

TDK-dolgozat

Bc. László Szoboszlai

Univerzita J. Selyeho – Selye János Egyetem
Fakulta ekonómie a informatiky – Gazdaságtudományi és informatikai kar
Katedra informatiky – Informatikai tanszék

TDK-dolgozat

A dolgozat címe: Maximális folyamkereső algoritmusok vizualizációja

Témavezető: RNDr. Štefan Gubo, PhD.

Beosztás: Adjunktus

Szerző: Bc. László Szoboszlai

Tanulmányi szak: Alkalmazott informatika

Évfolyam: Mesterképzés 1. évfolyam

Nappali képzés

Komárno 2024

Absztrakt

Bc. Szoboszlai László: Maximális folyamkereső algoritmusok vizualizációja

[Felvidéki tudományos diákköri munka] Univerzita J. Selyeho. Gazdaságtudományi és informatikai kar, Informatikai tanszék

Témavezető: RNDr. Štefan Gubo, PhD., Komárom, 2024

A felvidéki tudományos diákköri munka olyan szoftver létrehozásával foglalkozik, amely maximális folyam problémát megvalósító algoritmusokat vizualizál. A program egy felületet biztosít, amin a felhasználó képes hálózati gráfokat létrehozni. Az elkészített gráfokat fájlba lehet menteni, hogy később újra be lehessen őket olvasni. A felületen elkészített gráfokon maximális folyam szimulációt lehet futtatni Ford-Fulkerson, Edmonds-Karp és Boykov-Kolmogorov algoritmusokkal. Ezt a felhasználó megteheti egyszerre, vagy lépésenként is. A felhasználó a szoftver használatának segítségével megismerkedhet az imént említett algoritmusok működésével, vagy kiszámolhatja vele a gráfok maximális folyamának értékét illusztrációval.

Kulcsszavak: maximális folyam, szoftver, hálózati gráf gráfelmélet, gráfalgoritmusok, hálózati gráf, maximális folyam, vizualizáció, Ford-Fulkerson, Edmonds-Karp

Abstract

Bc. László Szoboszlai: Visualization of maximum flow search algorithms

[Scientific student work] Univerzita J. Selyeho. Faculty of Economics and Informatics,
Department of Informatics

Supervisor: RNDr. Štefan Gubo, PhD., Komárno, 2024

The purpose of this scientific student work is to create a software for visualization maximum flow algorithms. The program provides an interface for the user to create flow networks. The prepared graphs can be saved to the disc as files for later use. The saved graphs can be loaded. The software can run maximum flow simulations on the graphs using Ford-Fulkerson, Edmonds-Karp and Boykov-Kolmogorov algorithms. The user can run the entire simulation at once, or step by step. The user can learn how these algorithms work by using this software, or the user can use the software to calculate maximum flow with illustrations.

Keywords: graph theory, graph algorithms, flow network, maximum flow, visualization, Ford-Fulkerson, Edmonds-Karp

Tartalomjegyzék

Bevezetés.....	6
1 A maximális folyam probléma	7
1.1 Hálózati gráf.....	7
1.2 Maximális folyam problémát megvalósító algoritmusok.....	7
2 A szoftver megvalósítása.....	9
2.1 Tárolás	9
2.2 Megjelenítés	10
2.3 A Node osztály a csúcspontok kezeléséhez.....	10
2.4 Az SPLine osztály a kapcsolatok kezeléséhez	12
2.5 A Form előfeltételei a Graph osztály működéséhez.....	17
2.6 A Graph osztály	18
2.7 A Graph osztály szimulációért felelős metódusai	23
2.8 A Graph osztály eseménykezelői	28
2.9 A Form megvalósítása.....	30
2.10 Az InputForm osztály.....	33
Befejezés	35
Resumé.....	36
Idézett forrásmunkák.....	37
Ábrajegyzék	39

Bevezetés

Ez a munka egy az ismertebb maximális folyam problémával foglalkozó algoritmusok működésének vizualizációjára szánt szoftver megvalósítás létrehozásához nyújt útmutatást. A munka alapján elkészített szoftver megvalósítás lehetőséget nyújt hálózati gráfok létrehozására, mentésére, módosítására, és ezeken a gráfokon történő szimulációra. A szimulációk segítségével megérthetjük a maximális folyam algoritmusok működését, vagy komplexebb gráfok esetén kiszámíthatjuk a maximális folyam értékét. A munka két fő fejezetet tartalmaz, amelyek alfejezetekre bonthatóak.

Az első ilyen fő fejezet egyfajta elméleti rész, amely a gráfokkal és a maximális folyam problémával kapcsolatban mélyíti el ismereteinket. Két alfejezete van. Az első a hálózati gráfok fogalmát és a maximális folyam problémát ismerteti. A második pedig a maximális folyam problémával foglalkozó algoritmusokat tárgyalja.

A második fő fejezet az szoftver megvalósításáról szól. Tíz alfejezettel rendelkezik. Az első alfejezet a gráf memóriában és lemezen való tárolásáról szól. A második alfejezet a megjelenítés megvalósításáról és az azt akadályozó problémák megkerülését tárgyalja. Szó esik a főbb megjelenítésért felelős típusokról és osztályokról. A harmadik fejezet a *Node* osztály megvalósítását írja le, ezt az osztályt a csúcsponatok megjelenítésére használtuk. Leírja az osztály működését és felépítését. A negyedik fejezet az *SPLine* osztály létrehozását tárgyalja, ami a kapcsolatok reprezentálására szolgál. Kiemeli a lényegesebb eljárásokat és képleteket. Leírja a koordinátáinak meghatározásakor felmerülő problémákat, és az ezeknél használt képleteket ábrákkal szemlélteti. Az ötödik leírja, hogy milyen előfeltételeknek kell teljesülniük a *Form* felületen, hogy a *Graph* osztály implementálható legyen. A hatodik fejezet a *Graph* osztály implementálását írja le. A hetedik fejezet a szimulációért felelős metódusokat tartalmazza, közéjük tartozik a Ford-Fulkerson és az Edmonds-Karp algoritmusok, és ezek megjelenítéséért felelős metódusok megvalósítása is. A nyolcadik a vezérléshez szükséges eseménykezelők megvalósítását tartalmazza. A kilencedik a *Form1* osztály megvalósítást fejt ki. A tizedik fejezet pedig a bemenet bekéréséért felelős *InputForm* osztály leírását tartalmazza.

1 A maximális folyam probléma

A maximális folyam probléma célja, megkeresni azt a legnagyobb mennyiséget, ami át tud folyni az adott hálózati gráfon. Erre a problémára több megközelítése is létezik. A számítástechnikai oldalról is már több a problémát megvalósító algoritmust alkottak az idők során. Többek között ilyenek például a Ford-Fulkerson és az Edmonds-Karp algoritmusok is.

1.1 Hálózati gráf

Egy hálózati gráf két elemből épülhet fel. Csúcspontokból, melyeket angolul *node*-oknak nevezünk, és kapcsolatokból, melyeket angolul *link*-nek, vagy *arc*-nak nevezünk. A kapcsolatok a gráf élei, melyeket angolul *edge*-nek nevezünk. A csúcspontok egyfajta elágazásként működnek a kapcsolatok számára, korlátlan mennyiségű kapcsolatot lehet velük asszociálni. Meghatározott irányuk és kapacitásuk van. A kapcsolatokon csak egy irányban haladhat a folyam, tehát egy irányított gráfról beszélünk. Egy hálózat gráfban két lényeges csúcspontot tudunk kiemelni, a forrás és a nyelő csúcspontokat. Ezek a csúcspontok azért különlegesek, mert meghatározzák a folyam kiindulási, és végpontját. Mindkét csúcspontra jellemző, hogy míg a kiindulási csúcspont, azaz a forrás korlátlan mennyiségű folyamat tud beengedni a hálózatba, addig a nyelő szintén korlátlan mennyiségű folyamat tud kiengedni a hálózatból. A hálózat csúcspontjait jellemzi a megmaradási szabály, amit a Széchenyi Egyetem oktatója Hajba Tamás a következőképpen fogalmazott meg: „A gráf pontjai a vezetékek elágazási pontjai, amelyekben, ha nem a termelőről és a fogyasztóról van szó, víz nem gyűlhet össze. Vagyis a befolyó vízmennyiségnek egy pontban meg kell egyeznie a kifolyó vízmennyiséggel. A hálózati gráf ezen tulajdonságát megmaradási szabálynak nevezzük.” (Hajba, 2006)

1.2 Maximális folyam problémát megvalósító algoritmusok

Mielőtt konkrét algoritmusokról beszélnénk át kell vennünk pár lényeges fogalmat a témával kapcsolatban. Ilyen például a *Residual Graph*, avagy fennmaradó gráf. Ez egy olyan gráfot reprezentál, ami a pillanatnyilag fennmaradó élkapacitásokat tartalmazza. Az *Augmenting Paths*, vagy ahogy mi a megvalósításban neveztük *Correction Paths*, magyarul javító utaknak nevezzük azokat az utakat a gráfban, amelyek a forrástól a nyelőig haladnak, és még van fennmaradó kapacitásuk. Ebben a megvalósításban három maximális folyam problémát megvalósító algoritmust tárgyalunk, a Ford-Fulkerson, az Edmonds-Karp és a Boykov-Kolmogorov algoritmust. Mindhárom algoritmus a *Residual Graph* bejárásával keres javító utakat, majd találat esetén eltávolítja a talált kapacitást az említett gráfból. A főbb különbség az algoritmusok között a keresés módja. Az Edmonds-Karp algoritmust Jack R. Edmonds és Richard M. Karp publikálta 1972-ben. Komplexitása $O(V \cdot E^2)$, ahol V a csúcsok és E az élek

száma. Az Edmonds-Karp BFS-t, azaz Breadth First Search-öt (szélességi keresés) (GeeksforGeeks, 2024) használ a javító utak megtalálására. A BFS hullámonként, avagy rétegenként halad, ez azt jelenti, hogy mindig a vizsgált csúcs közvetlen kapcsolatait vizsgálja, aztán azoknak a közvetlen kapcsolatait és így tovább. A BFS a felfedezésük sorrendjében vizsgálja a csúcspontokat. Az adott csúcs összes szomszédját bejárja, majd azok szomszédjait, és így tovább, amíg nem talál javítótutat, vagy el nem fogynak a csúcsok. Mi a megvalósítás során egy sor (Queue) adatstruktúrát alkalmaztunk ezen tulajdonság implementálására. A Ford-Fulkerson algoritmust 1956-ban publikálta Lester Randolph Ford Junior és Delbert Ray Fulkerson. Az algoritmus komplexitása $O(f \cdot E)$ ahol f -a maximális folyam, és E -a gráf éleinek száma. A Ford-Fulkerson az Edmonds-Karpal szemben DFS-t, azaz Depth First Search-öt (mélységi keresés) (GeeksforGeeks, 2024) használ. A DFS és a BFS között az a különbség, hogy a DFS nem a felfedezésük sorrendjében vizsgálja a csúcspontokat, hanem mindig a vizsgált csúcspont első közvetlen szomszédját vizsgálja, és így halad mélyebbre a gráfban egészen addig, amíg nem találja meg a nyelőt, vagy el nem fogy a kapcsolat, amelyen haladhat. Az utóbbi esetben a soron következő csúccsal próbálkozik, azaz visszalép. Ennek megvalósítására verem (Stack) adatstruktúrát használtunk. A Boykov-Kolmogorov algoritmus 2004-ben jelent meg (An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision, 2004). Alkotói Yuri Boykov és Vladimir Kolmogorov. Az algoritmus feltételezett komplexitása megközelítőleg $O(E)$, ahol E -a gráf éleinek száma, de ez nem bizonyított érték. A korábban említett algoritmusok a forrás csúcspontból kiindulva építenek ki keresési fát minden alkalommal amikor egy javítótutat találnak. Ezzel szemben a Boykov-Kolmogorov megvalósítás két ilyen fát épít ki mind a forrás és a nyelő csomópontokból is. A BFS és DFS keresésekkel ellentétben ez az algoritmus megőrzi az általa összeállított keresési fákat egy javítótút felfedezése után is, majd a mentett értékeken folytatja a keresést. Minden olyan csúcst, amelynek van olyan szomszédja, ami nem tartozik egyik keresési fához sem aktív csúcsként tart nyilván. A következő iteráció során az aktív csúcsok összes szabad szomszédja az adott keresési fához kerül úgy, hogy az adott aktív csúcs lesz a mostantól a fához tartozó csúcs szülője. Az iteráció végén az aktív csúcsok átadják aktív státuszukat gyermekeiknek, majd ők maguk passzívvá válnak. Amennyiben a két keresési fából egy-egy csúcs szomszédságba kerül, akkor javítótutat találtunk. A rá következő javítótút keresésekor a telített élek eltávolítása után árvává vált csúcsokat vagy egy közvetlen szomszédságban lévő fához csatoljuk, vagy amennyiben nincs ilyen, szabad csúccsá tesszük.

2 A szoftver megvalósítása

A szoftver létrehozásához a következő programozási környezetet és segédprogramokat használtuk:

Programozási környezet: Visual Studio 2022 Community Edition, verzió: 17.10.0 Preview 2.0

A használt dotnet keretrendszer: *NETFramework 4.8*

A használt programozási nyelv és verziója: *C# 7.3*

A súgóhoz használt fájl létrehozására a *HelpNDoc 9* programot használtuk. A megvalósítást a dotnet által nyújtott *Windows Forms App (NET Framework)* sablonból kiindulva alkottuk meg.

2.1 Tárolás

Mielőtt egy gráfon bármely műveletet el tudnánk végezni, vagy csupán meg tudnánk azt jeleníteni, valamilyen formában tárolnunk kell a gráfot a memóriában. A gráfot csúcsmátrix formájában tároltuk, erre a legalkalmasabb típus a *List<List<int>>* típusú integer mátrix volt, mivel a program futása közben a gráf gyakran változik. Az *int[,]* ugyan teljesítmény szempontjából gyorsabb, mint egy *List* mátrix, de mérete, azaz sorainak és oszlopainak száma allokálásától fix, tehát később nem módosítható. Továbbá nem jelölhetjük meg *readonly* (csak olvasható) típusként mivel ez megakadályozza az újra allokálás lehetőségét, viszont a *List* típusú mátrixoknál ezt megtehetjük. A csúcsmátrixban a sor száma határozza meg a kapcsolat kiindulási csúcspontját, az oszlop pedig a célt. A kettő által meghatározott cellában pedig a kapcsolat maximális átbocsájtó képességének értéke található. A megjelenítéshez a csúcspontok elnevezését és koordináta-rendszerbeli pozícióját is tárolnunk kell. A létrehozott gráfokat a lemezre is mentenünk kellett, hogy később újra betölthessük őket. Ebben az esetben viszont a memóriatakarékosság az elsődleges, mivel a gráfokat ritkán kell betölteni és menteni, így a teljesítmény itt nem szempont, ezért erre a célra a legalkalmasabb tárolási forma egy módosított éllista volt.

A	C
41	15 5
A 26	66 1-4 3-1
B 220	73 0-3 3-1 4-6
C 559	130
D 143	222 4-2
E 368	101 1-1 2-2 3-1

1. ábra - Gráf tárolása fájlban

Az első sor tartalmazza a forrás és a nyelő csúcspontok nevét. A második a csúcspont- és a betűméretet, valamint a *TriangleO* változó értékét, ami a kapcsolatok által tartalmazott nyíl háromszögének méretét határozza meg. Az ezt követő sorokban pedig található az éllista szóközőkkel tagolva, ebben a sorrendben: a csúcspont neve, koordinátái (x|y), kapcsolódó csúcspont száma - a kapcsolat maximális átbocsájtó képessége, ... A fájlba mentés során a Windowsba beépített *deflate* tömörítést használtuk, és saját „.gbin” kiterjesztéssel mentettük el a fájlt.

2.2 Megjelenítés

A megjelenítéshez szükségünk volt egy felületre, amire ki tudtuk rajzolni a csúcspontokat és kapcsolatokat. erre a célra a *System.Windows.Controls.Canvas* osztályt használtuk, amiben meg tudtunk jeleníteni *System.Windows.Shapes* és *System.Windows.Controls* béli alakzatokat és elemeket egyaránt. Ezt egy *System.Windows.Forms.Form* ablakon belül jelenítettük meg. Problémát okozott, hogy egy *Form* objektumban nem tudunk megjeleníteni egy *Canvas* objektumot, mivel két különböző névtérről beszélünk, sőt a *Canvas* egy WPF (Windows Presentation Foundation) objektum. Mivel egy Windows forms projekttel dolgoztunk, ezért a *Form*-hoz csak *System.Windows.Forms.Control* típusú objektumot lehetett hozzáadni. Ennek megkerülésére beiktattunk egy köztes elemet, egy úgynevezett *elementhost* objektumot, ami a *WindowsFormsIntegration* elnevezésű assembly referencia projekthez való hozzáadása után érhető el. Az *elementhost* objektum WPF objektumok becsatolására alkalmazható, így az *elementhost* hozzáadható a *Form*-hoz, és a *Canvas* pedig becsatolható az *elementhost*-ba. További probléma, hogy mivel a *Canvas* egy WPF elem, ezért befolyásolja koordináta értékeit a Windows DPI skálázás, ezért ezt az értéket figyelembe kell venni a számításoknál. Most már, hogy elérhetővé vált számunkra a felület, létre kell hoznunk a grafikus objektumokat kezelő osztályainkat, amelyek meg fognak jelenni a *Canvas*-on. Két ilyen osztályt hoztunk létre: a *Node*-ot, ami a csúcspontokat kezeli, és a *SPLine*-t, ami a kapcsolatokat. Ezeket egy *Graph* elnevezésű osztály használja fel, ami az egész gráf megjelenítését, gráfok mentését és betöltését, a szimulációk futtatását és a *Form*-al történő kommunikációt kezeli.

2.3 A Node osztály a csúcspontok kezeléséhez

Feladata a csúcspontok kirajzolása, amelyek egy ellipszisből és egy szövegdobozból állnak, a szövegdoboz az ellipszis közepén helyezkedik el, beleszámolva a betűméretet is a pozíciójába. Ellipszisként egy *System.Windows.Shapes.Ellipse* osztályból származtatott objektumot használtunk. A szövegdobozhoz pedig egy *System.Windows.Controls.TextBlock* objektumot. Az osztály konstruktorában meg kell adni a *Canvas*-t amire rajzolunk, és a csúcspont nevét

reprezentáló *stringet*, továbbá opcionális paraméterként megadható az ellipszis alapértelmezett színét is. Ez alap esetben piros, azaz *Brushes.Red*. A konstruktor ezeken kívül elmenti a globálisan meghatározott betű és az ellipszisre vonatkozó méreteket, melyek a későbbiekben már nem módosíthatóak. A meghatározott méretek alapján létrehozza az *Ellipse* objektumot és a szövegdobozt, majd ezeket hozzáadja a *Canvas*-hoz. A globális méretek a *Graph* osztályban statikusan vannak meghatározva.

```
public Node(Canvas canvas, string label, Brush defColor = null)
{
    size = Graph.NodeSize;
    fontsize = Graph.FontSize;
    Label.Text = label;
    this.canvas = canvas;
    if (defColor != null) {
        this.defColor = defColor;
    }
    Ellipse = new Ellipse {
        Width = size,
        Height = size,
        Fill = this.defColor
    };
    Label.FontSize = fontsize;
    canvas.Children.Add(Ellipse);
    canvas.Children.Add(this.Label);
}
```

1. kódrészlet - A *Node* osztály konstruktora

Az osztály az általa kezelt grafikus objektumokra gyakorolt jobb egérgomb felengedésekor keletkező eseményeket is kezeli. *System.Windows.Input.MouseButtonEventHandler* típussal dolgozik, amelynek módosításakor amennyiben volt korábban függvény hozzárendelve, eltávolítja azt a grafikus objektumokról, és hozzájuk fűzi az újat.

```
private MouseButtonEventHandler eventHandler;
public MouseButtonEventHandler EventHandler
{
    set {
        if (value != null) {
            if (eventHandler != null) {
                Ellipse.MouseRightButtonUp -= eventHandler;
                Label.MouseRightButtonUp -= eventHandler;
            }
            eventHandler = value;
            Ellipse.MouseRightButtonUp += eventHandler;
            Label.MouseRightButtonUp += eventHandler;
        }
    }
}
```

2. kódrészlet - *EventHandler* kezelése osztályok szintjén

Az osztályban a következő függvény megvalósításokat hoztuk létre:

A *ChangeLabel* függvény segítségével módosítható a csúcspont elnevezése. A szövegdoboz új pozíciója a következő képlet alapján van meghatározva.

1. egyenlet - A csúcspont címének *X* koordinátája

$$\text{Position.X} + \text{size} / 2 - (\text{fontsize} / 2.75) * \text{Label.Text.Length}$$

Ahol a *Position System.Windows.Point* típusú változó, ami a csúcspont jelenlegi koordinátáit tartalmazza, a *Label* a szövegdoboz, a *size* a csúcspont mérete, és a *fontsize* pedig a betűméret. A *GetLabel* függvénnyel lekérhető a csúcspont jelenlegi elnevezése. Szükség volt a csúcspontok pozíciójának módosítására, ezt a *SetPosition(double x, double y)* függvény bevezetésével értük el. A függvény nem csak a csúcspont *Canvas*-on lévő pozícióját módosítja, hanem kiszámolja a korábban említett egyenlet (lásd 1. egyenlet) segítségével a szövegdoboz X és Y koordinátáit is a megadott új pozícióhoz viszonyítva. Az Y koordinátát a következő képlet segítségével határozza meg.

2. egyenlet - A csúcspont címének Y koordinátája

$$y + size / 2 - fontsize / 2$$

Ahol az *y* a *SetPosition* függvényben megadott *y* változó. Az *IsInside(Point p)* függvény, ami egy *boolean* változót ad vissza meghatározza Pitagorasz tétel segítségével, hogy a megadott *p* pont a csúcsponton belül található-e. Az ellipszis színének megváltoztatására a *SetColor(Brush newcolor)* függvény használható. Ez nem módosítja az alapértelmezett színt (*defColor*), csak az ellipszist (*Ellipse.Fill*). A *ResetColor()* paranccsal visszaállítható az alapértelmezett értékre.

2.4 Az SPLine osztály a kapcsolatok kezeléséhez

Az osztályt a kapcsolat *Canvas*-on történő reprezentálására hoztuk létre. Ennek megvalósítására egy Bézier-görbét, egy háromszöget és egy szövegdobozt használtunk. Az osztályból létrehozott objektum eltárolja a maximális és jelenlegi áteresztő képességet, és egy-egy a forrás és cél csúcspontokra mutató hivatkozást is. Ezek alapján lehet azonosítani a kapcsolatot. A háromszög egy *System.Windows.Shapes.Polygon* típusú objektum. A szövegdoboz a már korábban használt *System.Windows.Controls.TextBlock* típus, mely enyhén szürkített háttérrel rendelkezik. Itt ARGB színmodellt alkalmaztunk, hogy enyhén átlátszóvá tessük a szövegdoboz hátterét, így amennyiben fed egy másik grafikus objektumot az nem okozhat problémát. Maga a görbe, amit alkalmazunk, az egy *System.Windows.Shapes.Path* típus, ami a *System.Windows.Media* névtérben található *PathGeometry*, *PathFigure*, *PathSegmentCollection*, *PointCollection* és *PolyBezierSegment* típusok kombinációjából épül fel. Ennek működéséhez a *GenControlPoints(Point[] points)* függvény számolja ki a szükséges koordinátákat, amelyek meghatározzák a Bézier-görbe éleit. A függvény és a görbe megvalósításának alapjai a (Rocky Mountain Computer Consulting, 2023) (user1228, 2012) forrásokon alapulnak. A görbe ívének méretét a *tension* változó határozza meg, amelynek mértéke függ a kezdő és végpont távolságától. A szoftver korábbi verzióiban görbékkel kötöttük össze a csúcspontokat, viszont ezeket az átláthatóság kedvéért egyenesekre cseréltük. Ezt úgy

értük el, hogy a görbéknek csak két koordinátát adtunk meg 3 vagy több helyett. A görbék íveit és koordinátáit meghatározó kódrészletek és függvények a szoftverben maradtak esztétikai és tovább fejleszthetőségi szempontokból.

```
private Point[] GenControlPoints(Point[] points)
{
    double distance = Math.Sqrt(Math.Pow(points[0].X -
        points.Last().X, 2) + Math.Pow(points[0].Y -
        points.Last().Y, 2));

    double tension = 0.1 + 0.9 * (1 / (1 + Math.Pow(Math.E, 0.02
        * (distance - 60))));

    double c_scale = tension / 0.5 * 0.175;
    List<Point> pts = new List<Point> {
        points[0]
    };

    for (int i = 0; i < points.Length - 1; i++) {
        Point pt = points[i];
        Point pt_before = points[Math.Max(i - 1, 0)];
        Point pt_after = points[i + 1];
        Point pt_after2 = points[Math.Min(i + 2, points.Length -
            1)];
        pts.Add(new Point(pt.X + c_scale * (pt_after.X -
            pt_before.X), pt.Y + c_scale * (pt_after.Y -
            pt_before.Y)));
        pts.Add(new Point(pt_after.X - c_scale * (pt_after2.X -
            pt.X), pt_after.Y - c_scale * (pt_after2.Y - pt.Y)));
        pts.Add(pt_after);
    }
    return pts.ToArray();
}
```

3. kódrészlet - *GenControlPoints* függvény

Az osztály szintén el van látva egy *EventHandler* tulajdonsággal (lásd 2. kódrészlet) ami a görbére gyakorolt jobb klikk eseményt kezeli. A végpontok koordinátáit a *Points* tulajdonságon keresztül lehet meghatározni, ami a koordináta tömb módosítása után újra rajzolja a görbét és kiszámolja a többi szükséges koordinátát a *GenControlPoints* (lásd 3.kódrészlet) segítségével.

```
private Point[] _points = new Point[2] {
    new Point(0, 0), new Point(0, 0) };

public Point[] Points
{
    get { return _points; }
    set {
        _points = value;
        Redraw();
    }
}
```

4. kódrészlet - *Points* tulajdonság megvalósítása

Ugyan a görbe koordinátái ezzel a megoldással történő megvalósításával elérhetőek külső osztályokból is, a program futtatása során ezt a lehetőséget nem használjuk ki. Ugyanis erre a

célra létrehoztunk egy alternatív megvalósítást az osztályon belül, ami automatizálja a koordináta meghatározást a két végpontban lévő csúcspontok pozíciója alapján. Erre szolgál a *Snap(List<List<int>> graphm, List<Node> nodes)* függvény, melynek bemenete maga a gráf mátrix, ami a *graphm* változónevet fogja felvenni, és az összes a gráfban szereplő *Node* objektumot tartalmazó tömb, ami *nodes*ként fog megjelenni.

```
public void Snap(List<List<int>> graphm, List<Node> nodes)
{
    label.Text = used_stream_size.ToString() + " / " + stream_size.ToString();
    double nodesizehalf = Graph.NodeSize / 2.0;
    List<Point> newarray = new List<Point>();
    Point beginning = new Point(FromNode.Position.X + nodesizehalf, FromNode.Position.Y + nodesizehalf);
    Point ending = new Point(ToNode.Position.X + nodesizehalf, ToNode.Position.Y + nodesizehalf);

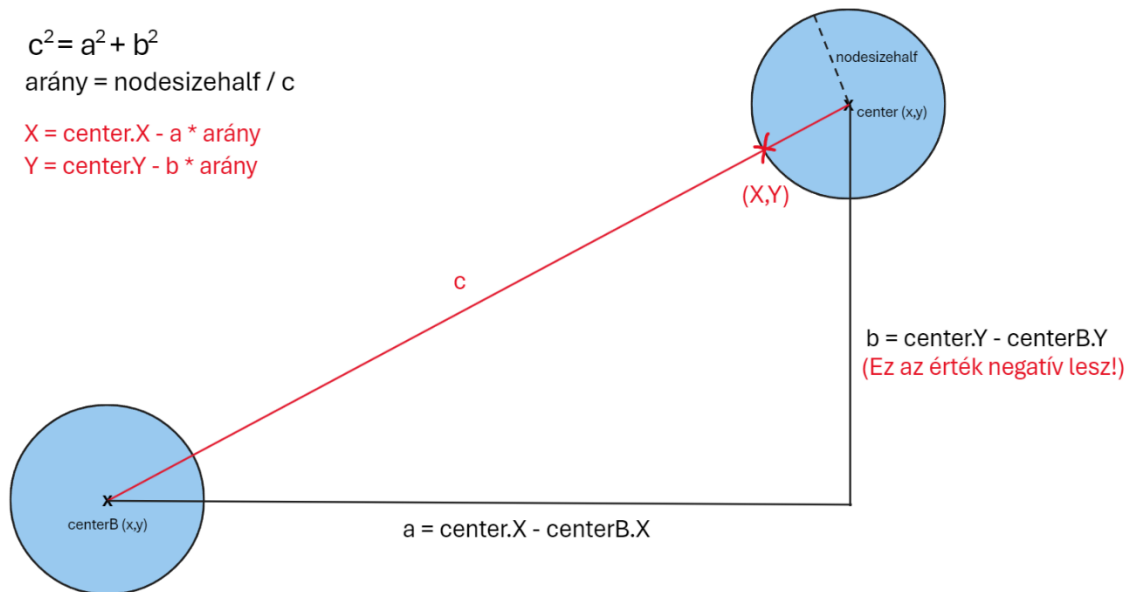
    double alpha;
    int i = nodes.IndexOf(FromNode), j = nodes.IndexOf(ToNode);
    if (graphm[i][j] != 0 && graphm[j][i] != 0) {
        Point aux_point = Rotate2D(beginning, ending, 45);
        double a = aux_point.Y - beginning.Y;
        double b = aux_point.X - beginning.X;
        double c = Math.Sqrt(Math.Pow(a, 2) + Math.Pow(b, 2));
        double ratio = nodesizehalf / c;
        newarray.Add(new Point(beginning.X + b * ratio, beginning.Y + a * ratio));
        newarray.Add(Rotate2D(ending, new Point(ending.X - b * ratio - b * (Triangle0 / c), ending.Y - a * ratio - a * (Triangle0 / c)), -90));
        labelCoordinates.X = newarray[0].X + (newarray[1].X - newarray[0].X) / 2 - (label.Text.Length / 3) * Graph.FontSize;
        labelCoordinates.Y = newarray[0].Y + (newarray[1].Y - newarray[0].Y) / 2;
        alpha = Math.Asin((ending.Y - beginning.Y) / c) / (Math.PI / 180) * (beginning.X > ending.X ? -1 : 1) + (beginning.X > ending.X ? 180 : 0);
    }
    else {
        double a = ending.Y - beginning.Y;
        double b = ending.X - beginning.X;
        double c = Math.Sqrt(Math.Pow(a, 2) + Math.Pow(b, 2));
        double ratio = nodesizehalf / c;
        newarray.Add(new Point(beginning.X + b * ratio, beginning.Y + a * ratio));
        newarray.Add(new Point(ending.X - b * ratio - b * (Triangle0 / c), ending.Y - a * ratio - a * (Triangle0 / c)));
        labelCoordinates.X = beginning.X + (ending.X - beginning.X) / 2 - (label.Text.Length / 3) * Graph.FontSize;
        labelCoordinates.Y = beginning.Y + (ending.Y - beginning.Y) / 2 + Triangle0;
        alpha = Math.Asin(a / c) / (Math.PI / 180) * (beginning.X > ending.X ? -1 : 1) + (beginning.X > ending.X ? 180 : 0);
    }
    trianglepoints = Arrow(newarray.Last(), alpha);

    Points = newarray.ToArray();
}
```

5. kódrészlet - Snap függvény

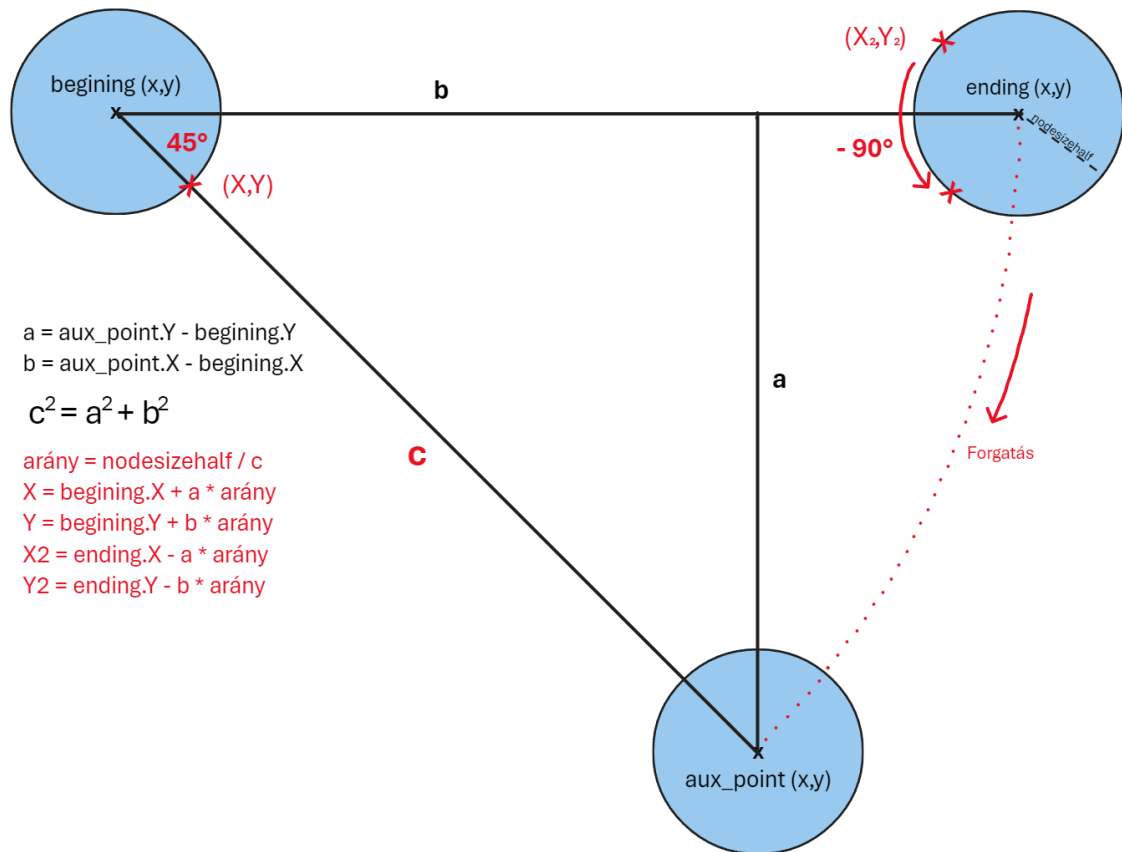
A függvény először beállítja a szövegdoboz tartalmát, ha esetleg időközben módosult, vagy nem volt meghatározva eddig, hiszen ennek koordinátáit is ez az eljárás határozza meg, ehhez pedig tudni kell a szövegdoboz pontos méretét, avagy ez esetben az általa tartalmazott karakterek számát. A függvénynek ezen kívül még szüksége van a csúcspontok méretének felére, ezt tartalmazza a *nodesizehalf* változó. Erre azért van szükség, mert a *beginning* és az *ending* változóknál tárolt végpontok koordinátáinál nem az adott csúcspont kezdő pontjaira

(ami a bal felső sarok) van szükség, hanem a csúcspont középpontjára, ezért ezekhez hozzá kellett adnunk ezt az értéket. Az *alpha* változó a kapcsolat végén található háromszög, avagy nyílhegy szögét határozza meg. Az eljárás megállapítja a gráf mátrix alapján, hogy oda vissza is van e kapcsolat a két csúcspont között, ha nincs kiszámolja a csúcspontok peremén lévő a másik csúcspontokhoz legközelebbi koordinátát. Ha a két végpontot összekötjük egy egyenessel, akkor a végpont körül elhelyezkedő kör, melynek sugara a *nodesizehalf*, és az egyenes kimetszi ezt a pontot. Ez a Pitagorasz tétel alapján kiszámolt arány segítségével meghatározható.



2. ábra - Kapcsolat végpontkoordinátáinak képlete

A kiszámolt koordinátákat még módosítanunk kellett a *TriangleO* tulajdonság mértékével, hogy a végére férjen a háromszög. Ha két csúcspont között oda és vissza is van kapcsolat, akkor a két kapcsolatot kissé eltoljuk egymástól, egyiket lejjebb, a másikat feljebb. Ezt a következőképpen valósítottuk meg: Ismert két pont, a két csúcspont középpontja, ezek *beginning* és *ending*ként vannak jelölve a kódban. Felvettünk egy harmadik pontot, ezt nevezzük *aux_point*-nak (mint auxiliary point, azaz segéd pont), amit úgy határoztunk meg, hogy a végpontot (*ending*) elforgattuk 45 fokkal a kezdőpontot (*beginning*) használva, mint origót, azaz középpontot. Az új ponttól számoltuk ki az arányokat a kapcsolat kezdő és végpontjainak meghatározásánál (lásd 2.ábra, csak ebben az esetben a centerB az *aux_point*). A végpontot még meg kellett fordítanunk a csúcspont körül -90 fokkal mivel a számítások alapján pont a csúcspont másik oldalára jött ki a koordinátája.



3. ábra - Párhuzamos kapcsolatok végpontkoordinátáinak képlete

A háromszög koordinátáit az *Arrow(Point center, double degree)* eljárással határoztuk meg, melynek bemenete a háromszög, azaz ábrázolás szempontjából a „nyíl” középpontja amit a *center* változóba mentettünk, és az alakzat dőlésszöge fokokban meghatározva. A függvény kimenete egy *System.Windows.Media.PointCollection* típusú változó, ami a háromszög 3 koordinátáját tartalmazza.

```
private PointCollection Arrow(Point center, double degree)
{
    return new PointCollection(new Point[] {
        Rotate2D(center, new Point(center.X - Triangle0,
            center.Y - Triangle0), degree),
        Rotate2D(center, new Point(center.X - Triangle0,
            center.Y + Triangle0), degree),
        Rotate2D(center, new Point(center.X + Triangle0,
            center.Y), degree)
    });
}
```

6. kódrészlet - Arrow függvény

A *Rotate2D(Point origo, Point target, double degree)* függvény síkbeli mátrix transzformációt hajt végre, melynek célja egy meghatározott origó körül elforgatni egy megadott koordinátát a koordinátarendszerben, és visszaadni *Point* típusként az új koordinátát. A művelet elvégzéséhez meg kell adnunk az origót, az adott koordinátát, amit forgatni szeretnénk az origó körül, és a

forgatás mértékét fokokban meghatározva. Ezt követően az eljárás átalakítja a fokot radiánra, amivel el tudja végezni a számításokat a következő képletek alapján (Alpert, 2009):

3. egyenlet - Síkbeli forgatás képletei

$$\begin{aligned} X &= \cos(\theta) * (P.X - O.X) - \sin(\theta) * (P.Y - O.Y) + O.X \\ Y &= \sin(\theta) * (P.X - O.X) + \cos(\theta) * (P.Y - O.Y) + O.Y \end{aligned}$$

Ahol X és Y az új koordináták, amit ki kell számolnunk. A θ a szög, amivel elforgatjuk a megadott pontot radiánban. A P az elforgatandó pont, és az O az origó. A függvény megvalósítása a következő:

```
private static Point Rotate2D(Point origo, Point target, double degree)
{
    double rad = degree * (Math.PI / 180);
    double nX = (target.X - origo.X) * Math.Cos(rad) - (target.Y - origo.Y) * Math.Sin(rad) + origo.X;
    double nY = (target.X - origo.X) * Math.Sin(rad) + (target.Y - origo.Y) * Math.Cos(rad) + origo.Y;
    return new Point(nX, nY);
}
```

7. kódrészlet - Rotate2D függvény

Az osztály ezeken kívül még a következő megvalósításokat tartalmazza: *UsedStreamSize* tulajdonság, melynek segítségével beállítható, hogy a kapcsolat kapacitásából mennyit használ fel a jelen pillanatban. Az *SplineThickness* tulajdonság, amivel a kapcsolat görbéjének vonalvastagsága állítható. A *Draw* függvény, amit azért hoztunk létre, hogy ráhelyezze a *Canvas*-ra a korábban említett grafikus objektumokat. A konstruktorban automatikusan meghívásra kerül. Összerakja a *Path*-ot az elemeiből, felhelyezi az objektumokat a *Canvas*-ra, és még felhelyezi a görbére (*Path*) az eseménykezelőt. A *ReDraw* függvény eltávolítja az objektumokat, majd meghívja a *Draw* eljárást. A *Highlight* függvény kék színűre színezi a görbét és a háromszöget, ezt arra a célra hoztuk létre, hogy szimuláció futtatás közben meg tudjuk különböztetni, hogy melyik kapcsolatokról van szó az adott pillanatban. Hogy a kapcsolat maximális kapacitása később is módosítható legyen, létrehoztuk a *SetStreamSize* függvényt.

2.5 A Form előfeltételei a Graph osztály működéséhez

A *Graph* osztály úgy terveztük, hogy minden a megjelenítéssel, szimulációkkal, tárolással kapcsolatos műveletet ez az osztály hajtson végre. A *Form* csak a megjelenítéshez szükséges eszközöket biztosítja a *Graph*-ból példányosított objektumok számára. Ezek közé tartozik például az *elementhost* amiben a *Canvas* található, és a különböző feliratokat tartalmazó elemek. A *Form*-nak a következő elemeket kell tartalmazni a megadott elnevezéssel ahhoz, hogy a *Graph* osztály objektumai működni tudjanak vele:

- `System.Windows.Forms.Integration.ElementHost` objektum `elementHost1` elnevezéssel. A Canvas megjelenítéséhez.
- 2x `System.Windows.Forms.Label` objektum: `sourcelabel`, `sinklabel` elnevezéssel. A `sourcelabel` a forrás csúcspont, és a `sinklabel` a nyelő vagy cél csúcspont nevének megjelenítéséhez.
- `System.Windows.Forms.TextBox` objektum `CPath_box` elnevezéssel. A javítóutak listázásához.
- 3x `System.Windows.Forms.ContextMenuStrip` objektum helyi menük megjelenítésére `nodeContext`, `splineContext` és `defaContext` néven.

Ezeknek az elemeknek *public*, azaz publikus tulajdonságúnak kell lenniük. A *Form* ezeken kívül még a vezérlés megvalósításához tartalmaz elemeket.

2.6 A Graph osztály

Ez az osztály tartalmazza a gráffal kapcsolatos összes adatot, és a megjelenítéshez és szimulációkhoz szükséges metódusokat is. A *Graph* osztály a memóriába foglalja: a csúcspontok koordinátáit a `node_coordinates` változóban. Erre a merevlemezre történő mentéshez és a gráf újra rajzolásánál van szükség, a csúcspontok önmagukban is tartalmazzák a saját koordinátaikat, ezért erre a változóra a közvetlen működéshez nincs szükség. A csúcspontok elnevezését `nodenames` néven tároltuk. A gráf csúcsmátrixa `graph_matrix`. A *Canvas*-t és a *Form*-ot is meg kell adni a konstruktorban, amelyeket hivatkozásként tároltunk. A csúcspontokat `nodes`, a kapcsolatokat pedig `splines` dinamikus tömbökben tároltuk. A csúcspont- és betűméretet osztályszinten statikus tulajdonságokként tároltuk, ami azt jelenti, hogy objektumonként nem külön változóként viselkednek, hanem magában az osztályban van tárolva az érték, és ha módosul, az az összes objektumra hatással van.

```
public static double NodeSize { get; set; } = 30;
public static double FontSize { get; set; } = 11;
```

8. kódrészlet - Csúcspont- és betűméret

Az osztály két konstruktorral rendelkezik. Az első „üres” gráfok létrehozására szolgál, az üres esetünkben annyit jelent, hogy egy előre létrehozott módosítható sablont tölt be a program. A konstruktor definiálja a csúcsmátrixot, a csúcspont neveket, a koordinátákat, a forrás és cél csúcspontokat, majd meghívja az *Initialization* függvényt, ami a két konstruktor közös része. Az *Initialization* függvény beállítja a *Form*-on a forrás és cél csúcspontok nevét, megkezdi gráf kirajzolását a *Draw* függvény helyi megvalósításával, és felhelyezi a *Canvas*-ra a hozzá tartozó eseménykezelőket. A *Graph* osztály, és a *SPLine* osztály *Draw* függvényei csak a nevükben

egyeznek. A *ResetScale* függvény azért kell lefuttatni az első konstruktorban, mert ha belenagyítunk a gráfba és ezt követően új „üres” gráfot hozunk létre, akkor az új méretezés megmarad, mivel a betű- és csúcspontméret globális. A második konstruktorban erre nincs szükség. A második konstruktor fájlból olvas be egy korábban már lementett gráfot. A fájl tartalmát kiolvasás után egy *string content* változóba helyeztük (lásd. 1. ábra). Ezt szétbontottuk sorokra, és a sorokat szóközönként tagoltuk. A sorok sorrendje számít, az első csúcspont az éllistában az első a gráf mátrixban is. A kapcsolatok mindig az adott sor által reprezentált csúcspontból indulnak ki. A Graph osztály vezérlésénél az eseménykezelők működéséhez szükség volt egy olyan függvénycsoport létrehozásához, amely a rákattintott grafikus objektum alapján, legyen az akár görbe, csúcspont szövegesdoboz, vagy annak ellipszise meg tudja állapítani az adott grafikus objektum szülő objektumát, azaz az azt tartalmazó Node, vagy Spline objektumot. Erre a célra készítettük el a Search függvényt, melynek négy úgynevezett „túlterhelése” (overload) van. Ez azt jelenti, hogy ugyan ezen a néven 4 különböző bemenetű, vagy visszatérési értékkel rendelkező megvalósítás létezik ugyanazon osztályban. Egy megvalósítás van Node típusú visszatérési értékkel, ami az ellipsziseket vizsgálva járja be a Node-okat. A maradék két megvalósításnak Spline a visszatérési típusa. Ebből az első Path típus alapján, azaz a görbe alapján keresi a szülő Spline-t, a másik a forrás és a nyelő csúcspont alapján keres. A negyedik egy speciális megvalósítás, ami object típust ad vissza, mivel Node és Spline objektumokat is egyaránt visszaadhat, ezért ezeket object őssztályra alakítja. Szövegdobozok alapján keres. A csúcspontok átnevezéséhez megvalósítottunk egy RenameNode(string _newname) eljárást. Az ehhez hasonló egérrel megvalósított vezérlésű metódusok alapja, hogy az egérkattintásra csatolt eseménykezelő, miután megállapította a Search függvények segítségével, hogy melyik csúcspontról, vagy kapcsolatról van szó, azt elmenti egy globális RightTarget (azért right mert jobb klikk) változóba, és ezen végzi el a műveleteket a metódus. A RenameNode esetén sincs ez másképp, miután meggyőződött róla, hogy a RightTarget nem null, azaz van valami a RightTarget-ben, és hogy ez a valami Node típusú-e, elvégzi rajta az átnevezési műveletet, ami a Node osztály metódusain keresztül van megvalósítva. A RenameNode-nak elsősorban azonosítással kapcsolatos szerepe van.

```

public void RenameNode(string _newname)
{
    if (RightTarget != null) {
        if (RightTarget is Node node) {
            string newname = _newname.Replace(' ', '_');
            if (!nodenames.Contains(newname)) {
                string prev_name = node.GetLabel();
                node.ChangeLabel(newname);
                nodenames[nodenames.IndexOf(prev_name)] = newname;
            }
            RightTarget = null;
        }
    }
}

```

9. kódrészlet - *RenameNode* függvény

A következő *RightTarget* alapú megvalósításokat alakítottuk még ki a vezérléshez:

A *SetSinkNode* eljárás, felépítése hasonló a *RenameNode*-hoz (lásd. 9.kódrészlet), csak a csúcspont neve helyett a nyelő nevét állítja át a gráfban. Az ellenőrzések végett strukturális szempontból az összes *RightTarget* alapú megvalósítás hasonló. A *SetSourceNode* függvény ugyan azt csinálja, mint a *SetSinkNode*, csak a forrást állítja át. *SetStreamSizeOfSPLine* átállítja a kapcsolat maximális kapacitását. A *RemoveNode* függvény ugyan még mindig *RightTarget* alapú megvalósítás, de sokkal komplexebb struktúrával rendelkezik, mint a korábban említett társai. Ahogy a neve is sugallja, azért hoztuk létre ezt a megvalósítást, hogy csúcspontokat törölhessünk, de ez nem csak a felületről történő eltávolítással jár, hanem az adott csúcspont adatait a gráf mátrixból, és a *nodes* tömbből is el kell távolítani, az adott csúcspont összes kapcsolatával együtt. Ehhez meg kell találni az összes kapcsolatot, és a csúcsponttal együtt le kell rajtuk futtatni a *Dispose* eljárást, amit az *IDisposable* interfész megvalósításakor hoztunk létre osztályaikban.

```

public void RemoveNode()
{
    if (RightTarget != null) {
        if (RightTarget is Node node) {
            string node_name = node.GetLabel();
            List<IDisposable> toDispose = new List<IDisposable>();
            nodes.Remove(node);
            SPLine[] array = splines.ToArray();
            for (int i = array.Length - 1; i >= 0; i--) {
                SPLine sp = array[i];
                if (sp.FromNode == node || sp.ToNode == node) {
                    toDispose.Add(sp);
                    splines.RemoveAt(i);
                }
            }
            toDispose.Add(node);
            foreach (IDisposable disp in toDispose) {
                disp.Dispose();
            }
            int index = nodenames.IndexOf(node_name);
            nodenames.RemoveAt(index);
            graph_matrix.RemoveAt(index);
            foreach (List<int> row in graph_matrix) {
                row.RemoveAt(index);
            }
            node_coordinates.RemoveAt(index);
            RightTarget = null;
        }
    }
}

```

10. kódrészlet - RemoveNode függvény

A *RemoveSPLine* eljárás egyszerűbb megvalósítás, csak egy kapcsolatot kell törölnie, ehhez csak egy *Dispose* függvény kell meghívnia, és a gráf mátrixban csak egy értéket kell átírnia nullára. Minden *RightTarget* alapú megvalósítás futtatása után üríteni kell a *RightTarget* tartalmát *null*-ra állítva annak értékét. A következő függvények már nem *RightTarget* alapú megvalósítások. A *NewNode* függvény a *RemoveNode* ellentéte, új csúcspontot hoz létre, ehhez módosítani kell a gráf mátrixot új sort és oszlopot felvéve, a *nodes* tömbhöz pedig hozzá kell adnia az új *Node*-ot. A mátrix és a tömb végére kerülnek az új elemek. Számít a sorrend! A csúcspont neve az utolsó csúcspont nevének utolsó karakterét követő ASCII karakter, amennyiben ez nincs használatban más csúcspont által, ellenkező esetben az ezt követő. Az új csúcspont koordinátáit úgy állapítottuk meg, hogy a *Graph* osztály követi a *Canvas*-on történő egérmozgást egy *mouse* globális változóban, ez lett az új *Node* koordinátája.

```

public void NewNode()
{
    string name = "" +
        Convert.ToChar(Encoding.ASCII.GetBytes(nodenames.Last()).Last() + 1);
    for (int i = 0; i < nodenames.Count; i++) {
        if (nodenames[i].Equals(name)) {
            name = "" + Convert.ToChar(Encoding.ASCII.GetBytes(name).Last() +
                1);
            i = -1;
        }
    }
    Node n = new Node(canvas, name) {
        EventHandler = MouseUpNode
    };
    n.SetPosition(mouse.X, mouse.Y);
    nodes.Add(n);
    nodenames.Add(name);
    node_coordinates.Add(new Point(mouse.X, mouse.Y));
    foreach (List<int> row in graph_matrix) {
        row.Add(0);
    }
    List<int> newrow = new List<int>();
    for (int i = 0; i <= graph_matrix.Count; i++) {
        newrow.Add(0);
    }
    graph_matrix.Add(newrow);
}

```

11. kódrészlet - NewNode függvény

A *NewSPLine* függvényt új kapcsolatok kialakítására hoztuk létre. Az erre a célra létrehozott eseménykezelő megjegyzi a jobb egérgomb lenyomásának és felengedésének bekövetkeztekor érintett *Node* objektumokat, és ezek között a megadott sorrendben ezen függvény segítségével létrehozza a kapcsolatot. A függvény által létrehozott kapcsolat elhelyezését az *SPLine* osztály *Snap* (lásd 5.kódrészlet) eljárásával valósítottuk meg. A gráf elmentését a meghatározott formátum alapján (lásd 1.ábra) a *Save* metódussal tettük lehetővé. Az eljárásnak egy opcionális bemenete van, a *boolean* típusú *saveas* paraméter, ami alap esetben hamisra van állítva. Meghatározza, hogy kényszerítetten új fájlba mentse-e a változtatásokat, vagy, ha hamis, akkor felülírja a korábbi mentést. Ha kényszerített mentés történt, akkor az ezt követő mentések az új fájlba történnek. A gráf kirajzolására létrehoztuk a *Draw* függvényt, ami a Canvas felületének tisztítása után a gráf mátrix, a csúcspontok koordinátái és elnevezése alapján a *Node* és *SPLine* osztályok objektumaiból összerakja a gráfot. Háttérrel rajzol a *Canvas*-nak, mivel csak akkor lehet követni a *Canvas*-on az egérmozgást, ha az egér egy grafikus objektum felett halad el. Az eljárás az *SPLine*-ok és *Node*-ok létrehozásakor megadja az eseménykezelőket is az objektumoknak, amik ezen osztály szintjén vannak definiálva. A háttér az egész gráf újra rajzolása nélküli átméretezésére létrehoztuk a *BGRectResize* eljárást, ami a globálisan *BGRect* néven tárolt háttér szélességét és magasságát módosítja a szintén globálisan tárolt *CanvasSize* tulajdonság méretére, ami a *Canvas* jelenlegi méretét tartalmazza. A *BGRectResize* és a *Draw* függvény is felhasználja a *CheckNodesInside* metódust, melynek feladat megállapítani, hogy van-e olyan csúcspont, aminek bármely pontja a *Canvas*-on kívül van. A csúcspont kezdő

koordinátái és szélessége (a csúcspontméret) alapján ellenőriz, tehát nem az ellipszis pontjait vizsgálja, hanem egy elméleti szinten köré rajzolt kockát. Ha a *Canvas*-on kívül van egy *Node*, akkor megjelenít egy figyelmeztető szövegdobozt a *Canvas* bal alsó sarkában.

```
private void CheckNodesInside()
{
    bool warn = false;
    foreach (Node x in nodes) {
        DpiScale scale = form.DpiScale;
        if (x.Position.X < 0 || x.Position.Y < 0 || x.Position.X + NodeSize >
            CanvasSize.X / scale.DpiScaleX || x.Position.Y + NodeSize >
            CanvasSize.Y / scale.DpiScaleY) {
            warn = true;
            break;
        }
    }
    if (warn) {
        if (!canvas.Children.Contains(warningbox)) {
            canvas.Children.Add(warningbox);
            Canvas.SetBottom(warningbox, 0);
            Canvas.SetLeft(warningbox, 0);
            Panel.SetZIndex(warningbox, 10);
        }
    }
    else {
        if (canvas.Children.Contains(warningbox)) {
            canvas.Children.Remove(warningbox);
        }
    }
}
```

12. kódrészlet - *CheckNodesInside* függvény

2.7 A Graph osztály szimulációért felelős metódusai

A szimulációért felelős függvények közé tartoznak mind a szimuláció grafikus megjelenítéséért, mind a Ford-Fulkerson, Edmonds-Karp és Boykov-Kolmogorov algoritmusok működéséért felelős megvalósítások. Magára a szimulációhoz szükséges számítások elvégzésére létrehoztuk a *MaxFlow(Method method = Method.DFS, bool stepbystep = false)* függvényt, amely két opcionális bementtel rendelkezik. A *method* paraméter meghatározza, hogy a maximális folyam kereső algoritmus milyen módszert alkalmazzon a keresés során, az értéke lehet: *Method.BFS*, ami az Edmonds-Karp algoritmus megvalósítása, *Method.DFS*, ami a Ford-Fulkerson megvalósítása, vagy *Method.Boykov_Kolmogorov*, ami a Boykov-Kolmogorov algoritmus megvalósítása. A második paraméter a lépésről lépésre módot teszi lehetővé. A *MaxFlow* függvény a (GeeksforGeeks, 2023) forrás fordFulkerson függvénye alapján, a Boykov-Kolmogorov megfelelő működéséért felelős részek pedig a (An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision, 2004) forrás alapján vannak megvalósítva. Két fontosabb változót tudunk kiemelni a metódusból, a *residGraph* változó, ami a gráf mátrix pontos mását veszi fel a műveletek végrehajtása előtt, viszont a gráf mátrixal ellentétben a típusa egy *integer* kétdimenziós tömb lesz, mivel ez jelentősen gyorsabb a *List* megfelelőjénél. A másik kiemelhető változó a *parent*

egydimenziós *integer* tömb, amiben a meghatározott útvonalat tároltuk. A függvény a megtalált útvonalakat eltávolítja a *residGraph*-ból, majd addig futtatja újra a keresést, amíg már nincs fellelhető útvonal. A megtalált útvonalakat a *MaxFlow* függvény a *local_flow_paths* globális változóba menti el, a keresési fát pedig a *local_Trees* változóba. Érdekes megjegyezni, hogy a *BFS* és *DFS* eljárásoknál a *local_Trees* csak megjelenítés céljából kerül mentésre, ugyanis ezen változó számítási célra történő felhasználására csak a Boykov-Kolmogorov algoritmusban kerül sor. A futásidőt is méri, amit a globális *LastCalcTime* változóba ment.

```
public int MaxFlow(Method method = Method.DFS, bool stepbystep = false)
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    int s = nodenames.IndexOf(source);
    int t = nodenames.IndexOf(sink);
    if (s == -1 || t == -1) {
        return 0;
    }
    int maxflow = 0;
    local_flow_paths.Clear();
    local_Trees.Clear();
    Tree[] trees = new Tree[nodenames.Count];
    trees[s] = Tree.S;
    trees[t] = Tree.T;
    int[] parent = new int[nodenames.Count];
    int[,] residGraph = new int[nodenames.Count, nodenames.Count];
    for (int i = 0; i < nodenames.Count; i++) {
        for (int j = 0; j < nodenames.Count; j++) {
            residGraph[i, j] = graph_matrix[i][j];
        }
    }
    if (method != Method.Boykov_Kolmogorov) {
        while (method == Method.BFS ? BFS(residGraph, parent, trees, s, t) :
            DFS(residGraph, parent, trees, s, t)) {
            List<int> currentpath = new List<int>();
            int pathflow = int.MaxValue;
            for (int i = t; i != s; i = parent[i]) {
                currentpath.Add(i);
                int j = parent[i];
                pathflow = Math.Min(pathflow, residGraph[j, i]);
            }
            currentpath.Add(s);
            currentpath.Reverse();
            currentpath.Add(pathflow);
            local_Trees.Add((Tree[])trees.Clone());
            for (int i = 0; i < trees.Length; i++) {
                trees[i] = Tree.Free;
            }
            trees[s] = Tree.S;
            trees[t] = Tree.T;
            local_flow_paths.Add(currentpath);
            for (int i = t; i != s; i = parent[i]) {
                int j = parent[i];
                residGraph[j, i] -= pathflow;
                residGraph[i, j] += pathflow;
            }
            maxflow += pathflow;
        }
    }
    else {
        //Boykov_Kolmogorov
        Tuple<bool, int, int> output;
        Queue<int> active = new Queue<int>(), orphan = new Queue<int>();
        active.Enqueue(s);
        active.Enqueue(t);
        while ((output = Boykov_Kolmogorov(residGraph, parent, trees, active, s, t)).Item1) {
            //Augmentation fázis
            List<int> currentpath = new List<int>(), revcurrentpath = new List<int>();
            int pathflow = Tree_Cap(residGraph, output.Item2, output.Item3, trees[output.Item2]);
            int itS = (trees[output.Item2] == Tree.S) ? output.Item2 : output.Item3;
            int itT = (trees[output.Item2] == Tree.S) ? output.Item3 : output.Item2;
            // S-nek kell lennie (szabványos működés)
            for (int i = itS; parent[i] != -1; i = parent[i]) {
                currentpath.Add(i);
                int j = parent[i];
                pathflow = Math.Min(pathflow, residGraph[j, i]);
            }
        }
    }
}
```

```

    }
    // T-nek kell lennie (ellentétes működés)
    for (int i = iT; parent[i] != -1; i = parent[i]) {
        revcurrentpath.Add(i);
        int j = parent[i];
        pathflow = Math.Min(pathflow, residGraph[i, j]); ;
    }
    currentpath.Add(s);
    revcurrentpath.Add(t);
    currentpath.Reverse();
    currentpath.AddRange(revcurrentpath);
    residGraph[itS, iT] -= pathflow;
    residGraph[itT, itS] += pathflow;
    // S-nek kell lennie (szabványos működés)
    for (int i = itS; parent[i] != -1; i = parent[i]) {
        int j = parent[i];
        residGraph[j, i] -= pathflow;
        residGraph[i, j] += pathflow;
    }
    // T-nek kell lennie (ellentétes működés)
    for (int i = iT; parent[i] != -1; i = parent[i]) {
        int j = parent[i];
        residGraph[i, j] -= pathflow;
        residGraph[j, i] += pathflow;
    }
    maxflow += pathflow;
    // - élek telítettségének kezelése
    for (int index = 0; index < currentpath.Count - 1; index++) {
        int i = currentpath[index];
        int j = currentpath[index + 1];
        if (residGraph[i, j] < 1) {
            if (trees[i] == Tree.S && trees[j] == Tree.S) {
                parent[j] = -10;
                orphan.Enqueue(j);
            }
            else if (trees[i] == Tree.T && trees[j] == Tree.T) {
                parent[i] = -10;
                orphan.Enqueue(i);
            }
        }
    }
    local_Trees.Add((Tree[])trees.Clone());
    //Adoption fázis
    List<int> toRemove = new List<int>();
    while (orphan.Count > 0) {
        int p = orphan.Dequeue();
        for (int i = 0; i < nodenames.Count; i++) {
            if (trees[i] == trees[p] && Tree_Cap(residGraph, i, p, trees[i]) > 0) {
                int io = parent[i];
                while (io >= 0) {
                    io = parent[io];
                }
                if (io == -1) {
                    parent[p] = i;
                    break;
                }
            }
        }
        if (parent[p] == -10) {
            for (int i = 0; i < nodenames.Count; i++) {
                if (trees[i] == trees[p]) {
                    if (Tree_Cap(residGraph, i, p, trees[i]) > 0) {
                        active.Enqueue(i);
                    }
                    if (parent[i] == p) {
                        orphan.Enqueue(i);
                        parent[i] = -10;
                    }
                }
            }
            trees[p] = Tree.Free;
            toRemove.Add(p);
        }
    }
    active = RemoveElementsFromQueue(active, toRemove);
    currentpath.Add(pathflow);
    local_flow_paths.Add(currentpath);
}
}
stopwatch.Stop();
LastCalcTime = stopwatch.Elapsed.TotalSeconds;

```

```

    if (!stepbystep) {
        SetStreamValues(residGraph);
    }
    return maxflow;
}

```

13. kódrészlet - MaxFlow függvény

A DFS és BFS függvényeket a (GeeksforGeeks, 2023) (Javatpoint, 2021) források alapján készítettük el, ezek nem rekurzív megvalósítások. A két eljárások között az a legszembevetőbb különbség, hogy míg a *DFS* verem alapú megoldást használ, ami a LIFO, azaz utoljára be-először ki megvalósítást alkalmaz, addig a *BFS* sort használ, ami viszont FIFO, vagyis először be- először ki megvalósítás. A *visited* bittömb arra szolgál, hogy számontarthassuk, hogy mely csúcspontokat vizsgáltuk már meg. A *queue*, vagy a *DFS* esetében a *stack* változóban tároltuk azokat a csúcspontokat, amelyekbe az adott pillanatban vizsgált csúcspontból közvetlenül el lehet jutni a kapcsolatokon keresztül. A függvények a meghatározott útvonalat, a *parent* változón keresztül adják vissza a *MaxFlow*-nak. Az eljárások működéséhez 4 bemenetet kell megadni, a *residGraph*-ot, ezt csak olvasni fogja a függvény, a *parent* ki- bemeneti változót, az *s* forráscsúcspont számát, és a *t*-t ami a nyelő sorszámát reprezentálja. A függvények kimenete egy *boolean* típus, ami azt határozza meg, hogy a függvény talált-e útvonalat, vagy sem.

```

private bool DFS(int[,] residGraph, int[] parent, int s, int t)
{
    BitArray visited = new BitArray(nodenames.Count, false);
    Stack<int> stack = new Stack<int>();
    stack.Push(s);
    parent[s] = -1;
    while (stack.Count > 0) {
        int u = stack.Pop();
        if (!visited.Get(u)) {
            visited.Set(u, true);
            for (int i = 0; i < nodenames.Count; i++) {
                if (visited.Get(i) == false && residGraph[u, i] > 0) {
                    if (i == t) {
                        parent[i] = u;
                        return true;
                    }
                    stack.Push(i);
                    parent[i] = u;
                }
            }
        }
    }
    return false;
}

```

14. kódrészlet - DFS függvény

A *Boykov_Kolmogorov* függvény a DFS és BFS-el szemben egy három elemű *Tuple*-t ad vissza, melynek első eleme azt határozza meg, hogy talált-e javítótutat, és ha igen, akkor a rá következő két elem a *parent* tömb azon elemeit határozza meg, ahol a két keresési fa találkozott. Az eljárásnak bemenetként 6 változóra van szüksége. Ebből a DFS-hez képest a két új változó a

tree tömb ami a keresési fákban felfedezett csúcsok hovatartozását határozza meg, és az *active* sor, ami az aktív csúcsok sorszámát tartalmazza.

```
private Tuple<bool, int, int> Boykov_Kolmogorov(int[, ] residGraph, int[] parent,
Tree[] tree, Queue<int> active, int s, int t)
{
    parent[s] = -1;
    parent[t] = -1;
    while (active.Count > 0) {
        int p = active.Peek();
        for (int i = 0; i < nodenames.Count; i++) {
            if (Tree_Cap(residGraph, p, i, tree[p]) > 0) {
                if (tree[i] == Tree.Free) {
                    tree[i] = tree[p];
                    parent[i] = p;
                    active.Enqueue(i);
                }
                else if (tree[p] != tree[i]) {
                    return new Tuple<bool, int, int>(true, p, i);
                }
            }
        }
        active.Dequeue();
    }
    return new Tuple<bool, int, int>(false, -1, -1);
}
```

15. kódrészlet - Boykov_Kolmogorov függvény

A szimuláció megjelenítéséért felelős megvalósítások a *SetStreamValues* és a *SetStreamValues_StepByStep* eljárások. A *local_flow_paths* változóban tárolt adatok dolgozzák fel. A korábban említett *MaxFlow* függvény futtatása során hoztuk létre ezt a változót, ami az általa meghatározott útvonalakat tárolja. Az utolsó elem a felhasznált kapacitást, míg az előtte lévő elemek a csúcspontok sorrendjét tartalmazzák. A *SetStreamValues* a gráfon csak a kapcsolatok kapacitás értékeit módosítja, még hozzá, a felhasznált mennyiséget. Ezen kívül beleírja az útvonalakat a javítóutakat tartalmazó szövegdobozba. Ezzel szemben a *SetStreamValues_StepByStep* a *local_flow_paths* egy adott elemének indexét kell megadni, amit az eljárás megjelenít a szövegdobozba, a *Canvas*-on nem csak a kapacitásokat írja ki, hanem még kijelöli az útvonalat is, úgy, hogy a kapcsolatokat kékre színezi, valamint a csúcsok beszínezésével még az adott keresési fát/fákat is kijelöli.

```

public int SetStreamValues_StepByStep(int index)
{
    foreach (SPLine spl in splines) {
        spl.Redraw();
    }
    SPLine sP;
    int[] element = local_flow_paths[index].ToArray();
    string text = nodenames[element[0]];
    for (int i = 1; i < element.Length - 1; i++) {
        text += " -> " + nodenames[element[i]];
        if ((sP = Search(nodes[element[i - 1]], nodes[element[i]])) != null) {
            sP.UsedStreamSize += element[element.Length - 1];
            sP.Highlight();
        }
    }
    text += " | amount:" + element[element.Length - 1];
    form.CPath_box.AppendText(text + "\r\n");
    if (local_Trees.Count > 0) {
        for (int i = 0; i < nodes.Count; i++) {
            Node node = nodes[i];
            node.ResetColor();
            if (local_Trees[index][i] != Tree.Free) {
                node.SetColor(local_Trees[index][i] == Tree.T ?
                    Brushes.DodgerBlue : Brushes.Orange);
            }
        }
    }
    return element[element.Length - 1];
}

```

16. kódrészlet - *SetStreamValues_StepByStep* függvény

2.8 A Graph osztály eseménykezelői

A *Canvas* vezérlését egérhasználatra optimalizálva oldottuk meg, billentyűzettel nem lehetséges. A gráf nagyításához létrehoztuk a *Zoom* eseménykezelőt. A *Zoom* eseménykezelőt az egér görgőjére helyeztük fel. Csak akkor fog lefutni, ha a *ControlsEnabled* változó igazra van állítva, vagyis nincs letiltva a vezérlés. Ezt a biztonsági mechanizmust minden eseménykezelőbe beépítettük. A függvény a görgetés irányából megállapítja, hogy nagyítani, vagy zsugorítani szeretne a felhasználó, ezután a *Scale2D* metódussal kiszámolja az új csúcspont koordinátákat, átállítja a csúcspont- és betűméretet, majd a *Draw* függvény meghívásával újra rajzolja a gráfot. A *Canvas* felületén történő egérmozgást a *MouseMoveOnCanvas*-al kezeljük. Ez egy többfunkciós eseménykezelő, elsődleges feladata az egér pozíciójának feljegyzése, amit a *mouse* változóba ment, de bizonyos egérműveleteket is lekezel. Ilyen például a bal egérgomb lenyomásakor végrehajtandó csúcspont mozgató művelet. Ha a bal egérgombot lenyomva tartjuk egy csúcsponton, és azt elkezdjük így a felületen húzni, akkor az adott csúcspont követni fogja az egér mozgását. Ezt a *Snap* metódus meghívása követi. A *MouseMoveOnCanvas* az egész gráf mozgására is nyújt lehetőséget, ami magán a *Canvas*-on végzett bal egérgombos mozgató műveletekkel idézhető elő. A *MouseMoveOnCanvas* eseménykezelő utolsó lépésként meghívja a *CheckNodesInside* eljárást (lásd 12.kódrészlet).

```

private void MouseMoveOnCanvas(object sender, MouseEventArgs args)
{
    if (ControlsEnabled) {
        mouse = args.GetPosition(canvas);
        if (args.LeftButton == MouseButtonState.Pressed) {
            if (target != null) {
                target.SetPosition(mouse.X + targetOffset.X,
                mouse.Y + targetOffset.Y);
                foreach (SPLine line in targets_sps) {
                    line.Snap(graph_matrix, nodes);
                }
            }
            else if (globalOffset != null) {
                for (int i = 0; i < nodes.Count; i++) {
                    nodes[i].SetPosition(mouse.X + globalOffset[i].X,
                    mouse.Y + globalOffset[i].Y);
                }
                foreach (SPLine line in splines) {
                    line.Snap(graph_matrix, nodes);
                }
            }
            CheckNodesInside();
        }
    }
}

```

17. kódrészlet - *MouseMoveOnCanvas* függvény

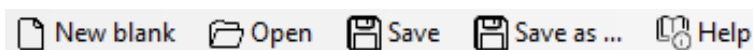
A *MouseDownOnCanvas* a *Canvas*-on történő bal egérgomb lenyomásának eseménykezelője. *args* bemenetének típusa *System.EventArgs*. Beállítja a *targetOffset*-et, vagy a *globalOffset*-et. A *targets_sps* is meghatározza, ami a csúcsponthoz közvetlenül kapcsolódó kapcsolatokat tartalmazza. A *MouseUpOnCanvas* eseménykezelő szinkronizálja a mozgatót követően a csúcspontok új koordinátáit a *node_coordinates* változóba. Az új kapcsolatok kialakítására a jobb egérgomb egy csúcspontot történő lenyomásával, majd egy másik, a cél csúcspont feletti felengedésével valósítottuk meg. ehhez két eseménykezelőre volt szükség, a *RightMouseDownOnCanvas*-ra és a *RightMouseUpOnCanvas*-ra. A *RightMouseDownOnCanvas* elmenti azt a csúcspontot, ami felett lenyomtuk a jobb egérgombot. A *RightMouseUpOnCanvas* létrehozza a *SPLine*-t, amennyiben egy csúcspont felett engedték fel az egeret. A *RightMouseUpOnCanvasBG* a *Canvas* háttér objektumán (*BGRect*) figyeli a jobb egérgomb felengedését. Ha ez az esemény bekövetkezik az adott koordinátákban a *form* szintjén megjeleníti az alapértelmezett helyi menüt (*defaContext*), ami lehetőséget nyújt új csúcspont létrehozására. A *MouseUpNode* eseménykezelő megvalósítás feladata a *Node* osztály objektumain történő jobb egérgomb felengedésére történő reakció. A *nodeContext* helyi menüt jeleníti meg. A *MouseUpSPLine* ugyanez, csak a kapcsolatokkal, és az *splineContext*-et jeleníti meg. A *Scale2D* síkbeli koordináták origóhoz viszonyított skálázására szolgáló függvény. Egy *System.Windows.Point* típust ad vissza, mint visszatérési értéket. A bemenete két *Point* és egy *double* változó, amelyek az origó és a skálázandó pont koordinátáit, valamint a skálázás mértékét határozzák meg.

```
private static Point Scale2D(Point origo, Point target, double amount)
{
    double nX = (target.X - origo.X) * amount + origo.X;
    double nY = (target.Y - origo.Y) * amount + origo.Y;
    return new Point(nX, nY);
}
```

18. kódrészlet - Scale2D függvény

2.9 A Form megvalósítása

A Canvas-unkat is tartalmazó Form megvalósításánál a dotnet által létrehozott Form1 típusú a Form osztályból örököltetett sablont fejlesztettük tovább. A Form1 konstruktora létrehozza a Canvas-t és belehelyezi az elementhost1-be. Létrehoz egy üres gráfot, és hozzáadja a Form-hoz a MoveEvent eseménykezelőt, amit az ablak aktiválására és mozgatására reagáló eseményekre helyez fel. A MoveEvent eseménykezelő feladata, hogy az elementhost képernyőkoordinátáit mentse le változás esetén, ez a képernyő bal felső sarkától számított pixelkoordinátákat jelenti, valamint a jelenlegi DPI skálázás értékeit is nyilvántartja, ami több képernyős rendszereknél változhat. Erre azért van szükség, mert a helyi menük meghívásakor, a mouse változóban tárolt egérkoordináták relatívak, azaz csak a Canvas bal felső sarkától vannak meghatározva. Egy helyi menü meghívásához képernyőkoordinátákat kell megadni, ezért ennek a két pozíciónak az összeadásával megkaphatjuk a szükséges értékeket. A Form-unk fel van szerelve egy úgynevezett ToolStripMenu-vel. Ezt magyarul, eszközöket tartalmazó menüszalagnak is nevezhetnénk. Ezen találhatóak a gráfok mentésével és megnyitásával kapcsolatos vezérlést megvalósító elemek.



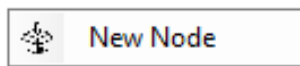
4. ábra - ToolStripMenu menüszalag

A „New blank” gomb új „üres” gráfot hoz létre. Az „Open” gomb feladata már korábban mentett gráfok betöltése. A „Save” gomb elmenti a gráfot egy fájlba a *Graph* osztály *Save* metódusával. A „Save as ...” gomb ugyan ezt csinálja, csak kényszerített mentést hajt végre, úgy, hogy a *Save* eljárásba igaz értéket ad meg bementként. A „Help” elem megnyomásakor egy útmutató jelenik meg a képernyőn.

```
Help.ShowHelp(this, System.Windows.Forms.Application.StartupPath +
    "\\MaximumFlowSimulator_HELP.chm");
```

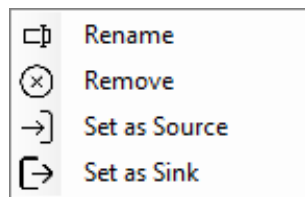
19. kódrészlet - Útmutató meghívása fájlból

A következő szekcióban a helyi menüket és az általuk felkínált lehetőségeket fogjuk tárgyalni.



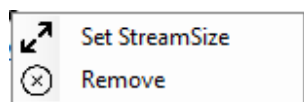
5. ábra - defaContext helyi menü

A *defaContext* helyi menü a *Canvas* hátterére jobb egérgombbal történő kattintással hívható elő. Csak egy lehetőséget kínál, amivel új csúcspontokat hozhatunk létre.



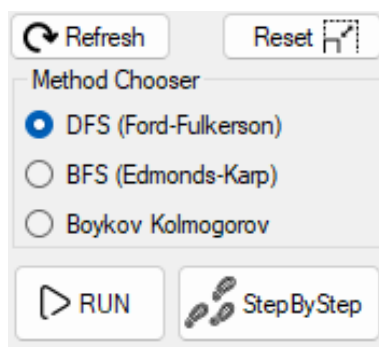
6. ábra - *nodeContext* helyi menü

A *nodeContext* egy a *Canvas*-on lévő csúcspontra történő jobb egérgattintás műveletével hívható elő. Ez a menü négy lehetőséget kínál. Az első a „Rename” gomb, ami átnevezi az adott csúcspontot. Az új nevet az általunk létrehozott *InputForm* osztállyal szerzi meg. A „Remove” gomb a *Graph* osztály *RemoveNode* eljárását használja, ami csúcspontokat töröl. A „Set as Sink” és „Set as Source” gombok a gráf nyelő és forrás csúcspontjait állítják át, erre a *Graph* osztály *SetSinkNode* és *SetSourceNode* függvényeit használják fel.



7. ábra - *splineContext* helyi menü

Az *splineContext* egy görbére, azaz *SPLine* objektumra kattintással hívható meg. Két lehetőséget kínál fel a menü: a „Set StreamSize” és a „Remove” opciókat. A „Set StreamSize” lehetőség átállítja a kapcsolat maximális kapacitását. *InputForm*-al bekéri az új értéket és a *Graph* osztály *SetStreamSizeOfSPLine* függvényével beállítja azt. A „Remove” opció pedig eltávolítja a kapcsolatot a *RemoveSPLine* metódussal. A *Form*-unkat ezen kívül még négy a vezérlést megvalósító gombbal, és egy az algoritmusok kiválasztására szolgáló ablakkal szereltük fel.



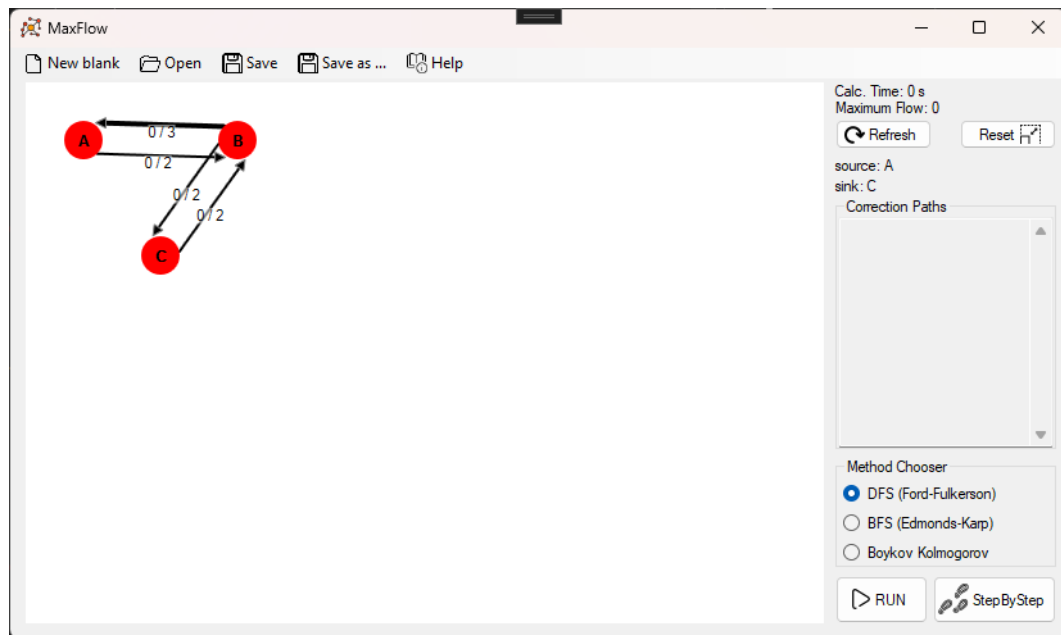
8. ábra - A vezérlést megvalósító gombok

Ezek a „Refresh”, „Reset”, „Run” és „StepByStep” gombok, az algoritmus választó felület pedig a „Method Chooser”. A „Refresh” gomb a *Graph* osztály *Draw* metódusával újra rajzolja a gráfot. A „Reset” is hasonló, de ez az újra rajzolás előtt meghívja a *Graph.ResetScale()*

parancsot, így visszaállítva a csúcspont- és betűméretet. A „Run” gomb működése a „StepByStep” gomb lenyomásától függ. Ha nem nyomtuk meg a „StepByStep” gombot, akkor a *MaxFlow* függvény lefuttatásával megjeleníti a maximális folyam szimuláció végeredményét. Ha lenyomtuk a gombot, akkor a „Run” gomb egy „Cancel”, azaz mégsem feliratú gombbá alakul, és ezen túl a megnyomása megszakítja a StepByStep szimulációt. A „StepByStep” megnyomása lecseréli a „Run” gombot „Cancel” gombra, és saját magát pedig „Next” gombra. Ezen túl, ha megnyomjuk a „Next” gombot, akkor a következő lépést jelöli ki a gráfon a szimulációban. A „StepByStep” Gomb megnyomása ezen kívül még felfüggeszti a vezérlést is. Erre a *ChangeSBSSStatus* függvényt használja. A *ChangeSBSSStatus* átálítja a *stepbystep* globális változót, ami meghatározza, hogy meg e nyomtuk a „StepByStep” gombot. Ezt követően lecseréli a két korábban említett gombot, és a *stepbystep* változó értékétől függően letiltja, vagy aktiválja a vezérlést. A *Form1* osztályban implementáltunk egy billentyűzetkombinációs mentési megvalósítást is, amit a *ProcessCmdKey* függvény felülírásával valósítottunk meg. Az új függvény megvizsgálja, hogy ctrl+s kombinációt nyomtunk-e meg, ha igen, akkor a *Graph* osztály *Save* (lásd 15.kódrészlet) metódusával elvégzi a mentést, és igaz (true) értéket ad vissza, jelezvén, hogy lekezelte a billentyűzetparancsot. Egyébként az ősosztályból meghívja az eredeti megvalósítást.

```
protected override bool ProcessCmdKey(ref Message msg, Keys keyData)
{
    if (keyData == (Keys.Control | Keys.S)) {
        graph.Save();
        return true;
    }
    return base.ProcessCmdKey(ref msg, keyData);
}
```

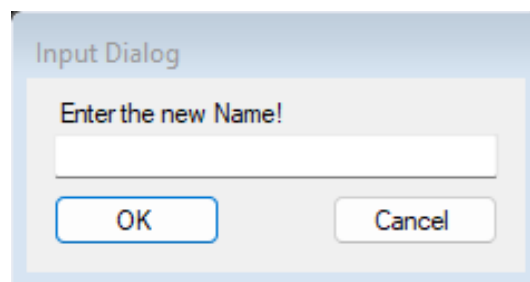
20. kódrészlet - *ProcessCmdKey* felülírása



9. ábra - A Form1 felülete

2.10 Az InputForm osztály

Egy *Form*-ból származtatott osztály, mely biztosít egy felületet valamiféle adat bevitelére. A felület tartalmaz egy szövegdobozt, ami utasítja a felhasználót, hogy miféle adatot kell megadnia, egy *System.Windows.Forms.TextBox* típusú beviteli mezőt, amibe a felhasználó megadhatja az adatokat, és két gombot, amelyekkel el lehet fogadtatni a bemenetet (OK), vagy meg lehet szakítani a műveletet (Cancel).



10. ábra - InputForm felülete

Az *InputForm* konstruktora nem publikus, hanem privát, azaz az osztályon kívül nem hívható meg. Az osztály nem példányosítható. Az *InputForm* rendelkezik egy az osztály szintjén globális *output string* típusú változóval, melynek értéke alap esetben *string.Empty*, tehát üres *string*. A konstruktor a *Form* megjelenítése után a *TextBox* típusú beviteli mező értékét ennek a változónak értékére állítja. A „Cancel” gomb megnyomásakor a beviteli mező tartalma nem kerül mentésre az *output*-ba, értéke *string.Empty* marad. Az ablak ezt követően bezárul. Ha az „OK” gombot nyomjuk meg, akkor a program lefrissíti az *output* értékét, és ezt követően zárja be az ablakot. Az osztály rendelkezik egy *ChangeLabel* függvénnyel, ami a megadott szövegre

cseréli az utasításokat tartalmazó szövegdozoz tartalmát. Annak ellenére, hogy az osztályból publikus konstruktor híján nem lehet objektumokat létrehozni, a *ChangeLabel* publikus, mivel az osztályon belüli statikus függvényekből csak így hívható meg. Az osztály négy statikus publikus megvalósítással rendelkezik, ami használhatóvá teszi. Ezek a *ShowForm* metódus túlterhelései (overload). Mind a négy megvalósítás majdnem ugyan azt csinálja, csak különböző lehetőségeket nyújtanak. A *ShowForm()* létrehoz egy *InputForm* objektumot a privát konstruktorral. Ezt azért teheti meg, mert ez a függvény az osztályon belül van. A *Form* *ShowDialog* függvényével megjeleníti a felületet. A program addig várakozik ennél a parancsnál, amíg nem nyomjuk meg a *Form*-on vagy az „OK”, vagy a „Cancel” gombot. Ha az ablak bezárult, visszaadja az *output* változó tartalmát. A *ShowForm(string labeltext)* megvalósítás a megadott *labeltext* bemenet értékére cseréli ki az utasítást a felületen, majd elvégzi a *ShowForm* korábban említett műveleteit. A *ShowForm(IWin32Window owner)* ugyan az, mint a *ShowForm()*, csak megszabható az ablak tulajdonosa, például a *Form1*. A *ShowForm(IWin32Window owner, string labeltext)*-el pedig létrehozható egy saját utasítással, és tulajdonossal rendelkező ablak.

```
public static string ShowForm(IWin32Window owner, string labeltext)
{
    InputForm form = new InputForm();
    form.ChangeLabel(labeltext);
    form.ShowDialog(owner);
    return form.output;
}
```

21. kódrészlet - *ShowForm* függvény

Befejezés

A munka megírása során megismerkedtünk a maximális folyam probléma és a hálózati gráfok fogalmával, elemeztük a Ford-Fulkerson, az Edmonds-Karp és a Boykov-Kolmogorov algoritmusok működését, több megvalósítást is szemügyre vettünk mind elméleti, mind gyakorlati szempontokból. A munkát két fő részre bontottuk.

Az első részbe megismerkedtünk az alapfogalmakkal és az algoritmusokkal kapcsolatos főbb információkkal. Boncolgattuk a gráf, mint fogalom jelentését. Megismerkedtünk a Ford-Fulkerson, az Edmonds-Karp és a Boykov-Kolmogorov algoritmusok közötti különbséggel.

A második részben a program gyakorlati megvalósításáról esett szó. Átvettük a program osztályszintű felépítését, metódusainak és eseménykezelőink célját. Szemügyre vettünk fontosabb hibákat a megvalósításban. A munkában leírt megvalósítások nem mind hibátlanok. Az általunk létrehozott szoftver megvalósítás képes hálózati folyamatok megjelenítésére, azok módosítására és mentésére későbbi betöltés céljából. Ezeken a hálózati gráfokon biztosít lehetőséget maximális folyam szimulációk futtatására. A szimuláció lépésről lépésre is futtatható, így a felhasználó megértheti azoknak működését. A megvalósítás olyan környezetet biztosít a felhasználók számára, amely szemléltető eszköze lehet a maximális folyam problémát megvalósító algoritmusok működésének. Sok lehetőséget hagytunk a szoftver osztályaiban és függvényeiben a továbbfejlesztésre. Ilyen például a kapcsolatoknál használt *GenControlPoints* függvény is, ami úgy fejleszthető tovább, hogy létrehozhatunk egy olyan megvalósítást, ami biztosítja, hogy ha átfedésbe kerül a görbével egy csúcspont, akkor azt ki lehessen kerülni. A program megalkotása közben a C# programozási nyelv sok hasznos és új funkcióját is alkalmaztuk, amit az olvasók szíves figyelmébe ajánlunk.

Resumé

A munka olyan szoftver létrehozásával foglalkozik, amely maximális folyam problémát megvalósító algoritmusokat vizualizál. A program egy felületet biztosít, amin a felhasználó képes gráfokat létrehozni. Az elkészített gráfokat fájlba lehet menteni, hogy később újra be lehessen őket olvasni. A felületen elkészített gráfokon maximális folyam szimulációt lehet futtatni Ford-Fulkerson, Edmonds-Karp és Boykov-Kolmogorov algoritmusokkal. Ezt a felhasználó megteheti egyszerre, vagy lépésenként is. A felhasználó a szoftver használatának segítségével megismerkedhet az imént említett algoritmusok működésével, vagy kiszámolhatja vele a gráfok maximális folyamának értékét illusztrációval. A munka tartalmazza a hálózati gráf koncepcióját, általános műveleteit, és a „backend” oldalról pedig lehetséges tárolási formáit is leírja. Megismerkedtet a Ford-Fulkerson, az Edmonds-Karp és a Boykov-Kolmogorov algoritmusokkal és a köztük lévő különbséggel. Melyek között az volt a leglényegesebb különbség, hogy amíg a Ford-Fulkerson egy verem alapú tárolási megközelítést alkalmaz a felfedezett csúcspontok tárolására, addig az Edmonds-Karp egy sort használ. A Boykov-Kolmogorov pedig két keresési fát épít ki a hálózati gráf két végéből a Ford-Fulkerson és az Edmonds-Karp algoritmussal szemben, melyek csak egyet a forrás csúcspontból. A verem alapú megvalósítással történő keresést mélységi keresésnek, a soros megvalósítással történőt pedig szélességi keresésnek nevezik. A szoftver megvalósításában több lehetőség is van a továbbfejlesztésre.

ACM CCS klasszifikáció:

- Software and its engineering~Software creation and management~Designing software**
- Theory of computation~Design and analysis of algorithms~Graph algorithms analysis~Network flows
- Mathematics of computing~Discrete mathematics~Graph theory~Graph algorithms**

Idézett forrásmunkák

Alpert, Sophie. 2009. C# - Rotate a point by another point in 2D - Stack Overflow. *Stack Overflow*. [Online] 2009. 4 24. [Hivatkozva: 2024. 03 27.] <https://stackoverflow.com/questions/786472/rotate-a-point-by-another-point-in-2d>.

An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. **Boykov, Yuri és Kolmogorov, Vladimir. 2004.** 9, hely nélk. : IEEE Transactions on PAMI, 2004., 26. kötet. 1939-3539.

Copsey, Reed. 2011. C# - Using Canvas in Windows Form .net application - Stack Overflow. *Stack Overflow*. [Online] 2011. 1 17. [Hivatkozva: 2024. 04 01.] <https://stackoverflow.com/questions/4716379/using-canvas-in-windows-form-net-application>.

Fischer, Quirin. 2015. Ford-Fulkerson Algorithm. [Online] IDP Project of Quirin Fischer at Chair M9 of Technische Universität München, 2015. [Hivatkozva: 2024. 04 01.] https://algorithms.discrete.ma.tum.de/graph-algorithms/flow-ford-fulkerson/index_en.html.

GeeksforGeeks. 2024. Breadth First Search or BFS for a Graph. *GeeksforGeeks*. [Online] GeeksforGeeks, 2024. 9 26. [Hivatkozva: 2024. 12 16.] <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>.

—. **2024.** Depth First Search or DFS for a Graph - GeeksforGeeks. *GeeksforGeeks*. [Online] 2024. 02 16. [Hivatkozva: 2024. 04 01.] <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>.

—. **2023.** Ford-Fulkerson Algorithm for Maximum Flow Problem - GeeksforGeeks. *GeeksforGeeks*. [Online] 2023. 6 1. [Hivatkozva: 2024. 03 29.] <https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/>.

Hajba, Tamás. 2006. *Folyamok - előadás*. Hajba Tamás, Győr : Széchenyi Egyetem, 2006.

Heller, Jamie. 2017. Flow Network Theory using Edmonds-Karp Algorithm. *github*. [Online] Tufts University, 2017. [Hivatkozva: 2024. 04 01.] <https://jamieheller.github.io/theory.html>.

help-info.de. 2016. How to display chm file in c# winforms application - Stack Overflow. *Stack Overflow*. [Online] 2016. 4 15. [Hivatkozva: 2024. 04 01.] <https://stackoverflow.com/questions/36639348/how-to-display-chm-file-in-c-sharp-winforms-application>.

IBE Software. 2023. How to create a CHM help file: use a modern help authoring software instead of the outdated HTML Help Workshop | HelpNDoc. *HelpNDoc*. [Online] IBE Software, 2023. 06 04. [Hivatkozva: 2024. 04 01.] <https://www.helpndoc.com/news-and-articles/2023->

01-04-how-to-create-a-chm-help-file-use-a-modern-help-authoring-software-instead-of-the-outdated-html-help-workshop/.

Javatpoint. 2021. DFS Algorithm - javatpoint. *javatpoint*. [Online] 2021. [Hivatkozva: 2024. 03 29.] <https://www.javatpoint.com/depth-first-search-algorithm>.

kdbanman. 2015. C# - Forms not responding to KeyDown events - Stack Overflow. *Stack Overflow*. [Online] 2015. 12 8. [Hivatkozva: 2024. 04 01.] <https://stackoverflow.com/questions/3172731/forms-not-responding-to-keydown-events>.

Microsoft. 2024. BitArray Class (System.Collections) | Microsoft Learn. *Microsoft Learn*. [Online] Microsoft, 2024. [Hivatkozva: 2024. 04 01.] <https://learn.microsoft.com/en-us/dotnet/api/system.collections.bitarray?view=net-8.0>.

Rocky Mountain Computer Consulting, Inc. 2023. C# Helper: Draw a smooth curve in WPF and C#. *C# Helper*. [Online] Rocky Mountain Computer Consulting, Inc., 2023. [Hivatkozva: 2024. 03 24.] https://www.csharpHelper.com/howtos/wpf_draw_curve.html.

Rodricks, Robin. 2013. C# - Event when a window gets maximized/un-maximized - Stack Overflow. *Stack Overflow*. [Online] 2013. 5 18. [Hivatkozva: 2024. 04 01.] <https://stackoverflow.com/questions/1295999/event-when-a-window-gets-maximized-un-maximized>.

Rosen, Kenneth H. 2019. *Discrete mathematics and its applications* | *Eighth edition*. New York : McGraw Hill, 2019. 978-1-259-67651-2.

user1228. 2012. C# - How to draw a smooth curved line in WPF? - Stack Overflow. *Stack Overflow*. [Online] 2012. 5 7. [Hivatkozva: 2024. 03 24.] <https://stackoverflow.com/questions/10486906/how-to-draw-a-smooth-curved-line-in-wpf>.

Ábrajegyzék

1. ábra - Gráf tárolása fájlban.....	9
2. ábra - Kapcsolat végpontkoordinátáinak képlete	15
3. ábra - Párhuzamos kapcsolatok végpontkoordinátáinak képlete.....	16
4. ábra - ToolStripMenu menüszalag	30
5. ábra - defaContext helyi menü	30
6. ábra - nodeContext helyi menü	31
7. ábra - splineContext helyi menü.....	31
8. ábra - A vezérlést megvalósító gombok	31
9. ábra - A Form1 felülete.....	33
10. ábra - InputForm felülete.....	33
1. egyenlet - A csúcspont címének X koordinátája	11
2. egyenlet - A csúcspont címének Y koordinátája.....	12
3. egyenlet - Síkbeli forgatás képletei	17
1. kódrészlet - A Node osztály konstruktora	11
2. kódrészlet - EventHandler kezelése osztályok szintjén.....	11
3. kódrészlet - GenControlPoints függvény	13
4. kódrészlet - Points tulajdonság megvalósítása	13
5. kódrészlet - Snap függvény	14
6. kódrészlet - Arrow függvény.....	16
7. kódrészlet - Rotate2D függvény.....	17
8. kódrészlet - Csúcspont- és betűméret.....	18
9. kódrészlet - RenameNode függvény	20
10. kódrészlet - RemoveNode függvény	21
11. kódrészlet - NewNode függvény	22
12. kódrészlet - CheckNodesInside függvény.....	23
13. kódrészlet - MaxFlow függvény	26
14. kódrészlet - DFS függvény.....	26
15. kódrészlet - Boykov_Kolmogorov függvény.....	27
16. kódrészlet - SetStreamValues_StepByStep függvény	28
17. kódrészlet - MouseMoveOnCanvas függvény	29
18. kódrészlet - Scale2D függvény	30
19. kódrészlet - Útmutató meghívása fájlból	30
20. kódrészlet - ProcessCmdKey felülírása.....	32
21. kódrészlet - ShowForm függvény	34