CheetSheet

Algoritmi di ricerca non euristici

Ricerca ad Albero

```
FUNCTION Ricerca-Albero (problema)
    inizializza la frontiera con stato iniziale del problema
    LOOP DO
        IF la frontiera è vuota
            return fallimento
        Scegli* un nodo foglia da espandere e rimuovilo dalla fron
        IF il nodo contiene uno stato obiettivo
            RETURN la soluzione corrispondente
        Espandi il nodo e aggiungi i successori alla frontiera
END
```

Ricerca in ampiezza (BF)

BF su/ad albero

```
FUNCTION Ricerca-Ampiezza-A (problema)
    nodo = un nodo con stato problema.stato-iniziale e costo-di-ca
    IF problema.Test-Obiettivo(nodo.Stato)
        RETURN Soluzione(nodo)
    frontiera = coda FIFO con nodo come unico elemento
    LOOP DO
        IF vuota?(frontiera)
            RETURN fallimento
        nodo = POP(frontiera)
        FOR EACH azione IN problema.Azioni(nodo.stato) DO
            figlio = Nodo-Figlio(problema, nodo, azione)
            IF Problema.TestObiettivo(figlio.stato)
                RETURN Soluzione(figlio)
            frontiera = Inserisci(figlio, frontiera)
END
```

BF su grafo

```
FUNCTION Ricerca-Ampiezza-G (problema)
    nodo = nodo con stato problema.stato-iniziale e costo-cammino
    IF problema.TestObiettivo(nodo.Stato)
        RETURN Soluzione (nodo)
    frontiera = coda FIFO con nodo come unico elemento
    esplorati = insieme vuoto
    LOOP DO
        IF Vuota?(frontiera)
            RETURN fallimento
        aggiungi nodo.Stato a esplorati
        FOR EACH azione IN problema.Azioni(nodo.Stato) DO
            IF figlio.Stato non è in esplorati e non è in frontier
                IF Problema.TestObiettivo(figlio.Stato)
                    RETURN Soluzione(figlio)
                frontiera = Inserisci(figlio, frontiera)
END
```

Analisi della complessità

- Completo
- Se gli operatori hanno tutti lo stesso costo $k \overset{quindi}{\implies} g(n) = k \cdot depth(n)$ dove $g(n)$ è il costo del cammino per arrivare a $n \implies$ **Ottimale**
- **Complessità nel tempo** (nodi generati)

$$T(b,d) = 1 + b + b^2 + \cdots + b^d \rightarrow O(b^d)$$

- **Complessità in spazio** (nodi in memoria)

$$O(b^d)\ [\text{frontiera}]$$

dove b è il fattore di diramazione e d la profondità dell'albero

Ricerca in profondità (DF)

DF su albero

```
FUNZIONE Ricerca-Profondità-A (problema)
    nodo = Nodo con stato problema.stato-iniziale e costo-di-cammi
    IF Problema.TestObiettivo(nodo.Stato)
        RETURN Soluzione(nodo)
    frontiera = coda FIFO con nodo come unico elemento
    LOOP DO
        IF vuota?(frontiera)
```

```
            RETURN fallimento
        nodo = POP(frontiera)
        FOR EACH azione IN problema.Azioni(nodo.stato) DO
            figlio = nodo.Figlio (problema, nodo, azione)
            IF figlio.Stato non è in frontiera
                IF Problema.TestObiettivo(figlio.stato)
                    RETURN Soluzione(figlio)
            frontiera = Inserisci(figlio, frontiera)
END
```

DF su grafo

```
FUNCTION Ricerca-Ampiezza-G (problema)
    nodo = nodo con stato problema.stato-iniziale e costo-cammino
    IF problema.TestObiettivo(nodo.Stato)
        RETURN Soluzione (nodo)
    frontiera = coda FIFO con nodo come unico elemento
    esplorati = insieme vuoto
    LOOP DO
        IF Vuota?(frontiera)
            RETURN fallimento
        aggiungi nodo.Stato a esplorati
        FOR EACH azione IN problema.Azioni(nodo.Stato) DO
            figlio = nodo.Figlio (problema, nodo, azione)
            IF figlio.Stato non è né in esplorati né in frontiera
                IF problema.TestObiettivo (figlio.stato)
                    RETURN Soluzione(figlio)
            frontiera = Inserisci(figlio, frontiera)
END
```

DF Ricorsivo

```
FUNCTION Ricerca-DF-A (problema)
    RETURNS Ricerca-DF-ricorsiva(nodo,problema)
END

FUNCTION Ricerca-DF-ricorsiva(nodo, problema)
    IF problema.TestObiettivo(nodo.stato)
        RETURN Soluzione(nodo)
    ELSE
        FOR EACH azione IN problema.Azioni(nodo.Stato) DO
            figlio = Nodo-Figlio (problema, nodo, azione)
            risultato = Ricerca-DF-ricorsiva(figlio, problema)
```

```
            IF risultato != fallimento
                RETURN risultato
        RETURN fallimento
END
```

Analisi della complessità

SU ALBERO

- **Complessità in tempo:** $O(b^m)$ (che può essere $> O(b^d)$)
- **Complessità in spazio:** bm
  dove m è la lunghezza max dei cammini nello spazio degli stati e b il fattore di diramazione
- **Non Completa** e **Non Ottimale**

SU GRAFO

- Si perdono i vantaggi di memoria (si torna da bm a $texttuttiipossibilistati$)
- diventa **Completa** in spazi degli stati finiti
- **Non ottimale**

RICORSIVA

- Più efficiente in occupazione di memoria perché mantiene solo il cammino corrente (m nodi al caso pessimo)

Ricerca in profondità limitata (DL)

L'algoritmo è uguale a quello della ricerca in profondità ma si cerca fino a un certo livello l

Analisi della complessità

- **Completa** per problemi in cui si conosce il limite superiore per la profondità della soluzione $\overset{quindi}{\implies}$ **Completo** se $d < l$
  dove d è la profondità del nodo obiettivo più superficiale e l è il limite
- **Non Ottimale**
- **Complessità in tempo:** $O(b^l)$
- **Complessità in spazio:** $O(bl)$

Ricerca con approfondimento iterativo (ID)

Si fa DL con $l = 1, 2, 3, ...$ fino a trovare la soluzione

Analisi della complessità

Miglior compromesso tra BF e DF

- Vantaggi della BF
  Completo e **Ottimale** se il costo delle operazioni è fisso
- Con tempi analoghi, ma costo memoria come DF
- **Complessità in tempo:** $O(b^d)$
- **Complessità in spazio:** $O(bd)$

Direzione della ricerca

IN AVANTI (O GUIDATA DAI DATI)

Si esplora lo spazio della ricerca dallo stato iniziale allo stato obiettivo.
Si preferisce quando l'obiettivo è chiaramente definito o si possono formulare una serie limitata di ipotesi

ALL'INDIETRO (O GUIDATA DALL'OBIETTIVO)

Si esplora lo spazio di ricerca a partire da uno stato goal e riconducendosi a sotto-goal fino a trovare uno stato iniziale.
Si preferisce quando gli obiettivi possibili sono molti

Ricerca bidirezionale

Si fa DL nelle due direzioni fino ad incontrarsi

ANALISI

- **Complessità in tempo:** $O(b^{d/2})$
- **Complessità in spazio:** $O(b^{d/2})$
  Non è sempre applicabile, ad es. se i predecessori non sono definiti o ci sono troppi stati obiettivo

Ricerca di costo uniforme (UC)

UC su albero

```
FUNCTION Ricerca-UC-A (problema)
    nodo = un nodo con stato il problema.stato-iniziale e costo-di
```

```
    frontiera = una coda con priorità con nodo come unico elemento
    LOOP DO
        IF Vuota?(frontiera)
            RETURN fallimento
        nodo = POP(frontiera)
        IF problema.TestObiettivo(nodo.Stato)
            RETURN Soluzione(nodo)
        FOR EACH azione in problema.Azioni(nodo.Stato) DO
            figlio = Nodo-Figlio(problema, nodo, azione)
        frontiera = Inserisci(figlio, frontiera)
END
```

UC su grafo

```
FUNCTION Ricerca-UC-G (problema)
    nodo = un nodo con stato il problema.stato-iniziale e costo-di
    rontiera = una coda con priorità con nodo come unico elemento
    esplorati = insieme vuoto
    LOOP DO
        IF Vuota?(frontiera)
            RETURN fallimento
        nodo = POP(frontiera);
        IF problema.TestObiettivo(nodo.Stato)
            RETURN Soluzione(nodo)
        aggiungi nodo.Stato a esplorati
        FOR EACH azione in problema.Azioni(nodo.Stato) DO
            figlio = Nodo-Figlio(problema, nodo, azione)
            IF figlio.Stato non è in esplorati e non è in frontier
                frontiera = Inserisci(figlio, frontiera)
            ELSE IF figlio.Stato è in frontiera con Costo-cammino
                sostituisci quel nodo frontiera con figlio
END
```

Analisi della complessità

- **Ottimale** e **Completo** se il costo degli archi è maggiore di un $\varepsilon > 0$
- **Complessità:** $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
  dove C^* è il costo della soluzione ottima e $\lfloor C^*/\varepsilon \rfloor$ è il numero di mosse nel caso peggiore (arrotondato per difetto)
  Quando ogni azione ha lo stesso costo $O(1+d)$ (simile a BF)

Confronto delle strategie (versioni su albero)

Criterio	BF	UC	DF	DL	ID	Bidirezionale
Completezza	si	si (^)	no	si(+)	si	si (£)
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Spazio	$O(d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Ottimalità	si (*)	si(^)	no	no	si (*)	si (£)

Algoritmi di ricerca euristica

Greedy Best-First

```
FUNZIONE Greedy-Best-First(problema, euristica)
    nodo-iniziale = nodo con stato = problema.stato-iniziale, cost
    frontiera = coda con nodo-iniziale come unico elemento
    esplorati = insieme vuoto
    LOOP DO
        IF Vuota?(frontiera)
            RETURN fallimento
        nodo = POP(frontiera)
        IF problema.TestObiettivo(nodo.Stato)
            RETURN Soluzione(nodo)
        aggiungi nodo.Stato a esplorati
        FOR EACH azione IN problema.Azioni(nodo.Stato) DO
            figlio = NodoFiglio(problema, nodo, azione)
            IF figlio.Stato non è in esplorati e non è in frontier
                figlio.costo-totale = euristica(figlio.Stato)
                frontiera = Inserisci(figlio, frontiera)
            ELSE IF figlio è nella frontiera con costo-totale magg
                sostituisci nodo nella frontiera con figlio
    END
END
```

Analisi della complessità

- **Complessità:** Dipende dalla qualità dell'euristica utilizzata e dalla struttura del grafo di ricerca. In generale, l'algoritmo ha una complessità temporale e spaziale di $O(b^m)$, dove b è il fattore di branching massimo e m è la profondità massima della soluzione.
- **Ottimalità:** Dipende dall'euristica utilizzata. Se l'euristica è ammissibile, cioè se non sovrastima mai il costo effettivo per raggiungere l'obiettivo,

allora l'algoritmo è garantito di trovare la soluzione ottima se esiste una soluzione.

- **Completezza:** dipende anche dall'euristica utilizzata. Se l'euristica è consistente, cioè se soddisfa la proprietà di monotonicità, allora l'algoritmo è garantito di trovare la soluzione ottima se esiste una soluzione. Tuttavia, se l'euristica non è consistente, l'algoritmo potrebbe non terminare o trovare una soluzione non ottima.

Algoritmo A e A*

La differenza tra i due sta nella funzione $h(n)$ (`euristica`), se questa è ammissibile (non sovrastima) si tratta di `A*` altrimenti `A`.

```
FUNZIONE Ricerca-A* (problema)
  nodo-iniziale = nodo con stato=problema.stato-iniziale, costo-ca
  frontiera = coda di priorità con nodo-iniziale come unico elemen
  esplorati = insieme vuoto
  LOOP DO
    SE frontiera è vuota
      RETURN fallimento
    nodo-attuale = POP(frontiera)
    SE problema.Test-Obiettivo(nodo-attuale.stato)
      RETURN Soluzione(nodo-attuale)
    AGGIUNGI nodo-attuale.stato a esplorati
    PER OGNI azione IN problema.Azioni(nodo-attuale.stato) FARE
      figlio = Nodo-Figlio(problema, nodo-attuale, azione)
      SE figlio.stato non è in esplorati e non è in frontiera
        valore-g = nodo-attuale.costo-cammino + costo(nodo-attuale
        valore-h = h(figlio.stato)
        valore-f = valore-g + valore-h
        frontiera = Inserisci(figlio, frontiera) con priorità valo
      ALTRIMENTI SE figlio.stato è in frontiera e il suo valore-f
        aggiorna la priorità di figlio in frontiera con il nuovo v
  END LOOP
END
```

Analisi della complessità

A

- **Ottimalità** e **Completezza** non garantite

- La **Complessità** dipende dall'euristica utilizzata. In particolare, se l'euristica non è consistente, l'algoritmo potrebbe espandere molti più nodi rispetto all'algoritmo A* con la stessa euristica. In generale, l'algoritmo A con un'euristica non ammissibile è utilizzato solo in casi in cui la complessità computazionale dell'euristica ammissibile è troppo elevata.

A*

- **Complessità:** $O(b^m)$
  L'algoritmo A *ha una complessità temporale dipendente dalla qualità dell'euristica utilizzata, ma in generale è esponenziale nella complessità della soluzione ottima. In particolare, se l'euristica è ammissibile e consistente, allora l'algoritmo A* garantisce di trovare la soluzione ottima con un tempo di esecuzione che è proporzionale al numero di nodi generati dallo spazio degli stati.
- **Ottimalità:** Ottimo se h(n) è ammissibile
  L'algoritmo A* garantisce di trovare la soluzione ottima se l'euristica utilizzata è ammissibile, cioè se non sovrastima il costo della soluzione minima.
- **Completezza:** Sì, se il costo è limitato superiormente e h(n) è ammissibile
  L'algoritmo A *garantisce di trovare la soluzione ottima se lo spazio degli stati è finito e se l'euristica utilizzata è ammissibile. Se lo spazio degli stati è infinito, l'algoritmo può non terminare. Tuttavia, se l'euristica è consistente, allora A* è completo anche in spazi degli stati infiniti.

Beam Search

É come un Best First ma ad ogni passo `k` mantiene solo i nodi più promettenti (`k` l'ampiezza del raggio (beam))

```
FUNCTION Ricerca-Beam(problema, k)
    nodo = nodo con stato problema.stato-iniziale e costo-cammino
    IF problema.TestObiettivo(nodo.Stato)
        RETURN Soluzione(nodo)
    frontiera = k nodi generati da nodo
    LOOP DO
        IF Vuota?(frontiera)
            RETURN fallimento
```

```
        nuovi-nodi = {}
        FOR EACH nodo IN frontiera DO
            FOR EACH azione IN problema.Azioni(nodo.Stato) DO
                figlio = Nodo-Figlio(problema, nodo, azione)
                IF Problema.TestObiettivo(figlio.Stato)
                    RETURN Soluzione(figlio)
                aggiungi figlio a nuovi-nodi
            frontiera = k nodi migliori di nuovi-nodi
END
```

`k` rappresenta la larghezza della beam, ovvero il numero di nodi migliori da mantenere nella frontiera ad ogni passo.

Analisi della complessità

- **Complessità:** dipende dalla larghezza della fascia `k` che viene esplorata, e può essere esponenziale nel caso peggiore, come la ricerca in ampiezza.
- **Ottimalità:** dipende dall'euristica utilizzata. In generale, non è ottimale, a meno che l'euristica sia adatta a guidare l'algoritmo verso la soluzione ottimale.
- **Completezza:** dipende dalla larghezza della fascia `k` e dalla struttura del problema. In generale, non è completo perché può terminare prematuramente quando la soluzione non è presente nella fascia `k`.

A *con approfondimento iterativo (IDA)*

Combina A* con ID, ad ogni passo ricerca in profondità con un limite dato dalla funzione f (e non dalla profondità). Il `limite` viene incrementato ad ogni iterazione e il punto critico è di quanto viene incrementato `limite`.

```
FUNCTION IDA*-Search(problema)
    limite = f(problema.stato-iniziale)
    LOOP DO
        result = Ricerca-Limite(problema, limite, 0)
        IF result = soluzione THEN
            RETURN soluzione
        ELSE IF result = infinito THEN
            RETURN fallimento
        limite = result
    END LOOP
END
```

```
FUNCTION Ricerca-Limite(problema, limite, costo-cammino)
    nodo = nodo con stato problema.stato-attuale e f(nodo) <= limi
    IF problema.TestObiettivo(nodo.stato) THEN
        RETURN Soluzione(nodo)
    minimo = infinito
    FOR EACH azione IN problema.Azioni(nodo.stato) DO
        figlio = Nodo-Figlio(problema, nodo, azione)
        valutazione = Ricerca-Limite(problema, limite, costo-cammi
        IF valutazione = soluzione THEN
            RETURN soluzione
        ELSE IF valutazione < minimo THEN
            minimo = valutazione
    RETURN minimo
END
```

Analisi della complessità

- **Complessità:** è simile a quella dell'algoritmo A*: esponenziale nel caso peggiore, ma migliore della ricerca esaustiva.
    - *Complessità in spazio: $O(bd)$*
- **Ottimalità:** è ottimale (l'euristica è sempre ammissibile)
- **Completezza:** dipende dal fatto che lo spazio degli stati sia finito o infinito. Se è finito, allora IDA *è completo. Se invece lo spazio degli stati è infinito, IDA* può non terminare o non trovare la soluzione ottima. Tuttavia, è comunque completo in un sottoinsieme di spazi degli stati infiniti.

Ricerca best-first ricorsiva (RBFS)

É simile a DF Ricorsivo, tiene tracia ad ogni livello del miglior percorso alternativo. Invece di fare backtracking in caso di fallimento, interrompe l'esplorazione quando trova un nodo promettente (secondo f).

```
FUNCTION Ricerca-Best-First-Ricorsiva(problema)
    RETURN RBFS(problema, CreaNodo(problema.stato-iniziale), infin
END

FUNCTION RBFS (problema, nodo, limite)
    IF problema.TestObiettivo(nodo.Stato)
        RETURN Soluzione(nodo)
    successori = []
```

```
    FOR EACH azione IN problema.Azioni(nodo.Stato) DO
        aggiungi Nodo-Figlio(problema, nodo, azione) a successori
    IF successori è vuoto
        RETURN (fallimento, infty)
    FOR EACH s in successori DO
        s.f = max(s.g + s.h, nodo.f)
    LOOP DO
        migliore = nodo con f minimo tra tutti i successori
        IF migliore.f > limite
            RETURN (fallimento, migliore.f)
        alternativa = secondo nodo con f minimo tra tutti i succes
        (risultato, migliore.f) = RBFS(problema, migliore, min(lim
        IF risultato != fallimento
            RETURN risultato
    END LOOP
END
```

Analisi della complessità

- **Complessità:** dipende dalla qualità della funzione euristica utilizzata, ma in generale ha una complessità esponenziale. Tuttavia, in pratica, l'algoritmo può convergere molto rapidamente rispetto ad altre strategie di ricerca informate, specialmente se la soluzione è raggiungibile a partire da uno stato iniziale relativamente vicino.
- **Ottimalità:** garantisce la ricerca dell'ottimo globale in spazi di ricerca con alberi con pesi positivi. Tuttavia, in caso di cicli di pesi negativi, RBFS potrebbe non trovare l'ottimo globale.
- **Completezza:** non è completo in spazi di ricerca infiniti. Tuttavia, se lo spazio di ricerca è finito o limitato dalla profondità massima, RBFS garantisce di trovare una soluzione se esiste.

A *con memoria limitata (MA)* in versione semplice (SMA*)

Procede come A* fino ad esaurimento della memoria disponibile, poi "dimentica" il nodo peggiore, doppo aver aggiornato il valore del padre. A parità di f si sceglie il nodo migliore più recente e si dimentica il nodo peggiore più vecchio

Analisi della complessità

- **Complessità:** dipende dall'euristica utilizzata. In generale, si può dire che SMA *è meno efficiente di A* ma più efficiente di IDA*.
- **Ottimalità:** ottimale se il cammino soluzione sta in memoria
- **Completezza:** completo, ovvero troverà sempre una soluzione se esiste, a meno che il limite della memoria non sia raggiunto.

Algoritmi model checking

TV-Consegue

```
FUNCTION TV-Consegue(KB, a)
    simboli <- una lista dei simboli proposizionali contenuti in K
    RETURN TV-Verifica-Tutto(KB, a, simboli, {})
end
```

```
FUNCTION TV-Verifica-Tutto(KB, a, simboli, modello)
    IF Vuoto?(simboli) THEN
        IF PL-Vero?(KB, modello) THEN RETURN PL-Vero?(KB, modello)
        ELSE RETURN true // quando KB è false, restituisce sempre
    ELSE DO
        P <- Primo(simboli); resto <- Resto(simboli)
        RETURN TV-Verifica-Tutto(KB, a, resto, modello<-{P=true})
                AND
                TV-Verifica-Tutto(KB, a, resto, modello<-{P=false}
end
```

DPLL

```
FUNCTION DPLL-Soddisfacibile?(s)
            inputs: s, una formula della logica proposizionale
    clausole ← l'insieme di clausole nella rappresentazione CNF di
    simboli ← una lista di tutti i simboli proposizionali in s
    RETURN DPLL(clausole, simboli, { })
END
```

```
FUNCTION DPLL(clausole, simboli, modello)
    IF ogni clausola in clausole è vera in modello
        THEN RETURN true
    IF qualche clausola in clausole è falsa in modello
        THEN RETURN false
    P, valore ← Trova-Simbolo-Puro(simboli, clausole, modello)
```

```
    OF P è diverso da null
        THEN RETURN DPLL(clausole, simboli - P, modello ← {P = val
    P, valore ← Trova-Clausola-Unitaria(clausole, modello)
    IF P è diverso da null
        THEN RETURN DPLL(clausole, simboli - P, modello ← {P = val
    P ← Primo(simboli); resto ← Resto(simboli)
    RETURN
        DPLL(clausole, resto, modello ← {P = true}) or
        DPLL(clausole, resto, modello ← {P = false})
END
```

WalkSat

```
FUNCTION WalkSAT(clausole, p, max_flips)
            inputs: clausole, un insieme di clausole
    p, la probabilità di effettuare una “camminata casuale”, tipic
    max_flips, numero massimo di inversioni di valore prima di abb

    modello ← un assegnamento casuale di valori di verità ai simbo

    FOR i = 1 TO max_flips DO
        IF modello soddisfa clausole
            THEN RETURN modello
        clausola ← una clausola, falsa in modello, scelta casualme
                                                    nell’insieme claus
        IF Random(0, 1) =< p
            THEN inverti il valore in modello di un simbolo scelto
                                                casualmente in clausol
        ELSE inverti il valore di verità del simbolo in clausole c
                            massimizza il numero di clausole soddisfat
    RETURN fallimento
```

Calcolo Proposizionale

Leggi

- $(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$ Commutatività di $\wedge$
- $(\alpha \vee \beta) \equiv (\beta \vee \alpha)$ Commutatività di $\vee$
- $((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$ Associatività di $\wedge$
- $((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$ Associatività di $\vee$
- $\neg(\neg\alpha) \equiv \alpha$ eliminazione della doppia negazione

- $(\alpha \implies \beta) \equiv (\neg\beta \implies \neg\alpha)$ contrapposizione
- $(\alpha \implies \beta) \equiv (\neg\alpha \lor \beta)$ eliminazione dell'implicazione
- $(\alpha \iff \beta) \equiv ((\alpha \implies \beta) \land (\beta \implies \alpha))$ eliminazione della doppia implicazione
- $\neg(\alpha \land \beta) \equiv (\neg\alpha \lor \neg\beta)$ De Morgan
- $\neg(\alpha \lor \beta) \equiv (\neg\alpha \land \neg\beta)$ De Morgan
- $(\alpha \land (\beta \lor \gamma)) \equiv ((\alpha \land \beta) \lor (\alpha \land \gamma))$ distributività di $\land$ su $\lor$
- $(\alpha \lor (\beta \land \gamma)) \equiv ((\alpha \lor \beta) \land (\alpha \lor \gamma))$ distributività di $\lor$ su $\land$

Algoritmi Machine Learning

Find-S

```
    FOR EACH x in h
        IF the a_i in h is satisfied by x
            THEN replace a_i in the next more general constraing t
                                                    is staisfied b
```

Candidate-Elimination

```
    FOR EACH x in h
        IF the a_i in h is satisfied by x
            THEN replace a_i in the next more general constraing t
                                                    is staisfied b
        ELSE replace g_i in the next more specific contrai....
```

Local Search

$$w_{new} = w + \eta\Delta w \Bigg| \Delta w_0 = -\frac{\partial E(w)}{\partial w_0} = 2(y - h(x)) \Bigg| \Delta w_1 = \frac{\partial E(w)}{\partial w} = 2(y - h_w(x)) \cdot x$$

ID3

```
ID3 (X, T, Attrs)
    create Root node
    IF tutti gli X sono + RETURN Root con classe +
    IF tutti gli X sono - RETURN Root con classe -
    IF Attrs è vuoto RETURN Root con la classe più comune di T in
    ELSE
        A <-- miglio attributo, attributo di decisione per Root <-
        FOR EACH possibile valore v_i di A
```

```
            aggiungi un nuovo ramo sotto Root, per testare a = v_i
            X_i <-- sottoinsieme di X con A = k
            IF X_i è vuoto THEN aggiungi una nuova fogli acon la c
            ELSE aggiungi il sottoalbero generato da ID3(X_i, T, A
    RETURN Root
```

Formule

Linear Models

- Errore quadratico: $E(w) = \sum_{p=1}^{l}(y_p - h_w(x_p))^2$ (anche Loss function)
- Regressione polinomiale: $h_w(x) = \sum_{j=0}^{M} w_j x^j$
- Linear basisi expansion: $h_w(x) = \sum_{k=0}^{N} w_k \phi_k(x)$
- Ride Regression / Regolarizzazione di Tikhonov: $Loss(h_w) = \sum_{p=1}^{l}(y_p - h_w(x_p))^2 + \lambda||w||^2$
- Classificazione: $h(x) = sign(wx + w_0)$

Decision tree

- entropia: $p_+ log_2 p_+ - p_- log_2 p_-$
- Gain: $Gain(S, A) = Entropy(S) - \sum_{v \in Values(S)} \frac{|S_V|}{|S|} Entropy(S_V)$
- GainRatio: $GainRatio(S, A) = \frac{Gain(S,A)}{SplitInformation(S,A)}$
- SplitInInformation: $SplitInformation(S, A) = -\sum_{i=1}^{c} \frac{|S_i|}{|S|} log_2 \frac{|S_i|}{|S|}$

Support Vector Machine (SVM)

- margini: $margin \propto \frac{2}{||w||}$
- VC-dim: $\frac{1}{margin}$
- support vector: $h(x) = sign(w^T x + b) = sign(\sum_{p=1}^{l} \alpha_p y_p x_p^T x + b) = \boxed{sign(\sum_{p \in SV} \alpha_p y_p x_p^T x + b)}$
- soft margin: minimizzare $\frac{|w|^2}{2} + C \cdot \sum_p \xi_p$ tale che $(wx_p + b)y_p \geq 1 - \xi_p$ e $\xi_p \geq 0 \; \forall p$
- funzione kernel: $h(x) = sign(\sum_{p \in SV} \alpha_p y_p \boxed{K(x_p, x)})$ con $k(x_i, x_j) = \phi(x_i)^T \phi(x_j)$
  - kernel lineare: $K(x_i, x_j) = x_i^T x_j$

- kernel polinomiale: $p : K(x_i, x_j) = (1 + x_i^T x_j)^k$
- kernel rbf: $K(x_i, x_j) = e^{-\frac{||x_i - x_j||^2}{2\sigma^2}}$

STL

- rischio empirico: $R_{emp} = \frac{1}{l} \sum_{p=1}^{l} (d_p - h(x_p))^2$
- VC-bounds: garatisce con prob. $1 - \delta$ che $R \leq R_{emp} + \varepsilon(\frac{1}{l}, VC, \frac{1}{\delta})$