Probleme NP-complete

Capitolul 13

- ❖ **Preliminarii**
- ❖ **Scurtarea timpului de execuție**
- ❖ **Ordinea explorării soluțiilor**
- ❖ **Particularitățile problemelor**
- ❖ **Concluzii**
- ❖ **Rezumat**
- ❖ **Implementări sugerate**
- ❖ **Probleme propuse**
- ❖ **Soluțiile problemelor**

Problemele nedeterminist polinomiale ridică dificultăți destul de mari majorității elevilor care participă la concursurile de programare. În cadrul acestui capitol vom prezenta câteva modalități de abordare a acestora. Aceste abordări pot fi utilizate atât în cazul problemelor NP, cât şi în cazul în care, deşi există algoritmi polinomiali pentru rezolvare, elevul nu îi cunoaşte.

13.1. Preliminarii

Înaintea abordării unei probleme *NP* este necesară o analiză atentă. Anumite probleme se pretează mai bine anumitor tipuri de rezolvări.

Vom aminti acum cele mai uzuale metode de rezolvare, fără a intra în detalii. În cadrul acestui capitol vom prezenta câteva dintre ele, urmând ca în capitolele următoare să prezentăm şi altele. Aşadar, cele mai importante modalități de abordare pentru probleme *NP* sunt:

- explorarea în lățime a spațiului soluțiilor;
- explorarea în adâncime a spațiului soluțiilor;
- metode euristice;
- metode bazate pe nedeterminism;
- algoritmi genetici;
- algoritmi pseudo-polinomiali;
- scheme de aproximare;
- rețele neuronale.

Dintre metodele amintite mai sus, primele trei ar trebui să fie cunoscute, motiv pentru care le vom prezenta doar pe scurt.

Metodele de rezolvare pot fi combinate. De exemplu, înaintea aplicării metodei backtracking se poate căuta o soluție printr-o metodă euristică; rezultatul obținut va fi folosit pentru evitarea explorării unor cazuri neinteresante.

13.2. Scurtarea timpului de execuție

În cadrul acestei secțiuni vom prezenta câteva modalități de micşorare a duratei de execuție a unei implementări care utilizează un algoritm exponențial.

13.2.1. Simplificarea problemei

De multe ori, problema poate fi simplificată înaintea aplicării algoritmului de rezolvare. După obținerea soluției pentru varianta simplificată, soluția pentru întreaga problemă se obține foarte rapid, printr-o metodă polinomială.

De exemplu, în cazul problemei colorării nodurilor unui graf cu K culori, există o simplificare dată de următoarea propoziție: "*Problema pentru un graf G în care există un nod P cu gradul mai mic decât K se poate rezolva uşor dacă rezolvăm problema pentru graful G \ {P}.*"

Demonstrația este evidentă. Pentru a obține soluția pentru graful G se obține soluția pentru graful $G \setminus \{P\}$, după care se colorează nodul P cu o culoare care nu a fost atribuită nici unui vecin. Datorită faptului că numărul culorilor este mai mare decât gradul nodului P, colorarea este posibilă.

Astfel, paşii algoritmului de rezolvare sunt următorii:

- se elimină din graf toate nodurile care au gradul mai mic decât K;
- se repetă pasul anterior până când nu mai există în graf noduri cu gradul mai mic decât K (după eliminările de la primul pas, gradele altor noduri pot deveni mai mici decât K);
- se rezolvă problema pentru graful rămas (cu alte cuvinte, pentru problema particulară în care gradele tuturor nodurilor sunt cel puțin egale cu K); aceasta nu mai poate fi simplificată prin aceeaşi metodă.
- se construieşte soluția problemei inițiale, colorând nodurile eliminate în ordinea inversă eliminării lor.

13.2.2. Eliminarea soluțiilor neinteresante

Majoritatea abordărilor prezentate pot fi îmbunătățite folosindu-se o tehnică de eliminare a unor elemente din spațiul soluțiilor (engl. *pruning*).

Eliminarea poate fi realizată euristic (se estimează, rapid, dacă elementul curent "promite" să conducă la o soluție bună şi, dacă nu, este eliminat), dar există şi probleme pentru care eliminarea este sigură.

De exemplu, în cazul unor probleme de minim, poate există o funcție care aproximează "costul minim" al transformării configurației curente în soluție a problemei. Această funcție este numită euristică optimistă. Dacă suma dintre costul configurației şi funcția aceasta este mai mare decât costul celei mai bune soluții obținute anterior în cadrul algoritmului, configurația curentă poate fi eliminată, deoarece nu va genera o soluție mai bună.

Cu cât funcția aproximează mai bine costul transformării, cu atât algoritmul este mai performant, deoarece elementele neperformante sunt excluse, împreună cu toate elementele (tot neperformante) care ar fi fost introduse de ele.

13.3. Ordinea explorării soluțiilor

Metodele bazate pe explorarea în lățime a grafului configurațiilor şi cele bazate pe explorarea în adâncime utilizează deseori o anumită ordine în care sunt explorate soluțiile. Problema pe care se va baza expunerea este cea a jocului Perspico.

13.3.1. Jocul Perspico

Vom prezenta acum problema celebrului joc Perspico, apărută sub diverse forme la multe concursuri de programare. Pentru cei care nu au avut plăcerea de a juca acest joc, prezentăm enunțul problemei:

Se consideră o matrice cu patru linii şi patru coloane care conține toate numerele cuprinse între 0 şi 15. Folosind un număr minim de mutări, trebuie să se obțină configurația:

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
```

Mutările permise sunt interschimbări ale elementului 0 cu unul dintre elementele de pe pozițiile vecine pe orizontală şi verticală.

De exemplu, din configurația:

```
 7 14  9  3
 6  1  2  8
15  0 11 12
13  4  5 10
```

se poate ajunge în una dintre următoarele patru configurații:

```
 7 14  9  3      7 14  9  3      7 14  9  3      7 14  9  3
 6  0  2  8      6  1  2  8      6  1  2  8      6  1  2  8
15  1 11 12      0 15 11 12     15  4 11 12     15 11  0 12
13  4  5 10     13  4  5 10     13  0  5 10     13  4  5 10
```

Evident, nu vom avea patru variante în toate situațiile; dacă valoarea 0 se află într-un colț, vom avea doar două posibilități de continuare, iar dacă se află pe una din laturi (dar nu în colț), vom avea trei.

13.3.2. Explorarea în lățime

În cazul explorării în lățime, configurațiile se introduc într-o coadă. O aceeaşi configurație nu va fi introdusă de două ori. Pentru aceasta se folosesc structuri de date avansate care permit căutarea rapidă, cum ar fi tabelele de dispersie. Dacă numărul configurațiilor este redus, se poate folosi o structură de selecție simplă (o mulțime sau un vector de valori logice). Principalul dezavantaj constă în depăşirea rapidă a capacității memoriei disponibile, dacă spațiul configurațiilor este mare.

13.3.3. Explorarea în adâncime

În cazul explorării în adâncime, metoda cea mai des folosită este backtracking. Practic, se foloseşte o stivă. La fiecare pas se introduce în stivă una dintre configurațiile vecine celei din vârful stivei; după ce căutarea pentru aceasta s-a terminat, se va introduce o altă configurație vecină etc. Memoria disponibilă nu este o problemă (structura de bază este o stivă de configurații), în schimb există posibilitatea explorării repetate a aceleiaşi stări. În plus, soluțiile foarte apropiate de configurația de plecare pot fi pierdute.

13.3.4. DFSID

Există o metodă care înlătură neajunsul explorării în adâncime, depăşind şi limitarea de memorie dată de explorarea în lățime. Tehnica se numeşte ***D**epth **F**irst **S**earch with **I**terative **D**eepening* (pe scurt DFSID) şi constă în parcurgerea în adâncime a regiunilor din graful configurațiilor apropiate de configurația de start. Este explorată inițial mulțimea configurațiilor aflate la distanța 1, apoi la distanța 2, 3 etc. față de configurația inițială (aici prin "distanță" înțelegem lungimea drumului minim în graful configurațiilor; pentru jocul *Perspico* aceasta reprezintă numărul de mutări efectuate). Algoritmul se încheie la găsirea unei soluții bune sau la depăşirea timpului acordat. Evident că *DFSID* duce la explorarea repetată a unor noduri ale grafului, dar acesta nu este un neajuns foarte mare, comparativ cu avantajele aduse.

De exemplu, dacă la fiecare introducere în stivă ar exista exact patru opțiuni, atunci *DFSID* ar genera, la pasul K, $4^1 + 4^2 + 4^3 + \ldots + 4^K$ configurații. Numărul de configurații generate la toți paşii precedenți este net inferior (demonstrația se face prin inducție şi este foarte simplă), deci performanța, din punct de vedere al vitezei, scade de mai puțin de două ori față de cea a explorării în lățime.

Tehnica *DFSID* se referă la modificarea adâncimii stivei în *backtracking*, dar există şi alte variante ale acestui stil de abordare. De exemplu, în problema ciclului hamiltonian se poate încerca o abordare în care nu se acceptă soluții cu un cost mai mare decât K. Dacă nu este găsită nici o soluție, se procedează la incrementarea lui K cu un anumit pas şi se reia explorarea.

Toate aceste abordări pot fi îmbunătățite prin tehnicile de *pruning* amintite anterior. Alegerea unei metode trebuie să fie, de obicei, completată de găsirea unor tehnici de eliminare a nodurilor neinteresante.

13.3.5. A*

Există o variantă mai bună a explorării în lățime, şi anume algoritmul A*, folosit pentru problemele de minim. În cazul acestui algoritm, se foloseşte tehnica de *pruning* bazată pe euristica optimistă.

La fiecare pas, sunt introduşi în coadă succesorii celei mai promițătoare configurații. Astfel, pentru fiecare configurație se calculează suma dintre costul ei şi euristica optimistă. Configurația cu această sumă minimă va fi expandată. Algoritmul garantează obținerea soluției optime; sunt necesare structuri de date care permit găsirea eficientă a minimului.

Pentru problema jocului Perspico se va folosi euristica optimistă descrisă anterior. Se observă că ea poate fi calculată foarte uşor la trecerea de la o configurație la alta (practic se schimbă distanța unui singur element din matrice față de poziția sa finală).

În general, un program care implementează algoritmul *A** este mult mai complex decât unul care implementează *DFSID*. În plus, memoria necesară este exponențială pentru cazurile defavorabile. De aceea, cu toate că explorează mai multe configurații, *DFSID* combinat cu *pruning* este mai indicat în multe cazuri.

13.4. Particularitățile problemelor

În cadrul acestei secțiuni vom prezenta câteva probleme pentru care vom identifica particularități care permit găsirea unor algoritmi cu timpi de execuție redus.

În multe cazuri, particularitățile problemelor pot permite optimizări interesante. Vom examina modalități de abordare pentru două probleme *NP*.

13.4.1. Enunțurile

Cele două probleme, foarte asemănătoare pot fi enunțate astfel:

Se consideră un graf neorientat cu cel mult 100 de noduri şi 1000 de muchii.

1) Să se selecteze o submulțime S a mulțimii nodurilor, de cardinal minim, astfel încât fiecare nod care nu este în S să aibă cel puțin un vecin în S (din mulțimea de noduri S să se "vadă" toate nodurile).

2) Să se selecteze o submulțime S a mulțimii nodurilor, de cardinal minim, astfel încât fiecare muchie din graf să aibă cel puțin unul dintre nodurile incidente în S (din mulțimea de noduri S să se "vadă" toate muchiile).

13.4.2. Modalitatea de rezolvare

În majoritatea cazurilor, problema 2) este mai simplă decât 1). Se observă că, dacă un nod *X* nu se află în *S*, toți vecinii lui se vor afla în mod obligatoriu în *S*. În caz contrar, o parte dintre muchiile incidente lui *X* nu vor fi "văzute" din S.

Această observație conduce la o soluție backtracking. Nodurile sunt procesate în ordinea descrescătoare a gradelor; la o iterație este introdus în stivă fie nodul curent

(ceea ce elimină multe muchii, introducând un singur nod în S), fie toți vecinii săi (operație care micşorează mult graful, dar introduce noduri suplimentare în S). Dacă numărul de noduri din S este mai mare decât un optim găsit anterior, căutarea pe varianta curentă este abandonată.

Deşi implementarea este foarte simplă, această abordare poate fi mult îmbunătățită. De exemplu, nodurile de grad 1 nu vor fi selectate în soluția optimă (sau, dacă ar fi, o soluție la fel de bună se obține înlocuindu-le cu vecinii lor), deci pot fi eliminate din graf înainte de începerea căutării şi nodurile adiacente lor vor fi introduse în soluție (ceea ce duce la eliminarea unor muchii "văzute" de aceste noduri şi, eventual, la apariția unor alte noduri de grad 1 etc.). Excepție fac perechile de noduri de grad 1, caz în care un nod din fiecare pereche este eliminat, iar celălalt este selectat automat.

Alte posibile optimizări sunt următoarele:

- la un moment dat, un nod care are toți vecinii în S (introduşi anterior) nu va fi introdus în S pentru că nu aduce nici o îmbunătățire; în plus, se măreşte inutil cardinalul mulțimii S;
- dacă graful nu este conex, problema se va rezolva separat pentru fiecare componentă conexă; optimizarea se poate aplica şi pe parcurs (în momentul pierderii conexității grafului prin eliminarea unor noduri şi muchii);
- se poate folosi o euristică optimistă pentru eliminarea unor variante; de exemplu, dacă în graf au mai rămas N_1 noduri şi M_1 muchii, este necesar ca suma gradelor nodurilor care vor fi introduse în soluție să fie cel puțin M_1; se aleg din cele N_1 noduri rămase nodurile cu gradele cele mai mari (relativ la cele M_1 muchii), până când suma gradelor ajunge să fie cel puțin M_1 (chiar dacă aceste noduri nu "văd" cele M_1 muchii); dacă numărul de noduri introduse, împreună cu numărul de noduri existente, depăşesc optimul obținut anterior, varianta curentă trebuie abandonată.

Presupunând că nu se mai introduc optimizări în căutare, pasul următor îl constituie scrierea unui program care să execute operațiile descrise. Optimizările de cod contează foarte mult, deoarece vor fi examinate mai multe variante şi şansele de a se obține o soluție mai bună cresc.

Problema 1) este un caz particular al problemei acoperirii cu mulțimi (engl. *set covering*), cunoscută ca fiind NP-completă. Astfel, o mulțime este formată dintr-un nod şi din vecinii lui. Acest fapt nu este suficient pentru a demonstra că 1) este *NP*, dar în cazul de față problema 1) este într-adevăr NP, deoarece există posibilitatea de a reduce în timp polinomial problema acoperirii cu mulțimi la problema 1).

Mai mult, particularitățile problemei 1) fac ca anumite optimizări cunoscute pentru *set covering* să funcționeze foarte bine. Din aceste motive putem trata 1) printr-o abordare care rezolvă problema mai generală. Este de reținut faptul că o astfel de abordate (rezolvarea unei probleme mai generale) nu este recomandată în majoritatea cazurilor, deoarece se pierde informația specifică problemei.

În continuare vom discuta problema ***acoperirii cu mulțimi***. Începem cu două observații fundamentale:

- dacă o mulțime este inclusă în alta, ea nu va intra în soluția finală;
- dacă un element din mulțimea {1, 2, …, *N*} este conținut într-o singură mulțime, aceasta va face parte din soluția finală.

În plus, intuitiv este avantajos să dăm şanse mai mari mulțimilor cu multe elemente să facă parte din soluția finală. Din nefericire, algoritmul *greedy* care ar rezulta din această ultimă observație nu furnizează întotdeauna soluția optimă.

Observațiile conduc la o rezolvare backtracking, care poate fi implementată recursiv. Aceasta funcționează astfel:

- pasul "elimină": se elimină toate mulțimile care sunt incluse în alte mulțimi;
- pasul "găseşte": se caută un element conținut într-o singură mulțime; dacă un astfel de element este găsit, mulțimea respectivă este selectată şi se continuă cu autoapelarea rutinei recursive;
- dacă pasul "găseşte" eşuează, se va autoapela rutina recursivă selectând, pe rând, fiecare mulțime rămasă, în ordinea descrescătoare a numărului de elemente;
- la apelul următor se va lucra cu mulțimile rămase, din care se elimină toate elementele care se găsesc în mulțimi deja selectate (ceea ce poate duce la succese noi ale paşilor "elimină" şi "găseşte").

13.5. Concluzii

Problemele *NP* apar în numeroase domenii. Deşi se ştie că timpul necesar pentru rezolvare este exponențial, aceasta nu înseamnă că studiul lor ar trebui abandonat. Alternativa este de a găsi soluții practice cât mai eficiente.

Pentru abordarea cu succes a unei probleme *NP* este necesară o analiză atentă, cunoaşterea tuturor opțiunilor şi alegerea celor mai bune soluții. Aceasta poate implica testări complexe ale mai multor programe, folosind date de test cât mai apropiate de cele care vor apărea pe parcursul utilizării soluției finale.

Utilitatea abordării problemelor *NP* nu poate fi pusă la îndoială. O soluție performantă pentru problema 1) prezentată anterior poate duce, de exemplu, la scăderea costurilor necesare pentru deschiderea unei rețele de magazine care să acopere cât mai eficient o anumită zonă.

Este evident faptul că dacă întâlnim o problemă NP-completă nu este totul pierdut. Putem identifica numeroase posibilități de optimizare şi putem profita de orice particularitate a ei. În plus, chiar dacă problema nu este de fapt NP-completă, aceste tehnici pot fi oricum utilizate dacă nu avem un algoritm mai eficient.

13.6. Rezumat

În cadrul acestui capitol am tratat modul în care pot fi abordate problemele NP-complete. Am arătat cum pot fi îmbunătățiți timpii de execuție prin simplificarea problemei sau prin eliminarea soluțiilor neinteresante.

De asemenea, am descris câteva posibilități de explorare a spațiului soluțiilor şi am arătat cum putem profita de particularitățile problemelor pentru a găsi soluții cât mai performante.

În final am concluzionat că studiul NP-completitudinii este util şi că metodele descrise pot fi utilizate şi pentru problemele din clasa P.

13.7. Implementări sugerate

Pentru a implementa cât mai rapid soluțiile unor probleme NP-complete, vă sugerăm să aplicați tehnicile descrise în cadrul acestui capitol pentru a implementa soluții cât mai performante pentru:

1. problema celor N dame de şah care trebuie aşezate pe o tablă de latură N fără să există două dame care se atacă reciproc;
2. problema determinării unei submulțimi a unei mulțimi de numere reale, astfel încât suma elementelor submulțimii să fie o valoare dată;
3. problema determinării clicii maximale a unui graf;
4. problema jocului Perspico.

13.8. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate acestea sunt NP-complete, dar pot fi rezolvate în timpul de execuție specificat dacă algoritmii exponențiali folosiți sunt optimizați.

13.8.1. Scânduri

Un student avea, la un moment dat, N scânduri de lungimi a_1, a_2, ..., a_N a căror utilitate va rămâne, pentru totdeauna, un mister. Mânat de porniri distructive, datorită notelor obținute în ultima sesiune, el a luat un topor şi a tăiat fiecare scândură în două bucăți. Astfel a obținut $2 \cdot N$ bucăți de lungimi b_1, b_2, ..., $b_{2 \cdot N}$.

Câteva zile mai târziu, studentului i-a părut rău că a stricat scândurile (doar el ştie de ce) şi a dorit să reconstituie scândurile inițiale. Din nefericire, bucățile s-au amestecat, deci el nu a mai ştiut cum să le lipească.

Va trebui să determinați perechile de bucăți care trebuie lipite pentru a reconstitui scândurile inițiale.

Date de intrare
Prima linie a fişierului de intrare **SCANDURI.IN** conţine valoarea N care reprezintă numărul scândurilor iniţiale. Pe următoarele N linii se află câte un număr întreg, reprezentând lungimile celor N scânduri iniţiale. Pe următoarele $2 \cdot N$ linii se află câte un număr întreg, care reprezintă cele $2 \cdot N$ lungimi ale bucăţilor obţinute după tăiere.

Date de ieşire
Pe prima linie a fişierului de ieşire **SCANDURI.OUT** se va afla numărul K al scândurilor reconstituite. Pe următoarele K linii se află câte trei numere întregi S, X şi Y ($S = X + Y$) cu semnificaţia: o scândură de lungime S este reconstituită din două bucăţi de lungime X şi Y.

Numerele S reprezintă lungimi ale scândurilor iniţiale. Dacă o valoare S apare la începutul unor linii din fişierul de ieşire de x ori, atunci trebuie să existe cel puţin x scânduri de lungime S.

Numerele X şi Y reprezintă lungimi ale bucăţilor obţinute după tăiere. Dacă o valoare X apare ca al doilea sau al treilea număr pe o linie a fişierului de ieşire de y ori, atunci trebuie să existe cel puţin y bucăţi obţinute după tăiere care au lungimea X.

Restricţii şi precizări
- $1 \le N, b_i \le 100$;
- nu există mai mult de cinci scânduri care să aibă aceeaşi lungime;
- nu există mai mult de cinci bucăţi obţinute după tăiere care să aibă aceeaşi lungime;
- datele de intrare sunt alese în aşa fel încât să se poată reconstitui toate cele N scânduri ($K = N$);
- în cazul în care sunt reconstituite toate cele N scânduri, se va acorda întregul punctaj alocat unui anumit test.
- în cazul în care nu sunt reconstituite toate cele N scânduri, dar sunt reconstituite cel puţin $[3 \cdot N / 4]$ dintre acestea, se vor acorda $[P / 2]$ din cele P puncte alocate unui anumit test.

Exemplu

SCANDURI.IN	SCANDURI.OUT
6 10 15 20 25 30 35 5 5	6 15 10 5 20 10 10 25 10 15 30 15 15 35 20 15 10 5 5

```
5
10
10
10
10
15
15
15
15
20
```

Timp de execuție: 5 secunde/test

13.8.2. Litere

Pe o tablă se află *N* cerculețe unite între ele prin linii. Avem la dispoziție un număr total de *K* litere distincte. În fiecare dintre cerculețe, va trebui să scriem o literă astfel încât două cerculețe care sunt unite printr-o linie să conțină litere diferite. Evident, fiecare literă poate fi utilizată de oricâte ori dacă este satisfăcută condiția precedentă.

Date de intrare

Prima linie a fișierului de intrare **`LITERE.IN`** conține numărul *N* al cercurilor, numărul *M* al liniilor şi numărul *K* al literelor, separate prin câte un spațiu. Cea de-a doua linie va conține un şir format din *K* litere mari ale alfabetului englezesc, neseparate prin spații. Pe următoarele *N* linii se află câte două numere întregi *x* şi *y*, separate printr-un spațiu, cu semnificația: există o linie care uneşte cercurile identificate prin *x* şi *y*.

Date de ieşire

În cazul în care există cel puțin o soluție, fişierul de ieşire **`LITERE.OUT`** va conține un şir format din *K* litere, neseparate prin spații. Cea de-a *i*-a literă va fi scrisă în cercul identificat prin *i*.

În caz contrar, fişierul de ieşire va conține doar valoarea `-1`.

Restricții şi precizări

- $1 \le N \le 20$;
- $1 \le M \le 50$;
- $2 \le K \le N$;
- cercurile sunt identificate prin numere cuprinse între 1 şi *N*;
- există cel mult o linie între două cercuri *x* şi *y*;
- dacă există mai multe soluții, va fi generată doar una dintre ele.

Exemplu

LITERE.IN	LITERE.OUT
5 4 3 ABC	AABAA

Timp de execuție: 3 secunde/test

13.9. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare. Nu vom mai analiza complexitatea algoritmilor deoarece, datorită optimizărilor aduse, aceasta nu mai este relevantă.

13.9.1. Scânduri

Enunțul acestei probleme conține un indiciu destul de clar al faptului că ea este *NP*-completă şi anume faptul că se acordă punctaje parțiale.

Chiar dacă acordarea de punctaje parțiale nu implică neapărat *NP*-completitudinea unei probleme, acest fapt este un indiciu demn de luat în seamă. La concursurile de programare, marea majoritate a problemelor în care se acordă punctaje parțiale sunt *NP*-complete. O parte dintre excepții îl constituie problemele în care există mai multe subpuncte care trebuie rezolvate, fiecare dintre acestea fiind punctat separat.

În continuare vom prezenta modul în care va trebui construit algoritmul exponențial de rezolvare a acestei probleme care să se încadreze în limita de timp impusă.

Pentru a determina soluția vom folosi metoda *backtracking*. Restricțiile impuse în enunț (cel mult cinci scânduri cu aceeaşi lungime şi cel mult cinci bucăți cu aceeaşi lungime) permit scrierea de programe care să găsească soluția relativ repede.

Pentru a micşora cât mai mult posibil spațiul de căutare a soluțiilor va trebui să determinăm câte posibilități de reconstituire a unei scânduri există. Operația poate fi realizată foarte simplu în timp pătratic. Ulterior, vom ordona scândurile în funcție de numărul posibilităților de reconstituire.

Apoi, la fiecare pas al algoritmului care foloseşte metoda *backtracking* vom considera scândurile rămase în ordinea din şirul sortat. După alegerea a două bucăți care, prin lipire, duc la obținerea unei scânduri, numărul posibilităților de reconstituire se schimbă. Noile valori pot fi obținute în timp liniar. Va urma o reordonare a şirului care trebuie realizată în timp liniar-logaritmic. În momentul în care am reconstituit toate scândurile, vom afişa soluția.

13.9.2. Litere

Chiar dacă pentru această problemă nu se acordă punctaje parțiale, observăm imediat că ea este *NP*-completă. Practic avem de-a face cu problema *k*-colorării unui graf ale cărui noduri reprezintă cercurile, iar muchiile reprezintă liniile dintre cercuri.

Culorile pe baza cărora se va realiza colorarea sunt reprezentate de literele care trebui scrise în cercuri.

Aşadar, chiar dacă indiciul referitor la punctajele parțiale lipseşte, putem recunoaşte foarte rapid o problemă *NP*-completă. Recunoaşterea acestor probleme este un aspect deosebit de important în diverse situații.

O modalitate de rezolvare a problemei *k*-colorării a fost prezentată în cadrul acestui capitol, motiv pentru care nu o vom mai reda aici.