Calculabilitate

- Seminar 1 -

October 9, 2022

Contents

1 Motivație

Ne propunem sa identificăm acele probleme care pot fi rezolvate mecanic. Vom descoperi că există un număr foarte mare de probleme practice pentru care nu putem construi o soluție care să meargă pentru **orice** intrare validă. De exemplu:

- Dacă modificăm un program, nu putem verifica automat dacă noul program este echivalent computațional (pentru aceleași intrări, produce același rezultat) cu programul inițial.
- Dat fiind un program oarecare și o intrare validă, nu putem estima automat proprietăți precum timpul de execuție, memoria consumată sau de câte ori sunt citite informații de pe disc.
- Nu putem verifica automat dacă un program oarecare este vulnerabil (e.g. un utilizator neautorizat poate accesa informații confidențiale). Analog, nu putem verifica automat dacă un program este un virus sau nu.

2 Tipuri de probleme

Printr-o problemă înțelegem o mulțime de întrebări, referitoare la proprietățile unor entități, care acceptă răspunsuri precise, finite și a căror corectitudine poate fi riguros demonstrată. Există mai multe tipuri de probleme, dintre care menționăm:

- **Probleme de decizie**: Verifică o proprietate a datelor de intrare (ex: "Este numărul X prim?"). Soluțiile sunt de tip boolean: "DA"/"NU", "Adevarat/Fals", etc.
- **Probleme de optimizare**: Selectează soluția/soluțiile care optimizează o funcție de cost (ex: "Care este cel mai mare divizor prim al numărului X?"). În multe cazuri, problemele de optimizare pot fi reformulate ca probleme de decizie.

3 Calculabilitate

Fie funcția $f_P : I \to O$ care leagă fiecare intrare validă pentru problema Q de răspunsul corect pentru intrarea respectivă.

Definitia 3.1. Spunem că f_P este **efectiv calculabilă** (eng. computable) sau **recursivă** dacă există o metodă efectivă de rezolvare a problemei Q. [1, 2]

Definitia 3.2. O **metoda efectivă** (eng. effective method) este o mulțime finită de instrucțiuni, a căror execuție se termină în timp finit și implică o cantitate finită de date pentru a obține un rezultat, mereu același pentru date identice.

În particular, dacă pentru o problemă de decizie există o metodă efectivă de rezolvare vom spune că este **decidabilă** (eng. decidable). De exemplu, verificarea dacă un număr natural este prim este o problemă decidabilă, deoarece putem defini următorul program:

Listing 1: Verificare dacă un număr este prim in pseudo-cod

```
def checkPrime(x)

    if (x <= 1)
        return False

    for (d = 2; d * d <= x; d++)
        if (x % d == 0)
            return False

    return True
```

Acest program poate fi considerat o metodă efectivă fiindcă se termină pe orice intrare și returnează mereu răspunsul corect.

3.1 Cât de puternic este un limbaj de programare?

Un algoritm poate fi scris în cuvinte (dacă descrierea este riguroasă și lipsită de ambiguitate), în pseudocod, într-un limbaj de programare (C, Java, Python), într-un sistem formal (calcul lambda, logică combinațională, formule aritmetice), sau într-un alt limbaj caracteristic unui sistem care permite manipularea de date (automate pe stări, automate pushdown).

Dar sunt aceste medii de programare echivalente computațional? Este posibil ca un anumit limbaj să nu permită scrierea unui algoritm, respectiv să nu poată fi folosit pentru rezolvarea unei anumite probleme? Avem nevoie de un etalon comun pentru a le putea compara. În acest scop vom folosi **mașina Turing**.

Imaginați-vă o mașină care funcționează cu o bandă infinită de simboluri, un set finit de stări interne și un cap de control așezat pe bandă, capabil sa scaneze simbolurile de sub el. Inițial, pe bandă este scrisă doar intrarea, restul de slot-uri din bandă având simbolul nul. Capul este așezat pe primul simbol al intrării, iar starea internă este o anumită stare inițială. Fiecare instrucțiune a mașinii este de forma: "Când mașina este în starea internă q, și capul de

control se află peste simbolul X, suprascrie simbolul X cu Y, intră în starea internă w și mută capul la stânga/la dreapta". Când mașina ajunge într-o combinație simbol/stare pentru care nu are instrucțiune, se blochează. Mașina se oprește și returnează *Adevărat* sau *Fals* când și dacă ajunge într-una din cele două stări aferente fiecărui răspuns.

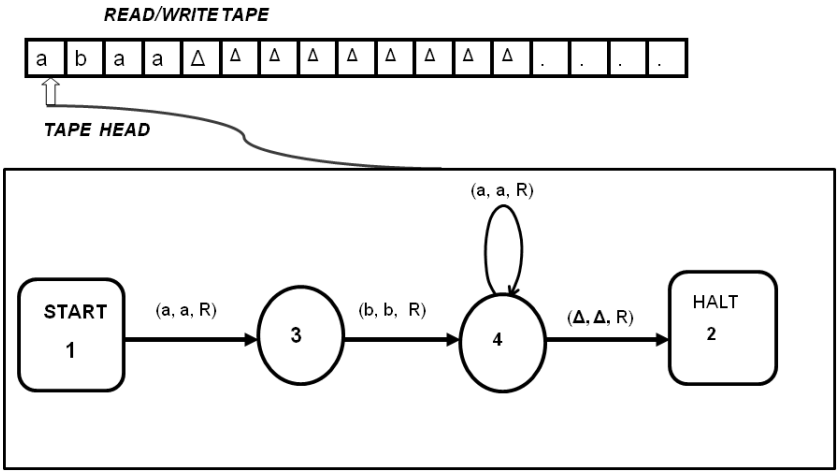

Figure 1: Exemplu Mașină Turing[1]
Pe fiecare muchie avem (*simbol curent*, *simbol înlocuitor*, *direcția de mutat capul*)

Definitia 3.3. O funcție este **calculabilă în sens Turing** (eng. Turing computable) daca există o mașină Turing care se termină în timp finit și returnează răspunsul corect pentru orice intrare validă.

Sună cam primitiv, nu? Pentru a specifica o mașina Turing care verifică dacă un număr este prim am avea nevoie de sute de stări interne, respectiv să executăm mii de instrucțiuni pentru a realiza verificarea propriu-zisă. Din acest motiv, nimeni nu folosește în practică mașinile Turing pentru a descrie algoritmi. Totuși, mașina Turing are o importanță teoretică foarte mare:

Teza Church Turing. Orice funcție este **efectiv calculabilă** dacă și numai dacă este **calculabilă în sens Turing**.

Teza enunța că **orice** problemă care poate fi rezolvată printr-o metodă efectivă poate fi rezolvată și de o mașină Turing.

Definitia 3.4. Spunem că un mediu de programare este **Turing complete** dacă poate fi folosit pentru a simula orice mașina Turing.

Practic, mediile de programare Turing complete sunt echivalente computațional, ele putând să rezolve aceleași probleme, respectiv toate problemele care pot fi rezolvate.

4 Tipuri de mulțimi

În acest laborator vom considera că mulțimile intrărilor valide ale programelor sunt ***numărabile***, adică exista o funcție bijectiva intre acestea și mulțimea numerelor naturale. O asemenea funcție bijectivă definește o numerotare a mulțimii/o ordine a elementelor ei. Un exemplu de mulțime numărabilă este mulțimea șirurilor de caractere peste un alfabet finit, pentru că acestea pot fi ordonate(a se vedea pentru un exemplu observația 1 din secțiunea 7).

[1]Animație MT: http://www.asethome.org/mathfoundations/tmd/

- Fie funcția $f_P : I \to Bool$ asociată problemei de decizie Q.
- (Reminder) Dacă există un program care pentru orice intrare $x \in I$ calculează în timp finit răspunsul pentru problema Q, atunci Q este **decidabilă**.
- Fie $A \subseteq I$ mulțimea de intrări pentru care răspunsul corect la problemă este "Da"/"Adevărat".

Definitia 4.1. O mulțime A este **recursivă** dacă și numai dacă există un program P care răspunde în timp finit dacă un element aparține sau nu mulțimii A.

$$P(x) = \begin{cases} 1 & \text{pentru } x \in A \\ 0 & \text{pentru } x \notin A \end{cases}$$

Definitia 4.2. Spunem că o mulțime A este **recursiv-numărabilă** dacă (alternativ):

- Există un program care calculează răspunsul în timp finit pentru un element $x \in I$ dacă $x \in A$ (dar poate să cicleze la infinit pentru $x \notin A$).

$$P(x) = \begin{cases} 1 & \text{pentru } x \in A \\ \perp & \text{pentru } x \notin A \end{cases}$$

- Există un program G care la fiecare apel generează succesiv un element din A. Cu alte cuvinte, orice element $x \in A$ va fi returnat după un număr finit de apeluri ale funcției G.

5 Exerciții Rezolvate

1. Demonstrați că următoarele proprietăți sunt echivalente pentru o mulțime A (care în cazul în care sunt îndeplinite este recursiv enumerabilă):

   - Există un program P astfel încât:

   $$P(x) = \begin{cases} 1 & \text{pentru } x \in A \\ \perp & \text{pentru } x \notin A \end{cases}$$

   - Există un program G care la fiecare apel generează un element din A și garantează că va returna oricare element din A după un număr finit de apeluri.

   Presupunem ca mulțimea intrărilor pentru programul P este mulțimea numerelor naturale.

   Rezolvare:
   Începem cu demonstrația $P \to G$ (folosind P putem obține G).

   Programul $G()$ ar trebui să întoarcă la fiecare apel un element nou din mulțimea A. Pentru asta, vom folosi o variabilă statică A care va memora elementele întoarse deja și își va păstra starea între apelurile funcției.

```
G(){
    static A = {}
    for (t = 0; ; t++)
        for (x = 0; x <= t; x++)
            if (x ∉ A) {
```

```
                    ruleaza P(x) pentru t pasi
                    if (P(x) s-a oprit) {
                        A = A ∪ {x}
                        return x
                    }
                }
        }
```

Ca să verificăm, trebuie să ne punem următoarele întrebări despre funcția $G()$:

- Este posibil ca ea să returneze o valoare care nu este membru a mulțimii A?
  Nu, deoarece $G()$ nu returnează decât valorile de input pentru care programul P s-a oprit, iar prin definiție programul P nu se oprește decât pentru valori membre ale lui A.
- Este posibil ca un element al lui A să nu fie returnat niciodată de $G()$?
  Nu, pentru orice membru x al lui A, programul P se va opri după un număr finit de pași, e. După un număr finit de apeluri succesive ale lui $G()$, următorul apel va ajunge la un pas $t \geq max(e, x)$ în care x va fi adăugat la soluție și va fi întors de $G()$.

Acum mai trebuie să demonstrăm dependența $G \rightarrow P$, care este mai simplă:
Presupunem că există un generator G și îl definim pe P astfel:

```
int P(int x) {
    while (G() != x);
    return 1;
}
```

Din moment ce $G()$ garantează că va returna oricare intrare acceptată după un număr finit de apeluri, avem garanția și că P se va termina dacă și numai dacă $x \in A$. Deci P este corect definit.

Deducțiile în ambele sensuri au fost demonstrate deci echivalența este adevărată.

2. Fie două mulțimi, A și $B = N \backslash A$, ambele recursiv-enumerabile. Demonstrați că ambele sunt recursive.

Rezolvare:
Dacă cele două mulțimi sunt recursiv-enumerabile inseamnă că fiecare are câte o funcție generator, fie GA și GB. Cu ele putem defini următorul program:

```
int P(int x) {
    int sol;
    while (true) {
        if (GA() == x) {
            sol = 1;
            break;
        }
        if (GB() == x) {
            sol = 0;
```

```
            break;
        }
    }
    return sol;
}
```

Din moment ce A și B sunt complementare știm că orice $x \in N$ aparține fie lui A fie lui B. Fiecare dintre funcțiile generator garantează să returneze orice membru al său după un număr finit de apeluri deci P este garantat să se termine și să returneze 1 pentru $x \in A$ și 0 pentru $x \in B$.
Acest program demonstrează recursivitatea ambelor mulțimi. Pentru a respecta definiția și pentru mulțimea B nu trebuie decât să negăm rezultatul lui P.

3. Fie mulțimile A, B, C astfel încât: A este recursivă, $A \cup B$ nu este nici măcar recursiv-enumerabilă, iar $B \oplus C$ este recursivă.
Ce se poate spune despre B și C?

Rezolvare:
În continuare ne vom referi la mulțimea mulțimilor recursive ca R, mulțimea mulțimilor recursiv enumerabile ca RE și mulțimea mulțimilor nici măcar recursiv enumerabile ca NRE.
Din moment ce $A \in R$ și $A \cup B \in NRE$ intuim că $B \in NRE$. Vom demonstra acest lucru printr-o contradicție. Prespunem că B este recursiv-enumerabilă, deci există un program PB care se oprește și returnează 1 pentru toți membrii lui B. Știm că A este recursivă deci există un program PA care se termină pe orice input și returnează corect aparenența la A.
Vom defini următorul program $PAuB$:

```
int PAuB(int x) {
    if PA(x) return 1;
    else return PB(x);
}
```

Acest program se termină și returnează 1 pentru orice $x \in A \cup B$. Asta inseamnă că $A \cup B \in RE$ dar $A \cup B \in NRE$. Contradicție.
De aici rezultă că B nu este nici măcar recursiv-enumerabilă, $B \in NRE$.

Fiindcă $B \oplus C$ este recursivă iar B nu este nici măcar recursiv-enumerabilă putem deduce faptul că C este nerecursivă. Facem acest lucru din nou prin reducere la absurd: Presupunem că C este recursivă, deci există un program PC care se oprește și returnează apartenența intrării la mulțimea C.
Pe programul care rezolvă $B \oplus C$ îl vom numi $PBoC$. Atfel, putem să definim PB:

```
int PB(int x) {
    if PBoC(x) return not PC(x);
    else return PC(x);
}
```

Acest program efectuează în esență: $(B \oplus C) \oplus C = B$. Din moment ce se termină și calculează corect apartenența la B, rezultă că B este recursivă, dar deja am stabilit că B nu este nici măcar recursiv-enumerabilă. Contradicție.
Rezultă că C este nerecursivă.

6 Exerciții

1. Fie mulțimile A, B, C disjuncte ($A \cap B = \emptyset$, $B \cap C = \emptyset$, $A \cap C = \emptyset$), și complementare ($A \cup B \cup C = \mathbb{N}$). Demonstrați că dacă A, B și C sunt recursiv-enumerabile, atunci acestea sunt și recursive.

2. Fie mulțimea A recursivă și mulțimea B nerecursivă dar recursiv-enumerabilă. Este mulțimea $A \cup B$ recursiv-enumerabilă?

3. *Bonus: De la exercițiul anterior, demonstrați prin exemple că mulțimea $A \cup B$ poate fi și recursivă și nerecursivă.

4. Mulțimea A este recursiv-enumerabilă dar nerecursivă. Ce putem spune despre mulțimea $B = N \setminus A$?

5. Fie mulțimea A nici măcar recursiv-enumerabilă, și mulțimea $B \subset A$ astfel încât $A \setminus B$ este recursivă. Ce putem spune despre B?

6. *Bonus: Fie mulțimea A recursiv enumerabilă, și mulțimea $B = \{x + 1 | x \in A\}$. Știind că $A \cap B = \emptyset$, demonstrați că dacă $A \cup B$ recursivă atunci și A și B sunt recursive.

7. *Bonus: Fie mulțimea $A \subset N$ care nu are nicio pereche de elemente consecutive și mulțimea $B = \{x + 1 | x \in A\}$. Demonstrați că dacă $A \cup B$ recursivă atunci și A și B sunt recursive.

7 Observații

1. Mulțimea șirurile unui alfabet finit este recursiv-enumerabilă din moment ce putem ordona șirurile după mărime crescător și lexicografic. Spre exemplu mulțimea șirurilor cu caracter din alfabetul a, b poate fi ordonată ["", "a", "b", "aa", "ab", "ba", "bb", "aaa"...].
Pentru orice mulțime recursiv enumerabilă există o metodă efectivă de a trece de la elemente din ea la indicele lor, aferent ordonării stabilite, și invers, de la indice la element. Deci practic în rezolvări când ne folosim de intrare ca de un număr natural, noi facem implicit conversia spre indicele ei. Când avem nevoie de intrarea problemei, facem implicit conversia înapoi.

2. Teza Church-Turing nu este demonstrată decât pe diverse formalizări ale ideii de funcție efectiv calculabilă, acesta în sine fiind un termen informal, care nu poate fi introdus într-o demonstrație formală. Totuși, accepțiunea comunității este că teza este adevărată.

References

[1] Cristian A. Giumale. *Introducere în Analiza Algoritmilor*. Polirom, 2004.

[2] Neil Immerman. Computability and complexity. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2018 edition, 2018.