La classe **P**

Indice

In questa dispensa studiamo la classe **P**. Inizialmente, cerchiamo di capire perché la classe **P** è considerata la classe dei problemi trattabili computazionalmente. Successivamente, dopo avere brevemente descritto la (semplice) struttura di questa classe, presenteremo qualche esempio di dimostrazione di appartenenza di problemi alla classe **P**. Ricordando quanto dimostrato nella Didpensa 6 circa la correlazione polinomiale fra macchine di Turing e programmi scritti nel linguaggio **PascalMinimo**, potremo riferirci a programmi scritti in tale linguaggio piuttosto che a macchine di Turing per dimostrare l'appartenenza di un linguaggio (o, equivalentemente, di un problema decisionale) alla classe **P**.
Nel Paragrafo 8.3 torniamo ad occuparci di funzioni booleane in forma congiuntiva normale, mostrando che, se una funzione booleana f è in forma congiuntiva normale e le sue clausole contengono due letterali ciascuna, allora decidere se esiste una assegnazione di verità che soddisfa f richiede tempo polinomiale in $|f|$. Nel Paragrafo 8.4 definiamo il concetto di *colorabilità* di un grafo e dimostriamo che decidere se un grafo può essere colorato con 2 colori è un problema in **P**. Infine, nel Paragrafo 8.5 mostriamo che trasformare una qualsiasi funzione booleana in forma congiuntiva normale richiede tempo polinomiale nelle sue dimensioni: si tratta di un esempio di algoritmo polinomiale che corrisponde ad una macchina di Turing di tipo trasduttore, ossia, di un algoritmo che calcola una funzione in **FP**.
In tutti gli algoritmi che descriveremo in questa dispensa, per maggiore chiarezza, nella linea successiva a quella in cui viene specificato l'input, indicheremo anche l'output calcolato dall'algoritmo.

8.1 Classe P e trattabilità

La classe **P** è, genericamente, considerata la classe dei problemi *trattabili*, ossia, dei problemi di cui si riesce a calcolare una soluzione in tempi *ragionevoli*. Cerchiamo, ora, di capire il significato del termine "ragionevole" e la portata della precedente affermazione.

Sia Γ un problema decisionale, e siano T_1 e T_2 due macchine di Turing deterministiche che lo decidono tali che, per ogni $x \in I_\Gamma$, $dtime(T_1,x) \leq 2^{|x|}$ e $dtime(T_2,x) \leq |x|^2$.
Ragionando grossolanamente, supponiamo di disporre di un calcolatore in grado di eseguire $1000000000 = 10^9$ istruzioni di una macchina di Turing al secondo ed eseguiamo su tale calcolatore la computazione $T_2(\hat{x})$, per qualche $\hat{x} \in I_\Gamma$ tale che $|x| = 10^6$: allora, il calcolatore termina la computazione in $10^{12}/10^9 = 1000$ secondi, ossia, in poco più di un quarto d'ora. Se, invece, utilizziamo lo stesso calcolatore per eseguire la computazione $T_1(x)$, allora dovremo attendere

$$\frac{2^{10^6}}{10^9} = \frac{2^{10^6}}{2^{9\log_2 10}} > \frac{2^{10^6}}{2^{90}} = 2^{10^6-90} > 2^{10^4}$$

secondi, ossia, più di 2^{10^3} anni: una quantità di tempo senza dubbio irragionevole!
Dunque, intuitivamente, con la locuzione "in tempi ragionevoli" intendiamo riferirci ad un lasso di tempo che permetta all'interessato di utilizzare la risposta.

Osserviamo, ora, che, anche eseguendo sul calcolatore di cui all'esempio precedente la computazione $T_3(\hat{x})$, dove T_3 decide Γ e $dtime(T_3,x) \leq |x|^3$, il tempo di attesa della risposta (dell'ordine di 10^9 secondi) è ancora estremamente minore di quello richiesto dalla computazione $T_1(\hat{x})$. In effetti, se indichiamo con T_k una macchina che decide Γ tale che $dtime(T_k,x) \leq |x|^k$. Perché il tempo di attesa della risposta della computazione $T_k(\hat{x}$ sia paragonabile al tempo di attesa della risposta della computazione $T_1(\hat{x}$, l'esponente k deve essere tanto, tanto grande. E, anche una volta individuato un valore $\hat{k}$ tale che $|\hat{x}|^{\hat{k}} \approx 2^{|\hat{x}|}$, per tutte le parole $y \in I_\Gamma$ tali che $|y| \geq 10|\hat{x}|$, si avrà che $|y|^{\hat{k}}$ è infinitamente più piccolo di $2^{|y|}$.
Generalizzando (e ricordando la teoria dei limiti dall'analisi matematica), per ogni intero $k \in \mathbb{N}$ e per ogni funzione super-polinomiale $f : \mathbb{N} \to \mathbb{N}$, esiste un n_0 tale che, per ogni $n \geq n_0$, n^k è infinitamente più piccolo di $f(n)$.

Per quanto osservato sino ad ora, un problema è considerato trattabile quando esiste una macchina di Turing deterministica (o, come vedremo a breve, un algoritmo) che lo decide in tempo polinomiale. Ossia, **P** è la classe dei problemi trattabili.

8.2 La struttura della classe P

La classe **P** ha una struttura molto semplice. infatti, già sappiamo che co**P** = **P** (Corollario 6.2 della Dispensa 6). Inoltre, i linguaggi (o, equivalentemente, i problemi decisionali) in **P** sono indistinguibili se confrontati per mezzo della riducibilità polinomiale, come dimostrato nel seguente teorema.

Teorema 8.1: *Sia $L \subseteq \Sigma^*$ un linguaggio tale che $L \neq \emptyset$ e $L \neq \Sigma^*$. Se $L \in \mathbf{P}$, allora L è **P**-completo rispetto alla riducibilità polinomiale.*

Dimostrazione: Sia $L \subseteq \Sigma^*$ un linguaggio in **P** tale che $L \neq \emptyset$ e $L \neq \Sigma^*$. Allora esistono due parole $y, z \in \Sigma^*$ tali che $y \in L$ e $z \notin L$; osserviamo che, poiché $L \in \mathbf{P}$ e, qiondi, in particolare, è decidibile, le due parole y e z possono essere effettivamente calcolate. Infatti, sia T la macchina di Turing deterministica che decide (in tempo polinomiale) L: utilizzando T, definiamo il seguente trasduttore $\overline{T}$, dotato di due nastri di output che calcola y e z:

1) $h \rightarrow 0$;

2) fino a quando non ha scritto una parola sul secondo nastro e una parola sul terzo nastro:

   2.a) genera (deterministicamente) tutte le stringhe binarie di lunghezza h scrivendole sul primo nastro (separate da $\square$);

   2.b) per ogni x scritto sul primo nastro, simula la computazione $T(x)$: se $T(x)$ accetta scrive x sul secondo nastro, se $T(x)$ rigetta scrive x sul terzo nastro.

Poiché L è decidibile, ciascuna simulazione delle computazioni di T al passo 2.a) termina; inoltre, poiché $L \neq \emptyset$ e $L \neq \Sigma^*$, il loop al passo 2) viene eseguito un numero finito di volte, Dunque, in tempo finito, $\overline{T}$ calcola y e z.

Sia $L_1 \in \mathbf{P}$, con $L \subseteq \Sigma_1$; definiamo la seguente funzione $f : \Sigma_1^* \rightarrow \Sigma^*$: per ogni $x \in \Sigma_1^*$,

$$f(x) = \begin{cases} y & \text{se } x \in L_1; \\ z & \text{se } x \notin L_1. \end{cases}$$

Per costruzione, $x \in L_1$ se e soltanto se $f(x) \in L$. Inoltre, poiché $L_1 \in \mathbf{P}$ e poiché y e z sono due parole note (calcolate da $\overline{T}$, per così dire, prima ancora di aver definito f), f è calcolabile in tempo polinomiale. Questo prova che f è una ridazione da L_1 a L.
Poiché L_1 è un qualsiasi linguaggio in **P**, questo prova che L è **P**-completo rispetto alla riduzione polinomiale. $\square$

Per completezza, osserviamo che, comunque, al fine di caratterizzare i problemi più complessi all'interno della classe **P**, è stata introdotta una definizione di riducibilità più restrittiva della riducibilità polinomiale, la **LOGSPACE**-riducibilità: una riduzione f da un linguaggio L_1 ad un linguaggio L_2 è una **LOGSPACE**-riduzione se può essere calcolata in spazio logaritmico nella dimensione dell'input. Poiché, in virtù del Teorema 6.7 della Dispensa 6,

$$DSPACE(\log n) \subseteq DTIME(2^{\mathbf{O}(1)\log n}) = DTIME(n^{\mathbf{O}(1)}),$$

la **LOGSPACE**-riducibilità è una particolare riducibilità polinomiale. Tipicamente, con il temine **P**-completo ci si riferisce alla completezza rispetto alla **LOGSPACE**-riducibilità: per questa ragione, nel Teorema 8.1 abbiamo scritto esplicitamente che ci si riferiva alla riducibiltà polinomiale.
Lo studio dei problemi **P**-completi esula, comunque, dagli obiettivi di questo corso.

8.3 Il problema 2-SODDISFACIBILITÀ

Sia $X = \{x_1, x_2, \ldots, x_n\}$ un insieme di variabili booleane. Ricordiamo che una funzione booleana (o predicato) f nelle variabili X è in forma congiuntiva normale se $f(X)$ è una congiunzione di clausole, cioè, $f(X) = c_1 \wedge c_2 \wedge \ldots \wedge c_m$ ed ogni *clausola* c_j è una disgiunzione di letterali in X, ovvero, ad esempio, $c_j = x_1 \vee \neg x_n \vee x_2$. Una funzione booleana in forma congiuntiva normale in cui ogni clausola è costituita da esattamente due letterali ciascuna è detta essere in forma *2-congiuntiva normale*.

Nel problema decisionale 2-SODDISFACIBILITÀ (in breve, 2-SAT) ci si chiede, data una funzione booleana $f(X)$ in forma 2-congiuntiva normale, se f è soddisfacibile, ovvero, se esiste una assegnazione di verità $a : X \rightarrow \{vero, falso\}^n$ tale che $f(a(X)) = \texttt{vero}$.

Formalmente, la terna $\langle I_{2SAT}, S_{2SAT}, \pi_{2SAT} \rangle$ che definisce il problema 2-SODDISFACIBILITÀ è la seguente:

- $I_{2SAT} = \{\langle f, X \rangle : f$ è una funzione booleana nelle variabili in X in forma 2-congiuntiva normale $\}$;

- $S_{2SAT}(f,X) = \{a : X \to \{vero, falso\}\}$;
- $\pi_{2SAT}(f,XS_{2SAT}(f,X)) = \exists a \in S_{2SAT}(f,X) : f(a(X))$.

Dimostriamo ora che 2-SAT appartiene alla classe **P** presentando un algoritmo polinomiale che, data $f \in I_{2\mathrm{SAT}}$, decide se $f \in$ 2-SAT. Prima di affrontare questo compito, abbiamo bisogno di premettere alcune notazioni.
Intanto, poiché per soddisfare una funzione booleana f in forma 2-congiuntiva normale è necessario soddisfare *tutte* le clausole che la compongono, in quanto segue, per semplicità, rappresenteremo f nella forma di *insieme di clausole* piuttosto che nella forma di congiunzione di clausole.
Poi, indichiamo con il termine *letterale* una variabile in X oppure la sua negazione: $l_h = x_h$ oppure $l_h = \neg x_h$. In questo modo, possiamo sempre esprimere una clausola in f come la disgiunzione di due letterali, ossia, $c_j = l_{j_1} \vee l_{j_2}$. Osserviamo che, se $c_p = x_1 \vee x_2 = l_{p_1} \vee l_{p_2}$ e $c_q = \neg x_1 \vee x_3 = l_{q_1} \vee l_{q_2}$, allora $l_{p_1} \neq l_{q_1}$. Infine, se il letterale l corrisponde alla variabile booleana x, allora $\neg l$ corrisponde a $\neg x$; viceversa, se il letterale l corrisponde alla variabile booleana $\neg x$, allora $\neg l$ corrisponde a x.
Infine, diciamo che *un letterale l_p è vincolato da x_i* (o da $\neg x_i$) se esiste in f la clausola $x_i \vee l_p$ (rispettivamente, la clausola $\neg x_i \vee l_p$).

L'algoritmo che decide 2-SAT è basato sulla seguente, fondamentale, osservazione: ogni volta che assegniamo un valore di verità ad una variabile $x_i \in X$, partizioniamo l'insieme delle clausole nei tre sottoinsiemi descritti di seguito.

1. Se abbiamo assegnato ad x_i il valore `vero`, allora ogni clausola di f che contiene il letterale x_i è soddisfatta *indipendentemente dalla assegnazione di verità alla rimanente variabile che la compone*; chiamiamo queste clausole *soddisfatte da x_i* . Allo stesso modo, se abbiamo assegnato ad x_i il valore `falso`, allora ogni clausola di f che contiene il letterale $\neg x_i$ è soddisfatta indipendentemente dalla assegnazione di verità alla rimanente variabile; sono queste le clausole *soddisfatte da $\neg x_i$*.

2. Se abbiamo assegnato ad x_i il valore `vero`, allora ogni clausola di f che contiene il letterale $\neg x_i$ è soddisfatta se e soltanto se l'altro letterale che compare in essa assume il valore `vero`; sono queste le clausole *direttamente vincolate da x_i*. A loro volta, ogni clausola di f che contiene la negazione di un letterale che compare in una clausola direttamente vincolata da x_i (che chiameremo clausola *vincolata di livello 1* da x_i) è soddisfatta se e soltanto se l'altro letterale assume valore `vero`. In generale, ogni clausola di f che contiene la negazione di un letterale che compare in una clausola vincolata di livello k da x_i (che chiameremo clausola *vincolate di livello $k+1$* da x_i) è soddisfatta se e soltanto se l'altro letterale assume valore `vero`. Dunque, assegnare ad x_i il valore `vero` implica, ai fini della soddisfacibilità della funzione f, una assegnazione di verità ai letterali di *tutte e sole* le clausole che sono (direttamente o di qualsiasi livello) vincolate da x_i.

   Allo stesso modo, assegnare ad x_i il valore `falso` implica, ai fini della soddisfacibilità della funzione f, una assegnazione di verità ai letterali di *tutte e sole* le clausole che sono (direttamente o di qualsiasi livello) vincolate da $\neg x_i$.

3. Se abbiamo assegnato ad x_i il valore `vero` (o il valore `falso`), allora tutte le clausole di f che non sono vincolate dalla variabile x_i (o dalla sua negazione) sono *indipendenti* da tale assegnazione.

Possiamo ora descrivere formalmente l'algoritmo che decide $f \in$ 2-SAT, diretta conseguenza delle osservazioni sopra riportate.

L'algoritmo `A:2SAT` descritto in Tabella 8.1, codificato in linguaggio **PascalMinimo** (senza specificare le strutture dati che rappresentano le collezioni), è una successione di (al più $2n$) tentativi di assegnazioni di valori di verità alle variabili che compaiono in f, implementata mediante il ciclo **while** nelle linee 4-30: durante l'iterazione i-esima si tenta di assegnare un valore di verità (`vero` o `falso`) alla variabile x_i, se ad essa non è già stato assegnato alcun valore di verità durante le iterazioni precedenti. L'algoritmo utilizza tre variabili di tipo insieme, X_A, L_A e L_V, che durante la i-esima iterazione del ciclo **while** hanno il seguente significato: X_A è l'insieme delle variabili cui è stato definitivamente assegnato il valore `vero` o il valore `falso` durante qualche iterazione j con $j < i$ (quindi, inizialmente, $X_A = \emptyset$), L_A è l'insieme dei letterali cui si sta tentando di assegnare un valore di verità durante l'iterazione i, e L_V è l'insieme dei letterali vincolati da qualche letterale in L_A.
Come abbiamo premesso, durante l'i-esima iterazione del ciclo **while** alle linee 4-30, se alla variabile x_i era stato assegnato in precedenza un valore di verità (ossia, $x_i \notin X_A$, linea 5 del codice), si verifica se ad x_i si può assegnare il

Input: $f = \{c_1, c_2, \ldots, c_m\}$, con $c_j = (c_{j_1} \vee c_{j_2})$ e
$c_{j_i} \in \{x_1, \ldots, x_n\}$ per ogni $i = 1,2$ e $j = 1, \ldots, m$.
Output: accetta o rigetta.

```
1   X_A ← ∅;
2   contradd ← falso;
3   i ← 1;
4   while (i ≤ n ∧ contradd = falso) do begin
5       if (x_i ∉ X_A) then begin
6           L_A ← {x_i};                                   L_A contiene i letterali cui stiamo
                                                           tentando di assegnare un valore
7           L_V ← {x_i};                                   L_V contiene i letterali
                                                           che vincolano altri letterali
8           g ← f − {c_j = x_i ∨ l per qualche letterale l};   eliminiamo da f le clausole
                                                           soddisfatte da x_i = vero
9           while (L_V ≠ ∅ ∧ contradd = falso) do begin
10              estrai l_h da L_V;
11              for (c ∈ g) do begin
12                  if ([c = ¬l_h ∨ l_s] ∧ [¬l_s ∉ L_A]) then L_A ← L_A ∪ {l_s};
13                  else contradd ← vero;
14                  L_V ← L_V ∪ {l_s};
15                  g ← g − {c = l_s ∨ l per qualche letterale l};   eliminiamo da g le clausole
                                                           soddisfatte da l_s = vero;
16              end
17              if (contradd = vero e x_i ∈ L_A) then begin
18                  contradd ← falso;
19                  L_A ← {¬x_i};
20                  L_V ← {¬x_i};
21                  g ← f − {c_j = ¬x_i ∨ l per qualche letterale l};
22              end
23          end
24          if (contradd = falso) then begin
25              X_A ← X_A ∪ {x ∈ X : ∃l ∈ L_A[l = x ∨ l = ¬x]};
26              f ← g;
27          end
28      end
29      i ← i + 1;
30  end
31  if (contradd = vero) then Output: rigetta;
32  else Output: accetta.
```

Tabella 8.1: Algoritmo `A:2SAT`.

valore `vero` oppure il valore `falso`. Descriviamo in quanto segue l'iterazione *i*-esima del ciclo **while** nel caso in cui $x_i \notin X_A$ in maggiore dettaglio.

a) Proviamo ad assegnare ad x_i il valore `vero`: questo avviene inizializzando gli insiemi L_A (linea 6) e L_V (linea 7) a contenere il solo elemento x_i. Poi, alla linea 8, rimuoviamo dalla funzione le clausole soddisfatte dall'assegnazione $x_i =$ `vero`. Successivamente, ci occupiamo di verificare (linee 10-16) se l'assegnazione $x_i =$ `vero` induce nella catena di clausole ad essa vincolate (a qualsiasi livello) una contraddizione oppure una assegnazione che le soddisfa tutte. La contraddizione viene individuata quando, alle linee 12-13, si tenta di inserire nell'insieme L_A un letterale quando L_A ne contiene già la negazione. Se si arriva ad avere $L_V = \emptyset$ senza essere mai incorsi in una cotraddizione, in virtù di quanto osservato in precedenza, possiamo assegnare definitivamente $x_i =$ `vero`, eseguire le istruzioni di chiusura della iterazione *i*-esima (linee 24-27, che descriveremo al successivo punto c)) e passare a considerare la variabile x_{i+1} (ossia, possiamo iniziare la $(i+1)$-esima iterazione del ciclo **while** alle linee 4-30).

b) Se, invece, l'assegnazione $x_i =$ `vero` induce una contraddizione nelle clausole ad essa vincolate (ad esempio, ciò avviene con le due clausole $(\neg x_i \vee x_1)$ e $(\neg x_i \vee \neg x_1)$ oppure con le tre clausole $(\neg x_i \vee x_2)$, $(\neg x_i \vee x_3)$ e $(\neg x_2 \vee \neg x_3)$), allora proviamo ad assegnare ad x_i il valore `falso` inizializzando gli insiemi L_A e L_V a contenere il solo elemento $\neg x_i$ (linee 19 e 20). Successivamente, eseguiamo esattamente le stesse operazioni che avevamo eseguito con il tentativo $x_i =$ `vero`: rimuoviamo dalla funzione le clausole soddisfatte dall'assegnazione $x_i =$ `falso` (linea 21) e verifichiamo (linee 10-16) se l'assegnazione $x_i =$ `falso` induce nella catena di clausole ad essa vincolate (a qualsiasi livello) una contraddizione oppure una assegnazione che le soddisfa tutte. Osserviamo che la condizione dell'istruzione **if** alla linea 17 contiene il vincolo $x_i \in L_A$: questo perché tentiamo l'assegnazione $x_i =$ `falso` una sola volta, ossia, solo al termine del tentativo di assegnazione $x_i =$ `vero` nel caso in cui essa abbia indotto una contraddizione. Analogamente al punto a) precedente, se si arriva ad avere $L_V = \emptyset$ senza essere mai incorsi in una cotraddizione, possiamo assegnare definitivamente $x_i =$ `falso`, eseguire le istruzioni di chiusura della iterazione *i*-esima (linee 24-27, che descriveremo al punto c)) e passare a considerare la variabile x_{i+1} (ossia, possiamo iniziare la $(i+1)$-esima iterazione del ciclo **while** alle linee 4-30).

c) Se le clausole vincolate dall'assegnazione $x_i =$ `vero` oppure le clausole vincolate dall'assegnazione $x_i =$ `falso` non inducono una contraddizione, allora procediamo a marcare come assegnate le variabili vincolate, rispettivamente, a x_i o a $\neg x_i$ (linea 25) ed eliminiamo dalla funzione f le clausole così soddisfatte (linea 26); procediamo poi con l'iterazione successiva.

Analizziamo ora il costo di tale algoritmo. Il corpo del ciclo **while** delle linee 4-30 viene ripetuto, nel caso peggiore (in cui, ad ogni iterazione viene assegnato il valore ad esattamente una variabile in X) n volte: ad ogni iterazione, il ciclo **while** interno (linee 9-23) viene ripetuto, al più $2n$ volte: infatti, la scansione delle clausole eseguita dal ciclo **for** (linee11-16) viene ripetuta al più due volte (quando, al termine della prima, la variabile contradd ha assunto il valore `vero` e viene eseguita l'istruzione **if** alle linee 17-22) e, durante ciascuna scansione, la dimensione massima che L_V può assumere è n. Ogni iterazione del ciclo **for** costa, nel caso peggiore, $O(n \cdot m)$ poiché, ad ogni iterazione, è necessario eseguire il controllo all'istruzione 12 (e l'insieme L_A contiene al più n letterali), e l'istruzione alla linea 15 (e g contiene al più m clausole, ciascuna di dimensione costante). In definitiva, il ciclo **for** costa $O(nm^2)$. Pertanto, il costo di una iterazione del ciclo **while** interno è $O(n^2m^2)$. Possiamo quindi concludere che il costo dell'algoritmo `A:2SAT` è in $O(n^3m^2)$ e questo dimostra che 2-SAT$\in$ **P**.

8.4 Il problema 2-COLORABILITÀ

Sia $G = (V,E)$ un grafo non orientato e k un intero positivo. Il grafo G si dice *k-colorabile* se esiste una funzione $c : V \to \{1,2,\ldots,k\}$ tale che, per ogni $u,v \in V$,

$$(u,v) \in E \to c(u) \neq c(v).$$

Possiamo, ora, definire il problema decisionale 2-COLORABILITÀ: dato un grafo non orientato G, decidere se G è 2-colorabile. Formalmente, la tripla $\langle I_{2COL}, S_{2COL}, \pi_{2COL}\rangle$ che definisce il problema 2-COLORABILITÀ è la seguente:

- $I_{2COL} = \{\langle G = (V,E)\rangle : G$ è un grafo non orientato $\}$;

- $S_{2COL}(G) = \{c : V \to \{1,2\}\}$;
- $\pi_{2COL}(G, S_{2COL}(G)) = \exists c \in S_{2COL}(G) : \forall u,v \in V\ [\ c(u) = c(v) \to (u,v) \notin E\ ]$ o, equivalentemente,

$$\pi_{2COL}(G, S_{2COL}(G)) = \exists c \in S_{2COL}(G) : \forall (u,v) \in E\ [\ c(u) \neq c(v)\ ].$$

Invece di descrivere un algoritmo che risolve direttamente il problema 2-COLORABILITÀ, mostriamo nel seguito una riduzione polinomiale da 2-COLORABILITÀ a 2-SODDISFACIBILITÀ; tale riduzione, insieme all'algoritmo che decide 2-SODDISFACIBILITÀ, induce un algoritmo che decide 2-COLORABILITÀ. Infatti, dato un grafo G istanza di 2-COLORABILITÀ, tale algoritmo opera come segue:

1) utilizzando la riduzione da 2-COLORABILITÀ a 2-SODDISFACIBILITÀ, trasforma G in una funzione booleana f istanza di 2-SODDISFACIBILITÀ;
2) esegui l'algoritmo `A:2SAT` in Tabella 8.1 con input f.

Le proprietà della riduzione implicano che, se l'esecuzione dell'algoritmo `A:2SAT` con input f termina accettando f allora G è 2-colorabile, altrimenti G non è 2-colorabile. Quindi, l'algoritmo descritto dai punti 1) e 2) sopra è corretto. Inoltre, esso opera in tempo polinomiale, infatti: il punto 1) richiede tempo polinomiale, perché esegue una riduzione polinomiale, e calcola una funzione f tale che $|f| = p(|G|)$ per qualche polinomio p; infine, il punto 2) richiede tempo in $O(|f|^3) = O(p(|G|)^3)$.

Descriviamo ora la riduzione da 2-COLORABILITÀ a 2-SODDISFACIBILITÀ. Sia $G = (V,E)$ un grafo non orientato; osserviamo che una funzione $c : V \to \{1,2\}$ è una 2-colorazione per G se e soltanto se

$$\forall (u,v) \in E : [(c(u) = 1) \wedge (c(v) = 2)] \vee [(c(u) = 2) \wedge (c(v) = 1)]\,.$$

Questo è equivalente a

$$\forall (u,v) \in E : [(c(u) = 1) \wedge \neg(c(v) = 1)] \vee [\neg(c(u) = 1) \wedge (c(v) = 1)]\,,$$

da cui, applicando le leggi di De Morgan,

$$\forall (u,v) \in E : \{(c(u) = 1) \vee [\neg(c(u) = 1) \wedge (c(v) = 1)]\} \wedge \{\neg(c(v) = 1) \vee [\neg(c(u) = 1) \wedge (c(v) = 1)]\},$$

$$\forall (u,v) \in E : [(c(u) = 1) \vee \neg(c(u) = 1)] \wedge [(c(u) = 1) \vee (c(v) = 1)] \wedge$$

$$\wedge [\neg(c(v) = 1) \vee \neg(c(u) = 1)] \wedge [\neg(c(v) = 1) \vee (c(v) = 1)],$$

$$\forall (u,v) \in E : \mathit{vero} \wedge [(c(u) = 1) \vee (c(v) = 1)] \wedge [\neg(c(v) = 1) \vee \neg(c(u) = 1)] \wedge \mathit{vero},$$

ossia, G è 2-colorabile se e soltanto se

$$\forall (u,v) \in E : [(c(u) = 1) \vee (c(v) = 1)] \wedge [\neg(c(v) = 1) \vee \neg(c(u) = 1)].$$

Possiamo, a questo punto, associare ad ogni nodo u di G la variabile booleana $x_u = (c(u) = 1)$ ed ottenere che G è 2-colorabile se e soltanto se

$$\forall (u,v) \in E : (x_u \vee x_v) \wedge (\neg x_u \vee \neg x_v)$$

è soddisfacibile.

Denotiamo ora con $E = \{e_1, \ldots, e_m\}$ l'insieme degli archi di G e con $e_i = (u_i, v_i)$ l'i-esimo arco. Allora, possiamo concludere che G è 2-colorabile se e soltanto se l'istanza di 2-SODDISFACIBILITÀ

$$f(G) = (x_{u_1} \vee x_{v_1}) \wedge (\neg x_{u_1} \vee \neg x_{v_1}) \wedge (x_{u_2} \vee x_{v_2}) \wedge (\neg x_{u_2} \vee \neg x_{v_2}) \wedge \ldots \wedge (x_{u_m} \vee x_{v_m}) \wedge (\neg x_{u_m} \vee \neg x_{v_m})$$

è soddisfacibile. Banalmente, il calcolo di $f(G)$ richiede tempo polinomiale: infatti, ad ogni arco in G corrisponde in $f(G)$ un insieme di un numero costante (pari a 2) di clausole ed il calcolo di ciascuna di esse richiede tempo polinomiale in $|G|$.

Input:	formula booleana f, contenente i soli operatori $\wedge$, $\vee$, $\neg$, e priva di quantificatori esistenziali e universali.
Output:	insieme C di clausole.
1	$C \leftarrow \emptyset$;
2	$Y \leftarrow \emptyset$;
3	`CNF-RIC`(f, C, Y);
4	**Output**: C.

Tabella 8.2: Algoritmo `A:toCNF`.

8.5 Trasformazione di una funzione booleana in forma congiuntiva normale

Ci proponiamo, in questo paragrafo, di dimostrare il teorema seguente.

Teorema 8.2: *Data una qualunque formula booleana f contenente i soli operatori $\wedge$, $\vee$ e $\neg$, è possibile derivare da essa in tempo polinomiale una formula booleana f' in forma congiuntiva normale tale che f' è soddisfacibile se e soltanto se f è soddisfacibile.*

Dimostrazione:. Osserviamo, innanzi tutto, che, poiché f contiene i soli operatori $\wedge$, $\vee$ e $\neg$, allora una delle seguenti eventualità deve necessariamente verificarsi:

1) esistono due funzioni booleane, f_1 ed f_2, contenenti i soli operatori $\wedge$, $\vee$ e $\neg$, tali che

$$f = f_1 \wedge f_2;$$

2) esiste una funzione booleana f_1 contenente i soli operatori $\wedge$, $\vee$ e $\neg$, tale che

$$f = \neg f_1;$$

3) esistono due funzioni booleane, f_1 ed f_2, contenenti i soli operatori $\wedge$, $\vee$ e $\neg$, tali che

$$f = f_1 \vee f_2.$$

Nel primo caso, per dimostrare il teorema, è sufficiente dimostrarlo separatamente per f_1 e per f_2. Nel secondo caso, è necessario applicare le leggi di De Morgan e ridursi ad uno degli altri due casi. Infine, nel terzo caso osserviamo che, se denotiamo con y una variabile booleana che non compare in f, allora f è equivalente alla formula $(f_1 \vee y) \wedge (\neg y \vee f_2)$. Pertanto, in questo terzo caso, f' viene derivata calcolando una formula f'_1 in forma CNF ed equivalente ad f_1 e una formula f'_2 in forma CNF ed equivalente ad f_2 e, successivamente, applicando la proprietà distributiva dell'operatore $\vee$ alle due sotto-formule $(f'_1 \vee y)$ e $(\neg y \vee f'_2)$. Osserviamo che questa ultima operazione si implementa semplicemente aggiungendo a ciascuna clausola di f'_1 il letterale y ed a ciascuna clausola di f'_2 il letterale $\neg y$.

Queste osservazioni conducono naturalmente alla definizione dell'algoritmo ricorsivo `A:toCNF` descritto in Tabella 8.2. dove `CNF-RIC` è la funzione ricorsiva, descritta in Tabella 8.3. Essa utilizza tre parametri, g, D e Z, che hanno semantica differente: il parametro g è un *parametro di input*, ossia, viene utilizzato nella funzione in sola lettura, mentre D e Z sono *parametri di output*, ossia, vengono modificati nel corso dell'esecuzione della funzione e le modifiche rimangono visibili al termine dell'esecuzione (in analogia con i parametri di tipo puntatore nel linguaggio C, o di tipo riferimento nel linguaggio Java). In particolare, il parametro D è l'insieme delle clausole costruite sulla funzione g ad un certo istante della computazione: la prima invocazione di `CNF-RIC` da parte dell'algoritmo `A:toCNF` avviene con tale parametro inizializzato all'insieme vuoto e, al termine delle invocazioni ricorsive, esso è l'insieme delle clausole corrispondente a g.

Calcoliamo ora la complessità dell'algoritmo `A:CNF`, che coincide con quella della funzione `CNF-RIC`.
Indichiamo con $T_{RIC}(f)$ il numero di passi eseguiti dalla funzione `CNF-RIC` con input f. Dalla definizione di `CNF-RIC` segue che

$$T_{RIC}(f) = \begin{cases} |f| + T_{RIC}(f_1) & \text{se } f = \neg f_1 \\ T_{RIC}(f_1) + T_{RIC}(f_2) & \text{se } f = f_1 \wedge f_2 \\ 4 + T_{RIC}(f_1) + |D_1| + T_{RIC}(f_2) + |D_2| & \text{se } f = f_1 \vee f_2 \\ |f| & \text{se } f \text{ è una disgiunzione di letterali.} \end{cases} \tag{8.1}$$

void	CNF−RIC(**in:**g; **out:** D, Z)	g è un predicato del primo ordine, D è un insieme di clausole, Z è un insieme di variabili booleane
1	**if** (g è una disgiunzione di letterali) **then** $D \leftarrow D \cup \{g\}$;	in tal caso, g può essere un singolo letterale
2	**else if** ($g = \neg g_1$) **then begin**	
3	$g_2 \leftarrow$ risultato dell'applicazione delle leggi di De Morgan a g_1;	
4	CNF−RIC(g_2, D, Z);	
5	**end**	
6	**else if** ($g = g_1 \wedge g_2$) **then begin**	
7	CNF−RIC(g_1, D, Z);	
8	CNF−RIC(g_2, D, Z);	
9	**end**	
10	**else if** ($g = g_1 \vee g_2$) **then begin**	
11	$p \leftarrow |Z| + 1$;	
12	$Z \leftarrow Z \cup \{y_p\}$;	
13	$D1 \leftarrow \emptyset$;	
14	CNF−RIC(g_1, $D1$, Z);	
15	**for** ($i \leftarrow 1$; $i \leq |D1|$; $i \leftarrow i+1$) **do**	
16	$D \leftarrow D \cup \{(D1[i] \vee y_p)\}$;	
17	$D2 \leftarrow \emptyset$;	
18	CNF−RIC(g_2, $D2$, Z);	
19	**for** ($i \leftarrow 1$; $i \leq |D2|$; $i \leftarrow i+1$) **do**	
20	$D \leftarrow D \cup \{D2[i] \vee \neg y_p\}$;	
21	**end**	

Tabella 8.3: Algoritmo CNF−RIC.

Osserviamo che, ogni volta che CNF−RIC viene invocata ricorsivamente, la funzione invocata prende come parametro una funzione booleana contenente almeno un letterale oppure un operatore in meno rispetto al parametro ricevuto in ingresso dalla funzione invocante, ossia, $|f_1| < |f|$ e $|f_2| v |f|$. Inoltre, $|f_1| + |f_2| \leq |f|$.

Dimostriamo, ora, che $T_{RIC}(f) \leq 5|f|^2$.
La prova è per induzione. Se $|f| = 1$ (ossia, f è costituita da un solo letterale non negato), allora nella funzione CNF−RIC viene eseguita la sola istruzione alla linea 1, che ha costo pari a $|f|$; dunque, $T_{RIC}(f) = 1 < 5|f|^2$. Supponiamo ora che l'asserto sia vero quando $|f| \leq k$, e proviamolo per $|f| = k+1$. La prova di tale passo induttivo avviene distinguendo i casi possibili in cui può presentarsi f.

1. Se f è una disgiunzione di letterali, allora viene eseguita la sola istruzione alla linea 1, che ha costo pari a $|f| = k+1 < 5(k+1)^2$.
2. Se $f = \neg f_1$, allora $T_{RIC}(f) = |f| + T_{RIC}(f_1)$ e $|f_1| = k$. Allora, dall'ipotesi induttiva segue che

$$T_{RIC}(f) = |f| + T_{RIC}(f_1) = k+1+T_{RIC}(f_1) \leq k+1+5k^2 < 5(k+1)^2.$$

3. Se $f = f_1 \wedge f_2$, allora $T_{RIC}(f) = |f| + T_{RIC}(f_1) + T_{RIC}(f_2)$, con $|f_1| + |f_2| = k$, $|f_1| = k_1 < k$ e $|f_2| = k_2 < k$. Allora, dall'ipotesi induttiva segue che

$$\begin{aligned} T_{RIC}(f) &= |f| + T_{RIC}(f_1) + T_{RIC}(f_2) = k+1+T_{RIC}(f_1) + T_{RIC}(f_2) \\ &\leq k+1+5k_1^2+5k_2^2 = k+1+5(k_1^2+k_2^2) \\ &\leq k+1+5(k_1+k_2)^2 = k+1+5k^2 \\ &< 5(k+1)^2. \end{aligned}$$

4. Se $f = f_1 \vee f_2$, allora $T_{RIC}(f) = 4 + |D_1| + |D_2| + T_{RIC}(f_1) + T_{RIC}(f_2)$ con $|D_1| + |D_2| \leq |f|$, $|f_1| + |f_2| = k$, $|f_1| = k_1 < k$ e $|f_2| = k_2 < k$. Allora, dall'ipotesi induttiva segue che

$$\begin{aligned} T_{RIC}(f) &= 4 + |f| + T_{RIC}(f_1) + T_{RIC}(f_2) = k + 5 + T_{RIC}(f_1) + T_{RIC}(f_2) \\ &\leq k + 5 + 5k_1^2 + 5k_2^2 = k + 5 + 5(k_1^2 + k_2^2) \\ &\leq k + 5 + 5(k_1 + k_2)^2 = k + 5 + 5k^2 \\ &< 5(k+1)^2. \end{aligned}$$

Questo conclude la prova del teorema. □