Decidabilitate

Rezolvarea mecanică a unei probleme ridică două întrebări esențiale: (1) problema este rezolvabilă mecanic; (2) există algoritmi eficienți de rezolvare? Răspunsul poate evita căutarea imposibilului şi poate sugera alternative de rezolvare, cum ar fi aproximarea soluției sau rezolvarea euristică. Abia apoi, odată construit, algoritmul trebuie analizat din punctul de vedere al corectitudinii şi performanțelor. Cursul prezintă elemente introductive din teoria calculabilității, inclusiv conceptul esențial de decidabilitate. Datorită caracterului intuitiv, se preferă explicarea acestui concept folosind platforma nedeterminismului, deşi este posibilă o abordare din perspectiva limbajelor formale.

D.1 Decidabilitate

Teoria calculabilității sau, echivalent, studiul limitelor de rezolvare a problemelor folosind maşini de calcul, foloseşte proprietăți ale mulțimilor, printre care esențiale sunt numărabilitatea şi recursivitatea. Orice abordare, chiar dacă este cu tentă intuitivă – aşa cum este cea de față – nu poate deci evita prezentarea unor elemente de bază din teoria mulțimilor. Proprietățile elementare de mai jos se referă la mulțimi infinite, pentru care respectivele proprietăți nu sunt banale. De asemenea, majoritatea problemelor interesante în practică se referă la mulțimi infinite.

- O mulțime A este echipotentă cu o mulțime B dacă există $f:A\to B$, bijectivă.

- O mulțime infinită A este numărabilă dacă A este echipotentă cu mulțimea numerelor naturale N. O numerotare a mulțimii A este o funcție bijectivă $a: N \to A$, astfel încât $A = \{a_k \mid k\in N\}$, unde – prin notație – a_n este $a(n)$. Echivalent, o mulțime este numărabilă dacă există o numerotare a sa.

Spunem că o mulțime infinită şi numărabilă este ∞-numărabilă, iar o mulțime infinită şi nenumărabilă este ∞-nenumărabilă. Evident, o mulțime finită este numărabilă, fiind echipotentă cu o submulțime finită din N.

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

Fie A ∞-numărabilă şi $B \subseteq A$, B infinită. Există numerotarea $A = \{a_k \mid k \in N\}$ Construim numerotarea[1]:

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

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

Propoziția D.2 Fie A ∞-nenumărabilă, iar B o mulțime ∞-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 ∞-numărabilă. Imposibil. ■

Propoziția D.3 Mulțimea $Hom(N,N)$ (a funcțiilor din N în N) este ∞-nenumărabilă.

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

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

Funcția g, totală peste N, decide dacă un element oarecare din N aparține mulțimii A (sau, echivalent, spunem că g decide A).

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 şi deci funcția f nu este surjectivă. Înseamnă că N nu este echipotentă cu $Hom(N,N)$. ■

Numărabilitatea unei mulțimi $A \subseteq N$ este în strânsă legătură cu existența unui program, acceptat de o maşină de calcul nespecificată, capabil să decidă apartenența unui element oarecare $x \in N$ la A. Fără a restrânge generalitatea discuției, considerăm că maşina de calcul (abstractă) folosită este o maşină Turing (vezi Calculabilitate.pdf).

[1] Demonstrația arată doar că numerotarea b există. Existența unui program care să calculeze b este o altă problemă care se referă la "calculabilitatea" funcției b. Răspunsul depinde de proprietățile funcției b.

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

În ceea ce priveşte limbajul în care sunt scrise programele, o să-l imaginăm ca un limbaj de programare asemănător cu C. Perspectiva pragmatică este corectă deoarece programele scrise în asemenea limbaje de programare de nivel înalt pot fi convertite în programe echivalente scrise în "cod maşină Turing".

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

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

$$w_0 = \lambda$$
$$w_j = \min_{lex}\{w \in \Sigma^* \setminus \bigcup_{k<j}\{w_k\}\},\ j \geq 1,$$

unde $\min_{lex}(M)$ desemnează şirul cel mai mic în ordine lexicografică[2] din M. Fie $\Sigma_{i,j} \subseteq \Sigma^*$ mulțimea şirurilor ce codifică programe din $P_{i,j}$. Conform propoziției (D.1) mulțimea $\Sigma_{i,j}$ este numărabilă. Conform echipotenței dintre $\Sigma_{i,j}$ şi $P_{i,j}$, rezultă că $P_{i,j}$ este numărabilă. ■

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

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

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

Primul concept se referă la calculabilitate efectivă şi este de natură filozofică. O funcție este efectiv calculabilă dacă există o procedură efectivă care calculează valoarea funcției pentru orice parametru din domeniul de definiție al acesteia. Prin procedură efectivă se înțelege un proces de calcul finit (care foloseşte reprezentări finite ale datelor şi se termină după o perioadă finită de timp), bine definit (prelucrările sunt specificate neambiguu încât pot fi executate mecanic) şi concluziv (care produce

[2] Mulțimea Σ este finită şi, deci, numărabilă. Înseamnă că există o relație de ordine totală peste Σ, relație ce stă la baza unei relații de ordine lexicografică totală peste Σ^*.

un rezultat ce reprezintă valoarea funcției). Deoarece însăşi 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 maşină de calcul particulară, anume maşina 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 maşini Turing. Cu alte cuvinte, teza susține că Turing-calculabilitatea este universală şi, implicit, că funcțiile efectiv calculabile sunt cele recursive. Deşi teza nu a fost demonstrată, există suport teoretic consistent care pledează în favoarea ei. Bunăoară, s-au inventat maşini abstracte de calcul (teorii matematice ale calculului) aparent diferite din perspectiva modului în care este reprezentată problema rezolvată şi este imaginat procesul de calcul aferent rezolvării, demonstrându-se echivalența lor cu maşina Turing.

Conform tezei Church-Turing admitem că orice funcție efectiv calculabilă este recursivă. Astfel, putem defini intuitiv o funcție recursivă ca fiind programabilă şi, reciproc, putem susține că orice program (indiferent de limbajul de programare în care este scris) calculează o funcție recursivă. O perspectivă formală a funcțiilor recursive este prezentată în "Calculabilitate.pdf".

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

Propoziția D.5 Mulțimea F_r este ∞-numărabilă.

Construim numerotarea:

- f_0 este funcția calculată de programul cu indicele P_0.
- f_j, $j \geq 1$, este funcția calculată de programul cu indicele $\min\{k \mid P_k \in Prog_j\}$, unde $Prog_j = P_{1,1} \setminus \{P \in P_{1,1} \mid P$ calculează $f_k, k < j\}$ sunt programele de calcul ale funcțiilor deja numerotate. ■

Propoziția D.6 Există în $Hom(N,N)$ funcții care nu sunt recursive. Ca urmare a tezei Church-Turing, asemenea funcții nu sunt calculabile, indiferent de maşina de calcul folosită.

Demonstrația rezultă direct din propozițiile (D.2), (D.3) şi (D.5). Ca exemplu considerăm funcția:

$$f(n) = \begin{cases} 1+P_n(n), \text{ dacă } P_n(n) \text{ se termină} \\ 0, \text{ dacă } P_n(n) = \perp \end{cases}$$

unde P_n este programul de indice n din mulțimea $P_{1,1}$. Să presupunem că $f \in F_r$. 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 $x \in N$, deci şi pentru m. Conform definiției lui f, rezultă $P_m(m) = f(m) = 1 + P_m(m)$. Imposibil. ■

Propoziția (D.6) subliniază limita intrinsecă a calculului efectiv. Indiferent de limbaj, există probleme care nu pot fi rezolvate mecanic, folosind o maşină 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 şi funcții parțial-recursive (vezi "Calculabilitate.pdf"), 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 $n \in N$. În schimb, programul care calculează o funcție parțial-recursivă f se termină doar în punctele $n \in dom(f)$. 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 şi deci nu poate fi primitiv-recursivă. Dacă $dom(f)$ este restrâns la mulțimea indiciilor n astfel încât $P_n(n)$ se termină, atunci f devine efectiv-calculabilă. Într-adevăr, programul `R(n){return 1+P`$_n$`(n);}` se termină dacă $P_n(n)$ se termină. Mai mult, dacă m este indicele programului R, atunci $m \notin dom(f)$ pentru a avea $R(m) = P_m(m) = \perp$. Funcția f este parțial-recursivă şi poate verifica doar terminarea programelor $P_n(n)$ care se termină. Dacă $dom(f)$ ar fi cunoscut apriori, funcția f ar fi banală. Cum domeniul $dom(f)$ nu este cunoscut apriori nu putem şti dacă $R(n)$ se termină şi spunem că f semi-decide terminarea programelor din $P_{1,1}$. Prin urmare, cazul particular al propoziției (D.6) "terminarea programelor nu este poate fi decisă mecanic" poate fi nuanțat în forma propoziției (D.6').

Propoziția D.6' Terminarea programelor este semi-decidabilă. Demonstrația se găseşte în exemplul (D.2) referitor la definiția (D.6).

În urma discuției de mai sus, observăm că o submulțime $A \subset N$ poate fi cunoscută în două moduri: (1) prin intermediul unei funcții recursive totale sau, echivalent, folosind un program care decide dacă un element x aparține sau nu mulțimii A; (2) cu ajutorul unei funcții parțial-recursive cu domeniul A sau, echivalent, folosind un program care se termină doar dacă $x \in A$.

Rezolvarea mecanică a unei probleme, pentru care există un model reprezentat de o funcție recursivă f, este echivalentă cu deciderea apartenenței unui element (soluție potențială) la o mulțime dată (de soluții). Din acest punct de vedere, trebuie precizată legătura dintre calculul unui funcții şi decidabilitatea unei mulțimi.

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

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

Orice submulțime finită $A \subset N$ este recursivă: A este numărabilă, iar egalitatea numerelor naturale poate fi decisă mecanic. Putem construi lesne un program care decide apartenența unui element oarecare $n \in N$ la A, program bazat pe operația de comparare a numerelor.

```
P(n) {
   for(i=0; i < card(A); i++) if(n = A_i) 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 (deci numărabilă) este recursivă.

Definiția D.5 O submulțime $A \subseteq N$ este recursiv-numărabilă dacă (alternativ):

a) Există o funcție recursivă, $f: N \to N$ cu $dom(f)=A$.
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.

Un caz limită îl constituie mulțimea vidă pentru care funcția f este total nedefinită, calculabilă de către orice program $P \in P_{1,1}$ astfel încât $\forall x \in N \bullet P(x) = \perp$. De asemenea, în acest caz, apelul generatorului Q nu se termină. Alt caz limită este $A=N$, iar $f(x)=1$ pentru orice $x \in N$.

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

Cazul 1: (a) $\Rightarrow$ (b). Deoarece $f \in F_r$ 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 şi n sunt *statice*, adică îşi păstrează valoarea între apelurile programului.

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

Cazul 1: (b) $\Rightarrow$ (a). Fie Q un generator al mulțimii A. Construim programul:

```
P(n){
    static A = ∅;
    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∉A`, `n` nu este generat de `Q` şi programul `P(n)` nu se termină. Conform definiţiilor (D.2) şi (D.3), programul `P` calculează o funcţie recursivă cu domeniul `A` ■

Observând demonstraţia de mai sus, dacă mulţimea `A` este finită atunci generatorul `Q` nu se termină după ce a produs ultimul element din `A`. Dar, cunoscând cardinalitatea lui `A` se poate evita un asemenea apel al lui `Q`. Dacă nu există nici o informaţie relativă la mulţimea `A` atunci terminarea apelului lui `Q` nu poate fi determinată apriori. Astfel, intuitiv, recursiv-numărabilitatea unei mulţimi, considerată ca bază de rezolvare a unei probleme, conduce la o rezolvare "nesigură" în sensul că nu se poate şti apriori dacă procesul de rezolvare se termină şi, implicit, dacă problema are sau nu are soluţie.

Propoziţia D.8 O submulţime recursivă $A \subseteq N$ este recursiv-numărabilă.

Fie `A` o submulţime recursivă din `N`. Există o funcţie totală recursivă $f: N \to N$, deci un program `P` care calculează `f`, astfel încât $A = \{n \in N \mid P(n)=1\}$. 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 mulţimea `A`. ■

Propoziţia D.9 Fie $A \subseteq N$ o submulţime recursiv-numărabilă şi $B = N \setminus A$ recursiv-numărabilă. Atunci `A` este recursivă.

Cazul `A` este finită este banal; orice submulţime finită din `N` este recursivă. Dacă `B` este finită atunci este recursivă, iar testul $x \in A$ se reduce la $\neg(x \in B)$, deci `A` este recursivă. Pentru cazul `A` şi `B` mulţimi infinite, fie `QA` şi `QB` generatorii mulţimilor `A` şi, respectiv, `B`. Deoarece mulţimile sunt infinite, orice apel al generatorilor `QA` şi `QB` se termină în timp finit cu elemente din cele două mulţimi. Construim programul:

```
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` şi `B` vor fi generate. Deci `P` corespunde unei funcţii recursive totale care decide dacă $n \in N$ este în `A`. ■

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

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

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

2. Mulțimea $A = \{n \in \mathbf{N} \mid P_n(n)$ 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 şi, deci, un program $R \in P_{1,1}$ astfel încât $R(n) = 1+P_n(n)$, dacă $n \in A$, şi $R(n) = 0$, dacă $n \notin A$.

Deoarece $P_{1,1}$ este numărabilă (conform teoremei D.4), există $m \in \mathbf{N}$ astfel încât $R=P_m$. Programul R se termină pentru orice valoare din $\mathbf{N}$, inclusiv pentru m. Deci $m \in A$. Avem $R(m) = P_m(m) = 1 + P_m(m)$. Imposibil.

3. Fie mulțimea $B = \mathbf{N} \setminus A$, unde A este mulțimea de la punctul (b). B nu este recursivă şi 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 $T: \mathbf{N} \to \{0,1\}$ un predicat şi $M_T = \{n \in \mathbf{N} \mid T(n)=1\}$ mulțimea de adevăr a lui T, mulțime care caracterizează o proprietate $Prop$ a elementelor din $\mathbf{N}$.

- T este decidabil (proprietatea $Prop$ este decidabilă) dacă M_T este recursivă;
- T este semi-decidabil (proprietatea $Prop$ este semi-decidabilă) dacă M_T este recursiv-numărabilă şi nerecursivă;
- T este nedecidabil (proprietatea $Prop$ este nedecidabilă) dacă M_T nu este recursivă şi nici recursiv-numărabilă.

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

Exemplul D.2 Să demonstrăm propoziția (D.6'), arătând că predicatul T care testează dacă un program oarecare $P_n \in P_{1,1}$ se termină pentru o valoare fixată $x \in N$ este semi-decidabil.

Notăm M_T mulțimea corespunzătoare programelor care se termină în x. Conform exemplului din propoziția (D.6) ştim că M_T nu este recursivă. Să arătăm că este recursiv-numărabilă. Construim generatorul Q, ca de mai jos.

```
Q() {
    static M_T=∅, t=0;
    while(1) {
       for(n=0; n≤t; n++)
          if(n∉M_T ∧ P_n(x) se termină în t unități de timp)
            { M_T= M_T∪{n}; return n;}
       t++;
    }
}
```

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

Exemplul D.3 Rezolvarea unei probleme poate fi imaginată ca un proces de navigare într-un spațiu $S=(Stări, Tranziții)$ al stărilor problemei, ca cel din figura D.1. Atunci când navigăm într-un astfel de spațiu dorim să urmăm o cale care, plecând din starea curentă s, duce în mod cert spre o soluție.

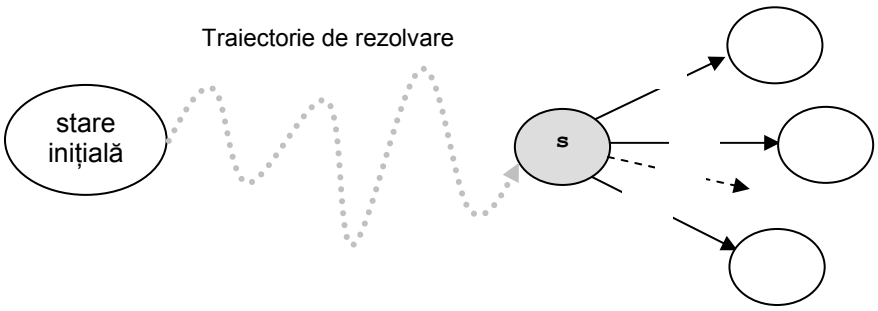

Figura D.1 Navigare în spațiul de stare al unei probleme

Navigarea poate fi realizată cu ajutorul unui predicat $T:Tranziții \rightarrow \{0,1\}$ 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 $M_T \subset Tranziții$, arce ce duc spre soluție, sperând să obținem un asemenea arc ce pleacă din starea curentă s.

```
avans_navigare(s) {
     static M_T = ∅;
     while(1) {
       if (∃(u,v)∈ M_T • u=s) return (s,v);
       M_T = M_T ∪ {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 şi o mulțime finită de reguli de rescriere $R=\{u::=v \mid u,v\in\Sigma^*\}$. O regulă $u::=v$ arată că şirul finit u, format cu simboluri din Σ, poate fi rescris în forma v. Spunem că şirul $y\in\Sigma^*$ este derivabil din şirul $x\in\Sigma^*$ dacă există o succesiune de reguli care aplicate începând cu x produc şirul y. De exemplu, pentru $\Sigma=\{a,b,c\}$ şi $R=\{a::=bab,\ c::=aca,\ a::=c\}$, există derivarea $ac \xrightarrow{a::=bab} babc \xrightarrow{c::=aca} babaca \xrightarrow{a::=c} bcbccc$.

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

Teorema lui Rice

Un caz particular (dar important) 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 şi un singur rezultat din N.

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

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

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 şi Q, anume pentru aceleaşi date de intrare programele fie produc aceleaşi rezultate fie nu se termină.

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

1. Nebanalitate: $Prop \neq \emptyset$ şi $Prop \subset P_{1,1}$.

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

Conform nebanalităţii, proprietatea caracterizează o submulţime proprie nevidă a programelor. Extensionalitatea reflectă comportarea computaţională a programelor independent de factori intensionali cum ar fi timpul execuţiei programului, numărul de instrucţiuni din textul programului etc. Extensionalitatea se referă strict la relaţia funcţională `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.

Demonstraţia se bazează pe stabilirea unei corespondenţe între terminarea programelor şi verificarea unei proprietăţi nebanale extensionale, astfel încât, dacă proprietatea este decidabilă[3] atunci terminarea programelor este decidabilă.

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

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

```
nonstop(n) {while(1); return n;}.
```

Pentru că `Prop` este extensională nebanală, `nonstop` poate fie să aparţină lui `Prop`, fie să nu aparţină lui `Prop`. Din acest punct de vedere, alegerea lui `nonstop` poate să pară bizară. Aparent, în cazul $nonstop \in Prop$, mulţimea programelor $P_{1,1}$ este divizată în două submulţimi: $Prop=P_{\perp}=\{P \in P_{1,1} \mid (\forall n \in N \bullet P(n)=\perp)\}$, ce conţine doar programele echivalente computaţional cu `nonstop` şi care prin extensionalitate satisfac `Prop`, şi programele din $P_{1,1} \setminus P_{\perp}$ care se termină pentru cel puţin o valoare din N şi, nefiind echivalente cu `nonstop`, nu satisfac `Prop`.

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

---

[3] O proprietate `Prop` este decidabilă dacă există un program $Test_{Prop} \in P_{1,1}$ care se termină pentru orice $n \in N$ astfel încât $Test_{Prop}(n)=1$ dacă programul P_n din $P_{1,1}$ este în `Prop` şi $Test_{Prop}(n)=0$ dacă $P_n \notin Prop$.

Caz 1. $nonstop \in Prop$

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 $Q \notin Prop$ şi, implicit, $Q \neq nonstop$, deci Q se termină pentru cel puțin o valoare din N.

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

```
R(n){P(x); return Q(n);}
```

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 $n \in N$. Prin urmare, există echivalența computațională:

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

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

Caz 2. $nonstop \notin Prop$.

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