Vizualizacija algoritma za problem maksimalnega pretoka omrežja

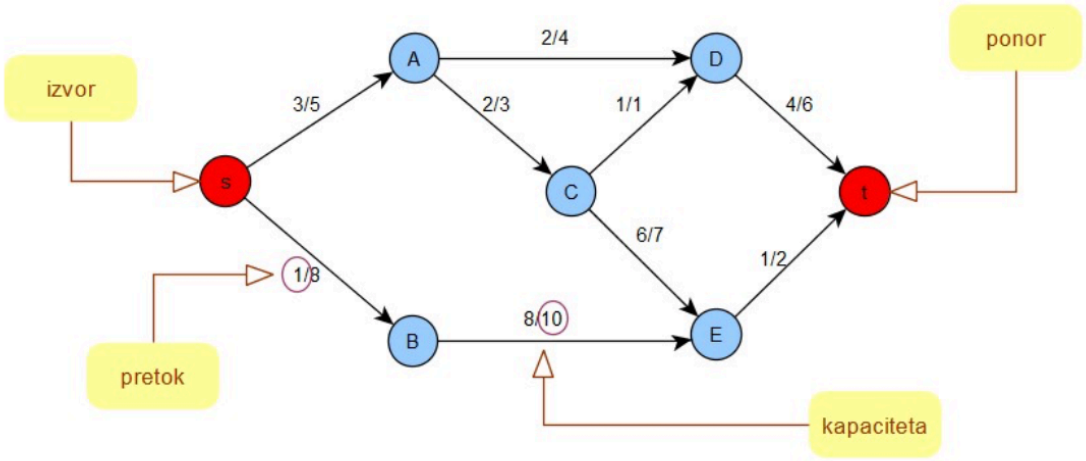
1. Zasnova programa

1. Problem maksimalnega pretoka skozi omrežje

Program bo rešil problem maksimalnega pretoka skozi omrežje z Ford-Fulkersonovim algoritmom. Gre za pogost problem, na katerega se da pretvoriti marsikateri omrežni problem. Skozi omrežje točk z povezavami, skozi katere lahko prepeljemo le omejeno število enot, želimo spravimo čimveč enot.

Osnovna terminologija problema je naslednja:

- začetno vozlišče imenujemo **izvor,**
- končno vozlišče imenujemo **ponor,**
- vrednosti na povezavah predstavljajo omejitev števila enot, ki jih lahko skozi povezavo prepeljemo, ta vrednost se imenuje **kapaciteta.**
- vrednost enot, ki jih spravimo do ponora imenujemo vrednost **f**,
- pot, ki nam to vrednost poveča, pa **f-povečujoča** pot.



Ford-Fulkersonov algoritem

Cilj je od izvora s do ponora t prepeljati največ enot, z upoštevanjem omejitvev podanih s kapacitetami povezav. To naredimo tako, da iščemo take poti, ki povečajo končno število prepeljanih enot do ponora t.

Ford-Fulkersonov algoritem za pretok skozi omrežje je zelo preprost. Prikazan je s spodnjo psevdo kodo:

Vhodni podatek: omrežje
Izhodni podatek: pretok z maksimalno vrednostjo
Potek:
1. Privzemimo, da so na vseh povezavah pretoki 0
2. Poiščemo f-povečujočo pot:
   - če obstaja: izračunamo pridružene kapacitete in vrednost f,
   - sicer: končamo in vrnemo f

Algoritem nam zagotavlja, da pridemo do rešitve ne glede na izbire poti. Prav tako nam zagotavlja, da bo rešitev celoštevilska, kar pomeni, da bodo vsi pretoki celoštevilski. Po teoriji iz linearnega programiranja se algoritem zagotovo ustavi.

2. Funkcionalnost programa

Program bo torej rešil problem maksimalnega pretoka omrežja z Ford-Fulkersonovim algoritmom, pri čemer je poudarek na vizualizaciji delovanja algoritma.

Delovanje bo prikazoval z preprostejšimi animacijami in sprembami barv, tako da bo med izvajenjem programa uporabnik dobil informacijo, za katero pot se je program v nekem trenutku odločil in kako je z potjo povečal vrednost enot, ki jih je do tekrat prepeljal od ponora do izvora. Dodatne informacije bo dobil skozi sporočila, ki se bodo posodabljala z uporabnikovo uporabo in skozi izvajanje algoritma. Ker so take aplikacije precej statične, želim s animacijami in sporočili izboljšati odzivnost programa.

Program bo imel 2 glavni funkciji:

- Risanje omrežja, pri čemer bo uporabnik imel možnosti:
  - vnos števila vozlišč omrežja, pri čemer bo to število omejeno,
  - dodaja novega vozlišča (do vnesene vrednosti),
  - izbris obstoječega vozlišča (mogoče!),
  - dodaja povezave,
  - spreminjanje kapacitete povezave,
  - označi izvir in ponor,
  - skozi sporočila bo videl, ali je graf že veljavno omrežje ali ne (če ni, ne more izvesti reševanja z algoritmom),
  - počisti risalno polje (začni znova, brez ponovnega zagona aplikacije).
- Prikaži zgled: namen je uporabniku prikazati veljaven primer omrežja in delovanja programa, v kolikor ga ne zanima rešitev njegovega primera in želi vpogled le v delovanje algoritma z programom.

3. Izgled programa

4. Delovanje programa

Program bo deloval na naslednji način. Ob odprtju aplikacije bo uporabnika čakalo prijazno sporočilo, ki mu bo sporočilo, da lahko s aplikacijo reši problem maksimalnega pretoka in da ima dve možnosti, ali ga sam nariše, ali si pa ogleda veljaven zgled omrežja.

Verjetno se bo najprej odločil za zgled. Ko bo stisnil na gumb *"Prikaži zgled"*, se bo na delu za risanje prikazalo omrežje z že določenimi kapacitetami, obenem pa bo gumb *"Reši zgled"* postal aktiven. Ob kliku nanj se bo prikazal del, ki nam kaže napredek animacije, pri čemer se bo algoritem začel izvajati in animacija poti dejansko začela teči. Vrednost f (torej pretok) bo po vsakem pretoku po (f-povečujoči) poti "popnila", število bo bilo torej animirano tako, da se bo za trenutek povečalo in spet zmanjšalo na prvotno vrednost, s čemer bo uporabnika opozorilo, da večamo to vrednost.

Sporočila nad risalnim delom se bodo vseskozi posodabljala. Med risanjem bo pisalo *"Animacija se izvaja, prosim počakajte."*, ob koncu *"Problem je rešen. Maksimalni pretok skozi dano omrežje je _."* itd.

V kolikor se uporabnik odloči, da bi rad narisal tudi svoje omrežje in klikne na gumb *"Nariši omrežje"*, pri čemer se mu bo prikazalo okno, ki ga bo vprašalo po številu vozlišč v omrežju (potem je implementacija algoritma za narisano omrežje malo preprostejša). Ko bo potrdil izbiro (na voljo bo imel le omejeno število vozlišč, na primer 10 ali 15), bo mu sporočilo sporočilo, da naj klikne nekam na risalno površino, da doda vozlišče in povleče iz vozlišča na vozlišče, da naredi povezavo. Pri tem se bo aktiviral gumb *"Končaj z risanjem"*, ob kliku nanj pa se bosta zgodila dve možni stvari:

- uporabnikov graf ne bo omrežje: v tem primeru bo se prikazalo ustrezno sporočilo,
- uporabnikov graf bo veljavno omrežje: v tem primeru se bo pojavilo sporočilo, da naj označi izvir, potem pa še ponor.

Ko bo izvir in ponor na veljavnem omrežju ustrezno označil, bo dobil na voljo možnost *"Reši problem"*. Problem se bo rešil na enak način kot zgled.

Nato ima uporabnik spet enake možnosti, lahko tudi sam zbriše risalno površino, sicer pa se to zgodi ob kliku na kateregakoli izmed gumbov *"Pokaži zgled"* ali *"Nariši omrežje"* (drugi gumbi pa so v tem trenutku onemogočeni).

2. Ideja implementacije

Ker imamo program z precej gumbi, ki morajo biti v nekih trenutkih onemogočeni, v nekaterih omogočeni, bi bilo najbrž pametno pri vsakem sproženem gumbu samo nastaviti, kateri gumb se je sprožil oziroma v katerem delu izvajanja aplikacije smo, vse odzive pa nastaviti v eni funkciji. Tako bo koda bolj pregledna, pa tudi kasnejše spremembe bodo lažje.

Status/del izvajanja aplikacije na katerem smo, lahko shranjujemo v `enum`-u, ker je aplikacija vedno natanko v enem stanju.

`Enum` bo imel naslednja stanja:

- `Initialized`: stanje pred katerim koli klikom
- `ShowExample`: stanje po kliku gumba *"Pokaži zgled"*
- `Solve`: stanje po kliku gumba *"Reši primer"*
- `Draw`: stanje po kliku gumba *"Nariši omrežje"* in pred dejanskim uporabnikovim risanjem
- `DrawingNetwork`: stanje med uporabnikovim risanjem, med tem stanjem mu dajemo vedeti, ali je njegov graf dejansko omrežje, ali še ni v redu
- `EndDrawing`: stanje po kliku gumba *"Končaj z risanjem"*, gumb lahko klikne šele ko je graf dejansko omrežje
- `ClearDrawingArea`: stanje po kliku gumba *"Pobriši omrežje"*, gumb se prikaže po izvedbi animacije oziroma rešitvi problema

Funkcija, ki bo poskrbela za ustrezne odzive bo implementirana z `switch` stavkom, ki bo preklapljal vrednost `enum`-a, ki bo bil ob zagonu aplikacije nastavljen na `.Initialized`.

Isti `enum` nam bo koristil tudi za odločanje, katero sporočilo prikazati v določenem trenutku. Namesto da stvari po klikih nastavljamo direktno v metodah, ki so poklicane ob teh dogodkih, je zagotovo za kasnejše spremembe in organizacijo bolje, da imamo vse na enem mestu.

Za shrambo nizov, ki se naj v nekem trenutku prikažejo naredimo posebno strukturo. Ker je smiselno, da ta struktura vsebuje tudi metodo, ki se glede na stanje aplikacije (oziroma kar `enum`-a) odloči, katero sporočilo prikazati in ga vrne, jo implementirajmo kar kot razred, ker je žal že prekompleksna za `struct`. V glavnem programu potem samo pokličemo to metodo (razreda za sporočila), z argumentom trenutnega stanja aplikacije, ki nam vrne ustrezno sporočilo in to nastavimo za tekst sporočila.

Implementacija algoritma v aplikacijo

Metode povezane z algoritmom bodo v posebnem razredu, ki bo vseboval tudi metode za dodajanje povezav in vse lasnosti ter pomožne metode povezano z dejansko izvedbo in implementacijo algoritma.

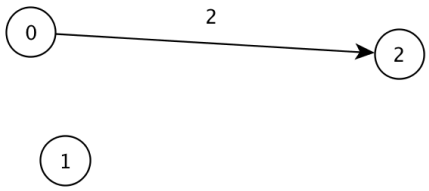
Omrežje bomo predstavili kot matriko oziroma kot tabelo, kot je navada pri takih problemih. Imamo omrežje z n vozlišči, imenujmo jih kar A_i, pri čemer `i` teče od `0` do `n`. Predstavimo ga torej kot matriko `G` velikosti `n x n`. Elementi `(i, j)` predstavljajo kapaciteto povezave med točkama A_i in A_j, pri čemer gre za usmerjene povezave. Element `(i, j)` predstavlja torej povezavo med vozliščema A_i —> A_j, medtem ko `(j, i)` predstavlja povezavo A_j —> A_i. Ničelni elementi matrike `G` pomenijo, da povezave ni, oziroma, da je kapaciteta enaka 0.

Za implementacijo je lažje, če vnaprej poznamo število vozlišč, da ne rabimo ob vsakem novem dodanem vozlišču prepisevati in večati tabele, ki predstavlja matriko. Zaenkrat uporabniku ne dajmo možnosti, da vozlišča po želji dodaja, brez da prej utemelji, koliko vozlišč bo graf vseboval. To lahko še vedno naredimo kasneje, če se bo tak način izkazal za nepriročnega za uporabnika.

Po številu vozlišč omrežja ga lahko vprašamo z novim oknom oziroma novim `Form`-om, ki ga lahko oblikujemo s oblikovalcem (`.cs Designer`-jem). Ta mora vsebovati neko sporočilu, ki uporabniku da vedeti, kaj od njega želi, okence, kamor lahko vnese število in gumba *"Potrdi"* in *"Prekliči"*, v primeru, da si je uporabnik premislil ali ponesreči pritisnil na gumb *"Nariši omrežje"*. Gumbom nastavimo odzive (`OK`, `Cancel`), tako da se okno ob poklicu iz aplikacije obnaša podobno kot `DialogBox`, in lahko v glavni aplikaciji zelo enostavno uporabljamo rezultate uporabnikove odločitve. Odzive nastavimo z spremembo lastnosti posameznega gumba, ki se imenuje `DialogResult`. Do vrednosti, ki jo je uporabnik vnesel, pa iz glavne aplikacije dostopamo tako, da novemu oknu nastavimo javno lastnost, ki shranjuje vrednost, ki predstavlja število vozlišč omrežja (avtomatsko nastavljena na 0). Ob vsaki spremembi se ta vrednost posodobi, ob uporabnikovem kliku na *"Potrdi"*, pa jo lahko enostavno preberemo in posredujemo razredu za algoritem maksimalnega pretoka.

Risanje omrežja

Kot smo rekli, bomo omrežje predstavili kot matriko `G` (v dejanski implementaciji bomo poimenovali majn generično). Vozlišča bomo shranjevali v tabeli tabeli `A`, kjer `i`-ti element predstavlja torej vozlišče A_i. vozlišča bomo predstavili kar kot napise (števil) z narisanimi krogci okrog njih, nekako takole:



Uporabnik bo povezavo naredil z vlečenjem iz enega vozlišča do drugega. Pri tem bomo vedeli, med katerima vozliščema je naredil povezavo tako, da bomo klike zaznavali na napisih, ki bodo torej tekst imele kar enak indeksu `i`, na katerem se nahajajo v tabeli.

(Tudi povezave in napise na njih bomo shranjevali, in sicer v dvo-dimenzionalni tabeli velikosti matrike `G` (več povezav tako ne more biti)), ob vsakem dodanem napisu kapacitete (ta napis bo lahko spreminjal) pa bomo napis shranili kot ime `label_i_j`, oziroma kot nekaj, kar nam iz svojega imena ob kliku nanj takoj pove, katero kapaciteto naj spremenimo.

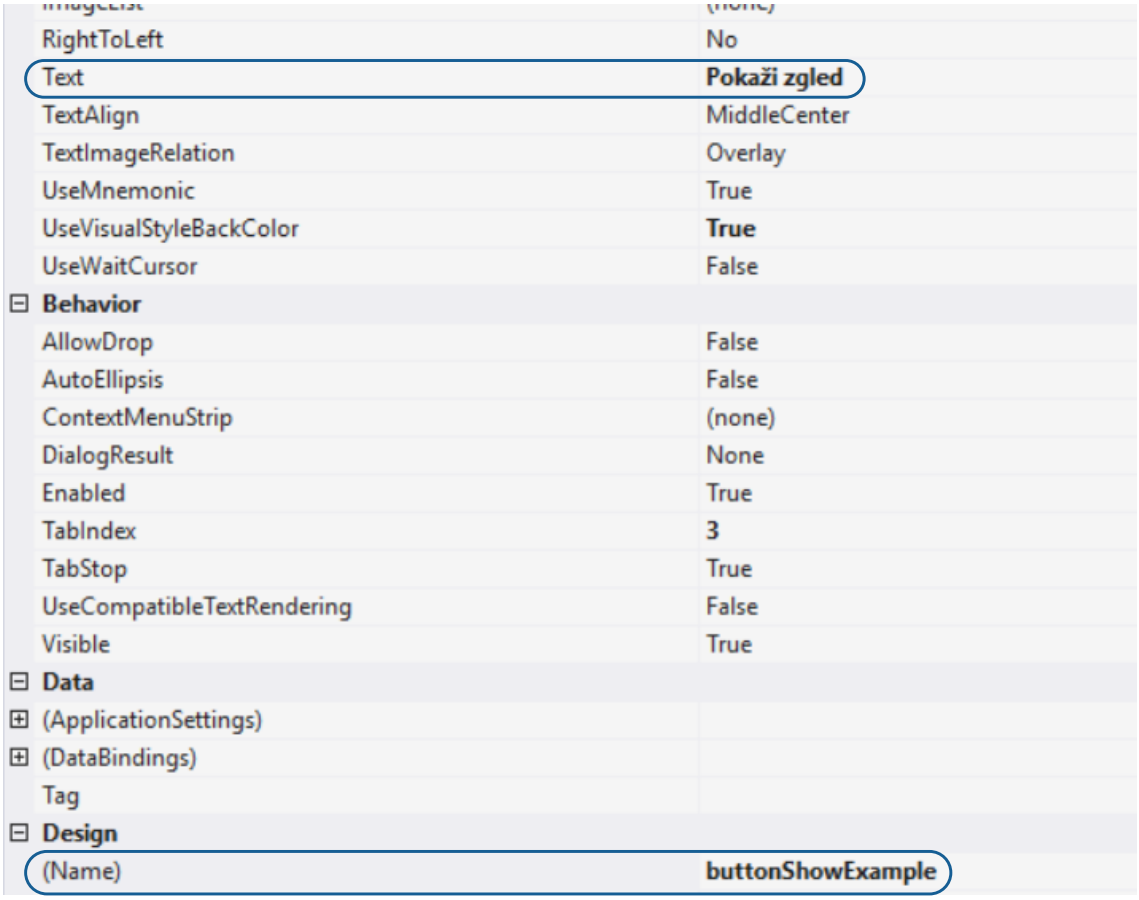
Da da ne bomo shranjevali preveč teh gradnikov, vseeno omejimo aplikacijo na na primer 15 vozlišč.

3. Implementacija

1. Začetni izgled

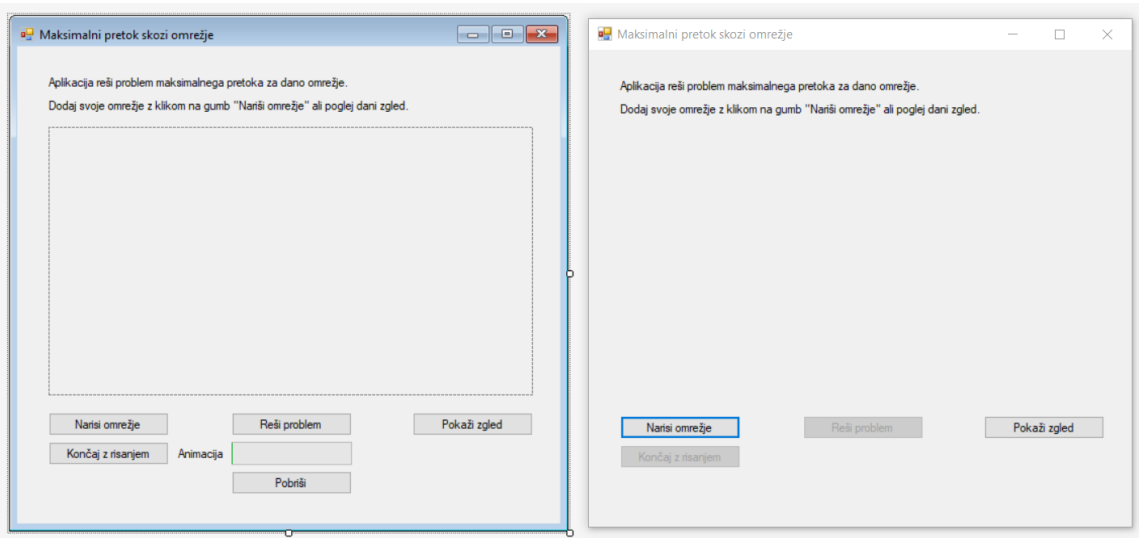
Ker je spreminjanje izgleda aplikacije z oblikovalcem (Visual Studio `Designer`) zelo enostavno, sedaj še ne rabimo točno poznati velikosti gradnikov. Na okno povlečemo gumbe, približno tako, kot smo načrtovali in dodamo tudi nekaj napisov, ki jih bomo kasneje najbrž še spremenili. Pri vsakem dodanem gradniku moramo čimprej spremeniti privzeto ime, da na to kasneje ne pozabimo, kar lahko hitro povzroči zmedo v projektu. Imena in metode imenujmo kar angleško.

Da bo organizacija kasneje preprostejša, **uvedimo konvencijo** po kateri imenujemo gradnike na naslednji način: `tipGradnikaNaseIme`, na primer gumb z napisom *"Pokaži zgled"* v `buttonShowExample` ali napis "Animacija" v `labelAnimation`. Ime in napis spremenimo pod zavihkom `Properties` na naslednji način:



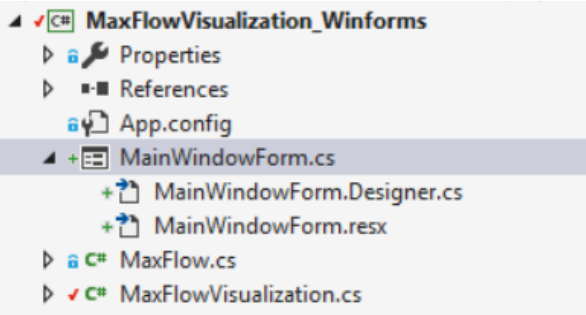
Stanje izvajanja algoritma bomo prikazovali z gradnikom `ProgressBar`, zaenkrat ga prav tako samo povlecimo na željeno mesto in ustrezno preimenujmo.

Gumbe kot so *"Končaj z risanjem"* je smiselno pred začetkom risanja skriti ali narediti neodzivne. Zaenkrat jih **naredimo neodzivne**, animacijo in njen "kazatelj napredka" pa kar **skrijmo** in ga bomo prikazali šele ob kliku na *"Reši problem"*. To naredimo prav tako pod zavihkom Properties. Nastavitve sprva izgledajo, kot da nimajo vizualnega učinka, ker se nič ne spremeni. Če aplikacijo poženemo, vidimo, da so nekateri gumbi neodzivni in del z animacijo skrit.



Izgled bomo po potrebi spreminjali in izboljševali še kasneje, to je le prototip, da lahko začnemo dodajati funkcionalnosti.

Najprej še samo **preimenujmo datoteke**, ki so sedaj poimenovane precej generično. Preimenovati jih moramo nujno z ukazom `Rename`, da Visual Studio ime spremeni v vseh referencah. `Form1.cs` spremenimo v `MainWindowForm.cs`.



2. Funcionalnost

Delovanje gumbov

Kot smo rekli, bomo funkcionalnost gumbov določali v eni metodi, ki se bo odločala, katere metode izvesti glede na stanje aplikacije (Ideja je bila razložena v poglavju *Ideja implementacije*).

Poimenujmo to funkcijo `processUserInput`, ker bo dejansko delala točno to; glede na uporabnikova dejanja se mora stanje aplikacije spremeniti. Vse te stvari bi lahko pisali direktno pri klikih gumbov, vendar ker je teh veliko, nekatere spremembe stanja aplikacije pa so povezane tudi z drugimi vhodnimi podatki, ne le kliki gumbov, bi se stvari zelo hitro zakomplicirale, če bi vse pisali pri vsaki funkciji posebej in ob vsaki spremembi morali razmišlati, točno po katerem kliku se mora določena metoda izvesti. Po klikih gumbov torej vsakič pokličimo metodo, ki poskrbi, da bo na klik uporabnik dobil ustrezen odziv aplikacije, kakšen ta odziv je, pa nas sedaj v bistvo še ne zanima.

```
///                                          USER INPUT:

5 references
private void buttonClicked() {
    processUserInput();
    updateMessage();
}

1 reference
private void buttonDraw_Click(object sender, EventArgs e){
    appState = AppState.Draw;
    buttonClicked();
}

1 reference
private void buttonShowExample_Click(object sender, EventArgs e) {
    appState = AppState.ShowExample;
    buttonClicked();
}

1 reference
private void buttonClearDrawingArea_Click(object sender, EventArgs e) {
    appState = AppState.ClearDrawingArea;
    buttonClicked();
}

1 reference
private void buttonEndDrawing_Click(object sender, EventArgs e) {
    appState = AppState.EndDrawing;
    buttonClicked();
}

1 reference
private void buttonSolve_Click(object sender, EventArgs e) {
    appState = AppState.Solve;
    buttonClicked();
}
```

Pri vsakem kliku torej spremenimo stanje aplikacije. O tem kako se glede na klike gumbov spremeni stanje aplikacije moramo razmisliti enkrat, potem pa nikoli več. Kasneje nas samo zanima, kaj v določenem stanju počnemo, kaj moramo uporabniku omogočiti in kaj onemogočiti, da ne pride do napak v aplikaciji.

Kot vidimo na sliki zgoraj, vsakič po kliku pokličemo pomožno metodo `buttonClicked`, ki prilagodi ustrezne nastavitve aplikacije in posodobi sporočilo, ki ga mora glede na stanje prikazati. Ker pri obeh metodah lahko izkoristimo dejstvo, da vedno vemo, v katerem stanju aplikacije smo, so spremenljivke podane tem metodama čisto nepotrebne. Poglejmo implementacijo metode, ki poskrbi za ustrezne odzive na uporabnikova dejanja.

```
/// <summary>
/// Responds to button clicks etc. depending on the state the app is at.
/// </summary>
1 reference
private void processUserInput() {
    switch (appState) {
        case AppState.Initialized:
            //TODO
            break;
        case AppState.ShowExample:
            showExample();
            enableSolveButton(true);
            break;
        case AppState.Solve:
            enableSolveButton(false); // user should be able to invoke this method only once
            solveCurrentExample();
            break;
        case AppState.Draw:
            enableSolveButton(false);
            //TODO in the method: change the message to say
            break;
        case AppState.EndDrawing:
            enableSolveButton(true);
            break;
        case AppState.DrawingNetwork:
            //TODO: drawing should be enabled only in this state
            enableSolveButton(false);
            break;
        case AppState.ClearDrawingArea:
            enableSolveButton(false);
            cleanDrawingArea();
            break;
        default:
            enableSolveButton(false);
            break;
    }
}
```

Metode `showExample`, `solveCurrentExample`, `cleanDrawingArea`, `enableSolveButton` še niso implementirane. To je le ogrodje za dejansko implementacijo, ki jo bomo dodajali sproti.

Da se znebimo napak v aplikaciji zaenkrat dodajmo samo prazne metode:

```
1 reference
private void showExample() {

}

1 reference
private void solveCurrentExample() {

}

1 reference
private void cleanDrawingArea() {

}
```

Opazimo lahko, da smo metodo `enableSolveButton(bool)` dodali vsem stanjem. Gumb *"Reši problem"* ne bi smel delovati tekrat, ko omrežja sploh nimamo. Lahko bi sicer v metodi preverili, ali omrežje imamo in v nasprotnem primeru sporočili uporabniku, da mora prej dodati omrežje, vendar omogočanje gumba samo v primerih, ko gumb dejansko lahko uporabi služi tudi kot napotek uporabniku, kako aplikacija deluje.

Ker je metoda zelo enostavna, jo kar implementirajmo, zraven pa še metodo za prikazovanje gumba za končanje risanja, ki prav tako mora biti omogočen samo v primerih, ko je uporabnik narisal graf, ki je omrežje.

```
2 references
private void enableSolveButton(bool shouldEnable) {
    buttonSolve.Enabled = shouldEnable;
}

0 references
private void enableEndDrawingButton(bool shouldEnable) {
    //TODO: Should only get enabled when the drawn graph is actually a network!
    buttonSolve.Enabled = shouldEnable;
}
```

Da česa ne pozabimo, si lahko delamo opombe na ustreznih mestih. Če jih poimenujemo `TODO`, nas Visual Studio kasneje lahko na njih spomni, kar je še posebej koristno v velikih aplikacijih, da ne rabimo vsega preiskovati in iskati, kje smo mislili še kaj izboljšati.

Prikazovanje sporočil

Prikazovanja sporočil, ki uporabniku pomagajo razumeti, kaj se v aplikaciji dogaja in kaj lahko dela, lahko spet zelo enostavno implementiramo. Rekli smo, da bomo nize, ki predstavljajo ustrezna sporočila hranili v posebnem razredu, zraven javne metode, ki vrne primeren niz, glede na stanje aplikacije (ki ga bomo metodi podali kot parameter).

Spet lahko uporabimo zelo podobno strukturo kot za odziv na uporabnika, torej da z `switch` stavkom preklapljamo stanje aplikacije.
Z klicom metode `getAppropriateMessage(AppState)` iz razreda `MainWindowForm` torej dobimo ustrezno sporočilo, glede na stanje aplikacije. Pri tem `switch` stavku break stavkov ne potrebujemo, ker v vsakem primeru vračamo, z čemer torej ne pademo v spodnje primere `switch` stavka

```
/// <summary> Class storing and returning messages based on app state.
2 references
class MessageText {
    private const string defaultText = "Dodajte svoje omrežje z klikom na gumb Nariši omrežje ali poglej dani zgled";
    private const string showExampleText = "Izbrali ste možnost za prikaz primera.";
    private const string solveText = "Prosim počakajte dokler se reševanje primera ne dokonča.";
    private const string drawGraphText = "Izbrali ste možnost za risanje novega grafa.";
    private const string endDrawingText = "Če želite, da se animacija reševanja izvede, stisnite na gumb Reši zgled.";
    private const string drawingText = "Stisnite za dodajo novega vozlišča, povlečite med vozliščema za dodajo nove povezave.";
    private const string clearText = "Pobrisali ste risalno površino.";

    1 reference
    public string getAppropriateMessage(AppState appState)  {
        switch (appState) {
            case AppState.Initialized:
                return defaultText;
            case AppState.ShowExample:
                return showExampleText;
            case AppState.Solve:
                return solveText;
            case AppState.Draw:
                return drawGraphText;
            case AppState.EndDrawing:
                return endDrawingText;
            case AppState.Drawing:
                return drawingText;
            case AppState.ClearDrawingArea:
                return clearText;
            default:
                return defaultText;
        }
    }
}
```

Razred shranimo v novi datoteki, poimenovani isto kot razred.

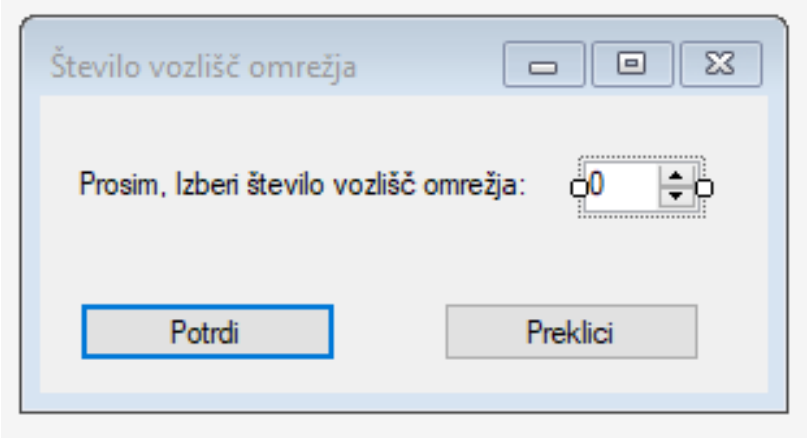
Risanje omrežja

• Prikaz okna za vnos števila vozlišč

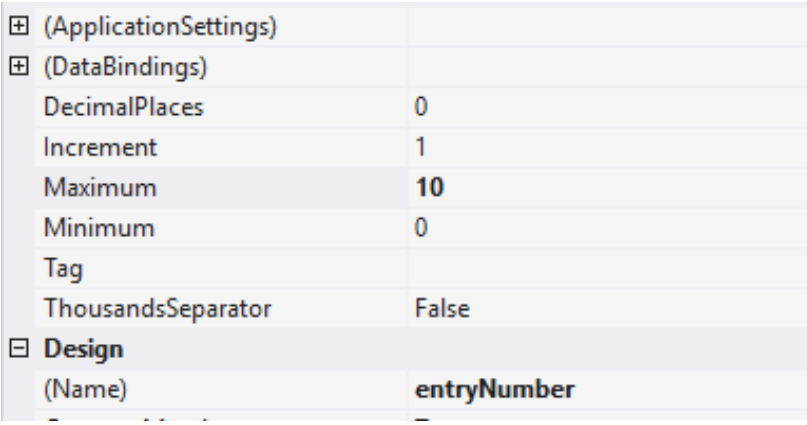
Kot smo povedali v delovanju aplikacije, uporabnik mora ob izbiri, da nariše svoje omrežje, specificirati koliko vozlišč misli v tem omrežju imeti.

To naredimo tako, da se mu prikaže okno, ki ga vpraša po tej informaciji. Ko vnese in potrdi svojo izbiro, mu sporočilo sporoči, da z kliki na risalno površino doda vozlišče, povezave pa z vlečenjem iz ene v drugo povezavo. Sporočila dodajmo kasneje pri izboljšavah aplikacije, sedaj implentirajmo funkcionalnost.

Novo okno implementiramo z novim `Form`-om, ki ga lahko oblikujemo s oblikovalcem (`.cs Designer`-jem). Okno oblikujemo nekako takole:



Omejimo pa tudi vrednost, ki jo uporabnik lahko vnese. To naredimo enostavno tako:



Kot smo že povedali v ideji implementacije, lahko gumbom nastavimo odzive, tako da lahko v glavni aplikaciji okno uporabljamo zelo podobno kot `DialogBox`. Odzive nastavimo z spremembo lastnosti posameznega gumba, ki se imenuje `DialogResult`, tako:

(DataBindings)	
(Name)	**buttonOK**
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoEllipsis	False
AutoSize	False
AutoSizeMode	GrowOnly
BackColor	Control
BackgroundImage	(none)
BackgroundImageLayout	Tile
CausesValidation	True
ContextMenuStrip	(none)
Cursor	Default
DialogResult	**OK**

Do vrednosti, ki jo je uporabnik vnesel, iz glavne aplikacije dostopamo tako, da novemu oknu nastavimo javno lastnost, ki shranjuje vrednost, ki predstavlja število vozlišč omrežja (avtomatsko nastavljena na 0). Ob vsaki spremembi se ta vrednost posodobi, ob uporabnikovem kliku na *"Potrdi"*, pa jo lahko enostavno preberemo in posredujemo razredu za algoritem maksimalnega pretoka. Implementacija okna je prikazana spodaj.

```
2 references | 0 changes | 0 authors, 0 changes
public partial class EnterNumberOfNodesForm : Form {
    2 references | 0 changes | 0 authors, 0 changes
    public int entryValue { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public EnterNumberOfNodesForm() {
        entryValue = 0;
        InitializeComponent();
    }

    1 reference | 0 changes | 0 authors, 0 changes
    private void buttonOK_Click(object sender, EventArgs e) {
        entryValue = (int)entryNumber.Value;
    }
}
```

V glavni aplikaciji okno potem prikažemo s klicom metode `showEntryBox`:.

```
///                                    ENTRY FORM - asks the user for the number of nodes his network will have:

1 reference | 0 changes | 0 authors, 0 changes
private void showEntryBox() {
    EnterNumberOfNodesForm entryForm = new EnterNumberOfNodesForm();

    if (entryForm.ShowDialog(this) == DialogResult.OK) { // shows the form and checks if user selected OK
        labelMainMessage.Text = entryForm.entryValue.ToString(); // if he did, we retrieve the value
    }
    entryForm.Dispose();
}
```

Vidimo, da je uporaba okna na tak način res zelo preprosta in skoraj enaka uporabi `DialogBox`-a.

To metodo pokličemo tekrat, ko se uporabnik odloči, da bi rad narisal graf, nas pa zanima, kako velik ta graf bo. To je torej tekrat, ko klikne na gumb *"Nariši omrežje"*. V metodo `processUserInput` pod primer `AppState.Draw` torej dodamo klic metode.

```
            solveCurrentExample();
            break;
        case AppState.Draw:
            enableSolveButton(false);
            showEntryBox();
```

- ## Dodajanje novega vozlišča

V urejevalcu videza dodamo risalno površino in jo ustrezno poimenujemo. Sedaj moramo napisati metode, ki bodo na uporabnikov klik na risalno površino dodale ustrezno vozlišče. Kot smo že premislili, bomo vozlišča risali kot številske napise z krogci okrog napisa. Do same risbe ne moremo dostopati. Krog lahko narišemo, ne moremo pa zaznavati klikov nanj. Do napisov pa lahko. Prav tako jim lahko enostavno nastavimo ime in jih shranimo, zato so idealni za implementacijo vozlišč, ki morajo biti različno poimenovana, pa še zmožni moramo biti zaznavati klike na njih (za dodajanje povezav).

Za dodajanje povezav bo pomembno, da vemo med katerima dvema vozliščema želimo dodati povezavo, zato jim moramo nastaviti imena, iz katerih bo razvidno, za katero vozlišče gre in jih shranjevati v na primer tabeli. Shraniti moramo naslednje lastnosti:

- informacijo o trenutnem številu vozlišč na risalni površini,
- maksimalnem številu vozlišč in
- tabelo, v kateri so vozlišča (oziroma samo napisi) shranjena.

Prav tako bomo morali napisati:

- metodo, ki inicializira tabelo vozlišč, da lahko vanjo dodajamo vozlišča,
- metodo, ki doda gradnik tipa napis v tabelo,
- metodo, ki izbriše vsa vozlišča (in izprazni tabelo) in
- metodo, ki dejansko doda vozlišče (napis in krogec) na risalno površino.

Ko imamo opravka s več med seboj povezanimi lastnosti in metodami, jih lahko spravimo v poseben razred za boljšo organizacijo in preglednost glavnega programa.

Dodajmo nov razred imenovan `LabelNodes` in napišimo katere lastnosti in metode smo presodili, da bo potreboval. Ker bomo zaradi risanja potrebovali tudi dostop do glavnega okna, risalne površine itd., rabimo v razredu tudi povezavo do glavnega okna. Ker sta metodi za inicializacijo tabele in dodatek novega elementa zelo enostavni, ju kar implementirajmo.

```
4 references
class LabelNodes {
    private MainWindow mainWindow;
    private Drawing drawing;

    7 references
    public int NumberOnScreen { get; set; }
    3 references
    public int MaxNumber { get; set; }
    public Label[] array;

    1 reference
    public LabelNodes(MainWindow mainWindow) {
        this.mainWindow = mainWindow;
        this.drawing = mainWindow.Drawing;
        NumberOnScreen = 0;
    }

    2 references
    public void InitializeLabelArray(int dimension) {
        this.MaxNumber = dimension;
        this.array = new Label[this.MaxNumber];
    }

    1 reference
    public void AddLabelNodeInArray(Label label) {
        this.array[this.NumberOnScreen] = label;
        this.NumberOnScreen++;
    }
```

Potrebovali bomo tudi javno metodo `AddNewLabelNode`, ki bo narisala krogec, dodala napis na ustrezno mesto in ga dodala v tabelo. Sedaj jo naredimo prazno.

Dodajmo kar risalno površino in metodo, ki zaznava klike na njej. Ker želimo, da v argumentih dobimo informacije o lokaciji klika, dodamo metodo na primer za dogodek `MouseDown`.

```
1 reference | 0 changes | 0 authors, 0 changes
private void DrawingAreaComponent_MouseDown(object sender, MouseEventArgs e) {
    posInDrArea = e.Location;
    processUserInput();
}
```

Spet odzivov na klike ne pišemo kar tukaj, ampak v metodi, ki smo jo dodali ravno v namen, da imamo vse odzive zbrane tam.

Metodi `processUserInput`, primeru `AppState.Drawing` dodamo klic metode `addNewNodeLabel`, čeprav še trenutno ni implementirana (je že, vendar prazna). To poskrbi, da se metoda za dodajanje vozlišč izvede točno ob klikih na risalno površino v stanju, ko pričakujemo, da uporabnik riše in nikoli v drugih stanjih.

Preden lahko začnemo risati v metodi `addNodeLabel`, moramo dodati shraniti in definirati nekaj lastnosti, povezanih z risanjem. Če bi naredili podoben premislek, kot za vozlišča, bi ugotovili, da rabimo precej metod in lastnosti povezanih z risanjem, kar pomeni, da je spet smiselna uporaba novega razreda.

Ta razred bomo imenovali `Drawing`. V tej nalogi ne bomo šli v podrobnosti implementacije, ker bi razlaga vsake vrstice bila mnogovezna, ne bi pa izboljšala našega razumevanja izdelave programa. V njej so uporabljeni samo osnovni koncepti risanja, vsebuje pa javne lastnosti:

- `Point areaLoc`: pozicioniranost risalne površine v oknu,
- `Point positionInArea`: tukaj bomo po vsakem kliku na risalno površino shranili zadnji klik,
- `Color PenColor`: barva, z katero rišemo,

in javne metode:

- `void DrawCircleAroundLastClick`: nariše krog okoli lokacije, kjer se je zgodil zadnji klik (na risalni površini, samo v primernem stanju aplikacije),
- `Point GetRelativeLocationOfLastClick`: vrne lokacijo, kjer želimo dodati novo vozlišče in
- `void ClearDrawingArea`: pobriše vse, kar je narisano na risalni površini.

Dejanska implementacija metod na naslednji strani.

```
references
class Drawing {
    private MainWindow mainWindow;

    private Graphics area;
    public Point AreaLoc; // drawing area location
    // mouse position in drawing area coordinates, gets set everytime user clicks on drawing area in .Drawing mode:
    public Point PositionInArea; // position in drawing area - where the user clicked to add a node

    private Pen circlePen;
    private Pen connectionPen;// we need a different pen for drawing lines, because of line caps on arrays
    private float penWidth;
    public Color PenColor;
    private Color backColor;
    public int CircleRadius;

    1 reference
    public Drawing(MainWindow mainWindow, PictureBox drAreaComp)  {
        this.mainWindow = mainWindow;

        area = drAreaComp.CreateGraphics();
        AreaLoc = drAreaComp.Location;
        this.backColor = drAreaComp.BackColor;
        CircleRadius = 15;
        penWidth = 2F;
        PenColor = Color.DarkBlue;
        circlePen = new Pen(PenColor, penWidth);
    }

    1 reference
    public void DrawCircleAroundLastClick() {
        Point circlePos = new Point(PositionInArea.X - CircleRadius, PositionInArea.Y - CircleRadius);
        Size circleSize = new Size(CircleRadius * 2, CircleRadius * 2);
        this.area.DrawEllipse(circlePen, rect: new Rectangle(circlePos, circleSize));
    }

    1 reference
    public Point GetRelativeLocationOfLastClick() {
        int x = PositionInArea.X + AreaLoc.X - CircleRadius / 3;
        int y = PositionInArea.Y + AreaLoc.Y - CircleRadius / 3;
        return new Point(x, y);
    }

    1 reference
    public void ClearDrawingArea() {
        mainWindow.LabelNodes.removeLabelNodes();
        this.area.Clear(backColor);
    }
```

Razreda `LabelNodes` in `Drawing` inicializiramo v pomožni metodi `initializeOnStart`, ki jo izvedemo z klicom v konstruktorju.

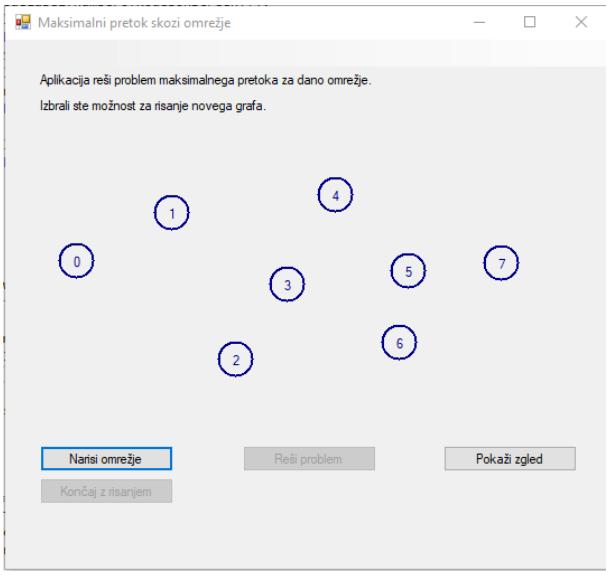
```
/// <summary> Initializes properties when constructor gets called, sets Drawing, ...
1 reference
private void initializeOnStart() {
    appState = AppState.Initialized;
    message = new MessageText();

    // drawing:
    Drawing = new Drawing(mainWindow: this, drAreaComp: DrawingAreaComponent);

    // Nodes:
    LabelNodes = new LabelNodes(mainWindow: this);
```

Sedaj lahko končno napišemo metodo, ki nariše novo vozlišče. Kot smo rekli, bomo v resnici dejansko risali le kroge (in povezave), napisi pa bodo vsi implementirani kot gradniki tipa `Label`. Prazno metodo v razredu `LabelNodes` dopolnimo na naslednji način.

```
1 reference
public void addNewNodeLabel() {
    if (this.NumberOnScreen >= this.MaxNumber)
        return;
    // gets executed only when the number of nodes of screen is smaller than the max number of nodes

    // Add label:
    Label newLabel = new Label();

    newLabel.Location = drawing.GetRelativeLocationOfLastClick();
    newLabel.Name = "label_" + NumberOnScreen.ToString();
    newLabel.Size = new Size(drawing.CircleRadius, drawing.CircleRadius); // size should be as big as the ci
    newLabel.ForeColor = drawing.PenColor;
    newLabel.Text = NumberOnScreen.ToString();

    mainWindow.Controls.Add(newLabel);
    newLabel.BringToFront();

    // subscribe label to mouse events (for connections):
    newLabel.MouseDown += new MouseEventHandler(mainWindow.labelNode_MouseDown);
    newLabel.MouseUp += new MouseEventHandler(mainWindow.labelNode_MouseUp);


    // Add label to array:
    AddLabelNodeInArray(newLabel);

    // Draw circle around the label:
    drawing.DrawCircleAroundLastClick();
}
```

Metoda doda nov napis z napisom `i` in imenom `label_i`, pri čemer `i` predstavlja trenutno število vozlišč na risalni površini (da so vozlišča ustrezno oštevilčena). Doda ga na mesto uporabnikovega klika. Pomembno je, da dodamo klic metode `gradnik.BringToFront`, ki priredi zaporedje risanja gradnikov tako, da tega postavi na vrh in s tem poskrbi, da gradnik sploh vidimo.

Če poženemo aplikacijo, stisnemo *"Nariši omrežje"*, v oknu vnesemo željeno število vozlišč in klikamo na risalno površino, se nam dodajo vozlišča. Delovanje je prikazano na priloženi sliki.



- ## Zaznavanje “vlečenja” povezav

Implementacija zaznavanja ali uporabnik še vedno vleče od enega vozlišča do drugega je zelo preprosta. Idejo smo že razložili: ko uporabnik klikne na risalno površino, nastavimo logično vrednost, ki nam pove, ali je vlečenja že konec na `True`, ko spusti miško, pa na `False`.

Tudi za povezavo bomo dodali svoj razred, najprej pa ga dodajmo za implementacijo strukture, ki shrani tri informacije:

- `Point StartLocation`: kje se je vlečenje začelo (tam je prvo vozlišče),
- `Point EndLocation`: kje se je končalo (tam je drugo vozlišče),
- `bool IsActive`: nam pove, če je vlečenje aktivno ali ne.

```
namespace MaxFlowVisualization_Winforms
{
    2 references
    class Drag {
        1 reference
        public Point StartLocation { get; set; }
        1 reference
        public Point EndLocation { get; set; }
        3 references
        public bool IsActive { get; set; }
    }
}
```

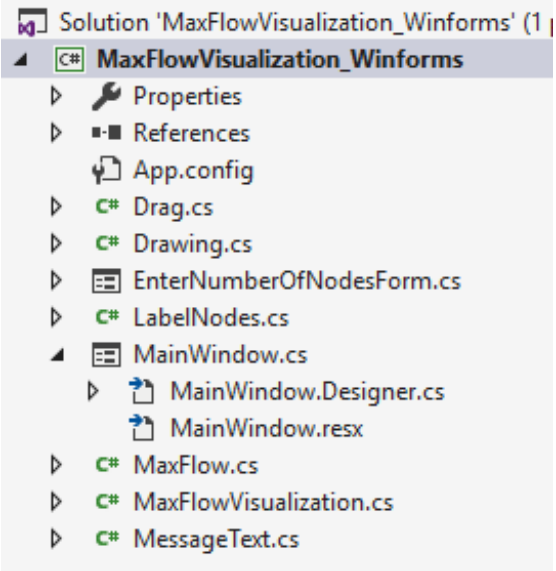
Dodajanje novega vozlišča smo napisali tako, da je vozlišče (napis) naročeno na dva dogodka.

```
// subscribe label to mouse events (for connections):
newLabel.MouseDown += new MouseEventHandler(mainWindow.labelNode_MouseDown);
newLabel.MouseUp += new MouseEventHandler(mainWindow.labelNode_MouseUp);
```

Pri zaznavanju klikov je zaenkrat pomembno samo to, da shranimo lokaciji klikov in beležimo, ali je vlečenje aktivno, ker bomo to informacijo potrebovali za dodajanje povezav.

```
1 reference
public void labelNode_MouseDown(object sender, MouseEventArgs e) {
    drag.StartLocation = e.Location;
    drag.isActive = true;
}

1 reference
public void labelNode_MouseUp(object sender, MouseEventArgs e) {
    drag.isActive = false;
    drag.EndLocation = e.Location;
}
```

Struktura bi do sedaj morala izgledati tako:



- ## Dodajanje povezav

hudlks

- Brisanje vozlišč in povezav

Reševanje danega omrežja z pomočjo algoritma

Prikazovanje primera omrežja

3. Vizualne izboljšave

Sprotna sporočila

Izgled okna

Animacije

4. Zaključek - kaj sem se naučila in težave

5. Viri

koda za ford fulkersonov algoritem!