Algoritmi de căutare a unui arbore normal într-un circuit electric

-Sesiunea de Comunicări Ştiinţifice organizată de Facultatea de Inginerie Electrică, Departamentul de Electrotehnică, 15 mai 2015

Radu-Dumitru Stochiţoiu
Denisa Sandu
Darius Florentin Neaţu
Răzvan Chiţu
Silvia Pripoae

Seria I CA, Facultatea de Automatică si Calculatoare,
Universitatea Politehnica Bucureşti

Indrumatori: Prof.dr.ing. Gabriela Ciuprina, Drd.ing. Ruxandra Bărbulescu

Abstract Studiul vizează validarea unui circuit electric prin găsirea unui arbore normal. Deoarece alcătuirea unui circuit care să admită un arbore ce cuprinde toate nodurile, format numai din surse ideale de tensiune şi rezistori, este dificilă în cazul în care numărul de laturi este mare, am decis realizarea unor programe care, parsând un netlist de tip LTSpice/PSpice, validează circuitul, în cazul în care este posibilă găsirea unui arbore normal, sau afişează mesaje de eroare în caz contrar.
Lucrarea analizează implementarea unui algoritm optim, atât din punct de vedere al timpului de execuţie, cât şi al memoriei utilizate, pe baza unor teste generate ce cuprind toate cazurile posibile. Acestea demonstrează comportamentul algoritmilor atât pe cazuri mici, cât şi pe cazuri cu sute de mii de laturi, le verifică din punct de vedere al corectitudinii şi le gasesc situaţii optime de funcţionare.

”Nature may reach the same result in many ways.”
Nikola Tesla

Cuprins

1 Parsarea unui Netlist

Programul nostru citeşte un Netlist LTSpice/PSpice dat ca parametru. Prima linie conţine numărul de noduri din circuit si numarul de laturi (drept comentariu). Am decis să citim numarul de noduri din prima linie pentru a putea realiza statistici asupra modalităţilor de lucru. Pentru o linie citită intr-un buffer se utilizează funcţia „strtok", obţinând astfel toate cuvintele delimitate de caracterul spaţiu.

Reţinerea datelor în memorie se face folosind o structură de date, numită sugestiv "Muchie", care reţine tipul unei muchii (SIT, SIC sau R), numele acesteia, precum şi două noduri, reprezentând extremităţile muchiei si valoarea atribuită acesteia.

Citirea se realizează linie cu linie pâna la sfârşitul fişierului. Se ignoră ultimele linii (liniile de tip comandă). Pe fiecare linie se vor afla numele muchiei - implicit şi tipul acesteia (R - rezistenţă, V - sursă ideală de tensiune, I - sursă ideală de curent) ca primă literă a numelui, nodul iniţial, nodul final şi valoarea atribuită muchiei. In funcţie de tipul muchiei, aceasta se va introduce într-unul din vectorii alocaţi dinamic specifici: sic, sit sau r.

2 Mod de abordare. Algoritmi

Pentru găsirea unui arbore normal, am decis să folosim trei algoritmi [6] şi să le comparăm rezultatele. Un arbore normal are urmatoarele proprietăţi:

- are N-1 laturi, N fiind numărul de noduri, astfel asigurând faptul că nu există cicluri;
- conţine toate SIT;
- poate conţine şi rezistenţe;
- nu conţine nicio SIC.

Iniţial, fiecare algoritm va verifica dacă numărul de SIT este mai mare decât N-1, deoarece astfel un arbore normal nu ar putea acoperi toate muchiile de acest tip.

2.1 Parcurgerea in adâncime

Acest algoritm incearcă să gasească un arbore normal adaugând pe rând muchii, astfel încât proprietatea de graf conex aciclic să ramână satisfăcută.

Iniţial, adaugăm la arbore toate muchiile de tip SIT (includerea acestora fiind necesară). Dacă acestea sunt în număr de $N - 1$, iar graful obţinut este într-adevăr un arbore, atunci toate nodurile sunt incluse în mulţimea capetelor acestor muchii. Acest lucru este verificat folosind o parcurgere în adâncime (DFS).

Dacă toate nodurile sunt vizitate ca urmare a unei astfel de parcurgeri, pornind dintr-un nod oarecare, laturile care conţin SIT-uri formează un arbore normal.

În caz contrar, continuăm să adăugăm laturi de tip R. După fiecare adăugare, verificăm dacă muchia adaugată completeaza un ciclu, adică ambele capete ale acesteia sunt in aceeaşi componentă conexă, în graful format din muchiile adăugate până atunci. Acest lucru se verifică tot folosind o parcurgere in adâncime.

Dacă am reuşit să adăugam $N - 1$ muchii, înseamnă că am obţinut un arbore normal.

Complexitatea acestui algoritm este $O(N * (N + L))$, deoarece pentru fiecare muchie adaugată, efectuăm o parcurgere in adâncime (în $O(N + L)$).

Memoria folosită este de ordinul $O(N + L)$, vecinii nodurilor fiind reprezentaţi prin liste de adiacenţă, nu prin matrice de adiacenţă, care ar fi avut nevoie de o complexitate $O(N^2)$.

2.2 Algoritmul lui Prim

Algoritmul lui Prim [1] găseşte arborele parţial de cost minim al unui graf conex ponderat, dar poate fi particularizat astfel incât muchiile de tip SIT au costul cel mai mic (ex. 0), muchiile de tip R au costul mediu (ex. 1), iar muchiile de tip SIC au costul cel mai mare (ex. 2), sau nu le luam in considerare.

Ştiind că arborele normal va conţine toate muchiile de tip SIT, putem considera orice muchie de acest tip ca fiind arborele iniţial, un arbore cu 2 noduri si o muchie. În continuare, vom căuta $N - 2$ muchii care să satisfacă condiţiile de apartenenţă la arborele normal.

Având un arbore curent, o muchie indeplineşte condiţiile de apartenenţă la acesta dacă şi numai dacă unul dintre nodurile sale este deja inclus în arbore, iar celălalt nu. Dacă ar fi ambele incluse deja, s-ar forma un ciclu, iar dacă nu ar fi niciunul, nu ar deveni un graf conex.

În cele $N - 2$ iteraţii vom căuta de fiecare dată printre muchiile de tip SIT (cu costul cel mai mic) dacă există una care să satisfacă condiţiile de apartenenţă la arbore, iar apoi printre muchiile de tip R (cu costul mediu). Daca nu se găseşte nicio muchie de tip SIT sau R,

şi este nevoie să căutam printre cele de tip SIC inseamnă că graful nu conţine un arbore normal, şi astfel nu este bine condiţionat.

Complexitatea acestui algoritm este $O(N * L)$, dar se comportă mult mai bine in practică, aproape $O(N * log(L))$, deoarece face $N - 2$ iteraţii, şi la fiecare iteraţie cazul cel mai defavorabil va face L iteraţii parcurgînd toate muchiile.

Memoria este, însă, un punct forte, deoarece folosim un tablou unidimensional de tip boolean (true sau false, 1 sau 0) de N elemente, pentru a ţine minte dacă un nod este sau nu înăuntrul arborelui normal, şi cei trei vectori de muchii, sit, sic si r. Rezultă memorie $O(N + L)$.

2.3 Algoritmul lui Kruskal

Algoritmul lui Kruskal [4] este de asemenea utilizat pentru găsirea unui arbore parţial de cost minim şi este şi cel mai eficient din punct de vedere al timpului de execuţie.

Acest algoritm încearcă să găsească un arbore de cost minim de acoperire, care să fie şi arbore normal. Prin urmare ar trebui sa găsim un APM (arbore parţial de cost minim), folosind toate muchiile din prima categorie (muchiile de tip SIT) şi oricâte, eventual niciuna, din a doua categorie (muchiile de tip R).

Pentru a găsi arborele normal această metodă foloseşte structura de păduri de mulţimi disjuncte. Astfel, se pleacă cu N mulţimi, mulţimea cu numărul i conţinând iniţial doar nodul i.

Se încearcă adăugarea tuturor muchiilor de tip SIT (cost 0). Prin adăugarea unei muchii se înţelege reuniunea mulţimilor din care fac parte cele două extremităţi. Daca nodurile fac parte din aceeaşi mulţime, atunci prin adăugarea unei noi muchii se va crea un ciclu, deci nu există nici un arbore care să înglobeze toate laturile SIT. Algoritmul se opreşte.

Daca s-au adăugat toate laturile SIT, se verifică dacă acestea erau in numar de $N - 1$. În caz afirmativ, arborele normal a fost găsit. Dacă nu s-a adăugat un numar de $N - 1$ laturi, atunci se completează cu laturi de tip R, având grijă să nu se creeze cicluri.

Dacă s-au epuizat laturile de tip R, verificăm dacă s-au adăugat in total $N - 1$ laturi. În caz afirmativ arborele normal a fost găsit.

Laturile de tip SIC nu vor fi niciodată adăugate.

O mulţime a fost reprezentată sub forma unui arbore, unde radăcina arborelui este reprezentantul acestei mulţimi. Structura de păduri de mulţimi disjuncte a fost optimizată folosind două euristici.

1) **Reuniunea dupa rang**. Pentru fiecare mulţime ţinem minte inălţimea arborelui care reprezintă acea mulţime. Atunci când vrem să unim 2 arbori, îl unim pe cel cu rang mai mic de cel cu rang mai mare.
2) **Compresia drumurilor**. Atunci când facem o interogare, după ce am aflat în ce mulţime se află nodul x, mai parcurgem o data drumul de la x la radacină şi unim toate nodurile direct de radacină. Astfel, data viitoare când vom avea o interogare pentru unul din aceste noduri, vom ajunge într-un singur pas la radacină. Este posibil ca atunci când facem compresia drumurilor înălţimea arborelui să se modifice, însă nu actualizam rangul (în cazul în care este folosită şi prima euristică). În acest caz, acel rang va deveni practic o limită superioară a înălţimii arborelui.

Aceste 2 euristici duc complexitatea finală la $O(log * N)$ – log „star" N pentru o operaţie de consultare a mulţimii din care face parte un nod şi $O(1)$ pentru o operaţie de reuniune a două mulţimi disjuncte. Complexitatea finală în timp a algoritmului devine $O(Llog * N)$ – log „star" N.

$log * N$ ("log star de N"), reprezintă inversa funcţiei lui Ackermann si poate fi aproximat cu $O(1)$.

Întrucât un Netlist LTSpice poate conţine în orice ordine laturile circuitului, prima problemă care apare este sortarea muchiilor. Se observă însă că avem doar 3 tipuri de categorii de muchii, dintre care am dori să le folosim pe cele de tip SIT înaintea celor de tip R. Putem astfel să utilizăm 3 containeri numiţi sit, r, sic iar atunci când citim o muchie să o plasăm în vectorul corespunzător.

Astfel, procesul de sortare este realizat la citire - $O(L)$.

Memoria folosită este de ordinul $O(N + L)$.

3 Testare

Verificarea algoritmilor a fost realizată prin alcătuirea unor teste ce ilustrează comportamentul celor 3 algoritmi. Testele au fost gândite astfel incât să pună în evidenţă diferenţele apărute între algoritmi din punct de vedere al timpului de execuţie, în condiţii de dimensiuni aproape egale de memorie utilizată. Astfel, am testat eficienţa algoritmilor în două

etape. În prima etapă am folosit circuite predefinite, concepute astfel incât să fie bine formulate, subliniind posibilitatea unui algoritm de a ajunge într-un timp util la rezultatul dorit.

Întrucât timpul de execuţie al algoritmului DFS (Depth-First Search) creşte cel mai mult în raport cu numărul de noduri / muchii, pentru el s-au realizat cele mai puţine teste. În mod similar, nici algoritmul Prim nu se execută într-un timp util pe circuite cu peste 300 000 de laturi. Algoritmul Kruskal este singurul care a suportat măsurarea într-un timp util a celor doi parametri urmăriţi pe toate testele.

Cea de-a doua etapă de testare presupune probarea algoritmilor pe circuite generate aleatoriu. Întrucat prima etapă de testare este concludentă în ceea ce priveşte memoria folosită, în aceste teste este măsurat doar timpul de rulare.

Din aceste teste se observă clar faptul că, la aproximativ aceeaşi dimensiune a memoriei folosite (acelaşi număr de muchii), eficienţa algoritmilor DFS si Prim scade drastic odată cu mărirea considerabilă a numărului de muchii. Daca pentru testele de mici dimensiuni cei trei algoritmi se comportă aproape identic, pe ultimul circuit, de cca. 127 000 de laturi, algoritmul DFS se execută in 287 de secunde, de 4000 de ori mai încet decât algoritmul Kruskal. Algoritmul Prim are un timp de rulare acceptabil, dar tot foarte mare în comparaţie cu cel al algoritmului Kruskal.

În concluzie, dacă în cazurile de dimensiuni mici nu este foarte important algoritmul folosit pentru validarea unui circuit, odată cu creşterea numărului de noduri şi de laturi, alegerea metodei devine un factor extrem de important.

4 Concluzii

Găsirea unui arbore normal este o condiţie necesară şi suficientă în cazul circuitelor de curent continuu în care rezistenţele sunt pozitive. Dacă nu există un arbore normal, atunci circuitul are surse în exces, şi nu modelează în mod corect realitatea, pentru că el conduce la o problemă matematică fără soluţii sau cu o infinitate de soluţii. În cazul unor circuite cu rezistenţe negative, atunci existenţa unui arbore normal este o condiţie necesară pentru buna formulare, dar nu suficientă.

Algoritmul Kruskal s-a dovedit a fi cel mai eficient în găsirea unui arbore normal. Implementarea lui este utilă deoarece uneori simulatoarele pot să nu dea mesaje de avertizare în cazul unor circuite prost formulate.

5 Bibliografie

[1] http://en.wikipedia.org/wiki/Prim%27s_algorithm

[2] http://en.wikipedia.org/wiki/Disjoint-set_data_structure

[3] http://en.wikipedia.org/wiki/Iterated_logarithm

[4] http://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[5] http://www.infoarena.ro/problema/apm

[6] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. *Introduction to Algorithms*(3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4

1. Anexă

Sursele care implementează cei trei algoritmi au fost scrise in limbajul C++ pentru a obţine o performanţă ridicată.

Implementarea a fost facută sub Linux. Dupa complilare, trebuie dat ca parametru numele netlist-ului, inclusiv extensia ".cir" , din directorul curent, asupra căruia dorim să facem verificările.

6.1. Muchie.h

```
#ifndef __MUCHIE__H
#define __MUCHIE__H

#include <string>

/* Tipuri folosite */
#define R 'R'
#define SIC 'I'
#define SIT 'V'

struct Muchie{
  /* Tipul unei muchii din circuit. R, V, I */
  char tip;

  /* Numele elementului de circuit. Ex. R1, V2 etc. */
  std::string nume;

  /* Extremitatile muchiei. */
  int nodInitial, nodFinal;

  /* Valoarea parametrului in S.I. */
  double valoare;
};

#endif /*__MUCHIE__H */
```

6.2. parser.h

```
#ifndef __PARSER__H__
#define __PARSER__H__

#include <fstream>
#include <iostream>
#include <cstdlib>
#include <algorithm>
#include <cstdio>
#include <cctype>
#include <cstring>
#include <vector>
#include "Muchie.h"

#define MAX_LINE 100

int N, L;
std::vector < Muchie > sic, sit, r;
char linie[MAX_LINE];
Muchie m;
bool existaRezistenteNegative = false;

void ReadInput(char *file_name) {
  FILE *f;
  f = fopen(file_name, "r");

  fgets(linie, MAX_LINE, f);

  int i = 0;
  for (; i < strlen(linie) && !isdigit(linie[i]); ++i);

  for (; i < strlen(linie) && isdigit(linie[i]); ++i)
    N = 10 * N + (linie[i] - '0');

  while(fgets(linie, MAX_LINE, f))
  {
    if(linie[0] == '.') break;
    char *p = strtok(linie, " ");
    m.tip = *p;
    m.nume = p;
    p = strtok(NULL, " ");
    m.nodInitial = atoi(p);
    p = strtok(NULL, " ");
    m.nodFinal = atoi(p);
    p = strtok(NULL, " ");
    m.valoare = atof(p);

    if(m.tip == SIT)
      sit.push_back(m);
    else if(m.tip == SIC)
      sic.push_back(m);
    else {
      r.push_back(m);
      if (m.valoare < 0) existaRezistenteNegative = true;
    }
  }

  L = sit.size() + r.size() + sic.size();

  fclose(f);
}
```

```
#endif /*__PARSER__H__*/
```

6.3. DFS.h

```
void Parcurgere( int nod, std::vector<int> *vecini, bool *vizitat ) {
      int i;

    vizitat[nod] = true;

    //Parcurgem toti vecinii, marcandu-i vizitati
    for ( i = 0; i < vecini[nod].size(); ++ i )
        if ( vizitat[vecini[nod][i]] == false )
            Parcurgere( vecini[nod][i], vecini, vizitat );
}

void DFS()
{
    std::vector<int> *vecini = new std::vector<int>[N];
    bool *vizitat = new bool[N]();
      bool existaArbore = false;
    int muchii_incluse = 0;
      int i;

    //Adaugam toate SIT la arbore
      for ( i = 0; i < sit.size(); ++ i ) {
        vecini[sit[i].nodInitial].push_back(sit[i].nodFinal);
        vecini[sit[i].nodFinal].push_back(sit[i].nodInitial);

        ++ muchii_incluse;
    }

    //Verificam daca toate nodurile sunt vizitate
    Parcurgere( 0, vecini, vizitat );
    existaArbore = true;
    for ( i = 0; i < N; ++ i )
        if ( vizitat[i] == false ) {
            existaArbore = false;
            break;
        }

    //Daca arborele contine N-1 muchii, dar nu toate varfurile, s-a format un
ciclu
    if ( existaArbore == false && muchii_incluse == (N - 1) ) {
        std::cout << "Laturile de tip SIT formeaza cel putin un ciclu.\n";
        std::cout << "Circuitul nu admite arbore normal.\n";
        std::cout << "Circuitul nu este bine formulat.\n";

        delete[] vecini;
        delete[] vizitat;
        return;
    }

    //Am gasit arborele normal, format din SIT
    if ( existaArbore == true ) {
        std::cout << "Laturile de tip SIT formeaza un arbore normal.\n";

        if ( existaRezistenteNegative ) {
            std::cout << "Atentie! Exista rezistente negative in circuit.\n";
        } else {
            std::cout << "Circuitul este bine formulat.\n";
        }

        delete[] vecini;
        delete[] vizitat;
```

```
            return;
        }

        //Adaugam muchii de tip R
        for ( i = 0; i < r.size(); ++ i ) {
            //Verificam daca adaugarea muchiei curente ar genera cicluri
            std::memset( vizitat, 0, N );
            Parcurgere( r[i].nodInitial, vecini, vizitat );

            //Daca nu, o adaugam la arbore
            if ( vizitat[r[i].nodFinal] == false ) {
                vecini[r[i].nodInitial].push_back( r[i].nodFinal );;
                vecini[r[i].nodFinal].push_back( r[i].nodInitial );

                ++ muchii_incluse;
            }

            //Arborele genrat contine toate varfurile
            if ( muchii_incluse == (N - 1) ) {
                std::cout << "Laturile de tip SIT si R formeaza un arbore
normal.\n";

                if ( existaRezistenteNegative ) {
                    std::cout << "Atentie! Exista rezistente negative in
circuit.\n";
                } else {
                    std::cout << "Circuitul este bine formulat.\n";
                }

                delete[] vecini;
                delete[] vizitat;
                return;
            }
          }

        //Am terminat parcurgerea fara a gasi arborele normal
        std::cout << "Nu se poate forma un arbore normal doar din SIT si
rezistente.\n";
        std::cout << "Circuitul nu admite arbore normal.\n";
        std::cout << "Circuitul nu este bine formulat.\n";

        delete[] vizitat;
        delete[] vecini;
}
```

6.4. Prim.h

```
void Prim()
{
  bool *vizitat = new bool[N];
  int muchie_gasita, i, j, nod1, nod2, nr_sit_folosite, folosescRezistente =
0;

  // Initial, niciun nod nu apartine arborelui cautat
  for (i = 0; i < N; ++i) vizitat[i] = 0;

  // Adaugam prima SIT in arbore daca exista
  if(sit.size() > 0)
  {
       vizitat[sit[0].nodInitial] = 1;
       vizitat[sit[0].nodFinal] = 1;
       // Contorizam numarul de SIT introduse in arbore
       nr_sit_folosite = 1;
  }
  else if(r.size() > 0)
  {
       vizitat[r[0].nodInitial] = 1;
       vizitat[r[0].nodFinal] = 1;
  }
  else
  {
       std::cout << "Nu se poate forma un arbore normal doar din SIT si
rezistente.\n";
    std::cout << "Circuitul nu admite arbore normal.\n";
    std::cout << "Circuitul nu este bine formulat.\n";
    delete[] vizitat;
    return;
  }

  // Cautam N-2 laturi, deoarece pe prima am adaugat-o deja
  for(i = 1; i < N - 1; ++i)
  {
       // Consideram ca nu am gasit muchia inca
    muchie_gasita = 0;

    // Cat timp nu am gasit muchie cautam prin SIT-uri
    for(j = 0; j < sit.size() && !muchie_gasita; ++j)
    {
      nod1 = sit[j].nodInitial;
      nod2 = sit[j].nodFinal;

      // Daca o extremitate a muchiei apartine arborelui si cealalta nu,
atunci
      // putem adauga si aceasta muchie la arbore, fara sa devina un graf
ciclic
      if(vizitat[nod1] != vizitat[nod2])
      {
       // Am gasit muchie si cele doua extremitati se afla in arbore
        muchie_gasita = 1;
        vizitat[nod1] = 1;
        vizitat[nod2] = 1;
        // Numarul de SIT folosite este incrementat
        nr_sit_folosite ++;
      }
    }
    // Daca nu am gasit muchie printre SIT, cat timp nu am gasit muchie
cautam
```

```
    // printre rezistente
    for(j = 0; j < r.size() && !muchie_gasita; ++j)
    {
      nod1 = r[j].nodInitial;
      nod2 = r[j].nodFinal;

      // Daca o extremitate a muchiei apartine arborelui si cealalta nu,
atunci
      // putem adauga si aceasta muchie la arbore, fara sa devina un graf
ciclic
      if(vizitat[nod1] != vizitat[nod2])
      {
       // Includem cel putin o rezistenta in arbore
        folosescRezistente = 1;
        // Am gasit muchie si cele doua doua extremitati se afla in arbore
        muchie_gasita = 1;
        vizitat[nod1] = 1;
        vizitat[nod2] = 1;
      }
    }

    // Daca nu se poate forma un arbore numai cu SIT si rezistente
    if(!muchie_gasita)
    {
      std::cout << "Nu se poate forma un arbore normal doar din SIT si
rezistente.\n";
      std::cout << "Circuitul nu admite arbore normal.\n";
      std::cout << "Circuitul nu este bine formulat.\n";
      delete[] vizitat;
      return;
    }
  }

  // Daca nu sunt folosite toate SIT
  if(nr_sit_folosite != sit.size())
  {
    std::cout << "Laturile de tip SIT formeaza cel putin un ciclu.\n";
    std::cout << "Circuitul nu admite arbore normal.\n";
    std::cout << "Circuitul nu este bine formulat.\n";
  }
  else
  {
      // Daca nu folosesc rezistente
    if(!folosescRezistente)
      std::cout << "Laturile de tip SIT formeaza un arbore normal.\n";
    else
      std::cout << "Laturile de tip SIT si R formeaza un arbore normal.\n";

      // Daca nu exista rezistente negative in circuit
    if(!existaRezistenteNegative)
      std::cout << "Circuitul este bine formulat.\n";
    else
      std::cout << "Atentie! Exista rezistente negative in circuit.\n";
  }

  delete[] vizitat;
}
```

6.5. Kruskal.h

```
/* Vectorul de tati asociat padurilor de multimi disjuncte. */
std::vector<int> Tata, rang;

int multime(int& nod) {
  if(Tata[nod] != nod) {
    Tata[nod] = multime(Tata[nod]);
  }
  return Tata[nod];
}


void reuneste(int& nod1, int& nod2) {
  /* Preiau multimile din care fac parte cele doua numere. */
  int m1 = multime(nod1), m2 = multime(nod2);

  /* Unesc multimea cu rang mai mic de cea cu rang mai mare */
  if (rang[m1] > rang[m2]) Tata[m2] = m1;
    else Tata[m2] = m1;

  /* In caz ca rangurile erau egale atunci cresc rangul noii multimi cu 1 */
  if (rang[m1] == rang[m2]) rang[m2]++;
}

void Kruskal() {

  /* Laturi = numarul curent de laturile al arborelui. */
  int nod1, nod2, Laturi = 0;

  /* Initial sunt N multimi. Fiecare are un singur element.
    Tatal subarborelui i este chiar i. Tata[i] = i. */

  for (int i = 0; i < N; ++i) {
    Tata.push_back(i);
    rang.push_back(1);
  }

  /* Un arbore normal inglobeaza toate laturile cu SIT. */
  for (int i = 0; i < sit.size(); i++) {
    nod1 = sit[i].nodInitial;
    nod2 = sit[i].nodFinal;

    /* Verific daca cele doua extremitati faca parte din componente
      conexe diferite. */
    if (multime(nod1) != multime(nod2)) {
      /* Afirmativ. Prin adaugarea muchiei se reunesc 2 componente conexe.
         NU se face ciclu. */
      reuneste(nod1, nod2);
      Laturi++;

    } else {
      std::cout << "Laturile de tip SIT formeaza cel putin un ciclu.\n";
      std::cout << "Circuitul nu admite arbore normal.\n";
      std::cout << "Circuitul nu este bine formulat.\n";
      return;
    }
  }
```

```
  /* S-a creat o padure de arbori formati doar cu laturi de tip SIT. */

  /* Este posibil ca toate laturile de tip SIT sa formeze un singur arbore.
*/
  if (Laturi == N-1) {
    std::cout << "Laturile de tip SIT formeaza un arbore normal.\n";
    if (existaRezistenteNegative) {
      std::cout << "Atentie! Exista rezistente negative in circuit.\n";
    } else {
      std::cout << "Circuitul este bine formulat.\n";
    }
    return;
  }

  /* Putem reuni arborii din padure cu laturi de tip R. */
  for (int i = 0; i < r.size(); i++) {
    nod1 = r[i].nodInitial;
    nod2 = r[i].nodFinal;

    /* Verific daca cele doua extremitati faca parte din componente
      conexe diferite. */
    if (multime(nod1) != multime(nod2)) {
      /* Afirmativ. Prin adaugarea muchiei se reunesc 2 componente conexe.
         NU se face ciclu. */
      reuneste(nod1, nod2);
      Laturi++;

      /* Verific daca am completat laturile arborelui. */
      if (Laturi == N - 1) {
        break;
      }
    }
  }

  /* Verific daca am completat laturile arborelui. */
  if (Laturi == N - 1) {
    std::cout << "Laturile de tip SIT si R formeaza un arbore normal.\n";
    if (existaRezistenteNegative) {
      std::cout << "Atentie! Exista rezistente negative in circuit.\n";
    } else {
      std::cout << "Circuitul este bine formulat.\n";
    }
    return;
  }

  /* NU s-a putut realiza un arbore folosind doar laturile de tip SIT si R.
    Prin urmare trebuie sa folosesc cel putin o muchie de tip SIC.
    Din definitia arborelui rezulta ca circuitul nu admite un arbore normal.
*/
  std::cout << "Nu se poate forma un arbore normal doar din SIT si
rezistente.\n";
  std::cout << "Circuitul nu admite arbore normal.\n";
  std::cout << "Circuitul nu este bine formulat.\n";
}
```

6.5. admiteArboreNormal.cpp

```
#include "Muchie.h"
#include "parser.h"
#include "Kruskal.h"
#include "Prim.h"

int main(int argc, char *argv[])
{
  ReadInput(argv[1]);

  if(sit.size() > N - 1)
  {
    printf("Numarul de SIT este mai mare decat %d, deci SIT formeaza
ciclu.\n", N-1);
    printf("Circuitul nu este bine formulat.\n");
    return 0;
  }

  std::cout << "prim\n";
  Prim();

  std::cout << "kruskal\n";
  Kruskal();

  std::cout << "DFS\n";
  DFS();

  return 0;
}
```