GENERAREA AUTOMATĂ A PROGRAMELOR DE CALCULATOR

Lucrare de licenţă

Univ. Babeş-Bolyai,
Cluj-Napoca, 1999

Student: Oltean Mihai
Coordonator: Prof. Univ. Dr. Militon Frenţiu

CUPRINS

1. INTRODUCERE

Undeva, prin preajma anului '94, un elev de clasa a XI-a se gândea să-şi construiască un unit **Pascal** care să îl ajute la implementarea rapidă a metodei **backtracking**. Pentru aceasta avea la îndemână un singur atu, care era repetat la nesfârşit în toate cărţile de informatică, ce tratau această metodă: *backtracking-ul se aplică problemelor a căror soluţie se poate reprezenta sub forma unui vector*... Deci, la prima vedere nimic dificil. Totul consta în a specifica care sunt elementele acelui vector, care sunt condiţiile de continuare şi care sunt condiţiile finale..., iar apoi soluţia ieşea ca pe bandă. Dar, abia mai târziu, acel elev (care de fapt eram eu) a înţeles că de fapt dificultatea unei probleme de informatică nu constă în determinarea tipului de algoritm care se aplică, ci în rezolvarea sutelor şi miilor de cazuri mărunţele care apar în jurul acesteia, cu alte cuvinte, de la *soluţie care se poate reprezenta sub forma unui vector* până efectiv la soluţie este o cale lungă şi spinoasă.

Puţin mai târziu, după ce am studiat bine şi celelalte metode de programare am dedus că fiecare dintre ele are un algoritm general de rezolvare, dar care nu era aşa de evident precum cel de la backtracking tocmai din cauză că la această metodă se sugera structura de date care trebuie folosită, şi anume un vector. Acum, când scriu aceste rânduri nu mi se mai pare aşa de dificilă găsirea unui algoritm general pentru o clasă de probleme. Pur şi simplu se foloseşte metoda rafinării în paşi succesivi. Adică, ca să fiu puţin mai explicit, se pleacă de la o problemă pentru care se scrie un algoritm. Apoi se analizează o altă problemă din aceeaşi clasă şi algoritmul iniţial se rescrie astfel încât să poată rezolva ambele probleme. Apoi se mai analizează încă o problemă... Şi nu trebuie să se continue această operaţie de mai mult de 10 (zece) ori, adică, cu alte cuvinte dacă se reuşeşte analizarea şi combinarea a 10 algoritmi reprezentativi, atunci rezultatul este suficient de general pentru a rezolva multe probleme din aceea clasă.

Programul nu l-am conceput ca pe un creator de elemente soft orientate pe metodele de programare. El este un mijloc şi nu un scop. De fapt este un mijloc pentru a crea un alt mijloc pe care îl voi folosi la crearea unui alt mijloc... etc, dar niciodată nu voi ajunge să creez

un scop. Toată viaţa nu vom face altceva decât să creăm mijloace, adică lucruri care ne vor ajuta să creăm alte lucruri.

Dorinţa mea este să îl integrez în ceva mai mare care nu se limitează la informatică. Dacă omul ar fi fost pus iniţial într-un mediu aşa arid precum sunt probleme de informatică atunci el n-ar fi fost niciodată inteligent. Complexitatea şi varietatea mediului în care a trăit l-au făcut capabil să gândească. Dacă ar fi fost înconjurat doar de biţii 0 şi 1 şi de instrucţiunile **for**, **if**, **while**, **begin**... nu ar fi fost în stare altceva decât să se mişte conform unor automatisme, care nici măcar nu ar fi fost inventate de el ci învăţate. Afirmaţia de mai sus se sprijină pe ceea ce am observat că se întâmplă cu majoritatea oamenilor după câţiva ani de la angajare: cad în automatisme şi rutină. Mă refer la orice fel de meserie; de la cea care implică doar munca fizică până la cea care implică doar munca intelectuală. Spre exemplu, singura diferenţă între un inginer şi un strungar este că cel dintâi execută automatisme cu un număr mai mare de paşi.

Dorinţa mea este să simulez modul de gândire uman. În mare spus, creierul face o generare a tuturor posibilităţilor, pentru cazurile de dimensiuni mici şi se foloseşte de euristici în cazurile de dimensiuni mari. Privit prin această prismă soft-ul meu pare a fi util. Mai trebuie o altă aplicaţie care să reuşească să reprezinte orice problemă adevărată sub formă de problemă de informatică, iar apoi folosirea acestui program ar putea da ceva rezultate.

Ceea ce am urmărit în primul rând la fiecare metodă a fost să o descriu într-un algoritm general, ceea ce prin prisma noţiunii de inteligenţă este complet şi iremediabil greşit. Calea pe care am apucat-o, din dorinţa de a crea un soft inteligent, duce într-o fundătură. Programul pe care l-am creat se poate numi oricum: sistem expert, instrument CASE, chiar şi soft de programare automată... etc, dar la nici un caz nu intră în categoria programelor inteligente. Dacă ar fi să enunţ un principiu care ar trebui să caracterizeze un soft inteligent, ceea ce aş spune este următorul lucru: *Este un soft care în 99% din cazuri face ceea ce îi ceri, dar mai rămâne un 1% în care se va comporta neaşteptat.* Atât timp cât programele se vor comporta cum te aştepţi, vor rămâne doar simple programe. Un calculator nu va intra niciodată la azilul de nebuni.

Nici implementarea unor algoritmi aleatori nu constituie o soluţie, atât timp cât elementul aleator va fi generat de un algoritm. Formula $x_{n+1}=a\ x_n \bmod M$, unde $M=2^{31}-1$, $a = 7^5$ şi x_0 arbitrar, nu spune nimic altceva, decât că un număr aleator nu este, de fapt, decât un număr nealeator şi că nimic neprevăzut nu se va întâmpla.

Creierul uman este într-adevăr în proporţie de 90% un calculator care funcţionează după nişte algoritmi bine definiţi. Greedy-ul euristic pentru cazurile de dimensiuni mari, Backtracking-ul pentru cazurile de dimensiuni mici, sunt metodele pe care le foloseşte creierul le foloseşte în orice situaţie. Dar creierul nu este doar atât. Este, şi aici aş dori să citez: *mai mult decât ar fi dacă ar fi numai ce este*. Ceea ce am realizat prin aplicaţia mea, se poate extinde. Următoarea etapă, pe care ar trebui să o urmeze cineva care a apucat-o deja pe această cale este să reuşească să reprezinte sub formă de algoritm orice problemă, iar apoi să genereze o soluţie, nu neapărat optimă, a ei. O următoare etapă ar consta din combinarea unui asemenea soft cu o puternică bază de cunoştiinţe. Evident aceasta trebuie să fie dinamică, adică să-şi poată singură adăuga, şterge, modifica cunoştiinţele. Dar pentru aceasta va trebui un alt soft, care va acţiona tot după algoritmii... Şi când toate aceste soft-uri le vom avea combinate, vom putea spune cu adevărat că am creat ... O ILUZIE! Nimic mai mult. Ceilalţi oameni se vor mira, şi-l vor găsi extraordinar, toţi se vor minuna, în afară de programatorul care l-a programat, care tot timpul se va întreba: *De ce mi-am pierdut eu atâţia ani din viaţă ca să fac un program care nu poate da rezultate mai bune decât un om?* (Aici *mai bune* trebuie înţeles într-un sens mai larg, căci spre exemplu efectuarea unei operaţii de înmulţire se va efectua întotdeauna mai rapid de către un calculator decât de către un om).

Dar, cel puţin pentru programarea automată şi inteligenţa artificială există un scop foarte bine definit: *omul*. Nu dorim să fim înconjuraţi de roboţi, ci de alţi oameni. Nu dorim să fim înconjuraţi de comportamente mecanice, care se mişcă după nişte algoritmi rigizi. Lucrul acesta se observă încă de pe acum în străduinţa design-erilor de a crea înfăţişări ale roboţilor cât mai apropiate de cele ale oamenilor. Aceeaşi tendinţă şi străduinţă se manifestă şi în rândul oamenilor de ştiinţă, mă refer în special a celor care lucrează în domeniu, de a crea maşini cu comportamente umanoide. Oricum, rezultatele sunt mai mici aici, căci suntem doar la început.

Am folosit puţin mai sus două noţiuni una lângă alta şi anume *programarea automată* şi *inteligenţa artificială*. Le consider inseparabile. Un soft de programare automată este definit ca fiind o aplicaţie care poartă un dialog cu utilizatorul pentru a obţine toate elementele necesare creării unui nou program. Dar precum se poate vedea şi din aplicaţia care constituie subiectul acestui material sau din alte aplicaţii de acest tip, nu se pot acoperi toate cazurile, cu alte cuvinte un algoritm nu poate pune toate întrebările, ci doar pe acelea pe care le ştiu, sau la care ştiu răspunde programatorii lui. De aceea pe lângă algoritm mai trebuie şi ceea ce

informaticienii numesc *inteligenţă artificială*. În caz contrar am avea de a face cu un instrument CASE.

Pentru că tot veni vorba despre acest CASE, cred că ar fi bine să explic puţin această noţiune şi care sunt diferenţele dintre un instrument CASE şi un programabil automat. CASE vine de la **Computer Aide Software Engineering**, adică calculatorul ajută la dezvoltarea softului. Există o multitudine de instrumente CASE, spre exemplu *AutoCorect*-ul unui editor de text. Mediile de programare din ziua de astăzi au de asemenea o mulţime de astfel de instrumente incorporate, de la supravegherea sintaxei, până la generarea automată de cod. Din această cauză este greu de făcut o deosebire netă între aceste două noţiuni. Nu mă refer aici la graniţe, ci efectiv la miez, adică devreme ce şi instrumentul CASE şi programabilul automat se ocupă cu generarea de cod pe baza unor specificaţii nu va fi întotdeauna uşor de spus când avem de a face cu una şi când cu alta. Totuşi, ceea ce putem spune cu certitudine este că programarea automată este pe o treaptă superioară faţă de instrumentele CASE. Un instrument CASE întotdeauna nu va fi decât un instrument CASE, pe când un soft de programare automată s-ar putea să devină mai mult decât atât.

Aplicaţia la care se referă acest material intră în categoria programării automate. Am să motivez în continuare această afirmaţie. Ceea ce am urmărit a fost rezolvarea problemelor orientată pe metode de programare, adică cu alte cuvinte am plecat invers, de la rezolvare spre enunţ. De fapt enunţul nici nu contează aici, important pentru cel care utilizează această aplicaţie este să ştie să integreze corect problema în categoria corespunzătoare, iar apoi să specifice elementele metodei folosite. În final, pe baza acestor specificaţii se va genera cod sursă **Pascal**, care apoi va putea fi compilat şi executat. Interfaţa aplicaţiei se apropie oarecum de noţiune de TAD (tip abstract de date), adică utilizatorul nu trebuie să fie preocupat de amănunte, ci trebuie doar să trateze în linii mari implementarea rezolvării, detaliile fiind apoi generate automat. Mai este subliniat faptul că nu se poate genera chiar totul doar din interfaţă. Este din cauză că, repet, un algoritm nu poate pune chiar toate întrebările, şi pentru aceasta utilizatorul va trebui să mai intervină pe alocuri.

2. METODA BACKTRACKING

1. Descrierea metodei

În informatică apare frecvent situaţia în care rezolvarea unei probleme conduce la determinarea unor vectori de forma:

$x=(x_1,x_2,...,x_n)$, unde

- fiecare componentă x_k aparţine unei mulţimi finite V_k;
- există anumite relaţii ce trebuie satisfăcute de componentele vectorului, numite condiţii interne, astfel încât x este o soluţie a problemei dacă şi numai dacă aceste condiţii sunt satisfăcute de componentele $x_1,x_2,...,x_n$ ale vectorului x.

Produsul cartezian $V_1xV_2x...V_n$ se numeşte spaţiul soluţiilor posibile. Soluţiile problemei sunt acele soluţii posibile care îndeplinesc condiţiile interne.

O modalitate de rezolvare a acestei probleme este generarea tuturor soluţiilor posibile (care sunt în număr de $|V_1|*|V_2|*...*|V_n|$), iar apoi testarea pentru fiecare din acestea a condiţiilor interne. Motivul principal pentru care această metodă de abordare nu este utilă ar fi numărul foarte mare de soluţii posibile. De exemplu numărul modalităţilor de completare a unei foi de PronoSport este de 3^14 = 1.594.323, deci un număr foarte mare. Mai precis, numărul soluţiilor posibile creşte exponenţial cu mărimea intrării. Este evident că nu întotdeauna algoritmii exponenţiali pot fi evitaţi (de exemplu în problema generării tuturor submulţimilor unei mulţimi cu n elemente, şi care sunt în număr de 2^n, deci un timp de lucru exponenţial), dar în multe cazuri problema care se pune este *cât de exponenţiali*? sunt algoritmii pe care îi folosim. Poate nu este evident pentru toată lumea, dar un algoritm cu timp de lucru 2^n este mult mai performat decât un algoritm cu timp de lucru 3^n. De exemplu pe un calculator care poate efectua 1 milion de operaţii pe secundă, efectuarea a 2^n operaţii, pentru n = 20, necesită 17.9 minute, iar pentru efectuarea a 3^n operaţii, pentru acelaşi n = 20, necesită 6.5 ani.

Metoda backtracking urmăreşte evitarea generării tuturor soluţiilor posibile, micşorându-se astfel complexitatea algoritmului şi scurtându-se timpul de execuţie. Trebuie precizat de la bun început că la nici un caz (exceptând eventual câteva cazuri particulare) nu vom reduce complexitatea algoritmului de la una exponenţială la una polinomială, ci doar la una mai puţin exponenţială (de exemplu, cum am spus şi mai sus, de la un 3^n la un 2^n).

Componentele vectorului x primesc valori în ordinea crescătoare a indicilor. Aceasta înseamnă că lui x_k nu i se atribuie o valoare decât după ce $x_1, \ldots, x_{k-1}$ au primit valori ce nu contrazic condiţiile interne. Mai mult decât atât valoarea lui x_k trebuie aleasă astfel încât $x_1, x_2, \ldots, x_k$ să îndeplinească anumite condiţii, numite *condiţii de continuare*, care sunt strict legate de condiţiile interne. Un alt tip de condiţii care mai există, sunt aşa zisele *condiţii finale* care teoretic nu se diferenţiază de condiţiile de continuare, dar pe care aplicaţia mea trebuie să le ia în considerare separat. O problemă în care apare necesitatea existenţei unor astfel de condiţii este problema plăţii unei sume având monezi de anumite tipuri. În vectorul soluţie punem de fiecare dată o monedă care adunată la cele precedente nu trebuie să depăşească suma dată. Dar în momentul în care punem ultima monedă trebuie să avem neapărat suma exact egală cu cea care trebuie să o plătim şi nu mai mică sau egală. Condiţiile de continuare în acest caz ar fi date de relaţia $\leq$,iar condiţiile finale sunt impuse de relaţia =. Cu alte cuvinte condiţiile de continuare nu sunt suficiente pentru obţinerea unei soluţii. Se întâmplă uneori, precum am văzut şi la problema precedentă, că este foarte posibil ca vectorul $x = (x_1, x_2, \ldots, x_n)$ să îndeplinească condiţiile de continuare, şi totuşi să nu fie soluţie din cauză că nu îndeplineşte condiţiile finale.

Neîndeplinirea condiţiilor de continuare exprimă faptul că oricum am alege $x_{k+1}, \ldots, x_n$ nu vom obţine o soluţie (deci condiţiile de continuare sunt strict necesare pentru obţinerea unei soluţii). Ca urmare se va trece la atribuirea unei valori lui x_k doar când condiţiile de continuare sunt îndeplinite. În cazul neîndeplinirii acestor condiţii se alege o nouă valoare pentru x_k, sau în cazul în care mulţimea finită de valori V_k a fost epuizată, se revine la componenta precedentă, adică x_{k-1} şi se va face o nouă alegere pentru aceasta. Această revenire dă numele metodei, exprimând faptul că atunci când nu mai putem avansa, urmărim înapoi secvenţa curentă din soluţie.

Alegerea condţiilor de continuare este foarte importantă, o alegere bună ducând la o reducere substanţială a numărului de calcule. În cazul ideal, condiţiile de continuare ar trebui să fie nu numai necesare, dar şi suficiente pentru obţinerea unei soluţii.

Prin metoda backtracking, orice vector soluţie este construit progresiv, începând cu prima componentă şi mergând până la ultima, cu eventuale reveniri asupra valorilor atribuite anterior. Reamintim că $x_1, \ldots, x_n$ primesc valori în mulţimile $V_1, \ldots, V_n$.

Se mai poate face o restricţie asupra mulţimilor $V_1, \ldots V_n$, din cauză că o mulţime V_k poate fi determinată, în unele cazuri doar după ce a fost completat vectorul x, până la poziţia $k-1$. În majoritatea cazurilor însă mulţimea V_k poate fi determinată de la început, dar în

momentul în care se fac atribuiri elementului x_k, alegerile se pot face doar dintr-o submulţime D_k a lui V_k (D_k să o numim mulţimea valorilor disponibile pentru alegerea lui x_k). Această mulţime D_k depinde exact de alegerile lui $x_1, \ldots, x_{k-1}$. Spre exemplu la generarea tuturor combinărilor de n obiecte luate câte m, elementele V_k sunt din domeniul $1..n$, dar la alegerea mulţimea valorilor disponibile D_k pentru x_k este $x_{k-1}+1...n$. De obicei calcularea elementelor mulţimilor D_k se face de obicei în funcţiile de continuare. Există însă unele situaţii în care este utilă (mă refer ca viteză de lucru a algoritmului) testarea condiţiilor de continuare direct pe mulţimile D_k şi nu pe toate elementele mulţimilor V_k. Spre exemplu la generarea tuturor combinărilor este mai utilă o instrucţiune **for** de la x_{k-1} până la n, decât testarea condiţiilor de continuare pentru toate valorile de la 1 la n. Un alt exemplu mai concludent este problema ieşirii din labirint. Este clar că mulţimea valorilor V_k a elementelor care alcătuiesc drumul este egală cu mulţimea perechile (i,j), unde i şi j sunt orice cameră liberă din matricea care codifică labirintul. Dar la un moment dat, este clar că pentru a continua drumul avem doar maxim trei posibilităţi de ales, deci nu este necesară testarea tuturor condiţiilor de continuare pentru toate căsuţele labirintului, ci doar pentru acelea maxim trei în care se poate ajunge dintr-o mutare. Oricum, la nivel teoretic existenţa mulţimilor D_k nu aduce nimic nou, deci în continuarea acestei expuneri o să evit folosirea lor.

În urma atribuirii vor rezulta nişte mulţimi C_k incluse în V_k denumite şi mulţimi consumate. Pentru fiecare componentă x_k există o mulţime C_k a valorilor consumate la momentul curent.

O descriere completă a metodei se poate face prin precizarea, în momentul în care se încearcă atribuirea unei valori x_k, a următoarelor elemente:

1) valorile curente $v_1, \ldots, v_{k-1}$ ale componentelor $x_1, \ldots, x_{k-1}$;
2) mulţimile $C_1, \ldots, C_{k-1}$ de valori consumate pentru fiecare dintre componentele $x_1, \ldots, x_{k-1}$.

Această descriere poate fi sintetizată într-un tabel numit configuraţie, având forma următoare:

$$\left(\begin{array}{c|c} v_1 \ldots v_{k-1} & x_k\ x_{k+1} \ldots x_n \\ C_1 \ldots C_{k-1} & C_k\ \varnothing \ldots\ \varnothing \end{array}\right)$$

Semnificaţia unei astfel de configuraţii este următoarea:

a) în încercarea de a construi un vector soluţie, componentele $x_1, \ldots, x_{k-1}$ au primit valorile $v_1, \ldots, v_{k-1}$;
b) aceste valori satisfac condiţiile de continuare;

c) urmează să se facă o încercare de atribuire asupra componentei x_k; deoarece valorile consumate până în acest moment sunt cele din C_k, ea urmează a primi o valoare $x_k \in V_k \setminus C_k$;

d) valorile consumate pentru componentele $x_1, \ldots, x_{k-1}$ sunt cele din mulţimile $C_1, \ldots, C_{k-1}$, cu precizarea că valorile curente $v_1, \ldots, v_{k-1}$ sunt considerate consumate, deci apar în mulţimile $C_1, \ldots, C_{k-1}$;

e) pentru componentele $x_{k+1}, \ldots, x_n$ nu s-a consumat nici o valoare şi prin urmare $C_{k+1} = \ldots = C_n = \varnothing$;

f) până în acest moment au fost deja construite:

- eventuale soluţii de forma $(v_1, \ldots, v_{k-1}, c_k, \ldots)$ cu $c_k \in C_k$
- eventuale soluţii de forma $(c_1, \ldots, c_{k-1}, \ldots)$ cu $(c_1, \ldots, c_{k-1}) \in C_1 x \ldots C_{k-1}$ şi $(c_1, \ldots, c_{k-1}) \neq (v_1, \ldots, v_{k-1})$.

Metoda backtracking începe prin a fi aplicată în situaţia în care nu a fost consumată nici o valoare şi se încearcă o atribuire asupra primei componente. Acest lucru este specificat de **configuraţia iniţială** care este de forma:

$$\begin{pmatrix} | & x_1 & \ldots & x_n \\ | & \varnothing & \ldots & \varnothing \end{pmatrix}$$

Configuraţia finală este de forma :

$$\begin{pmatrix} v_1 & \ldots & v_n & | \\ C_1 & \ldots & C_n & | \end{pmatrix}$$

În cursul aplicării metodei, o configuraţie poate fi obiectul a 4 tipuri de modificări, prezentate în cele ce urmează:

1. Atribuie şi avansează

$$\begin{pmatrix} v_1 \ldots v_{k-1} & | & x_k & x_{k+1} \ldots x_n \\ C_1 \ldots C_{k-1} & | & C_k & \varnothing \ldots \varnothing \end{pmatrix} \quad V_k \rightarrow \quad \begin{pmatrix} v_1 \ldots v_{k-1} & v_k & | & x_{k+1} \ldots x_n \\ C_1 \ldots C_{k-1} & C_k \cup \{v_k\} & | & \varnothing \ldots \varnothing \end{pmatrix}$$

2. Încercare eşuată

$$\begin{pmatrix} v_1 \ldots v_{k-1} & | & x_k & x_{k+1} \ldots x_n \\ C_1 \ldots C_{k-1} & | & C_k & \varnothing \ldots \varnothing \end{pmatrix} \quad V_k \; = = = \quad \begin{pmatrix} v_1 \ldots v_{k-1} & | & x_k & x_{k+1} \ldots x_n \\ C_1 \ldots C_{k-1} & | & C_k \cup \{v_k\} & \varnothing \ldots \varnothing \end{pmatrix}$$

3. Revenire

$$\begin{pmatrix} v_1 \ldots v_{k-1} & | & x_k & x_{k+1} \ldots x_n \\ C_1 \ldots C_{k-1} & | & C_k & \varnothing \ldots \varnothing \end{pmatrix} \quad \leftarrow \quad \begin{pmatrix} v_1 \ldots v_{k-2} & | & x_{k-1} & x_k \ldots x_n \\ C_1 \ldots C_{k-2} & | & C_{k-1} & \varnothing \ldots \varnothing \end{pmatrix}$$

4. **Revenire după construirea unei soluţii**

$$\begin{pmatrix} v_1 \ldots\ldots v_n & | \\ C_1 \ldots\ldots C_n & | \end{pmatrix} \quad <<< \quad \begin{pmatrix} v_1 \ldots v_{n-1} & | & x_n \\ C_1 \ldots C_{n-1} & | & C_n \end{pmatrix}$$

Aceste etape ne dau posibilitatea construirii unui algoritm general care permite generarea unui cadru universal al rezolvării problemelor prin metoda backtracking. În funcţie de implementare, algoritmul poate fi recursiv sau nerecursiv:

1) Implementare nerecursivă

```
 {iniţializează mulţimile de valori V1,...Vn}
 {construieşte configuraţia iniţială:}
k = 1
Ci = ∅
while k>0
  {configuraţia nu este finală }
   if k=n+1
   then {configuraţia este de tip soluţie }
        reţine soluţia x = (x1,...,xn)
        k = k-1 {revenire după construirea unei soluţii }
   else if Ck≠Vk
        then { mai există valori neconsumate }
             alege o valoare vk din Vk\Ck
             Ck = Ck∪ {vk}
             if v1,...,vk satisfac condiţiile de continuare
             then {atribuie şi avansează }
                   xk = vk
                   k = k+1
             else {încercare eşuată }
        else {revenire}
              Ck = ∅
              k = k-1
```

2) Implementare recursivă

În majoritatea cazurilor nu este necesar calculul mulţimilor C_k, de aceea în algoritmul care urmează am să evit această operaţie:

```
procedure back (k )
  if k=n+1 {configuratia este finala}
  then afiseaza_solutie
  else for c in Vk do {pentru fiecare element din Vk}
          xk = c
          if continuare(k)
          then back(k+1)
end
```

Acestea fiind spuse, consider că prezentarea teoretică a metodei a fost suficientă pentru a se putea trece la prezentarea părţii de implementare a aplicaţiei.

2. Implementarea în aplicaţie

Trebuie urmărite câteva elemente definitorii pentru metoda backtracking, iar în rest totul nu depinde decât de preferinţele programatorului, adică în cazul de faţă ale mele. Printre elementele strict personale ale metodei backtracking enumăr: tipul elementelor vectorului, domeniul elementelor vectorului, numărul soluţiilor, costul unei soluţii, lungimea unei soluţii, condiţiile de continuare, condiţiile finale, modalitatea de construire a vectorului soluţie. Mai există încâ câteva elemente care trebuie specificate, dar din cauză că sunt comune tuturor metodelor de programare le-am expus în capitolul precedent. Acestea ar fi datele de intrare, datele de ieşire, iniţializarea, finalizarea, unde se stochează soluţia. Totuşi despre datele de ieşire şi despre iniţializare am să vorbesc şi în acest capitol, deoarece ele conţin elemente specifice acestei metode.

2.1 Tipul elementelor vectorului

Această parte ţine de implementarea efectivă în limbajului Pascal. Vectorul poate avea unul din tipurile de bază (byte, integer, longint, word, shortint, comp, real, string), sau un tip definit de către utilizator şi care poate fi de tip mulţime, enumerare, tablou sau înregistrare.

2.2 Domeniul elementelor vectorului

Există două mari modalităţi pentru a specifica mulţimea valorilor pe care le poate lua un element al vectorului la un moment dat. Prima dintre ele, care este însă şi cea mai dificilă de implementat, constă în returnarea prin intermediul procedurii `ConstruiesteMultime` a mulţimii de elemente disponibile la momentul curent (adică, cu alte cuvinte un V_k). Forma procedurii `ConstruiesteMultime` este standard, şi constă în două grupuri de parametrii: primul grup conţine informaţii despre soluţia parţială construită până la momentul curent (de obicei se trimite valoarea lui `k`); al doilea grup constă din doi parametrii: primul parametru este un vector, sau o listă formată din elementele care alcătuiesc mulţimea V_k curentă, iar al doilea parametru este numărul de elemente al acestui vector sau liste. Un exemplu de problemă la care se poate folosi această procedură este săriturii calului de şah (astfel încât toate căsuţele să fie atinse exact o dată), în care la un moment dat, prin intermediul lui

`ConstruiesteMultime` se poate returna mulţimea căsuţelor în care se poate sări la următoarea mutare.

Cea de a doua modalitate de specificare a mulţimii valorilor pe care le poate lua un element la un moment dat este cea directă şi constă în precizarea exactă a domeniului acestor elemente. De exemplu dacă vectorul soluţie are componente de tip întreg, atunci o valoare de tipul `a..b` este o specificare corectă a unui domeniu. În cazul acesta, limitele `a` şi `b` sunt nu neapărat constante, ci chiar şi variabile sau expresii aritmetice care conţin constante şi variabile ce trebuie însă să poată fi evaluate la momentul folosirii. De asemenea evaluarea lor trebuie făcută la întreg, deoarece în implementare folosesc o structură de tip **for**. Dacă tipul de bază al soluţiei este **string**, atunci de asemenea se poate specifica un domeniu de tip `a..b`, unde `a` şi `b` sunt de data aceasta şiruri de caractere. Se consideră că relaţia de ordine dintre acestea este cea de ordine lexicografică. De fapt, relaţia aceasta de ordine care se impune între elemente determină o puternică limitare asupra programelor generate. Dacă în cazul tipului de date întreg, am putut impune o relaţie de ordine <, care de altfel este foarte des întâlnită, în cazul şirurilor de caractere, a mulţimilor, a tipului enumerare sau tablou, va fi foarte greu să impun o anumită relaţie de ordine. Bineînţeles că eu pot spune mulţimea `A < B`, dacă şi numai dacă reprezentarea sub formă binară (cea folosită şi pentru tipul **set** în Pascal) a lui `A` este mai mică decât a lui `B`, dar această restricţie îmi va reduce mult din generalitate. Mai există tipul de date înregistrare (**record**), care este rezolvat foarte simplu în cazul în care are doar membrii întregi, şi anume se specifică pentru fiecare din membrii domeniul sub forma `a..b`. Un exemplu de soluţie în care se foloseşte un vector de record-uri de întregi este celebra problemă a săriturii calului pe tabla de şah, în care un drum al calului constă dintr-o înşiruire de poziţii de pe tablă, adică cu alte cuvinte o înşiruire de perechi (`i,j`).

2.3 Iniţializare

Este necesar în anumite situaţii să se facă iniţializarea anumitor poziţii din vectorul soluţie. Cele mai des întâlnite iniţializări sunt acelea în care se atribuie valori primei poziţii (având indicele 1) sau poziţiei 0. Oricum dacă aceste două iniţializări nu sunt suficiente, atunci utilizatorul acestei aplicaţii poate interveni în codul sursă generat, mai precis poate interveni în interiorul procedurii iniţializează.

Iniţializarea unei poziţii se face prin specificarea unei valori poziţiei respective. Se păstrează sintaxa **Pascal**. Exemplu de problemă în care este utilă înităializarea poziţie 0 cu o valoare este generarea combinărilor de `n` elemente luate câte `m`. Deoarece întotdeauna un

element al soluţie este strict mai mare decât precedentul, este util să setăm valoarea poziţiei 0 la 0, astfel că primul element valid va putea lua toate valorile până la `n`.

2.4 Condiţii de continuare

Un element al vectorului soluţie are de ales între patru posibilităţi: să fie independent de toate elementele, să fie dependent doar de cel de dinaintea lui, să fie dependent de toate elementele de dinaintea lui, şi să fie dependent şi de ultimul şi de restul celorlalte. Cea mai des întâlnită formă de dependenţă faţă de toate celelalte de dinainte este relaţia $\neq$. În celelalte cazuri relaţiile de dependenţă se specifică prin intermediul a două funcţii booleane: `RelatieDeToate` şi `RelatieDeUltimul`.

Există cazuri în care trebuie testată apartenenţa elementului curent construit la un domeniu elementelor vectorului. Spre exemplu în cazul problemei săriturii calului pe tabla de şah, este clar că adăugând la poziţia `(x,y)` perechea `(1,2)` nu sunt sigur dacă ceea ce rezultă este sau nu pe tablă.

2.5 Condiţii finale

Spre deosebire de condiţiile de continuare, condiţiile de final pot fi specificate mult mai strict. În continuare am să menţionez 4 (patru) categorii de cazuri în care se poate termina de construit o soluţie:

- la final vectorul are o lungime dată. De exemplu la problema generării permutărilor vectorul soluţie are o lungime fixă `n`.
- ultima poziţie are o valoare specificată. De exemplu la determinarea unui drum între două noduri ale unui graf. În acest caz apare şi necesitatea iniţializării primei poziţii a vectorului soluţie cu primul nod al drumului cerut.
- vectorul nu mai poate fi mărit, adică mulţimea V_k care urma să fie construită este vidă. Exemplu este următoarea problemă: un comisvoiajor dispune de o sumă de bani pentru a trece prin n oraşe. În fiecare oraş `k` el poate cheltui suma de `a[k]` lei. Se cere un traseu al lui astfel încât să poată vizita cât mai multe oraşe.
- în momentul în care între elemente există o relaţie. De exemplu la descompunerea unei sume în monezi de valori date, avem că terminarea construirii vectorului soluţie se face când suma elementelor este chiar valoarea dată.

2.6 Numărul soluţiilor

Se poate cere determinarea:

- tuturor soluţiilor problemei; spre exemplu la generarea tuturor permutărilor. În acest caz codul sursă generat va conţine un vector de soluţii denumit `solutii` şi eventual un vector `lungimi`, în care se reţine lungimea fiecărei soluţii.
- doar a unei soluţii, care poate fi prima găsită, sau a `x`-a găsită. Dacă se cere prima, atunci codul sursă generat conţine doar un vector global `solutie` în care se reţine soluţia, şi eventual o variabilă `lungime` pentru lungimea soluţiei. Dacă se cere o altă soluţie, atunci în cazul în care soluţia nu are o lungime fixă şi un cost fix este necesară introducerea variabilei globale `solutii` pentru reţinerea tuturor soluţiilor, iar apoi se afişează cea cu numărul de ordine dorit. În celelalte cazuri nu este necesară prezenţa vectorului `solutii`, ci doar a unei variabile care contorizează numărul de ordine al soluţiei generate.
- a primelor `x` soluţii
- a unei soluţii a cărei număr de ordine îndeplineşte o condiţie dată

2.7 Lungimea soluţiei

Această specificare trebuie impusă în cazul în care nu se ştie foarte clar lungimea soluţiei. Spre exemplu la determinarea unui drum elementar de lungime minimă între două noduri ale unui graf, este clar că pe lângă condiţia ca la final să ajungem în nodul destinaţie, mai trebuie ca şi lungimea traseului să fie minimă.

Există de altfel trei mari cazuri de tratat:

- lungimea soluţiei trebuie să fie minimă
- lungimea soluţiei trebuie să fie maximă
- lungimea soluţiei nu contează

Dacă se alege unul din primele două cazuri, codul sursă generat va conţine o variabilă globală cu numele `lungime`. Aceasta va fi folosită pentru a se determina dacă soluţia are sau nu lungime extremă (maximă sau minimă), adică cu alte cuvinte se va iniţializa valoarea ei cu 0 sau MaxInt (în funcţie de extrema dorită), apoi după testul de terminare a construirii unei soluţii, se va testa dacă este extremă (maximă sau minimă). Dacă da, atunci în funcţie de numărul de soluţii dorit se va reţine sau nu într-un tablou al soluţiilor.

Se mai poate impune o restricţie asupra lungimii unei soluţii, adică, se poate limita lungimea ei la o anumită valoare specificată printr-o expresie aritmetică ce include constante întregi şi variabile evaluabile la momentul respectiv.

Mai există un lucru asupra căruia trebuie insistat aici, deşi el priveşte şi următorul aliniat, adică cel legat de costul unei soluţii. Există cazuri în care se cere, spre exemplu, o soluţie de cost minim şi de lungime maximă. Aici apare necesitatea ordonării evaluărilor de extrem. Adică cu alte cuvinte, cum alegem? Dintre toate soluţiile de cost minim o alegem pe cea de lungime maximă? Sau, dintre toate soluţiile de lungime maximă o alegem pe cea de cost minim? Cred că este evident că nu reprezintă unul şi acelaşi lucru. De aceea este introdusă noţiunea de prioritate. În aplicaţia mea se poate specifica, prin intermediul unei componente de verificare sau validare, dacă se doreşte ca lungimea să fie prioritară asupra costului sau invers.

2.8 Costul soluţiei

Costul soluţiei este o noţiune aproape asemănătoare cu lungimea soluţiei. Se poate cere ca, costul să fie minim, maxim, să aibă o valoare precizată. Există însă cazuri în care acest lucru nu este suficient, adică există probleme care admit mai multe funcţii de cost în acelaşi timp. Pentru unele funcţii se cere maximul, pentru altele minimul, deci nu se poate să existe o singură funcţie care să admită două tipuri de extreme. De exemplu, se poate cere să se ajungă din oraşul A în oraşul B, schimbând un număr minim de trenuri, dar în acelaşi timp se cere să se treacă printr-un număr maxim de oraşe. Datorită unor astfel de cazuri aplicaţia mea este prevăzută cu posibilitatea specificării mai multor funcţii de cost, fiecare având propriul ei tip de extrem. Pentru fiecare astfel de funcţie, este introdusă, în codul sursă generat, câte o variabilă denumită `cost1,...,costk`, care au rolul de a reţine costul unei soluţii. Aceste variabile globale şi pot fi folosite oriunde de către utilizator.

Apare şi aici, ceea ce am discutat la paragraful anterior, adică noţiunea de prioritate, pe care nu o mai reiau şi aici.

2.9 Construirea soluţiei

Primul lucru care trebuie specificat este indicele de la care începe construirea unei soluţii. De obicei se începe de la poziţia 1, ceea ce însemnă că apelul procedurii recursive se face cu `back(1)`. Dacă cumva poziţia de la care se începe este mai mare decât 1, atunci în cadrul iniţializării vor trebui specificate valori pentru elementele necompletate.

Apoi, un element nou poate fi construit în următoarele feluri:

- doar pe baza domeniului de definiţie; adică pur şi simplu se va face o parcurgere a tuturor elementelor din domeniu specificat la aliniatul despre **domeniul elementelor vectorului**.

- pe baza ultimului element introdus; există cazuri în care este mai uşoară şi evident mult mai eficientă calcularea unui nou element pe baza ultimului element introdus. Spre exemplu, la problema săriturii calului pe tabla de şah este clar că noua poziţie se poate afla doar în una din poziţiile (1,2),(2,1),(1,-2),(2,-1),(-1,2),(-1,-2),(-2,1),(-2,-1) faţă de poziţia precedentă. În acest caz (şi în multe altele asemănătoare) este mult mai utilă o astfel de specificare, decât parcurgerea întregului domeniu al elementelor şi testarea condiţiilor de continuare.
- pe baza tuturor elementelor anterioare.

2.10 Date de ieşire

Datorită faptului că soluţiile au o formă standard (adică se reprezintă sub forma unui vector), este normal ca în multe cazuri să se ceară să se afişeze elemente ale acestui vector. De aceea aplicaţiei mele i se poate specifica că se doreşte

- afişarea întregului vector (cum este de exemplu la generarea permutărilor) , sau
- ultima poziţie din vector (cum este cazul problemei ieşirii dintr-un labirint), sau,
- poziţie oarecare din vector (spre exemplu se cere afişarea oraşului aflat pe traseul de lungime minimă care leagă oraşul A de oraşul B. În acest caz se cere a fi afişată poziţia `lungime` / 2 a vectorului soluţie. Variabila `lungime` este definită globală şi indică lungimea unei soluţii când aceasta este minimă sau maximă).
- afişarea lungimii soluţiei (spre exemplu drumul de lungime minimă care leagă două oraşe)
- afişarea costului soluţiei (spre exemplu costul minim necesar pentru a ajunge din oraşul A în oraşul B cu trenul).
- eventual se mai poate cere şi afişarea timpului de execuţie al programului.

În cazul în care se doresc şi alte lucruri pentru afişat, programatorul poate interveni în codul sursă, al programului rezultat, ştiind că întotdeauna afişarea rezultatelor se face prin intermediul procedurii **Afişează**.

3. Exemple de utilizare

1 Generarea permutărilor

Pentru această problemă trebuie specificate următoarele lucruri:

- ✓ Numele fişierului de intrare este perm.in care conţine o singură valoare ce va fi citită în variabila de tip byte n.

- ✓ Tipul de bază al vectorului soluţie este byte.
- ✓ Domeniul unui element este 1..n
- ✓ Se doresc toate soluţiile
- ✓ Condiţiile de continuare sunt de tip condiţionare de toate cele de dinainte, şi anume diferit de toate cele de dinainte.
- ✓ La final vectorul are lungimea n
- ✓ Construirea vectorului se face pe baza domeniului

Codul sursă generat este următorul:

```
{Autor:  }
Program Backtracking;

uses crt;

type      TSolutie=array[0..20] of Byte;

var v:TSolutie;
    solutii:array[1..100] of TSolutie;
    nrsol:integer;
    n:Byte;

procedure Citire;
  var f:text;
begin
  assign(f,'perm.in');
  reset(f);
  readln(f,n);
  Close(f);
end;

procedure Afisare;
  var i,k:byte;
begin
  for k:=1 to nrsol do
  begin
    for i:=1 to n do
    begin
      write(solutii[k,i],',');
    end;
    writeln;
  end
end;

function LungimeData(k:byte):boolean;
begin
  LungimeData:=k=n;
end;

function final(k:byte):boolean;
begin
  final:=LungimeData(k);
end;
```

```
function Continuare(pozitie:integer; element:Byte):boolean;
  var c:byte;
begin
  Continuare:=true;
  for c:=1 to pozitie-1 do
    if v[c]=element
    then begin
           Continuare:=false;
           exit;
         end;
end;

procedure back(k:integer);
  var element:Byte;
y0:byte;
begin
  if final(k-1)
  then
  begin
    inc(nrsol);
    solutii[nrsol]:=v;
  end
  else begin
  for y0:=1 to n do
  begin
  element:=y0;
  if Continuare(k,y0)
  then begin
       v[k]:=element;
       back(k+1);
     end;
end;
  end
end;

procedure Initializeaza;
begin
  nrsol:=0;
end;

begin
  Citire;
  Initializeaza;
  back(1);
  Afisare;
end.
```

2 Generarea combinărilor

Pentru această problemă trebuie specificate următoarele lucruri:

- ✓ Numele fişierului de intrare este comb.in care conţine două valoari ce vor fi citite în variabilele de tip byte `n` şi `m`.
- ✓ Tipul de bază al vectorului soluţie este byte.
- ✓ Domeniul unui element este `v[k]+1..n`
- ✓ Se doresc toate soluţiile

- ✓ La final vectorul are lungimea m
- ✓ Construirea vectorului se face pe baza domeniului
- ✓ Se iniţializează poziţia 0 cu 0.

```
{Autor:  }
Program Backtracking;

uses crt;

type      TSolutie=array[0..20] of Byte;

var v:TSolutie;
    solutii:array[1..100] of TSolutie;
    n:Byte;
    m:Byte;
    nrsol:word;

procedure Citire;
  var f:text;
begin
  assign(f,'comb.in');
  reset(f);
  readln(f,n);
  readln(f,m);
  Close(f);
end;

procedure Afisare;
  var i,k:byte;
begin
  for k:=1 to nrsol do
  begin
    for i:=1 to m do
    begin
      write(solutii[k,i],',');
    end;
    writeln;
  end
end;

function LungimeData(k:byte):boolean;
begin
  LungimeData:=k=m;
end;

function final(k:byte):boolean;
begin
  final:=LungimeData(k);
end;

function Continuare(pozitie:integer; element:Byte):boolean;
begin
  Continuare:=true;
end;

procedure back(k:integer);
  var element:Byte;
y0:byte;
```

```
begin
  if final(k-1)
  then
  begin
    inc(nrsol);
    solutii[nrsol]:=v;
  end
  else begin
  for y0:=v[k]+1 to n do
  begin
  element:=y0;
  if Continuare(k,y0)
  then begin
       v[k]:=element;
       back(k+1);
     end;
end;
  end
end;

procedure Initializeaza;
begin
  nrsol:=0;
  v[0]:=0;
end;

begin
  Citire;
  Initializeaza;
  back(1);
  Afisare;
end.
```

3 Problema labirintului: se cere să se ajungă din camera A în camera B printr-un drum de lungime minimă.

Pentru această problemă trebuie specificate:

- ✓ Ca date de intrare ce se citesc din fişierul `labirint.in` avem dimensiunile labirintului (`n,m`), structura labirintului sub forma unei matrici cu 0 şi 1 (0 pentru cameră liberă şi 1 pentru perete), poziţia A (de plecare), poziţia B (de final).
- ✓ Domeniul de bază al elementelor vectorului soluţie este de tip înregistrare de doi întregi reprezentând coordonatele pe `x`, respectiv `y` a unei poziţii din labirint. Tipul acesta l-am denumit `poz.`
- ✓ Se iniţializează prima poziţie a vectorului soluţie cu poziţia A.
- ✓ Condiţia de final este ca să ajungem în poziţia B.
- ✓ Lungimea totală a drumului să fie minimă.
- ✓ Drumul se construieşte plecând de la ultima poziţie parcursă (`x,y`) şi mutând în una din poziţiile (`x+1,y-1`), (`x+1,y+1`), (`x-1,y+1`), (`x-1,y-1`).
- ✓ Domeniul de definiţie al unei poziţii este `1..n x 1..m.`

✓ Se face o testare de apartenenţă a unei poziţii la domeniul de definiţie al unui element.

```
{Autor:  }
Program Backtracking;

uses crt;

type poz=record x:byte; y:byte; end;
     TSolutie=array[0..20] of pozitie;

var v:TSolutie;
    solutie:TSolutie;
    lungime:byte;
    n:Byte;
    m:Byte;
    a:array[1..n,1..m] of byte;
    ix:Byte;
    iy:Byte;
    fx:Byte;
    fy:Byte;
    min:integer;
procedure Citire;
  var f:text;
    i1,    i2:byte;
begin
  assign(f,'labirint.in');
  reset(f);
  readln(f,n);
  readln(f,m);
  for i1:=1 to n do
  for i2:=1 to m do
    readln(f,a[i1,i2]);
  readln(f,ix);
  readln(f,iy);
  readln(f,fx);
  readln(f,fy);
  Close(f);
end;

procedure Afisare;
  var i,k:byte;
begin
  for i:=1 to min do
  begin
    write('(');
    write(solutie[i].x,',');
    write(solutie[i].y,',');
  writeln(')');
end;
end;

function UltimaPozitie(k:byte):boolean;
begin
UltimaPozitie:=(v[k].x=fx) and(v[k].y=fy);
end;

function final(k:byte):boolean;
begin
  final:= (a[v[k].x,v[k].y]=0) and UltimaPozitie(k);
end;
```

```
function Continuare(pozitie:integer; element:pozitie):boolean;
  var c:byte;
begin
  Continuare:=true;
  for c:=1 to pozitie-1 do
    if (v[c].x=element.x) and(v[c].y=element.y)
    then begin
           Continuare:=false;
           exit;
         end;
end;

procedure back(k:integer);
  var element:pozitie;
begin
  if final(k-1)
  then
  if k<min
  then begin
         min:=k-1;
         solutie:=v;
       end
  else
  else begin
  element.x:=v[k-1].x+1;
  element.y:=v[k-1].y+1;
  if Continuare(k,element)
  then begin
         v[k]:=element;
         back(k+1);
       end;
  element.x:=v[k-1].x+1;
  element.y:=v[k-1].y-1;
  if Continuare(k,element)
  then begin
         v[k]:=element;
         back(k+1);
       end;
  element.x:=v[k-1].x-1;
  element.y:=v[k-1].y+1;
  if Continuare(k,element)
  then begin
         v[k]:=element;
         back(k+1);
       end;
  element.x:=v[k-1].x-1;
  element.y:=v[k-1].y-1;
  if Continuare(k,element)
  then begin
         v[k]:=element;
         back(k+1);
       end;
  end
end;

procedure Initializeaza;
begin
  v[1].x:=ix;
  v[1].y:=iy;
  min:=10000;
```

```
end;

begin
  Citire;
  Initializeaza;
  back(2);
  Afisare;
end.
```

4. Restricţii şi dezavantaje

Restricţiile listate aici se datorează în cea mai mare măsură trecerii de la scrierea de mână a unui program, la proiectarea lui asistată. Aceeaşi problemă, apare spre exemplu la orice mediu vizual, deoarece componentele furnizate nu pot acoperi infinita diversitate necesară elaborării unei aplicaţii complexe. Oricum, ca şi în cazul acestor medii vizuale, există, şi în aplicaţia mea, posibilitatea inserării de cod sursă, printre codul sursă generat.

Să analizăm câteva din limitările existente:

La ieşire se pot afişa doar părţi din vectorul soluţie, sau doar câteva din soluţii (eventual toate dacă există), dar nu se poate afişa ceva combinaţie între elementele vectorului soluţie. Pentru aceasta este nevoie ca programatorul să intervină în cadrul codului sursă generat, mai precis trebuie să intervină în cadrul procedurii `Afiseaza`.

La iniţializare nu se pot da valori decât primei poziţii din soluţie şi eventual poziţiei 0. Pentru a elimina acest neajuns trebuie ca programatorul să intervină în cadrul procedurii Initializează, iar apoi să schimbe şi apelul `back`(1) sau `back`(2) în `back`(k), unde `k-1` este numărul poziţiilor iniţializate.

La tipul elementelor vectorului nu se ştie clar care este relaţia de ordine de exemplu pentru două mulţimi, sau pentru două şiruri de caractere.

Condiţiile de continuare sunt dintre cele mai variate. În majoritatea cazurilor ele vor trebui implementate de către utilizatorul programului.

Construirea vectorului va trebui, din nou, în multe din cazuri implementată de către programator.

3. METODA GREEDY

1. Prezentarea teoretică

Greedy, în limba engleză înseamnă lacom. Un om este etichetat ca fiind lacom, dacă acumulează bunuri fără a se gândi la viitor, urmărind doar profitul imediat. Această definiţie se poate transpune şi la metoda de programare greedy, şi anume: Un algoritm este de tip greedy, dacă îndeplineşte următoarele condiţii:

- ✓ soluţia este construită pas cu pas,
- ✓ se lucrează cu două mulţimi de elemente:
  - ♦ mulţimea elementelor care constituie deja o soluţie parţială (să o denumim SOLUTIE) şi
  - ♦ mulţimea elementelor candidate la solutie (pe aceasta să o denumim CANDIDATE)
- ✓ la fiecare pas se încearcă adăugarea unui element (sau a mai multor elemente) din mulţimea CANDIDATE în mulţimea SOLUTIE. Un element adăugat nu va mai fi scos din mulţimea SOLUTIE şi el va face parte din soluţia finală. O restricţie generală (impusă de aplicaţia mea) referitoare la elementele din cele două mulţimi este că acestea trebuie să fie numere întregi. Oricum acest lucru nu afectează generalitatea problemei, deoarece în cazul în care avem de a face cu obiecte mai complexe, le vom pune într-un vector şi vom lucra cu indici.

Metoda Greedy se aplică de obicei unor probleme de optimizare (spre exemplu găsirea celui mai scurt drum într-un graf, sau planificarea unor lucrări...). În aceste condiţii, apare necesitatea introducerii unei funcţii de optimizare, pe care o să o denumim funcţie *obiectiv*. Ea va da valoarea finală unei soluţii (spre exemplu această funcţie poate fi lungimea minimă a unui drum, sau numărul minim de monede necesare plăţii unei sume date...), şi reprezintă elementul care pune cel mai bine în evidenţă noţiunea de greedy (lacom). Am spus că o soluţie este întotdeauna construită pas cu pas, în fiecare moment adăugându-se la soluţia parţială un nou element (sau elemente). Datorită funcţiei obiectiv, la fiecare pas va fi adăugat în mulţimea SOLUTIE, cel mai promiţător element aflat în mulţimea CANDIDATE, prin promiţător putându-se înţelege cel care are valoarea cea mai mică, sau cea mai mare, sau cel

care are o valoare dată,... etc. Faptul că alegem, la fiecare pas, cel mai bun element nu înseamnă că şi soluţia finală va fi cea mai bună (comparativ cu celelalte soluţii posibile) şi de aceea algoritmii de tip greedy sunt folosiţi în multe situaţii pe post de algoritmi euristici (amintesc aici problema drumului hamiltonian într-un graf complet şi problema colorării nodurilor unui graf folosind un număr minim de culori).

Pentru alegerea celui mai promiţător element din mulţimea CANDIDATE avem nevoie de o funcţie care să returneze valoarea unui obiect. Întotdeauna valoarea unui obiect trebuie să fie întreagă sau reală. Nici acest lucru nu afectează generalitatea problemei.

Există cazuri în care valoarea unui obiect nu are nici o importanţă, deci obiectele pot fi considerate ca fiind de valori egale cu 1 (unu).

De asemenea există cazuri în care valoarea unui element(din mulţimea CANDIDATE) nu este fixă şi se modifică la fiecare pas. Spre exemplu, la algoritmul lui Dijkstra pentru determinarea celor mai scurte drumuri dintr-un vârf spre toate celelalte vârfuri, valoarea unui nod *k* va fi iniţial:

$$v[k] = \begin{cases} L(sursa, k) & daca\ exista\ arc\ de\ la\ sursa\ la\ k \\ \infty & daca\ nu\ exista\ arc\ de\ la\ sursa\ la\ k \end{cases}$$

Apoi toate nodurile *k* din CANDIDATE vor fi reactualizate după fiecare adăugare a unui nod *w* (de valoare minimă) astfel:

v[k] = min(v[k], v[w]+L[w,k])

Soluţia optimă (obţinută printr-un algoritm greedy) nu este unică, asta din cauză că la un moment dat mulţimea elementelor cele mai promiţătoare este formată din mai multe elemente (cel puţin unul). Alegerea unuia dintre acestea va duce la generarea unei soluţii.

Aceste lucruri fiind spuse se poate trece la prezentarea unui algoritm general, care va fi îmbunătăţit pe parcurs.

```
SOLUTIE := ∅
while CANDIDATE <> ∅ do
begin
  Alege cel mai promiţător element din CANDIDATE  {fie x acest element}
  SOLUTIE := SOLUTIE + {x}
  CANDIDATE := CANDIDATE - {x}
end
```

O ultimă observaţie mai trebuie făcută. Mulţimea SOLUTIE (în care se construieste o solutie) nu este neapărat o mulţime, asta din cauză că la unele probleme ne interesează şi ordinea selectării elementelor din soluţie. De aceea reprezentarea mulţimii SOLUTIE se face prin intermediul vectorului SOLUTIE. El poate fi:

- ✓ vector de întregi, dacă la fiecare pas alegem un singur element din CANDIDATI,
- ✓ vector de vectori de întregi, dacă la fiecare pas alegem un număr fix de elemente din CANDIDATI pe care le introducem în SOLUTIE. Reprezentativă pentru această situaţie este problema *generării ordinii de interclasare a n şiruri ordonate crescător*, care va fi prezentată mai jos,
- ✓ vector de multimi de întregi, dacă la fiecare pas alegem un număr oarecare de elemente şi le introducem în SOLUTIE.. Reprezentativă pentru această situaţie este problema colorării nodurilor grafului cu un număr minim de culori, care de asemenea va fi prezentată mai jos.

2. Implementarea ei în aplicaţie

Trebuie urmărite câteva elemente definitorii pentru metoda greedy, iar în rest totul nu depinde decât de preferinţele programatorului, adică în cazul de faţă ale mele. Printre elementele strict personale ale metodei greedy enumăr: valoarea unui obiect, funcţia de alegere, functia de final, construirea soluţiei. Mai există încâ câteva elemente care trebuie specificate, dar din cauză că sunt comune tuturor metodelor de programare le-am expus în capitolele precedente. Acestea ar fi datele de intrare, datele de ieşire, iniţializarea, finalizarea, unde se stochează soluţia. Totuşi despre datele de ieşire şi despre iniţializare am să vorbesc şi în acest capitol, deoarece ele conţin elemente specifice acestei metode.

2.1. Valoarea unui obiect

Valoarea unui obiect poate fi:

- ✓ fixă pe toată durata execuţiei algoritmului,
- ✓ variază după fiecare alegere a unui element din MULTIMEA CANDIDATI
- ✓ nu contează (în acest caz se consideră că obiectele au toate valoarea egală cu 1)

2.2. Iniţializare

Se iniţializează mulţimea CANDIDATI cu valoarea [1..NRCandidati] şi *NrSolutie* (care reprezintă numărul paşilor necesari pentru a obţine o soluţie) cu 0.

2.3. Conditii de final

La final mulţimea CANDIDATI trebuie să fie vidă, iar ceea ce obţinem în vectorul SOLUTIE trebuie să constituie o soluţie a problemei. Există situaţii când mulţimea de CANDIDATI nu este vidă, dar din ea nu se mai pot face alegeri. În acest caz se testează dacă ceea ce avem în SOLUTIE poate fi admisă ca fiind o soluţie corectă a problemei şi în caz afirmativ o afişăm.

2.4. Functia de alegere

Funcţia de alegere este funcţia care ne indică elementul cel mai promiţător de la fiecare pas. Avem de optat între 3 variante:

- la fiecare pas alegem elementul a cărui valoare este minimă,
- la fiecare pas alegem elementul a cărui valoare este maximă,
- la fiecare pas alegem elementul a cărui valoare este dată de utilizator.

2.5. Construirea solutiei

În cadrul acestui modul trebuie specificate câte elemente se aleg la un pas din CANDIDATI şi câte elemente se introduc la un pas în CANDIDATI. Un exemplu în care este necesară actualizarea mulţimii CANDIDATI problema ordinii de interclasare a n şiruri ordonate crescător, care este descrisă puţin mai jos. Referitor la operaţia de actualizare trebuie specificat faptul că la nici un caz nu vom adăuga în CANDIDATI un element care deja a fost introdus în SOLUTIE, ci doar eventuale elemente noi (probabil rezultate prin combinarea altor elemente).

Revenind la numărul de elemente care se poate extrage la un moment dat din CANDIDATI, avem următoarele situaţii:

- ✓ putem extrage doar 1 element (este cazul problemei *găsirii arborelui parţial de cost minim* folosind algoritmii lui Kruskal sau Prim. În acest caz la fiecare pas extragem muchia de cost minim, care îndeplineşte nişte condiţii...)
- ✓ putem extrage un număr fix de elemente (este cazul problemei *ordinii de interclasare a n şiruri ordonate crescător* când la fiecare pas extragem din CANDIDATI două cele mai mici lungimi de şiruri. Prin combinarea acestor două şiruri va rezulta un nou şir care va fi introdus în mulţimea CANDIDATI)
- ✓ putem extrage oricâte elemente (este cazul problemei *colorării cu număr minim de culori a nodurilor unui graf*, caz în care la fiecare pas extragem toate nodurile pe care le putem colora cu aceeaşi culoare).

2.6. Date de intrare

Întotdeauna pe prima linie a fişierului de intrare trebuie specificată mărimea iniţială a mulţimii CANDIDATI.

2.7. Afişare

Se pot cere următoarele lucruri:

- ✓ afişarea ordinii selectării obiectelor
- ✓ afişarea numărului de paşi după care se termină algoritmul (în cazul problemei colorării nodurilor grafului este vorba chiar de numărul minim de culori necesare)
- ✓ afişarea noilor valori ale obiectelor (în cazul problemei drumului minim folosind algoritmul lui Dijkstra, avem că noile valori ale nodurilor vor fi chiar lungimile drumului minim de la sursă la acel vârf).

3. Exemple de utilizare a aplicaţiei

1. Minimizarea timpului de aşteptare

Enunţ: O staţie PECO trebuie să satisfacă cererile a *n* clienţi. Timpul deservirii clientului *i* este t_i . Se cere să se determine o ordine de servire astfel încât să se minimizeze timpul total de aşteptare.

Rezolvare. Obiectele în acest caz constituie clienţii, fiecare client având valoarea t_i. La fiecare pas alegem clientul neselectat cu t_i minim. Pentru aceasta vom specifica:

- ✓ de la intrare se citeşte numărul clienţilor (variabila *NrCandidate*) şi timpii de aşteptare (variabila t)
- ✓ ca date de ieşire dorim ordinea de selectare
- ✓ valoarea obiectului *i* este t_i
- ✓ la fiecare pas se alege elementul de valoare minimă

Codul sursă generat pe baza acestor specificaţii este următorul:

```
Program Greedy;
uses crt;
const
  MaxN=100;
var
    NrSolutie:byte;
    Candidate : set of 1..MaxN;
    Solutie : array[1..MaxN] of byte;
    NrCandidate:byte;
    t:array[1..10] of byte;

procedure Citire;
  var f:text;
    i1:byte;
begin
  assign(f,'peco.in');
  reset(f);
  readln(f,NrCandidate);
  for i1:=1 to 10 do
    readln(f,t[i1]);
  Close(f);
end;


  procedure Afiseaza;
    var i,k:byte;
  begin
    for i := 1 to NrSolutie do
      writeln(Solutie[i]);
  end;
```

```
  function Valoare(i:byte):real;
  begin
    Valoare := t[i];
  end;

  procedure AlegeObiect(var ob:byte);
    var i,nrm:byte;
        m : real;
  begin
    m := 9999999; nrm := 0;
    for i := 1 to NrCandidate do
    begin
      if (i in Candidate) and (m > Valoare(i))
      then begin
             nrm := i; m := Valoare(i);
           end;
    end;
  Candidate := Candidate - [nrm];
  ob := nrm;
  end;

  procedure Initializeaza;
  begin
    Candidate := [1..NrCandidate];
    NrSolutie := 0;
  end;

begin {programul principal}
  Citeste;
  Initializeaza;
  while Candidate <> [] do
  begin
    inc(NrSolutie);
    AlegeObiect(Solutie[NrSolutie]);
  end;
end.
```

2 Minimizarea timpului de acces

Enunţ: Pe o bandă magnetică se află n programe, un program i fiind de lungime l_i şi este apelat cu probabilitatea p_i . Accesul la un program se face secvenţial. Se cere ordinea de memorare a programelor pe bandă, astfel încât să se minimizeze timpul mediu de citire a unui program.

Rezolvare. Soluţia este asemănătoare cu cea de la problema precedentă, modificându-se două lucruri: valoare unui obiect, care va fi în acest caz p_i/l_i şi alegerea obiectelor care se va face în ordine descrescătoare. Deci vor trebui specificate următoarele lucruri:

- ✓ de la intrare se citeşte numărul programelor (variabila *NrCandidate*) şi lungimile respectiv probabilităţile de acces ale programelor (variabilele *p* şi *l*)
- ✓ ca date de ieşire dorim ordinea de selectare
- ✓ valoarea obiectului *i* este p_i/l_i
- ✓ la fiecare pas se alege elementul de valoare maximă

Codul sursă generat pe baza acestor specificaţii este următorul:

```
Program Greedy;
uses crt;
const
  MaxN=100;
var
    NrSolutie:byte;
    Candidate : set of 1..MaxN;
    Solutie : array[1..MaxN] of byte;
    NrCandidate:byte;
    p:array[1..10] of real;
    l:array[1..10] of byte;

procedure Citire;
  var f:text;
    i1:byte;
begin
  assign(f,'peco.in');  reset(f);
  readln(f,NrCandidate);
  for i1:=1 to 10 do
    readln(f,p[i1]);
  for i1:=1 to 10 do
    readln(f,l[i1]);
  Close(f);
end;
```

```
 procedure Afiseaza;
  var i,k:byte;
 begin
  for i := 1 to NrSolutie do
    writeln(Solutie[i]);
 end;

 function Valoare(i:byte):real;
 begin
  Valoare := p[i]/l[i];
 end;

 procedure AlegeObiect(var ob:byte);
  var i,nrm:byte;
      m : real;
 begin
  m := 0; nrm := 0;
  for i := 1 to NrCandidate do
  begin
   if (i in Candidate) and (m < Valoare(i))
   then begin
          nrm := i; m := Valoare(i);
        end;
  end;
 Candidate := Candidate - [nrm];
 ob := nrm;
 end;

 procedure Initializeaza;
 begin
  Candidate := [1..NrCandidate];
  NrSolutie := 0;
 end;

begin {programul principal}
 Citeste;
 Initializeaza;
 while Candidate <> [] do
 begin
```

```
    inc(NrSolutie);
    AlegeObiect(Solutie[NrSolutie]);
  end;
end.
```

3. Interclasarea optimă a n şiruri

Rezolvare: Pentru simplificare vom lucra doar cu lungimile şirurilor de interclasat. Algoritmul pentru rezolvarea acestei probleme constă în interclasarea la fiecare pas a celor mai scurte două şiruri. Evident că după interclasarea a două şiruri se va obţine un nou şir al cărui lungime trebuie introdusă în mulţimea CANDIDATE. Deci trebuie specificate următoarele lucruri:

- ✓ ca date de intrare avem lungimile celor n şiruri,
- ✓ ca date de ieşire avem ordinea de interclasare,
- ✓ valoarea unui obiect este lungimea lui,
- ✓ la fiecare pas alegem două obiecte cu valorile cele mai mici,
- ✓ obiectul având ca valoare suma celor două alese se va introduce în CANDIDATE.

Codul sursă generat va fi următorul (la el mai trebuie inserat cod pentru adăugarea unei noi lungimi în vectorul cu lungimile şirurilor):

```
Program Greedy;
uses crt;
const
  MaxN=100;
var
    NrSolutie:byte;
    Candidate : set of 1..MaxN;
    Solutie : array[1..MaxN,1..2] of byte;
    NrCandidate:byte;
    lungimi:array[1..10] of byte;

procedure Citire;
  var f:text;
    i1:byte;
begin
  assign(f,'inter.in');
  reset(f);
  readln(f,NrCandidate);
  for i1:=1 to 10 do
```

```
    readln(f,lungimi[i1]);
  Close(f);
end;


  procedure Afiseaza;
    var i,k:byte;
  begin
    for i := 1 to NrSolutie do
    begin
      for k := 1 to 2 do
        write(Solutie[i][k],",");
      writeln;
    end;
  end;

  function Valoare(i:byte):real;
  begin
    Valoare := lungime[i];
  end;

  procedure AlegeObiect(var ob:byte);
    var i,nrm:byte;
        m : real;
  begin
    m := 9999999; nrm := 0;
    for i := 1 to NrCandidate do
    begin
      if (i in Candidate) and (m > Valoare(i))
      then begin
             nrm := i; m := Valoare(i);
           end;
    end;
  Candidate := Candidate - [nrm];
  ob := nrm;
  end;

  procedure MutaInapoi;
  begin
    inc(NrCandidati);
```

```
    Candidati := Candidati + [NrCandidati];
    Lungimi[NrCandidati] := lungimi[NrSolutie][1]+lungimi[NrSolutie][2];
  end;

  procedure Initializeaza;
  begin
    Candidate := [1..NrCandidate];
    NrSolutie := 0;
  end;

begin {programul principal}
  Citeste;
  Initializeaza;
  while Candidate <> [] do
  begin
    inc(NrSolutie);
    for k := 1 to 2 do
    begin
      AlegeObiect(Solutie[NrSolutie][k]);
    end;
    MutaInapoi;
  end;
end.
```

4. Coloarea nodurilor unui graf cu număr minim de culori

Rezolvare: Algoritmul folosit este evident euristic, şi constă în colorarea la fiecare pas a unui număr de noduri cu o culoare fixată, apoi procedeul repetându-se, cu o altă culoare, pentru nodurile rămase necolorate. Deci noi la fiecare pas trebuie să introducem în SOLUTIE o mulţime de noduri care vor avea aceeaşi culoare. Specificaţiile sunt următoarele:

- ✓ ca date de intrare avem o matrice *a* în care *a[i,j]* este 1 dacă nodurile *i* şi *j* sunt legate printr-o muchie, şi 0 în caz contrar.
- ✓ ca date de ieşire avem numărul de paşi după care se termină algoritmul (adică numărul minim de culori folosite)
- ✓ Valoarea unui obiect nu contează
- ✓ la fiecare pas se alege o submulţime de noduri cu proprietatea că oricare două dintre ele nu sunt unite printr-o muchie (deci submulţime intern stabilă). Implementarea acestei proceduri trebuie făcută de către utilizator.

Codul sursă generat este:

```
Program Greedy;

uses crt;

const
  MaxN=100;
  type multime=set of 1..1+MaxN div 8;
var
    NrSolutie,k:byte;
    Candidate : set of 1..MaxN;
    Solutie : array[1..MaxN] of multime;
    NrCandidate:byte;
    a:array[1..10,1..10] of byte;

procedure Citire;
  var f:text;
    i1,    i2:byte;
begin
  assign(f,'color.in');
  reset(f);
  readln(f,NrCandidate);
  for i1:=1 to 10 do
  for i2:=1 to 10 do
    readln(f,a[i1,i2]);
  Close(f);
end;


  procedure Afiseaza;
    var i,k:byte;
  begin
    writeln(NrSolutie);
  end;

  procedure AlegeObiect(var ob:multime);
    var i,j:byte;
  begin
    for i:= 1 to NrCandidate do
```

```
  begin
    if i in Candidate
    then begin
              ok:=true;
           for j := 1 to i-1 do
            if j in ob
            then if a[i,j] = 1
                    then ok := false;
           end;
     if ok then ob := ob + [i];
   end;
  Candidate := Candidate - ob;
end;

procedure Initializeaza;
begin
  Candidate := [1..NrCandidate];
  NrSolutie := 0;
end;

begin {programul principal}
  Citeste;
  Initializeaza;
  while Candidate <> [] do
  begin
    inc(NrSolutie);
    AlegeObiect(Solutie[NrSolutie]);
  end;

end.
```

4. Metoda Branch and Bound

1. Descrierea metodei

Metoda **Branch** and **Bound** (**Ramifică** şi **mărgineşte**) este înrudită cu metoda **backtracking**, diferenţele constau în ordinea de parcurgere a arborelui spaţiului stărilor şi modul în care se elimină subarborii care nu pot duce la rezultat.

Pentru început să stabilesc cadrul în jurul căruia se va desfăşura explicaţia: Avem o stare iniţială (să o numim s_0) şi o stare finală (să o numim s_t). Dintr-o stare se poate trece într-o altă stare prin luarea unei decizii. Este posibil ca dintr-o stare să se poată lua mai multe decizii, ceea ce implică faptul că dintr-o stare se pot trece în mai multe alte stări. Se cere, dacă există, şirul de decizii care trebuie luate pentru a se trece din starea iniţială în starea finală.

Pentru a simplifica problema vom presupune din start că există întotdeauna soluţie, în caz contrar aplicarea acestei metode va duce doar la epuizarea resurselor calculatorului.

Cu alte cuvinte avem acelaşi enunţ de problemă ca şi la backtracking, dar în schimb, aici deciziile nu sunt supuse unor condiţii de continuare.

După cum reiese din cele spuse mai sus, trebuie să construim un graf ale cărui noduri sunt stări, iar muchiile (mai precis arcele care sunt orientate de la un nod tată $s_{tată}$ spre un nod fiu s_{fiu} (indicând faptul că decizia respectivă a transformat starea $s_{tată}$ în starea s_{fiu})) reprezintă deciziile. Dacă permitem ca în graf să există noduri ce reprezintă aceeaşi stare, vom avea de a face cu un graf infinit (din punct de vedere al numărului nodurilor). Dacă, în schimb, vom introduce restricţia ca un nod să apară o singură dată, atunci graful se va transforma într-un arbore (deoarece într-un nod nu vor ajunge niciodată două arce, căci în momentul construirii arborelui, construire care se va face nod cu nod, nu vom lua niciodată o decizie care să ne ducă într-o stare deja existentă). Generarea arborelui se face până la prima apariţie a nodului final, căci în majoritatea cazurilor cardinalul spaţiului stărilor depăşeşte cu mult posibilităţile unui calculator. Să ne gândim doar la problema jocului căsuţelor **Perspicco** a cărui număr de stări este de 16! = 20.922.789.888.000.

Să analizăm puţin două dintre modalităţile principale de rezolvare a problemei, adică cu alte cuvinte pricipalele modalităţi de construire (parcurgere) ale arborelui stărilor:

1. Parcurgerea **DF** (**Depth-first**) este, aşa cum spune şi titlul o parcurgere în adâncime a grafului (arborelui) stărilor. Procedura **DF**(stare) se aplică în următorul fel: Se pleacă din starea iniţială $\mathbf{s_0}$ şi se apelează **DF**($\mathbf{s_0}$). Vor rezulta nodurile fii ale nodului pentru care s-a apelat procedura. Pentru fiecare nod rezultat (şi care nu a fost încă vizitat) se apelează iarăşi **DF**. Ne oprim în momentul în care vizităm nodul ce conţine starea finală. Dezavantajul principal al acestei metode este faptul că nu va da întotdeauna soluţia cu număr minim de decizii. Acest lucru este remediat de către parcurgerea **BF**.

2. Parcurgerea **BF** (**Breadth- first**) este, aşa cum spune şi titlul o parcurgere în lăţime a arborelui (grafului) stărilor. Pentru acest algoritm construim un şir de mulţimi de stări. În prima mulţime vom introduce doar starea iniţială. În a doua mulţime vom introduce stările fii ale stării iniţiale. Apoi la fiecare pas în mulţimea k vom introduce stările fii ale stărilor din mulţimea k-1. Ne vom opri în momentul în care vom genera mulţimea ce conţine starea finală. Prin această metodă vom avea întotdeauna soluţia optimă (în sensul de număr minim de decizii), dar în schimb trebuie parcurse destul de multe stări pentru a ajunge la final.

De fapt ambele metode (şi **DF** şi **BF**) au acest mare dezavantaj: parcurg multe stări suplimentare, care nu sunt necesare. Putem spune, fără teama de a greşi, că spaţiul stărilor este parcurs *orbeşte*, fără a se ţine cont de starea unde trebuie să ajungem. Scopul principal al acestei metode de programare este tocmai acesta, de a ne ghida cât mai mult către starea finală. În acest fel, multe dintre stările care ne îndepărtează de scopul final vor fi eliminate (sau cel puţin vor fi luate în considerare mai târziu).

Aceste considerente conduc la introducerea unei funcţii de cost a unei stări. Această funcţie trebuie să respecte următoarele două condiţii:

- ✓ costul stării finale trebuie să fie minim (în raport cu celelalte stări),
- ✓ costurile stărilor, care se află pe un drum minim (de lungime minimă) de la starea iniţială la starea finală, trebuie să fie descrescătoare.

Este evident că, în mod ideal, la fiecare pas am alege decizia care ne duce într-o stare de cost mai mic. Însă, în cazul real, câteva dificultăţi apar. Personal, văd două cauze principale pentru care această funcţie nu este optimă (în sensul că nu duce direct la soluţie):

- pentru o valoare dată a costului există mai multe stări care au aceeaşi valoare,

- dintr-o stare nu putem trece întotdeauna într-o alta pentru costul este mai mic (în caz contrar am fi avut de-a face cu greedy).

Există mai multe modalităţi pentru a calcula costul unei configuraţii. Îm modul ideal acestă funcţie de cost este astfel definită:

cost (s_{final}) := 0;
cost (s) := 1 + min {cost (v) | unde v este deja calculat şi există o decizie care trece s în v }.

Dar, din păcate, această funcţie nu o putem cunoaşte decât după ce am construit deja soluţia. De aceea noi vom lucra cu nişte aproximări euristice ale costului. Denumirea de euristice vine de la faptul că vor fi explorate şi stări care nu conduc la soluţia optimă (sau care nu duc la nici un fel de soluţie).

Mai întâi avem nevoie de o altă noţiune foarte importantă, dacă este aleasă bine, poate duce la scăderea drastică a numărului de configuraţii vizitate. Este vorba de *distanţa* dintre două stări. Şi această funcţie trebuie să îndeplinească câteva condiţii:

- ✓ Distanţa (A,A) := 0,
- ✓ Distanţa (A, s_{final}) >= Distanţa (B, s_{final}), dacă există o decizie care trece A în B.

Funcţia distanţă nu poate fi generalizată pentru orice problemă, adică nu există o formulă comună a ei pentru toate situaţiile. De alegerea ei, depinde foarte mult numărul de configuraţii pe care le vom vizita până ajungem la configuraţia finală. Aplicaţia mea, prevede totuşi o posibilitate independentă de tipul unei stări, şi anume consideră reprezentarea în memorie a unei stări, ca fiind o succesiune de octeţi (lucru care în practică chiar aşa şi este) şi calculează numărul de poziţii prin care cele două zone de memorie diferă. Există cazuri când această modalitate este chiar destul de bună, spre exemplu la jocul **Perspicco** , în care o stare este reprezentată sub forma unei matrici 4x4 (cu elemente n mulţimea 0..15). Dacă, în schimb avem stările ca fiind mulţimi, compararea ar trebui să se facă la nivel de bit şi nu de octet, deci folosind distanţa aceasta nu s-ar efectua un calcul destul de bun.

Revenind la cost, dintre euristicele mai des întâlnite aş aminti două:

1) cost := Nivelul curent + distanta dintre configuratia curenta si configuratia finala
2) cost := Costul starii parinte + distanta dintre configuratia curenta si configuratia finala

(prin nivel se înţelege numărul mutărilor prin care s-a ajuns la configuraţia curentă)

Menţionez, că totuşi, această ceea de a doua modalitate de cost duce mai mult la o parcurgere în lăţime a arborelui stărilor. În unele situaţii limită, în care este dificil implementarea unei funcţii de distanţă, se poate specifica ca o configuraţie să aibă un cost fix. Aici prin fix se înţelege independent de alte stări, dar dependent de nişte variabile legate efectiv de stare.

Acum, că am terminat cu explicarea noţiunii de cost, putem trece la definirea metodei şi a principalilor ei paşi. Pentru simplificare, vom lucra cu varianta standard a problemei, adică: *se dă o stare iniţială, o stare finală şi se cer deciziile*.

Structura unui element al listei cu stările are câmpurile

inf - în care se reţine o stare,

cost - în care se reţine costul unei stări,

urm, pred - legăturile spre următoarea stare din listă, respectiv spre starea părinte.

```
Crează_capul_listei {adică initializeaza câmpul inf cu starea initiala, }
                    { şi costul conform formulei definite               }
                    { va rezulta variabila cap                          }
while (cap <> StareaFinală) do
begin
   Expandează(cap)    { adică aplică lui cap toate deciziile posibile   }
                      { pentru fiecare stare rezultată din expandare }
                      {îi calculez costul şi o adaug unde trebuie      }
end
```

2. Implementarea ei în aplicaţie

Trebuie urmărite câteva elemente definitorii pentru metoda **branch and bound**, iar în rest totul nu depinde decât de preferinţele programatorului, adică în cazul de faţă ale mele. Printre elementele strict personale ale metodei enumăr: tipul unei stări, tipul elementelor listei, iniţializarea, condiţii de final, distanţa, funcţia de cost, construirea soluţiei.

Mai există încâ câteva elemente care trebuie specificate, dar din cauză că sunt comune tuturor metodelor de programare le-am expus în capitolul precedent. Acestea ar fi datele de

intrare, datele de ieşire, iniţializarea, finalizarea, unde se stochează soluţia. Totuşi despre datele de ieşire şi despre iniţializare am să vorbesc şi în acest capitol, deoarece ele conţin elemente specifice acestei metode.

2.1 Tipul unei stări

O stare poate fi de orice tip, de la cele de bază, până la cele definite de utilizator. În programul generat tipul unei stări (configuraţii) va fi denumit *TStare*.

2.2 Tipul elementelor listei

Este diferit de tipul unei stări, din cauză necesităţiilor impuse de programarea într-un limbaj anume, şi nu numai. Structura generală a unui element al *listei soluţie* va avea următoarea formă generală (care apoi va fi ajustată în funcţie de cerinţe):

```
type TLista = ^art;
     art = record
             inf : TStare;
             cost : integer;
             urm : TLista;
           end;
```

Să vedem acum care ar fi variantele acestui tip:

În primul rând legat de câmpul *inf* se pot face câteva precizări. În mod ideal acest câmp ar trebui să reţină exact o stare. Dar datorită faptului că uneori reprezentarea unei stări poate ocupa o cantitate apreciabilă de memorie (de exemplu o stare poate fi o matrice de 100 de poziţii), se recurge la alte modalităţi de reţinere. De obicei se încearcă codificarea stării respective, cu condiţia să se poată asigura un cod unic pentru fiecare configuraţie. De exemplu o matrice de 3x3 cu elementele în mulţimea 0,1 poate fi uşor codificată cu numerele din mulţimea {0,...,512}. Există cazuri în care nu ne interesează să ajungem la final într-o stare anume, ci dorim să ajungem într-o stare care îndeplineşte o condiţie anume. Asta înseamnă că mulţimea de configuraţii se va putea împărţi în clase; codificăm fiecare astfel de clasă şi aşa nu mai suntem nevoiţi să reţinem întreaga stare.

Cum am spus şi mai sus, este posibil ca reţinerea întregii stări să fie o mare consumatoare de memorie. Ca să evităm acest lucru ne legăm de decizii. Şi aici avem două situaţii: putem reţine fie întreg şirul de decizii care a generat starea curentă plecând de la

starea iniţială, fie reţinem la fiecare pas decizia care s-a luat pentru a se trece din starea anterioară în starea curentă. În prima situaţie va fi necesară la fiecare prelucrare a unei stări, o procedură (să o denumim **RefăStare**) care plecând de la şirul deciziilor va genera toate stările intermediare de la starea iniţială până la starea curentă inclusiv). În a doua situaţie mai este necesară în plus o procedură (să o denumim **ReturnSirulMutărilor**) care va trebui să refacă şirul decizilor care au generat starea curentă, iar apoi trebuie să apeleze procedura **RefăStare**. Mai mult decât atât, această ultimă modalitate, implică, necesitatea introducerii, în structura unui element al listei, a unui câmp (să-l numim **pred**) care va face legătura dintre starea curentă şi starea părinte (cea din care starea curentă a fost generată în urma luării unei anumite decizii).

În general, dacă dorim timp bun de execuţie pentru algoritm, atunci vom reţine întreaga stare, iar dacă dorim consum mai mic de memorie, folosim celelalte două modalităţi de stocare a unei stări (adică prin şirul de decizii sau decizia curentă).

2.3 Iniţializarea

De obicei, se iniţializează prima poziţie din listă cu starea iniţială. Există cazuri în care nici măcar starea iniţială nu se dă, ci doar nişte condiţii pe care le îndeplineşte aceasta. În acest caz, trebuie făcută o codificare a acestei stări.

2.4 Condiţii de final

Cea mai des întâlnită condiţie de final este ca ultima stare să fie una definită de utilizator. Şi aici se mai pot face câteva optimizări privitoare la timpul de execuţie, şi anume: nu este nevoie să testăm dacă starea curentă este egală cu starea finală, deoarece, spre exemplu în cazul jocului **Perspicco** ar însemna să comparăm două matrici, ceea ce la un număr foarte mare de stări va duce la încetinirea timpului de execuţie. Pentru a evita acest lucru, ne legăm de câmpul cost (care este doar o variabilă de tip întreg) şi observăm că, o configuraţie finală are acelaşi cost cu configuraţia părinte (în cazul în care lucrăm cu formula cost := *Costul starii parinte + distanta dintre configuratia curenta si configuratia finala*) sau va fi cu o unitate mai mare decât costul configuraţiei părinte în cazul în care lucrăm cu formula (cost := *Nivelul curent + distanta dintre configuratia curenta si configuratia finala*), deci condiţiile de final, în cele dpuă situaţii, se vor pune după cum urmează:

1. cap^.cost = cap^.pred^.cost

2. cap^.cost = cap^.nivel.

2.5 Distanţa

Cum am mai spus şi mai sus, distanţa nu are o formulă generală pentru toate problemele. Aplicaţia mea prevede însă două posibilităţi generale de a compara două stări, şi anume compararea la nivel de octet şi compararea la nivel de bit a reprezentării în memorie a celor două stări. Utilitatea celor două metode depinde foarte mult de problema la care se aplică. Spre exemplu la jocul **Perspicco**, în care avem de a face o matrice de 4x4 de octeţi, prima metodă de comparare este mai bună decât a doua, dar în cazul generării unei mulţimi intern stabilă maximală (şi se foloseşte tipul **set**) cea de a doua metodă este mai bună decât prima.

2.6 Funcţia de cost

Referitor la câmpul **cost** se pot face foarte multe speculaţii. Acest **cost** este de fapt elementul cheie al întregii metode **branch and bound**. Câteva restricţii, datorate implementării, se impun însă: În primul rând, din cauză că lista în care se vor reţine stările parcurse este crescătoare, trebuie ca, costul unei configuraţii să fie mai mare (sau egal) decât costul configuraţiei de provenienţă (tatăl). De aceea începutul unei formule generale ar putea fi dat: cost(configuraţie) := cost (configuraţia tată) + ...

Un alt început de formulă, ar putea fi următorul : cost(configuraţie) := Numărul nivelului (din arborele stărilor) pe care se află starea curentă + ...

Continuarea pentru aceste formule este aceeaşi: ...+ distanţa dintre configuraţia curentă şi configuraţia finală, lucru care se calculează prin intermediul funcţiei **Distanţa**. Trebuie însă făcută o observaţie deosebit de importantă referitor la cea de a doua formulă, şi anume la: *cost(configuraţie) := Numărul nivelului (din arborele stărilor) pe care se află starea curentă + distanţa dintre configuraţia curentă şi configuraţia finală*. Numărul nivelului stării curente este 1 + numărul nivelului stării părinte, şi se poate ca în unele situaţii distanţa dintre configuraţia curentă şi configuraţia părinte să fie cu cel puţin o unitate mai mică decât distanţa dintre configuraţia părinte şi configuraţia bunic (părinte al acesteia), deci costul configuraţiei curente va deveni mai mic sau egal cu costul configuraţiei părinte, ceea ce va duce la adăugarea (în listă) a stării curente înainte de starea părinte, ceea ce mai departe va duce la imposibilitatea analizării acestei configuraţii (deoarece lista cu stările va fi analizată în

ordinea crescătoare a costului, dar fără reveniri, adică întotdeauna se presupune că stările neanalizate încă au cost mai mare sau egal cu configuraţia care se studiază la pasul curentă.

Din fericire această problemă poate fi rezolvată şi rezultatul va duce la îmbunătăţirea simţitoare a vitezei de execuţie. Adăugarea unei stări se făcea în listă întotdeauna de la început (pentru a se garanta monotonia crescătoare), dar dacă adăugarea unei stări se va face nu de la primul element al listei, ci de la starea curentă încolo, atunci nu va mai exista situaţia ca o stare neanalizată (adică neexpandată) să fie pusă înaintea stării curente. Însă acest lucru va duce la pierderea proprietăţii de monotonie crescătoare a listei, dar în schimb adăugarea în listă se va face mult mai repede şi având în vedere că se fac multe adăugări, timpul total de execuţie al algoritmului va scădea.

2.7 Construirea soluţiei

În cadrul acestui modul trebuie specificate numărul deciziilor care se pot lua la un pas. În acest domeniu, aplicaţia mea impune câteva restricţii, şi anume:

- ✓ la fiecare pas se pot aplica aceleaşi decizii (deci luarea unei decizii nu depinde de numărul de ordine al pasului la care se ia)
- ✓ deciziile sunt de tip întreg, în caz contrar ele se vor numerota cu numere întregi consecutive mai mari sau egali cu 1. Motivul acestei restricţii este următorul: În programul generat, pentru fiecare decizie **i** se va apela procedura **Decizie** care va returna starea în care se trece la un moment dat.

2.8 Date de ieşire

La ieşire se poate cere:

- ✓ drumul până la starea finală. Acesta se poate afişa sub formă de succesiune de stări, sau sub formă de succesiune de decizii,
- ✓ starea finală,
- ✓ lungimea drumului.

3. Exemple de utilizare ale aplicaţiei

Jocul Perspicco

Jocul necesită o matrice având dimensiunea 4x4, ale cărei celule conţin numerele de la 1 la 15, cu excepţia uneia dintre ele care este liberă (conţine cifra 0). O mutare validă constă în mutarea unui număr dintr-o căsuţă (învecinată pe orizontală sau verticală cu cea liberă) în celula liberă. Căsuţa din care s-a făcut mutarea devine la rândul ei liberă. Se cere să se determine numărul minim de mutări pentru a se trece dintr-o configuraţie iniţială într-una finală.

Pentru rezolvarea acestei probleme utilizatorul trebuie să specifice următoarele lucruri:

- ✓ Datele de intrare se citesc din fişierul persp.in. El conţine două configuraţii (cea iniţială şi cea finală) specificate prin variabilele *wIni,wFin*.
- ✓ La ieşire se doreşte drumul până la configuraţia finală,
- ✓ Tipul unei stări este *TStare = array[1..4,1..4] of byte,*
- ✓ Tipul elementelor soluţiei este conţine *starea curentă,*
- ✓ Se iniţializează capul listei cu *WIni,*
- ✓ Starea de final trebuie să fie *WFin,*
- ✓ Distanta dintre două configuraţii este egală cu numărul de octeţi prin care cele două diferă,
- ✓ Costul este calculat după formula *cost := Nivelul curent + distanta dintre configuratia curenta si configuratia finala,*
- ✓ Numărul deciziilor care se pot lua la un pas este 4 (schimbare cu căsuţa din stânga, dreapta, sus, jos).

Codul sursă generat pe baza specificaţiilor de mai sus este, următorul, la care însă mai trebuie adăugat cod pentru specificarea modului în care acţionează fiecare decizie asupra unei configuraţii.

```
Program BranchAndBound;
uses crt;
type
    TStare = array[1..4,1..4] of byte;
    TLista = ^art;
    art = record
        inf : TStare;
        cost : integer;
```

```
            urm : TLista;
            pred : TLista;
            niv : byte;
          end;

 var
 StareHeap : pointer;
  wIni,wFin:TStare;

procedure Citire;
 var f:text;
      i1,i2:byte;
begin
 assign(f,'persp.in');
 reset(f);
 for i1:= 1 to 4 do
  for i2:=1 to 4 do
    readln(f,wIni[i1,i2]);
 for i1:= 1 to 4 do
  for i2:=1 to 4 do
    readln(f,wFin[i1,i2]);
 Close(f);
end;

 procedure Afisare;
  var
   i:byte;
 begin
 end;

 function CompararеLaNivelDeOctet(st1,st2:TStare):byte;
  var s1,s2:string;
      i,dif : byte;
 begin
  mov(st1,s1,sizeof(TStare));
  mov(st2,s2,sizeof(TStare));
  dif := 0;
  for i := 1 to sizeof(TStare) do
   if s1[i] <> s2[i]
   then inc(dif);
```

```
  CompRareLaNivelDeOctet := dif;
end;

function Distanta : integer;
  var
    stare:TStare;
begin
  stare := r^.inf;
  Distanta := CompRareLaNivelDeOctet(stare,StFinal);
end;

function CostulStarii:integer;
begin
  CostulStarii := r^.niv + Distanta;
end;

procedure Decizia(i:byte; stare:TStare; var StareNoua:TStare);
begin
{ aici se insereaza cod pentru generarea unei decizii}
end;

procedure ConstruiesteCap;
begin
  new(cap);
  cap^.urm := nil;
  cap^.pred := nil;
  cap^.niv := 0;
  cap^.inf := wIni;
  r := cap;
  cap^.cost := Distanta;
  new(r);
  r^.urm := cap;
  r^.cost := 0;
  r^.pred := nil;
  r^.niv := 0;
  cap^.pred := r;
end;

procedure AdaugaNod(stare:TLista);
  var t,q:lista;
```

```
      co:integer;
  begin
    co := stare^.cost;
    t:=cap; q:=cap;
    while (co>=t^.cost) and (t<>nil) do begin q:=t; t:=t^.urm; end;
    q^.urm:=r; r^.urm:=t; r^.pred:=cap;
  end;


begin {programul principal}
  mark(StareHeap);
  Citire;
  ConstruiesteCap;
  first := cap;
  while cap^.cost <> cap^.niv do
  begin
    for i := 1 to 4 do
    begin
      Decizia(i,cap^.inf,stare);
      new(r);
      r^.inf := stare;
      r^.urm := nil;
      r^.pred := cap;
      r^.niv := cap^.niv + 1;
      r^.cost := Distanta;
      Adauga(r);
    end;
    cap := cap^.urm;
  end;
  Afisare;
  release(StareHeap);
end.
```

Exemplul 2. Jocul Perspicco

Să rezolvăm aceeaşi problemă, dar de data aceasta să nu reţinem stările (deoarece o matrice de 4x4 cu elemente în mulţimea 0..15 ocupă 16 octeţi, ci mai bine să reţinem deciziile care trec jocul dintr-o stare în alta. Pentru aceasta trebuie să specificăm:

- ✓ Datele de intrare se citesc din fişierul persp.in. El conţine două configuraţii (cea iniţială şi cea finală) specificate prin variabilele *wIni*,*wFin*.

- ✓ La ieşire se doreşte drumul până la configuraţia finală,
- ✓ Tipul unei stări este *TStare = array[1..4,1..4] of byte*,
- ✓ Tipul elementelor soluţiei este conţine *decizia curentă*,
- ✓ Se initializează capul cu decizia 0 (care nu face nimic)
- ✓ Starea de final trebuie să fie *WFin*,
- ✓ Distanta dintre două configuraţii este egală cu numărul de octeţi prin care cele două diferă,
- ✓ Costul este calculat după formula *cost := Nivelul curent + distanta dintre configuratia curenta si configuratia finala*,
- ✓ Numărul deciziilor care se pot lua la un pas este 4 (schimbare cu căsuţa din stânga, dreapta, sus, jos).

Codul sursă generat pe baza specificaţiilor de mai sus este, următorul, la care însă mai trebuie adăugat cod pentru specificarea modului în care acţionează fiecare decizie asupra unei configuraţii.

```
Program BranchAndBound;
uses crt;
 type
    TStare = array[1..4,1..4] of byte;
    TMutari = array [1..MaxM] of byte;
    TLista = ^art;
    art = record
          inf : byte;
          cost : integer;
          urm : TLista;
          pred : TLista;
          niv : byte;
         end;

 var
 StareHeap : pointer;
  WIni,WFin:TStare;

procedure Citire;
 var f:text;
begin
 assign(f,'perspico.in');
 reset(f);
 readln(f,Numele variabilei);
```

```
  readln(f,WIni,WFin);
  Close(f);
end;

  procedure Afisare;
    var
     i:byte;
    mut : TMutari;
    nrm : byte;
  begin
    ReturnSirulMut(cap^.urm,mut,nrm);
    for i := 1 to nrm do
      write(f,"mut[i]," ");
  end;

  function CompărareLaNivelDeOctet(st1,st2:TStare):byte;
    var s1,s2:string;
        i,dif : byte;
  begin
    mov(st1,s1,sizeof(TStare));
    mov(st2,s2,sizeof(TStare));
    dif := 0;
    for i := 1 to sizeof(TStare) do
      if s1[i] <> s2[i]
      then inc(dif);
    CompărareLaNivelDeOctet := dif;
  end;

  function Distanta : integer;
    var
      stare : TStare;
      mut : TMutari;
      nrm : byte;
  begin
    ReturnSirulMut(r,mut,nrm);
    ReconstituieStare(mut,nrm,stare);
    Distanta := CompărareLaNivelDeOctet(stare,StFinal);
  end;

  function CostulStarii:integer;
```

```
begin
  CostulStarii := r^.niv + Distanta;
end;

procedure Decizia(i:byte; stare:TStare; var StareNoua:TStare);
begin
{ aici se insereaza cod pentru generarea unei decizii}
end;

procedure ReturnSirulMut(StCur:TLista; var mutari : TMutari; var NrMut:byte);
  var t: TLista;
      x,sch : byte;
begin
  t := StCur;
  NrMut := 0;
  while t^.pred <> nil do
  begin
    inc(NrMut);
    mutari[Nrmut] := t^.inf;
    t := t^.pred;
  end;
  for x := 1 to NrMut div 2 do
  begin
    sch := mutari[x];
    mutari[x] := mutari[NrMut-x+1];
    mutari[Nrmut-x+1] := sch;
  end;
end;

procedure ReconstituieStare(m:TMutari; Nm:byte; var conf:TStare);
   var i:byte;
       st : TStare;
begin
    conf := wIni;
   for i:= 1 to Nm do
   begin
      Decizia(i,conf, st);
      conf := st;
   end;
end;
```

```
  procedure ConstruiesteCap;
  begin
    new(cap);
    cap^.urm := nil;
    cap^.pred := nil;
    cap^.niv := 0;
    cap^.inf := wIni;
    r := cap;
    cap^.cost := Distanta;
    new(r);
    r^.urm := cap;
    r^.cost := 0;
    r^.pred := nil;
    r^.niv := 0;
    cap^.pred := r;
  end;

  procedure AdaugaNod(stare:TLista);
    var t,q:lista;
        co:integer;
  begin
    co := stare^.cost;
    t:=cap; q:=cap;
    while (co>=t^.cost) and (t<>nil) do begin q:=t; t:=t^.urm; end;
    q^.urm:=r; r^.urm:=t; r^.pred:=cap;
  end;


begin {programul principal}
  mark(StareHeap);
  Citire;
  ConstruiesteCap;
  first := cap;
  while cap^.cost <> cap^.niv do
  begin
    for i := 1 to 4 do
    begin
      Decizia(i,cap^.inf,stare);
      new(r);
```

```
      r^.inf := stare;
      r^.urm := nil;
      r^.pred := cap;
      r^.niv := cap^.niv + 1;
      r^.cost := Distanta;
      Adauga(r);
    end;
    cap := cap^.urm;
  end;
  Afisare;
  release(StareHeap);
end.
```

Bibliografie

- T. Bălănescu, Metoda backtracking(I), articol din Gazeta de Informatică Nr2/ 1993, Editura Libris, Cluj-Napoca.
- M. Frenţiu, V. Prejmerean, Algoritmică şi programare, curs litografiat, Universitatea Babeş-Bolyai, Cluj-Napoca, 1995.
- L.Livovschi, H. Georgescu, Sinteza şi analiza algoritmilor, Editura Ştiinţifică şi Enciclopedică, Bucureşti, 1986.
- M. Oltean, Culegere de probleme cu rezolvări în Pascal, Editura Libris, Cluj Napoca, 1996.
- M. Oltean, Probleme rezolvate în Pascal, Editura Albastră, Grupul Microinformatica, Cluj-Napoca, 1995.
- M.Oltean, M.Cocan, C++Builder în aplicaţii, Editura Albastră, Grupul Microinformatica, Cluj-Napoca, 1998.