Mesterséges intelligencia

gyakorlat

3. feladat

(optimális keresés)

Készítette:

Szathmáry László
II. PM.

Feladat: egy általam tetszőlegesen kiválasztott feladat optimális megoldásainak megkeresése

Királytúra

Adott egy 3X3-as tábla. A tábla egy tetszőleges mezőjébe egy királyt helyezünk. A feladat a tábla összes mezőjének a bejárása úgy, hogy minden mezőt csak egyszer érinthet, és a király bármely szomszédos mezőre léphet a sakk szabályainak megfelelően.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#define  N 3

struct rek {                                          //a láncolt lista szerkezete
  char * ut;
  int     koltseg;
  struct rek * kov;
};

class Kiraly {
 public:
   Kiraly();                                          //konstruktor
   void vezerlo();
   void kiir();
 private:
   char tomb[N][N];                                   //ez lesz az állapottér
   struct rek *fej, *akt;
   int balra_k, fel_balra_k, fel_k, fel_jobbra_k,
        jobbra_k, le_jobbra_k, le_k, le_balra_k;      //az egyes operátorokhoz költség tartozik

   void beker();                                      //fv.-prototípusok
   void hiba();
   void keres(int *, int *);
   int  vege();
   void torol();
   int  ures(int, int);
   void beszur(int, int, int);

   int balra_ef(int, int);               //1.
   void balra(int, int);
   int fel_balra_ef(int, int);           //2.
   void fel_balra(int, int);
   int fel_ef(int, int);                 //3.
   void fel(int, int);
   int fel_jobbra_ef(int, int);          //4.
   void fel_jobbra(int, int);
   int jobbra_ef(int, int);              //5.
   void jobbra(int, int);
   int le_jobbra_ef(int, int);           //6.
   void le_jobbra(int, int);
   int le_ef(int, int);                  //7.
   void le(int, int);
   int le_balra_ef(int, int);            //8.
   void le_balra(int, int);
};
```

I. Állapottér megadása

Az állapottér egy 3X3-as tábla, melyet egy kétdimenziós tömbbel, azaz mátrixszal reprezentálunk.
A tábla minden egyes mezőjéhez hozzárendelünk egy azonosítót (jelen esetben egy karaktert), amely a tábla egy mezőjét egyértelműen azonosítja.
A kiválasztható csúcsokat egy láncolt listában fogjuk elhelyezni. Mivel egy mezőt csak egyszer érinthetünk, ezért le kell tárolni az útvonalat. Ezt a mezőazonosítók segítségével tesszük meg, vagyis az út egy karaktersorozat lesz (a láncolt lista egy eleme így egy karaktersorozatból, az ehhez az úthoz tartozó költségből és egy mutatóból fog állni).

II. Kezdőállapot megadása

A kezdőállapot meghatározására nagyon jól használható a C++ konstruktora:

```
Kiraly::Kiraly()
{
  char count='a';

  for (int i=0; i<N; ++i)
    for (int j=0; j<N; ++j)
      tomb[i][j] = count++;                    //karakterekkel való feltöltés
  balra_k=2;
  fel_balra_k=1;
  fel_k=4;
  fel_jobbra_k=2;
  jobbra_k=3;
  le_jobbra_k=2;
  le_k=4;
  le_balra_k=1;
  beker();
}
```

A tömböt (**tomb**) feltöltjük karakterekkel a következőképpen:

a	b	c
d	e	f
g	h	i

Mivel optimális keresést akarunk megvalósítani, ezért az egyes operátorokhoz költség tartozik. (Feladatunk az lesz, hogy egy olyan utat találjunk, hogy az abban felhasznált operátorok összköltsége minimális legyen). Ezért megállapítjuk az egyes operátorok alkalmazásának költségét.

Ezután még meghívjuk a bekér "eljárást", mely tájékoztatja a felhasználót, hogy mit is csinál a program, ill. bekéri tőle azt a kezdőpozíciót, ahová kezdetben elhelyezzük a királyt:

```
void Kiraly::beker()
{
  int x,y;

  clrscr();
  printf("Feladat: egy %dX%d-os sakktabla bejarasa egy kirallyal ugy,\n"
         "hogy a kovetkezo koltsegek mellett optimalis megoldast kapjunk:\n\n"
         "balra:  %2d;  fel balra: %2d;  fel: %2d;  fel jobbra: %2d;\n"
         "jobbra: %2d;  le jobbra: %2d;  le:  %2d;  le   balra: %2d;\n\n"
         ,N,N, balra_k, fel_balra_k, fel_k, fel_jobbra_k,
               jobbra_k, le_jobbra_k, le_k, le_balra_k);
  printf("Adja meg a kezdopoziciot (a szamozas 0-tol kezdodik!):\n\n");
  printf("Sor    (0-%d): ",    N-1); scanf("%d", &x);
  printf("Oszlop (0-%d): ", N-1); scanf("%d", &y);

  fej = (struct rek *) malloc(sizeof(struct rek)); if (!fej) hiba();
  char *temp = "_\0"; temp[0]=tomb[x][y];
  fej->ut = strdup(temp); fej->koltseg=0; fej->kov=NULL;
}
```

A végén megtörténik a láncolt lista inicializálása is: a **fej**-nek helyet foglalunk, s annak a mezőnek megfelelő azonosítót tároljuk le benne, amelyet a felhasználó megadott. Mivel ez a start csúcs, így a költsége 0, ill. kezdetben még nincs rákövetkezője.

III. Végállapot megadása

```
int Kiraly::vege()
{
  return (strlen(akt->ut)==N*N);
}
```

A keresésnek akkor van vége, ha a kiterjesztésre kiválasztott csúcsot tesztelve azt tapasztaljuk, hogy már minden mező érintve volt, vagyis a letárolt út hossza 3X3.

IV. Operátorok

Mivel a királlyal 8 irányba lehet lépni, ezért 8 operátorra, s az ezekhez tartozó 8 darab előfeltételre lesz szükségünk. Mi a közös ezekben?

Előfeltétel: mindegyik kap egy x,y pozíciót (x-sor, y-oszlop), s meg kell vizsgálnia, hogy a megfelelő irányba lehet-e lépni, vagyis hogy ott még tart-e a tábla.

Továbbá: csak olyan mezőre léphetünk, ahol még nem jártunk.

Operátor: ezek meghívása egy csúcs (aktuális csúcs) kiterjesztésekor történik meg. Minden egyes operátor egy közös függvényt hív meg (**beszur**) a megfelelő paraméterekkel: azon pozíció koordinátája, ahova lépünk; s az ezen lépéshez tartozó költség.

```
int Kiraly::ures(int x, int y)
{
  char *p = akt->ut;
  while (*p)
    if (*p == tomb[x][y]) return 0;         //ha egyezés van: hiba, már jártunk ott
    else ++p;
  return 1;                                 //különben rendben, még nem voltunk ott
}
```

Az aktuális csúcsot szeretnénk kiterjeszteni, ehhez tartozik egy aktuális út. Az x,y pozícióra szeretnénk lépni, de meg kell nézni, hogy ott nem jártunk-e már. Akkor voltunk már ott, ha az x,y pozícióhoz tartozó mezőazonosító karakter már szerepelt az aktuális csúcs **út**-jában (mint karaktersorozatban).

```
void Kiraly::beszur(int sor, int oszlop, int koltseg)
{
  struct rek *egy=NULL, *ketto=fej;
  struct rek *uj = (struct rek *) malloc(sizeof(struct rek));
  char   *p;
  int      hossz = strlen(akt->ut);

  if (!uj) hiba();
  p = uj->ut = (char *) malloc(strlen(akt->ut)+2); if (!uj->ut) hiba();
  strcpy(uj->ut, akt->ut);

  p[hossz]=tomb[sor][oszlop]; p[hossz+1]='\0';
  uj->koltseg = akt->koltseg + koltseg;

  while (!(ketto==NULL || ketto->koltseg >= uj->koltseg))      //hely keresése
    egy=ketto, ketto=ketto->kov;

  egy->kov = uj; uj->kov = ketto;
}
```

Itt történik meg az aktuális csúcs kiterjesztése. Paraméterként megkapja, hogy hova lépünk, s ennek a lépésnek mennyi a költsége.
Lefoglalunk egy új listaelemet, az **ut** mezőjébe bemásoljuk az aktuális csúcsban lévő úthosszat, s MÉG hozzáírjuk annak a pozíciónak az azonosítóját, ahova lépünk.
Költség: a szülő költsége + az operátor alkalmazásának költsége. Ezután ezt az új listaelemet költség szerint *rendezetten* illesztjük be a listába => a fej mindig a legkisebb költségű elemre fog mutatni!

Általános megjegyzés:

A könnyebb áttekinthetőség kedvéért hívjuk a 0. indexszel jelölt sort (oszlopot) első sornak (oszlopnak), a 2. indexszel jelölt sort (oszlopot) pedig utolsó sornak (oszlopnak), stb.

```
/* 1. */
int Kiraly::balra_ef(int x, int y)
{
  int van_hely=(y>=1);
  return (van_hely && ures(x, y-1));
}
```

Akkor léphetünk balra, ha nem az első oszlopban vagyunk, ill. ahova lépnénk, annak a pozíciónak üresnek kell lennie.

```
void Kiraly::balra(int x, int y)
{
  beszur(x, y-1, balra_k);
}
```

Balra lépünk egyet, kiterjesztéssel egy új levélelemet kapunk.

```
/* 2. */
int Kiraly::fel_balra_ef(int x, int y)
{
  int van_hely=(x>=1 && y>=1);
  return (van_hely && ures(x-1, y-1));
}
```

Akkor léphetünk fel balra, ha x>= mint a második sor ÉS y>= mint a második sor, ill. ahova lépnénk, annak a pozíciónak üresnek kell lennie.

```
void Kiraly::fel_balra(int x, int y)
{
  beszur(x-1, y-1, fel_balra_k);
}
```

Fel balra lépünk egyet, kiterjesztéssel egy új levélelemet kapunk.

```
/* 3. */
int Kiraly::fel_ef(int x, int y)
{
  int van_hely=(x>=1);
  return (van_hely && ures(x-1, y));
}
```

Akkor léphetünk fel, ha nem az első sorban vagyunk, ill. ahova lépnénk, annak a pozíciónak üresnek kell lennie.

```
void Kiraly::fel(int x, int y)
{
  beszur(x-1, y, fel_k);
}
```

Fel lépünk egyet, kiterjesztéssel egy új levélelemet kapunk.

```
/* 4. */
int Kiraly::fel_jobbra_ef(int x, int y)
{
  int van_hely=(x>=1 && y<=N-2);
  return (van_hely && ures(x-1, y+1));
}
```

Akkor léphetünk fel jobbra, ha x>= mint a második sor ÉS y<= mint az utolsó előtti oszlop, ill. ahova lépnénk, annak a pozíciónak üresnek kell lennie.

```
void Kiraly::fel_jobbra(int x, int y)
{
  beszur(x-1, y+1, fel_jobbra_k);
}
```

Fel jobbra lépünk egyet, kiterjesztéssel egy új levélelemet kapunk.

```
/* 5. */
int Kiraly::jobbra_ef(int x, int y)
{
  int van_hely=(y<=N-2);
  return (van_hely && ures(x, y+1));
}
```

Akkor léphetünk jobbra, ha nem az utolsó oszlopban vagyunk, ill. ahova lépnénk, annak a pozíciónak üresnek kell lennie.

```
void Kiraly::jobbra(int x, int y)
{
  beszur(x, y+1, jobbra_k);
}
```

Jobbra lépünk egyet, kiterjesztéssel egy új levélelemet kapunk.

```
/* 6. */
int Kiraly::le_jobbra_ef(int x, int y)
{
  int van_hely=(x<=N-2 && y<=N-2);
  return (van_hely && ures(x+1, y+1));
}
```

Akkor léphetünk le jobbra, ha x<= mint az utolsó előtti sor ÉS y<= mint az utolsó előt-ti oszlop, ill. ahova lépnénk, annak a pozíciónak üresnek kell lennie.

```
void Kiraly::le_jobbra(int x, int y)
{
  beszur(x+1, y+1, le_jobbra_k);
}
```

Le lépünk egyet, kiterjesztéssel egy új levélelemet kapunk.

```
/* 7. */
int Kiraly::le_ef(int x, int y)
{
  int van_hely=(x<=N-2);
  return (van_hely && ures(x+1, y));
}
```

Akkor léphetünk le, ha nem az utolsó sorban vagyunk, ill. ahova lépnénk, annak a pozíciónak üresnek kell lennie.

```
void Kiraly::le(int x, int y)
{
  beszur(x+1, y, le_k);
}
```

Le lépünk egyet, kiterjesztéssel egy új levélelemet kapunk.

```
/* 8. */
int Kiraly::le_balra_ef(int x, int y)
{
  int van_hely=(x<=N-2 && y>=1);
  return (van_hely && ures(x+1, y-1));
}
```

Akkor léphetünk le balra, ha x<= mint az utolsó előtti sor ÉS y>= mint a második oszlop, ill. ahova lépnénk, annak a pozíciónak üresnek kell lennie.

```
void Kiraly::le_balra(int x, int y) {
  beszur(x+1, y-1, le_balra_k);
}
```

Le balra lépünk egyet, kiterjesztéssel egy új levélelemet kapunk.

```
void Kiraly::vezerlo()
{
  int x,y;

  keres(&x,&y);
  int z=2;                              //ez a Borland C++ 3.1 fordító miatt van itt
  while (!vege())                       //e nélkül a while blokk belsejébe ugrott
  {
    if (balra_ef(x,y))
        balra(x,y);             //1.
    if (fel_balra_ef(x,y))
        fel_balra(x,y);         //2.
    if (fel_ef(x,y))
        fel(x,y);               //3.
    if (fel_jobbra_ef(x,y))
        fel_jobbra(x,y);        //4.
    if (jobbra_ef(x,y))
        jobbra(x,y);            //5.
    if (le_jobbra_ef(x,y))
        le_jobbra(x,y);         //6.
    if (le_ef(x,y))
        le(x,y);                //7.
    if (le_balra_ef(x,y))
        le_balra(x,y);          //8.
    torol();
    keres(&x,&y);
  }
}
```

```
void Kiraly::keres(int *x, int *y)
{
  akt = fej;

  int hossz = strlen(akt->ut)-1;
  char *p   = akt->ut;
  for (int i=0; i<N; ++i)
    for (int j=0; j<N; ++j)
      if (tomb[i][j]==p[hossz]) *x=i, *y=j;
}
```

A vezérlő először is meghívja a keres "eljárást", melynek feladata, hogy az **akt** mutatót arra a listaelemre mutassa, amelynek a költsége minimális. Mivel már korábban *rendezetten* fűztük fel, ezért **akt**-ot a lista első elemére állítjuk.
A **keres**-ben még megállapítjuk, hogy a kiterjesztendő csúcsban letárolt út utolsó elemének (ami egy azonosító) mi az x,y pozíciója. A "cím" szerinti paraméterátadás miatt a vezérlő ezt megkapja, és így majd EBBŐL a pozícióból próbál majd továbblépni balra, jobbra, stb.

```
void Kiraly::torol()
{
  fej = akt->kov;
  free(akt->ut); free(akt);
}
```

Miután minden lehetséges irányba továbbléptünk az aktuális csúcsból, ezután őrá már nem lesz szükség (csak a levélelemekre, de a kiterjesztés során ő közbenső elem lett) => felszabadítjuk az általa lefoglalt tárterületet.

Ezután ismét megkeressük a legkisebb költségű csúcsot, kiterjesztjük, stb. Ezt addig ismételjük, míg végállapothoz nem jutunk.

```
void Kiraly::hiba()
{
  printf("Memoriafoglalasi hiba lepett fel.\n"); exit(-1);
}
```

Mivel dinamikusan foglalunk le tárterületet, ezért szükséges leellenőrizni a tárfoglalás sikeres vagy sikertelen voltát.

Ezek alapján a főprogram:

```
void main()
{
  Kiraly kiraly;          //példányosítás, konstruktor-> kezdőállapot meghatározása
  kiraly.vezerlo();
  kiraly.kiir();
}
```

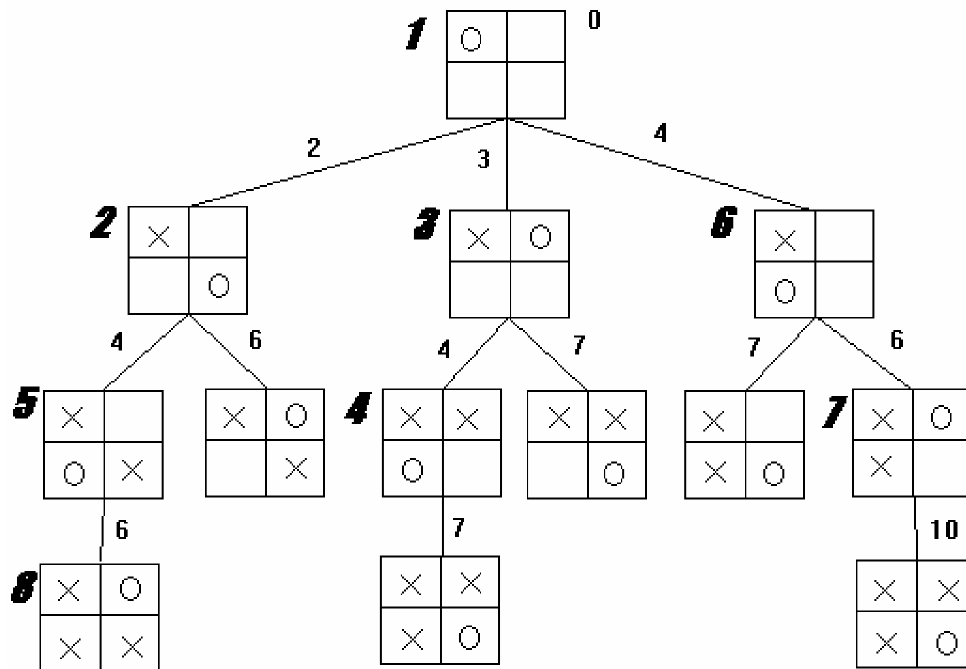
```
void Kiraly::kiir()
{
  int hossz = strlen(akt->ut);

  putchar('\n');
  char *p = akt->ut;
  for (int i=0; i<hossz; ++i)
  {
    for (int sor=0; sor<N; ++sor)
        for (int oszlop=0; oszlop<N; ++oszlop)
          if (tomb[sor][oszlop]==p[i])
            printf("%2d. lepes: (%d,%2d)\n", i+1,sor,oszlop);
  }
  printf("\nOsszkoltseg: %d\n", akt->koltseg);
}
```

A vezérlő az aktuális csúcs tesztelésekor megállapította, hogy célállapotba jutottunk, vagyis az **akt** mutató most arra a listaelemre mutat, amely kielégíti az optimális keresést. Az (**akt->ut**)-ban ott van a letárolt út, most már csak meg kell állapítani, hogy az egyes mezőazonosítók hol helyezkednek el a táblán, milyen x,y koordinátákkal rendelkeznek.

Futási eredmény:

```
Feladat: egy 3X3-os sakktabla bejarasa egy kirallyal ugy,
hogy a kovetkezo koltsegek mellett optimalis megoldast kapjunk:

balra:    2;  fel balra:   1;  fel:  4;  fel jobbra:  2;
jobbra:  3;  le jobbra:  2;  le:    4;  le   balra:   1;

Adja meg a kezdopoziciot (a szamozas 0-tol kezdodik!):

Sor    (0-2):           0
Oszlop (0-2):           0

 1. lepes: (0, 0)
 2. lepes: (1, 0)
 3. lepes: (2, 1)
 4. lepes: (2, 2)
 5. lepes: (1, 2)
 6. lepes: (0, 1)
 7. lepes: (0, 2)
 8. lepes: (1, 1)
 9. lepes: (2, 0)

Osszkoltseg: 19
```

Mivel a feladatot a **#define N 3** segítségével általánosan oldottuk meg, ezért ha N-et 2-re állítjuk, akkor a kiterjesztéseket a következőképpen lehetne szemléltetni (szerencsére N==2 esetben ez elég kicsi).



Az élek mentén fel van tüntetve a kiterjesztéssel kapott költség, mely a start csúcsban 0.
A csúcsok mellett dőlt betűvel jeleztem, hogy milyen sorrendben történnek a kiterjesztések. A **4**-es azért előzi meg az **5**-öst, mert a **2**-es kiterjesztéssel kaptunk egy 4 ill. 6 költségű csúcsot. Ezt követte a **3**-as, ahol szintén kaptunk egy 4-es költségűt, s mivel a láncolást költség szerint növekvő sorrendben oldottuk meg (nagyobb egyenlőre kerestünk, lásd **beszur** eljárás), ezért ez a 4-es költségű láncelem meg fogja előzni a **2**-es kiterjesztéssel kapottat.

Ebben az esetben a **futási eredmény**:

Feladat: egy 2X2-es sakktabla bejarasa egy kirallyal ugy,
hogy a kovetkezo koltsegek mellett optimalis megoldast kapjunk:

balra: 2; fel balra: 1; fel: 4; fel jobbra: 2;
jobbra: 3; le jobbra: 2; le: 4; le balra: 1;

Adja meg a kezdopoziciot (a szamozas 0-tol kezdodik!):

Sor (0-1): 0
Oszlop (0-1): 0
1. lepes: (0, 0)
2. lepes: (1, 1)

3. lepes: (1, 0)
4. lepes: (0, 1)

Osszkoltseg: 6