

3. Ragionamento con Vincoli

Dispensa ICon

versione: 19/10/2024, 12:32

Variabili e Vincoli · Risoluzione di CSP tramite Ricerca · Algoritmi Basati su Consistenza · Separazione dei Domini · Eliminazione di Variabili · Ricerca Locale · Varianti della Ricerca Locale · Algoritmi Basati su Popolazioni · Beam Search · Ottimizzazione

1 Variabili e Vincoli

Si vedrà come passare da spazi di stati a spazi definiti da caratteristiche.

Una **feature** viene descritta attraverso una **variabile**, spesso non *indipendenti* fra loro, e **vincoli rigidi** che specificano combinazioni lecite di assegnazioni alle variabili, e/o **vincoli flessibili** preferenze sulle assegnazioni.

Il *ragionamento* si svolge generando assegnazioni che soddisfino i vincoli rigidi e ottimizzino i vincoli flessibili.

1.1 Variabili e Assegnazioni

Si considereranno *problemi* descritti in termini di **variabili algebriche** ossia simboli usati per denotare caratteristiche del mondo (reale o immaginario): *mondi possibili*.

La *notazione* adotta nomi che iniziano per maiuscola, ad es. **X**: ogni variabile ha un **dominio** associato, ossia un insieme di valori che può assumere, denotato $dom(X)$.

Si considereranno *variabili discrete* con dominio finito o almeno enumerabile, ad es. variabili *binarie*, ossia con un dominio di 2 valori, come ad es. le variabili *booleane*, che hanno dominio $\{true, false\}$. Un altro tipo è quello delle *variabili continue*, i.e. non discrete. Ad es. variabili con dominio \mathbb{R} o un suo intervallo, e.g. $[0, 1]$.

Un'**assegnazione** è una funzione da un insieme di variabili ai loro domini: dato $\{X_1, X_2, \dots, X_k\}$ a X_i si assegna $v_i \in dom(X_i)$ per ogni $i = 1, \dots, k$:

$$\{X_1 = v_1, X_2 = v_2, \dots, X_k = v_k\}$$

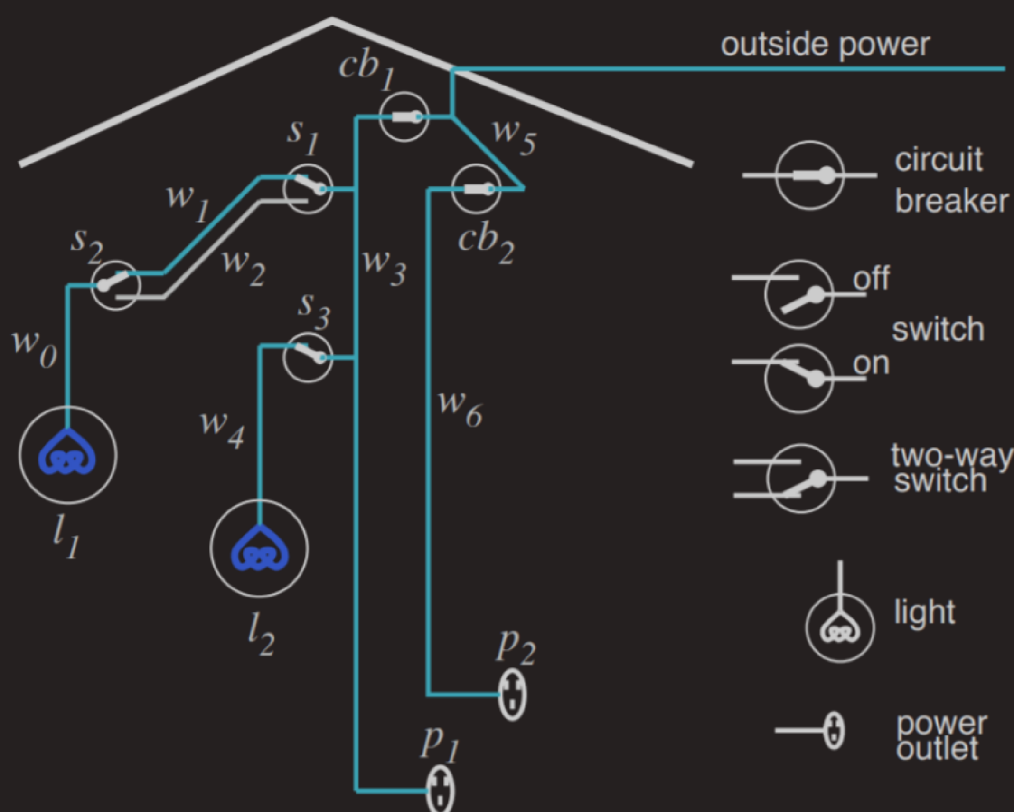
Essendo una *funzione* a ogni variabile viene assegnato un solo valore. Un'**assegnazione totale** riguarda tutte le variabili, altrimenti si dice **parziale**. Un'assegnazione totale rappresenta uno *stato* del mondo, detto *mondo possibile*.

Esempio — variabili e assegnazioni

- *Ora_Lezione* discreta
 - per denotare l'ora d'inizio
 $dom(Ora_Lezione) = \{9, 10, 11, 12, 13, 14, 15, 16, 17, 18\}$
- *Temperatura* continua
 - in °C: $dom(Temperatura)$ intervallo di reali $[-273.15; 50]$
- *Piove* booleana casuale
 - indica se stia piovendo o meno in un dato momento

L'assegnazione $\{Ora_Lezione = 11, Temperatura = 21.3, Piove = false\}$ specifica che "la lezione inizia alle 11, ci sono 21.3°C e non piove".

Esempio — diagnostica impianto elettrico



- esempi di variabili associate alle diverse componenti:
 - $S1_pos$ variabile binaria: indica la *posizione* del deviatore (switch) $s1$, dominio: $\{up, down\}$;
 - $S1_st$ variabile discreta: indica lo *stato* di $s1$, dominio: $\{ok, upside_down, short, intermittent, broken\}$;
 - $Number_of_broken_switches$ variabile intera: indica il *numero di deviatori rotti*;
 - $Current_w1$ variabile continua: indica la *corrente*, in Ampère, che passa per il cavo (wire) $w1$.
- Un'assegnazione totale specifica le posizioni, lo stato di ogni dispositivo, ecc.
 - ad es. $S1_pos = up, S2_pos = down, Cb1_st = ok, W3_st = broken, \dots$

Esempio — cruciverba

Possibili rappresentazioni in termini di variabili:

1. una per *definizione*, con *direzione* (orizzontale o verticale) e *numero* (casella iniziale)
 - dominio: parole di una data lunghezza
 - ad es., D_{2v} o *Due_verticale* con dominio dato dalle parole di 3 lettere {abc, abs, ace, ade, ..., zoo, ztl, zzz}
 - un'assegnazione totale associa una parola per ogni variabile
2. una lettera per ciascuna *casella*
 - dominio: insieme delle lettere dell'alfabeto
 - ad es., C_{11} casella in alto a sinistra con dominio {a, ..., z}
 - un'assegnazione totale associa una lettera a ogni casella

Esercizio — Trovare una rappresentazione per la soluzione di Sudoku.

Esempio — *guida turistica*

Si intendono pianificazione delle attività escursionistiche usando:

- due variabili per attività che indicano una *data*, ossia i giorni da dedicare all'attività e il *luogo*, dato l'insieme delle città da visitare;
- un'assegnazione totale assocerà *data* e *luogo* a ogni attività

In alternativa si possono rappresentare

- le *date* come variabili con l'insieme di tutte le coppie *attività-luogo* come dominio
- il numero di assegnazioni totali è il prodotto delle cardinalità dei domini delle variabili

Esempio — *colorazione grafi* (carte geografiche) con numero di colori limitato.

Esempio — Date le variabili A e B con $\text{dom}(A) = \{0, 1, 2\}$ e $\text{dom}(B) = \{\text{true}, \text{false}\}$

Le assegnazioni totali possibili sono:

- $w_0 = \{A = 0, B = \text{true}\}$
 - $w_1 = \{A = 0, B = \text{false}\}$
 - $w_2 = \{A = 1, B = \text{true}\}$
 - $w_3 = \{A = 1, B = \text{false}\}$
 - $w_4 = \{A = 2, B = \text{true}\}$
 - $w_5 = \{A = 2, B = \text{false}\}$
-

Compattezza della Rappresentazione

Date n variabili, con domini di cardinalità d si hanno d^n assegnazioni totali.

L'uso delle variabili offre un *vantaggio*: con poche variabili si descrivono molti stati:

- con 10 variabili binarie: $2^{10} \approx 10^3$ stati
- con 20 variabili binarie: $2^{20} \approx 10^6$ stati
- con 30 variabili binarie: $2^{30} \approx 10^9$ stati

- con 100 variabili binarie: $2^{100} \approx 10^{30}$ stati (= 1267650600228229401496703205376 stati)

Ragionare con 30 variabili è più facile che con un miliardo di *stati*, ma anche con 100 variabili non ci sarebbero grossi problemi, mentre è *impraticabile* ragionare esplicitamente con 2^{100} stati.

Tuttavia molti *problemi reali* possono essere definiti solo in termini di migliaia o anche milioni di variabili, come ad es. le previsioni meteo.

1.2 Vincoli

Dato un problema, le assegnazioni possono essere *ammissibili* o *non ammissibili*.

Un **vincolo rigido** / *hard constraint* specifica le assegnazioni lecite per una o più variabili e comprende: un **ambito** (o *scope*), ossia l'insieme S di variabili *coinvolte*, con una sua *arietà* $|S|$, e la **condizione**, una funzione booleana sulle assegnazioni alle variabili del vincolo che dovrà risultare vera solo per assegnazioni *lecite*.

Esempi di vincoli di diverse arietà:

- $B \leq 3$ *unario*
- $A \leq B$ *binario*
- $A + B = C$ *ternario*

La *definizione* di un vincolo può essere:

- **intensionale**, ossia in termini di formule logiche;
- **estensionale**, come elencazione delle assegnazioni lecite, come con le *relazioni* ovvero le *tabelle* di tuple nei DB relazionali.

Dati il vincolo c con ambito S e l'assegnazione A su variabili contenute in S , anche non tutte, si dirà che A **soddisfa** c se la condizione è *vera* per A ristretta all'ambito di c ovvero che A **viola** c in caso contrario.

Esempio — In una gita di 4 giorni, *attività* da svolgere rappresentate da variabili A , B , C , tutte con identico dominio $\{1, 2, 3, 4\}$.

Si consideri il vincolo su $\{A, B, C\}$ in forma *intensionale*

$$(A \leq B) \wedge (B < 3) \wedge (B < C) \wedge \neg(A = B \wedge C \leq 3)$$

che richiede che A deve venire prima o essere concomitante con B , B sia svolta prima del 3° giorno e prima di C e, infine, che se A è concomitante con B , C si svolga dopo il 3° giorno.

Lo stesso vincolo, definito *estensionalmente*, elenca le 4 assegnazioni lecite:

A	B	C
2	2	4
1	1	4
1	2	3
1	2	4

Ad es. è soddisfatto da $\{A = 1, B = 2, C = 3, D = 3, E = 1\}$ in quanto in tabella si trova $\{A = 1, B = 2, C = 3\}$, ristretta al suo ambito.

Esempio — *cruciverba*: esempi di vincoli

- dominio fatto di *parole*: le parole relative a definizioni orizzontali e verticali abbiano le stesse lettere negli incroci
 - dominio fatto di *lettere*: ogni sequenza di lettere contigue forma una parola lecita, ossia presente nel dizionario
-

1.3 Constraint Satisfaction Problem

Un **problema di soddisfacimento di vincoli** (CSP) è definito da un insieme di *variabili*, ognuna con un proprio *dominio*, e un insieme di *vincoli*. Una sua **soluzione** è un'assegnazione totale che soddisfa tutti i vincoli.

Un *CSP finito* ha un numero finito di variabili di dominio finito. Oltre a metodi per CSP finiti, si prenderanno in considerazione anche algoritmi per casi con variabili dal dominio infinito o *continuo*. Molti esempi vengono dal mondo dei giochi, quali il *Sudoku* o la *Criptoaritmetica*.

Esempio — robot consegne

Si può costruire un CSP definendo:

- le *attività* da svolgere: *a, b, c, d, e* e i *momenti* possibili: *1, 2, 3, 4*;
- le rispettive *variabili* *A, B, C, D, E*, tutte con lo stesso dominio, ossia con $\text{dom}(A) = \text{dom}(B) = \text{dom}(C) = \text{dom}(D) = \text{dom}(E) = \{1, 2, 3, 4\}$;
- l'insieme dei *vincoli*:
 $\{B \neq 3; C \neq 2; A \neq B; B \neq C; C < D; A = D; E < A; E < B; E < C; E < D; B \neq D\}$.

Per esercizio: trovare una soluzione (assegnazione che li soddisfi tutti).

Dal punto di vista algoritmico, si possono definire diversi problemi di complessità crescente legati ai CSP:

- determinare se esista una soluzione o meno;
- trovare una soluzione;
- contare il numero di soluzioni;
- enumerare tutte le soluzioni;
- trovare la soluzione migliore, data una misura di qualità;
- determinare se alcuni enunciati siano veri per tutte le soluzioni.

I CSP possono risultare difficili per il loro carattere multidimensionale (una dimensione per variabile): il compito-base è quello di *trovare una soluzione* (quando esiste). Già nel caso di CSP con domini finiti tale problema è *NP-completo*: metodi sistematici hanno una complessità esponenziale ma, ove possibile, si sfrutta la *struttura* dello spazio di ricerca.

2 Risoluzione di CSP tramite Ricerca

Algoritmo Generate-and-Test

Un algoritmo **Generate-and-Test** è un algoritmo *esaustivo* per CSP finiti semplice ma inefficiente. Per trovare *una* soluzione:

- si generano e controllano in modo sistematico, una alla volta, le possibili assegnazioni totali;
- si restituisce la prima che soddisfa *tutti* i vincoli.

Per trovare *tutte* le soluzioni

- si deve continuare a iterare, conservando le soluzioni trovate.

Esempio — Nell'esempio precedente

- spazio assegnazioni totali:

$$\mathcal{D} = \left\{ \begin{array}{l} \{A = 1, B = 1, C = 1, D = 1, E = 1\}, \\ \{A = 1, B = 1, C = 1, D = 1, E = 2\}, \\ \vdots \quad \quad \quad \vdots \quad \quad \quad \vdots \\ \{A = 4, B = 4, C = 4, D = 4, E = 4\} \end{array} \right\}$$

- $|\mathcal{D}| = 4^5 = 1024$ assegnazioni distinte da testare;
- con 15 variabili, 4^{15} ossia circa un miliardo;
- con 30 variabili, improponibile.

Osservazioni

- con n domini di cardinalità d si hanno d^n possibili assegnazioni totali;
- con e vincoli il numero totale di test è $O(ed^n)$: al crescere di n diventa rapidamente intrattabile; servono quindi soluzioni alternative.

Algoritmi su Grafo di Ricerca

L'idea conduttrice è quella di testare i vincoli su assegnazioni *parziali* di ambito limitato crescente: se un'assegnazione parziale viola un vincolo, anche le assegnazioni totali che la estendono lo violeranno quindi è possibile potare lo spazio di ricerca.

Esempio — *pianificazione consegne* (es. precedente)

- le assegnazioni con $A = 1$ e $B = 1$ violano $A \neq B$ indipendentemente dai valori assegnati alle altre variabili, quindi assegnando prima i valori ad A e B si può anticipare la scoperta di tale violazione senza considerare C , D o E e relativi test, risparmiando lavoro.

Ricerca in Profondità

DFS per cercare tutte le soluzioni di un CSP con variabili Vs e vincoli Cs , estendendo via via l'assegnazione *context*, dove:

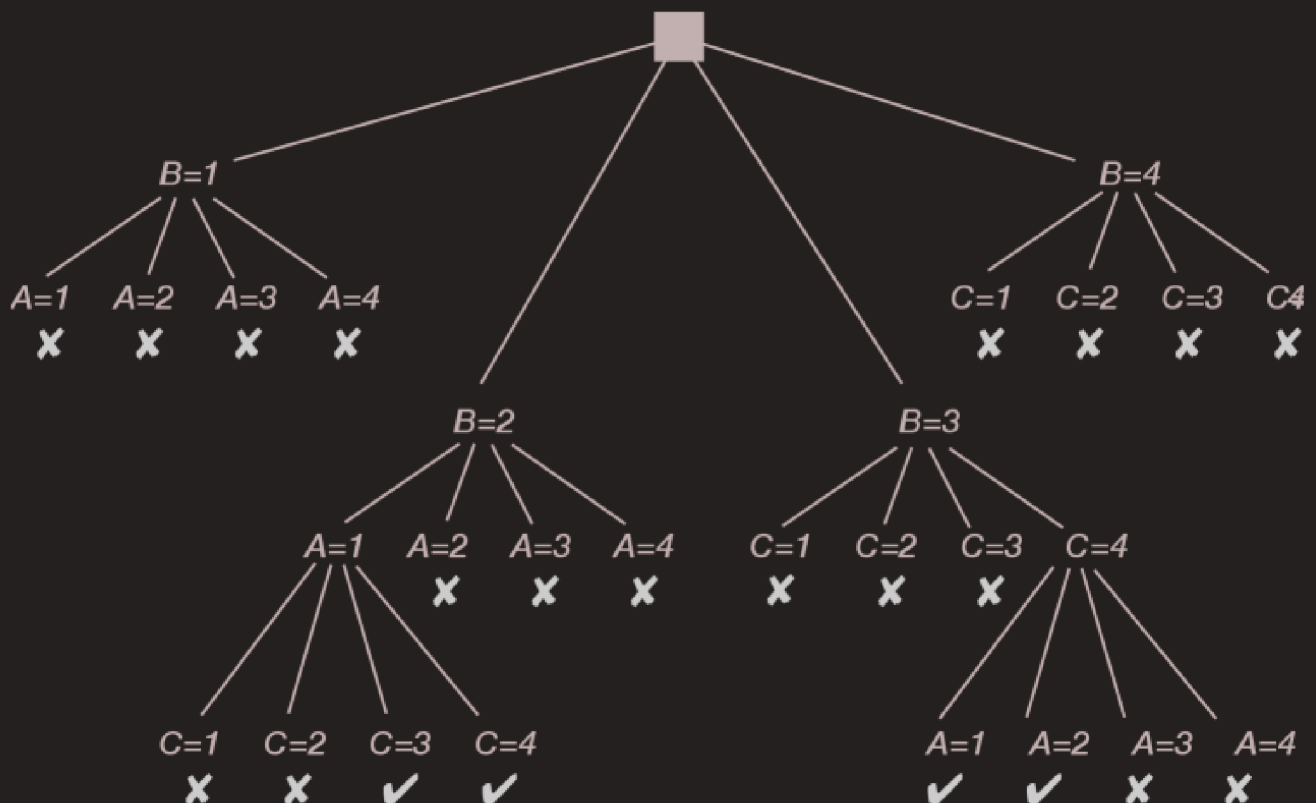
- Vs insieme delle variabili senza assegnazione in *context*;
- Cs insieme dei vincoli che coinvolgono almeno una variabile in Vs ;
- chiamata iniziale: $\text{DFS_solver}(Vs, Cs, \emptyset)$ con Vs insieme con tutte le variabili e Cs insieme con tutti i vincoli.

```
procedure DFS_solver( $Vs, Cs, context$ )
```

```
   $ce \leftarrow \{c \in Cs \mid c \text{ valutabile in } context\}$   
  if  $context$  viola un vincolo in  $ce$  then  
    return  $\emptyset$   
  else if  $Vs = \emptyset$  then  
    return  $context$   
  else  
    selezionare una variabile  $var \in Vs$   
     $sols \leftarrow \emptyset$   
    for  $val \in domain(var)$  do  
       $sols \leftarrow sols \cup DFS\_solver(Vs \setminus \{var\}, Cs \setminus ce, \{var = val\} \cup context)$   
    return  $sols$ 
```

Osservazione: l'albero di ricerca risultante ha dimensioni che dipendono dall'*ordine* di scelta delle variabili. Un ordine *statico*, ad es. sempre prima **A**, poi **B**, poi **C**, risulterà meno efficiente di uno *dinamico*. Ma l'ordine ottimale potrebbe essere più difficile da trovare.

Esempio — Dato un CSP con variabili **A**, **B** e **C**, tutte con dominio $\{1, 2, 3, 4\}$, e vincoli $A < B$ e $B < C$, il grafo ha 4 soluzioni:



- 8 assegnazioni totali e 16 parziali generate di cui si testa la consistenza, contro le $4^3 = 64$ del **GENERATE-AND-TEST**

3 Algoritmi Basati su Consistenza

Per **consistenza** s'intende la compatibilità, o coerenza logica, fra assegnazioni e vincoli.

Esempio — Nell'es. precedente A e B correlate dal vincolo $A < B$

- $A = 4$ non è consistente con ogni possibile assegnazione a B , essendo $\text{dom}(B) = \{1, 2, 3, 4\}$;
- nella ricerca con backtracking la mancata consistenza viene *riscoperta* ogni volta per le diverse assegnazioni a B e C ;
- tale inefficienza è evitabile eliminando 4 da $\text{dom}(A)$.

Rete di Vincoli

Una **rete di vincoli** (*constraint network*) indotta da un CSP è costituita da:

- un nodo (*circolare*) per ogni variabile, con un dizionario dom di possibili valori per ogni variabile X , denotati $\text{dom}[X]$, inizialmente impostato al suo dominio;
- un nodo (*rettangolare*) per ogni vincolo c ;
- un arco $\langle X, c \rangle$ per ogni variabile X nell'ambito del vincolo c .

NB *grafo bipartito*: archi fra due nodi di tipo diverso variabile e vincolo

Esempio — sempre nell'es. precedente

- *variabili*: A , B e C , tutte con dominio $\{1, 2, 3, 4\}$;
- *vincoli*: $A < B$ e $B < C$;
- *rete dei vincoli* corrispondente:



Esempio

- il vincolo $X \neq 4$ sarà rappresentato con l'arco:
 - $\langle X, X \neq 4 \rangle$
- il vincolo $X + Y = Z$ sarà rappresentato con tre archi:
 - $\langle X, X + Y = Z \rangle$
 - $\langle Y, X + Y = Z \rangle$
 - $\langle Z, X + Y = Z \rangle$

Consistenza degli Archi

L'arco $\langle X, c \rangle$ è **consistente** rispetto ai domini (*domain consistent*) sse $\forall x \in \text{dom}[X]: \{X = x\}$ soddisfa c .

Esempio — Data B con $\text{dom}[B] = \{1, 2, 3, 4\}$, si consideri il vincolo $B \neq 3$

- l'arco $\langle B, B \neq 3 \rangle$ non è consistente:
 - assegnando 3 a B si viola il vincolo;
 - eliminando 3 da $\text{dom}[B]$ l'arco tornerebbe consistente.

Consistenza degli Archi e delle Reti

Dato il vincolo c su $\{X, Y_1, \dots, Y_k\}$, l'arco $\langle X, c \rangle$ è **consistente** sse $\forall x \in \text{dom}[X], \exists y_1, \dots, y_k: y_i \in \text{dom}[Y_i]$ tale che $\{X = x, Y_1 = y_1, \dots, Y_k = y_k\}$ soddisfi c .

Una **rete consistente** (rispetto agli archi) contiene solo archi consistenti.

Osservazione: se $\langle X, c \rangle$ non è consistente allora per qualche valore di $\text{dom}[X]$ non ci sono valori di Y_1, \dots, Y_k tali che l'assegnazione risultante soddisfi c , quindi *eliminando* tali valori da $\text{dom}[X]$ si può ripristinare la consistenza di $\langle X, c \rangle$.



L'eliminazione di valori da un dominio può rendere altri archi *non consistenti*.

Esempio — Nella rete precedente



Tutti gli archi non sono consistenti dati i domini $\{1, 2, 3, 4\}$:

- $\langle A, A < B \rangle$ perché per $A = 4$ non ci sono valori per B per i quali $A < B$, togliendo 4 dal dominio di A , diventerebbe consistente;
- $\langle B, A < B \rangle$ perché non c'è un valore per A quando $B = 1$
- ...

per **Esercizio**

Algoritmo Basato sulla Consistenza degli Archi

L'idea di base è di rendere la rete consistente restringendo i domini.

Si considera l'insieme *to_do* degli archi potenzialmente non consistenti:

- si inizializza *to_do* con tutti gli archi del grafo;
- si ripete fino a svuotare *to_do*:
 - estratto un arco $\langle X, c \rangle$ da *to_do*
se $\langle X, c \rangle$ non è consistente, si deve restringere $\text{dom}[X]$;
 - aggiungere a *to_do* gli archi resi non consistenti dal passo precedente: $\langle Z, c' \rangle$, $c' \neq c$, con ambito che comprende X e una diversa Z .

Algoritmo **GENERALIZED ARC CONSISTENCY** (GAC)

```
procedure GAC( $Vs, dom, Cs, to\_do$ )  
  while  $to\_do \neq \emptyset$  do  
    seleziona e rimuovi  $\langle X, c \rangle$  da  $to\_do$   
     $\{Y_1, \dots, Y_k\} \leftarrow \text{scope}(c) \setminus \{X\}$   
     $ND \leftarrow \{x \mid x \in \text{dom}[X] \wedge \exists y_1 \in \text{dom}[Y_1], \dots, y_k \in \text{dom}[Y_k] : \\ \quad c(X = x, Y_1 = y_1, \dots, Y_k = y_k)\}$   
    if  $ND \neq \text{dom}[X]$  then  
       $to\_do \leftarrow to\_do \cup \{\langle Z, c' \rangle \mid \{X, Z\} \subseteq \text{scope}(c'), c' \neq c, Z \neq X\}$   
       $dom[X] \leftarrow ND$   
  return  $dom$ 
```

Esempio — Dato il CSP visto prima con la rete:



- Possibile sequenza di selezioni:

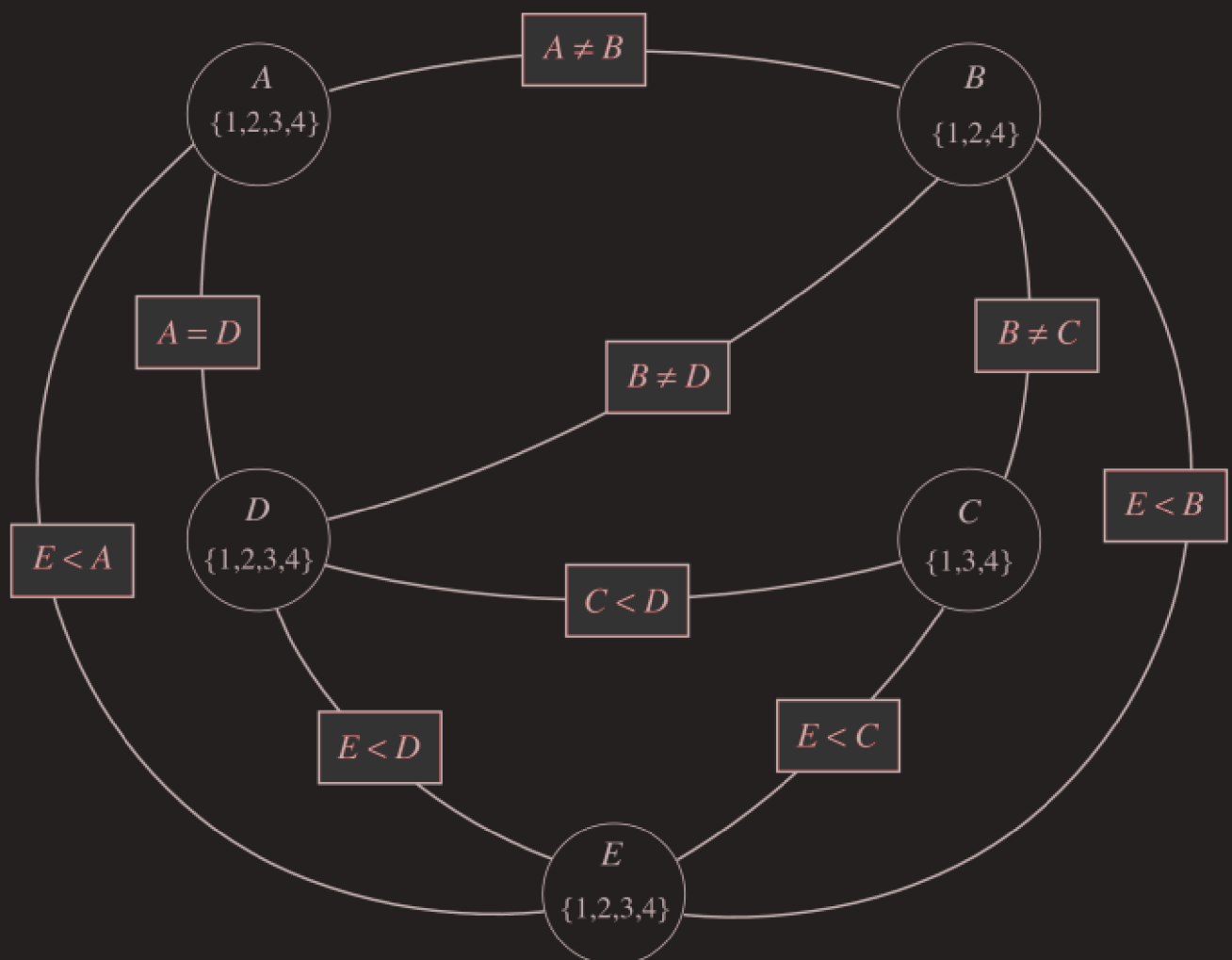
arco	dominio ridotto	aggiunto a <i>to_do</i>
$\langle A, A < B \rangle$	$dom[A] = \{1, 2, 3\}$	-
$\langle B, A < B \rangle$	$dom[B] = \{2, 3, 4\}$	-
$\langle B, B < C \rangle$	$dom[B] = \{2, 3\}$	$\langle A, A < B \rangle$
$\langle A, A < B \rangle$	$dom[A] = \{1, 2\}$	-
$\langle C, B < C \rangle$	$dom[C] = \{3, 4\}$	-

- GAC termina con $dom[A] = \{1, 2\}, dom[B] = \{2, 3\}, dom[C] = \{3, 4\}$: il problema non è risolto ma solo *semplificato*; un veloce calcolo soluzione è possibile usando la DFS con backtracking.

Esempio — Altro CSP precedente con variabili A, B, C, D, E dallo stesso dominio $\{1, 2, 3, 4\}$ e i vincoli:

$$(B \neq 3), (C \neq 2), (A \neq B), (B \neq C), (C < D), \\ (A = D), (E < A), (E < B), (E < C), (E < D), (B \neq D)$$

La rete corrispondente è la seguente:



Il resto nell'Esempio 4.19 del testo

o **Esercizio**

GAC termina con una rete consistente con domini *ridotti* e 3 casi possibili:

1. un dominio è vuoto quindi *nessuna soluzione*: se uno è vuoto, lo saranno anche altri domini ridotti ad esso connessi, già prima della terminazione;
2. i domini tutti ridotti a un solo valore quindi la *soluzione* è *unica*;
3. altrimenti, si ha un CSP *semplificato* cui applicare altri metodi.



Metodi Alternativi

Algoritmi basati sulla **consistenza dei percorsi**

4 Separazione dei Domini

L'*idea* dietro gli algoritmi di **Separazione dei Domini** è quella di decomporre il CSP in una serie di *casi disgiunti* da risolvere separatamente; le soluzioni sono ricostruite riunendo quelle trovate per i diversi casi.

Esempi

- X binaria, dominio $\{t, f\} \rightarrow$ due problemi ridotti:
 - trovare le soluzioni con $X = t$ e quelle con $X = f$
 - se ne basta una, secondo caso considerato solo se il primo non ha soluzione
- A con dominio $\{1, 2, 3, 4\}$, separabile in diverse maniere
 - un caso per ciascun valore: $A = 1, A = 2, A = 3, A = 4$
 - fa fare più strada con una sola suddivisione alla volta
 - due sottoinsiemi disgiunti: $A \in \{1, 2\}$ e $A \in \{3, 4\}$
 - taglia di più in meno passi (lavoro che non va rifatto per ogni valore)
 - ...

Nello schema di algoritmo seguente si integra l'approccio basato sulla consistenza in un algoritmo ricorsivo:

- si semplifica il CSP in input tramite **GAC()**
- se non è risolto direttamente:
 - si seleziona una variabile, con dominio almeno binario;
 - si partiziona il dominio ottenendo (2+) problemi semplificati (casi);
 - si risolvono *ricorsivamente* tali problemi

```
procedure Con_Solve( $Vs, dom, Cs, to\_do$ )  
   $dom_0 \leftarrow GAC(Vs, dom, Cs, to\_do)$   
  if  $\exists X: dom_0[X] = \emptyset$  then  
    return false  
  else if  $\forall X: |dom_0[X]| = 1$  then  
    return soluzione con  $X = x \in dom_0[X] \quad \forall X$   
  else  
    selezionare  $X$  tale che  $|dom_0[X]| > 1$   
    partizionare  $dom_0[X]$  in  $D_1$  e  $D_2$   
     $dom_1 \leftarrow$  copia di  $dom_0$  con  $dom_1[X] = D_1$   
     $dom_2 \leftarrow$  copia di  $dom_0$  con  $dom_2[X] = D_2$   
     $to\_dos \leftarrow \{ \langle Z, c' \rangle \mid \{X, Z\} \subseteq scope(c'), Z \neq X \}$   
    return Con_Solve( $\langle Vs, dom_1, Cs \rangle, to\_dos$ ) or  
           Con_Solve( $\langle Vs, dom_2, Cs \rangle, to\_dos$ )
```

Osservazioni:

- l'algoritmo fornisce *tutte* le soluzioni:
 - se un dominio vuoto non ci sono soluzioni (restituisce \perp);
 - se il dominio con un solo valore si ha una sola soluzione;
 - ritornando dalla ricorsione restituisce l'unione delle soluzioni dei 2 casi.
- c'è spazio anche per algoritmi di ricerca su grafo, ma qui contano le soluzioni
 - ad es. il DFS con spazi finiti.

Miglioramento possibile: se un'assegnazione rende il grafo *non connesso*, ogni componente può essere risolta *separatamente*. Una soluzione si ottiene ricombinando le soluzioni delle componenti. Il conteggio del numero di soluzioni è efficiente, ad es., se una componente dà 100 soluzioni, e l'altra 20 si avranno 2000 soluzioni totali.

5 Eliminazione di Variabili

L'**eliminazione di variabili** (*variable elimination*, **VE**) è una tecnica che semplifica la rete dei vincoli rimuovendo variabili. L'idea base è quella di eliminare le variabili una alla volta ottenendo CSP semplificati da risolvere e infine ricostruire le soluzioni dei CSP più complessi:

- eliminando X si costruisce un *nuovo vincolo* sulle rimanenti che rifletta gli effetti di X : esso sostituisce tutti i vincoli su X producendo una rete semplificata (CSP ridotto);
- alla fine, ogni soluzione del CSP ridotto va *estesa* per ottenere una soluzione del CSP comprendente X .

Eliminazione di una variabile:

Data X da eliminare:

1. considerate le relazioni di tutti i vincoli su X ,
sia $r_X(X, \bar{Y})$ il **join** di tali relazioni *influenza* di X sulle altre
 - \bar{Y} : ins. delle altre variabili nell'ambito di r_X *vicine* di X nel grafo dei vincoli
2. la *proiezione* di r_X su \bar{Y} sostituisce tutte le relazioni in cui occorre X

3. si ottiene un CSP ridotto, senza X , da risolvere *ricorsivamente*:
 - al *ritorno*, si estendono le tabelle-soluzioni per il CSP ridotto tramite **join** con r_X , per aggiungere la colonna delle assegnazioni a X
 - *caso base*: resta una sola variabile \rightarrow soluzione da restituire
 - = tabella con i valori del dominio consistenti con i vincoli

Esempio — Si consideri un CSP su A, B, C di dominio $\{1, 2, 3, 4\}$ e sia B la variabile da eliminare, inclusa nei vincoli: $A < B$ e $B < C$ (altre variabili possibili con i rispettivi vincoli). Per eliminarla, si fa il **join** tra le relazioni dei vincoli su B :

A	B		B	C		A	B	C
1	2		1	2	=	1	2	4
1	3		1	3		1	2	3
1	4	\bowtie	1	4		1	2	4
2	3		2	3		1	3	4
2	4		2	4		2	3	4
3	4		3	4				

- la sua proiezione su A e C induce una nuova relazione senza B :

A	C
1	3
1	4
2	4

- vincolo che sostituisce tutti quelli su B
 - che contiene tutte le info utili alla soluzione del resto della rete;
- con **VE** poi si risolve il resto della rete semplificata.

Per avere una/tutte le soluzioni a partire dalla soluzione del CSP ridotto: si memorizza la relazione del **join** su A, B, C per estendere la soluzione della rete ridotta includendo B .

```
procedure VE_CSP( $Vs, Cs$ )
```

Input

Vs : insieme di variabili
 Cs : insieme di vincoli su Vs

Output

relazione contenente tutte le assegnazioni consistenti

if $|Vs| = 1$ then

return join di tutte le relazioni in Cs

else

Selezionare $X \in Vs$ da eliminare

$CX \leftarrow \{c \in Cs \mid c \text{ coinvolge } X\}$

$R \leftarrow$ join di tutti i vincoli in CX

$NR \leftarrow$ proiezione di R sulle variabili $Vs \setminus \{X\}$

$S \leftarrow \text{VE_CSP}(Vs \setminus \{X\}, (Cs \setminus CX) \cup \{NR\})$

return $R \bowtie S$

Osservazioni

- *caso base*: rimane una sola variabile, quindi una soluzione esiste se ci sono righe nelle relazioni finali e saranno tutte relative a una sola variabile (insiemi di valori leciti) basterà intersecarle;

- caso *ricorsivo*: l'ordine di selezione delle variabili ha un impatto sull'efficienza; *al ritorno* se bastasse una sola soluzione, si restituisce solo una tupla di $R \bowtie S$. È garantito che sia parte d'una soluzione: se un valore di R non avesse tuple, non ci sarebbero soluzioni con tale valore.

Estensioni: **VE** può essere *combinato* con algoritmi basati su consistenza da usare per semplificare il problema quando si elimina una variabile. Le tabelle intermedie risulteranno più piccole.

6 Ricerca Locale

Cosa fare in caso di spazi molto grandi o, addirittura, *infiniti*?

Non è pensabile una ricerca sistematica dell'intero spazio. Si può puntare su metodi *mediamente efficienti* per trovare soluzioni, ma *senza garanzie*, anche quando esistono. Saranno quindi utili quando si sa già che, verosimilmente, ne esistono.

La classe dei metodi di **ricerca locale**, comunemente investigati in Ricerca Operativa e AI, comprende molte tecniche, con uno stesso schema-base:

- si inizia con un'assegnazione totale di un valore a ciascuna variabile;
- si tenta di migliorare l'assegnazione iterativamente effettuando passi di *miglioramento*, passi *casuali* e *ripartenze* da assegnazioni iniziali differenti.

```

procedure Local_search( $Vs, dom, Cs$ )

  Input
     $Vs$ : insieme di variabili
     $dom$ : funzione che restituisce il dominio di una variabile
     $Cs$ : insieme di vincoli da soddisfare
  Output
    assegnazione totale che soddisfa i vincoli
  Local
     $A$  dizionario di valori indicizzato dalle variabili in  $Vs$ 
  repeat // try
    for each  $X \in Vs$  do
       $A[X] \leftarrow$  valore (casuale) da  $dom(X)$ 
    // walk
    while not stop_walk() and  $A$  non soddisfa  $Cs$  do
      Selezionare  $Y \in Vs$  e un valore  $w \in dom(Y)$ 
       $A[Y] \leftarrow w$ 
    if  $A$  soddisfa  $Cs$  then
      return  $A$ 
  until terminazione
  
```

Ogni iterazione della **repeat** rappresenta un **tentativo** (*try*):

- con il primo **for each** si ha l'**inizializzazione casuale** di A ;
- per ogni assegnazione casuale successiva si ha una **ripartenza casuale** (*random restart*) in alternativa anche congetture più informate basate su euristiche o conoscenza pregressa, poi migliorate iterando.

Nel ciclo **while** si effettua una **ricerca locale** (*walk*) nello spazio delle assegnazioni:

- si seleziona una assegnazione tra i possibili **successori** di A che differiscono per il valore assegnato a una sola variabile;

- si ha lo *stop* se una soluzione è stata trovata o si avvera il criterio di `stop_walk()`, ad esempio raggiunto un numero massimo di iterate.

La fermata *non* è *garantita*: può divergere se il CSP non ha soluzione:

- in alcuni casi, anche se ne esistessero, potrebbe rimanere intrappolato in una regione;
- una garanzia di *completezza* dipende dai criteri di selezione e di stop.

Random Sampling

Random Sampling è la versione della ricerca locale in cui:

- `stop_walk()` risulta sempre `true` quindi il ciclo `while` non viene mai eseguito: si continua indefinitamente a provare assegnazioni casuali che possano soddisfare tutti i vincoli;
- l'algoritmo risulta *completo* ossia garantisce di trovare la soluzione se questa esiste, tuttavia il tempo richiesto non può essere limitato e tipicamente risulta molto lento;
- la sua efficienza dipende dalle dimensioni dei domini (loro prodotto) e dal numero di soluzioni esistenti.

Random Walk

Random Walk è la versione della ricerca locale in cui:

- `stop_walk()` risulta sempre `false` per cui non si hanno ripartenze casuali: si esce dal ciclo `while` solo se si trova una soluzione e, nel ciclo, si ripete la selezione casuale di una variabile e un valore da assegnarle;
- l'algoritmo è *completo*, con passi più veloci rispetto al resampling di tutte le variabili, ma può richiedere più passi, in base alla distribuzione delle soluzioni;
- quando le dimensioni dei domini delle variabili differiscono, si può, in alternativa, selezionare a caso una variabile e poi un valore del suo dominio *oppure* selezionare casualmente una coppia variabile-valore, il che favorisce la selezione di variabili con dominio più grande.

6.1 Massimo Miglioramento Iterativo

L'**ITERATIVE BEST IMPROVEMENT** è un metodo di ricerca locale con selezione del *miglior successore* (un'assegnazione) in termini di una **funzione di valutazione**. Essa sarà da minimizzare, nel caso di **GREEDY DESCENT** oppure da massimizzare, nel caso di **GREEDY ASCENT** (detto anche HILL CLIMBING). Nel seguito si considereranno solo funzioni da minimizzare, per l'altro metodo sarà sufficiente cambiare il segno della funzione. Nei casi di *parità* di valore (*tie*) della funzione si opera una scelta casuale. Una funzione di valutazione tipica per i CSP è il numero di **conflitti**, i.e. dei vincoli violati: con **0** conflitti si ha una soluzione. Tale funzione può essere raffinata *pesando* i vincoli in maniera differenziata.

Si distinguono:

- **ottimi locali**, ossia assegnazioni non migliorabili da alcun successore, come i *minimi* o *massimo locali* trovati, rispettivamente, con **GREEDY DESCENT** o **ASCENT**;
- **ottimi globali**, quelli con valutazione massima fra tutte le assegnazioni, che risultano comunque anche ottimi locali.

Ottimizzando il *numero di conflitti*, un CSP è *soddisfacibile* se si trova un ottimo globale con valore nullo ovvero è *non soddisfacibile* con un ottimo globale con valore positivo: se si trova

un minimo locale con valore positivo, non è detto che esso sia globale (e quindi il CSP non soddisfacibile) o meno.

Esempio — ancora esempio precedente sulle consegne:

Se **GREEDY DESCENT** inizia da $\{A = 2, B = 2, C = 3, D = 2, E = 1\}$ si hanno **3** conflitti: $A \neq B, B \neq D, C < D$.

Tracciando le assegnazioni (evidenziando il nuovo valore assegnato a ogni passo):

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	valutazione
2	2	3	2	1	3
2	4	3	2	1	1
2	4	3	4	1	2
4	4	3	4	1	2
4	2	3	4	1	0

- con **0** conflitti si ha una *soluzione*.



Con diverse inizializzazioni o diverse scelte in caso di pari valutazione possibili sequenze di assegnazioni e risultati differenti.

Incompletezza: si considera il *miglior successore* anche quando questo *non ha* una migliore valutazione rispetto all'assegnazione corrente, e.g. minor numero di conflitti. È possibile che l'algoritmo trovi ottimi locali che risultano successori *reciproci* e che continui a passare da uno all'altro senza poter trovare una soluzione. Questo rende l'algoritmo *incompleto*.

6.2 Algoritmi Stocastici

Per evitare minimi locali che non siano anche globali, negli **algoritmi stocastici** si ricorre a una maggiore *casualità*. Le mosse casuali previste saranno:

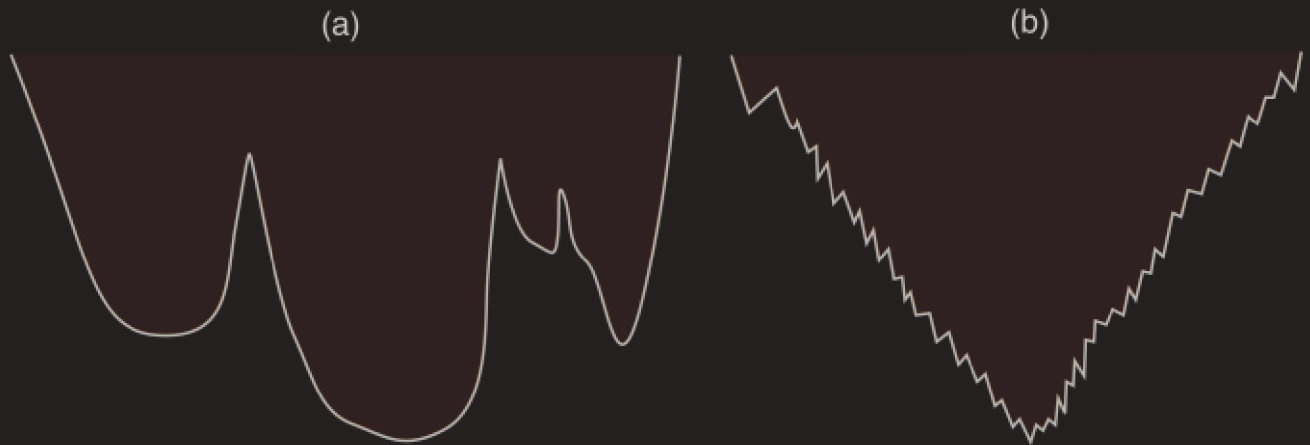
1. il **random restart** in cui valori scelti a caso per tutte le variabili: costituisce una mossa casuale *globale* (più costosa) che consente di ripartire da regioni anche completamente diverse dello spazio;

2. il **random step**, una mossa casuale *locale* da alternare a passi di ottimizzazione: utilizzata nel **GREEDY DESCENT** / **GREEDY ASCENT** permette passi in direzione opposta in modo da sfuggire a minimi / massimi locali.

Integrando *massimo miglioramento iterativo* e mosse *casuali* si definiscono algoritmi per la **ricerca locale stocastica**.

Esempio — Spazio 2D

- *successore* tramite piccolo passo dall'attuale posizione
 - verso sinistra o destra



- spazio di ricerca (a): **GREEDY DESCENT** può trovare facilmente minimi locali, serve un **RANDOM RESTART** che porti nella parte centrale (più profonda) nella quale si converge rapidamente verso uno globale; invece **RANDOM WALK** non funzionerebbe bene: richiederebbe molti piccoli passi casuali per uscire da un minimo locale;
- spazio di ricerca (b): **RANDOM RESTART** rimane bloccato a cercare tra numerosi minimi locali mentre **RANDOM WALK** con **GREEDY DESCENT** potrebbe evitare tali minimi locali (pochi passi casuali spesso sufficienti).

Con spazi con diverse caratteristiche in regioni differenti si può pensare di ricorrere a un cosiddetto **algorithm portfolio**.

7 Varianti della Ricerca Locale

Molte altre varianti sono possibili nella *scelta del successore* e nelle scelte *casuali*.

Successori e Domini

Con domini *limitati*, tutti i valori possono essere scelti per i successori. Con domini più *estesi*, si possono considerare solo alcuni valori, risparmiando tempo, in genere si considerano solo quelli *più vicini* ai precedenti, ma esistono metodi più sofisticati di selezione.

Tabu Search

La **tabu search** è una forma di ricerca locale *con memoria*, che evita la modifica di assegnazioni introdotte di recente. Si memorizzano le variabili modificate negli ultimi *t* passi (**tenure**) che andranno considerate *non selezionabili*. In tal modo si evitano i cicli dopo poche assegnazioni.

Va ottimizzato il parametro *t*. Nella sua implementazione, se è *piccolo* si può considerare una lista delle variabili modificate di recente, invece se è *grande*, si memorizza per ogni variabile il passo in cui si è avuto l'ultima sua modifica.

Passo di Massimo Miglioramento

In questo metodo di ricerca locale, si seleziona una *coppia* variabile-valore che porta al *miglioramento* di valutazione *massimale*. Non trattandosi necessariamente di un'unica coppia in caso di più coppie si opera una scelta casuale.

Nella sua implementazione *ingenua*, data l'assegnazione totale corrente, per ogni variabile X e ogni valore $v \in \text{dom}(X)$ diverso da quello corrente, si confronta l'assegnazione corrente con quella in cui si assegna $X = v$. Si seleziona quindi una delle coppie di *massimo miglioramento*. Si noti che la modifica potrebbe portare a *differenze negative* nella valutazione, ossia a peggioramenti. Inoltre, le variabili non coinvolte in vincoli possono essere trascurate.

In un'implementazione *alternativa*, si adotta una *coda con priorità* fatta di coppie variabile-valore pesate. Per ogni X e ogni $v \in \text{dom}(X)$ non assegnato in A (assegnazione corrente), in coda ci sarà una coppia $\langle X, v \rangle$ con *peso* $w = h(A') - h(A)$. Si considera allora il miglioramento dell'assegnazione A' ottenuta sostituendo $X = v$ rispetto ad A . Questo dipenderà dai valori assegnati a X e dai suoi vicini nella rete dei vincoli, non da quelli assegnati alle altre variabili. A ogni iterata si seleziona una *coppia-successore* di massimo miglioramento, ossia con *peso minimale*. Ogni nuova assegnazione comporta il ricalcolo dei pesi e il riordinamento della coda, ma solo per coppie con variabili presenti in vincoli il cui soddisfacimento è mutato.

Scelta a Due Fasi

L'algoritmo di **scelta a due fasi** prevede la selezione della coppia articolata in:

1. selezione della variabile per la riassegnazione
2. selezione del valore

Si gestisce una *coda* con priorità di variabili con associato un peso pari al *numero dei conflitti* in cui ognuna sia coinvolta. Ad ogni passo: 1) si seleziona X che partecipa a *più conflitti* e 2) le si cambia il valore assegnato, scegliendolo fra quelli che minimizzano il numero di conflitti *oppure* casualmente. Vanno infine ricalcolati i pesi per le variabili coinvolte in vincoli il cui soddisfacimento è mutato.

Algoritmo Any-Conflict

L'idea-base dell'algoritmo **Any-Conflict** è quella di scegliere, per la modifica, una **variabile conflittuale** ossia una che partecipa a uno o più conflitti. A ogni iterata: si seleziona casualmente una *_variabile conflittuale*, non necessariamente quella con più conflitti, quindi le si assegna, in alternativa, un valore che *minimizzi* il numero di conflitti *oppure* un valore casuale.

Sono possibili alcune varianti, ad es. in base al criterio di selezione casuale della variabile: si può scegliere prima un conflitto e poi una variabile coinvolta oppure si opera una semplice scelta casuale di una variabile conflittuale. La differenza dei due criteri sta nella probabilità di selezione di una variabile: nel primo essa dipende dal numero di conflitti in cui la variabile è coinvolta, nel secondo si ha la stessa probabilità per tutte le variabili.

Simulated Annealing

Il **simulated annealing** [RN16] prende in prestito una metafora dal dominio della *metallurgia* (ovvero dalla *termodinamica*): ad *alte* temperature deve comportarsi con maggiore casualità o plasticità (consentendo assegnazioni peggiorative) mentre a *basse* temperature deve consentire minore casualità in conseguenza di una maggiore durezza.

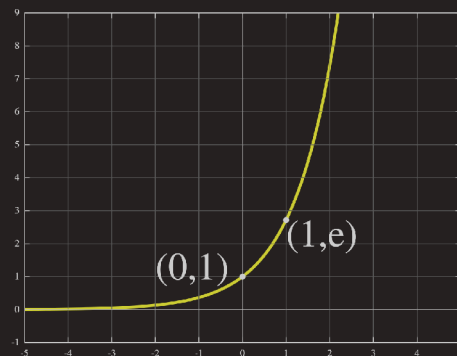
Come funzione di valutazione si adotta un'*euristica* basata sul numero di conflitti (da minimizzare).

L'algoritmo riduce lentamente la *temperatura* cui è legata una misura di probabilità:

- ad *alte temperature* si deve comportare come **RANDOM WALK** per poter evitare i minimi locali, alla ricerca di regioni con bassi valori dell'euristica, quindi i passi *peggiorativi* dovranno risultare più probabili;
- a *basse temperature*: si deve comportare come **GREEDY DESCENT**, portando direttamente verso i minimi (locali).

A ogni passo, data l'assegnazione *corrente* A : si sceglie a caso una variabile e un valore ottenendo una *nuova* assegnazione A' ; se A' non peggiora l'euristica andrà a rimpiazzare l'assegnazione corrente, altrimenti può farlo ma con una *probabilità* che dipende dalla temperatura e del valore dell'euristica.

Temperatura $T \in \mathbb{R}_+$



- sia $h(A)$ l'*euristica* da minimizzare, (tipicamente) il numero dei conflitti legati ad A ;
- se $h(A') \leq h(A)$ (A' migliorativa), si accetta direttamente la nuova A' altrimenti (A' peggiorativa), la si può accettare con *probabilità* $e^{-(h(A')-h(A))/T}$ (**distribuzione di Gibbs / Boltzmann**). Si noti che se $h(A') > h(A)$, l'esponente è negativo e tendendo $h(A') - h(A)$ a 0, sarà più probabile accettare A' . Infatti ad alte temperature l'esponente tende a 0 e la probabilità a 1, mentre a basse temperature, l'esponente tende a $-\infty$ e la probabilità a 0.

Esempio — Probabilità di accettazione di passi peggiorativi a diverse temperature T e differenze $k = h(A') - h(A)$:

T	$k = 1$	$k = 2$	$k = 3$
10	0.9	0.82	0.74
1	0.37	0.14	0.05
0.25	0.018	0.0003	0.000006
0.1	0.00005	$0.2 \cdot 10^{-8}$	$0.9 \cdot 10^{-13}$

Il cosiddetto **programma di annealing** specifica come ridurre la temperatura al progredire della ricerca: molto usato il *raffreddamento geometrico*, ad es. si parte da $T = 10$ e si moltiplica per 0.99 ad ogni passo, arrivando a 0.07 dopo 500 passi.

- a temperature *alte* (e.g. $T = 10$) si tende ad accettare passi che peggiorano di poco, con una leggera preferenza rispetto a passi che migliorano;
- a temperature *ridotte* (e.g. $T = 1$) i passi peggiorativi sono accettati molto meno frequentemente;
- a temperature *basse* (e.g. $T = 0.1$) i passi peggiorativi sono accettati molto raramente.

Ripartenza Casuale

L'algoritmo **random restart** (*ripartenza casuale*) permette di migliorare le prestazioni di un *algoritmo casuale debole*, uno che abbia successo in pochi casi specifici. Nel seguito sarà indicata con p la probabilità di successo d'una singola esecuzione. Per stimare le prestazioni di **RANDOM RESTART**, si considera una *sequenza* di n sue esecuzioni *indipendenti*. La probabilità di successo in almeno una di tali esecuzioni sarà:

$$1 - (1 - p)^n$$

essendo $(1 - p)^n$ la probabilità di fallimento in *tutti* gli n tentativi.

Esempio — Si consideri un algoritmo con $p = 0.5$

- ripetuto per 5 volte, trova una soluzione circa il 96.9% delle volte
- ripetuto 10 volte: 99.9%

Se $p = 0.1$

- ripetuto 10 volte: 65% di percentuale di successo
 - ripetuto 44 volte: 99%
-

RANDOM RESTART diventa *costoso* se sono coinvolte molte variabili. Nella sua variante **PARTIAL RESTART** si fanno assegnazioni solo ad *alcune* variabili, ad es. una data *percentuale* di esse, per consentire di spostarsi verso un'altra regione. In tal caso le esecuzioni non sono più *indipendenti* fra loro quindi si richiede un'analisi teorica più complessa.

8 Algoritmi Basati su Popolazioni

A differenza dagli algoritmi esaminati in precedenza che considerano un'assegnazione alla volta, tali metodi gestiscono **popolazioni** di **individui**, ossia insiemi di assegnazioni. Nella *beam search* si considerano le migliori k , variabile casualmente nella sua variante stocastica. Negli algoritmi evolutivisti si considerano i/le migliori k nella metafora riproduttiva.

9 Beam Search

La **beam search** si comporta in maniera simile agli algoritmi di massimo miglioramento iterativo, con la differenza di lavorare su più assegnazioni (fino a k) anziché su una sola. Essa ha *successo* quando viene trovata un'assegnazione soddisfacente. A ogni passo, si selezionano i migliori k *successori* (o anche meno qualora non ce ne fossero a sufficienza). La selezione è casuale in caso di parità di valutazione. Si itera con il nuovo insieme di k assegnazioni.

In caso di *memoria limitata* si sceglie k in base alla memoria disponibile. È possibile usare le diverse *varianti* di ricerca locale viste in precedenza. Occorrerà ovviamente più tempo nel cercare i migliori k successori. Per una ricerca più rapida, si può ricorrere ad approssimazioni dei migliori k individui.

Beam Search Stocastica

In tal caso si selezionano k individui *casualmente*, favorendo quelli con valutazione migliore. La probabilità di selezione sarà definita in funzione dell'euristica: per un individuo A

probabilità di selezione (*distribuzioni di Gibbs / Boltzmann* vista in precedenza) proporzionale a

$$e^{-h(A)/T}$$

essendo $h(A)$ la funzione di valutazione e una misura della T temperatura.

Questa variante consente più *diversità* nella popolazione: h riflette la capacità di *adattamento (fitness)* come in *biologia* dove vale il principio di *sopravvivenza dei più adattabili* per cui un individuo più *adattabile* ha maggiori probabilità di passare il proprio materiale genetico alle future generazioni. Questa è interpretabile come una forma di *riproduzione asessuata* in cui si produce una discendenza leggermente *mutata*. Si osservi che gli stessi individui potranno essere selezionati casualmente anche più volte.

Algoritmi Genetici

Anche questi metodi gestiscono *popolazioni* di individui. Restando in un'analogia metafora evolutiva, ogni assegnazione rappresenta il patrimonio genetico d'un individuo. Nuovi individui vengono creati dalla *combinazione* di *coppie* di *genitori* della generazione precedente. Un'operazione di combinazione tipicamente utilizzata è il **crossover** che prevede la *selezione* di una coppia di individui e la conseguente *generazione* della loro *prole*, ossia di individui ottenuti copiando parte delle assegnazioni da un genitore e il resto dall'altro.

NB operazione *aggiuntiva* rispetto alla mutazione.

Sono possibili diverse tecniche di selezione:

- nella *selezione proporzionale alla fitness* a ogni iterata, si *generano* k nuovi individui. Si opera una selezione casuale delle coppie, che favorisca gli individui più adatti con una probabilità dipende dall'incremento della misura di fitness apportato e da una misura della temperatura. Per ogni coppia, si opera un *crossover* e si fanno *mutare* casualmente alcuni (pochissimi) valori, per alcune variabili scelte a caso (il che richiama una forma di **RANDOM WALK**). Si passa infine a considerare la successiva generazione.

- nella *selezione a torneo*, si selezionano t individui (misura di *greediness*) e poi si scelgono i più adattabili fra questi.

```

procedure Genetic_algorithm( $Vs, Cs, S, k$ )

  Input
     $Vs$ : insieme di variabili
     $Cs$ : insieme di vincoli da soddisfare
     $S$ : programma di raffreddamento della temperatura
     $k$ : dim. popolazione - intero pari

  Output
    assegnazione totale che soddisfa i vincoli

  Locali
     $Pop$ : insieme di assegnazioni
     $T$ : n. reale

   $Pop \leftarrow k$  assegnazioni totali casuali
   $T$  inizializzato in base a  $S$ 
  repeat
    if  $A \in Pop$  soddisfa tutti i vincoli in  $Cs$  then
      return  $A$ 
     $Npop \leftarrow \emptyset$ 
    repeat  $k/2$  volte
       $A_1 \leftarrow \text{Random\_selection}(Pop, T)$ 
       $A_2 \leftarrow \text{Random\_selection}(Pop, T)$ 
       $N_1, N_2 \leftarrow \text{Crossover}(A_1, A_2)$ 
       $Npop \leftarrow Npop \cup \{\text{mutate}(N_1), \text{mutate}(N_2)\}$ 
     $Pop \leftarrow Npop$ 
     $T$  viene aggiornato in base a  $S$ 
  until terminazione

procedure Random_selection( $Pop, T$ )

  selezionare  $A$  da  $Pop$  con probabilità proporzionale a  $e^{-h(A)/T}$ 
  return  $A$ 

procedure Crossover( $A_1, A_2$ )

  selezionare casualmente un intero  $i$ ,  $1 \leq i < |Vs|$ 
   $N_1 \leftarrow \{(X_j = v_j) \in A_1 \mid j \leq i\} \cup \{(X_j = v_j) \in A_2 \mid j > i\}$ 
   $N_2 \leftarrow \{(X_j = v_j) \in A_2 \mid j \leq i\} \cup \{(X_j = v_j) \in A_1 \mid j > i\}$ 
  return  $N_1, N_2$ 

```

Vi sono diverse forme di crossover:

- **crossover uniforme**: si considerano due individui *genitori* e si generano due *figli*; per ogni variabile, si copia nel primo figlio il valore da uno dei genitori scelto in maniera casuale, quello dell'altro sarà copiato nel secondo;

- **one-point crossover**, molto comune: assumendo un *ordinamento* sulle variabili, si seleziona casualmente un *indice* i per generare ciascun figlio selezionando i valori per le variabili fino a i da un genitore e per le successive ($> i$) dall'altro (per l'altro figlio si agisce in modo *complementare*). L'*efficacia* dipenderà dall'ordinamento scelto in fase di progettazione.
-

Esempio — Nell'esempio precedente, si usa una funzione di costo basata sul numero di conflitti:

- L'assegnazione $A = 2, B = 2, C = 3, D = 1, E = 1$ ha un costo pari a 4
 - basso grazie a $E = 1$
 - un discendente che erediti $E = 1$ tenderà ad avere un costo più basso (sopravvivenza più probabile dei più adatti)
 - Altri individui hanno una valutazione bassa: $A = 4, B = 2, C = 3, D = 4, E = 4$ ha costo 4
 - a causa delle assegnazioni su A, B, C, D i figli che preservano questa proprietà saranno più adatti rispetto agli altri, candidandosi alla sopravvivenza
 - Combinando questi individui, tra i discendenti alcuni erediteranno cattive proprietà e non saranno scelti per tramandarle, altri quelle buone e avranno maggiori probabilità di sopravvivenza.
-

10 Ottimizzazione

A volte si hanno solo informazioni sulla preferibilità delle assegnazioni. Un **problema di ottimizzazione** richiede di trovare le *migliori* assegnazioni totali. Esso può essere formalizzato come segue:

- *dati*:
 - un insieme di *variabili* con un *dominio* associato;
 - una **funzione-obiettivo** dalle assegnazioni totali a \mathbb{R} , tipicamente una funzione di costo/errore/loss;
 - un **criterio di ottimalità**, tipicamente *minimizzare* la funzione-obiettivo;
- *trovare* un'assegnazione totale **ottimale** per il criterio adottato.

Un **problema di ottimizzazione vincolato** prevede anche *vincoli rigidi* che specificano le assegnazioni *ammissibili*, i.e. che soddisfino i vincoli (*feasible*). L'*obiettivo* è quello di trovare un'assegnazione ottimale ammissibile.

Esempio — *schedulazione esami*

cfr. [PM23] es. 4.29

- rigido: ogni esame si svolge in un'aula disponibile alla data ora
 - flessibile: min. #studenti con esami nello stesso giorno
-

Sul tema dell'ottimizzazione esiste una vasta letteratura scientifica [Optimization], con molte tecniche proposte: ad es. nelle tecniche di *programmazione lineare* [LP] sono coinvolte variabili continue, funzioni-obiettivo lineari e disequazioni lineari come vincoli. Quando un problema da risolvere, prevedendo anche dopo qualche trasformazione, rientra in *categorie classiche* converrà preferire algoritmi specifici.

In un *problema* di ottimizzazione **fattorizzato**, la funzione-obiettivo può essere fattorizzata in un insieme di **vincoli flessibili**, ciascuno con un *ambito* di variabili. Si considerano funzioni di **costo** che mappano i domini delle variabili su \mathbb{R} . Il criterio di ottimalità tipicamente utilizzato è la *minimizzazione della somma* dei costi dei vincoli flessibili.

Esempio — consegne con preferenze sui tempi invece che vincoli rigidi: costi associati alle *combinazioni* di valori (tempi)

- *obiettivo*: trovare una *schedulazione* con somma totale dei costi minimale;
- date A, C, D ed E con dominio $\{1, 2\}$ e B con dominio $\{1, 2, 3\}$
vincoli flessibili:

	A	B	$costo$		B	C	$costo$		B	D	$costo$
$c_1 :$	1	1	5	$c_2 :$	1	1	5	$c_3 :$	1	1	3
	1	2	2		1	2	2		1	2	0
	1	3	2		2	1	0		2	1	2
	2	1	0		2	2	4		2	2	2
	2	2	4		3	1	2		3	1	2
	2	3	3		3	2	0		3	2	4

- si considererà anche $c_4(A, C)$ di costo 3 se $A = C$ e nullo altrimenti

Somma puntuale di vincoli flessibili \rightarrow vincolo flessibile con:

- *ambito*: unione degli ambiti
- *costo* di un'assegnazione alle variabili nell'ambito:
somma dei costi delle assegnazioni nei vincoli flessibili

Esempio — Dati $c_1(A, B)$ e $c_2(B, C)$ dell'es. precedente:

- $c_1 + c_2$ funzione con ambito $\{A, B, C\}$, dato da

	A	B	C	$costo$
$c_1 + c_2 :$	1	1	1	10
	1	1	2	7
	1	2	1	2
	\vdots	\vdots	\vdots	\vdots

◦ e.g.

$$(c_1 + c_2)(A = 1, B = 1, C = 2) = c_1(A = 1, B = 1) + c_2(B = 1, C = 2) = 5 + 2 = 7$$

Rispetto ai CSP, l'ottimizzazione presenta un'ulteriore difficoltà: riconoscere le soluzioni solo *per confronto* fra assegnazioni. Si possono trattare i vincoli rigidi come quelli flessibili ma assegnando un costo *infinito* in caso di violazione: se il costo è finito non vi sarà stata nessuna loro violazione. In alternativa si può associare alla violazione di vincolo rigido un costo *elevato*, maggiore della somma di tutti i costi di vincoli flessibili. L'algoritmo di ottimizzazione dovrà trovare una soluzione con il minor numero di vincoli rigidi violati e, tra questi, quelli di costo minimo.

10.1 Metodi Sistematici per il Caso Discreto

Si parla di **ottimizzazione discreta** in caso di problema che coinvolge solo variabili *discrete*.

Possono essere adattati i metodi di ricerca di percorsi ottimali su grafo visti in precedenza. In questo caso il grafo avrà nodi contenenti assegnazioni a un insieme di variabili. I vicini di un nodo n si trovano scegliendo una variabile *var* non presente in n e le relative

assegnazioni di valori, consistenti con gli eventuali vincoli rigidi. Il costo di un arco è dato dalla somma dei costi dei vincoli valutati dopo l'assegnazione a *var*.

Il nodo di partenza conterrà l'assegnazione vuota mentre un nodo-goal conterrà un'assegnazione totale (dovrà essere consistente con tutti i vincoli).

I costi sono assegnati solo quando un vincolo soft può essere valutato. Per usare gli algoritmi A^* , **BRANCH-AND-BOUND** i costi-arco devono essere non-negativi. È possibile usare anche algoritmi per CSP come **GAC**, **VARIABLE ELIMINATION** si veda §4.8.1 in [PM23].

10.2 Ricerca Locale per l'Ottimizzazione

L'uso della *ricerca locale* nel caso problemi di ottimizzazione mira a minimizzare la funzione-obiettivo.

Gli algoritmi di ricerca locale (anche con ripartenze casuali) aggiornano via via e infine restituiscono la *migliore assegnazione trovata*. La terminazione può essere garantita fissando un numero massimo di iterate (o un tempo massimo).

È difficile stabilire se un'assegnazione totale trovata sia ottimale: si ha un **ottimo locale** trovando un'assegnazione non peggiore di tutte le *successive* possibili; un **ottimo globale** è dato da un'assegnazione non peggiore di *tutte* le assegnazioni. Senza la ricerca sistematica non si può sapere se la migliore assegnazione trovata localmente costituisca un ottimo globale o se ne esista altrove una migliore.

Per arrivare a una soluzione si può consentire la violazione di vincoli rigidi, e.g. adottando costi di violazione alti ma *finiti*.

10.3 Domini Continui: Discesa di Gradiente

In questo caso la ricerca locale è più complicata: bisogna stabilire come definire il *successore* di un'assegnazione.

L'idea dell'algoritmo di **discesa del gradiente**, **GRADIENT DESCENT** è quella di minimizzare la funzione di valutazione (purché *continua* e *differenziabile*). Come in un percorso in discesa, si fanno passi nelle direzioni più ripide. Il *successore* di un'assegnazione viene calcolato facendo un passo in discesa (segno negativo) di lunghezza proporzionale alla *pendenza* della funzione, ossia alla sua *derivata*. Esiste anche una ricerca di massimi di funzioni obiettivo tramite **risalita del gradiente**, **GRADIENT ASCENT** facendo passi di segno positivo.

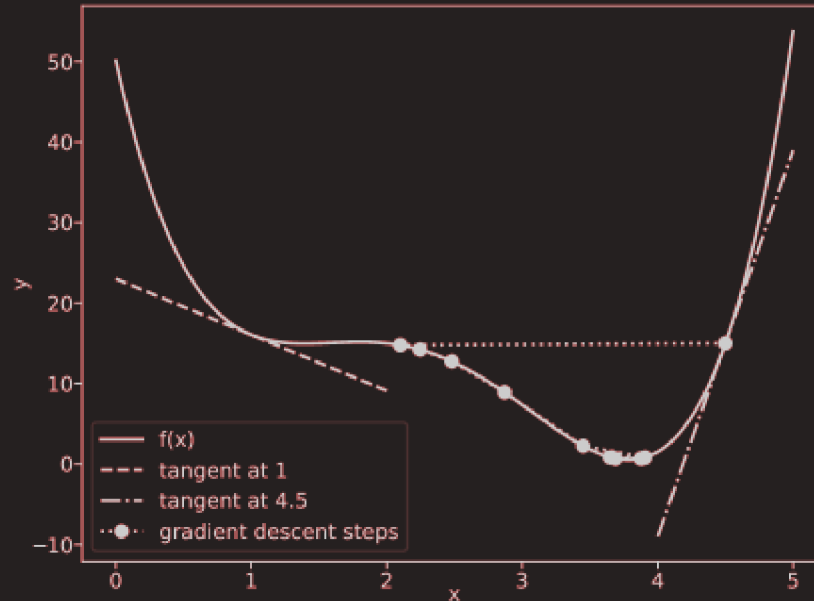
Nel caso *monodimensionale*, se a X è assegnato $v \in \mathbb{R}$ il valore successivo sarà:

$$v - \eta \cdot \frac{df}{dX}(v)$$

dove η è la misura del *passo* che determina la *rapidità* della discesa (se troppo grande, può superare il minimo, se troppo piccolo, il progresso è più lento); la derivata viene valutata in v :

$$\lim_{\epsilon \rightarrow 0} \frac{f(X = v + \epsilon) - f(X = v)}{\epsilon}$$

Esempio — Ricerca del minimo (locale) di $f(X)$:



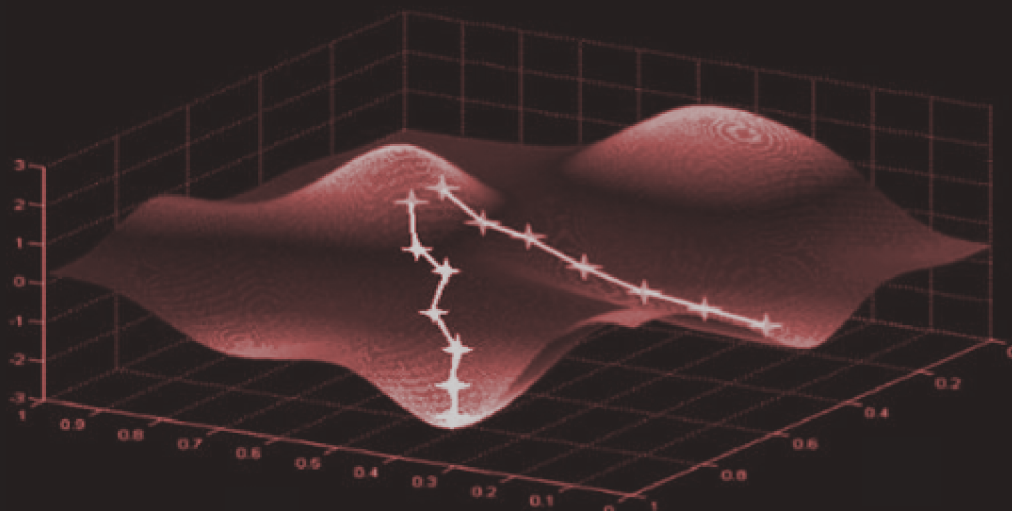
Nel caso *multidimensionale*, si fanno passi in tutte le direzioni, proporzionali a ogni *derivata parziale*. Date le variabili $\langle X_1, \dots, X_n \rangle$ e un'assegnazione iniziale di valori $\langle v_1, \dots, v_n \rangle$, l'assegnazione *successiva* si ottiene muovendosi lungo ogni direzione, in proporzione alla relativa pendenza di h (funzione di valutazione da minimizzare). Il nuovo valore per X_i si ottiene calcolando:

$$v_i \leftarrow v_i - \eta \cdot \frac{\partial h}{\partial X_i}(\vec{v}) \quad \forall i = 1, \dots, n$$

dove $\frac{\partial h}{\partial X_i}$ è la *derivata parziale* rispetto a X_i (le altre vengono trattate come costanti):

$$\frac{\partial h}{\partial X_i}(\vec{v}) = \lim_{\epsilon \rightarrow 0} \frac{h(v_1, \dots, v_i + \epsilon, \dots, v_n) - h(v_1, \dots, v_i, \dots, v_n)}{\epsilon}$$

Esempio — Discesa di gradiente: caso multidimensionale



L'*utilità* dell'algoritmo sarà evidente nell'apprendimento automatico del valore dei parametri di un modello, anche nel caso di modelli complessi (LLM) con 10^{12} parametri da

ottimizzare.

Vi sono numerose **varianti**, ad esempio per η si può usare una forma di ricerca binaria per cercare un valore ottimale, oppure potrebbe variare (decrescere) in funzione delle diverse iterate.

Per quanto riguarda la *convergenza*, in caso di funzioni regolari (*smooth*) con un minimo, **GD** converge a un minimo locale se η è sufficientemente piccolo. Quando è troppo grande allora è possibile che diverga, se, invece è troppo piccolo esso converge molto lentamente. Se il minimo locale è *unico* allora sarà anche un minimo *globale*. In caso di più minimi locali è necessaria una ulteriore *ricerca* per trovare il minimo globale. Ad esempio si può usare **RANDOM RESTART** o **RANDOM WALK**. È garantito il minimo globale solo quando sarà stato attraversato l'intero spazio di ricerca.

Riferimenti Bibliografici

- [PM23] D. Poole, A. Mackworth: *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press. 3a ed. 2023 (Ch.3)
- [PMG86] D. Poole, A. Mackworth, R. Goebel: *Computational Intelligence: A Logical Approach*. Oxford University Press. 1986
- [RN20] S.J. Russell, P. Norvig: *Artificial Intelligence*. Pearson. 4th Ed. 2020 (Ch.6,4)

Link

- [AIPython] sezione *Reasoning with Constraints* <https://artint.info/AIPython/>
- [Gradiente] wikipedia di funzioni vettoriali
- [CSP] wikipedia (in inglese)
- [CP] Programmazione a Vincoli wikipedia
- [GradientDescent] Discesa del Gradiente (steepest descent) wikipedia
- [LP] Linear Programming / Programmazione Lineare wikipedia (in inglese)
- [Optimization] Portale (in inglese)


Dispense ad esclusivo uso interno al corso.
formatted by [Markdeep 1.17](#) 

Figure tratte dal libro di testo [PM23], salvo diversa indicazione.