

Decidabilitate

Construcția algoritmilor ridică o întrebare fundamentală: problema poate fi rezolvată prin algoritmi eficienți sau asemenea algoritmi nu există? Răspunsul poate evita căutarea imposibilului și poate sugera alternative de rezolvare. În cele ce urmează sunt discutate, din perspectiva calculabilității, noțiuni elementare de teoria mulțimilor și se introduce conceptul esențial de decidabilitate. Se arată că există clase mari de probleme care nu pot fi rezolvate cu calculatorul.

D.1 Decidabilitate

- O mulțime A este echipotentă cu o mulțime B dacă există $f: A \rightarrow B$, bijectivă.
- O mulțime infinită A este numărabilă dacă A este echipotentă cu mulțimea numerelor naturale $\mathbb{N}$. O numerotare a mulțimii A este o funcție bijectivă $a: \mathbb{N} \rightarrow A$, astfel încât $a(n) =_{\text{def}} a_n$, iar $A = \{a_k \mid k \in \mathbb{N}\}$. Echivalent, o mulțime este numărabilă dacă există o numerotare a sa.
- Spunem că o mulțime infinită și numărabilă este ω -numărabilă, iar o mulțime infinită și nenumărabilă este ω -nenumărabilă.

Propoziția D.1 O submulțime infinită a unei mulțimi ω -numărabile este ω -numărabilă.

Fie A ω -numărabilă și $B \subseteq A$. Există numerotarea $A = \{a_k \mid k \in \mathbb{N}\}$. Construim numerotarea:

$$b_0 = a_j, \quad j = \min\{k \mid a_k \in B\}$$

$$b_i = a_j, \quad j = \min\{k \mid a_k \in B \setminus \bigcup_{q < i} \{b_q\}\}, \quad i \geq 1 \quad \blacksquare$$

Propoziția D.2 Fie A ω -nenumărabilă, iar B o mulțime ω -numărabilă, astfel încât $\forall x \in B \bullet x \in A$. Atunci $\exists x \in A \bullet x \notin B$ (altfel spus $A \setminus B \neq \emptyset$ sau $B \subset A$).

Presupunem că teorema nu este adevărată, deci $\neg \exists x \in A \bullet x \notin B$, adică $\forall x \in A \bullet x \in B$. Atunci $A = B$, iar A este ω -numărabilă. Imposibil. $\blacksquare$

Propoziția D.3 Mulțimea $\text{Hom}(\mathbb{N}, \mathbb{N})$ (a funcțiilor din $\mathbb{N}$ în $\mathbb{N}$) este ω -nenumărabilă.

Alegem $e_1 \neq e_2$ două elemente din $\mathbb{N}$. Presupunem că $\mathbb{N}$ este echipotentă cu $\text{Hom}(\mathbb{N}, \mathbb{N})$. Atunci, există $f: \mathbb{N} \rightarrow \text{Hom}(\mathbb{N}, \mathbb{N})$, bijectivă. Notăm f_e funcția $f(e)$ din $\text{Hom}(\mathbb{N}, \mathbb{N})$ și construim mulțimea (nevidă) $A = \{e \in \mathbb{N} \mid f_e(e) = e_1\}$.

Alegem din $\text{Hom}(N, N)$ funcția $g = f_x$ astfel încât: $g(e) = \begin{cases} e_2 & \text{dacă } e \in A \\ e_1 & \text{dacă } e \notin A \end{cases}$

Pentru $x \in A$: $g(x) = e_2$, conform definiției funcției g ;
 $g(x) = f_x(x) = e_1$, conform definiției mulțimii A . Imposibil.

Pentru $x \notin A$: $g(x) = e_1$, conform definiției funcției g ;
 $g(x) = f_x(x) \neq e_1$, conform definiției mulțimii A . Imposibil.

Rezultă că funcția g nu poate fi numerotată și deci funcția f nu este surjectivă. Înseamnă că N nu este echipotentă cu $\text{Hom}(N, N)$. ■

Definiția D.1 Numim $P_{i,j}$ mulțimea programelor având ca intrare un tuplu din $\prod_{i=1}^i N$ și ca ieșire un tuplu din $\prod_{j=1}^j N$. Considerăm că orice program $P \in P_{i,j}$ este executabil în sens Turing: există o mașină de calcul Turing care, pentru intrări valide, calculează în timp finit ieșirile programului. Convenim ca pentru un tuplu de intrare invalid programul să nu se termine. Scriem $P(x) = \perp$, dacă P nu se termină pentru tuplul x și $P(x) = r$, dacă P se termină cu rezultatul r pentru datele x .

În ceea ce privește mașina care execută programele, o să o imaginăm ca mașină virtuală, caracteristică unui limbaj de programare imperativ, de exemplu un limbaj asemănător cu C. Perspectiva pragmatică este corectă deoarece, teoretic, astfel de mașini de calcul pot fi reduse la mașini Turing.

Propoziția D.4 Mulțimea $P_{i,j}$ este numărabilă.

Programele din $P_{i,j}$ pot fi privite ca șiruri de lungime finită, șiruri formate cu simbolurile unui alfabet finit Σ . Notăm $\Sigma^* = \bigcup_j \Sigma_j$ mulțimea șirurilor cu simboluri din Σ , unde $\Sigma_0 = \{\lambda\}$, λ fiind șirul vid, iar $\Sigma_j = \{wx \mid w \in \Sigma_{j-1}, x \in \Sigma\}$, pentru $j \geq 1$. Mulțimea Σ^* este ω -numărabilă, pentru că putem construi numerotarea:

$$w_0 = \lambda$$

$$w_j = \min_{\text{lex}} \{w \in \Sigma^* \setminus \bigcup_{k < j} \{w_k\}\}, \quad j > 0,$$

unde $\min_{\text{lex}}(M)$ desemnează șirul cel mai mic în ordine lexicografică din M . Fie $\Sigma_{i,j} \subseteq \Sigma^*$ mulțimea șirurilor ce codifică programe din $P_{i,j}$. Conform propoziției (D.1) mulțimea $\Sigma_{i,j}$ este numărabilă. Conform echipotenței dintre $\Sigma_{i,j}$ și $P_{i,j}$, rezultă că $P_{i,j}$ este numărabilă. ■

Definiția D.2 O funcție $f: N \rightarrow N$ este calculabilă (în sens Turing) dacă există cel puțin un program $P \in P_{1,1}$ astfel încât:

$$P(x) = \begin{cases} f(x), & \text{pentru } x \in \text{dom}(f) \\ \perp, & \text{pentru } x \notin \text{dom}(f) \end{cases}$$

Teza Church-Turing. Orice funcție efectiv calculabilă este Turing-calculabilă. Teza stabilește o relație dintre două concepte diferite.

Primul concept se referă la calculabilitate efectivă și este de natură filozofică. Se acceptă că o funcție este efectiv calculabilă dacă există un algoritm care calculează valoarea funcției pentru orice parametru din domeniul de definiție al acesteia. Prin algoritm se înțelege aici un proces de calcul finit (care folosește reprezentări finite ale datelor și se termină după o perioadă finită de timp), bine definit (prelucrările sunt specificate neambiguu încât pot fi executate mecanic) și concludiv (care produce un rezultat neambiguu ce reprezintă valoarea funcției). Deoarece însăși natura procesului de calcul este o chestiune filozofică, fiind deasupra modului în care noi percepem diversele cazuri particulare de efectuare a unui asemenea proces, noțiunea de calculabilitate efectivă rămâne vagă.

Al doilea concept se referă la calculabilitate în sens Turing, deci la un proces de calcul care poate fi efectuat de către o mașină de calcul particulară, anume mașina Turing. Conceptul este exact, demonstrându-se că mulțimea funcțiilor parțial-recursive este chiar mulțimea funcțiilor Turing-calculabile.

Teza Church-Turing susține, fără a demonstra, că orice proces de calcul efectiv poate fi redus la unul specific unei mașini Turing. Cu alte cuvinte, teza susține că Turing-calculabilitatea este universală și, implicit, că funcțiile efectiv calculabile sunt cele recursive. Deși teza nu a fost demonstrată, există suport teoretic consistent care pledează în favoarea ei. Bunăoară, s-au inventat mașini abstracte de calcul aparent diferite din perspectiva modului în care este imaginat procesul de calcul, demonstrându-se echivalența lor cu mașina Turing.

Conform tezei Church-Turing admitem că orice funcție efectiv calculabilă este recursivă. Astfel, putem defini intuitiv o funcție recursivă ca fiind programabilă și, reciproc, putem susține că orice program calculează o funcție recursivă.

Definiția D.3 O funcție $f: \mathbb{N} \rightarrow \mathbb{N}$ este recursivă dacă este efectiv calculabilă. Cu alte cuvinte, există cel puțin un program $P \in \mathcal{P}_{1,1}$ care calculează f . Notăm cu $\mathcal{F}_{\mathcal{R}}$ mulțimea funcțiilor recursive din $\text{Hom}(\mathbb{N}, \mathbb{N})$.

Propoziția D.5 Mulțimea $\mathcal{F}_{\mathcal{R}}$ este ∞ -numărabilă.

Construim numerotarea:

- f_0 = funcția calculată de programul cu indicele p_0 ;
- f_j , $j > 0$, este funcția calculată de programul cu indicele cel mai mic din mulțimea $\mathcal{P}_{1,1} \setminus \{P \in \mathcal{P}_{1,1} \mid P \text{ calculează } f_k, k < j\}$ ■

Propoziția D.6 Există în $\text{Hom}(\mathbb{N}, \mathbb{N})$ funcții care nu sunt recursive, deci nu sunt programabile (nu sunt calculabile, indiferent de mașina de calcul folosită).

Demonstrația rezultă direct din propozițiile (D.2), (D.3) și (D.5). Ca exemplu de funcție necalculabilă considerăm funcția totală

$$f: \mathbb{N} \rightarrow \mathbb{N}, f(n) = \begin{cases} 1 + p_n(n), & \text{dacă } p_n(n) \text{ se termină} \\ 0, & \text{dacă } p_n(n) = \perp \end{cases}$$

unde p_n este programul de indice n din mulțimea $P_{1,1}$. Să presupunem că $f \in F$. Atunci, conform definiției, există un program $p_m \in P_{1,1}$ care calculează f . Pentru că f este totală, p_m se termină pentru orice $x \in \mathbb{N}$, deci și pentru m . Conform definiției lui f , rezultă $p_m(m) = 1 + p_m(m)$. Imposibil. ■

Propoziția (D.6) subliniază, indirect, limita puterii de expresie a limbajelor de programare. Indiferent de limbaj, există probleme care nu pot fi rezolvate mecanic, folosind o mașină de calcul, una dintre aceste probleme fiind deciderea terminării programelor. Din acest punct de vedere, propoziția (D.6) merită o analiză mai nuanțată.

Clasa funcțiilor recursive este divizată în funcții primitiv-recursive și funcții parțial-recursive, clasa funcțiilor primitiv-recursive fiind inclusă în clasa funcțiilor parțial-recursive. Funcțiile primitiv-recursive sunt totale, astfel încât un program care calculează o asemenea funcție se termină pentru orice $n \in \mathbb{N}$. În schimb, programul care calculează o funcție parțial-recursive f se termină doar în punctele $n \in \text{dom}(f)$. Demonstrația propoziției (D.6) se bazează tocmai pe totalitatea funcției f , funcție care decide terminarea programelor din $P_{1,1}$. Conform demonstrației, f nu poate fi totală și deci nu poate fi primitiv-recursive. Dacă $\text{dom}(f)$ este restrâns la mulțimea indicilor n astfel încât $p_n(n)$ se termină, atunci f devine efectiv-calculabilă. Într-adevăr, programul $R(n) \{ \text{return } 1 + p_n(n); \}$ se termină dacă $p_n(n)$ se termină. Mai mult, dacă m este indicele programului R , atunci $m \notin \text{dom}(f)$ pentru a avea $R(m) = p_m(m) = \perp$. Funcția f este parțial-recursive și poate verifica doar terminarea programelor $p_n(n)$ care se termină. Dacă $\text{dom}(f)$ ar fi cunoscut apriori, funcția f ar fi banală. Cum domeniul $\text{dom}(f)$ nu este cunoscut apriori nu putem ști dacă $R(n)$ se termină și spunem că f semi-decide terminarea programelor din $P_{1,1}$. Prin urmare, propoziția (D.6) poate fi nuanțată în forma "nu există nici o funcție recursivă care decide terminarea programelor" sau în forma propoziției (D.6').

Propoziția D.6' Terminarea programelor este semi-decidabilă. Pentru demonstrație, vezi exemplul D.2 referitor la definiția D.6.

În urma analizei de mai sus, putem observa că o submulțime $A \subseteq \mathbb{N}$ poate fi cunoscută în două moduri aparte: fie prin intermediul unei funcții recursive totale sau, echivalent, folosind un program care decide dacă un element x aparține sau nu mulțimii A , fie prin intermediul unei funcții parțial-recursive cu domeniul A sau, echivalent, folosind un program care se termină doar dacă $x \in A$.

Definiția D.4 O submulțime $A \subseteq \mathbb{N}$ este recursivă dacă există o funcție recursivă totală $f: \mathbb{N} \rightarrow \mathbb{N}$, astfel încât, pentru orice $n \in \mathbb{N}$:

$$f(n) = \begin{cases} 1, & \text{dacă } n \in A \\ 0, & \text{dacă } n \notin A \end{cases}$$

Orice submulțime finită $A \subset \mathbb{N}$ este recursivă: A este numărabilă și putem construi lesne un program care decide apartenența unui element oarecare $n \in \mathbb{N}$ la A , program bazat pe operația de comparare a numerelor.

```
P(n) {
    for(i=0; i < card(A); i++) if(n = Ai) return 1;
    return 0;
}
```

Programul de mai sus nu poate fi generalizat pentru o mulțime finită cu orice tip de elemente. Bunăoară, dacă elementele mulțimii A sunt funcții, egalitatea funcțiilor nu poate fi decisă mecanic. Prin urmare, nu orice mulțime finită este recursivă.

Definiția D.5 O submulțime $A \subset \mathbb{N}$ este recursiv-numărabilă dacă (alternativ):

- a) Există o funcție recursivă, $f: \mathbb{N} \rightarrow \mathbb{N}$ cu $\text{dom}(f) = A$.
- b) Există un program $Q \in P_{0,1}$ care la fiecare apel calculează un element din A . Q este un generator al mulțimii A .

Propoziția D.7 Alternativele (a) și (b) din definiția (D.5) sunt echivalente.

Cazul 1: (a) $\Rightarrow$ (b). Deoarece $f \in F_{\mathbb{N}}$ rezultă că există un program $P \in P_{1,1}$ care calculează f . P se termină doar pentru valorile din mulțimea A . Folosind P , construim generatorul Q după cum urmează, considerând că variabilele A și n sunt *statice*, adică își păstrează valoarea între apelurile programului.

```
Q() {
    static A = Ø, n = 0; // variabile cu valori nevolatile
    while(1) {
        for(m=0; m ≤ n; m++)
            if(m ∉ A ∧ P(m) se termină în n unități de timp) A
                {A = A ∪ {m}; return m;}
        n++;
    }
}
```

Cazul 1: (b) $\Rightarrow$ (a). Presupunem că există un generator Q al lui A . Construim programul:

```
P(n) {
    static A = Ø;
    while(1)
        if(n ∈ A) return 1; else A = A ∪ {Q()};
}
```

Evident, P se termină cu rezultatul 1 doar pentru valorile ce sunt generate de Q , deci pentru valorile din A . Dacă $n \notin A$, n nu este generat de Q și programul $P(n)$ nu se termină. Conform definițiilor (D.2) și (D.3), programul P calculează o funcție recursivă cu domeniul A ■

Propoziția D.8 O submulțime recursivă $A \subset \mathbb{N}$ este recursiv-numărabilă.

Fie A o submulțime recursivă din $\mathbb{N}$. Există o funcție totală recursivă $f: \mathbb{N} \rightarrow \mathbb{N}$, deci un program P care calculează f , astfel încât $A = \{n \in \mathbb{N} \mid P(n)=1\}$. Construim generatorul lui A :

```
Q() {
    static A = ∅;
    for(n = 0;; n++)
        if(n ∉ A ∧ P(n) = 1) {A = A ∪ {n}; return n;}
}
```

Evident, Q produce toate valorile n pentru care $P(n)=1$, deci chiar mulțimea A . ■

Propoziția D.9 Fie $A \subset \mathbb{N}$ o submulțime recursivă-numărabilă și $B = \mathbb{N} \setminus A$ recursiv-numărabilă. Atunci A este recursivă.

Cazul A finită este banal, pentru că orice mulțime finită cu elemente din $\mathbb{N}$ este recursivă. Fie A infinită, iar QA și QB generatorii mulțimilor A și, respectiv, B . Pentru că A și B sunt infinite, orice apel al generatorilor QA și QB se termină în timp finit cu elemente din cele două mulțimi. Construim programul $P \in P_{1,1}$

```
P(x) {
    static A = ∅, B = ∅;
    while(1) {
        if(x ∈ A) return 1;
        if(x ∈ B) return 0;
        A = A ∪ {QA()};
        B = B ∪ {QB()};
    }
}
```

Programul P se termină pentru că toate elementele din A și B vor fi generate. Deci P corespunde unei funcții recursive totale care decide dacă $n \in \mathbb{N}$ este în A . ■

Propoziția D.10 Există submulțimi din $\mathbb{N}$ care:

1. sunt recursive;
2. sunt recursiv-numărabile, dar nu sunt recursive;
3. nu sunt recursive și nici recursiv-numărabile.

1. Mulțimea numerelor prime este recursivă. Se poate construi lesne un program care testează apartenența $n \in \text{Prime}$ pentru orice $n \in \mathbb{N}$.

2. Mulțimea $A = \{n \in \mathbb{N} \mid P_n(n) \text{ se termină}\}$, unde P_n este programul de indice n din mulțimea $P_{1,1}$, este recursiv-numărabilă, dar nu este recursivă.

A este recursiv-numărabilă. Să arătăm că există un generator Q pentru elementele din A .

```

Q() {
  static A = ∅, n = 0;
  while(1) {
    for(m=0; m ≤ n; m++)
      if(Pm(m) se termină în n unități de timp ∧ m ∉ A)
        {A = A ∪ {m}; return m;}
    n++;
  }
}

```

A nu este recursivă. Presupunem că A este recursivă. Atunci există o funcție recursivă totală și, deci, un program $R \in P_{1,1}$ astfel încât $R(n) = 1 + P_n(n)$, dacă $n \in A$, și $R(n) = 0$, dacă $n \notin A$.

Deoarece $P_{1,1}$ este numărabilă (conform teoremei 3.4), există $m \in \mathbb{N}$ astfel încât $R = P_m$. Programul R se termină pentru orice valoare din $\mathbb{N}$, inclusiv pentru m. Deci $m \in A$. Avem $R(m) = P_m(m) = 1 + P_m(m)$. Imposibil.

3. Fie mulțimea $B = \mathbb{N} \setminus A$, unde A este mulțimea de la punctul (b). B nu este recursivă și nici recursiv-numărabilă.

Să presupunem că B este recursiv-numărabilă. Atunci, conform propoziției (D.9) A este recursivă. Imposibil. Să presupunem că B este recursivă. Atunci conform propoziției (D.8) este recursiv-numărabilă. Imposibil. ■

Definiția D.6 Fie $\tau: \mathbb{N} \rightarrow \{0, 1\}$ un predicat și $M_\tau = \{n \in \mathbb{N} \mid \tau(n) = 1\}$ mulțimea de adevăr a lui τ . Predicatul τ caracterizează o proprietate Prop a elementelor din $\mathbb{N}$.

- τ este decidabil (proprietatea Prop este decidabilă) dacă M_τ este recursivă;
- τ este semi-decidabil (proprietatea Prop este semi-decidabilă) dacă M_τ este recursiv-numărabilă și nerecursivă;
- τ este nedecidabil (proprietatea Prop este nedecidabilă) dacă M_τ nu este recursivă și nici recursiv-numărabilă.

Exemplul D.1 Predicatul $\tau: \text{Grafuri} \rightarrow \{0, 1\}$, care testează dacă un graf finit este aciclic este decidabil, programul de test constând într-o parcurgere banală a grafului.

Exemplul D.2 Să demonstrăm propoziția (3.6'), arătând că predicatul τ care testează dacă un program oarecare $P_n \in P_{1,1}$ se termină pentru o valoare fixată $x \in \mathbb{N}$ este semi-decidabil.

Notăm M_τ mulțimea corespunzătoare programelor care se termină în x. Conform exemplului din propoziția (3.6) știm că M_τ nu este recursivă. Să arătăm că este recursiv-numărabilă.

Construim generatorul q :

```

Q() {
  static  $M_T = \emptyset$ ,  $t=0$ ;
  while(1) {
    for( $n=0$ ;  $n \leq t$ ;  $n++$ )
      if( $n \notin M_T \wedge P_n(x)$  se termină în  $t$  unități de timp)
        {  $M_T = M_T \cup \{n\}$ ; return  $n$ ; }
     $t++$ ;
  }
}

```

La fiecare apel, generatorul reîntoarce indicele unui program din $P_{1,1}$ care se termină pentru x . Rezultă că predicatul τ este semi-decidabil. ■

Exemplul D.3 Rezolvarea unei probleme poate fi imaginată ca un proces de navigare într-un spațiu $S=(\text{stări}, \text{Tranziții})$ al stărilor problemei, ca cel din figura 3.1. Atunci când navigăm într-un astfel de spațiu dorim să urmărim o cale care, plecând din starea curentă s , duce în mod cert spre o soluție.

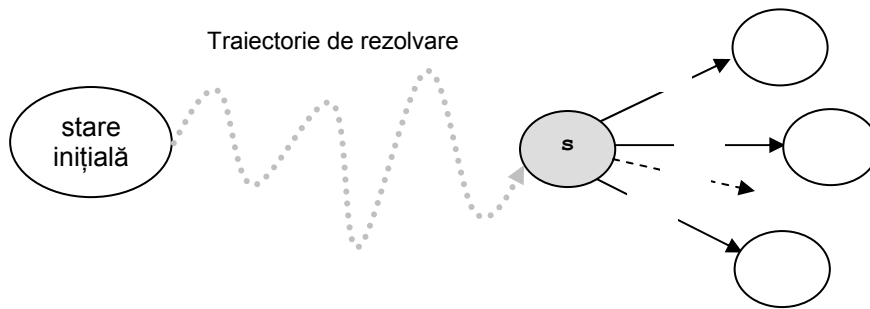


Figura 3.1 Navigare în spațiul de stare al unei probleme

Navigarea poate fi realizată cu ajutorul unui predicat $\tau: \text{Tranziții} \rightarrow \{0,1\}$ care determină ce arc $(s, ?)$ trebuie urmat. Dacă predicatul este decidabil atunci putem merge direct spre soluție. Dacă predicatul este semi-decidabil atunci o variantă ar fi să folosim un generator Q_T de arce din $M_T \subset \text{Tranziții}$, arce ce duc spre soluție, sperând să obținem un asemenea arc ce pleacă din starea curentă s .

```

avans_navigare(s) {
  static  $M_T = \emptyset$ ;
  while(1) {
    if ( $\exists (u,v) \in M_T \bullet u=s$ ) return (s,v);
     $M_T = M_T \cup \{Q_T()\}$ ;
  }
}

```

Într-o altă variantă, mai practică, putem folosi o euristică pentru a evalua arcul aparent cel mai promițător, fără a ne mai baza pe predicatul al cărui calcul s-ar putea să nu se termine într-un timp măsurabil. Astfel de probleme apar în domeniul inteligenței artificiale.

Exemplul D.4 Un sistem de rescriere (sau sistem semi-Thue, după numele matematicianului norvegian Axel Thue) constă dintr-un alfabet finit Σ și o mulțime finită de reguli de rescriere $R = \{u := v \mid u, v \in \Sigma^*\}$. O regulă $u := v$ arată că șirul finit u , format cu simboluri din Σ , poate fi rescris în forma v . Spunem că șirul $y \in \Sigma^*$ este derivabil din șirul $x \in \Sigma^*$ dacă există o succesiune de reguli care aplicate începând cu x produc șirul y . De exemplu, pentru $\Sigma = \{a, b, c\}$ și $R = \{a := bab, c := aca, a := c\}$, există derivarea $ac \xrightarrow{a := bab} babc \xrightarrow{c := aca} babaca \xrightarrow{a := c} bcbccc$.

Fie problema: să se decidă dacă un șir oarecare y poate fi derivat dintr-un șir oarecare x , conform unui sistem de rescriere dat. Predicatul care decide proprietatea este semi-decidabil și, implicit, problema este semidecidabilă. Similar, există probleme nedecidabile referitoare la proprietățile gramaticilor independente de context. Astfel, dacă $L(G_1)$ și $L(G_2)$ sunt limbajele generate de gramaticile independente de context G_1 și G_2 , atunci testul $L(G_1) \cap L(G_2) = \emptyset$ este semi-decidabil.

Teorema lui Rice

Un caz particular de nerezolvabilitate mecanică constă în probarea proprietăților *extensionale nebanale* ale programelor, problemă de decizie aflată la baza multor aplicații practice. Pentru simplitate, discuția de mai jos este focalizată asupra programelor $P_{1,1}$ cu o singură intrare și un singur rezultat din $\mathbb{N}$.

O proprietate a programelor este privită ca mulțime a programelor (sau a indicilor programelor) care satisfac respectiva proprietate. De exemplu, proprietatea

$$Pow2 =_{\text{def}} \{P \in P_{1,1} \mid (\forall n \in \mathbb{N} \bullet P(n) = n^2)\}$$

se poate citi: un program $P \in P_{1,1}$ are proprietatea $Pow2$, sau alternativ $P \in Pow2$, dacă $P(n)$ calculează n^2 .

Să notăm $P=Q$ echivalența computațională a programelor P și Q , anume pentru aceleași date de intrare programele fie produc aceleași rezultate fie nu se termină.

Definiția D.7 O proprietate $Prop$ a programelor $P_{1,1}$ este *extensională nebanală* dacă:

1. Nebanalitate: $Prop \neq \emptyset$ și $Prop \subset P_{1,1}$.
2. Extensionalitate: pentru oricare programe $P \in P_{1,1}$ și $Q \in P_{1,1}$, astfel încât $P=Q$, există dubla implicație $P \in Prop \Leftrightarrow Q \in Prop$. Cu alte cuvinte, dacă un program P are proprietatea $Prop$ atunci orice program echivalent cu P are proprietatea $Prop$.

Conform nebanalității, proprietatea caracterizează o submulțime proprie nevidă a programelor. Extensionalitatea reflectă comportarea programelor independent de factori intensionali cum ar fi timpul execuției programului, numărul de instrucțiuni din textul programului etc. Extensionalitatea se referă strict la relația funcțională $P(i) = r$ calculată de program. De exemplu, proprietatea $Pow2$ este extensională nebanală.

Teorema D.1 (Rice). Orice proprietate extensională nebanală a programelor nu este decidabilă.

Demonstrația se bazează pe stabilirea unei corespondențe între terminarea programelor și verificarea unei proprietăți nebanale extensionale, astfel încât, dacă proprietatea este decidabilă¹ atunci terminarea programelor este decidabilă.

Fie Prop o proprietate oarecare, extensională nebanală, a programelor $P_{1,1}$. Să presupunem că Prop este decidabilă.

Fie $\text{nonstop} \in P_{1,1}$ un program care nu se termină pentru orice $n \in \mathbb{N}$, anume $\forall n \in \mathbb{N} \bullet \text{nonstop}(n) = \perp$. De exemplu,

$\text{nonstop}(n) \text{ } \{\text{while}(1); \text{return } n;\}.$

Pentru că Prop este extensională nebanală, nonstop poate fie să aparțină lui Prop , fie să nu aparțină lui Prop . Din acest punct de vedere, alegerea lui nonstop poate să pară bizară. Aparent, în cazul $\text{nonstop} \in \text{Prop}$, mulțimea programelor $P_{1,1}$ este divizată în două submulțimi: $\text{Prop} = P_{\perp} = \{P \in P_{1,1} \mid (\forall n \in \mathbb{N} \bullet P(n) = \perp)\}$, ce conține doar programele echivalente computațional cu nonstop și care prin extensionalitate satisfac Prop , și programele din $P_{1,1} \setminus P_{\perp}$ care se termină pentru cel puțin o valoare din $\mathbb{N}$ și, nefiind echivalente cu nonstop , nu satisfac Prop .

Observația de mai sus nu este corectă. Într-adevăr, dacă $\text{Prop} =_{\text{def}} \{P \in P_{1,1} \mid \forall m \in \mathbb{N} \bullet P(m) \neq \perp \Rightarrow P(m) = f(m)\}$, unde $f(m)$ desemnează o valoare ce depinde de m , atunci în afara programelor din $P_{\perp}$ în Prop sunt și acele programe din $P_{1,1}$ care calculează valori conforme cu f , inclusiv programele care calculează funcția $f: \mathbb{N} \rightarrow \mathbb{N}$. Prin urmare, alegerea programului nonstop ca bază a demonstrației nu afectează generalitatea acesteia.

Caz 1. $\text{nonstop} \in \text{Prop}$

Datorită extensionalității lui Prop , toate programele echivalente computațional cu nonstop sunt în Prop . De asemenea, datorită nebanalității, există cel puțin un program $Q \in P_{1,1}$ astfel încât $Q \notin \text{Prop}$ și, implicit, $Q \neq \text{nonstop}$, deci Q se termină pentru cel puțin o valoare din $\mathbb{N}$.

Fie $P \in P_{1,1}$ un program oarecare. Să arătăm că deciderea terminării $P(x)$, pentru $x \in \mathbb{N}$ fixat, se reduce la decidabilitatea lui Prop . Construim programul:

$R(n) \text{ } \{P(x); \text{return } Q(n);\}$

¹ O proprietate Prop este decidabilă dacă există un program $\text{Test}_{\text{Prop}} \in P_{1,1}$ care se termină pentru orice $n \in \mathbb{N}$ astfel încât $\text{Test}_{\text{Prop}}(n) = 1$ dacă programul P_n din $P_{1,1}$ este în Prop și $\text{Test}_{\text{Prop}}(n) = 0$ dacă $P_n \notin \text{Prop}$.

Dacă $P(x)$ se termină atunci $R(n)=Q(n)$, eventual $R(n)=Q(n)=\perp$. Dacă $P(x)$ nu se termină atunci $R(n)$ nu se termină pentru orice $n \in \mathbb{N}$. Prin urmare, există echivalența computațională:

$$R = \begin{cases} \text{nonstop, dacă } P(x) = \perp \\ Q, \text{ dacă } P(x) \neq \perp \end{cases}$$

Dar, datorită extensionalității proprietății Prop , echivalența $R=\text{nonstop}$ impune $R \in \text{Prop}$, iar echivalența $R=Q$ impune $R \notin \text{Prop}$. Prin urmare, dacă $R \in \text{Prop}$ atunci $P(x)$ nu se termină și, reciproc, dacă $R \notin \text{Prop}$ atunci $P(x)$ se termină.

Caz 2. $\text{nonstop} \notin \text{Prop}$.

Atunci $\text{nonstop} \in \overline{\text{Prop}}$, unde $\overline{\text{Prop}} = P_{1,1} \setminus \text{Prop}$. În acest caz, folosind un raționament similar cu cel de la cazul (1) se ajunge la concluzia că dacă $R \in \overline{\text{Prop}}$ atunci $P(x)$ nu se termină și, reciproc, dacă $R \in \text{Prop}$ atunci $P(x)$ se termină.

În ambele cazuri rezultă că dacă Prop este decidabilă atunci terminarea $P(x)$ este decidabilă. Imposibil, conform propoziției (D.6'). Deci, ipoteza conform căreia Prop este decidabilă nu poate fi adevărată. ■

Teorema lui Rice are implicații practice imediate. Bunăoară, echivalența programelor, care se reduce la verificarea proprietății $P(n)=f(n)$, unde f este o funcție care descrie comportarea dorită a programului, este nedecidabilă. Prin urmare, o aplicație care selectează un program dintr-o bibliotecă de programe folosind exclusiv codul programului și specificația ce formează cheia de căutare este nerealizabilă, în cazul general. De asemenea, o problemă de genul "să se verifice dacă modulul de program m poate funcționa corect pentru datele d ", unde m și d sunt alese la întâmplare, este nedecidabilă doar pe baza analizei mecanice a codului modulului. Este necesară o informație explicită care, pentru fiecare modul m , să indice datele ce pot fi prelucrate și, eventual, alte particularități ale modulului.