Шеста београдска гимназија

Милана Ракића 33

Београд

**Матурски рад из програмирања**

**ПРОГРАМСКА РЕШЕЊА ЗА ИЗЛАЗ ИЗ ЛАВИРИНТА**

Професор ментор: Ученик:

Мирјана Ћетојевић Милош Младеновић IV-10

Београд, мај 2023. године

САДРЖАЈ

**Страна**

[1. УВОД 3](#_Toc134279623)

[1.1. Развој решења за претрагу пута кроз лавиринт кроз историју 3](#_Toc134279624)

[2. РАЗЛИЧИТИ НАЧИНИ ПРОГРАМСКЕ РЕПРЕЗЕНТАЦИЈА ЛАВИРИТА 4](#_Toc134279625)

[2.1. Програмска репрезентација лавиринта примењена у симулатору 4](#_Toc134279626)

[2.1.1. Програмска репрезентација поља лавиринта 4](#_Toc134279627)

[2.1.2. Репрезентација лавиринта помоћу одговарајуће класе 7](#_Toc134279628)

[3. РАЗЛИЧИТИ ПРИСТУПИ У ПРОНАЛАЖЕЊУ ПУТАЊЕ КРОЗ ЛАВИРИНТ 11](#_Toc134279629)

[3.1. Алгоритам претраге у ширину (BFS) 11](#_Toc134279630)

[3.2. Алгоритам претраге у дубину (DFS) 13](#_Toc134279631)

[3.3. Дајкстрин алгоритам 15](#_Toc134279632)

[3.4. Похлепни алгоритам претраге по најбољем својству (Best-first Search) 18](#_Toc134279633)

[3.4.1. Манхетн удаљеност 19](#_Toc134279634)

[3.4.2. Дијагонална удаљеност 20](#_Toc134279635)

[3.4.3. Еуклидска удаљеност 20](#_Toc134279636)

[3.5. А\* алгоритам 23](#_Toc134279637)

[4. Закључак 27](#_Toc134279638)

[5. Прилог 28](#_Toc134279639)

[Литература 29](#_Toc134279640)

# УВОД

Лавиринти су конструкције које су кроз историју биле инспирација за многа уметничка дела и предмет проучавања у различитим научним дисциплинама. Лавиринт се појављује у разним културама и цивилизацијама кроз историју, као симбол загонетке, опасности и изазова. Један од најпознатијих примера је грчка митологија, према којој је Минотаур, полу бик полу човек, био затворен у лавиринту на Криту, а Тезеј је морао да пронађе пут кроз лавиринт и убије Минотаура како би ослободио Атињане данка. У античкој Грчкој лавиринти су били популарни као облик забаве и рекреације, а у средњем веку су често били укључени у утврђења и замкове као облик одбране. Данас се лавиринти често користе у архитектури и пејзажном дизајну као декоративни елементи или као средство за медитацију и релаксацију.

## Развој решења за претрагу пута кроз лавиринт кроз историју

Од старог века, па до данас, проналажење пута кроз лавиринт представљало је изазов вредан проучавања. Различите културе и цивилизације кроз историју су стварале различита решења за пролазак кроз лавиринте, од класичног физичког сналажења до разних симболичких и духовних приступа. Решење које је по митологији користио Тезеј је подразумевало провлачење конопца кроз лавиринт помоћу којег је знао којим је путем пролазио. Тај приступ се може упоредити са алгоритмом претраге у дубину, који је данас у широкој употреби. Још један приступ, који је посебно проучаван током 19. века је алгоритам праћења зида, према коме ако особа од улаза прати један (леви или десни) зид мора да дође до излаза. Џон Плеџ је проучавао овај алгоритам и унапређивао га за употребу у нетипичним лавиринтима.

Посебно интересовање за проблем проласка кроз лавиринт настало је у савремено доба развојем технологије. Развојем технологије, појавила су се различита програмска решења за овај проблем, далеко ефикаснија и поузданија од претходно поменутих. Решења проласка кроз лавиринт се данас примењују у различитим областима, од видео игара до роботских, навигационих и телекомуникационих система. Примена ових решења је поготово широка када узмемо у обзир да је лавиринт заправо врста графа, а алгоритам за пролазак кроз лавиринт алгоритам за претрагу графа.

Нека од данас најчешће коришћених алгоритама за решевање овог проблема ће бити објешњена у наставку овог рада. У циљу описа тих алгоритама, за овај рад је креирана Windows десктоп апликација помоћу које се сваки од алгоритама може симулирати на произвољном лавиринту.

# РАЗЛИЧИТИ НАЧИНИ ПРОГРАМСКЕ РЕПРЕЗЕНТАЦИЈА ЛАВИРИТА

Прва, и најинтуитивнија могућност програмске репрезентације лавиринта је репрезентација матрицом. Овај приступ подразумева матрицу у којој свако поље може бити у једном од два стања – проходно или непроходно. Овај приступ је одговарајући у већини ситуација, јер се алгоритми претраге на њему лако могу имплементирати. Међутим, мана овог приступа је алоцирање понекад сувишне меморије када говоримо о лавиринтима великих димензија. На споменутом симулатору креираном за потребе овог рада је коришћен овај приступ јер је симулатор намењен илустрацији алгоритма на лавиринту који уцртава корисник максималних димензија 50x50.

Као што је већ споменуто лавиринт се може посматрати као репрезентација неусмеронг графа код кога сваки је сваки чвор репрезентација једне позиције у лавиринту. Тај чвор може да има до 4 или 8 повезаних чворова, тј. суседних позиција, у зависности од тога да ли је дозвољено дијагонално кретање (са изузетком чворова ивичних позиција – они имају до 3 или 2 повезана чвора у ћошковима, а на страницама до 5 или 3). Овакав граф се може реализовати помоћу објеката класе чворова повезаних показивачима, као и матрицом суседности у којој члан (*i,j*) означава постојање или непостојање везе између чворова *i* и *j.* Овако репрезентовани графови су такође погодни за имплементацију алгоритама претраге.

## Програмска репрезентација лавиринта примењена у симулатору

У симулатору креираном за потребе овог рада, као што је наведено, за репрезентацију лавиринта се користи матрица у којој свако поље може бити проходно и непроходно. Поље такве матрице би у основном случају могло да се представи као једна логичка променљива са вредностима тачно или нетачно. Међутим како је у конкретном случају потребно пољима придружити више својстава, креирана је сложенија класа чија су поља објекти.

Такође, иако у основном случају лавиринт може бити стандардана дводимензионална матрица, ради лакшег управљања лавиринтом, извршавања претраге и омогућавања интеракције корисника са лавиринтом, креирана је класа Lavirint са свим додатно потребним атрибутима и методама.

### Програмска репрезентација поља лавиринта

public class Polje

{

public StanjePolja stanje;

public Point Pozicija;

public SolidBrush sb;

public int stranica;

public static int xstart, ystart;

public double G, H;

public Polje parent;

public Polje(int str, Point pozicija, int xstart, int ystart, StanjePolja stanje = StanjePolja.zid)

{

this.Pozicija = pozicija;

this.stanje = stanje;

this.stranica = str;

Polje.xstart = xstart;

Polje.ystart = ystart;

sb = new SolidBrush((stanje == StanjePolja.zid) ? Boje.boja\_Zid : Boje.boja\_Slobodno);

}

public void Crtaj(Graphics g)

{

g.FillRectangle(sb, xstart+Pozicija.X \* (stranica + 2), ystart+Pozicija.Y \* (stranica + 2), stranica, stranica);

if (stanje == StanjePolja.start)

using (Image startImage = Properties.Resources.start)

{

g.DrawImage(startImage, xstart + Pozicija.X \* (stranica + 2), ystart + Pozicija.Y \* (stranica + 2), stranica, stranica);

G = 0;

}

else if (stanje == StanjePolja.cilj)

using (Image finishImage = Properties.Resources.finish)

{

g.DrawImage(finishImage, xstart + Pozicija.X \* (stranica + 2), ystart + Pozicija.Y \* (stranica + 2), stranica, stranica);

}

}

public void Click(Graphics g, int StartOrCilj, bool sveZauzeto)

{

if (StartOrCilj == 0)

{

stanje = stanje == StanjePolja.slobodno ? StanjePolja.zid : StanjePolja.slobodno;

sb = new SolidBrush((stanje == StanjePolja.slobodno) ? Boje.boja\_Slobodno : Boje.boja\_Zid);

}

else

{

sb = new SolidBrush(sveZauzeto ? Boje.boja\_Zid : Boje.boja\_Slobodno);

if (StartOrCilj == 1)

{

((MainForm)Application.OpenForms[0]).lavirint.Start = Pozicija;

stanje = StanjePolja.start;

}

else if (StartOrCilj == 2)

{

((MainForm)Application.OpenForms[0]).lavirint.Finish = Pozicija;

stanje = StanjePolja.cilj;

}

}

Crtaj(g);

}

public void Clear(Graphics g, bool zid)

{

stanje = zid ? StanjePolja.zid : StanjePolja.slobodno;

sb = new SolidBrush(zid ? Boje.boja\_Zid : Boje.boja\_Slobodno);

Crtaj(g);

}

public void MarkOtvoren(Graphics g)

{

sb.Color = Boje.boja\_Otvoren;

Crtaj(g);

}

public void CrtajPutanju(Graphics g, int delta\_t)

{

sb.Color = Boje.boja\_Putanja;

this.Crtaj(g);

Thread.Sleep(delta\_t);

Console.Beep();

if (stanje != StanjePolja.start)

this.parent.CrtajPutanju(g, delta\_t);

}

public void Resize(int stranica, int xstart, int ystart)

{

this.stranica = stranica;

Polje.xstart = xstart;

Polje.ystart = ystart;

}

public void Restart(bool zauzeto, Graphics g)

{

sb.Color = zauzeto ? Boje.boja\_Zid : Boje.boja\_Slobodno;

stanje = zauzeto ? StanjePolja.zid : StanjePolja.slobodno;

Crtaj(g);

}

}

* 1. Атрибути који су намењени за тачно и јасно исцртавање поља у корисничком интерфејсу су:
     1. pozicija – позиција конкретног поља у лавиринту;
     2. xstart, ystart – координате горњег левог ћошка лавиринта, у односу на који се исцртавају сва поља. Ови атрибути су статички јер су у овој имплементацији заједнички свим објектима класе поља;
     3. stranica - дужина, тј. ширина једног поља (ова вредност се израчунава на основу броја поља и димензија корисничког прозора).
  2. Атрибути који ће се користити у алгоритмима претраге су:
     1. stanje – атрибут набројивог типа, који одређује у ком стању је поље тренутно. Набројиви тип овог атрибута је дефинисан на следећи начин:

public enum StanjePolja { zid, slobodno, start, cilj }

* + 1. G, H – атрибути реалног типа који се користе у Дајкстрином алгоритму, A\* алгоритму и претрази по најбољем својству и чине срж тих алгоритама. О њима ће бити више речи у наставку.
    2. parent – показивач на родитељско, тј. претходно поље у путањи према траженом пољу у лавиринту. Та путања је преко пказивача parent састављена као листа повезаних чворова, у којој је последњи чвор тражено поље. Реконструкција пронађене путање ће се вршити помоћу ових повезаности.
  1. Методе класе поља:
     1. Crtaj – исцртава поље у корисничком интерфејсу на одговарајући начин, у зависности од стања, позиције и димензије поља
     2. Click – врши промену стања поља, а позива се када корисник мишем кликне на поље у циљу уцртавања произвољног лавиринта. На крају ова метода позива претходно наведену методу за цртање, како би измене на пољу биле приказане.
     3. Clear, Restart и Resize су још неке од метода које омогућавају интуитивну интеракцију косрисника са пољима (брисање садржаја, поновно постављање стања поља и промена величине поља)
     4. MarkOtvoren и CrtajPutanju су методе које се позивају кроз рад алгоритама претраге и служе за графички приказ поља проверених у току претраге, и за реконструкцију пронађене путање помоћу показивача са поља на претходно поље (parent) у путањи. Метода CrtajPutanju ту реконструкцију врши на рекурзиван начин.

### Репрезентација лавиринта помоћу одговарајуће класе

Следи код класе лавирината, у којој су изузети садржаји метода коришћених у алгоритмима претраге, јер ће те методе и алгоритми бити објешњене у наставку:

public class Lavirint

{

public Polje[,] polja;

public int stranicaPolja;

public Tuple<int,int> Dimenzije; //U poljima

public int xstart = 0, ystart = 0;

public Point Start;

public Point Finish;

public bool sveZazeto\_onStart = true;

int simulator\_delta\_t = 50;

public Lavirint(int FormWidth, int FormHeight, Tuple<int,int> dimenzije)

{

Dimenzije = dimenzije;

stranicaPolja = FormWidth / dimenzije.Item1 - 2;

if(stranicaPolja\*dimenzije.Item2 > (FormHeight - dimenzije.Item2 \* 2))

stranicaPolja = FormHeight / dimenzije.Item2 - 2;

if(dimenzije.Item1 \* (stranicaPolja+2) < FormWidth)

xstart = (FormWidth - dimenzije.Item1 \* (stranicaPolja+2)) / 2;

if (dimenzije.Item2 \* (stranicaPolja + 2) < FormHeight)

ystart = (FormHeight - dimenzije.Item2 \* (stranicaPolja+2)) / 2;

polja = new Polje[Dimenzije.Item2, Dimenzije.Item1];

Start = new Point(-1,-1);

Finish= new Point(-1,-1);

for (int i = 0; i < Dimenzije.Item2; i++)

{

for (int j = 0; j < Dimenzije.Item1; j++)

{

polja[i, j] = new Polje(stranicaPolja,new Point(j,i), xstart, ystart);

}

}

}

public void Resize(int FormWidth, int FormHeight)

{

stranicaPolja = FormWidth / Dimenzije.Item1 - 2;

if (stranicaPolja \* Dimenzije.Item2 > (FormHeight - Dimenzije.Item2 \* 2))

stranicaPolja = FormHeight / Dimenzije.Item2 - 2;

if (Dimenzije.Item1 \* (stranicaPolja + 2) < FormWidth)

xstart = (FormWidth - Dimenzije.Item1 \* (stranicaPolja + 2)) / 2;

if (Dimenzije.Item2 \* (stranicaPolja + 2) < FormHeight)

ystart = (FormHeight - Dimenzije.Item2 \* (stranicaPolja + 2)) / 2;

for (int i = 0; i < Dimenzije.Item2; i++)

for (int j = 0; j < Dimenzije.Item1; j++)

polja[i, j].Resize(stranicaPolja,xstart,ystart);

}

public void Crtaj(Graphics g)

{

for (int i = 0; i < Dimenzije.Item2; i++)

{

for (int j = 0; j < Dimenzije.Item1; j++)

{

polja[i, j].Crtaj(g);

}

}

}

public Point Polje\_by\_XY(int x, int y)

{

int i = (int)((x - xstart) / (stranicaPolja+2));

int j = (int)((y - ystart) / (stranicaPolja+2));

return new Point(i > (Dimenzije.Item1 - 1) ? (Dimenzije.Item1 - 1) : i, j > (Dimenzije.Item2 - 1) ? (Dimenzije.Item2 - 1) : j);

}

public void SetStartFinish(Graphics g, int x, int y, bool start)

{

if (start)

{

if(Start.X != -1)

polja[Start.Y, Start.X].Clear(g,sveZazeto\_onStart);

Start = new Point(x, y);

polja[y, x].Click(g, 1, false);

}

else

{

if(Finish.X != -1)

polja[Finish.Y, Finish.X].Clear(g, sveZazeto\_onStart);

Finish = new Point(x, y);

polja[y, x].Click(g, 2, false);

}

}

IEnumerable<Polje> SusednaPolja(Polje p, bool strane4)

{...}

public void AStar(bool strane4, Graphics g)

{...}

public Polje MinimumF(HashSet<Polje> nodes)

{...}

public Polje MinimumH(HashSet<Polje> nodes)

{...}

public Polje MinimumG(HashSet<Polje> nodes)

{...}

public void BestFS(bool strane4, Graphics g)

{...}

public void Dijkstra(bool strane4, Graphics g)

{...}

public void BFS(bool strane4, Graphics g)

{...}

public void DFS(bool strane4, Graphics g)

{...}

public void Clear(bool svezauzeto, Graphics g)

{

Start = new Point(-1,-1);

Finish = new Point(-1,-1);

sveZazeto\_onStart = svezauzeto;

for (int i = 0; i < Dimenzije.Item2; i++)

for (int j = 0; j < Dimenzije.Item1; j++)

polja[i, j].Restart(svezauzeto,g);

}

public void RemovePaths(Graphics g) {

for (int i = 0; i < Dimenzije.Item2; i++)

for (int j = 0; j < Dimenzije.Item1; j++)

{

polja[i,j].sb.Color = (polja[i,j].stanje == StanjePolja.zid) ? Boje.boja\_Zid : Boje.boja\_Slobodno;

polja[i, j].Crtaj(g);

}

}

}

}

* 1. Атрибути који чине класу Lavirint су:
     1. polja – Матрица објеката класе Polje, која је основа у овој репрезентацији лавиринта;
     2. stranicaPolja, xstart, ystart – Дужина странице поља и координате горње леве тачке лавиринта, то су атрибути намењени за прецизно цртање лавиринта на корисничком интерфејсу;
     3. Start, Finish – показивачи на стартно поље у лавиринту и поље до кога треба пронаћи путању;
     4. sveZazeto\_onStart – логичка променљива која показује да ли је лавирнт био потпуно проходан, или потпуно непроходан пре него је корисник почео да уцртава лавиринт. Користи се у методама за приказ лавиринта,
     5. simulator\_delta\_t – целобројна вредност, која представља време у милисекундама које треба да протекне између две графичке измене у лавиринту приликом претраге, тј. успорава претрагу тако да корисник може да испрати догађеје у њој. Израчунава се на основу удаљености старта од циља;
  2. Методе и конструктори класе лавирината:
     1. Lavirint – конструктор класе којим се креира нови објекат на основу димензија лавирнта у пољима и димензија корисничког прозора у пикселима;
     2. Resize – метода која се позива приликом промене величине корисничког прозора и омогућава респонсивност апликације;
     3. Crtaj – метода која исцртава лавиринт на графички интерфејс на основу његових особина;
     4. Polje\_by\_XY – метода која враћа позцију поља у лавиринту на основу координата на екрану. Користи се при препознавању поља на које је кориник кликнуо мишем;
     5. Clear – метода која брише измене на лавиринту и омогућава кориснику да крене са уцртавањем лавиринта из почетка;
     6. RemovePaths – метода која уклања графички приказ пронађене путање и проверених поља у лавиринту, те приказује само проходност поља.

# РАЗЛИЧИТИ ПРИСТУПИ У ПРОНАЛАЖЕЊУ ПУТАЊЕ КРОЗ ЛАВИРИНТ

Од појаве рачунарства развила су се многа различита решења за претрагу пута кроз лавиринт. Модерна решења овог проблема су заправо имплементације алгоритама за претрагу графова јер су лавиринти, као што је наведено, заправо неусмерени графови. Алгоритами који се користе у ову сврху, а данас су у широкој употреби се могу поделити на информисане и неинформисане алгоритме. Неинформисани, или слепи, алгоритми претраге су они који у датом моменту немају слику о њиховом домену, па могу само да препознају да ли је одређено поље тражено или не, али не могу да доносе одлуке о томе које поље је боље обићи следеће у току претраге. За разлику од њих информисани алгоритми претраге имају слику о домену на коме раде и о траженом пољу. Зато је у информисаним алгоритмима могуће применити хеуристичке функције помоћу којих се доноси одлука о томе које од суседних поља треба обићи следеће у датом моменту. У наставку ће бити описани алгоритми оба типа, који су примењени у креираном симулатору.

## Алгоритам претраге у ширину (BFS)

Алгоритам претраге у ширину (енг. Breadth-first Search) је један од неинформисаних алгоритама претраге. Овај алгоритам започиње претрагу од стартног поља и у следећем кораку обилази сва суседна поља редом (тј. све чворове који су директно повезани са стартним чвором графа). Затим обилази сва поља следећег слоја (сва поља која од стартног поља дели слој претходно проверених поља). Тај поступак се понавља све док се не дође до траженог поља или до момента када више нема могућности за даљи обилазак. У овом алгоритму се уобичајено користи ред (енг. queue) за чување поља, тј. чворова, који су „наследници“ чворова претходног слоја и који се редом обилазе по принципу „први ушао први изашао“ (енг. First In First Out - FIFO). Приликом обиласка поља једног слоја у исти ред се додају (Enqueue) поља следећег слојa, тј. наследничка поља, а проверена поља претходног слоја се избацују (Dequeue).

Следи метода класе Lavirint која је раније изостављена у класи Lavirint, а која врши претрагу у ширину, кораци алгоритма су објашњени кроз код у коментарима:

/// <summary>

/// Algoritam pretrage u širinu

/// </summary>

/// <param name="strane4"> Logička promenljiva, pokazuje da li je dozvoljeno dijagonalno kretanje ili ne</param>

/// <param name="g"> Grafika korisničkog interfejsak po kojoj se iscrtava prolaz kroz lavirint </param>

public void BFS(bool strane4, Graphics g)

{

//Izračunava se vreme potrebno da bi se algoritam usporio radi bolje preglednosti, prema proporciji

//25 : P = X : 40, P-povrsina pravougaonika odredjena start i finish tackama - obrnuta proporcija dobijena testriranjem

simulator\_delta\_t = (int)(25 \* 40 / ((Math.Abs(Start.X - Finish.X)+1) \* (Math.Abs(Start.Y - Finish.Y)+1)));

Queue<Polje> Polja\_za\_proveru = new Queue<Polje>();

HashSet<Polje> poseceno = new HashSet<Polje>();

Polje startno\_polje = polja[Start.Y, Start.X];

//Startno polje se dodaje polje u red za proveru

Polja\_za\_proveru.Enqueue(startno\_polje);

//Iteracija se ponavlja sve dok ima mogućnosti za proveru

while (Polja\_za\_proveru.Count > 0)

{

//Trenutno polje se skida sa početka reda, to je polje koje je najduže bilo u redu (FIFO)

Polje trenutno = Polja\_za\_proveru.Dequeue();

//Ako je trenutno polje traženo, pretraga je uspešno završena, a rekosnstruisana putanja se iscrtava

if (trenutno.Pozicija.X == Finish.X && trenutno.Pozicija.Y == Finish.Y)

{

trenutno.CrtajPutanju(g, 2\*simulator\_delta\_t); //Grafički prikaz rekonstruisane putanje na lavirintu

return;

}

//Trenutno polje se dodaje u skup posećenih polja, da se ne bi ponavljalo

poseceno.Add(trenutno);

// Prolazak kroz sva susedna polja trenutnog polja, susedna polja se određuju i na osnovu toga da li je dozvoljeno dijagnalno kretanje

foreach (Polje sused in SusednaPolja(trenutno, strane4))

{

//Ako je već posećeno ili dodato u red, preskače se

if (poseceno.Contains(sused) || Polja\_za\_proveru.Contains(sused))

continue;

//Trenutno polje se postavlja kao prethodničko susednom polju

sused.parent = trenutno;

//Susedno polje se dodaje na kraj reda kako bi bilo provereno tokom dalje iteracije

Polja\_za\_proveru.Enqueue(sused);

//Usporavanje algortma radi bolje preglednosti

Thread.Sleep(simulator\_delta\_t);

//Grafiički prikaz otvorenog polja

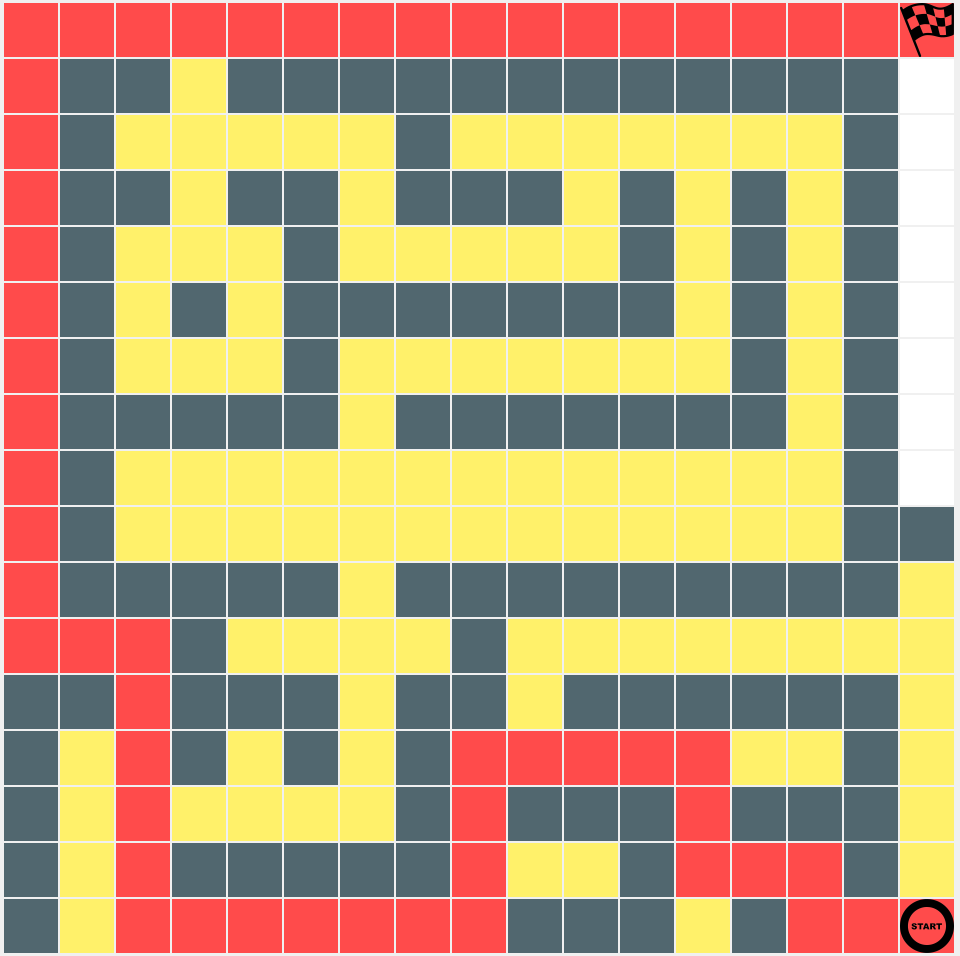
sused.MarkOtvoren(g);

}

}

}

На слици 1 дат је графички приказ резултата алгоритма претраге у ширину у симулатору. Сивом бојом означене су препреке, тј. непроходна поља, белом проходна поља, жутом бојом сва отворена поља у току претраге, а црвеном бојом означена је пронађена путања.



Слика - Графички приказ резултата претраге у ширину

## Алгоритам претраге у дубину (DFS)

Алгоритам претраге у дубину је још један од неинформисаних алгоритама претраге. Алгоритам претраге у дубину, као и већина других, претрагу започиње од стартног поља. Након провере стартног поља узима се једно, прво, суседно поље као почетно поље прве гране графа (обично се различите гране пролазе редом с лева на десно). Овим алгоритмом се пролази кроз сва поља једне гране од почетног поља (или поља где постоји гранање) до највеће могуће дубине, на којој се налази или мртви крај или тражено поље, а затим се враћа у назад до првог претходног гранања уколико тражено поље није пронађено (енг. backtracking) и тај поступак се понавља све док се не дође до траженог поља или до момента када су све могућности тј. гране проверене. Овај алгоритам за меморисање поља која треба обићи користи структуру стек, која функционише по принципу „последњи ушао први изашао“ (eng. LIFO). Редослед кoјим се поља налазе и додају у стек омогућава обилажење целе гране пре преласка на следећу. Алгоритам претраге у дубину се углавном користи ради провере да ли је пролаз кроз лавиринт (тј. кроз граф) уопште могућ, а не када је потребно пронаћи најкраћу путању, јер овај алгоритам не гарантује проналазак најкраћег пута, већ било ког.

У наставку следи метода за претрагу у дубину класе Lavirint, која је ранаије изостављена из кода те класе:

/// <summary>

/// Algoritam pretrage u dubinu - DFS

/// </summary>

/// <param name="strane4"> Logička promenljiva, pokazuje da li je dozvoljeno dijagonalno kretanje ili ne</param>

/// <param name="g"> Grafika korisničkog interfejsak po kojoj se iscrtava prolaz kroz lavirint </param>

public void DFS(bool strane4, Graphics g)

{

//Izračunava se vreme potrebno da bi se algoritam uspori radi bolje preglednosti, prema proporciji

simulator\_delta\_t = (int)(25 \* 40 / ((Math.Abs(Start.X - Finish.X)+1) \* (Math.Abs(Start.Y - Finish.Y)+1)));

Stack<Polje> Polja\_za\_proveru = new Stack<Polje>();

HashSet<Polje> poseceno = new HashSet<Polje>();

Polje startNode = polja[Start.Y, Start.X];

//Početno polje se dodaje u stek polja za proveru

Polja\_za\_proveru.Push(startNode);

//Iteracija se ponavlja sve dok ima mogućnosti za proveru

while (Polja\_za\_proveru.Count > 0)

{

//Za trenutno polje u ovom momentu se skida polje sa kraja steka

Polje trenutno = Polja\_za\_proveru.Pop();

//Ako je trenutno polje traženo, pretraga je uspešno završena, a rekosnstruisana putanja se iscrtava

if (trenutno.Pozicija.X == Finish.X && trenutno.Pozicija.Y == Finish.Y)

{

trenutno.CrtajPutanju(g, 2 \* simulator\_delta\_t);

return;

}

//trenutno polje se dodaje u skup posećenih polja, da se ne bi ponavljalo

poseceno.Add(trenutno);

//Grafički prikaz prolaza kroz polje

trenutno.MarkOtvoren(g);

//Usporavanje algoitma radi bolje preglednosti

Thread.Sleep(simulator\_delta\_t);

foreach (Polje sused in SusednaPolja(trenutno, strane4))

{

if (!poseceno.Contains(sused))

{

//Neposećeni sused se dodaje na kraj hijerarhije u dosadašnjoj putanji (pomoću pokazivača na prethodnika)

sused.parent = trenutno;

//Taj sused se dodaje na kraj steka, kako bi bio proveren

Polja\_za\_proveru.Push(sused);

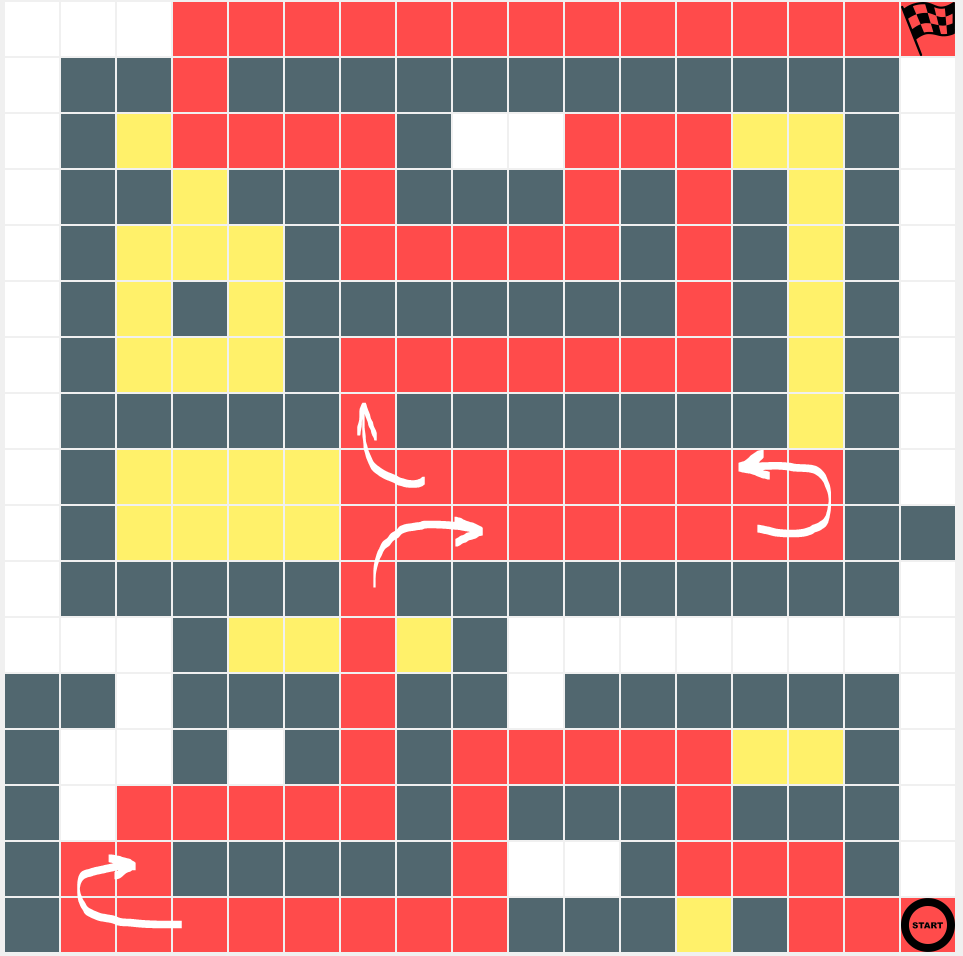
}

}

}

}

Графички приказ резултата ове претраге дат је на слици 2. Као што је наведено овај алгоритам не проналази најкраћи пут, те је црвеном бојом означена путања која је прва пронађена.



Слика - Графички приказ резултата претраге у дубину

## Дајкстрин алгоритам

Дајкстрин алгоритам је такође неинформисани алгоритам претраге. Овај алгоритам је осмислио холандски научник Едсгер Дајкстра 1956. године, а објавио га 1959. године у часопису Numerische Mathematik.

Дајкстрин алгоритам је алгоритам претраге који пролналази гарантовано најкраћу путању између два чвора тј. поља ако она постоји. Како би то пронашао проналазе се и најкраће удаљености стартног поља од свих других поља (све док се не пронађе пут до траженог поља). Овај алгоритам кроз претрагу сваком посећеном пољу додељује вредност која представља удаљеност тог поља од стартног поља (често означена као h вредност).

Дајкстрин алгоритам претрагу такође почиње од стартног поља, те на почетку стартном пољу додељује 0 као удаљаност од самог себе. У следећем кораку показивачи на суседна поља се памте у одређеном контејнеру података како би била посећена касније и као удаљеност од стартног поља се 1 додељује пољима која деле страницу са почетним пољем, а суседима по дијагонали. Тај контејнер се често реализује као приоритетни ред, у коме су поља са мањим g вредностима у врху. Међутим, у креираном симулатору због специфичног меморисања удаљености од почетка као аутибута у објекту, коришћење приоритетног реда није било адекватно, већ се отворена поља чувају као скуп.

У следећем кораку за даљи пролаз се бира поље са најмањом удаљеношћу од стартног поља из претходно поменутог контејнера. Ово поље се сада уклања из скупа отворених поља и додаје се у скуп затворених поља. Тада, али и увек када се посети одређено поље израчунава се удаљеност његових непосећених суседа од стартног поља као збир удаљености тог, тренутног, поља од стартног поља и удаљености тренутног поља од суеда. Уколико суседно поље већ има додељену удаљеност од почетног поља новоизрачуната вредност ће се доделити само ако је мања од старе, јер то значи да је пронађена краћа путања. Уколико тај сусед нема додељену вредност израчуната вредност ће се свакако доделити, јер то значи да је пронађена прва путања до тог поља. Када је пронађена прва путања до одређеног поља или путања која је краћа од раније пронађене, том пољу се као претходник поставља тренутно поље преко кога је g вредност израчуната. Ово израчунавање се врши за сва поља суседна тренутном која раније нису затворена и додају се у скуп отворених поља која треба обићи. Овакав пролаз кроз лавиринт се наставља и кораци се понављају све док се не дође до траженог поља или до тренутка када више не постоје могућности за даљи пролаз.

Следи метода Дајкстриног алгоритма претраге која припада класи Lavirint. Објашњења конкретних корака се налазе у коментарима:

/// <summary>

/// Dajkstrin algoritam pretrage najkraće putanje

/// </summary>

/// <param name="strane4"> Logička promenljiva, pokazuje da li je dozvoljeno dijagonalno kretanje ili ne</param>

/// <param name="g"> Grafika korisničkog interfejsak po kojoj se iscrtava prolaz kroz lavirint </param>

public void Dijkstra(bool strane4, Graphics g)

{

//Izračunava se vreme potrebno da bi se algoritam usporio radi bolje preglednosti, prema proporciji

//25 : P = X : 40, P-povrsina pravougaonika odredjena start i finish tackama - obrnuta proporcija dobijena testriranjem

simulator\_delta\_t = (int)(25 \* 40 / ((Math.Abs(Start.X - Finish.X)+1) \* (Math.Abs(Start.Y - Finish.Y)+1)));

HashSet<Polje> open = new HashSet<Polje>();

HashSet<Polje> closed = new HashSet<Polje>();

Polje StartnoPolje = polja[Start.Y, Start.X];

// Startno polje je za 0 odaljeno samo od sebe

StartnoPolje.G = 0;

// Startno polje se dodaje u open listu radi dalje provere

open.Add(StartnoPolje);

while (open.Count > 0)

{

// Za trenutno polje se uzima polje iz open liste koje je je najbliže startnom polju - minimalan G faktor

// Ovaj korak se može izbeći korišćenjem priority queue-a, što u ovom slučaju nije bilo potpuno adekvatno

Polje trenutno = MinimumG(open);

// Odabrano trenutno polje se uklanja iz skupa otvorenih i stavlja u skup zatvorenih polja, kako se ne bi ponovilo

open.Remove(trenutno);

closed.Add(trenutno);

// Prolazak kroz sva susedna polja, po dijagonali ili ne, što je određeno sa strane4

foreach (Polje sused in SusednaPolja(trenutno, strane4))

{

// Ako je polje već proverno, preskače se

if (closed.Contains(sused))

continue;

// Razdaljina od starnog polja do suseda (moguceG) je jednaka zbiru razdaljina startnog od trenutnog i trenutnog od susednog

// Razdaljina trenutnog od susednog je 1 ako dele stranicu, a sqrt(2) ako su susedni dijagonalno

double moguceG = trenutno.G + 1;

if (sused.Pozicija.X != trenutno.Pozicija.X && sused.Pozicija.Y != trenutno.Pozicija.Y)

moguceG += Math.Sqrt(2) - 1;

// a) Ako je polje neotvoreno to znaci da mu nikad nije dodeljena G vrednost, pa u tom slucaju svakako treba postaviti G vrednost prvi put

// b) G vrednost treba izmeniti kada je nova manja (bolja) od G vrednosti koja je izračunata ranije na drugi način (iz druge putanje)

// Kada se postavlja nova G vrednost, treba postaviti i novog prethodnika polja, jer je pronadjen novi optimalniji put

if (!open.Contains(sused) || moguceG < sused.G)

{

sused.G = moguceG;

sused.parent = trenutno;

// Ako polje već nije, treba da dodati u skup otvorenih polja, i grafički ga označiti kao otvoreno

if (!open.Contains(sused))

{

open.Add(sused);

sused.MarkOtvoren(g);

// Usporavanje izvršavanja radi preglednosti

Thread.Sleep(simulator\_delta\_t);

}

}

//Ako je traženo polje pronađeno, pretraga je gotova i putanja se može iscrtati pomoću pokazivača na roditeljska polja

if (sused.Pozicija.X == Finish.X && sused.Pozicija.Y == Finish.Y)

{

sused.CrtajPutanju(g, 2\*simulator\_delta\_t);

return;

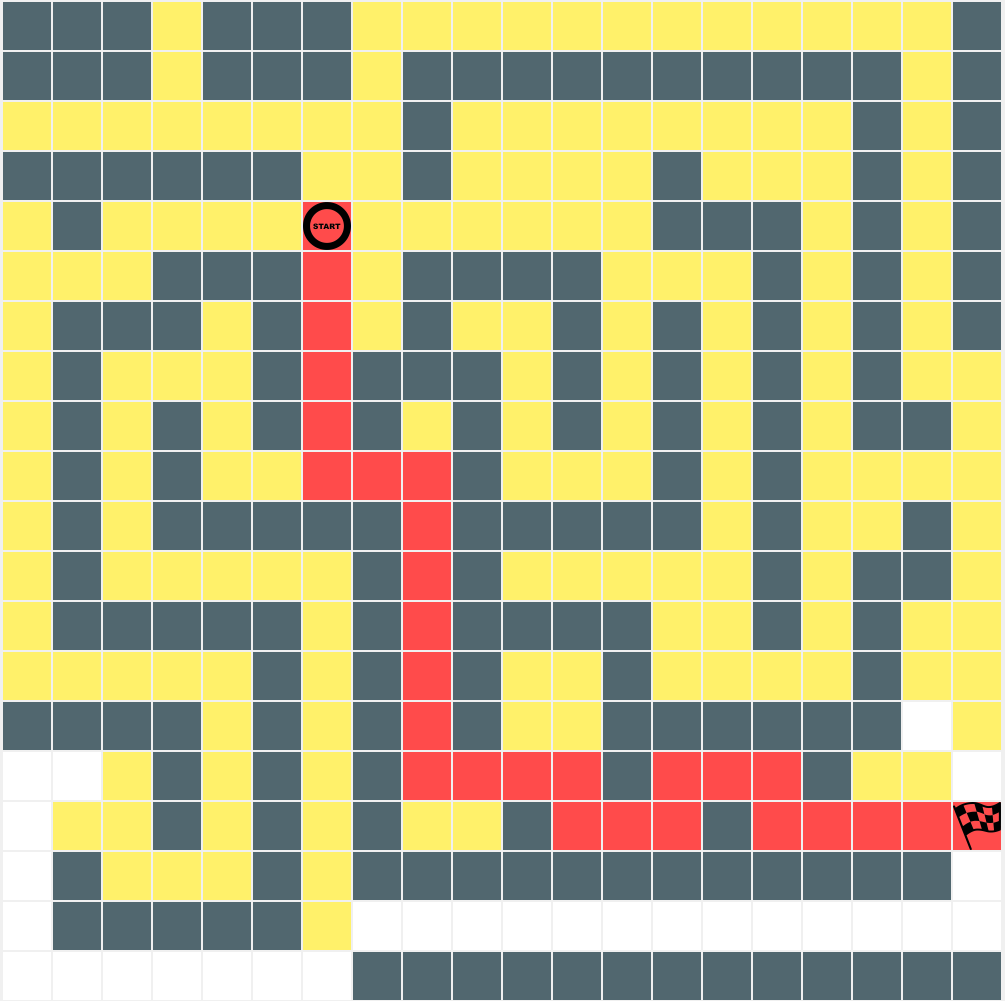
}

}

}

}

На слици 3. дат је резултат Дајкстриног алгоритма графички приказан у корисничком интерфејсу. Црвеном бојом означена је најкраћа путања између два дата поља.



Слика - Графички приказ резултата Дајкстриног алгоритма

## Похлепни алгоритам претраге по најбољем својству (Best-first Search)

Похлепни алгоритам претраге по најбољем својству (eng. Greedy Best-first Search) или чиста хеуристичка претрага је један од информисаних алгоритама претраге за разлику од претходно описаних. Информисаност овог алгоритма омогућава коришћење хеурстичке функције за одабир најбољег следећег поља у датом моменту. Информисаност се конкретно заснива на познавању позиције траженог поља, па тако и домена на коме се алгоритам извршава.

Претрага у овом алгоритму као и код претходних креће од почетног поља. У првом кораку се обилази почетно поље и суседи се додају у скуп отворених поља. Затим се као следеће поље за обилазак бира поље из скупа отворених чија је предвиђена удаљеност од циља (обично означена као h вредност) најмања, тј. бира се поље које има „најбоље својство“. При обиласку тог, али и касније било ког поља оно се склања из скупа отворених и додаје се у скуп затворених како се обилазак не би понављао. Процес обилажења тренутног и одабира следећег које је по претпоставци најбоље се понавља све док се не дође до траженог поља или до тренутка када више нема могућности за обилазак, тј. када је скуп отворених поља испражњен.

Похлепност, тј. избор поља које има најбоље својство и „делује“ као најоптималније у датом моменту даоводи до тога на овај алгоритам не гарантује проналазак најкраће путање већ било које. Овај алгоритам, иако не гарантује проналазак најкраћег пута, често је погодан због његове брзине омогућене херуистичим функцијама.

Хеуристичке функције су, као што је наведено, функције које предвиђају удаљеност (у општем случају тежину) путање између две чвора тј. поља. Хеуристичке функције растојања могу да предвиђају на више начина. Следе неки од њих:

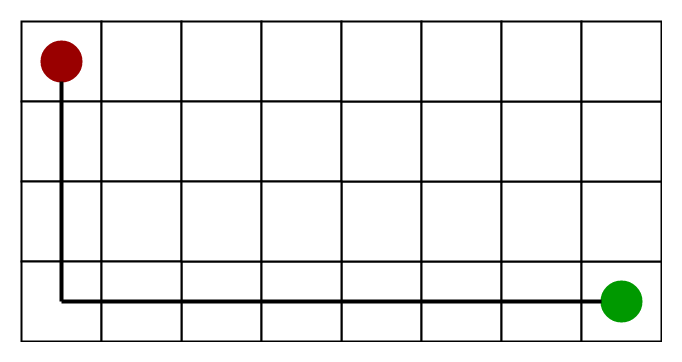
### Манхетн удаљеност

Манхетн удаљеност се израчунава као збир разлика X координата и Y координата две дате тачке.

*,*

где су

Овако израчуната удаљеност се користи у хеуристичким функцијама које треба да предвиде дужину путње при чему је дозвољено кретање искључиво између поља која деле страницу (у 4 правца).



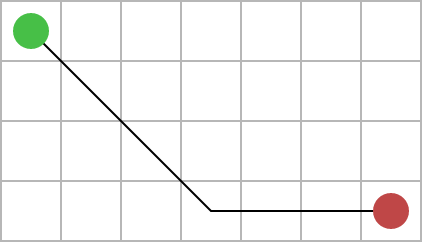
Слика - Манхетн удаљеност

### Дијагонална удаљеност

Када је кретање кроз лавиринт дозвољено и по дијагонали (у 8 праваца) Манхетн удаљеност неће бити адекватна. Дијагонална удаљеност урачунава и дијагноално кретање између два поља (имају заједничко теме). За тежину дијагоналног кретања између дав поља се обично узима , а у правцу 1, што се може променити по потреби.

,

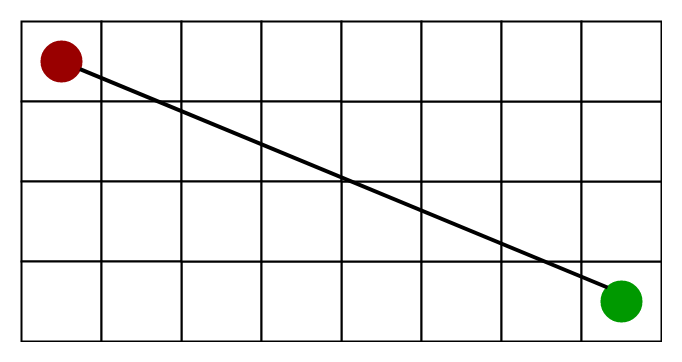
где je D тежина кретања између два поља која деле страницу, а D2 тежина кретања између поља која деле теме. Када је D=1, a D2=, ова удаљеност се назива октилна удаљеност, а Чебишевљево растојање када су и D и D2 једнака 1. Чебишевљево растојање је познато у шаху код кретања краља између два поља.



Слика – Дијагонална, октилна, удаљеност

### Еуклидска удаљеност

Еуклидска удаљеност је удаљеност при којој је дозвољено кретање под било којим углом, не само дијагонално. У свим случајевима је мања или једнака од претходне две, па уколико би се користила у случају када није дозвољено кретање под сваким углом често би подцењивала реално стање.



Слика - Еуклидска удаљеност

Конкретно у хеуристичкој функцији у симулатору креираном за потребе овог рада, употребљавају се октилно и манхетн растојање у зависности од тога да ли је дијагонално кретање дозвољено. Хеуристичка функција се код овог алгоритма позива на почетку извршавања алгоритма, и тако предвиђања постају позната код свих поља. Следи хеуристичка метода класе Lavirint:

public void CalculateH(bool strane4)

{

for (int i = 0; i < Dimenzije.Item2; i++)

for (int j = 0; j < Dimenzije.Item1; j++)

{

double dY = Math.Abs(Finish.Y - i);

double dX = Math.Abs(Finish.X - j);

if (strane4)

//Manhetn rastojanje

polja[i, j].H = dY + dX;

else

//Oktilno rastojanje

polja[i, j].H = (Math.Sqrt(2) \* Math.Min(dY, dX) + Math.Abs(dY - dX));

}

}

Следи метода похлепне претраге по најбољем својству, раније изостављена у класи Lavirint. Конкретни кораци су објашњени кроз коментаре:

/// <summary>

/// Pohlepni algoritam pretrage po najboljem svojstvu

/// </summary>

/// <param name="strane4"> Logička promenljiva, pokazuje da li je dozvoljeno dijagonalno kretanje ili ne</param>

/// <param name="g"> Grafika korisničkog interfejsak po kojoj se iscrtava prolaz kroz lavirint </param>

public void BestFS(bool strane4, Graphics g)

{

//Izračunava se vreme potrebno da bi se algoritam usporio radi bolje preglednosti, prema proporciji

//25 : P = X : 40, P-povrsina pravougaonika odredjena start i finish tackama - obrnuta proporcija dobijena testriranjem

simulator\_delta\_t = (int)(25 \* 40 / ((Math.Abs(Start.X - Finish.X) + 1) \* (Math.Abs(Start.Y - Finish.Y) + 1)));

HashSet<Polje> open = new HashSet<Polje>();

HashSet<Polje> closed = new HashSet<Polje>();

CalculateH(strane4);

// Po;etno polje se oyna;ava kao otvoreno, dodavanjem u skup

open.Add(polja[Start.Y, Start.X]);

while (open.Count > 0)

{

//Za trenutno polje se uzima ono koje trenutno ima najbolju H vrednost u skupu otvorenih polja

Polje trenutno = MinimumH(open);

//Trenutno polje se uklanja iz otvorenih, a dodaje u zatvorena polja

open.Remove(trenutno);

closed.Add(trenutno);

//Usporavanje izvršavanja radi preglednosti

Thread.Sleep(simulator\_delta\_t);

//Grafički prikaz otvorenog polja na GKI

trenutno.MarkOtvoren(g);

foreach (Polje sused in SusednaPolja(trenutno, strane4))

{

//Ako je sused zatvoren, preskače se kako se ne bi ponavljao

if (closed.Contains(sused))

{

continue;

}

//Ako već nije sused se dodaje u skup otvorenih polja

if (!open.Contains(sused))

{

open.Add(sused);

}

//Trenutno polje se postavlja kao prethodnik susednom u hijerarhiji putanje

sused.parent = trenutno;

//Ako je sused traženo polje, pretraga je uspešno završena, i putanja se iscrtava

if (sused.Pozicija.X == Finish.X && sused.Pozicija.Y == Finish.Y)

{

sused.CrtajPutanju(g, 2 \* simulator\_delta\_t);

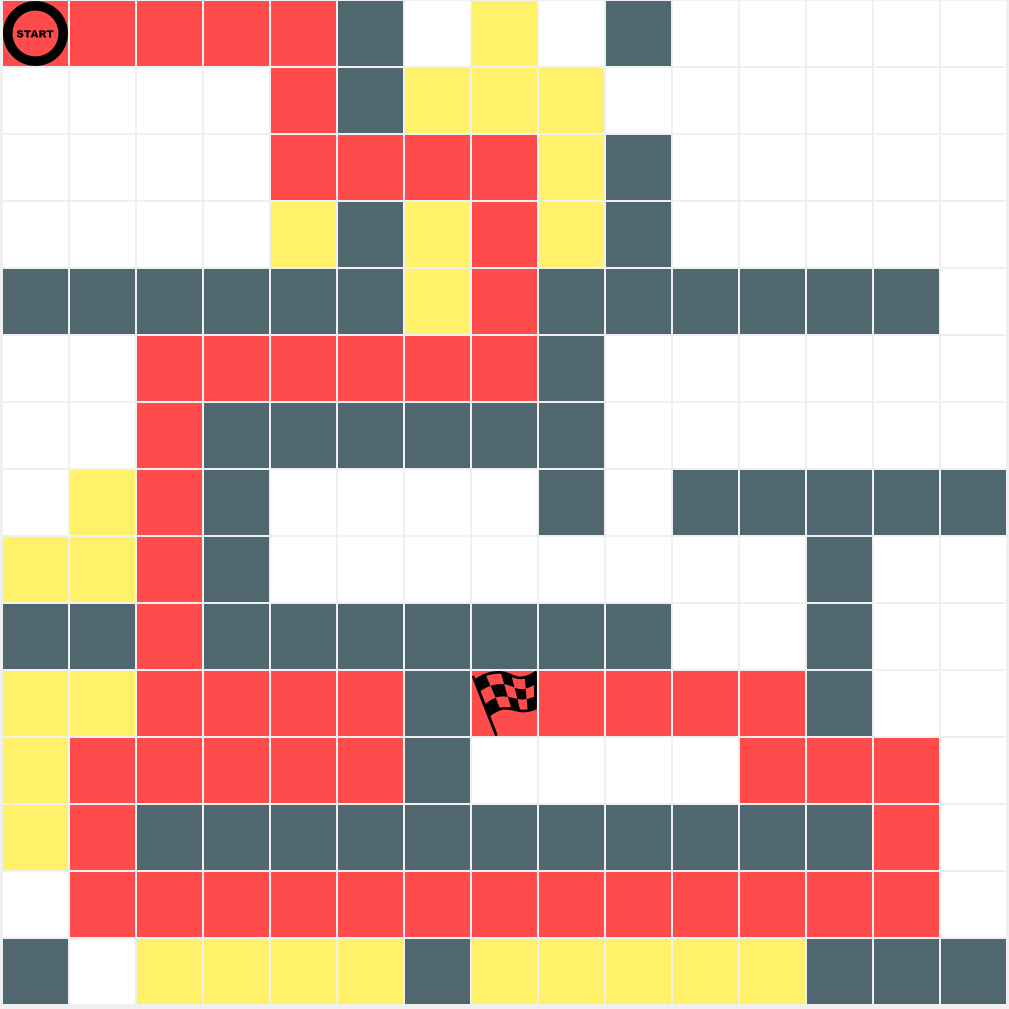
return;

}

}

}

}



Слика - Приказ резултата претраге по најбољем својству у ГКИ

Као што се види на слици 7 путања пронађена овим алгоритмом није најкраћа, али је број отворених поља значајно мањи него код Дајкстриног алгоритма, што говори о брзини претраге по најбољем својству.

## А\* алгоритам

A\* алгоритам је још један информисан алгоритам претраге. Ово је један од најпопуларнијих алгоритама, који се користи у вештачкој интелигенцији. А\* алгоритам је заправо уопштена верзија Дајкстриног алгоритма која користи и хеуристичку функцију као алгоритам претраге по најбољем својству. Овај алгоритам су на истраживачком институту развили Стенфорд 1968. године Питр Харт, Нилс Нилсон и Бертрам Рафаел. А\* алгоритам гарантује проналазак најкраће путање.

Алгоритам А\* хеуристичком функцијом смањује број отворених поља у односу на Дајкстрин алгоритам, али и даље користи растојање поља од почетног поља које је срж Дајкстриног алгоритма. Наиме, као и у Дајкстрином алгоритму, чворови који треба да се обиђу се чувају у реду или сличној структури (скуп отворених поља) и узимају се сортирано према одређеном критеријуму. Тај критеријум је код неинформисаног Дајкстриног алгоритма била искључиво вредност *g* (удаљенос од почетка). Међутим, у А\* алгоритму овај критеријум је проширен и предсавља га збир *g* и *h* вредности, где је вредност *h* резултат хеуристичке функције. Тај збир се уобичајено обележава као вредност *f.*

У А\* алгоритму хеуристичка функција може да процењује дистанцу поља од циља истим методама као у описаном алгоритму претраге по најбољем својству. Као и у алгоритму претраге по најбољем својству, у А\* алгоритму је од кључне важности да се одабере адекватна метода. Уколико би хеуристичка функција преценила путању до циља проналазак најкраће путање не би био гарантован, а у крајњем случају А\* би функционисао исто као прохлепна претрага по најбољем својству, тј. чиста хеуристичка претрага. С тога, важно је да хеуристичка процена увек буде мања или једнака реалној дужини путање, што се добија одабиром адекватне хеуристичке функције. Са друге стране уколико би резултат хеуристичке функције био увек 0 или занемарљиво мали, А\* би функционисао као Дајкстрин алгоритам и изгубило би се на брзини.

У свим аспектима, осим у одабиру следећег поља у датом тренутку А\* алгоритам функционише исто као Дајкстрин алгоритам, те није потребно поново објашњавати. Хеуристичка функција која се користи у креираном А\* алгоритму иста је као раније описана код похлепног чисто хеурисичког алгоритма. Зато, следи код методе А\* алгоритма која припада класи Lavirint, а која је раније изостављена.

/// <summary>

/// Pohlepni algoritam pretrage po najboljem svojstvu

/// </summary>

/// <param name="strane4"> Logička promenljiva, pokazuje da li je dozvoljeno dijagonalno kretanje ili ne</param>

/// <param name="g"> Grafika korisničkog interfejsak po kojoj se iscrtava prolaz kroz lavirint </param>

public void AStar(bool strane4, Graphics g)

{

//Izračunava se vreme potrebno da bi se algoritam usporio radi bolje preglednosti, prema proporciji

//25 : P = X : 40, P-povrsina pravougaonika odredjena start i finish tackama - obrnuta proporcija dobijena testriranjem

simulator\_delta\_t = (int)(25 \* 40 / ((Math.Abs(Start.X - Finish.X) + 1) \* (Math.Abs(Start.Y - Finish.Y) + 1)));

HashSet<Polje> open = new HashSet<Polje>();

HashSet<Polje> closed = new HashSet<Polje>();

// Izračunavaju se predviđene razdawine između polja lavirinta i cilja, heurističkom metodom

CalculateH(strane4);

// Početno poqe se dodaje u skup otvorenih polja

open.Add(polja[Start.Y, Start.X]);

// Iteracija traje dok ima mogućnosti dalje pretrage ili do trenutka kada je prektinuta pronalaskom cilja

while (open.Count > 0)

{

// Za trenutno polje se uzima polje iz open skupa sa najmanjom F vrednošću

Polje trenutno = MinimumF(open);

Thread.Sleep(simulator\_delta\_t);

// Trenutno polje se uklanja iz skupa otvorenih i stavlja u skup zatvorenih polja

open.Remove(trenutno);

closed.Add(trenutno);

// Ako je cilj pronđen, pretraga je gotova i putanja može da se iscrta

if (trenutno.Pozicija.X == Finish.X && trenutno.Pozicija.Y == Finish.Y)

{

trenutno.CrtajPutanju(g, 2\*simulator\_delta\_t);

return;

}

foreach (Polje sused in SusednaPolja(trenutno, strane4))

{

// Posećena polja ne treba ponovo da se obilaze, pa se preskaču

if (closed.Contains(sused))

{

continue;

}

// Razdaljina od starnog polja do suseda (moguceG) je jednaka zbiru razdaljina startnog od trenutnog i trenutnog od susednog

// Razdaljina trenutnog od susednog je 1 ako dele stranicu, a sqrt(2) ako su susedni dijagonalno

double moguceG = trenutno.G + (sused.Pozicija.X != trenutno.Pozicija.X && sused.Pozicija.Y != trenutno.Pozicija.Y ? Math.Sqrt(2) : 1);

// a) Ako je polje neotvoreno to znaci da mu nikad nije dodeljena G vrednost, pa u tom slucaju svakako treba postaviti G vrednost prvi put

// b) G vrednost treba izmeniti kada je nova manja (bolja) od G vrednosti koja je izračunata ranije na drugi način (iz druge putanje)

// Kada se postavlja nova G vrednost, treba postaviti i novog prethodnika polja, jer je pronadjen novi optimalniji put

if (!open.Contains(sused) || moguceG < sused.G)

{

sused.G = moguceG;

sused.parent = trenutno;

// Ako polje već nije, treba da dodati u skup otvorenih polja, i grafički ga označiti kao otvoreno

if (!open.Contains(sused))

{

open.Add(sused);

// susedno polje se označava kao otvoreno u GKI

sused.MarkOtvoren(g);

// Usporavanje izvršavanja radi preglednosti

Thread.Sleep(simulator\_delta\_t);

}

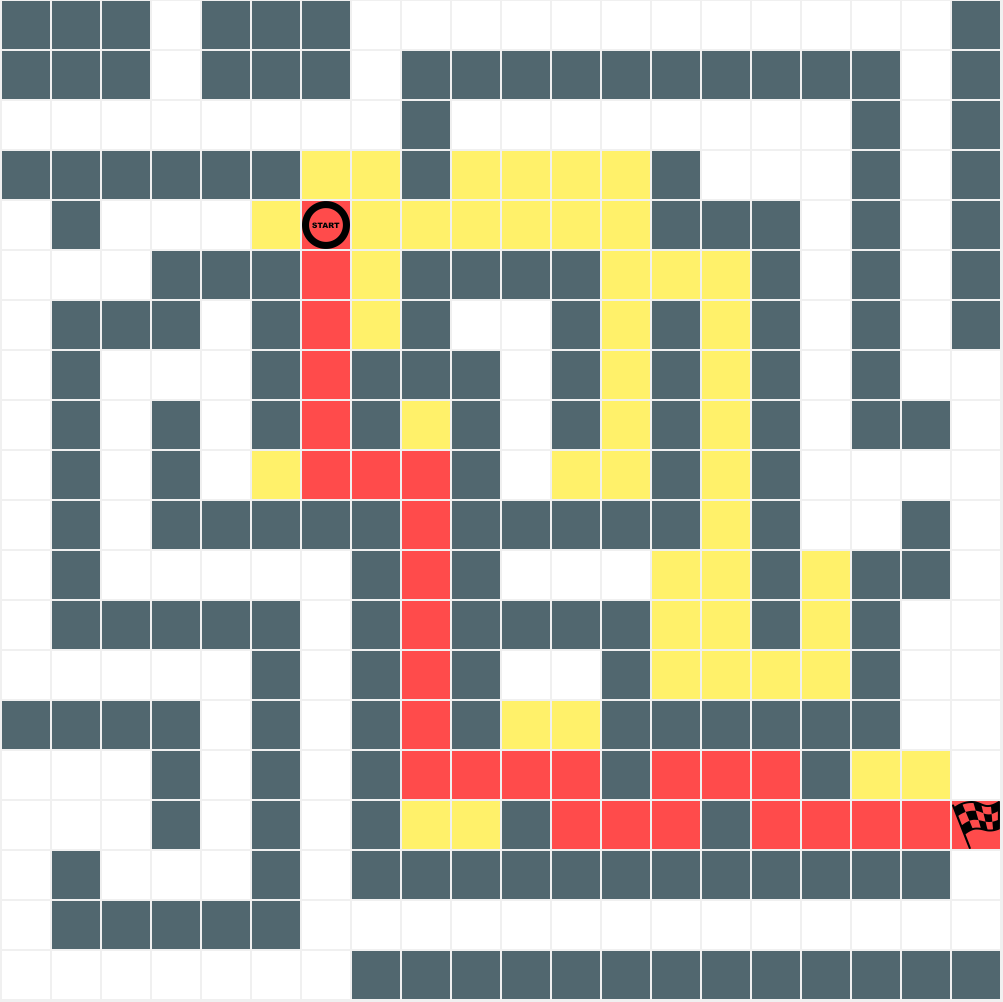
}

}

}

}

Резултат А\* алгоритма приказан у графичком корисничком интерфејсу приказан је на слици 8. Као што се може уочити, овај алгоритам је пронашао најкраћу путању, означену црвеном бојом, а отворио је знатно мање поља од Дајкстриног алгоритма, на истом лавиринту (слика 3).



Слика - Резултат А\* алгоритма на ГКИ

# Закључак

У овом раду описани су различити пристуи у решавању проблема програмског проласка кроз лавиринт. Описано је пет алгоритама, међу којима су три неиформисана алгоритма претраге и два информисана. Најчешће коришћени алгоритми за проналазак гарантовано најкраћег пута су А\* и Дајкстрин алгоритам. Међутим, међу таковим алгоритмима А\* алгоритам показује најбољу брзину и ефикасност у претрази лавиринта због своје информисаности. Брзина алгоритама претраге је обично обрнуто сразмерна броју посећених поља, који је код информисаних алгоритама мањи. Код проблема који не захтевају сигуран проналазак најкраћег пута алгоритам претраге у дубину и чиста хеуристичка претрага такође показују ефиксонст, мада то за алгоритам претраге у дубину важи само у специфичним случајевима.

Описани алгоритми претраге су кључни алати у решавању различитих проблема графова и сродних структура (међу којима су лавиринти), а избор алгоритма зависи од специфичности проблема, циљева које желимо да постигнемо, али и временских и просторних ограничења која конкретан случај може да наметне. Примена ових алгоритама је широка, користе се у вештачкој интелигенцији (поготово А\*), навигационим системима, графичкој обради слика, дигиталној безбедности, телекомуникацијама и многим другим областима. Суштина алгоритама претраге је иста у свим наведеним областима, као и у описаном проблему проласка кроз лавиринт.

# Прилог

Следи адреса GitHub репозиторијумa где се налази целокупан пројекат креиран за потребе овог рада, пројекат је креиран у програмском језику C#: [2004Milos/Izlaz-iz-lavirinta (github.com)](https://github.com/2004Milos/Izlaz-iz-lavirinta)

# Литература

1. Dijkstra, Edsger, A note on two problems in connexion with graphs, у: Numerische Mathematik, 1959, стр. 269-271.
2. Vöcking, Berthold и др. Algorithms Unplugged, Springer, Берлин, 2011, стр. 69-75.
3. [www.geeksforgeeks.org](http://www.geeksforgeeks.org) (<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>), 26.04.2023.
4. [www.geeksforgeeks.org](http://www.geeksforgeeks.org) (<https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>) 26.04.2023.
5. [www.theory.stanford.edu](http://www.theory.stanford.edu) (<https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>)
6. [www.theory.stanford.edu](http://www.theory.stanford.edu) (<https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>)

Слике

Слика 1 - Графички приказ резултата претраге у ширину 13

Слика 2 - Графички приказ резултата претраге у дубину 15

Слика 3 - Графички приказ резултата Дајкстриног алгоритма 18

Слика 4 - Манхетн удаљеност 20

Слика 5 – Дијагонална, октилна, удаљеност 20

Слика 6 - Еуклидска удаљеност 21

Слика 7 - Приказ резултата претраге по најбољем својству у ГКИ 23

Слика 8 - Резултат А\* алгоритма на ГКИ 26

Датум предаје матурског рада: \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Комисија:

Председник \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Испитивач \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Члан \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Коментар:

Датум одбране: \_\_\_\_\_\_\_\_\_\_\_\_\_ Оцена\_\_\_\_\_\_\_\_\_\_ (\_\_\_)