Mesterséges intelligencia

gyakorlat

4. feladat

(A* algoritmus)

Készítette:

Szathmáry László
II. PM.

Feladat: egy általam tetszőlegesen kiválasztott feladat megoldásainak megkeresése A* algoritmus segítségével.

Lóugrás

Egy 8X8-as sakktáblán adott két mező. Hogyan lehet egyikből a másikba lóugrásokkal eljutni?

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#define  N 8

class Huszar;
class List {
 friend class Huszar;                    //mert majd a Huszar osztály tagfv.-eiből módosítjuk
 private:
   char *ut;
   int  koltseg, heur;
   List *kov;
};
class Huszar {
 public:
   Huszar();                             //konstruktor
   void vezerlo();
   void kiir();
 private:
   char tomb[N][N];                      //ez lesz az állapottér
   List *fej, *akt;
   int balra_le_k, balra_fel_k, fel_balra_k, fel_jobbra_k,
         jobbra_fel_k, jobbra_le_k, le_jobbra_k, le_balra_k, min_koltseg;          //költségek
   int start_x, start_y, end_x, end_y;

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

   int balra_le_ef(int, int);      //1.            //az egyes op.-ok s előfeltételeik
   void balra_le(int, int);
   int balra_fel_ef(int, int);     //2.
   void balra_fel(int, int);
   int fel_balra_ef(int, int);     //3.
   void fel_balra(int, int);
   int fel_jobbra_ef(int, int);    //4.
   void fel_jobbra(int, int);
   int jobbra_fel_ef(int, int);    //5.
   void jobbra_fel(int, int);
   int jobbra_le_ef(int, int);     //6.
   void jobbra_le(int, int);
   int le_jobbra_ef(int, int);     //7.
   void le_jobbra(int, int);
   int le_balra_ef(int, int);      //8.
   void le_balra(int, int);
```

};

I. Állapottér megadása

Az állapottér egy 8X8-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. (Ha megengednénk, hogy egy mezőt többször érintsünk, akkor felesleges körutak jöhetnének létre). 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, heurisztikából és egy mutatóból fog állni).

Mivel A* algoritmust akarunk megvalósítani, ezért minden egyes csúcs esetén le kell tárolni az addig megtett út költségét (g(n)), ill. a heurisztikának (h(n)) olyannak kell lenni, hogy alulról becsülje a még hátralévő optimális út költségét ((h(n)<=h*(n)). Az A* alg. tehát hátra és előre is tekint, s "speciális" heurisztikája miatt optimális megoldást fog adni, ha létezik megoldás.

II. Kezdőállapot megadása

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

```
Huszar::Huszar()
{
  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_le_k=2;   balra_fel_k=3; fel_balra_k=1; fel_jobbra_k=1;
  jobbra_fel_k=3; jobbra_le_k=2; le_jobbra_k=4; le_balra_k=4;

  int koltsegek[8] = {balra_le_k, balra_fel_k, fel_balra_k, fel_jobbra_k,
        jobbra_fel_k, jobbra_le_k, le_jobbra_k, le_balra_k};
  for (i=1, min_koltseg=koltsegek[0]; i<8; ++i)
    if (koltsegek[i]<min_koltseg) min_koltseg = koltsegek[i];

  beker();

  fej = new List; if (!fej) hiba();
  char *temp = "_\0"; temp[0]=tomb[start_x][start_y];
  fej->ut = strdup(temp); if (!(fej->ut)) hiba();
  fej->koltseg=0; fej->heur = heurisztika(start_x, start_y);
  fej->kov=NULL;
}
```

A tömböt (**tomb**) karakterekkel töltjük fel (egy karakter így egy mezőt egyértelműen fog majd azonosítani).

Az egyes operátorokhoz költség tartozik, ezeket is meghatározzuk.

A legkisebb költségű operátor költségét egy **min_koltseg** nevű változóban tároljuk le. Erre a csúcsokbeli heurisztika meghatározásánál lesz szükség:

```
int Huszar::heurisztika(int x, int y)
{
  int sum = (abs(x - end_x) + abs(y - end_y))/3;
  return (sum * min_koltseg);
}
```

Megnézzük, hogy az adott pozíció a célpozíciótól milyen messze van X ill. Y irányban. Vesszük a kettő összegét: így megkapjuk, hogy x és y irányban összesen hányat kell még lépni, hogy a célpozícióba érkezzünk. Egy lóugrással 3 mezőt tudunk lépni, vagyis ebből a távolságból 3-at tudunk "lefaragni" 1 lépéssel. Ha tehát a távolságot elosztjuk 3-mal, s vesszük ennek az egész részét (a C egész egésszel való osztásakor automatikusan az eredmény egész részét adja vissza, ezt kihasználtuk), akkor megkapjuk, hogy legjobb esetben is legalább hány lépésre van szükség. Ez láthatóan egy alsó becslést adott a lépések számára. Ha ezt még megszorozzuk a legkisebb költségű operátor költségével, akkor a még hátralévő út költségére egy alsó becslést adtunk meg (h(n)<=h*(n)), s A* alg. esetén egy ilyenre van szükség => optimális megoldást fogunk kapni.

Visszatérve a kezdőállapot megadásához: meghívjuk a **beker()** függvényt, melynek feladata a felhasználó tájékoztatása, ill. a kezdő- és célpozíció bekérése:

```
void Huszar::beker()
{
  clrscr();
  printf("Feleadat: egy %dX%d-es sakktablan kijelolunk ket mezot, s az\n"
         "egyikbol egy huszarral el kell jutni a masikba. Optimalis\n"
         "megoldast keresunk a kovetkezo koltsegek mellett:\n\n"
         "balra  le:  %2d; balra fel: %2d;  fel balra: %2d;  fel jobbra: %2d;\n"
         "jobbra fel: %2d; jobbra le: %2d;  le jobbra:  %2d; le   balra: %2d;\n\n"
         ,N,N, balra_le_k,   balra_fel_k, fel_balra_k, fel_jobbra_k,
               jobbra_fel_k, jobbra_le_k, le_jobbra_k, le_balra_k);
  printf("Adja meg a start- es a celmezo koordinatait!\n"
         "A szamozas 0-tol %d-ig terjed!\n\n", N-1);
  cout << "Adja meg a kezdopoziciot:\n" << "X: "; cin >> start_x;
  cout << "Y: "; cin >> start_y;
  cout << "Adja meg a celpoziciot:\n" << "X: "; cin >> end_x;
  cout << "Y: "; cin >> end_y;
  if (start_x == end_x && start_y == end_y)
  {
    cout << "Mivel a ket pozicio megegyezik, ezert oda 0 koltseggel jutunk el.\n";
    exit(-1);
  }
}
```

Miután sikerült meghatározni, hogy a táblán honnan hová szeretnénk eljutni megtörténhet a láncolt lista inicializálása.

A listának kezdetben egyetlen egy eleme lesz. Ennek lefoglaljuk a tárterületet, majd feltöltjük a rekord mezőit: az eddig megtett út hossza 1, azaz annak a pozíciónak az azonosítóját helyezzük el, ahol éppen állunk.
A költség a start csúcsban 0, heurisztikáját pedig meghatározzuk. Mivel nincs rákövetkező elem, ezért a **kov** mutatót NULL-ra állítjuk. A későbbiekben ezt a start csúcsot fogjuk majd kiterjeszteni, s így fog bővülni a lista.

III. Végállapot megadása

```
int Huszar::vege()
{
  int hossz = strlen(akt->ut)-1, x,y;
  char *p   = akt->ut;
  for (int i=0; i<N; ++i)                  //a mezőazonosító karakterből előállítjuk
    for (int j=0; j<N; ++j)                //annak táblabeli x,y koordinátáit
      if (tomb[i][j]==p[hossz]) x=i, y=j;

  return (x==end_x && y==end_y);
}
```

A keresésnek akkor van vége, ha a kiterjesztésre kiválasztott csúcsot tesztelve azt tapasztaljuk, hogy annak pozíciója megegyezik a célcsúcs pozíciójával.

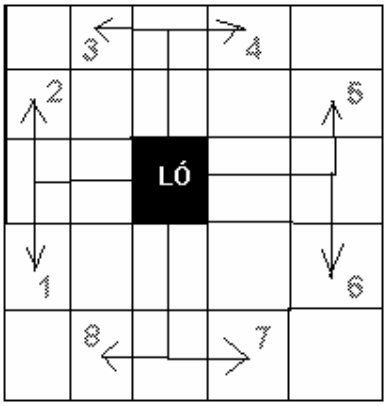
IV. Operátorok

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

Előfeltétel: mindegyik kap egy x,y pozíciót, s meg kell vizsgálnia, hogy innen a megfelelő irányba lehet-e lépni. Ahova lépünk, annak a pozíciónak nem szabad a táblán kívülre esnie, ill. üresnek kell lennie (vagyis a letárolt útban, mint karaktersorozatban nem szerepelhet annak a mezőnek az azonosítója).

Operátor: kap egy x,y pozíciót, mely azt jelenti, hogy onnan kell lépnie a megfelelő irányba.

A következő ábra a 8 lehetséges lépést szemlélteti:



1.: balra le (2 balra, 1 le)
2.: balra fel (2 balra, 1 fel)
3.: fel balra (2 fel, 1 balra)
4.: fel jobbra (2 fel, 1 jobbra)
5.: jobbra fel (2 jobbra, 1 fel)
6.: jobbra le (2 jobbra, 1 le)
7.: le jobbra (2 le, 1 jobbra)
8.: le balra (2 le, 1 balra)

Általános megjegyzés:

A könnyebb áttekinthetőség kedvéért hívjuk a 0. indexszel jelölt sort (oszlopot) első sornak (oszlopnak), a 7. indexszel jelölt sort (oszlopot) pedig utolsó sornak (oszlopnak), stb.

Az előfeltételekhez:

Mivel a C rövidzár kiértékelést vall, ezért az ÉS-sel (&&) összekötött feltételek esetén ha bármelyik feltétel hamis, akkor hamis értékkel (0) fog a függvény visszatérni.

```
int Huszar::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 nem jártunk-e már ott. Akkor voltunk már ott, ha az x,y pozícióhoz tartozó mezőazonosító karakter már szerepelt az aktuális csúcs **ut**-jában (mint karaktersorozatban).

```
/* 1. */
int Huszar::balra_le_ef(int x, int y)
{
  if (x<=N-2 && y>=2 && ures(x+1, y-2)) return 1;
  else return 0;
}
```

Megvizsgálandó, hogy onnan (x,y pozíció) lehet-e balra le lépni (azaz x<= mint az utolsó előtti sor ÉS y>= mint a harmadik oszlop), vagyis a célpozíció a táblán belül van-e, s ha ez is teljesül, akkor a célpozíciónak üresnek kell lennie.

```
void Huszar::balra_le(int x, int y)
{
  beszur(x+1, y-2, balra_le_k, heurisztika(x+1, y-2));
}
```

Meghívunk egy olyan függvényt, amelyet a paraméterezés segítségével az összes operátor meghívhat, használhat. Paraméterként azt az x,y pozíciót kell megadni, AHOVA lépünk + ennek a lépésnek a költségét + az így létrejövő új nyílt csúcs heurisztikáját.

```
/* 2. */
int Huszar::balra_fel_ef(int x, int y)
{
  if (x>=1 && y>=2 && ures(x-1, y-2)) return 1;
  else return 0;
}
```

Megvizsgálandó, hogy onnan (x,y pozíció) lehet-e balra fel lépni (azaz x>= mint a második sor ÉS y>= mint a harmadik oszlop), vagyis a célpozíció a táblán belül van-e, s ha ez is teljesül, akkor a célpozíciónak üresnek kell lennie.

```
void Huszar::balra_fel(int x, int y)
{
  beszur(x-1, y-2, balra_fel_k, heurisztika(x-1, y-2));
}
```

Balra fel lépünk, ebbe az irányba kiterjesztjük az aktuális csúcsot.

```
/* 3. */
int Huszar::fel_balra_ef(int x, int y)
{
  if (x>=2 && y>=1 && ures(x-2, y-1)) return 1;
  else return 0;
}
```

Megvizsgálandó, hogy onnan (x,y pozíció) lehet-e fel balra lépni (azaz x>= mint a harmadik sor ÉS y>= mint a második oszlop), vagyis a célpozíció a táblán belül van-e, s ha ez is teljesül, akkor a célpozíciónak üresnek kell lennie.

```
void Huszar::fel_balra(int x, int y)
{
  beszur(x-2, y-1, fel_balra_k, heurisztika(x-2, y-1));
}
```

Fel balra lépünk, ebbe az irányba kiterjesztjük az aktuális csúcsot.

```
/* 4. */
int Huszar::fel_jobbra_ef(int x, int y) {
  if (x>=2 && y<=N-2 && ures(x-2, y+1)) return 1;
  else return 0;
}
```

Megvizsgálandó, hogy onnan (x,y pozíció) lehet-e fel jobbra lépni (azaz x>= mint a harmadik sor ÉS y<= mint az utolsó előtti oszlop), vagyis a célpozíció a táblán belül van-e, s ha ez is teljesül, akkor a célpozíciónak üresnek kell lennie.

```
void Huszar::fel_jobbra(int x, int y)
{
  beszur(x-2, y+1, fel_jobbra_k, heurisztika(x-2, y+1));
}
```

Fel jobbra lépünk, ebbe az irányba kiterjesztjük az aktuális csúcsot.

```
/* 5. */
int Huszar::jobbra_fel_ef(int x, int y)
{
  if (x>=1 && y<=N-3 && ures(x-1, y+2)) return 1;
  else return 0;
}
```

Megvizsgálandó, hogy onnan (x,y pozíció) lehet-e jobbra fel lépni (azaz x>= mint a második sor ÉS y<= mint a hatodik oszlop), vagyis a célpozíció a táblán belül van-e, s ha ez is teljesül, akkor a célpozíciónak üresnek kell lennie.

```
void Huszar::jobbra_fel(int x, int y)
{
  beszur(x-1, y+2, jobbra_fel_k, heurisztika(x-1, y+2));
}
```

Jobbra fel lépünk, ebbe az irányba kiterjesztjük az aktuális csúcsot.

```
/* 6. */
int Huszar::jobbra_le_ef(int x, int y)
{
  if (x<=N-2 && y<=N-3 && ures(x+1, y+2)) return 1;
  else return 0;
}
```

Megvizsgálandó, hogy onnan (x,y pozíció) lehet-e jobbra le lépni (azaz x<= mint az utolsó előtti sor ÉS y<= mint a hatodik oszlop), vagyis a célpozíció a táblán belül van-e, s ha ez is teljesül, akkor a célpozíciónak üresnek kell lennie.

```
void Huszar::jobbra_le(int x, int y)
{
  beszur(x+1, y+2, jobbra_le_k, heurisztika(x+1, y+2));
}
```

Jobbra le lépünk, ebbe az irányba kiterjesztjük az aktuális csúcsot.

```
/* 7. */
int Huszar::le_jobbra_ef(int x, int y)
{
  if (x<=N-3 && y<=N-2 && ures(x+2, y+1)) return 1;
  else return 0;
}
```

Megvizsgálandó, hogy onnan (x,y pozíció) lehet-e le jobbra lépni (azaz x<= mint a hatodik sor ÉS y<= mint az utolsó előtti oszlop), vagyis a célpozíció a táblán belül van-e, s ha ez is teljesül, akkor a célpozíciónak üresnek kell lennie.

```
void Huszar::le_jobbra(int x, int y)
{
  beszur(x+2, y+1, le_jobbra_k, heurisztika(x+2, y+1));
}
```

Le jobbra lépünk, ebbe az irányba kiterjesztjük az aktuális csúcsot.

```
/* 8. */
int Huszar::le_balra_ef(int x, int y)
{
  if (x<=N-3 && y>=1 && ures(x+2, y-1)) return 1;
  else return 0;
}
```

Megvizsgálandó, hogy onnan (x,y pozíció) lehet-e le balra lépni (azaz x<= mint a hatodik sor ÉS y>= mint a második oszlop), vagyis a célpozíció a táblán belül van-e, s ha ez is teljesül, akkor a célpozíciónak üresnek kell lennie.

```
void Huszar::le_balra(int x, int y)
{
  beszur(x+2, y-1, le_balra_k, heurisztika(x+2, y-1));
}
```

Le balra lépünk, ebbe az irányba kiterjesztjük az aktuális csúcsot.

```
void Huszar::beszur(int sor, int oszlop, int koltseg, int heur)
{
  List *uj = new List; if (!uj) hiba();

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

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

  List *temp = fej, *elozo=NULL;
  while (temp)
  {
    p = temp->ut;
    if ( p[strlen(temp->ut)-1] == tomb[sor][oszlop] )
          if (uj->koltseg + uj->heur < temp->koltseg + temp->heur)
          {
            if (!elozo)                                    //legelső listaelem kifűzése
```

```
          {
            temp = temp->kov;
            free(fej->ut); delete fej; fej = temp;
          }
          else                                          //közbenső listaelem kifűzése
          {
            elozo->kov = temp->kov;
            free(temp->ut); delete temp;
          }
          uj->kov = fej; fej = uj;                      //az uj listaelem hozzáadása
          return;
        }
        else
        {
          free(uj->ut); delete uj;                      //az uj elem törlése
          return;
        }
    else elozo=temp, temp = temp->kov;
  }
  uj->kov = fej; fej = uj;
}
```

A **beszur()** fv. feladata, hogy a láncolt listába beszúrja a kiterjesztés során újonnan előálló és így nyílttá váló csúcsot.
Először is lefoglalódik neki a tárterület, s a rekord mezői feltöltődnek adatokkal: az **ut** örökli annak a csúcsnak az **ut** részét, amelyet kiterjesztettünk, s még kibővül egy karakterrel: annak a mezőnek az azonosítójával, ahova léptünk. Költség: a szülő költsége + a megfelelő irányba való lépés költsége. Meghatározzuk a heurisztikát is.

Ha ez a csúcs még nem szerepelt a nyílt csúcsok között, akkor beszúrjuk (külső **if** szerkezet: **temp** mutató végigszalad a listán). Ha sehol sincs egyezés, akkor be lehet szúrni, egy új nyílt csúcsot kaptunk.
Ha valahol volt egyezés, akkor jön a belső **if**: ha az új csúcs költsége+heurisztikája kisebb, mint a listában letárolté, akkor frissítés: listából törlés, helyébe az új elem behelyezése. Ha egy csúcsba egy nagyobb vagy egyenlő (költség+heurisztika)-val jutottunk, akkor marad a régi; az újat nem fogjuk befűzni, sőt fel is szabadítjuk az általa lefoglalt tárterületet.

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

  keres(x, y);
  int z=0;                          //ez a Borland C++ 3.1 fordító miatt van itt
  while (!vege())                   //nélküle a while blokk közepébe ugrott
  {
    if (balra_le_ef(x,y))
        balra_le(x,y);              //1.
    if (balra_fel_ef(x,y))
        balra_fel(x,y);             //2.
    if (fel_balra_ef(x,y))
        fel_balra(x,y);             //3.
    if (fel_jobbra_ef(x,y))
        fel_jobbra(x,y);            //4.
    if (jobbra_fel_ef(x,y))
        jobbra_fel(x,y);            //5.
```

```
    if (jobbra_le_ef(x,y))
        jobbra_le(x,y);          //6.
    if (le_jobbra_ef(x,y))
        le_jobbra(x,y);          //7.
    if (le_balra_ef(x,y))
        le_balra(x,y);           //8.
    torol();
    keres(x, y);
  }
}
```

A vezérlő először meghívja a **keres** nevű fv.-t, melynek feladata: az **akt** mutatót a legkisebb (költség+heurisztika)-jú listaelemre állítja, majd az ebben letárolt út alapján megállapítja, hogy a táblán hol helyezkedik el a huszár => ezt a vezérlő az x,y változóban visszakapja (a "cím" szerinti paraméterátadás miatt), s így majd ezt a csúcsot próbálja meg kiterjeszteni az összes lehetséges irányba. A kiterjesztés után már nincs szükség a listában az **akt** csúcsra (amelyiket éppen kiterjesztettünk), így ki lehet törölni. Ismét keresünk egy csúcsot, hogy melyiket kellene kiterjeszteni. Kiválasztás után persze mindig tesztelni kell => akkor lesz vége a keresésnek, ha a kiterjesztésre kiválasztott csúcsot tesztelve azt tapasztaljuk, hogy az kielégíti a célfeltételt, vagyis ha a keresett csúcsban vagyunk.

```
void Huszar::keres(int &x, int &y)          //"cím" szerinti paraméterátadás
{
  List *futo = fej->kov;

  akt = fej;
  while (futo)
  {
    if (futo->koltseg + futo->heur < akt->koltseg + akt->heur) akt = futo;
    futo = futo->kov;
  }

  int hossz = strlen(akt->ut)-1;          //azonosítóból x,y koordináták visszanyerése
  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;
}


void Huszar::torol()
{
  List *elozo = NULL, *temp=fej;

  while (temp!=akt) elozo=temp, temp=temp->kov;

  if (!elozo)
  {
    temp=fej->kov;
    free(fej->ut); delete fej; fej = temp;
  }
  else
```

```
  {
    elozo->kov = temp->kov;
    free(akt->ut); delete akt;
  }
}
```

Ez egyszerűen csak kitörli a listából az **akt** mutatóval megjelölt elemet.Mivel ezt már minden lehetséges irányba kiterjesztettük, ő így zárttá vált => nincs már rá szükség.

```
void Huszar::hiba()
{
  cerr << "Memoriafoglalasi hiba lepett fel.\n";
  exit(-1);
}
```

Dinamikus tárfoglalás esetén illik mindig leellenőrizni a művelet sikerességét. Ha nem volt sikeres a memóriaallokáció, akkor fatális hiba lépett fel, kilépünk a programból, de előtte még küldünk egy hibaüzenetet.

```
void Huszar::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);
}
```

akt mutató éppen arra a csúcsra mutat, amely megfelelt a célfeltételnek. **akt ut**-jában megtalálható azon mezőazonosítók sorozata, amely úton eljutottunk a start csúcsból a célcsúcsba. Ezen azonosítókat visszaalakítjuk x,y koordinátákká, kiírjuk az egyes lépéseket, majd az összköltséget is.

```
void main()
{
  Huszar huszar;                    //példányosítás
  huszar.vezerlo();
  huszar.kiir();
}
```

A főprg.-ban példányosítunk, meghívjuk a vezérlőt, mely addig keres, míg megoldást nem talál, majd ha ez megvan, akkor kiíratjuk az eredményt.

Futási eredmény:

Feladat: egy 8X8-as sakktablan kijelolunk ket mezot, s az egyikbol egy huszarral el kell jutni a masikba. Optimalis megoldast keresunk a kovetkezo koltsegek mellett:

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

Adja meg a start- es a celmezo koordinatait!
A szamozas 0-tol 7-ig terjed!
Adja meg a kezdopoziciot:
X: 0
Y: 0
Adja meg a celpoziciot:
X: 6
Y: 4

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

Osszkoltseg: 12