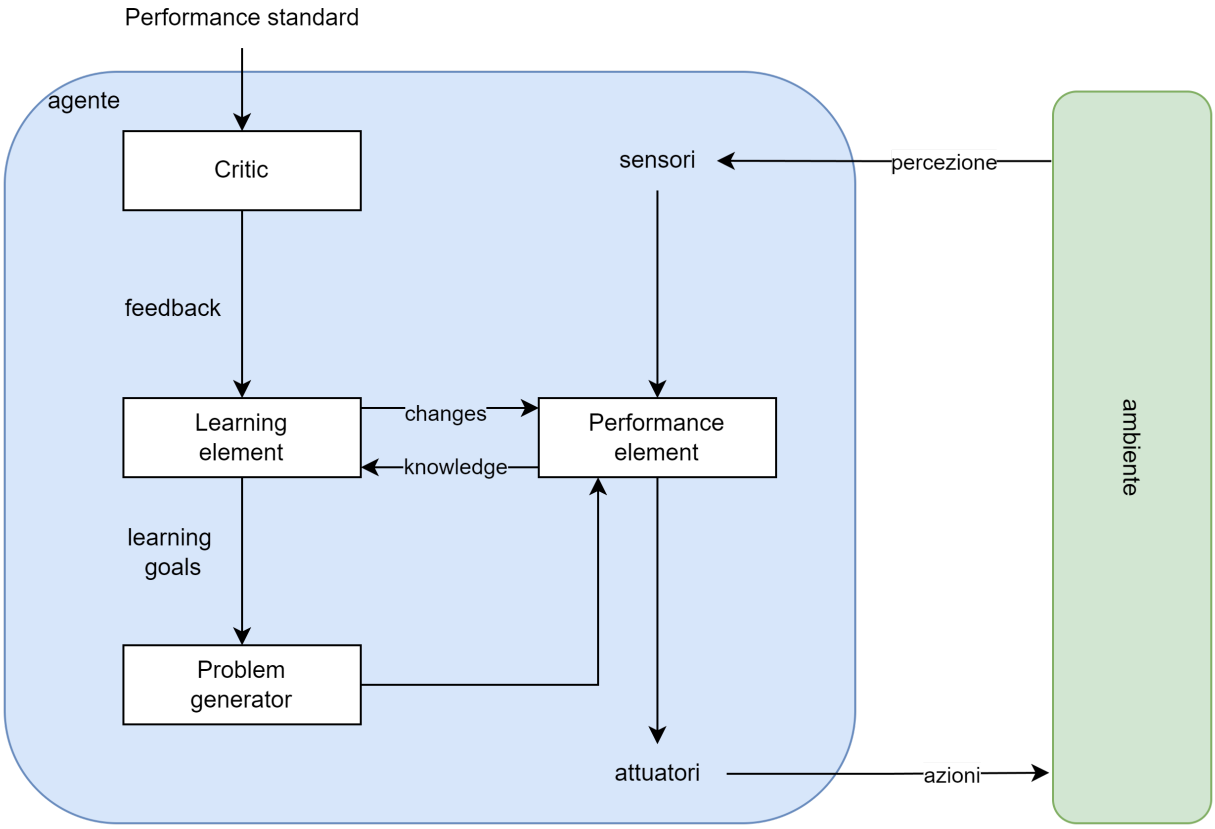
Un agente di questo tipo sceglie l'azione che massimizza **l'utilità attesa** delle conseguenze dell'azione.



Learning-agents

L'apprendimento permette all'agente di operare in un ambiente inizialmente sconosciuto e di diventare più competente di quanto la sua conoscenza iniziale gli permetta.

Un agente che apprende può essere suddiviso in 4 componenti concettuali:



- elemento di apprendimento: responsabile degli avanzamenti fatti, ottiene feedback dalla critica su come l'agente si sta comportando e determina come l'elemento di performance deve essere modificato per fare meglio in futuro
- elemento di performance: quello che nei precedenti modelli era l'intero agente, prende in input le percezioni e restituisce le azioni
- critica: fornisce feedback sul successo delle azioni dell'agente
- generatore di problemi: suggerisce azioni che poteranno a nuove esperienze, suggerisce azioni di esplorazione. Potrebbe identificare parti del modello che

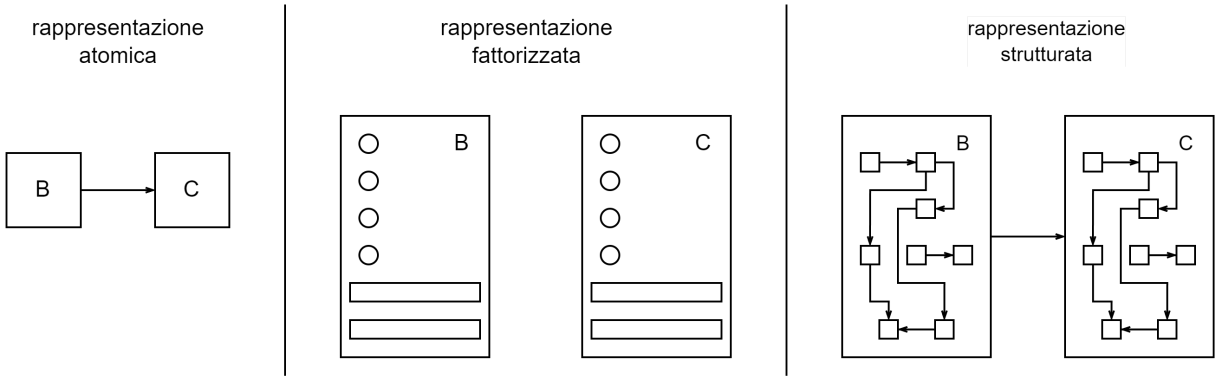
hanno bisogno di miglioramenti e suggerire esperimenti.

L'apprendimento può essere riassunto come il processo di modificazione di ogni componente dell'agente per fare in modo che essi siano più in linea con il feedback esterno, migliorando le performance complessive dell'agente.

Come funzionano i componenti degli agenti

Un componente può rappresentare il mondo che lo circonda in maniera:

- atomica
- fattorizzata
- strutturata



Nella rappresentazione atomica ogni stato è indivisibile, è trattato come una scatola nera. Due stati atomici non hanno nulla in comune, li distingue l'unica informazione che contengono.

Nella rappresentazione fattorizzata ogni stato è suddiviso in un insieme fissato di variabili e attributi con un proprio valore. Due stati in questa rappresentazione possono condividere il valore di alcune variabili.

Nella rappresentazione strutturata gli stati contengono oggetti in relazione tra loro.

Le rappresentazioni elencate vanno dalla meno alla più rappresentativa. Con l'aumento dell'espressività aumenta anche la difficolta di apprendimento quindi gli agenti intelligenti hanno bisogno di operare con diverse rappresentazioni simultaneamente.

Un’altra distinzione che possiamo fare riguardo al funzionamento del componente è basata sulla locazione delle informazioni; possiamo distinguere tra **rappresentazioni localizzate**, in cui esiste una mappatura 1-a-1 tra un concetto e una locazione di memoria, e **rappresentazioni distribuite**, in cui la rappresentazione di un concetto è distribuita in più locazioni.

Risovere problemi tramite la ricerca

> **Obiettivo sezione: capire come gli agenti possono trovare una sequenza di azioni che gli permetta di raggiungere il suo obiettivo.**

Quando l’azione corretta da solvere non è immediatamente ovvia, l’agente potrebbe aver bisogno di pianificare per considerare una sequenza di azioni che portino a uno stato “finale” (raggiungimento obiettivo).

Questo tipo di agente è detto **problem-solving agent** e il processo che intraprende è detto **ricerca**.

I problem-solving agents utilizzano la rappresentazione atomica.

Possiamo distinguere tra **ricerca informata**, in cui l’agente può stimare quanto è lontano dall’obiettivo, e **ricerca non informata**.

Problem-solving agents

Processo di risoluzione del problema:

- **formulazione dell’obiettivo**: l’agente adotta l’obiettivo, limitando le azioni da considerare;
- **formulazione del problema**: l’agente elabora una descrizione degli stati e delle azioni necessarie per il raggiungimento dell’obiettivo, creando un modello astratto della parte significativa del mondo;
- **ricerca**: prima di intraprendere qualsiasi azione nel mondo reale, l’agente simula la sequenza di azioni sul suo modello, cercando fino a quando non trova una sequenza che gli permette di raggiungere l’obiettivo. Tale sequenza è detta soluzione;

- **esecuzione**: l'agente esegue le azioni della soluzione, una alla volta.

Se l'agente si trova in un mondo deterministico, conosciuto e completamente osservabile, allora può ignorare le percezioni durante la fase di esecuzione → **sistema ad anello aperto**

se la soluzione ha qualche possibilità di non essere corretta allora l'agente deve continurare a monitorare le percezioni → **sistema ad anello chiuso**

Problemi di ricerca e soluzioni

Un problema di ricerca può essere formalmente definito come segue:

- un insieme di possibili stati in cui può trovarsi l'ambiente (**spazio degli stati**)
- lo **stato iniziale** in cui l'agente inizia
- un insieme di **stati finali**
- le **azioni disponibili** all'agente; dato uno stato *s* la funzione `ACTIONS(s)` restituisce tutte le azioni che possono essere eseguite in *s*. Diciamo che ognuna di queste azioni è **applicabile** in *s*.
- **modello di transizione**: descrive cosa fa ogni azione
- una **funzione di costo dell'azione**, `ACTION-COST(s,a,s')` che restiuisce il costo di applicare l'azione *a* nello stato *s* per raggiungere lo stato *s'*. Un problem-solving agent dovrebbe usare una funzione di costo dell'azione che rifletta la sua misurazione delle performance

Una sequenza di azioni forma un **cammino** e una soluzione è un cammino dallo stato iniziale a uno degli stati finali.
Assumiamo che i costi delle azioni siano additivi, così il costo di un cammino è la somma dei costi delle azioni che lo compongono.
Una
soluzione ottimale ha il costo di cammino minore.

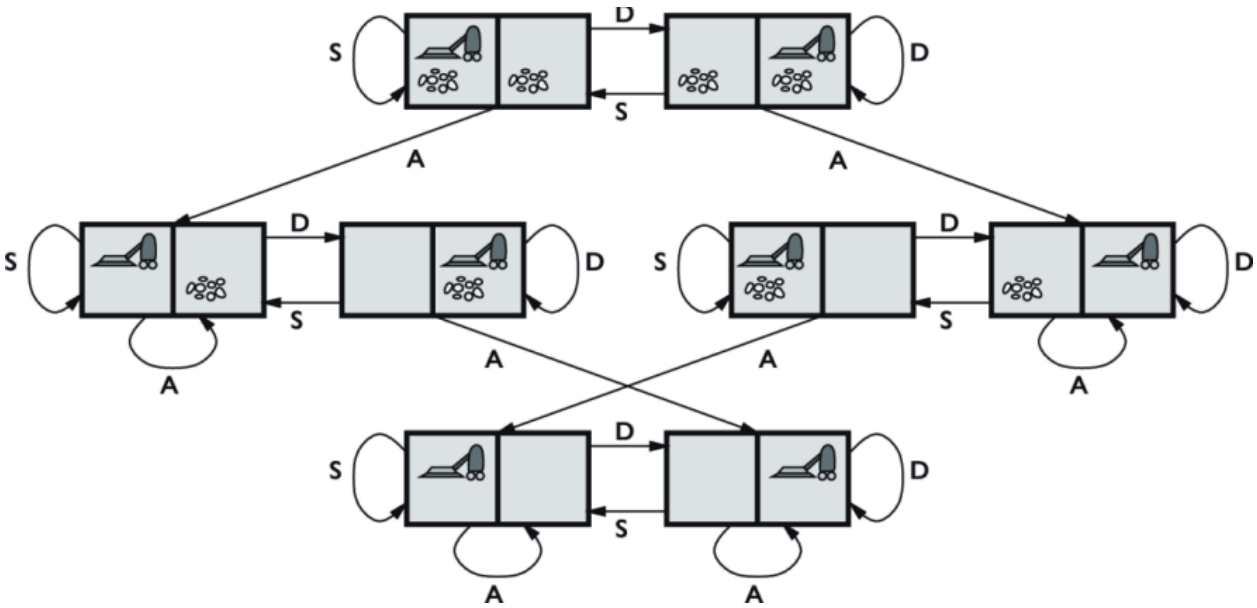
Lo spazio degli stati può essere rappresentato come un grafo in cui i vertici sono gli stati e gli archi diretti sono le azioni.

Esempi di problemi

I problemi si possono dividere in:

- **standardizzati**: nascono per illustrare o testare vari tipi di metodi di risoluzione di problemi
- **del mondo reale**

Problemi standardizzati



Il problema del vaccum cleaner può essere formulato come segue:

- **spazio degli stati**: gli stati ci dicono quali oggetti si trovano in quali celle, gli oggetti sono: l'agente e lo sporco. In questo caso abbiamo due celle quindi l'agente si può trovare in una delle due e ogni cella può essere sporca o pulita, abbiamo 2∗2∗2 = 8 stati
- **stato iniziale**: ogni stato può essere quello iniziale
- **azioni disponibili**: se abbiamo due celle, le azioni disponibili sono: pulisci, spostati a dx, spostati a sn
- **modello di transizione**:
  - pulisci: rimuove lo sporco dalla cella in cui si trova l'agente
  - destra: sposta l'agente nella cella a destra, a meno che non ci sia un muro

    - sinistra: sposta l'agente nella cella a sinistra, a meno che non ci sia un muro
- **stati finali**: gli stati in cui ogni cella è pulita
- **costo delle azioni**: ogni azione costa 1

Problemi del mondo reale

Problema del viaggio aereo:

- **spazio degli stati**: ogni stato contiene una posizione e il tempo attuale.
- **stato iniziale**: l'aereoporto di partenza dell'utente
- **azioni disponibili**: prendere un qualsiasi volo dall'aereoporto corrente
- **modello di transizione**: lo stato risultante dall'azione di prendere un volo avrà come posizione l'aeroporto di arrivo e come tempo l'ora di arrivo del volo
- **stati finali**: la città di destinazione
- **costo delle azioni**: combinazione di costi monetari, comodità, tempo di attesa, qualità del volo...

Algoritmi di ricerca

> Un algoritmo di ricerca prende in input un problema di ricerca e restituisce una soluzione o un'indicazione di fallimento.

Gli algoritmi analizzati sovrappongono un albero di ricerca al grafo dello spazio degli stati, formando vari cammini dallo stato iniziale, cercando di trovare un cammino che raggiunge uno stato finale.

> **Da ricordare:** differenza tra spazio degli stati e albero di ricerca
>
> Spazio degli stati: descrive tutti i possibili stati dell'ambiente e le azioni che permettono di passare da uno all'altro.
> Albero di ricerca: cammini tra gli stati che raggiungono uno stato finale.

Best-First search

per espandere la frontiera sceglie il nodo che minimizza il valore di una funzione *f(n)*, mantiene una coda di priorità ordinata in base al valore di tale funzione.

Finchè la frontiera non è vuota, estrae il primo elemento dalla coda, controlla se è un nodo finale: se lo è lo restituisce altrimenti continua espandendo la frontiera.

```
function best-first-search(problem, f) returns a solution node or failure
    node = Node(state=problem.initial)
    frontier = priority queue ordered by f
    frontier.add(node)
    reached = lookup table
    reached[problem.initial] = node
    while not isEmpty(frontier)
        node = frontier.pop()
        if problem.isGoal(node.state) than return node
        for each child in expand(problem, node) do
            s = child.state
            if s is not in reached or child.pathCost < reached[s].pathCost then
                reached[s] = child
                frontier.add(child)
    return failure

    function expand(problem, node) yields nodes
        s = node.state
        for each action in problem.actions(s) do
            s'= problem.result(s, action)
            cost = node.pathCost + problem.actionCost(s, action, s')
            yiel Node(state=s', parent=node, action=action, pathCost=cost)
```

Cammini ridondanti

Quando generiamo un cammino all'interno di un grafo possono generarsi dei cammini ridondanti, di cui i cicli sono un caso particolare.

Eliminando la possibilità di generare cammini ridondanti velocizziamo la ricerca.

Chiamiamo **graph-search** un algoritmo di ricerca che controlla i cammini ridondanti e **tree-like search** uno che non lo fa.

Le modalità con le quali possiamo affrontare il problema sono:

- possiamo ricordare tutti gli stati raggiunti in precedenza in modo da mantenere solo il cammino più conveniente e evitare cammini ridondanti, scelta migliore quando ci sono molti cammini ridondanti
- quando il problema non ha cammini ridondanti o è improbabile che li abbia possiamo tralasciare il problema e non mantenere alcuna memoria
- possiamo fare un compromesso e controllare solo se ci sono cicli, visto che ogni nodo ha un genitore possiamo seguire la catena dei genitori e controllare se lo stato corrente è già apparso nel cammino; in questo modo non abbiamo bisogno di memoria aggiuntiva.

Performance di un algoritmo di ricerca

Possiamo misurare le performance di un algoritmo di ricerca in 4 modi:

- **completezza**: l'algoritmo trova la soluzione quando ne esiste una e riporta un fallimento in caso contrario; per essere completo, un algoritmo deve essere sistematico nel modo in cui esplora uno spazio degli stati infinito visitando, se necessario, tutti gli stati connessi a quello iniziale.
- **ottimalità di costo**: l'algoritmo trova la soluzione con costo minore
- **complessità temporale**: quanto ci mette l'algoritmo a trovare la soluzione, può essere misurata in secondi o in numero di azioni e stati considerati
- **complessità spaziale**: quanta memoria è necessaria per effettuare la ricerca

La complessità temporale e quella spaziale sono relazionate a qualche caratteristica dell'albero su cui si effettua la ricerca, generalmente si considerano il numero di nodi e di archi ma in molti problemi di intelligenza artificiale il grafo è solo implicito, quindi consideriamo la **profondità (d)** ovvero il numero di azioni nella soluzione ottimale e **il fattore di ramificazione (b)** ovvero il numero di successori di un nodo che devono essere considerati.

Strategie di ricerca non informata

Gli algoritmi di ricerca non informati sono quelli che non sanno quanto sono vicini alla soluzione ad ogni passo.

Breadth-first search

quando tutte le azioni hanno lo stesso costo possiamo usare l'algoritmo **breath-first search** che espande la radice, poi i suoi figli e così via.

Questo algoritmo è sistematico quindi completo. Possiamo implementarlo richiamando `bestFirstSeach` usando come *f(n)* la profondità del nodo (ovvero il numero di azioni necessarie per raggiungerlo).

Possiamo modificare bestFirstSearch per renderlo più efficiente in questo contesto:

- usando una coda FIFO al posto della coda di priorità in quanto i nodi aggiunti prima hanno *f(n)* minore di quelli aggiunti successivamente
- *reached* può essere un insieme invece che una mappa, in quanto quando abbiamo raggiunto un nodo non possiamo trovare un cammino migliore
- possiamo effettuare early goal test ovvero testare se un nodo è soluzione appena viene generato

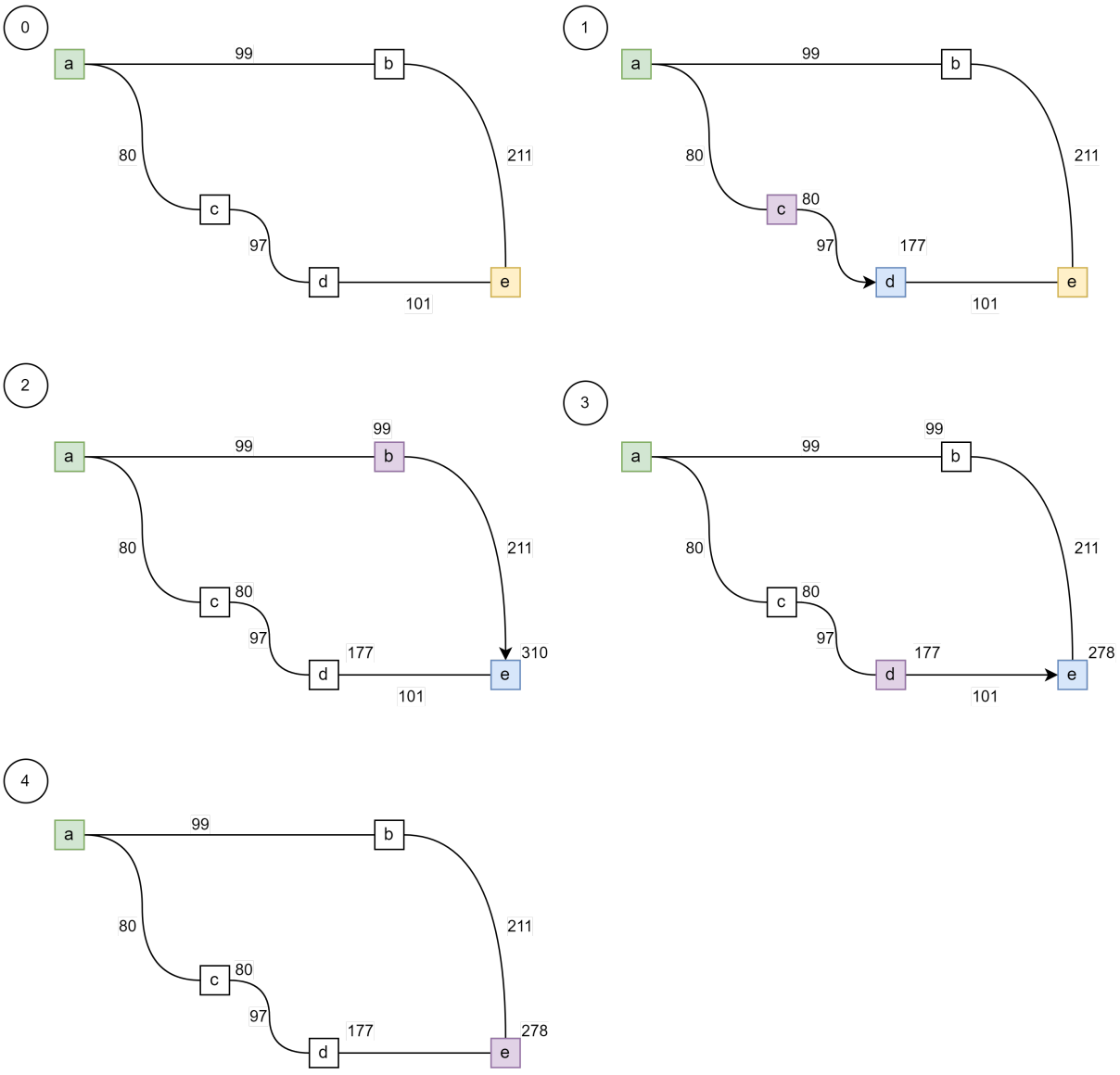
```
function breadthFirstSearch(problem) returns a solution node or failure
    node = Node(problem.initial)
    if problem.isGoal(node) then return node
    frontier = new FIFO queue
    frontier.add(node)
    reached = {problem.initial}
    while not isEmpty(frontier) do
        for each child in expand(problem, node) do
            s = child.state
            if problem.isGoal(s) then return child
            if s not in reached then
                reached.add(s)
                frontier.add(child)
    return failure
```

Questo algoritmo è ottimale nel costo quando le azioni hanno tutte lo stesso costo.

Il suo costo in tempo e spazio è $O(b^d)$ se la soluzione si trova a profondità d.

Dijkstra

Quando le azioni hanno costi diversi possiamo usare bestFirstSearch con f(n) = costo del cammino dalla radice al nodo n; questo algoritmo è detto di Dijkstra o **ricerca costo-uniforme**.



1. scelgo il nodo meno costoso (c) e lo espando, aggiungo d alla frontiera con il costo di 177
2. scelgo il nodo meno costoso della frontiera (b) e lo espnado, aggiungo e alla frontiera con il costo di 310
3. scelgo il nodo meno costoso dalla frontiera (d) e lo espando, aggiungo e alla frontiera con il costo di 278
4. scelgo il nodo meno costoso dalla frontiera (e) che è soluzione e quindi mi fermo

Se il costo della soluzione è C* e il costo minimo di un arco (azione) è ϵ allora abbiamo C^*/ϵ livelli e il costo dell'algoritmo è $O(b^{C^*/\epsilon})$.

È completo e ottimale nel costo.

Depth-first search

La DFS espande il nodo più profondo della frontiera. Non ha bisogno di mantenere i nodi visitati in quanto espande la frontiera fino a che il nodo ha figli e poi torna indietro fino al nodo più vicino con figli non visitati.

Non è ottimale nel costo, restituisce la prima soluzione anche se non è la meno costosa.

Per spazi degli stati finiti è efficiente e completo.

Viene usato perché ha bisogno di poca memoria = $O(bm)$ dove m è la massima profondità dell'albero.

Iterative deepening

Possiamo limitare la DFS imponendole un limite di profondità, ogni volta che espandiamo la frontiera controlliamo se siamo a una profondità maggiore del limite e in caso interrompiamo la ricerca. Questo approccio si chiama **depth-limited search**.

Possiamo combinare la BFS e la depth-limited search per creare un algoritmo che non abbia bisogno di troppa memoria e che sia completo.

```
function iterativeDeepeningSearch(problem) returns a solution node or failure
  for depth = 0 to INF
```

```
        result = depthLimitedSearch(problem, depth)
        if result != cutoff then return result

function depthLimitedSearch(problem, d)
    frontier = new LIFO queue
    frontier.add(problem.initial)
    result = failure
    while not isEmpty(frontier) do
        node = forntier.pop()
        if problem.isGoal(node) then return node
        if node.depth() > l then
            result = cutoff
        else if not isCycle(node) do
            for each child in expand(problem, node) do
                frontier.add(child)
    return result
```

La complessità temporale è $O(b^d)$ quando esiste una soluzione, $O(b^m)$ altrimenti.

Comparazione

criterio	BFS	uniform cost	DFS	iterative deepening
competo	si	si	no	si
ottimale in costo	si	si	no	si
tempo	$O(b^d)$	$O(b^{C^*/\epsilon})$	$O(b^m)$	$O(b^d)$
spazio	$O(b^d)$	$O(b^{C^*/\epsilon})$	$O(bm)$	$O(bd)$

Ricerca informata

Gli algoritmi di ricerca informata utilizzano delle informazioni relative al dominio del problema per approssimare la distanza dalla soluzione.

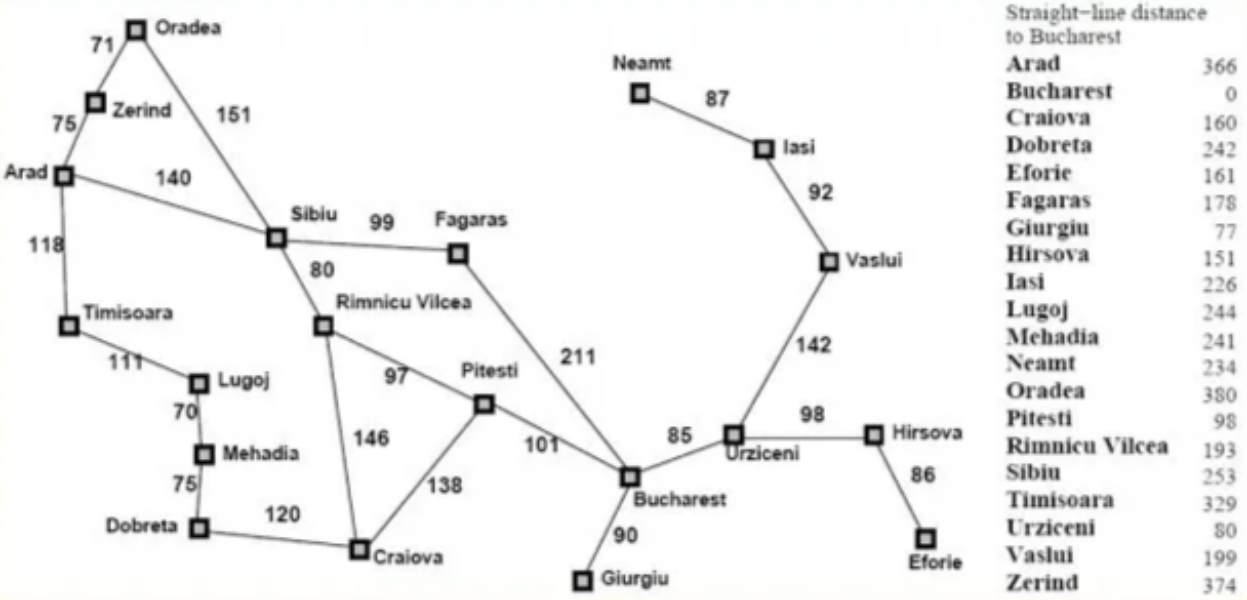
Queste informazioni hanno la forma di una funzione detta **euristica** denotata *h(n)*

h(n) = costo stimato del cammino migliore da n allo stato finale

Greedy best first search

Questo algoritmo è una versione modificata di best-first search in cui usiamo h(n) = distanza in linea d'aria tra n e la soluzione, al posto di f(n).

Per sapere che la distanza in linea d'aria è una buona euristica abbiamo bisogno di conoscenza dell'ambiente.



Nel problema della ricerca del percorso migliore da Arad a Bucarest: il primo nodo ad essere espnaso sarà Sibiu, poi Fagaras, poi Bucarest che è la destinazione.

Il risultato restituito non è il migliore, ma è quello che seleziona ad ogni passo la sottosoluzione che sembra migliore.

A-star

Algoritmo di ricerca informata implementato come un best-first search con $f(n) = g(n) + h(n)$

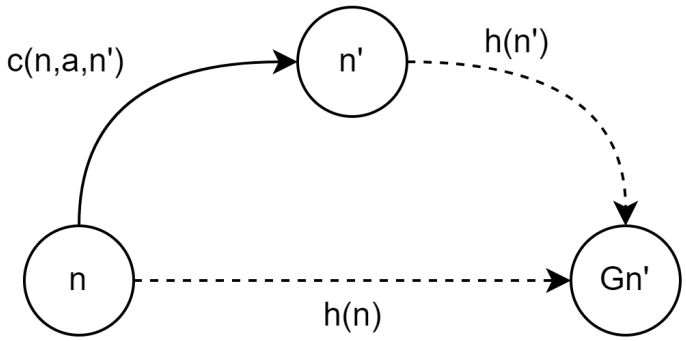
dove g è il costo del cammino dal nodo iniziale a n e h è la stima del costo del percorso da n a un nodo finale.

L'algortimo A-star è completo, il fatto che sia ottimale nel costo dipende da alcune proprietà dell'euristica.

La proprietà principale che deve avere l'euristica è **l'ammissibilità**, un'euristica ammissibile non sovrastima mai il costo del raggiungimento dell'obiettivo.

Con un'euristica ammissibile, A-star è ottimale nel costo.

Una proprietà più forte per l'euristica è la **consistenza**, un'euristica è consistente se per ogni nodo *n* e per ogni successore di n *n'* generato da un'azione *a* abbiamo: $h(n) \leq c(n, a, n') + h(n')$
graficamenente:



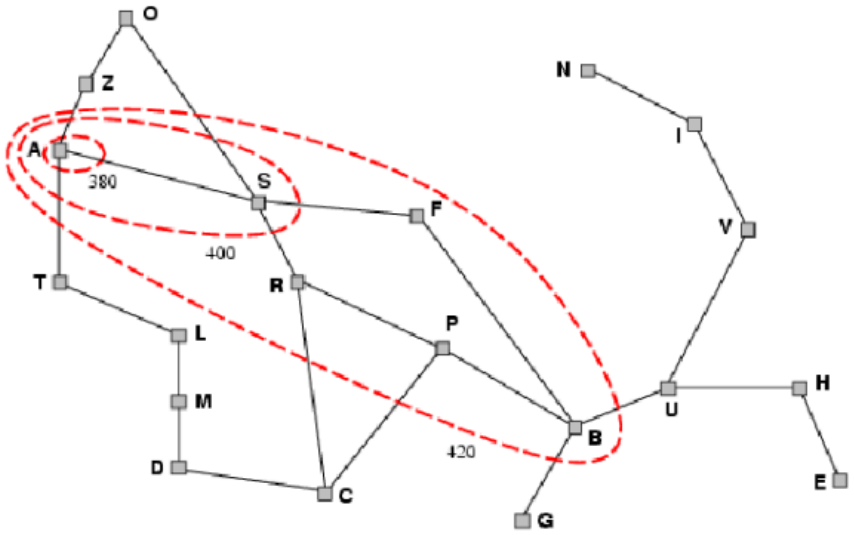
ovvero la consistenza è espressa da una forma di disequanzione triangolare, che afferma che un lato del triangolo non può essere maggiore della somma degli altri due.

> **Da ricordare:** rapporto tra euristiche consistenti e ammissibili
>
> ogni euristica consistente è ammissibile ma non è vero il vice versa

Contorni di ricerca

Possiamo disegnare delle regioni sullo spazio degli stati che contengono i nodi che hanno lo stesso valore di f(n).

Ad esempio, per A-star avremo delle regioni ovali etichettate con il valore di f(n) che hanno tutti i nodi in esse compresi.



Euristiche inammissibili

Se utilizziamo A-star con un'euristica inammissibile, che può sovrastimare, possiamo trovare delle soluzioni soddisfacenti in meno tempo.

In questo caso abbiamo $f(n) = g(n) + W * h(n)$ con W>1

IDA* = iterative deepening A*

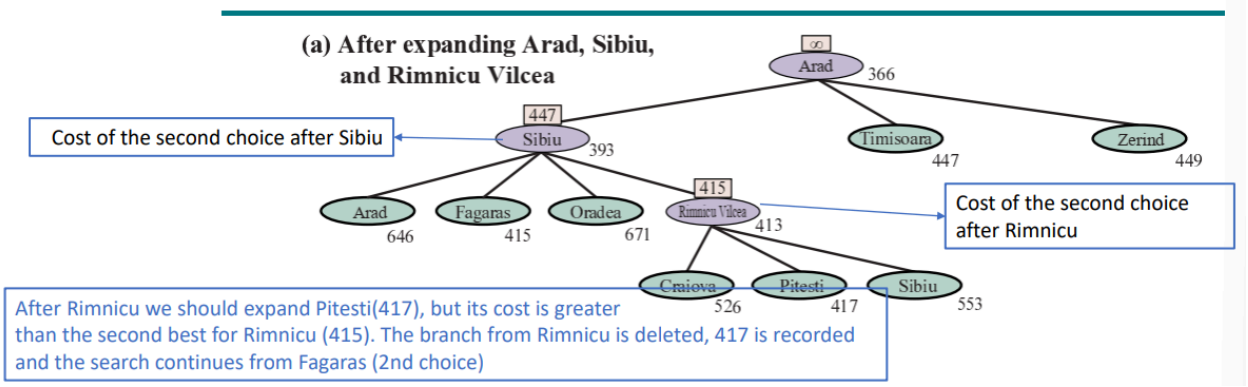
Ogni iterazione dell'algoritmo è una DFS che tiene traccia del costo $f(n) = g(n) + h(n)$ di ogni nodo generato.

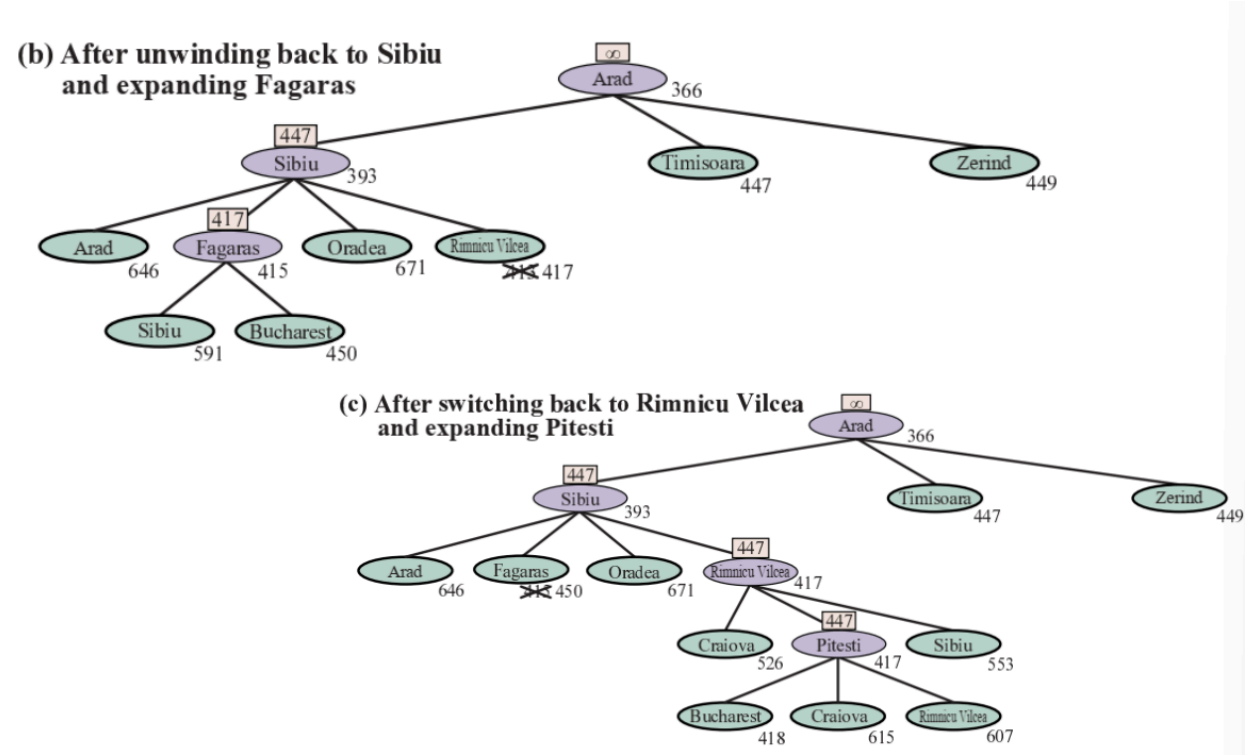
Quando viene generato un nodo n' tale che f(n')>f(n) dell'iterazione, si prende f(n') come cutoff per l'iterazione successiva.

Questo algoritmo è efficiente quando i nodi sono equamente distribuiti nei livelli di cutoff. È inefficiente quando ci sono pochi nodi in ogni livello.

RBFS = recursive best-first search

Algoritmo che lavora come una DFS ricorsiva, ma non continua ad esplorare un cammino fino al nodo senza figli, mantiene due possibili direzioni e il loro costo, quando la prima direzione diventa peggiore della seconda l'algoritmo esplora la seconda.





Questo algoritmo è ottimale nel costo quando l'euristica è ammissibile.

Ricerca avversaria e giochi

Esploriamo gli ambienti competitivi in cui due o più agenti hanno obiettivi in conflitto e generano problemi di ricerca avversaria.

Giochi a somma zero

I giochi a somma zero sono quelli in cui un giocatore cerca di massimizzare un valore e l'opponente cerca di mminimizzarlo; non possono concludersi in una vittoria per entrambi i giocatori.

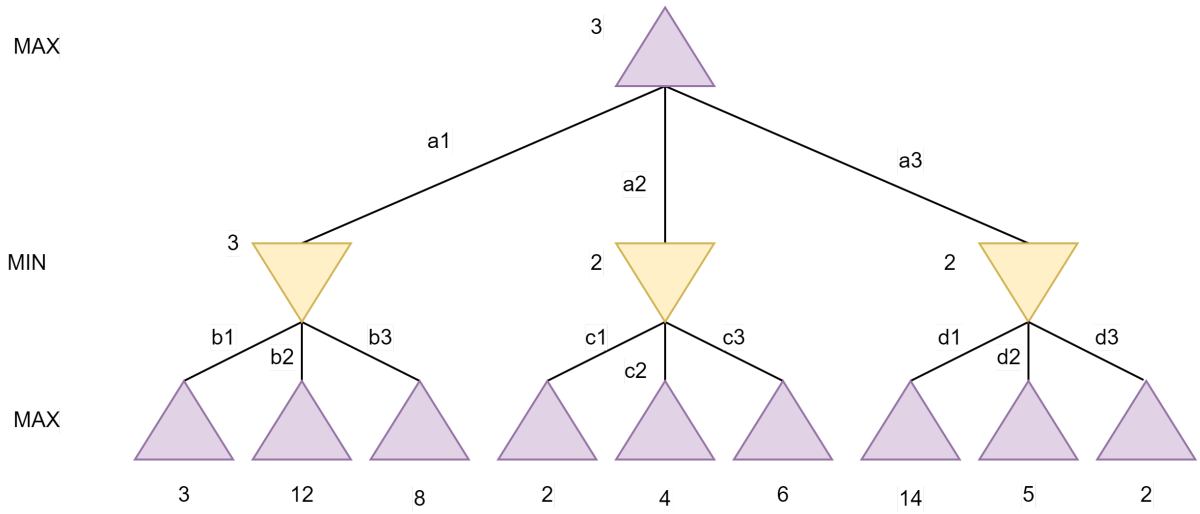
Formalmente possiamo definire un gioco come:

- s0: stato iniziale
- TO-MOVE(s): restiotuisce il giocatore che deve giocare nello stato s
- ACTIONS(s): l'insieme delle azioni possibili nello stato s
- RESULT(s,a): restituisce lo stato a cui si arriva applicando l'azione a nello stato s
- IS-TERMINAL(s): ci dice se lo stato è finale e quindi se il gioco è concluso
- UTILITY(s,p): restiuisce il punteggio del giocatore p quando il gioco termina nello stato s

La strategia ottimale può essere determinata trovando il valore MINMAX di ogni stato nell'albero. Il valore minmax è l'utilità per un giocatore di trovarsi in quello stato.

Per il giocatore MAX abbiamo:

```
minmax(s) =
   if (isTerminal(s)) { utility(s,MAX)}
   if (toMove(s) == MAX) {max for a in Actions(s) of minmax(result(s,a))}
   if (toMove(s) == MIN) {min for a in Actions(s) of minmax(result(s,a))}
```



Algoritmo di ricerca minmax

A partire dalla funzione minmax(s) possiamo scriver un algoritmo che trova la migliore mossa per MAX provando tutte le aioni e scegliendo quella che ha stato risultante con il valore di minmax maggiore.

Questo algoritmo è ricorsivo e attaversa tutto il grafo arrivando alle foglie e poi torna indietro per calcolare i valori minmax di tutte le azioni.

```
function minmaxSearch(game, state) returns an action
    player = game.toMove(state)
    value, move = maxValue(game, state)
    return move

function maxValue(game, state) returns a (utility, move) pair
    if game.isTerminal(state) then return game.utility(state, player), null
    v, move = -INF
    for each a in game.actions(state) do
        v2, a2 = minValue(game, game.result(state, a))
        if v2>v then
            v, move = v2, a
    return v, move

function minValue(game, state) returns a (utility, move) pair
```

```
if game.isTerminal(state) then return game.utility(state, player), null
v, move = +INF
for each a in game.actions(state) do
    v2, a2 = maxValue(game, game.result(state, a))
    if v2<v then
        v, move = v2, a
return v, move
```

La complessità spaziale è $O(bm)$ e quella temporale $O(b^m)$ dove b è il numero di mosse possibili a ogni punto e m è la massima profondità dell'albero.

alpha-beta pruning

Durante l'esecuzione di minmaxSearch ci sono percorsi che non vale la pena esplorare perché abbiamo già un'alternativa migliore, quindi possiamo tagliare via parti dell'albero e risparmiare tempo.

Consideriamo un nodo n in qualsiasi punto dell'albero raggiungibile dal giocatore; se il giocatore ha una scelta migliore allo stesso livello o a un livello precedente allora il giocatore non si muoverà mai in n quindi possiamo rimuoverlo

Machine learning

> **Obiettivo sezione:** descrivere gli agenti che possono migliorare il loro comportamento tramite lo studio delle esperienze passate e predizioni sul futuro.

Un agente impara se migliora le sue performance dopo aver compiuto osservazioni sul mondo. Quando l'agente è un computer chiamiamo il processo di apprendimento **machine learning**; l'agente osserva alcuni dati, crea un modello basato su questi dati e usa il modello sia come ipotesi che come soluzione per il problema.

Vogliamo che l'agente sia capace di imparare per adattarsi alle variazioni del problema che deve risolvere e perché ci sono problemi per cui non possiamo

direttamente scrivere un algoritmo (es riconoscimento facciale).

I problemi di apprendimento si dividono in:

- **classificazione**: l'output è un elemento di un insieme finito di valori (vero/falso o soleggiato/piovoso/nuvoloso)
- **regressione**: l'output è un numero

Tipologie di apprendimento

Le principali tipologie di apprendimento sono:

- **apprendimento supervisionato**: l'agente osserva coppie di input-aoutput e impara una funzione che mappa gli input negli output. Es: input = immagine di un autobus, output="bus"
  questo tipo di output è detto
  label
- **apprendimento non-supervisionato**: l'agente impara a riconscere un pattern nei dati senza un particolare feedback. Il compito principale per questo tipo di apprendimento è il **clustering**, ovvero il raggruppamento di grandi moli di dati.
- **apprendimento per rinforzo**: l'agente apprende tramite una serie di rinforzi positivi e negativi; quando gli viene fornito un rinforzo sta a lui decidere quali azioni lo hanno prodotto e quindi quali azioni ripetere (se il rinforzo era positivo) e quali evitare (se il rinforzo era negativo).

Apprendimento supervisionato

Formalmente, l'obiettivo dell'apprendimento supervisionato è:

dato un **training set** di N coppie input-output di esempio: $(x_1, y_1), (x_2, y_2)...$ generati da una funzione sconosciuta $f(x) = y$ trovare una funzione *h* che approssimi il comportamento di *f*.

La funzione *h* è detta **ipotesi** sul mondo. È tratta da uno **spazio delle ipotesi** *H* di possibili funzioni.

Ogni output y_i è detto verità fondamentale (**ground truth**), la vera risposta che chiediamo al modello di predire.

Per scegliere lo spazio delle ipotesi possiamo effettuare un'**analisi dati esploratoria** per esaminare i dati con test statistici e di visualizzazione per ottenere qualche intuizione riguardo a quale spazio delle ipotesi possa essere appropriato.

La bontà dell'ipotesi non si misura in base a quanto bene approssima gli output del training set bensì in base a come si comporta con dati mai visti prima (detti **test set**). Diciamo che h generalizza bene se predice accuratamente l'output del test set.

Un modo per analizzare gli spazi delle ipotesi è quello di considerare il loro **bias** e la **varianza** che producono.
Per bias intendiamo la tendenza di un'ipotesi predittiva a deviare dal valore atteso, quando mediato su diversi training set.
Per varianza si intende l'influenza delle variazioni dei dati sull'ipotesi, ovvero se a piccole variazioni dei dati corrispondono grandi variazioni dell'ipotesi.

È necessario fare un compromesso tra bias e varianza ovvero tra ipotesi più complesse con meno bias e ipotesi più generiche con bassa varianza.

Underfitting/overfitting = diciamo che un'ipotesi è underfitting se fallisce nel trovare un pattern nei dati; diciamo che un'ipotesi è overfitting quando si preoccupa troppo al particolare data set e performa male sui dati di test.

Classificazione

Classificare significa assegnare oggetti a una di molte categorie predefinite.

I dati in input a un task di classificazione sono una collezione di record. Ogni record (anche detto istanza o esempio) è caratterizzato da una tupla (**x**,y) dove x è l' insieme degli attribuiti e y è la label della classe.

Gli attributi possono essere sia discreti che continui mentre y deve assumere valori discreti (questo differenzia la classificazione dalla regressione).

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hiber-nates	Class Label
human	warm-blooded	hair	yes	no	no	yes	no	mammal
python	cold-blooded	scales	no	no	no	no	yes	reptile
salmon	cold-blooded	scales	no	yes	no	no	no	fish
whale	warm-blooded	hair	yes	yes	no	no	no	mammal
frog	cold-blooded	none	no	semi	no	yes	yes	amphibiar
komodo dragon	cold-blooded	scales	no	no	no	yes	no	reptile
bat	warm-blooded	hair	yes	no	yes	yes	yes	mammal
pigeon	warm-blooded	feathers	no	no	yes	yes	no	bird
cat	warm-blooded	fur	yes	no	no	yes	no	mammal
leopard shark	cold-blooded	scales	yes	yes	no	no	no	fish
turtle	cold-blooded	scales	no	semi	no	yes	no	reptile
penguin	warm-blooded	feathers	no	semi	no	yes	no	bird
porcupine	warm-blooded	quills	yes	no	no	yes	yes	mammal
eel	cold-blooded	scales	no	yes	no	no	no	fish
salamander	cold-blooded	none	no	semi	no	yes	yes	amphibiar

In questa tabella si classificano delle caratteristiche di alcuni esseri viventi per individuare se sono mammiferi, pesci, rettili o anfibi.

Lo scopo finale è quello di saper predire la classe di record sconosciuti.

Classificare significa imparare un funzione f detta funzione target che mappa ogni insieme di attributi x in una etichetta di classe y.

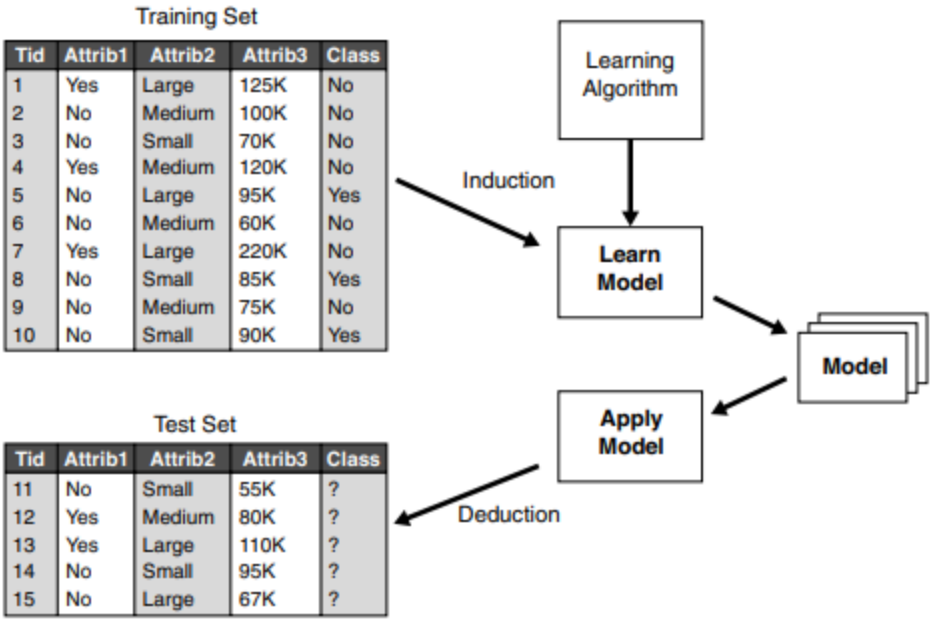
La funzione target è anche detta modello di classificazione; un modello di classificazione serve a:

- definire quali caratteristiche fanno in modo che un determinato oggetto venga inserito in una certa classe (explanatory modeling)
- Predire la classe di record sconosciuti (predictive modeling)

Come risolvere problemi di classificazione

Una tecnica di classificazione (classifier) è un parroccio sistematico per costruire modelli di classificazione (funzione f) a partire da un input data set.

Ogni tecnica include un algortimo di apprendimento che identifica il modello che descrive al meglio la relazione tra l'insieme degli attributi e la label della classe che gli è stata assegnata.



Le performance di un modello di classificazione si misurano in base a quante sono le label corrette assegnate ai record del test set. Per fare ciò si costruisce una **confusion matrix**, che per un problema di classificazione binaria può essere simile alla seguente:

		Predicted Class	
		$Class = 1$	$Class = 0$
Actual Class	$Class = 1$	f_{11}	f_{10}
	$Class = 0$	f_{01}	f_{00}

ogni elemento della tabella f_{ij} denota il numero di record della calsse i a cui è stata assegnata la classe j. Quindi il numero di record corretti è dato dalla somma degli f_{ii}. Possiamo definire l'**accuratezza** del modello:

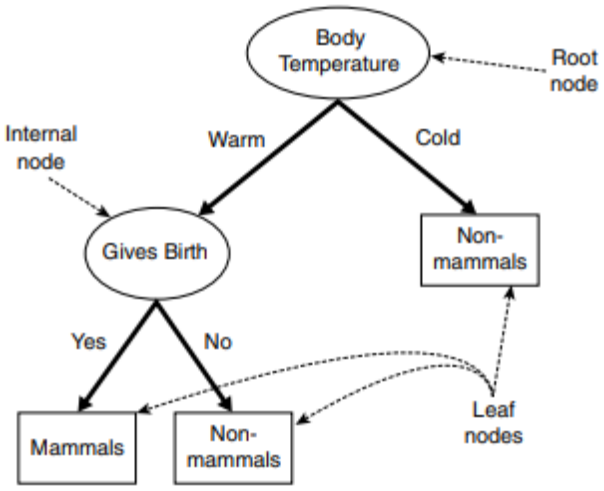
$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} = \frac{f_{11} + f_{00}}{f_{11} + f_{10} + f_{01} + f_{00}}.$$

e il suo **tasso di errore:**

$$\text{Error rate} = \frac{\text{Number of wrong predictions}}{\text{Total number of predictions}} = \frac{f_{10} + f_{01}}{f_{11} + f_{10} + f_{01} + f_{00}}.$$

Decision tree

Una tecnica di classficazione utilizza un albero decisionale per classificare i record, ponendo domande sulle loro caratteristiche.



Le foglie dell'albero sono le possibili etichette.

Dobbiamo capire: quali domande porre e quando, ovvero ad ogni passo della costruzione del nostro albero dobbiamo selezionare una caratteristica, a analizzarne i valori e suddividere i record in sottoinsiemi omogeni nel valore (o nel range di valori) di quella caratteristica.

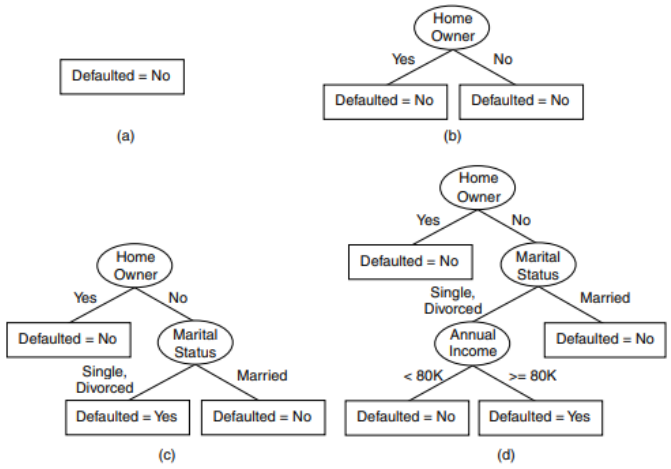
Algoritmo di Hunt

Sia D_t il set dei training records e $y = \{y_1, ..., y_c\}$ l'insieme delle label.
L'algoritmo procede così:

1. se i record in D_t hanno tutti la stessa classe allora il nodo t è puro ed è una foglia dell'albero etichettata con y_t
2. se D_t contiene record che appartengono a più classi, viene selezionato una **attribute test condition** che partiziona l'insieme in insiemi più piccoli. Viene creato un nodo per ogni risposta del test. Si applica l'algoritmo a tali nodi.

Esempio:

	binary	categorical	continuous	class
Tid	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes



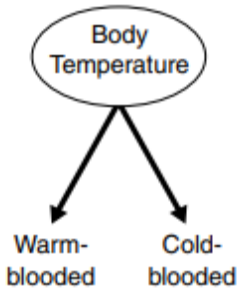
Come esprimere le condizioni di test

Le condizioni di test e le possibili risposte variano in base al tipo di attributo su cui si crea la condizione di test.

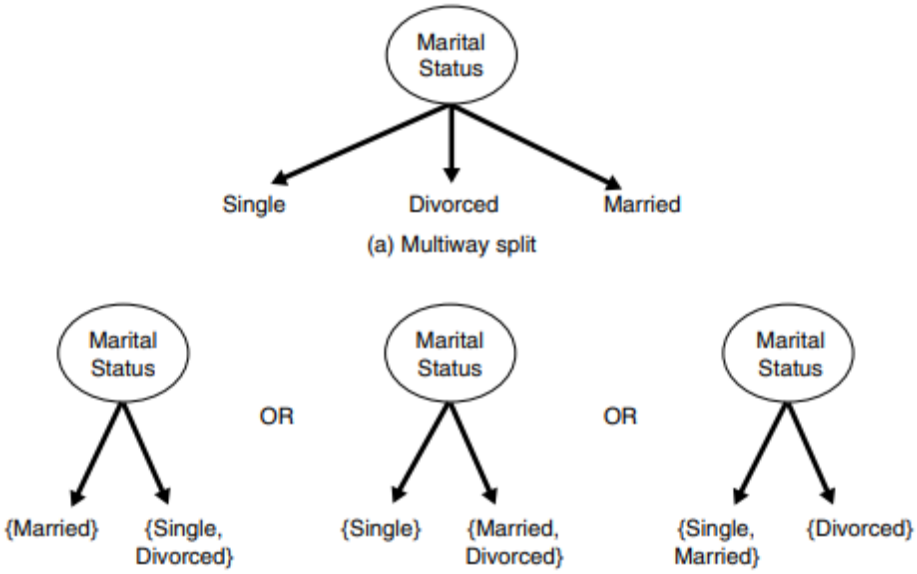
Gli attributi possono essere suddivisi in:

- binari
- nominali
- ordinali: piccolo, medio, grande
- continui

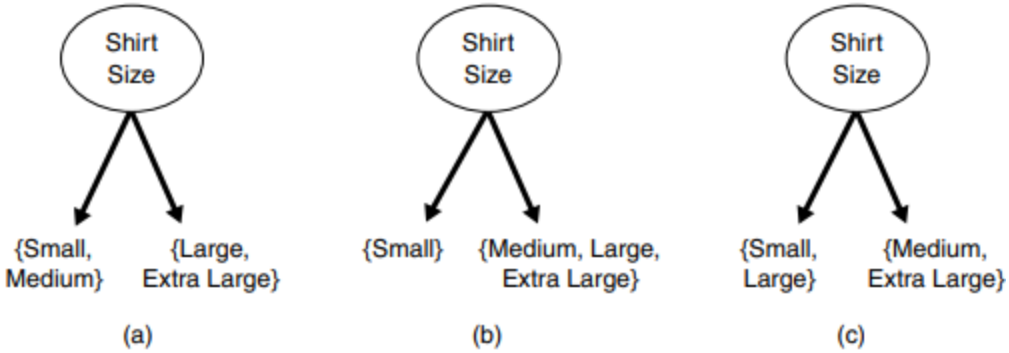
Attributi Binari: Una condizione di test su un attributo binario causa la creazione di due nodi figli



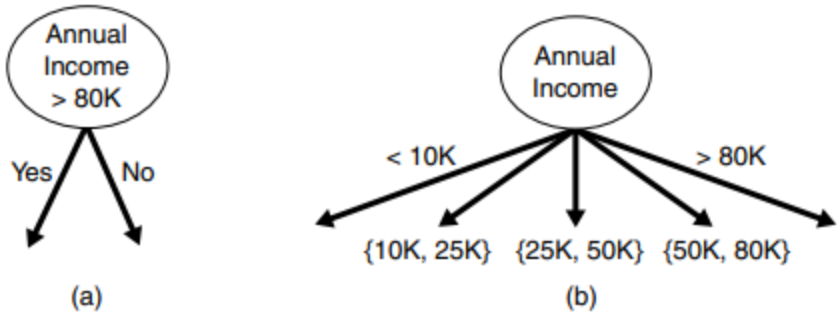
Attributi Nominali: Un attributo nomila può avere molti valori (es: stato civile=single, sposato, divorziato) quindi il numero di nodi figli generati da una condizione su questo tipo di attributo dipende dal numero di possibili valori e sul modo in cui si decide di raggrupparli.



Attributi Ordinali: Gli attributi ordinali si comportano come i nominali con la differenza che possiamo raggruppare solo se i gruppi rispettano l'ordinamento. Nell'esempio l'opzione c è sbagliata.



Attributi Continui: Per gli attributi continui le espressioni di test possono essere espresse come comparazioni con un valore $A \leq v$ oppure come range :



Come scegliere la condizione di test

Alcune condizioni di test sono migliori di altre, in termini di purezza di dati che producono.

Misuriamo la bontà di una condizione in base alla distribuzione dei record nelle classi prima e dopo lo split.

Denotiamo con $p(i|t)$ la frazione di record che appartengono alla classe i nel nodo t.

Misuriamo la purezza dei record tramite l'indice Gini:

$$\text{Gini(t)} = 1 - \sum_{i=0}^{c-1} [p(i|t)]^2$$

Node N_1	Count
Class=0	0
Class=1	6

$\text{Gini} = 1 - (0/6)^2 - (6/6)^2 = 0$
$\text{Entropy} = -(0/6)\log_2(0/6) - (6/6)\log_2(6/6) = 0$
$\text{Error} = 1 - \max[0/6, 6/6] = 0$

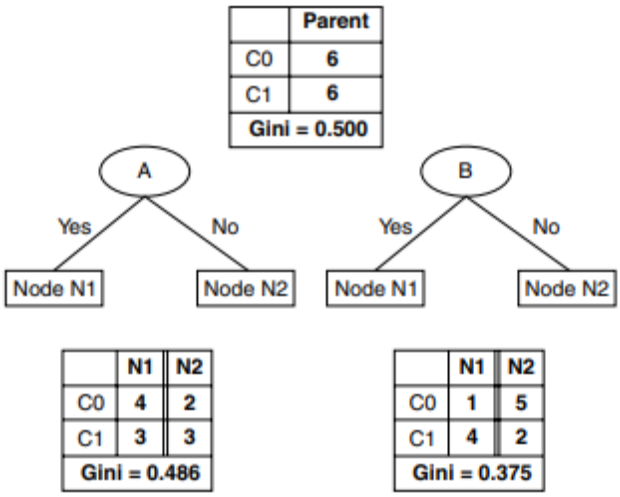
Node N_2	Count
Class=0	1
Class=1	5

$\text{Gini} = 1 - (1/6)^2 - (5/6)^2 = 0.278$
$\text{Entropy} = -(1/6)\log_2(1/6) - (5/6)\log_2(5/6) = 0.650$
$\text{Error} = 1 - \max[1/6, 5/6] = 0.167$

Node N_3	Count
Class=0	3
Class=1	3

$\text{Gini} = 1 - (3/6)^2 - (3/6)^2 = 0.5$
$\text{Entropy} = -(3/6)\log_2(3/6) - (3/6)\log_2(3/6) = 1$
$\text{Error} = 1 - \max[3/6, 3/6] = 0.5$

Per determinare quanto è buona una condizione di split confrontiamo l'indice di gini del nodo prima di suddividere con quella dei figli, la condizione che produce figli con minor indice di gini vince (**information gain**).



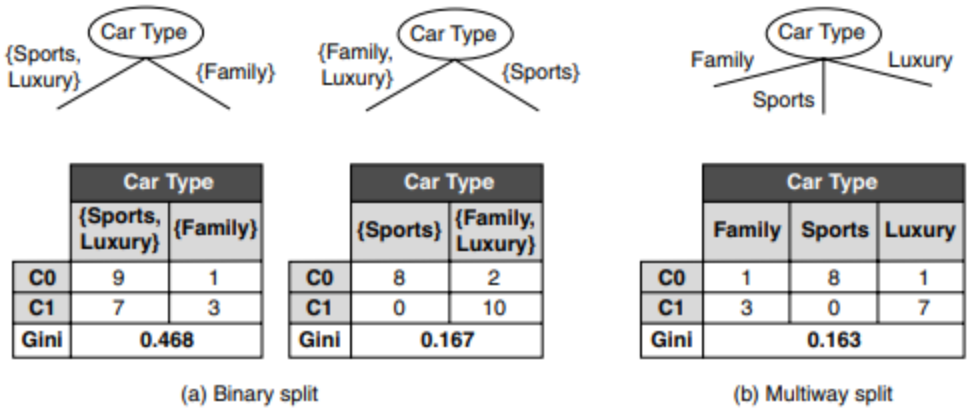
In questo esempio preferiamo la condizione B.

▼ Conti

Per la decisione A:

$$\text{Gini}(\text{N1}) = 1 - (4/7)^2 - (3/7)^2 = 0.489$$
$$\text{Gini}(\text{N2}) = 1 - (2/5)^2 - (3/5)^2 = 0.48$$
$$\text{Gini} = (7/12) * 0.489 + (5/12) * 0.48 = 0.486$$

Altro esempio:



Attributi continui

Class	No		No		No		Yes		Yes		Yes		No		No		No		No	
	Annual Income																			
Sorted Values →	60		70		75		85		90		95		100		120		125		220	
Split Positions →	55		65		72		80		87		92		97		110		122		172		230	
	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>
Yes	0	3	0	3	0	3	0	3	1	2	2	1	3	0	3	0	3	0	3	0	3	0
No	0	7	1	6	2	5	3	4	3	4	3	4	3	4	4	3	5	2	6	1	7	0
Gini	0.420		0.400		0.375		0.343		0.417		0.400		0.300		0.343		0.375		0.400		0.420	

Overfitting

Gli errori commessi da un modello di classificazione possono essere divisi in: **training errors** e **generalization errors.** Gli errori di addestramento sono quelli commessi sui dati di addestramento. Gli errori di generalizzazione sono quelli commessi su dati mai visti.

Un buon modello deve avere pochi errori di addestramento e pochi errori di generalizzazione. Un modello che si adatta troppo bene ai dati di addestramento può avere errori di generalizzazione maggiori → **overfitting.**

(Si ha **underfitting** quando il modello non ha ancora appreso la struttura dei dati.)

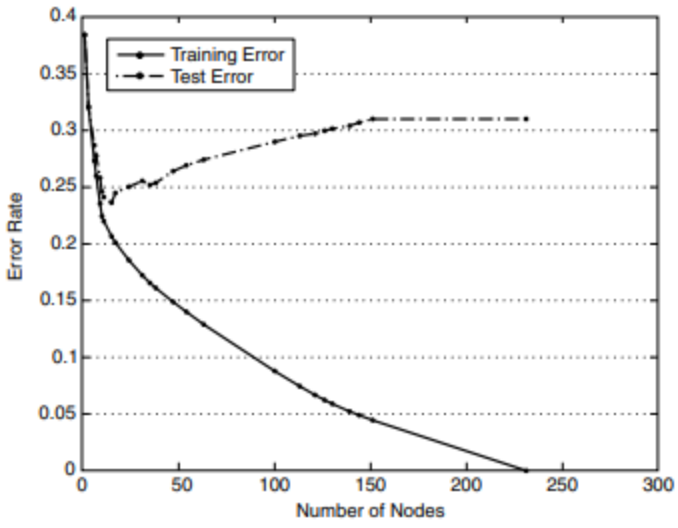


Figure 4.23. Training and test error rates.

La complessità del modello incide fortemente sul possibile overfitting, modelli più complessi hanno maggiori probabilità di generare overfitting perché la

complessità (i nodi in più) possono essere stati generati solo per aderire meglio ai dati di addestramento.

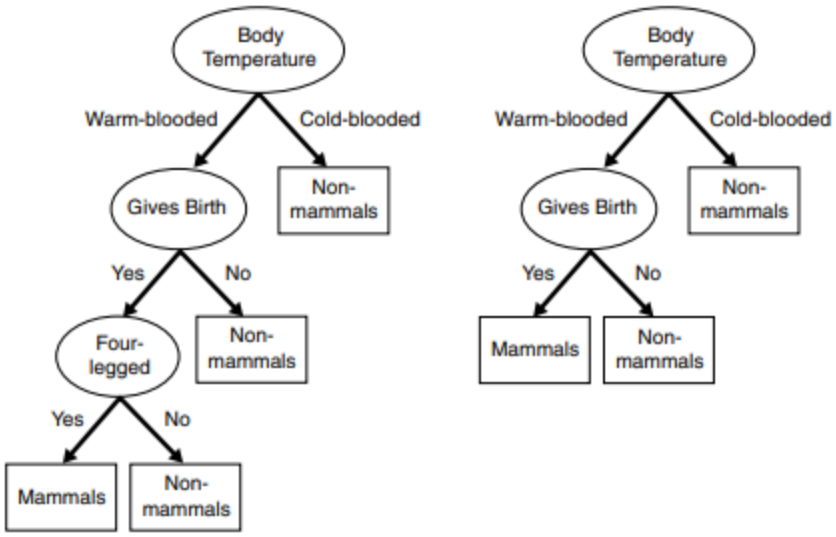
Le principali cause di overfitting sono:

- presenza di rumore (record nel training set che non sono stati classificati correttamente)
- mancanza di esempi rappresentativi

Overfitting per presenza di rumore

Name	Body Temperature	Gives Birth	Four-legged	Hibernates	Class Label
porcupine	warm-blooded	yes	yes	yes	yes
cat	warm-blooded	yes	yes	no	yes
bat	warm-blooded	yes	no	yes	no*
whale	warm-blooded	yes	no	no	no*
salamander	cold-blooded	no	yes	yes	no
komodo dragon	cold-blooded	no	yes	no	no
python	cold-blooded	no	no	yes	no
salmon	cold-blooded	no	no	no	no
eagle	warm-blooded	no	no	no	no
guppy	cold-blooded	yes	no	no	no

Quando nel trainig set ci sono record etichettati male è possibile che il modello aggiunga dei nodi all'albero di decisione per adeguarsi alla discrepanza. Ciò produce errori di generalizzazione.



Overfitting dovuto a mancanza di dati rappresentativi

L'overfitting si può generare anche quando non ci sono abbastanza record nel training set per fare in modo che il modello possa capire come generalizzare.

Stima degli errori di generalizzazione

Possiamo stimare l'errore di generalizzazione di un albero per capire qual è il miglior grado di complessità a cui arrivare.

Come primo approccio possiamo calcolare l'errore di generalizzazione come la somma degli errori sul training set e di un termine di penalità per la complessità del modello:

$$e_g(T) = \frac{\sum_{i=1}^{k}[e(t_i) + \Omega(t_i)]}{\sum_{i=1}^{k} n(t_i)} = \frac{e(T) + \Omega(T)}{N_t},$$

dove:

- $e(t_i)$ è il numero di record classificati male dal nodo t_i
- $\Omega(t_i)$ è la penalità associata al nodo t_i
- N_t è il numero totale di record nel training set

Gestione dell'overfitting

Le strategie per la gestione dell'overfitting sono:

- **pre-pruning**: si utilizza una condizione di stop più restrittiva
- **post-pruning**: si determina l'albero e poi si procede e eliminare nodi dal basso

Class Imbalance problem

Gli insiemi di dati con classi distribuite in maniera sbilanciata sono molto frequenti in determinati contesti (es: numero di transazioni fraudolente su carte di credito = 1/100) e in molti di questi contesti è importante classificare bene i record con la classe meno frequente.

In questi casi le metriche come l'accuratezza non sono affidabili (es: una algoritmo che classifca tutte le transazioni come legittime ha un'accuratezza del 99%);

Metriche alternative

Per i problemi di classificazione binaria, etichiettiamo con + la classe rare e con il - la classe frequente:

		Predicted Class	
		+	−
Actual Class	+	f_{++} (TP)	f_{+-} (FN)
	−	f_{-+} (FP)	f_{--} (TN)

- TP= veri positivi
- FN = falsi negativi (numero di record positivi classificati negativi)
- FP = falsi positivi (numero di record negativi classificati positivi)
- TN = veri negativi

Introduciamo due nuove metriche:

$$\text{Precision, } p = \frac{TP}{TP + FP}$$
$$\text{Recall, } r = \frac{TP}{TP + FN}$$

La **precisione** misura la frazione di record realmente positivi tra i classificati positivi (TP/Predicted +)
Il
recall misura la frazione di esempi positivi classificati come tali (TP/Actual +)

L'obiettivo è costruire un modello che massimizzi sia la precisione che il recall.

La precisione e il recall possono essere sintetizzate in una terza metrica: la **misura F1**

$$F_1 = \frac{2rp}{r + p} = \frac{2 \times TP}{2 \times TP + FP + FN}$$

La F1 è la media armonica tra precisione e recall e tende a essere vicina alla più piccola delle due.

A

		PREDICTED CLASS	
		Class=Yes	Class=No
ACTUAL CLASS	Class=Yes	40	10
	Class=No	10	40

Precision (p) = 0.8
Recall (r) = 0.8
F - measure (F) = 0.8
Accuracy = 0.8

B

		PREDICTED CLASS	
		Class=Yes	Class=No
ACTUAL CLASS	Class=Yes	40	10
	Class=No	1000	4000

Precision (p) =~ 0.04
Recall (r) = 0.8
F - measure (F) =~ 0.08
Accuracy =~ 0.8

ROC = receiver operating characteristic curve

Tale curva è un metodo grafico per visualizzare il compromesso tra TPR (true positive rate) e FPR (false positive rate).

Mettiamo FPR come asse x e TPR come asse y, grafichiamo le performance dei modelli del classificatore:

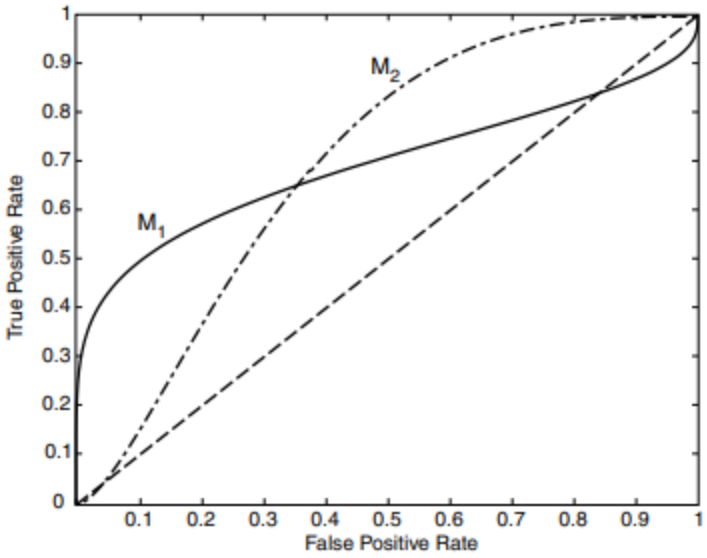


Figure 5.41. ROC curves for two different classifiers.

I punti critici sono:

- (TPR=0, FPR=0): il modello classifica tutto come negativo
- (TPR=1, FPR=1): il modello classifica tutto come positivo
- (TPR=1, FPR=0): ideale

Il modello ideale si avvicina all'angolo upper-left mentre un modello che sceglie randomicamente connette i punti (0,0) e (1,1).

La curva ROC viene utilizzata per comparare le performance di classificatori diversi.

Artificial neural network (ANN)

Una rete neurale artificiale è un modello ispirato dalla struttura e dalle funzioni delle reti neurali umane.

Una ANN è costituita da nodi e archi diretti.

Percettrone

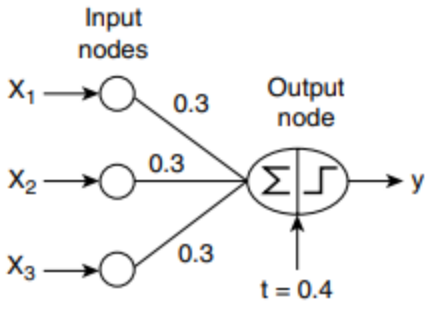
Il percettrone è il modello ANN più semplice, è costituito da due tipi di nodi: i nodi input e il nodo output. I nodi input rappresentano gli attributi dei record, il nodo output rappresenta l'output del modello.

Ogni nodo input è connesso tramite un arco pesato al nodo output. Il peso dell'arco rappresenta la forza della connessione sinaptica delle reti neurali umane.

Allenare un percettroce significa adattare i pesi degli archi in modo che il modello si adatti alla relazione input-output dei dati.

X_1	X_2	X_3	y
1	0	0	−1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	−1
0	1	0	−1
0	1	1	1
0	0	0	−1

(a) Data set.



(b) Perceptron.

Figure 5.14. Modeling a boolean function using a perceptron.

Il nodo output è un dispositivo matematico che calcola la somma pesata degli input, sottrae il termine di bias e produce un input applicando la funzione segno.

Nell'esempio abbiamo:

$$\hat{y} = \begin{cases} 1, & \text{if } 0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4 > 0; \\ -1, & \text{if } 0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4 < 0. \end{cases}$$

in generale:

$$\hat{y} = sign[w_d x_d + w_{d-1} x_{d-1} + \ldots + w_1 x_1 + w_0 x_0] = sign(\mathbf{w} \cdot \mathbf{x}),$$

dove w è il vettore dei pesi e x è il vettore degli input.

Durante la fase di addestramento il vettore dei pesi **w** viene aggiustato fino a che gli output del modello diventano consistenti con i veri output del training set.

Algorithm 5.4 Perceptron learning algorithm.

1: Let $D = \{(\mathbf{x}_i, y_i) \mid i = 1, 2, \ldots, N\}$ be the set of training examples.
2: Initialize the weight vector with random values, $\mathbf{w}^{(0)}$
3: **repeat**
4: **for** each training example $(\mathbf{x}_i, y_i) \in D$ **do**
5: Compute the predicted output $\hat{y}_i^{(k)}$
6: **for** each weight w_j **do**
7: Update the weight, $w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij}$.
8: **end for**
9: **end for**
10: **until** stopping condition is met

La formula dell'aggiornamento dei pesi è:

$$w_j^{(k+1)} = w_j^{(k)} + \lambda\left(y_i - \hat{y}_i^{(k)}\right)x_{ij},$$

λ (compreso tra 0 e 1) è un termine detto **tasso di apprendimento.** Il peso alla k+1 iterazione è la somma tra il peso all'iterazione precedente e un termine proporzionale all'errore. Se la predizione al passo k è corretta allora il peso rimane invariato.

Il percettrone converge a una soluzione ottima quando il problema è linearmente separabile.

X_1	X_2	y
0	0	−1
1	0	1
0	1	1
1	1	−1

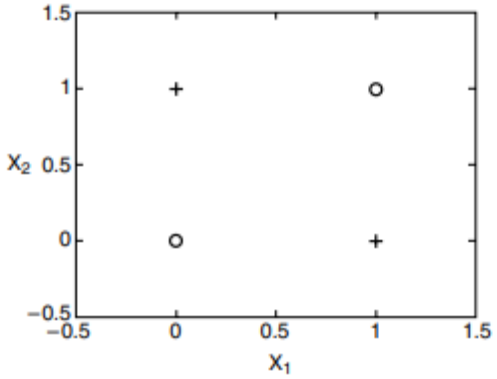
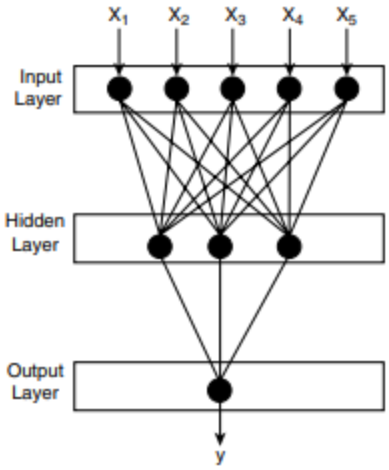


Figure 5.16. XOR classification problem. No linear hyperplane can separate the two classes.

Multilayer artificial network



Le ANN in generale sono più complesse del percettrone:

- hanno più livelli tra quello di input e quello di output. Tali livelli possono essere feedforward (i nodi di un livello sono collegati solo a quelli del livello successivo) o reccurrent (i nodi di un livello possono essere collegati anche a quelli dello stesso livello o a quelli del livello precedente)
- il modello può usare funzioni di attivazioni diverse dalla funzione segno

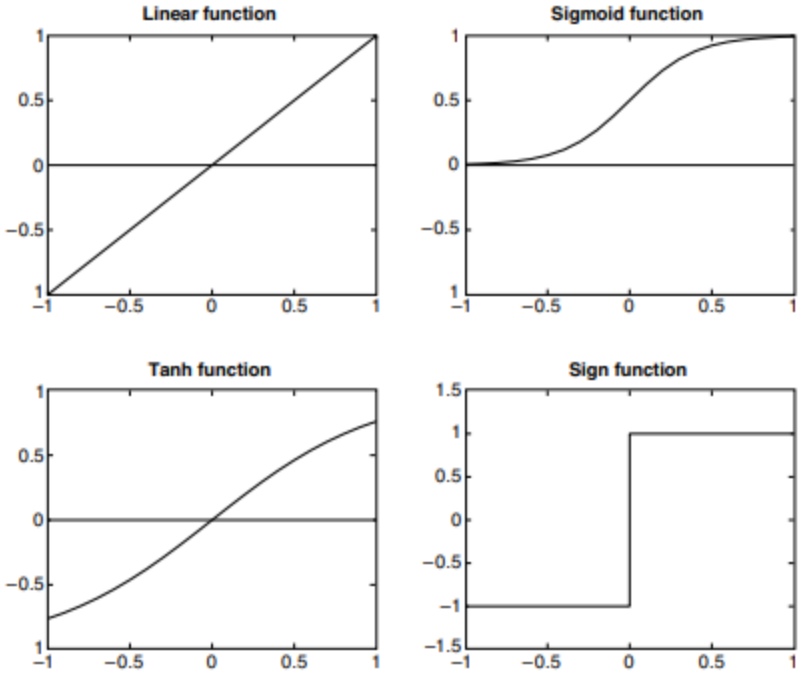


Figure 5.18. Types of activation functions in artificial neural networks.

La maggiore complessità del modello permette di modellare relazioni più complesse tra i dati in input e i loro output.

Problemi di design

Prima di allenare una rete neurale si deve:

- determinare il numero di input node
- determinare il numero di ouput node
- determinare la topologia della rete (quanti hidden layer, feedforward o recurrent)
- assegnare i pesi iniziali e i valori di bias
- pulire i dati in input eliminando i record con attributi mancanti

CNN

CNN = reti neurali convoluzionali

Una rete nurale convoluzionale è uno specifico tipo di rete neurale feedforward che impara le caratteristiche dei record tramite applicazione di filtri.

Una CNN consiste di:

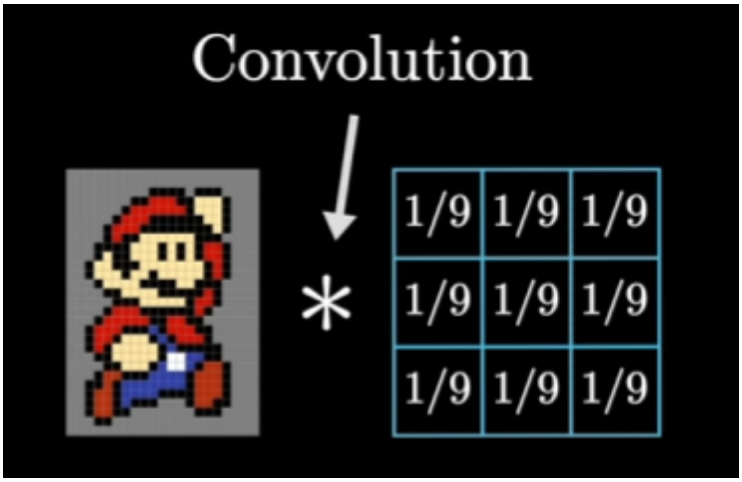
- un input layer
- dei layer nascosti
- un output layer

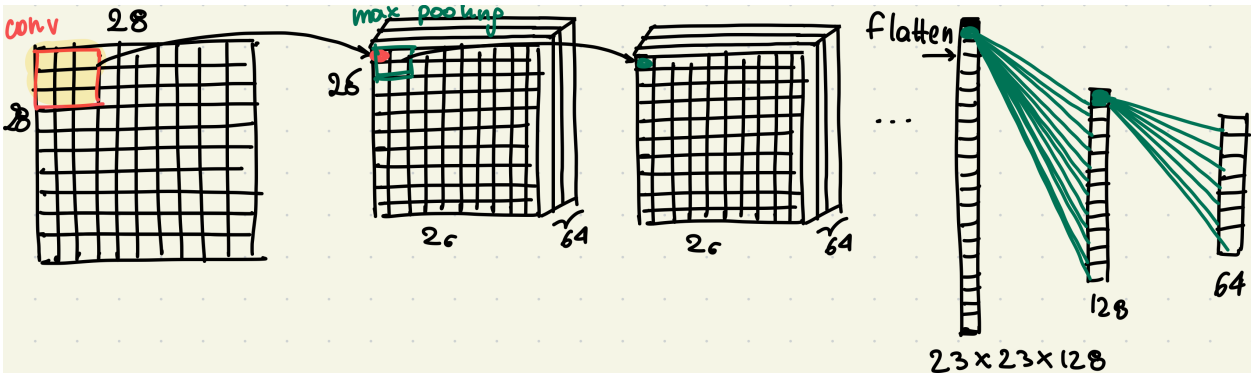
```
model = models.Sequential()
model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D())
model.add(layers.Conv2D(128, (3,3), activation='relu'))
model.add(layers.MaxPooling2D())
model.add(layers.Conv2D(128, (3,3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
```

```
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(37, activation='softmax'))
```

I principali livelli di una CNN sono:

- **convoluzione**: all'input vengono sovrapposte delle matrici di dimensione fissata, es 3×3, contenente dei pesi; ogni filtro ha lo scopo di rilevale delle caratteristiche locali. La matrice viene fatta scorrere sull'input, viene calcolato il prodotto scalare tra i valori dell'input e i pesi della matrice.
  Il risultato del prodotto scalare passa tramite la
  funzione di attivazione che introduce non linearità.
  Ogni filtro è specializzato per rilevare un pattern o una caratteristica specifica (es: rilevamento dei bordi, degli angoli).
  Ogni filtro produce una mappa delle caratteristiche (
  feature map) che rappresenta la presenza o l'assenza del pattern che il filtro cerca.
- **pooling**: operazione che ha lo scopo di ridurre la feature map, mantenendo le informazioni più rilevanti. Ci sono diversi tipi di pooling: maxpooling prende il valore maggiore tra quelli di una regione
- **fully connected layer**: ogni valore del vettore in input è collegato a ogni neurone del layer successivo attraverso un peso





Tecniche alternative di classificazione

Come creare un modello di classificazione con tecniche diverse dall'albero decisionale.

Rule-based classifier

Il rule-based classifier è una tecnica di classificazione basata su una serie di if... else

Le regole del modello sono rappresentate come una serie di OR $R = (r_1 \vee r_2 \vee r_3 ... \vee r_k)$ dove R viene detto **rule set** e r_i è la regola di classificazione o disgiunzione.

$$
\begin{array}{ll}
r_1: & (\text{Gives Birth} = \text{no}) \wedge (\text{Aerial Creature} = \text{yes}) \longrightarrow \text{Birds} \\
r_2: & (\text{Gives Birth} = \text{no}) \wedge (\text{Aquatic Creature} = \text{yes}) \longrightarrow \text{Fishes} \\
r_3: & (\text{Gives Birth} = \text{yes}) \wedge (\text{Body Temperature} = \text{warm-blooded}) \longrightarrow \text{Mammals} \\
r_4: & (\text{Gives Birth} = \text{no}) \wedge (\text{Aerial Creature} = \text{no}) \longrightarrow \text{Reptiles} \\
r_5: & (\text{Aquatic Creature} = \text{semi}) \longrightarrow \text{Amphibians}
\end{array}
$$

Esempio di classificazione dei vertebrati

Ogni regola di classificazione può essere espressa come:

$$
\begin{gathered}
r_i : (\text{Condition}_i) \rightarrow y_i \\
(\text{Condition}_i) = (A_1 \text{op}\, v_1) \wedge ... \wedge (A_k \text{op}\, v_k)
\end{gathered}
$$

la parte sinistra della regola di classificazione viene detta **precondizione** e essa contiene un insieme di congiunti: $(A_1 \text{op}\, v_1)$

la parte destra è detta
conseguenza e contiene la classe predetta y_i

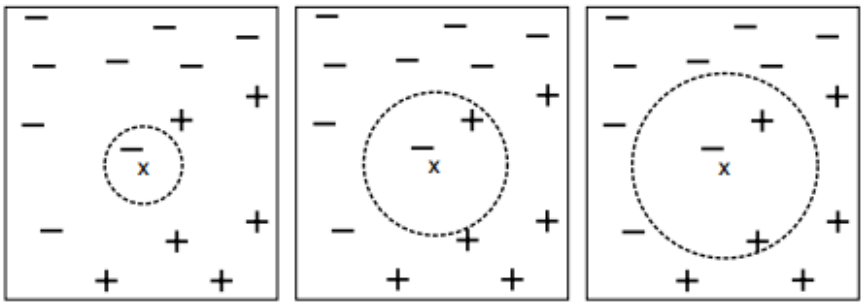
Una regola r *copre* un record x quando la precondizione di r.

Nearest-neighbor classifier

> if it walks like a duck, quaks like a duk, and looks like a duck, then it's probably a duck.

Classificatori che utilizzano i record di training più vicini al record da classificare per determinarne la classe.

Questo tipo di classificatore rappresenta ogni record come un punto in uno spazio di *d* dimensioni, dove d è il numero di attributi. Dato un record di test si calcola la sua distanza dal resto dei punti nel training set e si considerano i k punti più vicini.



(a) 1-nearest neighbor (b) 2-nearest neighbor (c) 3-nearest neighbor

Se k è troppo piccolo potremmo incorrere nel rischio di overfitting per via del rumore nei dati, se k è troppo grande il punto potrebbe essere classificato male perché stiamo considerando punti che non sono effettivamente vicini.

Bayesian classifiers

In molte applicazioni la relazione tra gli attributio e la classe è non-deterministica. Ovvero l'etichetta di un test record non può essere predetta con certezza anche

se l'insieme di attibuti è uguale a quello di un record dell'insieme di addestramento.

In questi casi la relazione tra gli attributi e la classe che deve essere assegnata al record è probabilistica.

Il teorema di Bayes combina la conoscenza pregressa sulle classi con quella evinta dai dati.

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}.$$

Possiamo applicare il teorema di Bayes alla classificazione. Denotiamo con **X** l'insieme degli attributi e con Y la classe. Se la classe ha una relazione non deterministica con gli attributi possiamo trattare **X** e Y come variabili casuali e catturare la loro relazione con $P(Y|\mathbf{X})$

Es:

	binary	categorical	continuous	class
Tid	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Se per un dato record con attibuti **X** se $P(\mathrm{No}|\mathbf{X}) > P(\mathrm{Yes}|\mathbf{X})$ allora il record sarà etichettato come `no`.

Definiamo **class-conditional probability:** $P(\mathbf{X}|Y)$

quindi possiamo riscrivere il teorema di Bayes:

$$P(Y|\mathbf{X}) = \frac{P(\mathbf{X}|Y) \times P(Y)}{P(\mathbf{X})}.$$

Possiamo calcolare $P(\mathbf{X}|Y)$ tramite due metodi:

- naive Bayes classifier
- Bayesian belief network

Naive Bayes classifier

Questo calssificatore assume che gli attributi dei record siano indipendenti, data l'etichetta y.

Questa indipendenza può essere formalizzata come segue:

$$P(\mathbf{X}|Y = y) = \prod_{i=1}^{d} P(X_i|Y = y),$$

Invece di calcolare la class-conditional probability per ogni cocmbinazione di **X** dobbiamo stimare la probabilità condizionata di ogni X_i, data Y.

Quindi la formula di Bayes si trasforma in:

$$P(Y|\mathbf{X}) = \frac{P(Y)\prod_{i=1}^{d} P(X_i|Y)}{P(\mathbf{X})}.$$

Calcoliamo $P(X_i|Y)$ diversame in base al tipo di attributo:

- se X_i assume valori discreti: la probabilità condizionata $P(X_i = x_i|Y = y)$ è la frazione di record del training set nella classe y che hanno x_i come valore di X_i
- se X_i assume valori continui possiamo:
  - discretizzare tali valori in intervalli e ricadere nel primo caso

- possiao usare una determinata distribuzione di probabilità per X_i, come ad esempio una Gaussiana

$$P(X_i = x_i|Y = y_j) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} \exp^{-\frac{(x_i-\mu_{ij})^2}{2\sigma_{ij}^2}}.$$

Bayesian belief network

L'indipendenza assunta dal naive Bayes classifier può essere troppo stringente. Il modello BNN permette di specificare quali coppie di attributi sono dipendenti.

Questo modello fornisce una rappresentazione grafica della relazione probabilistica in un insieme di variabili casuali. Gli elementi fondamentali sono due:

- un grafo diretto aciclico che rappresenta le relazioni di dipendenza tra insiemi di variabili
- una tabella di probabilità che associa a ogni nodo i suoi immediati parenti

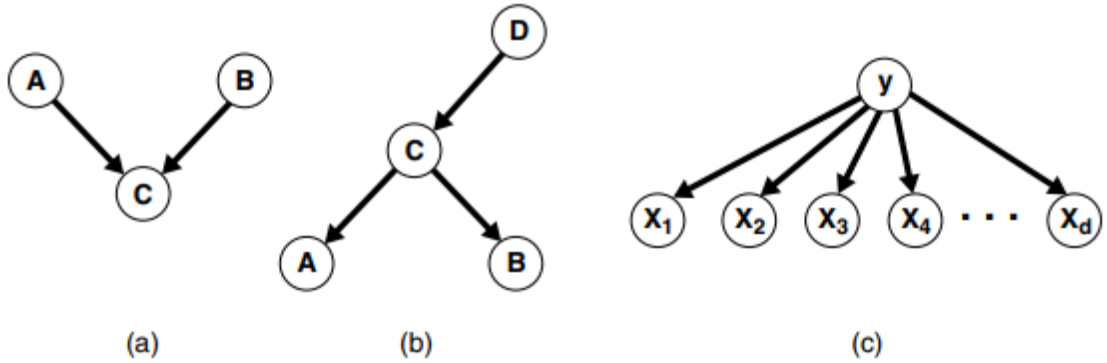


Figure 5.12. Representing probabilistic relationships using directed acyclic graphs.

Nella figura a abbiamo che C è dipendente da A e B.

In generale:

Proprietà:

un nodo in una rete Bayesiana è indipendente dai nodi che non sono suoi discendenti.

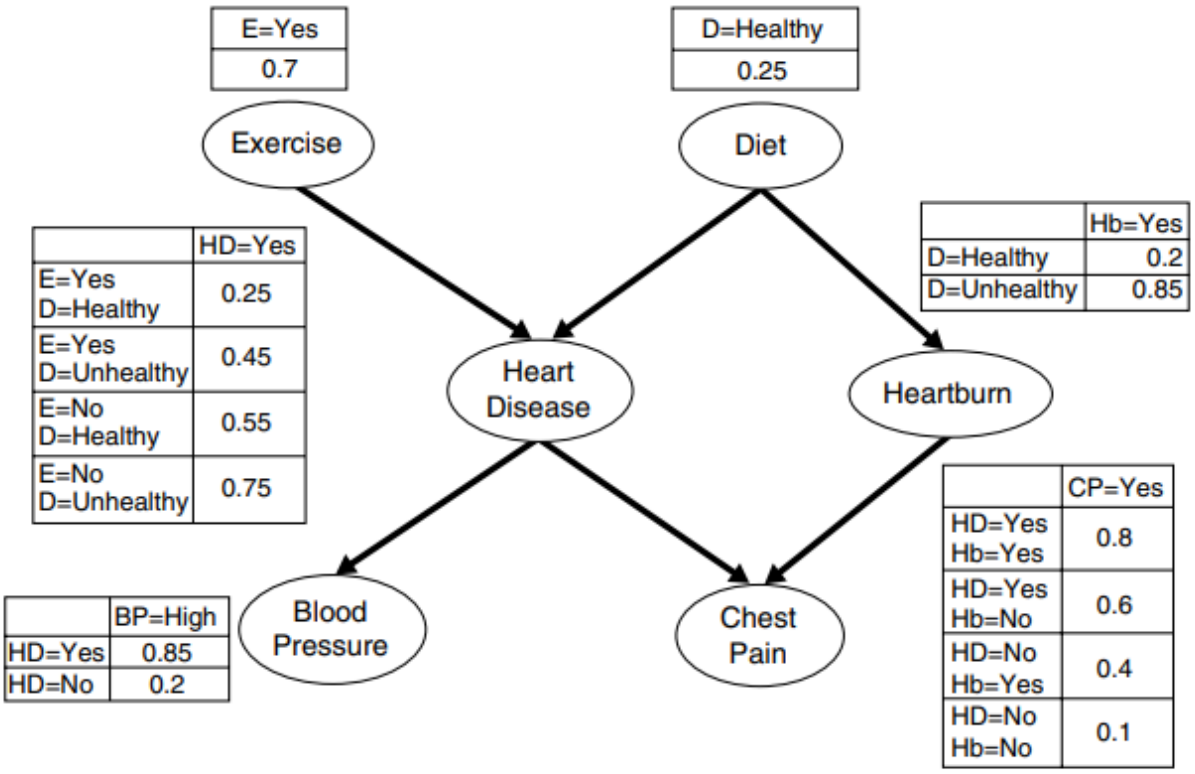


Figure 5.13. A Bayesian belief network for detecting heart disease and heartburn in patients.