МАТЕМАТИЧКА ГИМНАЗИЈА

МАТУРСКИ РАД

из предмета Програмирање и програмски језици

Логичка игра Сокобан

Ученик **Ментор**

Владимир Плескоњић **проф. Ана Зековић**

1510Б07

Београд, јун 2011.

Логичка игра Сокобан

Сокобан је игра која је развијена у Јапану и објављена 1982. године. Након првобитног објављивања, прављени су бројни наставци као и разне њене варијације. Игра се састоји од великог броја нивоа у којима је циљ померити све кутије на унапред обележена поља, узимајући у обзир одређена правила при кретању и померању.

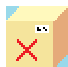
Правила игре

Сваки ниво је представљен у облику табличне дворане испуњене зидовима и собама, која садржи одређен пар кутија и циљних поља (циљева), као и једног играча којим корисник управља. Он се може кретати за по једно поље горе, доле, лево или десно. Играч се не може наћи на пољу заједно са зидом или кутијом. Ако покуша да стане на поље на коме се налази кутија, играч ће заузети то поље, а кутија ће бити померена за једно поље даље у правцу кретања играча, под условом да се ту не налазе зид или друга кутија. Овим се остварује гурање кутије, али је није могуће и вући. Ниво је решен тек онда када играч успе да помери све кутије на циљеве. Није битно која се кутија налази на ком циљу.

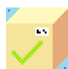
Објекти у игри

Игра се састоји од четири објекта: играча, кутија, циљева и зидова. Слике које их представљају се налазе у фолдеру **slike**.

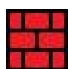
Играч – Корисник управља њиме уз помоћ тастера горе, доле, лево и десно.



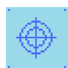
Кутија – Објекат који играч гурањем помера по нивоу да би га решио.



(Овако изгледа кутија док се налази циљном пољу.)



Зид – Играч не може стати нити гурнути кутију на поље на коме се налази зид.



Циљ – Објекат на који треба поставити кутију. Ниво је решен тек онда када се на сваком циљу нађе кутија.

Структура пројекта

Игра Сокобан се састоји из две апликације. Једна служи за играње, а друга за осмишљавање и креирање нових нивоа. Пројекат такође садржи и један **.dll** фајл у коме се налази метода за решавање нивоа.

Апликација за играње

Служи за играње Сокобана и решавање нивоа. Након покретања нивоа, кориснику се пружају могућности да га покрене испочетка и да се врати неколико потеза уназад у случају да је починио грешку при решавању. Такође, програм може да покуша да сам нађе решење нивоа, односно да испише поруку да је то немогуће учинити ако је заиста тако. Сам програм поседује једанаест променљивих:

```
Polje_Objekti[,] O;
Polje_Ciljevi[,] C;
int nw, nh, brp, granica, r;
string nivo;
int[] potezi;
Resenje R;
bool[] kpom;
```

Контроле **Polje_Objekti** и **Polje_Ciljevi** садрже по један **picturebox** и једну променљиву типа **char** који приказују ситуацију на пољу које представљају. **Polje_Ciljevi** служи за представљање циљева, док **Polje_Objekti** служи за све остале типове објеката. Вредност променљиве типа **char** назване **simbol** у контроли **Polje_Objekti** је “@” за случај да се на пољу које представља налази играч, “+” ако се ту истовремено налазе играч и циљно поље, “#” ако се ту налази зид, “*O*” ако се ту налази кутија, а “*Q*” ако се на том пољу налазе истовремено кутија и циљ. Истоимена променљива у контроли **Polje_Ciljevi** узима вредност “.” ако је њено поље циљно. У обе контроле бланко знак означава да се на њиховом пољу не налази ништа. Како су нивои представљени у облику таблице поља испуњених објектима, користимо променљиве **nw** и **nh** да нам представе димензије те таблице. Програм омогућава кориснику да се враћа уназад кроз одигране потезе, а променљиве **granica**, **brp**, **kpom** и **potezi** служе управо томе. У низу **potezi** се чувају вредности које означавају правце у којима се корисник кретао у разним потезима, док низ **kpom** говори да ли је у тим потезима померена кутија. Променљива **brp** говори по колико је потеза тренутно сачувано у ова два низа. Променљиве **R** и **r** служе представању решења нивоа до ког је рачунар дошао. Променљива **R** чува потезе неопходне за прелазак нивоа, док променљива **r** садржи индекс следећег потеза које доводи до решења. Коначно, променљива **nivo** садржи име нивоа који корисник тренутно решава, и помаже при његовом поновном учитавању за случај да корисник жели да крене да га решава испочетка.

При покретању програма се извршава наведени код.

```
private void Form1_Load(object sender, EventArgs e)
{
 nw = 30;
 nh = 20;
 nivo = "";
 granica = 500000;
 RestartujPoteze();
 r = 0;

 O = new Polje_Objekti[nw, nh];
 C = new Polje_Ciljevi[nw, nh];
 for (int i = 0; i < nw; i++)
  for (int j = 0; j < nh; j++)
  {
   O[i, j] = new Polje_Objekti();
   O[i, j].Location = new Point(4 + i * 32, 4 + j * 32);
   Controls.Add(O[i, j]);

   C[i, j] = new Polje_Ciljevi();
   C[i, j].Location = new Point(4 + i * 32, 4 + j * 32);
   Controls.Add(C[i, j]);
  }

 Width = nw * 32 + 108;
 Height = nh * 32 + 34;
 button1.Location = new Point(nw * 32 + 22, button1.Location.Y);
 button2.Location = new Point(nw * 32 + 22, button2.Location.Y);
 button3.Location = new Point(nw * 32 + 22, button3.Location.Y);
 button4.Location = new Point(nw * 32 + 22, button4.Location.Y);
 button5.Location = new Point(nw * 32 + 22, button5.Location.Y);
 label1.Location = new Point(nw * 32 + 8, label1.Location.Y);
 label2.Location = new Point(nw * 32 + 8, label2.Location.Y);
 textBox1.Location = new Point(nw * 32 + 8, textBox1.Location.Y);
}

private void RestartujPoteze()
{
 kpom = new bool[granica];
 for (int i = 0; i < granica; i++) kpom[i] = false;
 potezi = new int[granica];
 for (int i = 0; i < granica; i++) potezi[i] = -1;
 brp = 0;
 label2.Text = "Број потеза: " + Convert.ToString(brp);
 R = new Resenje();
 button4.Enabled = false;
}
```

У овом коду се иницијализују променљиве програма, постављају се **button**-и као и све контроле **Polje_Objekti** и **Polje_Ciljevi**. Овде се користе променљиве **nw** и **nh** да говоре колико контрола **Polja_Objekti** и **Polja_Ciljevi** треба поставити и на којим координатама у форми треба поставити **button**-е. **RestartujPoteze** служи да иницијализује низове у којима се памте потези играча као и потези решења до ког је рачунар дошао. Она се позива и при сваком новом покретању нивоа.

Корисник управља играчем уз помоћ тастера горе, доле, лево и десно. Ово се остварује у **event**-у **KeyDown**. Прво се проналази позиција играча у нивоу, а затим се играч помера у захтеваном смеру ако је то могуће и гура кутију у случају да може. Потез се памти у низу **potezi** (број **1** означава да је играч померен за једно поље нагоре, **2** да је померен надоле, **3** да је померен улево, а **4** да је померен удесно; исто важи и на свим другим местима у пројекту где је потребно памтити кретање играча у виду бројева) и у низ **kpom** се додаје вредност **true** уколико је играч гурнуо кутију, а потом програм проверава да ли је ниво решен. Ако јесте, корисник се обавештава о постигнутом успеху, а ниво се уклања.

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
 Point P = PozicijaIgraca;
 int p = P.X; int q = P.Y;
 bool k = false;
 if (p != -1 && q != -1)
 {
  if (e.KeyCode == Keys.Up && q > 0)
  {
   if (q > 0 && O[p, q - 1].Simbol == ' ')
   {
    O[p, q - 1].Simbol = '@';
    if (C[p, q - 1].Simbol == '.') O[p, q - 1].Simbol = '+';
    O[p, q].Simbol = ' ';
    potezi[brp] = 1;
    brp++;
    button5.Enabled = true;
    R = new Resenje();
    button4.Enabled = false;
   }
   else if (q > 1)
   if ((O[p, q - 1].Simbol == 'O' || O[p, q - 1].Simbol == 'Q')
   && O[p, q - 2].Simbol == ' ')
   {
    O[p, q - 1].Simbol = '@';
    if (C[p, q - 1].Simbol == '.') O[p, q - 1].Simbol = '+';
    O[p, q].Simbol = ' ';
    O[p, q - 2].Simbol = 'O';
    if (C[p, q - 2].Simbol == '.') O[p, q - 2].Simbol = 'Q';
    kpom[brp] = true;
    potezi[brp] = 1;
    brp++;
    button5.Enabled = true;
    k = true;
    R = new Resenje();
    button4.Enabled = false;
   }
  }
```

(Део кода који обавља померање за случај да је кликнути тастер доле, лево или десно је уклоњен зато што је аналоган случају за тастер горе.)

```
  label2.Text = "Број потеза: " + Convert.ToString(brp);
  if (k) ProveriKraj();
 }
}
```

```
private bool KrajNivoa()
{
 for (int i = 0; i < nw; i++)
  for (int j = 0; j < nh; j++)
   if (O[i, j].Simbol == 'O') return false;
 return true;
}

private void ProveriKraj()
{
 if (KrajNivoa())
 {
  MessageBox.Show("Честитам, прешли сте ниво за " +
   Convert.ToString(brp) + " потеза!");
  for (int i = 0; i < nw; i++)
   for (int j = 0; j < nh; j++)
   {
    O[i, j].Simbol = ' ';
    C[i, j].Simbol = ' ';
   }
  button2.Enabled = false;
  button3.Enabled = false;
  button4.Enabled = false;
  button5.Enabled = false;
  RestartujPoteze();
 }
}
```

Програм садржи пет **button**-а:

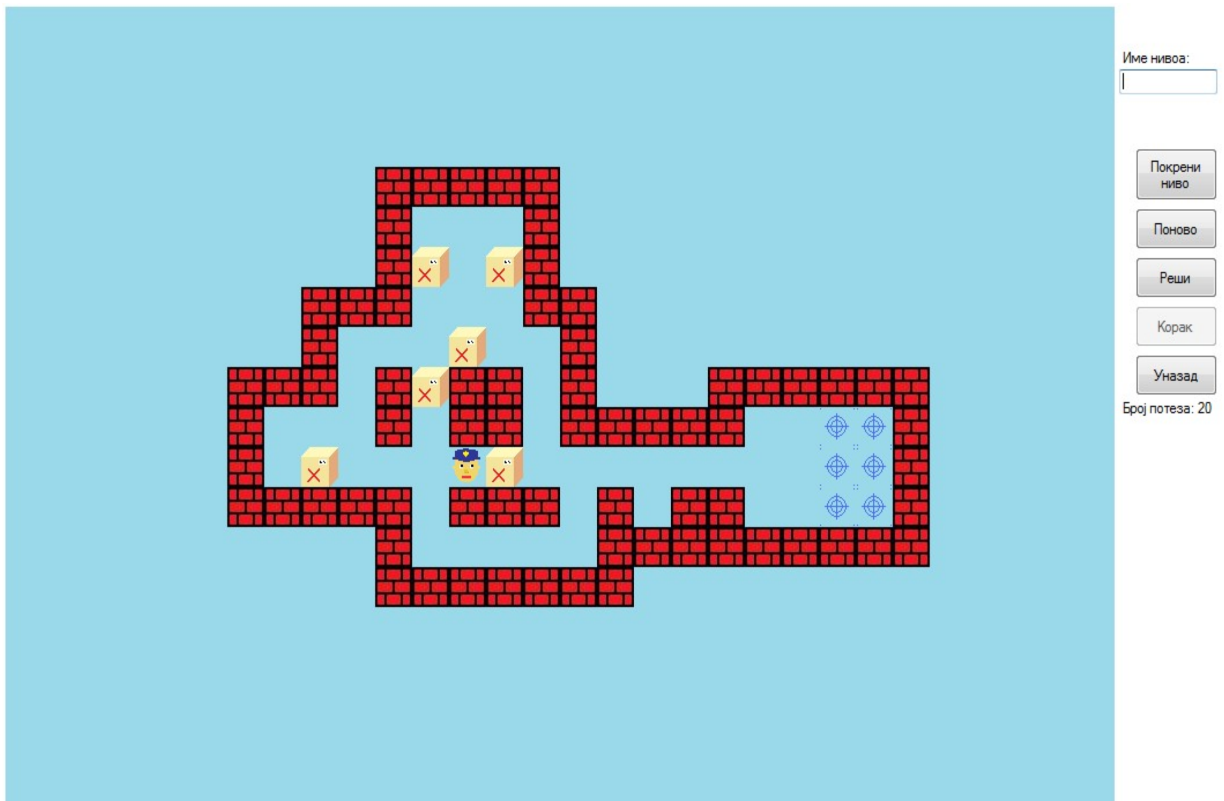
Покрени ниво (**button1**): Након што унесе име нивоа који жели да решава (без екстензије) корисник кликће на ово дугме и програм учитава и покреће тај ниво. За случај да корисник није унео име или је уписао име непостојећег нивоа програм ће га обавестити о грешци. На друге **button**-е се не може кликнути пре него што се ниво учита.

Поново (**button2**): Кликом на ово дугме ниво ће се поново учитати, омогућавајући кориснику да почне да га решава испочетка.

Реши (**button3**): Кликом на ово дугме се позива метода класе **Resavac** која покушава да реши ниво. Уколико није могуће решити ниво програм ће о томе обавестити корисника. У противном, програм ће сачувати нађено решење и обавестити корисника о својој успешности.
За време тражења решења програм се паузира. Треба напоменути да би се за поједине нивое овај процес могао дуго извршавати, што треба узети у обзир пре његовог покретања.

Корак (**button4**): Након што корисник кликне на ово дугме програм ће одиграти један потез свог решења. На ово дугме се не може кликнути пре него што програм пронађе решење, нити након што корисник одигра свој потез.

Уназад (**button5**): Кликом на ово дугме играч се враћа за једно поље уназад. Уколико се на овај начин ниво врати на своје почетно стање на ово дугме више није могуће кликнути до следећег одиграног потеза. Такође, ово дугме се може користити да би се вратило за један потез решења које је програм нашао, у случају да је на дугме **Корак** могуће кликнути.



Апликација за прављење нивоа

Служи да омогући кориснику да направи сопствене нивое. Да би то урадио, корисник додаје објекте у таблицу поља, а потом куца његово име и кликће на дугме **Сачувај**. Нивои се меморишу као текстуални фајлови где један симбол представља почетно стање на одговарајућем пољу ("@" означава играча, "+" означава играча који стоји на циљу, "#" означава зид, "*O*" означава кутију, "*Q*" означава кутију на циљу, "." означава циљ, а бланко знак означава празну позицију). Нивои се налазе у фолдеру **nivoi**.

Сам програм садржи три променљиве:

```
Polje[,] P;
int nw, nh;
```

Променљиве **nw** и **nh** имају исту улогу као и истоимене променљиве у апликацији за играње. Променљива **P** је матрица контрола назива **Polje**. **Polje** је контрола која садржи један **picturebox**, као и две променљиве типа **char** назива **simbol** и **novisimb**. Променљива **simbol** говори који се објекти налазе на одговарајућем пољу и утиче на то која се слика појављује у **picturebox**-у. Зато што корисник може да бира који објекат следећи жели да постави имамо променљиву **novisimb**. Она садржи симбол тог објекта. Кликом на **picturebox** ове контроле њен садржај се мења тако што се на њу поставља тражени објекат ако смо кликнуло левим дугметом миша, а брише се објекат ако смо кликнули десним. Могуће је поставити играча или кутију на поље које већ садржи циљ (и обрнуто). Све ово се остварује у **event**-у **MouseDown** за **picturebox** ове контроле. (**PromeniSliku** учитава слику из одговарајућег фајла и поставља је у **picturebox** контроле).

```
if (e.Button == MouseButtons.Left)
{
 if ((simbol == '.' && novisimb == 'O') ||
      (simbol == 'O' && novisimb == '.')) simbol = 'Q';
 else if ((simbol == '.' && novisimb == '@') ||
           (simbol == '@' && novisimb == '.')) simbol = '+';
 else simbol = novisimb;
 PromeniSliku();
}
else if (e.Button == MouseButtons.Right)
{
 if (simbol == 'Q') simbol = '.';
 else if (simbol == '+') simbol = '.';
 else simbol = ' ';
 PromeniSliku();
}
```

Слично као и код апликације за играње, при покретању програма се извршава код који иницијализује променљиве програма и поставља контроле на одговарајуће локације у форми.

```
private void Form1_Load(object sender, EventArgs e)
{
 nw = 30; nh = 20;
 P = new Polje[nw, nh];
 for (int i = 0; i < nw; i++)
  for (int j = 0; j < nh; j++)
  {
   P[i, j] = new Polje();
   P[i, j].Location = new Point(4 + i * 32, 4 + j * 32);
   Controls.Add(P[i, j]);
  }
 Width = nw * 32 + 115;
 Height = nh * 32 + 34;
 radioButton1.Location = new Point(nw * 32 + 10,
  radioButton1.Location.Y);
 radioButton2.Location = new Point(nw * 32 + 10,
  radioButton2.Location.Y);
 radioButton3.Location = new Point(nw * 32 + 10,
  radioButton3.Location.Y);
 radioButton4.Location = new Point(nw * 32 + 10,
  radioButton4.Location.Y);
 radioButton5.Location = new Point(nw * 32 + 10,
  radioButton5.Location.Y);
 button1.Location = new Point(nw * 32 + 17, button1.Location.Y);
 button2.Location = new Point(nw * 32 + 17, button2.Location.Y);
 button3.Location = new Point(nw * 32 + 17, button3.Location.Y);
 button4.Location = new Point(nw * 32 + 17, button4.Location.Y);
 button5.Location = new Point(nw * 32 + 17, button5.Location.Y);
 button6.Location = new Point(nw * 32 + 17, button6.Location.Y);
 button7.Location = new Point(nw * 32 + 17, button7.Location.Y);
 button8.Location = new Point(nw * 32 + 17, button8.Location.Y);
 label1.Location = new Point(nw * 32 + 8, label1.Location.Y);
 label2.Location = new Point(nw * 32 + 8, label2.Location.Y);
 textBox1.Location = new Point(nw * 32 + 8, textBox1.Location.Y);
 radioButton2.Checked = true;
}
```

Програм садржи пет **radiobutton**-а чијим се селектовањем бира објекат који корисник жели да постави. Након што корисник селектује неки **radiobutton** променљива **novisimb** се мења у сваком **Polju** матрице **P** одговарајућу вредност.

```
public void PromeniOcekivaniSimbol(char ch)
{
 novisimb = ch;
}

private void PromenaObjekta(char ch)
{
 for (int i = 0; i < nw; i++)
  for (int j = 0; j < nh; j++)
  {
   P[i, j].PromeniOcekivaniSimbol(ch);
  }
}
```

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
 PromenaObjekta('@');
}
```

(Аналогно за остале **radiobutton**-е.)

Програм садржи осам **button**-а:

Сачувај (**button1**): Након што заврши са креирањем нивоа, корисник кликће на ово дугме да би га програм сачувао у облику текстуалног фајла. Претходно је неопходно уписати име нивоа (без екстензије) у **textbox**. Програм неће дозволити да се ниво сачува ако корисник није поставио одговарајући број свих објеката или ако је ниво већ у решеном стању (ако су све кутије на циљевима). Да би ово проверио програм користи методу **BrojObjekta**.

```
private int BrojObjekta(char ch)
{
 int s = 0;

 for (int i = 0; i < nw; i++)
  for (int j = 0; j < nh; j++)
   if (P[i, j].Simbol == ch) s++;

 return s;
}
```

За случај да ниво са одабраним именом већ постоји, програм ће питати корисника да ли жели да обрише стари ниво са тим именом и сачува управо креирани.

Покрени (**button2**): Кликом на ово дугме се покреће ниво са наведеним именом, који корисник потом може да мења. У случају да корисник није унео име или је унео име непостојећег нивоа програм ће га обавестити о грешци.

Обриши све (**button3**): Служи да би се склонили сви објекти са свих поља.

```
for (int i = 0; i < nw; i++)
 for (int j = 0; j < nh; j++)
  P[i, j].Simbol = ' ';
```

Постави зидове (**button4**): Служи да би се на сва поља поставио објекат **зид**.

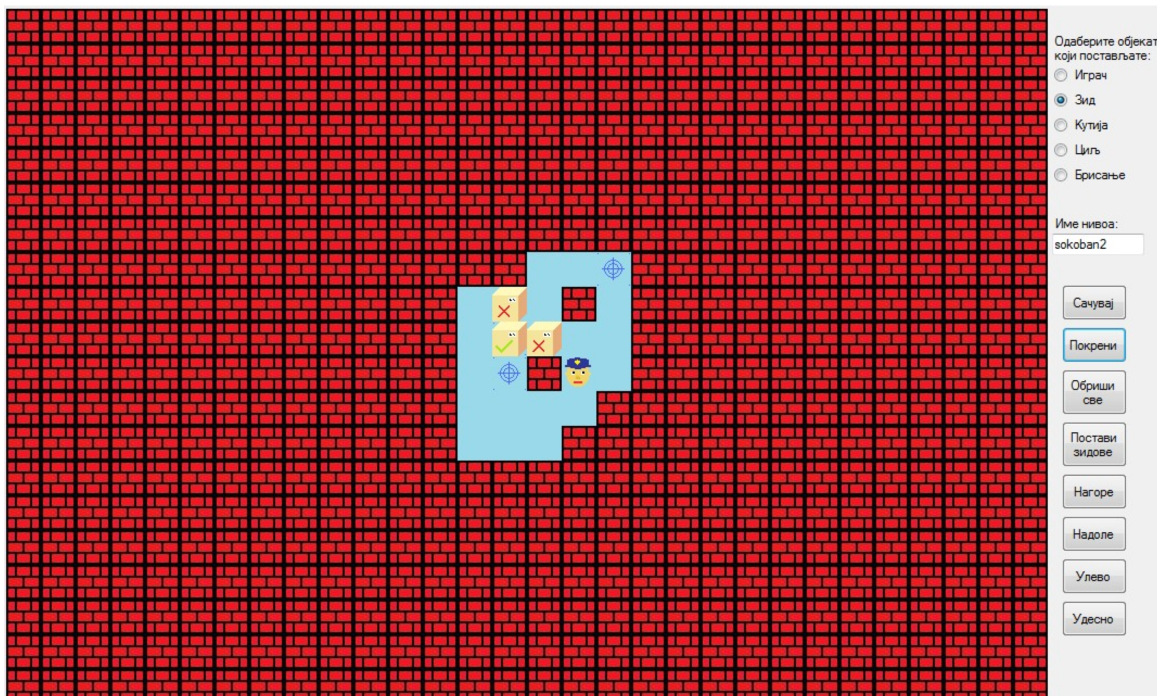
```
for (int i = 0; i < nw; i++)
 for (int j = 0; j < nh; j++)
  P[i, j].Simbol = '#';
```

Нагоре, **Надоле**, **Улево** и **Удесно** (**button5**, **button6**, **button7** и **button8**): Служе да би се садржај сваког поља померио за једно поље горе (односно доле, лево или десно).

Садржај граничних поља се помера на другу страну нивоа (на пример, ако кликнемо **Нагоре** сви објекти који се налазе на врху ће бити постављени на одговарајуће поље на дну нивоа).

```
private void button5_Click(object sender, EventArgs e)
{
 char T;
 for (int i = 0; i < nw; i++)
 {
  T = P[i, 0].Simbol;
  for (int j = 0; j < nh - 1; j++)
   P[i, j].Simbol = P[i, j + 1].Simbol;
  P[i, nh - 1].Simbol = T;
 }
}
```

(Аналогно изгледа и код за померање надоле, улево и удесно.)



Алгоритам решавања нивоа

У фајлу **KlasaZaResavanje.dll** се налазе класе и методе које омогућавају апликацији за играње да решава нивое.

Структура *namespace*-а

Namespace KlasaZaResavanje се састоји од девет класа:

Resavac: Ово је главна класа у **namespace**-у. При креирању прихвата садржај задатог нивоа, а садржи методу **NadjiResenje** која налази његово решење.

Resenje: Садржи низ бројева **R** који апликација за играње користи да памти потезе који воде решењу (приметимо да ова апликација садржи једну променљиву ове класе). Зато што се сама класа понаша као низ потребне су јој методе за додавање нових елемената и проширивање меморијског простора када је то потребно.

```
public void Prosiri()
{
 int[] R1 = new int[R.Length + 200];
 for (int i = 0; i < n; i++)
  R1[i] = R[i];
 R = R1;
}

public void Dodaj(int x)
{
 if (n == R.Length) Prosiri();
 R[n] = x;
 n++;
}
```

Такође садржи методу **Spoji** којом се једна променљива ове класе може спојити са другом.

```
public void Spoji(Resenje X)
{
 int[] P = new int[n + X.N];
 for (int i = 0; i < n; i++)
  P[i] = R[i];
 for (int i = 0; i < X.N; i++)
  P[n + i] = X.R[i];
 R = P;
 n = n + X.N;
}
```

Polja: Ради бржег решавања нивоа поља са одређеним особинама се маркирају и бележе у **bool** матрицама ове класе. Ово се дешава након креирања класе, а више речи о томе шта се тачно тада дешава биће касније.

Potez: Основна јединица кретања у алгоритму за решавање је једно померање кутије. Ова класа садржи три променљиве: **x**, **y** и **p**. Прве две памте позицију гурнуте кутије у потезу, док последња памти правац у коме је померена (има вредност **1** за горе, **2** за доле, **3** за лево и **4** за десно).

SkupPoteza: Ова класа се понаша као низ **Potez**-а. Због тога садржи методе **Dodaj** и **Prosiri** које могу да јој додају нови **Potez**, односно да прошире меморијски простор за памћење нових **Potez**-а.

Nivo: Памти све објекте који се налазе у нивоу као и њихове позиције уз помоћ матрица **Objekat** (типа **char**) и **Ciljevi** (типа **bool**). У ову класу се може поставити било који објекат из игре, а такође садржи и методе **PozicijaIgraca**, **SkloniIgraca**, **UnistiKutije**, **Kraj** (служи да провери да ли је ниво решен) и **Kopija** (ствара нову променљиву класе **Nivo** ако желимо да га мењамо током неких операција, али притом не желимо да променимо и оригинални **Nivo**).

```
public Point PozicijaIgraca()
{
 for (int i = 0; i < nw; i++)
  for (int j = 0; j < nh; j++)
   if (this[i, j] == '@')
    return new Point(i, j);

 return new Point(-1, -1);
}

public void SkloniIgraca()
{
 Point P = PozicijaIgraca();
 if (P.X >= 0 && P.X < nw && P.Y >= 0 && P.Y < nh)
  Objekat[P.X, P.Y] = ' ';
}

public void UnistiKutije()
{
 for (int i = 0; i < nw; i++)
  for (int j = 0; j < nh; j++)
   if (Objekat[i, j] == 'O')
    Objekat[i, j] = ' ';
}

public bool Kraj()
{
 for (int i = 0; i < nw; i++)
 for (int j = 0; j < nh; j++)
  if (Ciljevi[i, j] ^ Objekat[i, j] == 'O')
   return false;

 return true;
}
```

```
public Nivo Kopija()
{
 Nivo P = new Nivo(nw, nh);
 for (int i = 0; i < nw; i++)

  for (int j = 0; j < nh; j++)
  {
   P.Objekat[i, j] = Objekat[i, j];
   P.Ciljevi[i, j] = Ciljevi[i, j];
  }
 return P;
}
```

SkupNivoa: Понаша се као низ **Nivo**-а. Садржи методе **Zapamti** (додаје нови **Nivo** у низ), **Prosiri** (проширује меморијски простор за памћење нових **Nivo**-а) и **VecOdradjeno**. Последња наведена метода служи да провери да ли је неки ниво већ садржан у низу.

```
public bool VecOdradjeno(Nivo N)
{
 for (int i = 0; i < n; i++)
  if (Racunanja.Isto(S[i], N))
   return true;

 return false;
}
```

Зато што је основна јединица кретања у овом алгоритму гурање кутије, да би два нивоа била иста није неопходно да се играч налази на истим позицијама. Два нивоа су иста ако су све кутије на истим позицијама (зидове и циљеве не узимамо у обзир зато што се не могу померати, а цео алгоритам се извршава на истом ниовоу заданом у апликацији) и ако играч из своје позиције у једном нивоу може доћи до своје позиције у другом. Ово се проверава у методи **Isto** која се припада класи са називом **Racunanja**.

Racunanja: У овој класи се налазе методе које израчунавају многе ствари потребне другим класама.

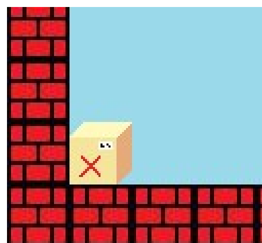
Pomocna: Ова класа служи да би нашла поља у датом нивоу до којих играч може доћи. Да би се ово урадило потребно је прво конструисати је, а потом позвати методу **Prohod**. Она као параметар узима један **bool** у зависности од кога ће уништити кутије у нивоу који јој је предат у конструктору. На овај начин можемо доћи до поља у нивоу на које играч може да стане у датој ситуацији, као и поља до којих играч може доћи независно од тога где се налазе кутије. Ово се остварује рекурзивном методом **ProhodRek**.

```
bool[,] B, D;
Nivo N;

private void ProhodRek(int x, int y)
{
 if (x >= 0 && y >= 0 && x < N.Width && y < N.Height && !D[x, y])
 {
  D[x, y] = true;
  if (N[x, y] == ' ')
  {
   B[x, y] = true;
   ProhodRek(x, y - 1); ProhodRek(x, y + 1);
   ProhodRek(x - 1, y); ProhodRek(x + 1, y);
  }
 }
}

public bool[,] Prohod(bool q)
{
 if (q == true) N.UnistiKutije();
 Point P = N.PozicijaIgraca();
 N[P.X, P.Y] = ' ';
 ProhodRek(P.X, P.Y);
 return B;
}
```

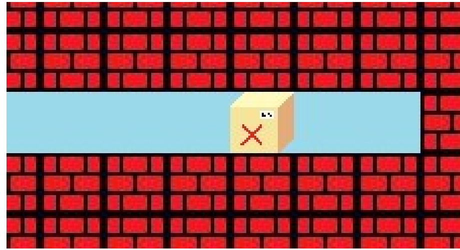
Матрица **D** памти поља на којима смо већ били, док **B** говори до којих поља играч може доћи.

Алгоритам

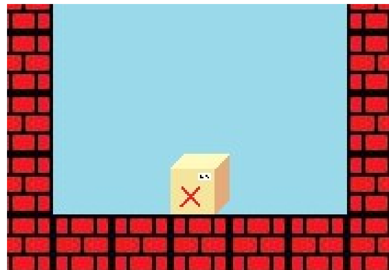
Након што преузме податке о нивоу из апликације, **Resavac** обележава сва поља са одређеним особинама. Постоје две врсте поља које се обележавају: тунели (који могу бити хоризонтални и вертикални) и мртва поља (енгл. *dead fields*). Хоризонтални тунели су поља изнад и испод којих се налазе зидови. Вертикални тунели су поља од којих се лево и десно налазе зидови. Поља са циљевима нису део тунела. Мртва поља су оне позиције у нивоу на којима не смемо допустити да се нађе кутија. Разлог овоме је то што се кутија из ових позиција не може довести до циља, те је онда немогуће решити ниво. На следећим сликама су приказани неки примери оваквих поља.



Кутија се не сме наћи на ћошковима зато што ју је потом немогуће померити у било ком правцу. Изузетак су они ћошкови на којима се налази циљ.

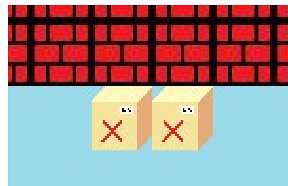


Тунели који се завршавају ћорсокацима такође спадају у мртва поља.

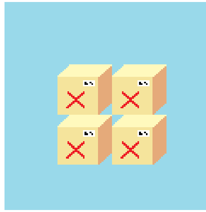


Кутија на слици изнад се може померати само између два зида и не може се довести до циља. Стога се она налази на мртвом пољу.

Програм ће такође све зидове и све позиције недоступне играчу обележити овим типом поља. Када апликација позове методу **NadjiResenje** класа ће прво проверити да ли се нека кутија налази на мртвом пољу. Ако је тако класа ће вратити поруку да је ниво немогуће решити. У противном ће позвати метод **Resi**. Овај метод је рекурзиван и базира се на једноставном *backtrack* алгоритму. Прво ће проверити да ли је тренутна ситуација већ била решавана (овде позивамо метод **VecOdradjeno** класе **SkupNivoa**). Ако није прво ће га запамтити, а потом пронаћи све потезе које је могуће повући у том нивоу. Потез не сме да доведе кутију на мртво поље, нити сме да учини да буде замрзнута (енгл. *frozen*). Кутија је замрзнута ако је позицијама других кутија блокирана те ју је немогуће гурнути у било ком правцу, а притом се не налази на циљу. На сликама испод су неки од случајева у којима је кутија замрзнута.



Кутије се не могу гурнути горе ни доле због зидова, а такође није могуће гурнути их лево ни десно због оне друге кутије.



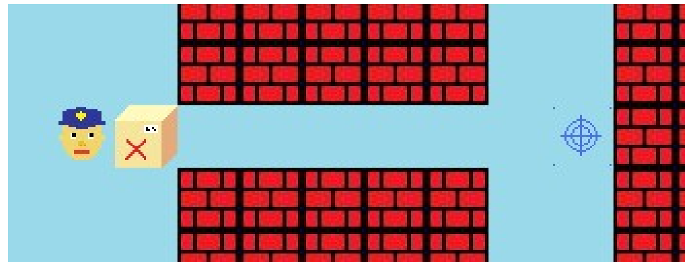
Ниједна кутија се не може померити због оних других кутија.

Након што нађе све потезе, метод ће их сортирати тако да они који доводе кутије ближе циљу имају предност. После ће метода за сваки од могућих потеза прво да га повуче, а потом да се рекурзивно позове и покуша да нађе решење. Ако нађе решење, метода ће вратити низ потеза који су довели до њега, а потом ће све рекурзивно позване методе да додају потез који су повукле у овај низ. Међутим, ако неки потез не може довести до решења, метода ће покушати са следећим потезом. Ако се деси да ниједан од могућих потеза не може довести до решења, метода ће то јавити оној методи која ју је рекурзивно позвала. Исто се дешава ако је немогуће повући било који потез из дате позиције. Сам код изгледа овако:

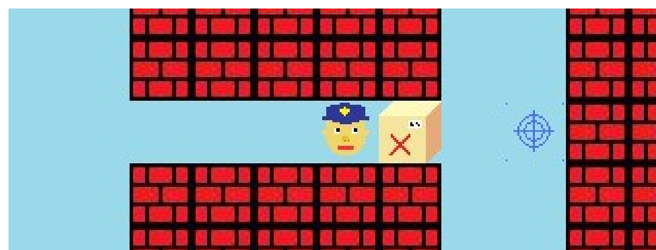
```
public SkupPoteza Resi(Nivo X)
{
 if (S.VecOdradjeno(X)) return new SkupPoteza();
 else
 {
  S.Zapamti(X.Kopija());
  if (X.Kraj())
  {
   Potez Po = new Potez();
   SkupPoteza S0 = new SkupPoteza();
   S0.Dodaj(Po);
   return S0;
  }
  else
  {
   SkupPoteza SP = Racunanja.OdrediPoteze(X, A.DLMat(), A.THMat(),
                   A.TVMat(), U);
   if (SP.N == 0) return new SkupPoteza();
   else
   {
    int i = 0;
    SkupPoteza S1 = new SkupPoteza();
    while (i < SP.N && S1.N == 0)
    {
     SkupPoteza S2 = Resi(Racunanja.PovuciPotez(X, SP[i], A));
     if (S2.N > 0) S1 = S2;
     i++;
    }
    if (S1.N == 0) return new SkupPoteza();
    else
    {
     S1.Dodaj(SP[i - 1]);
     return S1;
}}}}}
```

Ако метода није успела да нађе решење вратиће празан **SkupPoteza**, док ће у противном вратити **SkupPoteza** који се завршава посебним типом потеза. Овај потез има вредности **0** за обе координате позиције кутије која се помера и вредност **1** за правац. Ово би значило да се у овом потезу кутија помера ван граница нивоа (што је нелогично), док овакав потез уствари означава крај успешно пронађеног решења нивоа. Ако алгоритам успе да пронађе решење задатог нивоа, преостаје му да претвори низ потеза тог решења у формат који апликација може да користи.

Потези које класа садржи имају три вредности: **x**, **y** и **p** који означавају, редом, локацију кутије која се помера у нивоу и правац у коме се она помера. Међутим, апликација памти потезе као низ бројева који означава правце кретање играча. Класа ће све потезе решења редом претварати у низ оваквих бројева тако што ће наћи путању којом се играч креће у тим потезима, и спојити све путање у један низ. Да би се ово остварило користи се метода **PotezUPut**. Она проналази најкраћи пут од тренутне позиције играча до позиције поред кутије на коју играч треба да стане (ако кутију треба гурнути нагоре играч ће прво доћи до поља испод кутије, ако је треба гурнути лево играч ће прво доћи до поља десно од кутије, итд.), а потом додаје правце у којима се играч помера док гура кутију. Треба напоменути да ако би играч гурнуо кутију у тунел, алгоритам ће аутоматски реализовати гурање кутије на крај тог тунела. Ово се ради да би се алгоритам убрзао. Наиме, кутија се уопште не налази на битном пољу ако се налази на средини неког тунела.

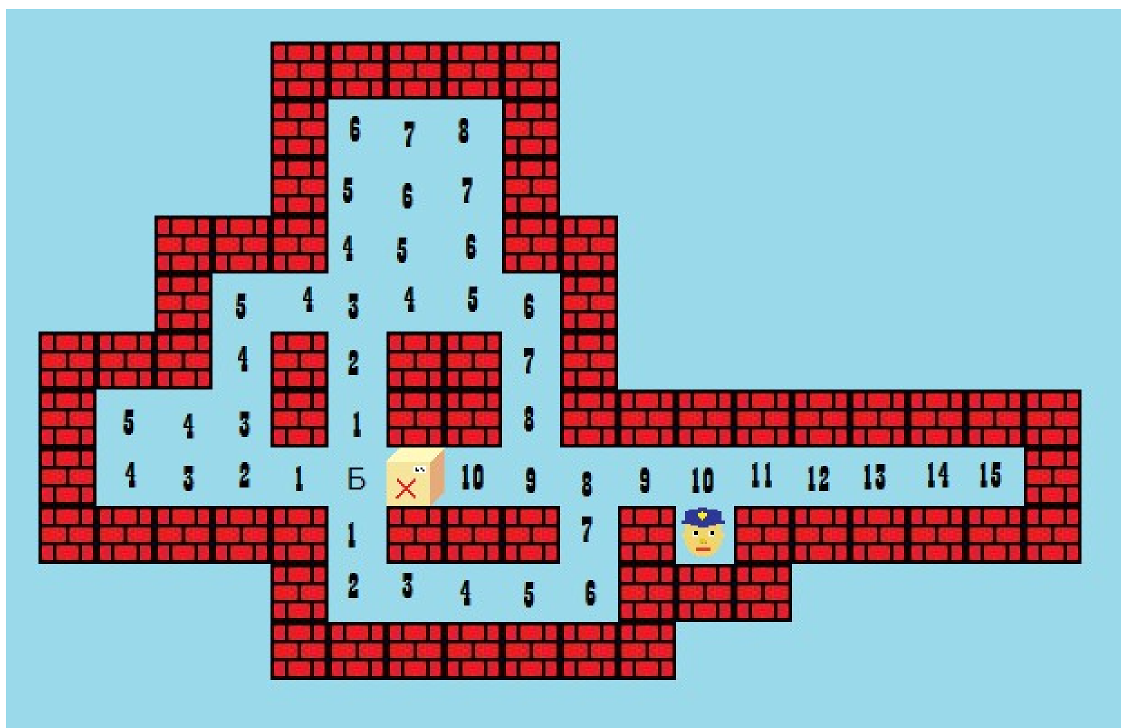


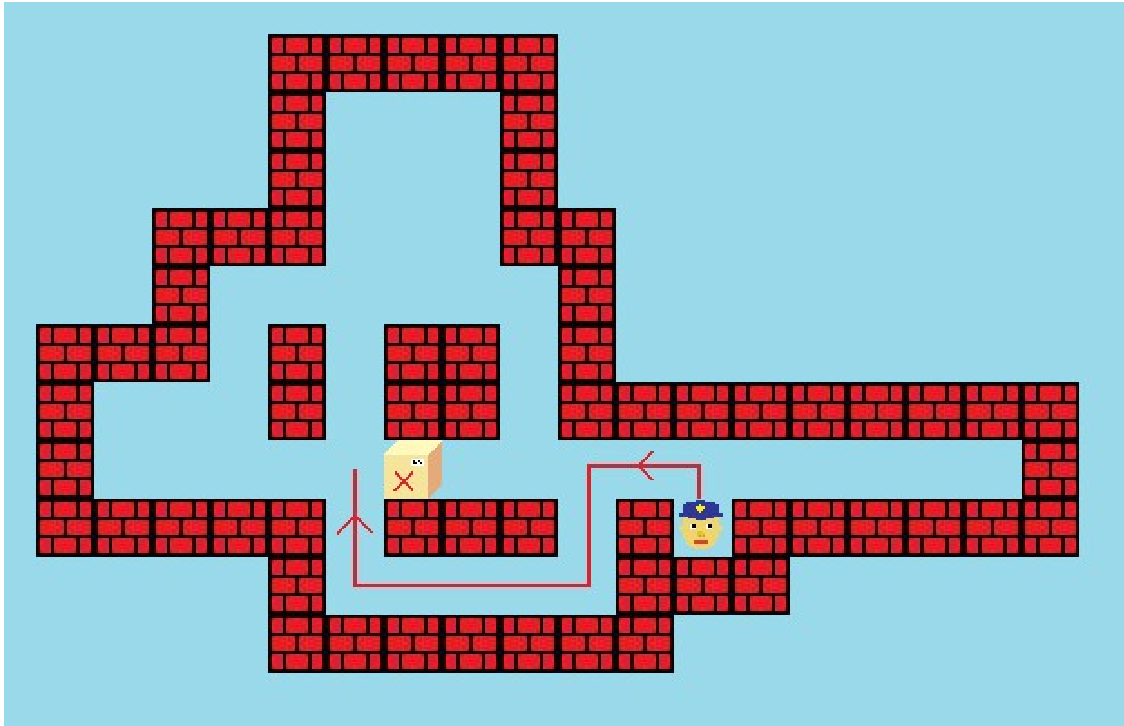
Ако би играч померио кутију за само једно поље, рекурзивна метода би потом тражила све потезе које је могуће повући у том тренутку. Како метода која проверава да ли су две ситуације у нивоу исте гледа локације кутија, више пута бисмо долазили до ситуација које се разликују само за локацију ове кутије у тунелу, а које су суштински исте. Због овога би се алгоритам много пута позивао за сличне ситуације и рачунао би их посебно чиме би се време његовог извршавања значајно продужило. Стога, ако алгоритам помери кутију на слици горе, ниво ће изгледати овако:



Да би пронашли најкраћи пут од неке позиције **А** до било које позиције **Б** у нивоу користимо методу **NajkraciPut**. Ова метода ће прво позвати методу **NadjuUdalj** која проналази дужину најкраћег пута од тачке **Б** до било ког слободног поља у нивоу (поља на коме се не налази ни кутија ни зид). Она то остварује тако што прво проналази поља на која можемо да дођемо са поља **Б** у једном потезу, даје тим пољима вредност **1** и ставља их у један низ. Потом се за свако поље у том низу проверава на која све поља може да се пређе са тих поља, а на која већ нисмо стали, и у та поља се уписује вредност **2**. Овај поступак се понавља, с тим што се у сваком следећем кораку увећава број који уписујемо у одговарајућа поља за један, тако да означавају дужину одговарајућих путева. Поступак завршавамо када обележимо сва поља у нивоу до којих је могуће доћи, након чега враћамо матрицу са овим вредностима.

Потом, метода **NajkraciPut** иницилизира променљиву **min** и низ бројева **P** (који памти правце у којима ћемо се кретати док будемо ишли од поља **А** до поља **Б**). Онда налази поље које се налази поред поља **А** да има најмању вредност броја записаног на себи (мисли се на број који смо генерисали у методи **NadjuUdalj**) и његову вредност уписује у променљиву **min**, а правац у коме ће се играч кретати се додаје у низ **P** и то поље се памти. Ово је поље које је део најкраћег пута. Потом се одређује следеће поље најкраћег пута као поље које се налази поред запамћеног поља обележено са најмањим бројем. То је још једно поље које се налази у путањи коју тражимо због чега додајемо још један правац у низ **P**. Овај поступак се понавља све док не дођемо до поља **Б**, након чега враћамо променљиву типа **Resenje** која садржи кретања играча неопходна да се дође од **А** до **Б**.
На следеће две слике је илустрован овај процес.





(Кутију желимо да померимо удесно.)

Након што ово уради за сваки потез, класа **Resavac** ће спојити све добијене низове бројева користећи методу **Spoji** класе **Resenje**, и новодобијени низ вратити апликацији. Међутим, ако је ниво немогуће решити враћени низ ће бити празан.

Закључак

Пројекат који сам направио се састоји из три велика дела: апликације **Sokoban**, апликације **Sokoban – pravljenje nivoa** и фајла **KlasaZaResavanje.dll**. Поред овога садржи и фолдере **slike** и **nivoi** којима се налазе подаци неопходни за успешно коришћење свих апликација у пројекту.

Поред играња разних нивоа, кориснику се пружа и могућност да осмисли и направи нове нивое које ће потом да испробава или да их зада рачунару у случају да не уме самостално да их реши.

Узевши у обзир величину пројекта (који има пар хиљада линија кода), могуће је да су почињени мали пропусти и ситне грешке које ће, уколико буду пронађене, бити благовремено исправљене, а логика решења унапређена.

Садржај

Логичка игра Сокобан - 1 -
Правила игре - 1 -
Објекти у игри - 1 -

Структура пројекта - 2 -

Апликација за играње - 2 -

Апликација за прављење нивоа - 7 -

Алгоритам решавања нивоа - 11 -
Структура *namespace*-а - 11 -
Алгоритам - 14 -

Закључак - 20 -

Литература - 22 -

Литература

[1] Pavel Klavík: **Sokoban Solver - a short documentation**, http://pavel.klavik.cz

[2] Stanley B. Lippman: **C# Primer: A Practical Approach**, Addison-Wesley, 2002.