BACTRACK – GYAKORLAT

VÁZLAT

Szlávi Péter

2003

TARTALOM

A. EMLÉKEZTETŐ

Az alábbiakban emlékeztetőként meggondoljuk a backtrack (-es keresés) algoritmusának lényegét, valamint értelmezzük a specifikáció néhány kellékét.[♣] Két konkrét feladaton keresztül vizsgálódunk: a jól ismert N-vezér és a munkavállalós problémán keresztül. (Az utóbbira kétféle megoldást is körvonalazunk.)

A1. A megoldás madártávlatból

… a szisztematikus előrehaladás és visszalépés bemutatása a két feladaton keresztül …

A2. Fogalmak…

Mindenek elé maga az absztrakt specifikáció.

Be: $N\in\mathbb{N}$, $M\in\mathbb{N}^*$ – N elemszámú méretsorozat,

$Y_i\in H_i^*$ $(i\in[1..N])$ – m_i elemszámú lehetőségsorozatok,

$fk: \bigcup_{i=1}^{N} \mathbb{N}^i \to \mathbb{L}$ – lehetőségek összeférése (indexeikkel megadva),

$ft: \mathbb{N}^2 \to \mathbb{L}$ – (az önmagában) kiválaszthatóság

Ki: $Van\in\mathbb{L}$, $X\in\mathbb{N}^*$ – az összeférő lehetőségek indexeinek sorozata

Ef: $N=Hossz(M) \wedge N\geq 2 \wedge$

$\forall i\in[1..N]: (M_j=Hossz(Y_j) \wedge HalmazFölsorolás(Y_j)) \wedge$

$\forall j\in[1..N], \forall i\in[1..j]: fk(n_1,...n_j) \Rightarrow fk(n_1,...n_i)$

Uf: $Van \equiv \exists Z\in[1..m_1]\times[1..m_2]\times...\times[1..m_N]: (\forall i\in[1..N]: ft(i,z_i) \wedge fk(Z)) \wedge$

$Van \Rightarrow (\forall i\in[1..N]: x_i\in[1..m_i] \wedge ft(i,x_i)) \wedge fk(x_1,..x_N)$

Nézzük a fenti fogalmak miként testesülnek meg a konkrét feladatokban!

N-vezér probléma	**N munka – N munkavállaló**
Mi az eredmény?	
N vezérhez tartozó a. sakktábla-mező koordináta ($\mathbb{N}^2$) $\Rightarrow (\mathbb{N}^2)^N$ b. sakktábla oszlop vagy sor-index ($\mathbb{N}$) $\Rightarrow (\mathbb{N})^N$	N munkavállalóhoz tartozó munka ($\mathbb{M}\mathrm{unka}$) $\Rightarrow \mathbb{M}\mathrm{unka}^N$ Megjegyzés: a $\mathbb{M}\mathrm{unka}$ halmaz egy *absztrakt halmaz* (elemei előre nem ismertek), de helyettesíthető $\mathbb{N}$ részhalmazával, ui.: soroljuk föl elemeit egy tömbben, és egy elemre eztán *index*ével hivatkozhatunk.
Mi a bemenet?	
N vezérhez tartozó a. *összes* sakktábla-mező koordináta ($\mathbb{N}^2$) $\Rightarrow (\mathbb{N}^2)^{N\cdot N}$	N munkavállalóhoz tartozó a. vállalható munkák felsorolása: $\mathbb{M}\mathrm{unka}^*$ (persze a Db és a munkasorozat megadásával)

[♣] Erősen építünk az előadás jelöléseire, koncepciójára.
(Lásd pl.: http://people.inf.elte.hu/szlavi/PrM1felev/Pdf/PrMea6.pdf).

b. *összes* sakktábla oszlop (sor)-index ($\mathbb{N}$) $\Rightarrow (\mathbb{N})^N$ Mindez persze csak „virtuálisan”, a tételhez való megfeleltetés kedvéért.	$\Rightarrow (\text{Munka}^*)^N$, b. *összes* (M) munka vállal*hat*ósága: $\mathbb{L}^M$, (ott igaz, amelyiket elvállalhatja) $\Rightarrow (\mathbb{L}^M)^N = (\mathbb{L})^{N \cdot M}$
Mi az fk függvény? Általában (jelentése és tulajdonsága) és konkrétan …	
nincs ütésben = *páronként nem ütik egymást*	nincs kiadva még a munka = (*páronként* ellenőrizve) *még senkié sem a munka*
Mi az ft függvény? Általában (jelentése és tulajdonsága) és konkrétan …	
∅	a. ∅ b. választható-e a munka az adott munkavállalóhoz? Vállalhatja-e a munkavállaló sajátmaga miatt?
Mi az uf?	
Van = … Van ⇒ …	Van = … Van ⇒ …

A3. A megoldás algoritmusa

A specifikáció értelmezése után térjünk át az algoritmus részleteire!

A legfelsőbb szint:

```
Típus
  TKeresett=Rekord(van:Logikai, melyik:Egész)
Változó
  i:Egész
  ker:TKeresett
i:=1; X(1..N):=0 [az i. kezdőszeletnél tartunk]
Ciklus amíg i≥1 és i≤N
  ker:=JóElem(i)
  Ha ker.van akkor  X(i):=ker.melyik; i:+1
            különben X(i):=0; i:-1
Ciklus vége
Van:=i≥1
[X-ben található a megoldás, ha létezik]
```

A `JóElem` függvény finomítása:

```
Függvény JóElem(i:Egész): TKeresett
  [i.-ben választható-e jó komponens?
   M(1..N) tartalmazza az egyes bemeneti sorozatok elemszámát]
  Változó
    j:Egész
```

```
    j:=X(i)+1
    Ciklus amíg j≤M(i) és (nem ft(i,j) vagy nem Lehetséges(i,j))
      j:+1
    Ciklus vége
    JóElem.van:=j≤M(i)
    Ha JóElem.van akkor JóElem.melyik:=j
  Függvény vége.
```

```
  Függvény Lehetséges(Konstans i,j:Egész):Logikai
    [az i. komponensként a j választása összefér-e az eddigiekkel?]
    Lehetséges:=fk(X(1),...,X(i-1),j)
  Függvény vége.
```

Ez gyakorta (=páronkénti ellenőrzés esete) egy eldöntés tételt jelent.

Végül az van csak hátra, hogy a konkrét feladatokhoz tartozó algoritmusokat elkészítsük.

Mi az algoritmus?

… | …

A4. Kódolás…

Az N-vezér probléma kódolásához egy keretprogramot biztosítunk, amit alább részletezünk. Letöltheti: VEZKERET.PAS, egy hozzá szükséges kellékkel együtt: VEZERAJZ.PAS, amivel a sakktábla rajta a vezérekkel „vizualizálható”. Az utóbbihoz tartozó használati tudnivalókat az eljárások fejsorai megadják:

```
Procedure Inicializalas(Var n: Index);
Procedure Varakozas(meddig: Integer);
Procedure MezoRajz(x,y: Integer; szin: Integer);
Procedure Letesz(i,j: Index);
Procedure Folvesz(i,j: Index);
Procedure TablaKi(n: Index);
```

A JóElem függvény összetett értéktípusa nem minden programozási nyelv számára kivitelezhető, ezért másként paraméterezzük. E függvényből lett eljárásnak a neve legyen: VanJóEset! (Innentől kezdve igazodunk a mikrológiában szereplő jelölésekhez.) Az alábbiakban lásd a keret kódját, amelyben *kékkel* szedtük a megalkotandó részeket.

```
1   Program N_Vezer_Keret;
2     Uses
3        Newdelay, Crt;
4     Const
5        MaxN  = 10; VarakozasHossz = 500;
6     Type
7        Index = 0..MaxN+1;
8        Hol   = Array [1..MaxN] of Index;
9     Var
10       Vezer : Hol;
11       N,i,j,
12       hova  : Index;
13       lehet : Boolean;
```

```
14
15        {$i VEZERAJZ.PAS}
16
17        Procedure VanMegoldas(n: Index);
18          Var
19            i,j: Index;
20        Begin
21          Window(1,1, 80,25); TextColor(White); TextBackGround(Black); ClrScr;
22          Write('Az N-vezérproblémának N=',n,' esetre van megoldása, mégpedig
23                 az alábbi:');
24          TablaKi(n);
25          Varakozas(0);
26        End;
27
28        Procedure NincsMegoldas(n: Index);
29        Begin
30          Window(1,1, 80,25); TextColor(White); TextBackGround(Black); ClrScr;
31          Write('Az N-vezérproblémának N=',n,' esetre nincs megoldása.');
32          Varakozas(0);
33        End;
34
35        Function Uti(vx1,vy1, vx2,vy2: Index): Boolean;
36        Begin
37          {Uti:=Mikor üti?}
38        End; {Uti}
39
40        Function RosszEset(x,y: Index): Boolean;
41          Var
42             j: Index;
43        Begin
44          {RosszEset:=Mikor rossz az eset?}
45        End;
46
47        Procedure VanJoEset(i: Index; Var talan: Boolean; Var ide: Index);
48        Begin
49          {ide:=Hova?;
50           talan:=Van jó eset?}
51        End;
52
53      Begin
54        Inicializalas(N);
55        {A backtrack lényegi algoritmusa.}
56        If i>0 then VanMegoldas(N)
57               else NincsMegoldas(N);
58      End.
```

A fenti megoldását a Függelék tartalmazza.

A további feladatokat e feladatsorból vettük:

http:/people.inf.elte.hu/szlavi/PrM1felev/Backtrack/BacktrackFeladatok.pdf

B. LABIRINTUSOS FELADAT

B1. A feladat

Egy éhes egérnek egy labirintusban elhelyeznek egy darab sajtot. Írjunk programot, amely segít az egérnek megkeresni a sajthoz vezető utat!

B2. Útban a megoldás felé – Problémák

1. **Mi az eredmény**?

   Válasz: egy „útvonal-megadás” (lépésirányok: `Irány=(Bal,Fel,Jobb,Le)`), ami egy előre nem látható hosszúságú (azaz az alap algoritmusbeli `N` nem fix) labirintusbeli

Ki ⇔ hely-, vagy iránysorozat: `Hely(0..MaxHossz:Hol)`, vagy `MerreTovább(0..MaxHossz:Irány)` tömb, ahol `MaxHossz` a labirintusbeli utak összhossza, a 0. elem a kezdő helyzet. (A `Hol` miben léte eddig még kérdéses, bár világos.) Döntsünk az `Irány`-ra alapozott mellett!

2. **Mik a bemeneti sorozatok**?

   Válasz: a négy irány halmazából éppen annyi, amennyi lépés kell a megoldáshoz. De

Be ⇔ természetesen ez nem valódi programbemenet, csak a tétel egyik bemeneti kellékének konkretizációja.

3. **Hogyan ábrázolandó a labirintus**?

Általában, a feladat bemenete (nem függ össze a backtrack algoritmussal)

Pl. egy „térképpel” (`Labirintus(Hol:Elem)` tömbbel[1]), amelyben háromféle érték fordulhat elő: `fal`, `út` és `sajt` (`Elem`). A `Hol` „logikusan” lehet síkbeli pontok koordinátapárja: `(1..N,1..M)`. Így tehát: `Labirintus(1..N,1..M: Elem)` –most már tudjuk, hogy– mátrix. Ekkor viszont az `Irány`-beli értékek elmozdulás-megfelelői már adódnak: `Bal`→`(1,0)`, `Fel`→`(0,1)`, `Jobb`→`(-1,0)`, `Le`→`(0,-1)`.

4. Ebben az esetben **mi az fk (kielégíthetőségi) és az ft függvény**?

   Válasz: `fk` legyen olyan, hogy ha $Hely_i$ az útvonal `i`. lépése utáni hely; és az `i`. lépés után meglépendő `ir` irány esetén teljesüljön: $fk(Hely_0,\ldots,Hely_{i-1},ir)=Igaz$ akkor, ha a $Hely_{i-1}+ir= Hely_i \notin \{Hely_0,\ldots Hely_{i-1}\}$, azaz itt még nem járt az egér; az $ft: Labirintus(Hely_{i-1}+ir)\neq fal$, azaz az úton marad. (A **+** művelet a vektor-összeadás.)

5. **Meddig tart a keresés**?

   Válasz: `N`-re nem építhetünk (ez most a labirintus egyik mérete, s nem az eredmény sorozat hossza, azaz a szükséges lépések száma), így merészen és logikusan így döntünk: addig tart a keresés, amíg a `sajt`ra nem találtunk, vagy a `RajtHely`-től indulva már minden lehetséges irányt kipróbáltunk, és nem tudtunk a `sajt`ig jutni. Ekkor a szegény egér éhes marad.

[1] Ne akadjunk fel azon, ha egyik-másik programozási nyelv ilyen típust indexként nem fogad el, hanem értsük: mit akar kifejezni!

B3. Algoritmus

A megvalósítás során tárolnunk célszerű (miért is?) az utat (`Hely` tömb), a(z ál)bemeneti sorozatokból (`Irány` sorozatokból) választott indexeket (`LépésIrányIndex` tömb). A konstans `Irány` tömb indexkonvenciója: 1 a `Bal`hoz, azaz az (1,0)-hoz, a 2 a `Fel`hez, azaz a (0,1)-hez éít., a 0 –szokás szerint– a még „ki nem választottat” jelenti. (Vagyis: `Irány: Tömb(0..4:Hol).`) A megoldás tehát: a `LépésIrányIndex` tömb és a tényleges hossza (az utóbbit a `LépésSzám` tartalmazza), és persze a megoldhatóságot kifejező `VanÚt` logikai változó.

A program valódi paraméterei: `Labirintus`, `RajtHely`. (A továbbiakban nem foglalkozunk a deklarációkkal.)

```
Program SajtKeresés:
   …
   Hely(0):=RajtHely [ahonnan indul az egér, pl. (1,1)]
X  i:=1; LépésIrányIndex(1..MaxHossz):=0
i≥N Ciklus amíg i≥1 és Labirintus(Hely(i-1))≠Sajt
      VanJóEset(i, van, j)
      Ha van akkor
        LépésIrányIndex(i):=j
        Hely(i):=Hely(i-1)+Irány(j) [+: vektorösszeadás]
        i:+1
      különben
        LépésIrányIndex(i):=0
        i:-1
      Elágazás vége
   Ciklus vége
   VanÚt:=i≥1
   Ha VanÚt akkor LépésSzám:=i-1
Eljárás vége.

Eljárás VanJóEset(Konstans i:Egész, Változó van:Logikai, j:Egész):
   j:=LépésIrányIndex(i)+1
   Ciklus amíg j≤4 és (RosszEset(i,j) vagy
         Labirintus(Hely(i-1)+Irány(j))=Fal
         [a Labirintusmátrix Hely(i-1)+Irány(j)
          koordinátájú pontja Fal értékű-e]
      j:+1
   Ciklus vége
   van:=j≤4
Eljárás vége.

Függvény RosszEset(Konstans i,j:Egész): Logikai
   k:=0
   Ciklus amíg k<i és Hely(k)≠Hely(i-1)+Irány(j)
      k:+1
   Ciklus vége
   RosszEset:=k<i
Függvény vége.
```

B4. További problémák, kérdések

6. Ha a `Labirintus`-beli értékként megengedünk egy „`JártÚt`” értéket, annak jelölésére, hogy arra már járt az egér, akkor lényegesen egyszerűsíthető az algoritmus. Hogyan? (A `RosszEset`-ben nincs szükség az eldöntés tétel alkalmazására.)
7. Milyen tulajdonsággal rendelkezik az egér útja, ha csak 1-szélességű folyosók vannak? (Nem biztos, hogy a legrövidebb, de fölösleges „hurkokat” nem tartalmaz.)
8. ... és ha lehetnek 2-, vagy több szélességű folyosók? Mi lehet a szerepe a „szűk” folyosóknak? (Tartalmazhat az egér útja olyan szakaszokat, amelyeken oda is és vissza is megy a fal mellett, s nem veszi észre, hogy járt már e folyosón. Az 1-szélességűnél képes levágni e hurkokat.)
9. Mit lehetne tenni az előző pontban említett probléma intelligens megoldására? (A folyosó teljes szélességében figyelni, hogy volt-e már ott; s ha igen, akkor levágni a hurkot; és természetesen ugyanígy figyeli a sajtot is.)

C. DOMINÓS FELADAT

C1. A feladat

Tegyük le az összes dominót úgy, hogy csak az egyik irányba tehetünk, és a dominókon az összes lehetséges párosítás ((0:0),(0:1),...,(0:9),(1:1),...,(9:9)) előfordul!

C2. Útban a megoldás felé – Problémák

1. **Eredmény...**: dominósorozat (a fordított dominót nem tároljuk ? darabszám=10+9+... +1=55).
2. **Bemeneti sorozatok**: dominók „halmaza”.
3. **Ábrázolás...**: a dominók = `Dominó(1..55[melyik],1..2 [oldal]:0..9 [hány pötty])`. (Persze az egyes dominók „olcsóbban” is kódolhatók: egy-egy kétjegyű, decimális számmal.)
4. **fk,ft...**: eddig még nem tettük le; s az (eddigi) utolsóhoz illeszthető-e. (Szorosan véve az `ft` értelmezését –„csak önmagában, saját maga miatt, az eddigi választásainktól függetlenül megfelelő”– „gyenge” kapcsolatot vehetünk észre az utolsóként választottal!)
5. **Meddig...**: amíg az összest le nem tettük.

C3. Algoritmus

```
Program Dominó:
   … [most N=55]
X  i:=1; DominóSor(1..N):=0
   Ciklus amíg i≥1 és i≤N
      VanJóEset(i, van, j)
      Ha van akkor  DominóSor(i):=j; i:+1
           különben DominóSor(i):=0; i:-1
   Ciklus vége
   Letehető:=i>N
   Ha Letehető akkor … DominóSor(1..N) a letevés …
   …
Program vége.


Eljárás VanJóEset(Konstans i:Egész, Változó van:Logikai, j:Egész):
   j:=DominóSor(i)+1
   Ciklus amíg j≤N és ( RosszEset(i,j) vagy
¬ft                nem UtolsóhozIllik(i,j) )
      j:+1
   Ciklus vége
   van:=j≤N
   Ha van és Dominó(DominóSor(i-1),2)≠Dominó(j,1)
         akkor Csere(Dominó(j,2),Dominó(j,1))
Eljárás vége.
```

```
Függvény RosszEset(Konstans i,j:Egész): Logikai
   k:=1
   Ciklus amíg k<i és DominóSor(k)≠j
      k:+1
   Ciklus vége
   RosszEset:=k<i
Függvény vége.

Függvény UtolsóhozIllik(Konstans i,j:Egész): Logikai
  UtolsóhozIllik:=i=1 vagy
                  Dominó(DominóSor(i-1),2)=Dominó(j,1) vagy
                  Dominó(DominóSor(i-1),2)=Dominó(j,2)
Függvény vége.
```

D. SZAKASZ-LEFEDÉSES FELADAT

D1. A feladat

Lefedhető-e egy adott hosszúságú szakasz egyszeresen a $H_1,...,H_N$ hosszúságú kisebb szakaszokkal?

D2. Útban a megoldás felé – Problémák

1. **Eredmény…**: szakasz-sorozat (**legfeljebb** `N` darab), pontosabban szakaszindex (`SzI(1..???)`).
2. **Bemeneti sorozatok**: a szakaszok „halmaza”.
3. **Ábrázolás…**: szakaszok hossza (`H(1..N)`).
4. **fk,ft…**: az eddig még nem kiválasztott szakasz; az eddig lefedett szakasz összhossza (=pillanatnyi hossz) ≤ lefedendő szakasz hossza (=`szakaszHossz`); itt is „implicit” kapcsolat van a korábbiakkal.
5. **Meddig…**: a lefedett összhossz (=pillanatnyi hossz) ≠ lefedendő szakasz hossza (`szakaszHossz`); és persze még nem használtuk föl az összes szakaszt.
   Utóbbi csak akkor következhet be, ha a szakaszok összhossza kisebb, mint a lefedendő szakaszé. Ez könnyen ellenőrizhető, így elkerülhető a backtrack-es algoritmus effajta „elvi” kudarca.

D3. Algoritmus

Tegyük föl, hogy elvileg akár létezhet megoldás, azaz $\sum_{i=1}^{N} H_i \geq H$.

Érdemes észre venni, hogy amikor majd az `fk` kiszámolást végző `VanJóEset` eljárást írjuk minduntalan ki kell számolnunk az eddig letett szakaszok összhosszát, amit könnyen elkerülhetnénk úgy, hogy „göngyölítjük” azt egy segédváltozóban. Ez legyen a `pillHossz`.
Ha így teszünk, akkor ennek módosulását nyomon kell követni a legfelsőbb szinten: ha „`VanJóEset`”, akkor `pillHossz`-növelés, ha „`NincsJóEset`”, akkor levonni a `pillHossz`-ból az utolsó, elfogadott választást. Ez egy problémás helyzet, hiszen ha egyáltalán nincs megoldás, visszalépünk –mint jól ismert– a „nem létező” 0. választáshoz, ami azt jelenti, hogy a hozzátartozó „hosszt” akarjuk a `pillHossz`-ból levonni. ... „Legkönnyedebb” megoldás: `H(0..N:Valós)` és `SzI(0..N:Egész)`.

```
                Program Lefedés:
 X                 pillHossz:=0; H(0):=0; SzI(0):=0
                   i:=1; SzI(1..N):=0
i≥N                Ciklus amíg i≥1 és pillHossz≠szakaszHossz
                      VanJóEset(i, van, j)
                      Ha van akkor  SzI(i):=j; pillHossz:+H(j);        i:+1
                           különben SzI(i):=0; pillHossz:-H(SzI(i-1)); i:-1
                   Ciklus vége
                   Lefedhető:=pillHossz=szakaszHossz
                Program vége.


                Eljárás VanJóEset(Konstans i:Egész, Változó van:Logikai, j:Egész):
                   j:=SzI(i)+1
                   Ciklus amíg j≤N és (RosszEset(i,j) vagy
¬ft                                    pillHossz+H(j)>szakaszHossz)
                      j:+1
                   Ciklus vége
                   van:=j≤N
                Eljárás vége.


                Függvény RosszEset(Konstans i,j:Egész): Logikai
                   k:=1
                   Ciklus amíg k<i és SzI(k)≠j
                      k:+1
                   Ciklus vége
                   RosszEset:=k<i
                Függvény vége.
```

FÜGGELÉK

Az N-vezéres feladat végeleges megoldása, amiben a pirossal szedett újdonságra koncentráltunk csak:

```
Program N_Vezer;
…

  Function Uti(vx1,vy1, vx2,vy2: Index): Boolean;
  Begin
    Uti:=(vy1=vy2) or (Abs(vy1-vy2)=vx1-vx2)
  End; {Uti}

  Function RosszEset(x,y: Index): Boolean;
    Var
       j: Index;
  Begin
    j:=1;
    While (j<=x-1) and not uti(x,y, j,Vezer[j]) do Inc(j);
    RosszEset:=j<=x-1
  End;

  Procedure VanJoEset(i: Index; Var talan: Boolean; Var ide: Index);
  Begin
    ide:=Vezer[i]+1;
    While (ide<=N) and RosszEset(i,ide) do Inc(ide);
    talan:=ide<=N;
  End;

Begin
  Inicializalas(N);
  i:=1;
  For j:=1 to N do Vezer[j]:=0;
  While i in [1..N] do
  Begin
    VanJoEset(i,lehet,hova);
    If lehet then
    Begin
      Letesz(i,hova);
      Inc(i); Vezer[i]:=hova;
    End
      else
    Begin
      If i>1 then Folvesz(i-1,Vezer[i-1]);
      Vezer[i]:=0; Dec(i);
    End;
    Varakozas(50);
  End;
  If i>0 then VanMegoldas(N)
         else NincsMegoldas(N);
End.
```