Eötvös Loránd Tudományegyetem

Informatikai Kar

Algoritmusok és Alkalmazásaik Tanszék

Dijkstra és a Bellman-Ford FIFO algoritmusok működésének szemléltetése

Dr. Ásványi Tibor

Egyetemi docens

Balázs Attila

Programtervező Informatikus BSc

Budapest, 2017.

1. Tartalomjegyzék

2. Bevezetés

A gráfok rendkívül gyakran használt adatstruktúrák és a gráfokkal foglalkozó algoritmusok alapvető szerepet játszanak a számításokban. Számos probléma fogalmazható meg a gráfok segítségével. Ezek közül talán az egyik legjelentősebb a legrövidebb utak problémája. Példa: Hogyan jussunk el leggyorsabban az egyik városból a másik városba.

A probléma sokrétűsége miatt a megoldásra is számos algoritmus született. A szakdolgozatom legfőbb célja, hogy bemutassam ezen eljárások legnépszerűbbjeit, rámutatva előnyeikre, hátrányaikra valamint a közöttük lévő legalapvetőbb különbségekre, hasonlóságokra.

Elsőként alapvető, a szakdolgozat megértéséhez elengedhetetlen gráfelméletbeli fogalmakról esik szó, utána pedig magát a problémákat definiáljuk. Ezek után következik az algoritmusok részletesebb elmagyarázása. Majd megmutatom, hogyan lehet őket implementálni az általam választott programozási nyelvbe.

Ezen kívül, a matematikai jelleg elkerülése végett a dolgozatban helyet kap történelmi háttér, valamint az algoritmusok szülőatyjainak főbb életmozzanatai, egyéb szerepük a matematikában.

3. Felhasználói Dokumentáció

3.1. Platform és eszközök

Az alkalmazás **Windows Presentation Foundation**[1] (WPF, kódnevén Avalon) keretrendszerben készült. A rendszert mely grafikus felhasználói felületek készítéséhez használatos osztálykönyvtár, a **Microsoft** fejlesztette. A WPF a .NET keretrendszer 3.0 verziójában jelent meg, kialakításában jelentősen különbözik a korábban azonos célú megoldástól, a Windows Forms-tól. A WPF egyik fő újítása a korábbi ablak tervező megoldáshoz képest a felület és az üzleti logika szétválasztása. az ablakok tervezése egy XML alapú jelölőnyelvvel, a XAML-lel történik. A jelölőnyelv, illetve a feladatok szétválasztása a programozók és látványért felelős munkatársak könnyebb együttműködését teszi lehetővé. A WPF alkalmazások grafikai elemei vektorgrafikusa, ezáltal lehetővé teszik az esztétikus átméretezést, és lényegesen kevesebb tárterületet foglalnak. A WPF, és a kisebb halmazának tekinthető Microsoft Silverlight egyaránt elérhetővé teszi az animációk deklaratív definiálását, és a megjelenítéshez a GDI mellőzésével DirectX-et használ, amellyel, sokkal jobb teljesítmény érhető el. Az adatkötés a felhasználói felület és az üzleti logika között teremti meg a kapcsolatot. Ez több úton-módon történhet, az egyirányú, egyszeri kapcsolattól a kétirányú szinkronban tartott kapcsolatig.

Elsősorban **Windows 10** operációs rendszerre fejlesztettem az alkalmazást.

A WPF-ben való fejlesztést a **Microsoft Visual Studio 2010** fejlesztőkörnyezet óta biztosítja a Microsoft, de én **Microsoft Visual Studio 2015**-ben írtam meg a programomat. Előnyei a tervezési időben történő kód elemzés és hibajelzés, jól elrendezett és átgondolt kezelőfelület, projektek menedzselése, és egyszerű gyors használat. Az alkalmazást nem kell telepíteni, csak a futtatható állományt kell elindítani, így nem előfeltétele, hogy a fejlesztőkörnyezet a számítógépünkön legyen.

A Visual Studio legújabb változata elérhető ingyenesen a Microsoft oldaláról (Community Edition). Az egyetem és Microsoft kapcsolatát használva a hallgatóknak ingyen elérhető a teljes verzió is.

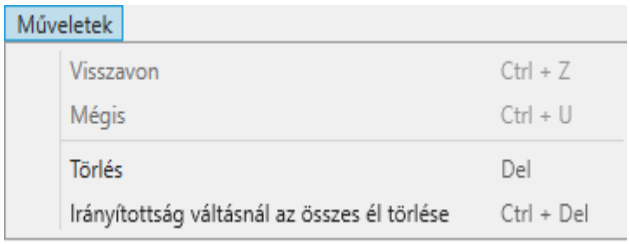
3.2. Program indítása

A programot telepíteni nem kell, a futtatható állományt a mellékelt DVD-ről fel kell másolni az eszközünkre, és indítható is.

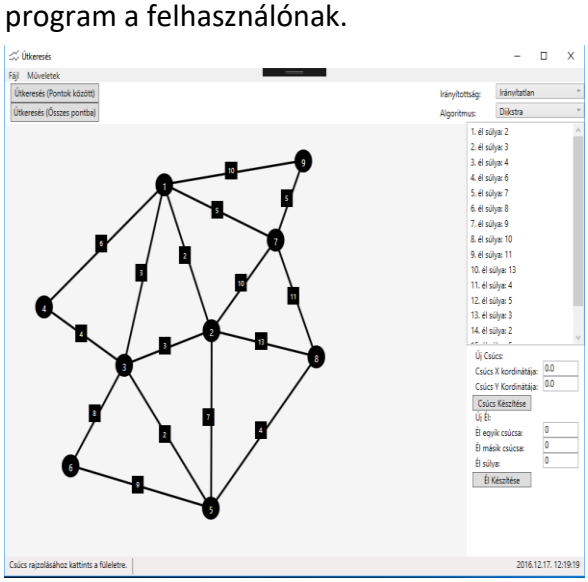
Az alkalmazás, a futtatható állomány elindítása után tárul elénk. A legnagyobb felületen tudjuk megrajzolni a gráfunkat, amin tudjuk futtatni a gráf algoritmusokat. Az alkalmazásból egyszerűen ki lehet lépni, de ilyenkor a munkánk is elveszik. Alapértelmezetten nem csinál mentést a legutolsó állapotról, mindig érdemes elmenteni mielőtt kilépünk.

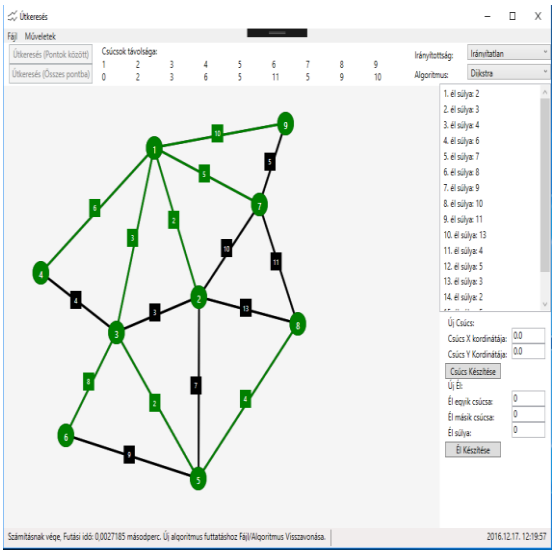
3.3. Alkalmazás felülete

Az alkalmazás feladata, hogy segítse a felhasználót egytől harminc csúcsig terjedő gráfokat kirajzolnia a gépen, és ezeken három különböző algoritmust is tudjon futtatni. A rajzolófelületen a bal gomb lenyomásával rakhatunk le egy csúcsot. Két csúcs kiválasztása után hozhatunk létre egy élt, aminek meg kell adni a súlyát. A jobb gomb lenyomásával áthelyezhetünk egy elemet a felületen. Létrehoztam egy menüsort, amiben könnyebben el lehet érni a különböző parancsokat. A fájl menüponton belül lehet új gráfot kezdeni, az algoritmust visszavonni, a gráfot menteni, és betölteni, valamint kilépni. A műveletek menüponton belül lehet az utolsó lépésünket visszavonni vagy megismételni. Esetleg ha egy csúcsot vagy élet véletlenül vittünk fel vagy csak nem szeretnénk, hogy ott legyen ki is lehet törölni. Ilyenkor a csúcshoz vagy élhez tartozó összes adat visszavonhatatlanul törlődik.



Magán a felületen még elérhető két gomb. Ezekkel lehet elindítani az útkereső algoritmust. Az egyik egy pontból a másikba, a másik egy pontból az összesbe futtatja az algoritmust. Ki lehet választani, hogy a gráfunk irányított vagy irányítás nélküli legyen, és hogy melyik algoritmus fusson. Emellett látszik az a táblázat, amelyik tartalmazza az egy csúcsból az összeshez jutó távolságot, ha egy csúcshoz nem vezet út, azt ebben a táblázatban egy végtelen karakterrel jelezzük a felhasználó felé. Ez minden alkalommal eltűnik, ha további szerkesztéseket végzünk a gráfon. Alatta egy lista van, amiben megtalálható az összes élnek a súlya. Ezeket ki lehet választani, és felülírni a korábbi súlyt. Következő részen található a billentyűzetről való bevitel. Itt hozzá lehet adni új csúcsot, és élt. A megfelelő ellenőrzések benne vannak, és ezeket jelzi is a felhasználó felé. Végül a felület legalján találunk egy státuszsort, ahol alapinformációkat ad a program a felhasználónak.

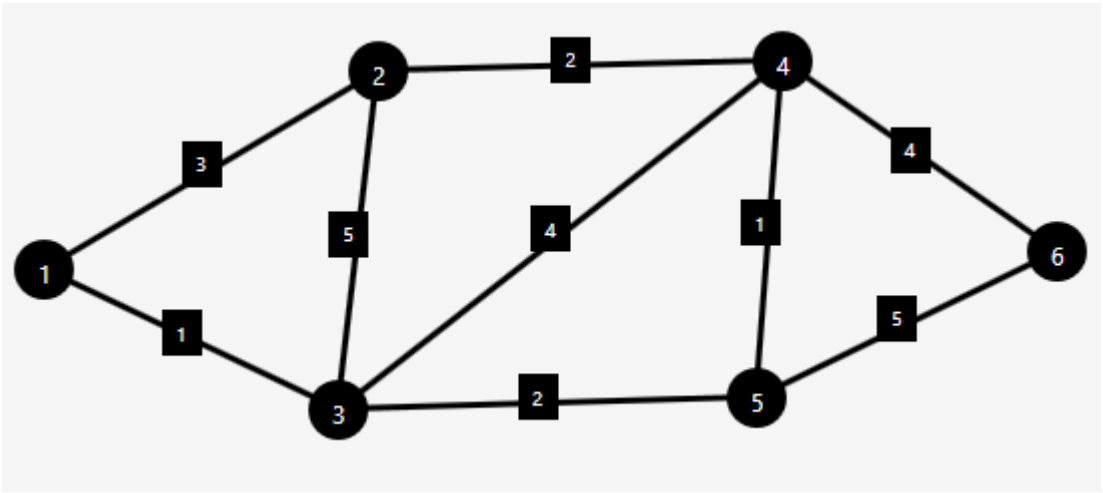




3.4. Gráf

Először beszéljünk a gráfokról, hogy érthető legyen mit is csinál a program.

Definíció (Gráf): Egy G gráf két halmazból áll: a csúcsok vagy pontok V halmazából, mely egy véges, nem üres halmaz, és az élek E halmazából, melyeknek elemei bizonyos V-beli párok.



3.5. Csúcsok

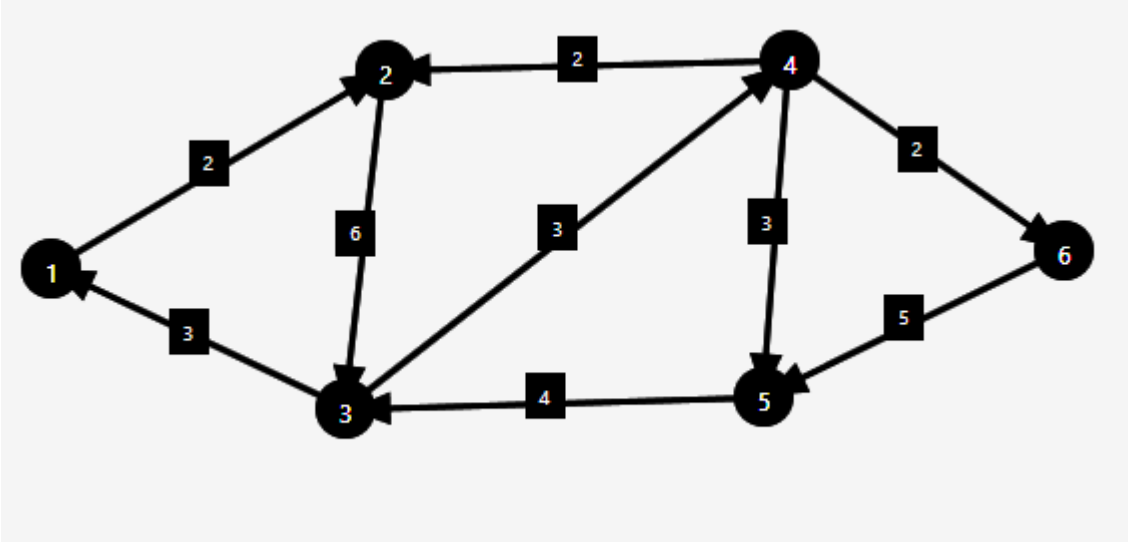
Az én esetemben a csúcsok gráfokban egy-egy pontként jelennek meg. Egy gráfban véges sok csúcs lehet. Ezeket kötik össze az élek. Minden csúcsnak van egy középpontja, egy egyedi azonosítója, össztávolsága, egy éllistája, amiben tároljuk, az i $\in$ V csúcs listájában tároljuk az i-ből kimenő éleket, és (amennyiben vannak) a súlyukat is. A listák összességét egy tömbben tároljuk. Ezek alapján egyszerűen megtudhatjuk, hogy ennek a csúcsnak, kik a szomszédjaik, ezzel is egyszerűbbé téve az algoritmus működését.

3.6. Élek

Az élek helyezkednek el két csúcs között. Egy csúcsból több él is jöhet. Egy él tartalmazza azt a csúcsot, amiből indul, és amibe megy, valamint a súlyát. Ezek nagyon fontos tulajdonságok, hogy ki tudjuk számolni egy gráfnak a legrövidebb útját. Mivel két pont

között végtelen sok pont van, ezért nem tudjuk ellenőrizni a hosszát, így egy élnek a középpontját is tárolom. Itt jelenítem meg, hogy mekkora is a súlya az élnek.

Irányított Gráf: Az irányított gráfban minden él irányított (másképp fogalmazva a csúcsok rendezettek). Az irányítást nyilak segítségével jelöljük. Az (u, v) irányított él jelölésére használni fogjuk az u → v változatot is.



Amennyiben nincs irányításunk, vagy ha minden él oda és vissza is irányítva van, akkor irányítatlan gráfról beszélünk. Ekkor nem teszünk különbséget az (u, v) és a (v, u) élpár között.

Mint ahogy már fentebb is utaltam rá, az objektumok (csúcsok) közötti kapcsolat sokszor jelentheti út létezését vagy kommunikáció lehetőségét. Ilyenkor gyakran költségek vagy súlyok tartozhatnak az élekhez, amelyek az út esetében időt vagy akár pénzt is jelenthetnek (gondoljunk csak az autópályákra, amelyek használatáért fizetni kell). Ezt a kapcsolatot egy valós értékű függvénnyel fogjuk leírni, melynek értelmezési tartománya a gráf élhalmaza, az értékkészlete pedig a valós számok halmaza és k-val, a költség szó kezdőbetűjével jelöljük, tehát k: E→$\mathbb{R}$. A súlyokat az élekre szokás írni.

Definíció (út): Egy út – akár irányított, akár irányítatlan gráfban – csúcsok olyan v_1 , … , v_k sorozata (lehet egyelemű is), ahol ∀i ∈ 1..k-1-re (v_i,v_{i+1}) a gráf éle. Irányított gráfoknál az u-ból v-be menő útra $u \rightarrow$ v jelöléssel is fogunk utalni.

Definíció (út hossza): Legyen adott egy G = (V, E) irányított vagy irányítatlan gráf a k(f), f ∈ E élsúlyokkal. A G gráf egy u-ból v-be menő útjának hossza az úton szereplő élek súlyának összege.

Legrövidebb út

Definíció (legrövidebb út): Legrövidebb út alatt a gráfelméletben egy minimális hosszúságú utat értünk egy gráf két különböző *u* és *v* csúcsa között. Amennyiben a gráfunk éleihez nem tartoznak súlyok, akkor ez egyet jelent egy olyan úttal *u* és *v* csúcs között, amelyben a legkevesebb él szerepel. Ha vannak súlyok a gráf élein, akkor pedig olyan útról beszélünk, amelynek élein szereplő súlyok összege minimális. Vagyis ha adott egy G = (V, E) gráf a k(f), f ∈ E élsúlyokkal, akkor $\mathrm{d}(\mathrm{u},\mathrm{v}) = min \sum_{f\in \mathrm{P}} k(f)$, ahol P tetszőleges út *u* és *v* között.

Egyes könyvekben ez a definíciója az u és v csúcsok közötti távolságnak, illetve a legrövidebb út súlyának, ha u ≠ v, ha u = v akkor ez a távolság, illetve súly 0, ha nincs út u és v között, akkor ∞.

Az alábbi változatokra osztható a probléma:

1. Legrövidebb utak egy kiinduló pont és az összes többi pont között: meg szeretnénk találni az összes v∈V csúcshoz egy adott s∈V kezdőcsúcsból odavezető legrövidebb utat.
2. Legrövidebb út két különböző csúcs között: keressük egy adott u csúcsból egy adott v csúcsba vezető egyik legrövidebb utat. (Ez az 1. probléma egy speciális esete.)
3. Legrövidebb utak az összes többi pont és egy végpont között. (Ez az 1. megfordítása.)
4. Legrövidebb utak az összes csúcspár között: keressük az összes u és v csúcspárra egy u-ból a v csúcsba vezető legrövidebb utat. (Visszavezethető az 1. problémára, ha minden lehetséges kezdőcsúcsra megoldjuk.)

Természetesen akadhat olyan legrövidebb út probléma, amelyekben előfordulnak negatív élek.

Definíció (negatív kör): A G = (V, E) irányított gráf olyan köre, amelyekben az élek súlyának összege negatív.

A negatív körök a tőzsdén és a pénzügynél jöhetnek elő a legkevésbé, de ezekkel az algoritmusokkal az ilyen problémákat is ki lehet szűrni, hogy ne forduljon elő olyan, hogy a pénzváltórendszerben kialakuljon egy negatív kör amivel mindig nyereségesen lehet kijönni.

Amennyiben nincs a gráfban negatív kör bármelyik $v \in V$ csúcs esetén a legrövidebb út súlya jóldefiniált marad. Ha vannak u-ból elérhető negatív körök, akkor viszont a legrövidebb út súlya definiáltalan lesz, ugyanis ilyen esetben mindig van kisebb súlyú rövidebb út, ha a feltételezett legrövidebbhez hozzávesszük a negatív kör egy bejárását. Ilyen esetekben ezt a távolságot $-\infty$-nek definiáljuk. (Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, 2001)

3.7. Az algoritmusok

A Dijkstra algoritmus

Dijkstra algoritmusa egy adott kezdőcsúcsból az összes többi csúcsba vezető legrövidebb utak problémáját egy súlyozott, irányított G = (V, E) gráfban, abban az esetben oldja meg, ha nincsenek negatív élek, vagyis ha minden $u \rightarrow v$ élre $k(u, v) \geq 0$ teljesül.

Az algoritmust Edsger Wybe Dijkstra, holland matematikus és informatikus alkotta meg 1956-ban.

Dijkstra algoritmusa szomszédossági éllistával

A Dijkstra algoritmus implicite azoknak a csúcsoknak az *S* halmazát tartja nyilván, amelyekhez már meghatározta az *u* kezdőcsúcsból odavezető legrövidebb út súlyát. Az algoritmus minden lépésben a legkisebb legrövidebb-út becslésű $x \in V - S$ csúcsot választja ki, beteszi az *x*-et S-be, és minden *x*-ből kivezető éllel egy-egy közelítést végez. A *V-S* csúcsok nyilvántartására egy *Q* prioritásos sort alkalmazunk minimum kiválasztásra, ahol a legkisebb d érték számít a legnagyobb prioritásúnak.

DIJKSTRA eljárás.

```
1.  procedure Dijkstra (G, u)
2.  begin
3.      for minden v ∈ V csúcsra do
4.      begin
5.          D[v] := ∞;
6.          szülő[v] := Ø;
7.      end
8.      D[u] := 0;
9.      var
10.         S := Ø;
11.         Q := csúcsokból álló sor;
12.     begin
13.         while Q nem üres do
14.         begin
15.             x := minimális Q;
16.             S := S ∪ {x};
17.             for ∀ v ∈ Szomszéd[x]-re do
18.             begin
19.                 if D[v] > D[x] + k(x,v) then
20.                 begin
21.                     D[v] := D[x] + k(x,v);
22.                     szülő[v] = x;
23.                 end
24.             end
25.         end
26.     end
27. end
```

A Dijkstra algoritmus mohó stratégiát alkalmaz, hiszen mindig az u startcsúcshoz „legközelebbi” csúcsot választja ki *V*-*S*-ből, hogy azután az *S* halmazba tegye.

A Bellman-Ford-algoritmus

A Bellman-Ford-algoritmus az adott kezdőcsúcsból induló legrövidebb utak problémáját abban az esetben oldja meg, amikor vannak az élek között negatív súlyú élek, de nem találunk a gráfban negatív kört.

Az algoritmus szülőatyjaként Richard Bellman-t és ifjabb Lester Randolph Ford-ot tisztelik.

Az eljárás

Adott egy k: $E \rightarrow \mathbb{R}$ súlyfüggvénnyel súlyozott irányított *G = (V, E)* gráf, ahol a kezdőcsúcs az *u*. Az algoritmus visszajelzi, ha van a gráfban *s*-ből elérhető negatív kör, ha nincs benne, akkor előállítja a megoldást.

BELLMAN-FORD eljárás.

```
1.  procedure Bellman-Ford (G, u)
2.  begin
3.      for ∀ v ∈ V csúcsra do
4.      begin
5.          D[v] := ∞;
6.          szülő[v] := Ø;
7.      end
8.      for i=1 to n–1 do
9.      begin
10.         for ∀ (u,v) élre do
11.         begin
12.             if D[v] > D[u] + k(u,v) then
13.             begin
14.                 D[v] := D[u] + k(u,v);
15.                 szülő[v] := u;
16.             end
17.         end
18.     end
19.     for ∀ (u,v) élre do
20.     begin
21.         if D[v] > D[u] + k(u,v) then
22.         begin
23.             return HAMIS;
24.         end
25.     end
26.     return IGAZ;
27. end
```

Az 3-7. sorok a kezdeti értékek beállítását mutatják. Aztán a 8-18. sorok ugyanazt az ellenőrzést végzik el, mint a Dijkstra algoritmus esetében, végül a 19-26. sorok keresnek negatív köröket.

A Bellman-Ford FIFO algoritmus

A Bellman-Ford FIFO egy gyorsított eljárása a Bellman Fordnak. Adott egy tetszőleges hálózat, amiben az utak költségei között vannak, ezt a hálózatot egyszerű gráffal ábrázoljuk *G = (V, E)*, ahol a kezdőcsúcs az *u*.

Ezt az algoritmust Robert Endre Tarjánnak köszönhetjük.

Az eljárás

BELLMAN-FORD FIFO eljárás.

```
1.  procedure Bellman-Ford FIFO (G, u)
2.  begin
3.      for ∀ v ∈ V csúcsra do
4.      begin
5.          D[v] := ∞; szülő[v] := Ø; state[v] := unlabeled;
6.      end
7.      D[u] := 0; e[u] := 0;
8.      var
9.          Q := csúcsokból álló sor;
10.     begin
11.         while Q nem üres do
12.         begin
13.             s := deQueue(Q); state[s] := scanned;
14.             for ∀ v ∈ (s,v) élre do
15.             begin
16.                 if D[v] > D[s] + k(s,v) then
17.                 begin
18.                     D[v] := D[s] + k(s,v); szülő[v] := s; e [v] := e[s] + 1;
19.                     if e[v] < n then
20.                     begin
21.                         if state[v] /= labeled then
22.                         begin
23.                             enQueue(Q,v); state[v] := labeled
24.                         end
25.                     end
26.                     else
27.                     begin
28.                         return v;
29.                     end
30.                 end
31.             end
32.         end
33.         return NIL;
34.     end
35. end
```

Az 3-7. sorig az algoritmushoz tartozó változókat inicializáljuk a számukra megfelelő értékekkel. A d tömb tárolja a távolságokat, a p tömb tárolja, hogy a csúcsot melyik csúcsból értük el, és a state tömbre szükségünk van ellenőrizni, hogy benne van-e a sorban az elem. A 8-18. sorig folyik az ellenőrzés, hogy melyik a legrövidebb út. A 19-29. sorban ellenőrizzük, hogy van-e negatív kör a gráfban, ha van, akkor visszaadjuk azt a csúcsot, ami a negatív körben helyezkedik el, ha NIL-t adunk vissza, akkor sikeresen lefutott az algoritmus.

4. Fejlesztői Dokumentáció

4.1. Feladat

Az alkalmazásom célja, hogy a felhasználó könnyebben meg tudja érteni a Dijkstra, Bellman-Ford és Bellman-Ford FIFO legrövidebb utakat kereső algoritmusok működését. Ebben segítséget nyújt, hogy meg tudja rajzolni egy felületen, és az általa megszerkesztett gráfokon tudja futtatni az algoritmusokat.

4.1.1. Látványterv

A kezelést egy menüsorral teszem könnyebbé. A képernyő bal felső sarkában található egymás mellett két lenyíló menü. Ezeket lenyitva tárulnak elénk az alkalmazás alapfunkciói. A menügombokat lényegre törő névvel, és gyorsbillentyűkkel látom el, ezzel gyorsabbá téve a használatát. A fájl menüpontban új gráfot, és a futtatott algoritmus visszavonását csinálhatjuk. Itt tudjuk elmenteni a gráfunkat, vagy egy korábban felrajzolt gráfot visszatölteni, és bezárni az alkalmazást. A műveletek menüpontban lehet visszavonni az utolsó felrajzolt csúcsot vagy élt. Valamint, ha visszavontuk a gráf egy részét, de mégis szükségünk van rá, akkor azt vissza tudjuk rajzoltatni a mégis gombbal. Ide kerül még fel egy kiválasztott csúcs vagy él törlése, és az, hogyha szeretnénk, akkor minden irányítottság váltásnál törlődhet az összes él.

A fő képernyőn jelenik meg a rajzoló felület, a két gombbal, amivel meg lehet indítani az algoritmus számításokat, a két **combobox**, amikkel ki lehet választani a gráf irányítottságát, és hogy melyik algoritmus fusson a gráfon, középen fent jelenik meg egy táblázat, ha egy csúcsból keressük az összes többi csúcsba távolságot. Alul a Status Bar ad információt arról, hogy éppen mit csinálhatunk, és ha csak két pont közötti távolságot számítottunk, akkor itt jelenik meg az út hossza. A jobb oldalon található egy **listbox**, amiben tároljuk az összes élnek a súlyát, alatta pedig a billentyűzetről felvihető csúcs és él beviteli mezők.

4.1.2. Szöveges fájl leírása

A legtöbb adatot a pozíciók tartalmazzák a szöveges fájlban. Egy csúcsnak a szöveges fájl eltárolja a nevét az átmérőjét, a középpontját, és azt a pontot, hogy honnan kezdje el felvinni a kör alakú ellipszist a képernyőre. A következő sorokban az élek tulajdonságai vannak, róluk tároljuk a kezdőcsúcsot, a végcsúcsot, és a súlyát. Ezeken a pontokon kívül még tároljuk, hogy hány csúcs és él került megrajzolásra, és a **combobox**-ok kiválasztott indexeit.

Minden egyes betöltésnél felvesszük az összes beolvasott csúcsot és élt újra, és ezután meghívjuk a rajzoló függvényeinket. Feltesszük, hogy jó a bemeneti fájl, és emiatt akár egy felhasználó egy fájlban is megszerkesztheti a gráfját, majd beolvasáskor a program azt is megrajzolhatja.

Maga a fájl a következő adatokat tartalmazza: Az első sorban balról haladva a csúcsok számát, az élek számát, az algoritmus **combobox** index-ét, valamint az irányítottság **combobox** index-ét. A következő sorok tartalmazzák egy-egy csúcs középpontját, a sugár megrajzolásának kezdőpontját, a csúcs elnevezését és átmérőjét. A fájl elején lévő élek számával jelzem, hogy az utolsó ennyi darab sor az éleket jelentik. Ezekben a sorokban először a kezdő csúcs, utána a vég csúcs, és harmadikként az él súlya helyezkedik el. Az adatokat külön sorokban tárolom a szöveges fájlban. A pozíciók X és Y koordinátáin kívül (amit pontosvesszővel), minden számot üres karakterrel választok el.

4.2. Platform és eszközök

Az alkalmazás jelenleg csak a Visual Studio 2015 fejlesztői környezet segítségével telepíthető. Egyelőre csak Windows operációs rendszert futtató számítógépre van lehetőségünk fordítani az alkalmazást.

Az alkalmazást Windows Presentation Foundation, .NET keretrendszerben fejlesztem. Ehhez C# nyelvet használok a Microsoft Visual Studio 2015 fejlesztőkörnyezetben. A Windows Presentation Foundation előnye, hogy sokkal folyékonyabbak az animációk, és szebbek a felületei. A legnépszerűbb programnyelvekkel (**C#**[2], C++, Javascript) programozható a működés, a megjelenést pedig **XAML** nyelvben kell megvalósítani.

A Visual Studio egyik előnye, hogy a projektet (*Solution*) több projektre is fel lehet bontani, ezzel modulárisan szeparálhatjuk a programot a főbb funkciói szerint. A legalsó modulokat (perzisztencia, adattárolás) megfelelő interfészekkel ellátva az alkalmazás később könnyedén fejleszthető lenne egyéb platformokra is, valamint könnyebben cserélhetőek lesznek az egyes modulok.

A Windows Presentation Foundation biztosít számunkra fájlkezelő funkciót. Ezt használva fogom a perzisztenciát megvalósítani. A háttérfolyamat implementálásához is nyújt megfelelő interfészeket.

A program felülete különböző ablakokra van felosztva. Minden laphoz tartozik egy XAML kód mely a lap vizuális leírását foglalja magába. Az ablakokhoz tartoznak C# osztályok is melyekkel programozhatjuk a működésüket, és a használt architektúra miatt ezek ki is lesznek használva.

A perzisztálást (adatlekezelést) az alkalmazás szöveges fájlokkal végzi. Minden gráf egy külön fájlban van eltárolva. Ez később továbbfejleszthető SQL adatbázis alapú, vagy felhő alapú adattárolássá.

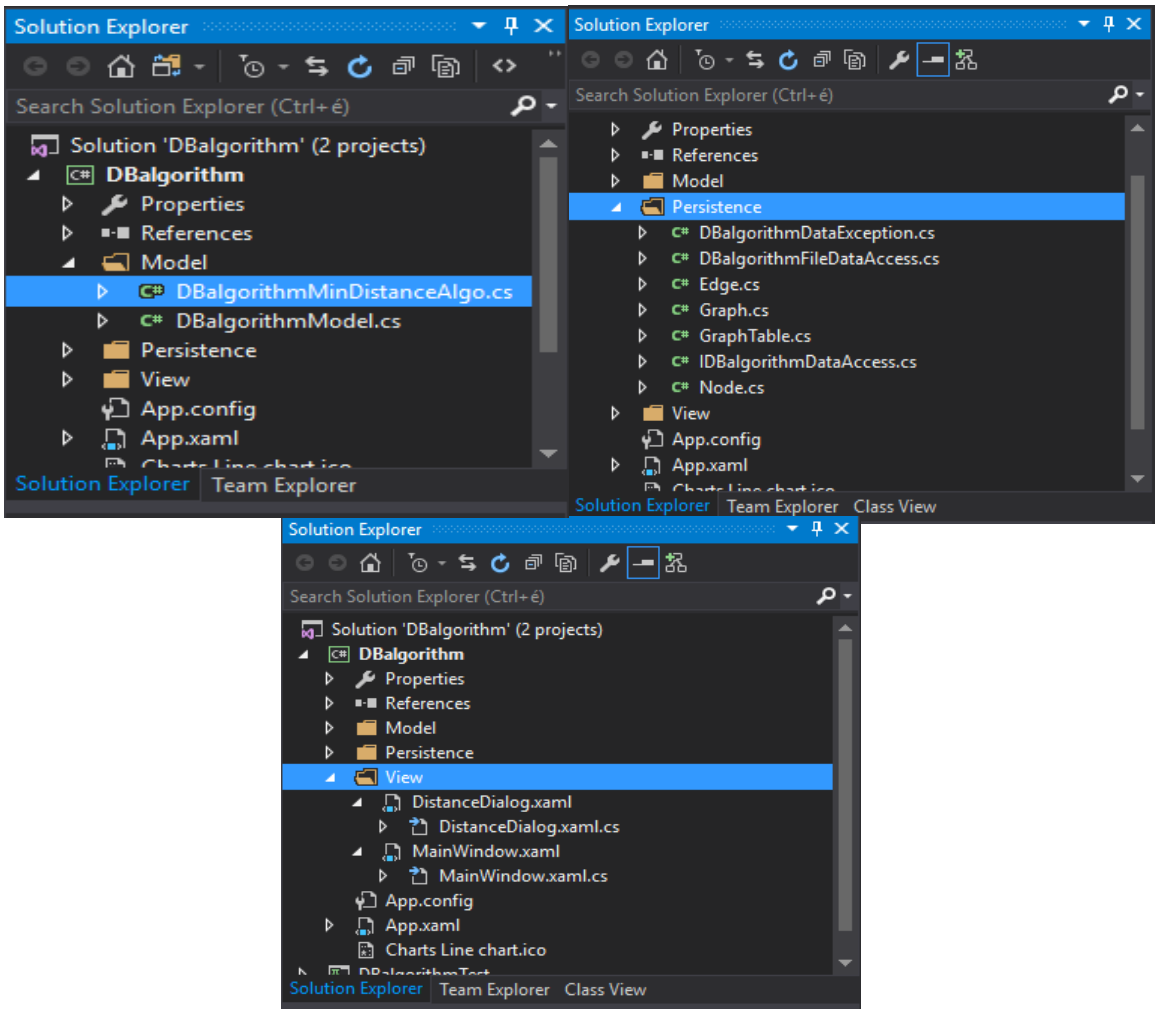
A program fejlesztését és tesztelését a saját laptopomon végzem, melyen jelenleg Windows 10 operációs rendszer fut.

4.3. Rendszerterv

4.3.1. Architektúra

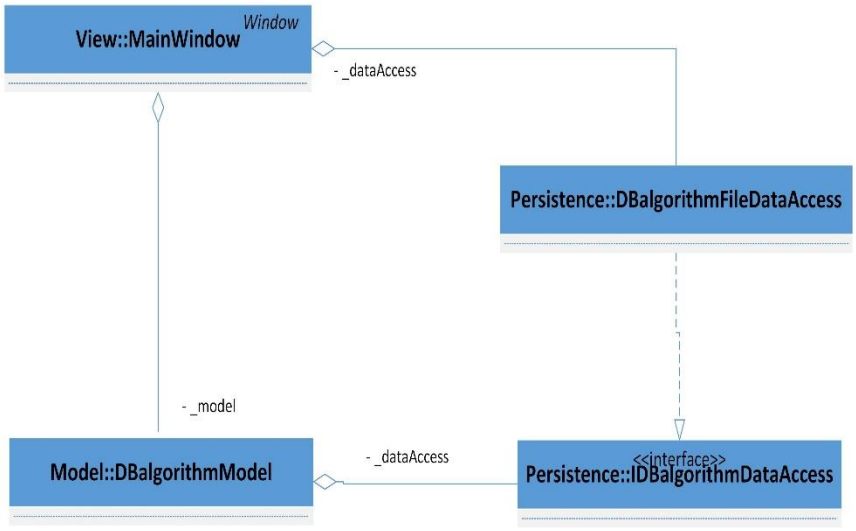
Az alkalmazásomat háromrétegű (three-tier) architektúrában fejlesztem, és ennek a szerkezeti felépítést használtam. Az adatelérés (Perzisztencia), üzleti logika (Modell) és a megjelenítés (Nézet) **külön modulok**ban fog elhelyezkedni a programrészektől. A különböző modulokat a perzisztencia, vagy adateléréstől kezdve építem fel a modellen keresztül egészen a nézetig. Az egyes rétegek között függőségek (dependency) alakulnak ki, mivel felhasználják egymás funkcionalitását. Cél a minél kisebb függőség elérése (loose coupling), ezért a függőségeket úgy kell megvalósítanunk, hogy a konkrét megvalósítástól ne, csak annak a felületétől (interfészétől) függjünk. A rétegek a függőségeknek csak az absztrakcióját látják, a konkrét megvalósítást külön adjuk át

nekik, ezt nevezzük függőség befecskendezésnek (dependency injection). A befecskendezés helye/módszere függvényében lehetnek különböző típusai (pl. konstruktor, metódus, interfész). Az architektúra szerkezetét és osztályait kézzel hoztam létre, felépítése a következő képeken látható:



A függőség befecskendezést a legfelső réteg, a nézet végzi. Jelen esetünkben a Windows Presentation Foundation a **Window** osztály egy leszármazottját, az **App** osztályt példányosítja indításkor. Az **App** osztály lesz, mely inicializálja a fő ablakot (**MainWindow**), ami a függőségeket és modulokat összeköti, és elkészíti a perzisztenciát és az üzleti logikát.

A következő képen látható, hogy az adatkezelés esetén elválasztjuk a felületet (**IDBalgorithmDataAccess**) a megvalósítástól (**DBalgorithmFileDataAccess**), utóbbit a nézet fogja befecskendezni a modellbe:



Az üzleti logikát a **DBalgorithmModel** osztály fedi le. Továbbá használja az alkalmazás perzisztenciáját is mely az **IDBalgorithmDataAccess** interfészt megvalósító **DBalgorithmFileDataAccess** osztályban definiált.

A névütközések elkerülése végett a modulok nem csak külön osztállyal, hanem saját névtérrel is rendelkeznek. Az alkalmazás legfelső névtere az **DBalgorithm** (Dijkstra and Bellman-Ford Algorithms). Ezen a névtéren belül a modulok saját névtereket valósítanak meg (például DBalgorithm.View, vagy DBalgorithm.Model).

A modulok interfészeihez is külön Osztály létrehozására van szükség. A platform független információk, definíciók és implementációk eltárolhatóak az interfészek projektjében így bármely nézethez és platformhoz hozzákapcsolhatóak viszont a platformfüggő megvalósításokat célszerű külön projektben (Class Library-ben) létrehozni így bármikor kicserélhető lesz egy másik implementációra.

4.3.1.1 Perzisztencia réteg

A Perzisztencia a program legalsó modulja, ez felel az adatok tárolásáért, tartja a kapcsolatot az adatbázissal és definiálja az adatszerkezeteket. A perzisztencia névtere **DBalgorithm.Persistence**.

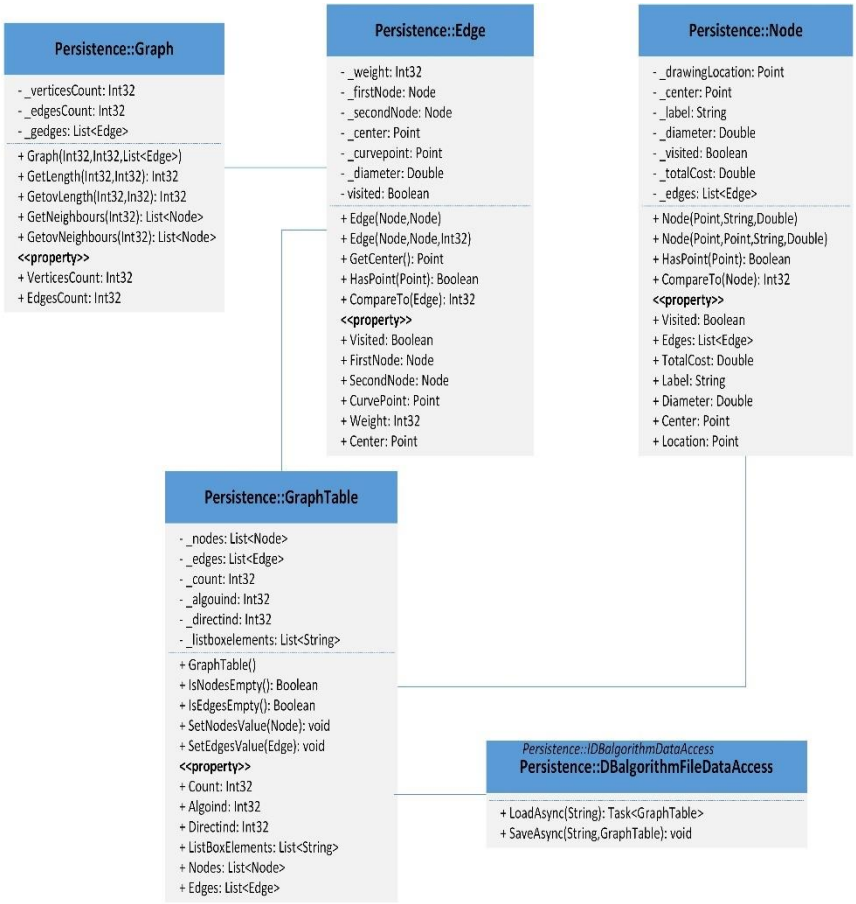
Az adatbázisban el kell tárolnunk azt, hogy hány csúcsunk és élünk van, a **combobox**-ok melyik indexen voltak a mentés pillanatában, valamint a csúcsok és az élek legfőbb

tulajdonságait. Ezeket az **adatszerkezeteket saját osztállyal definiáljuk**. Betöltéskor minden adat egy saját osztálypéldánnyal fog reprezentálódni, melyeket egy **kollekcióban** (például *List* adatszerkezetben) gyűjtünk össze és adjuk át az adatokat igénylő moduloknak. **Publikus tulajdonságokkal** hivatkozunk az adattagokra (például a csúcs nevét, koordinátáit stb.) így a **setter** és **getter** metódussal akár adatellenőrzést és feldolgozást is végezhetünk. Az adatszerkezetek rendelkeznek egy alapértelmezett konstruktorral, viszont van egy plusz paraméterrel túlterhelt konstruktoruk is. Az adatbázis szöveges fájlokba perzisztál ezért az adatszerkezeteket szöveggé alakítjuk mentéskor, és szövegből értelmezzük betöltéskor.

A perzisztencia interfészét az **IDBalgorithmDataAccess** interfész implementálja, melyben definiálva vannak a metódusok és függvények melyekkel a modell kényelmesen hozzáférhet az adatbázisban tárolt adatokhoz. A felsőbb rétegeknek nem kell foglalkozniuk a fájlok olvasásával, vagy az adatszerkezetek példányosításával. Ezeket a feladatokat a perzisztencia interfészét megvalósító osztálynak kell elvégeznie. A metódusok aszinkron hívások ezért minden visszatérési érték a **Task** generikus osztályba van becsomagolva. Névelnevezési konzisztenciát használtam, ezért az adatbetöltő függvények **Load** prefixszel, az adatmentő metódusok **Save**, az adatmódosító **Set** prefixszel kezdődnek, vagy **Empty** postfixszel végződnek. Minden adatszerkezeteket kezelő utasítás rendelkezik az **Async** posftixszel ezzel is jelezve az aszinkronitást.

Minden adatbázist a felhasználó nevezhet el, mikor létrejön, ezzel az azonosítóval hoz létre egy fájlt a belső tárhelyen. Ebben a fájlban tárolódnak a csúcsok, élek és a fő képernyő beállításai. A továbbiakban adatbázis alatt egy ilyen szöveges fájl tartalmát értem.

A perzisztencia réteg osztályai, és egymástól való függésük a következő képen látható:



4.3.1.1.1 DBalgorithmFileDataAccess osztály

A perzisztencia interfészét az **DBalgorithmFileDataAccess** osztály valósítja meg mely szöveges fájlba perzisztálja az adatokat. Az általam létrehozott típusok is ezen a szinten helyezkednek el.

- **SaveAsync (String path, GraphTable table)**

A perszitencia mentés függvényének két paramétere van, az egyik az elérési útvonala, a másik egy általunk megvalósított típus.

- **LoadAsync (String path)**

A perzisztencia betöltő függvénye, beolvassa a mentett fájl tartalmát. A függvénynek van egy paramétere, ami az útvonalat tárolja a fájlhoz. A fájlban megtalálható adatokat soronként olvassa be. Az adatok betöltéséért a **LoadAsync** függvény felel. Ez a függvény a pozíció adatokat tölti be, majd elkészíti őket a háttérben, és ezután elküldi a felületnek, ami ki fogja rajzolni a gráfot. Az csúcsok egy beépített **List** kollekciós adatszerkezetben adja vissza. Az élek betöltése is hasonló elven működik.

4.3.1.1.2 Node osztály

A perzisztencia csúcs osztályában lévő két darab konstruktorral hozhatunk létre egy csúcsot. Az egyik konstruktor csak egy új csúcs létrehozásakor kap szerepet, a másik egy csúcs áthelyezésénél fut le. Emellett itt helyezkedik el a **HasPoint (Point p)** függvény, ami az ellenőrzés, hogy a képernyőre való kattintáskor csúcsra kattintottunk-e.

4.3.1.1.3 Edge osztály

A perzisztencia él osztályában lévő két darab konstruktornak, és a **HasPoint (Point p)** a szerepe ugyanaz, mint a csúcsnál. Valamint a **GetCenter ()** függvény, ami euklideszi távolsággal kiszámolja a két pont távolságának hol a középpontja.

4.3.1.1.4 Graph osztály

Ebben az osztályban ellenőrzöm egy csúcs szomszédjait, valamint két csúcs közötti súlyt.

4.3.1.1.5 GraphTable osztály

Ezt az osztályt, a csúcsok és élek egy helyen való használata miatt hoztam létre.

Az adatkezelés során előforduló hibák kezelésére létrehoztam az **DBalgorithmDataException** osztályt mely a beépített **Exception** osztályból öröklődik. Konstruktora *string*-ként megkapja, a hibaüzenetet melyet átad az *Exception* osztály konstruktorának.

4.3.1.2 Modell réteg

A modellben lévő osztályok felelnek az alkalmazás üzleti logikájának lebonyolításáért. Adatszerkezeteket tölt be a perzisztencián keresztül, melyekkel további műveleteket végez. Menedzseli az új adatok létrehozását és a perzisztenciának átadását. Vezérli az algoritmusok futtatását, az új csúcsok, és élek hozzáadását, valamint a Windows Presentation Foundation által nyújtott extra funkciókat. A modell névtere **DBalgorithm.Model**.

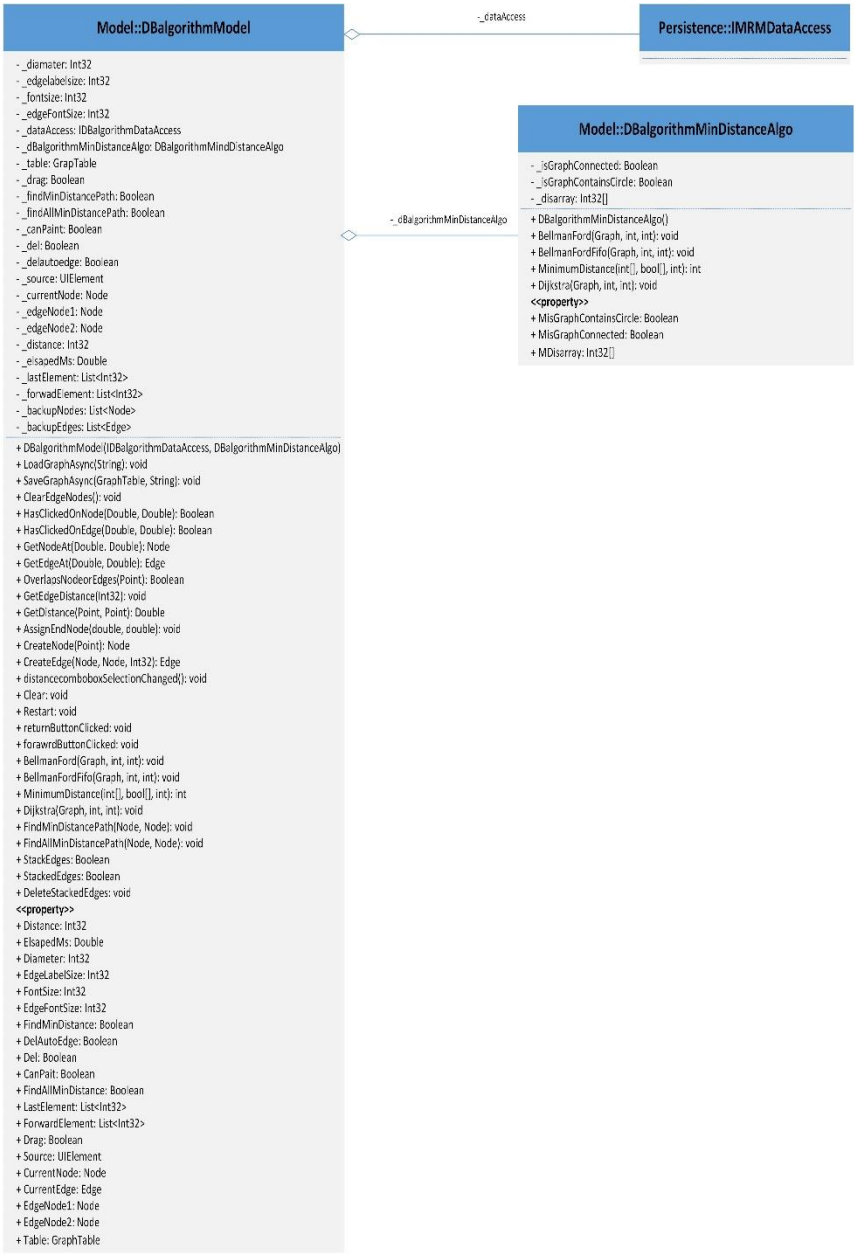
A modell cserélhetőségét nem tartottam fontosnak ezért a modell osztályhoz nem készültek interfészek.

A **DBalgorithmModel** osztály tartalmazza az alkalmazás logikájának a legfontosabb függvényeit melyekkel adatokat lehet betölteni a perzisztenciából és adatokat lehet létrehozni, szerkeszteni és frissíteni az adatbázisban. A perzisztencia felépítését és megvalósítását nem kell tudnia, elég csak az interfészt ismernie ezért, ha később a szöveges perzisztenciát lecserélnénk egy SQL adatbázis alapúra, akkor a modell osztály

változatlan maradna. A modell a felsőbb réteggel eseményeken keresztül kommunikál adatot pedig a függvények visszatérési értékével vagy publikus tulajdonságokkal adhat át. Feleslegesen nem kérdezünk le adatot a perzisztenciából, az adatok szerkesztése a memóriában a modellbe betöltött adatokon zajlik. Publikus tulajdonságokkal hivatkozunk a betöltött adatokra. Az adatokat a **Nodes**, **Edges,** és **ListBoxElements** kollekciós adatszerkezetek tartalmazzák. Nem definiáltam új adatszerkezeteket, a perzisztenciában definiáltak tökéletesen megfelelnek ezekre a célokra. A módosítások a betöltött adatokon zajlanak.

A felsőbb rétegeknek szüksége lehet a beállítások elérésére (például, ha a felhasználó módosítani vagy megtekinteni szeretné őket) ezért a beállításokhoz publikus tulajdonságokon keresztül (**Algoind, Directind**) férhetnek hozzá. Így nem szükséges mindkét beállítási opcióhoz egy áthidaló függvényt implementálni a modellben. Az adatbázisból való lekérdezésnél szükség van rá, mert az adatokat fel is kell dolgozni. Néhány feldolgozási művelet ki van emelve a modell publikus függvényeibe. Az eseményekhez tartoznak privát eseménykiváltó változók, a modell ezeket hívja meg a sikeres betöltések és lekérdezések után.

A modell réteg osztályai, és egymástól való függésük a következő képen látható:



4.3.1.2.1 DBalgorithmModel osztály

- **DBalgorithmModel (IDBalgorithmDataAccess _dataAccess, DBalgorithmMinDistanceAlgo _dBalgorithmMinDistanceAlgo)**

A modell privát mezőkben tárolja a perzisztencia példányát és a legrövidebb utak osztálynak is egy példányát. Függőség befecskendezés segítségével a példányok mutatóit a modell konstruktora paraméterként kapja meg. A konstruktorfüggvény eltárolja ezeket a mutatókat a privát mezőkben így a függőségek cseréléséhez elég csak a felsőbb szinteken módosítani.

- **LoadGraphAsync (String path)**

A modellbe a csúcsok, és élek betöltéséhez a **LoadGraphAsync** metódus meghívása szükséges mely a paraméterben kapott útvonal azonosítóhoz betölti a csúcs, és élek pozícióit. Az adatok betöltése a perzisztenciából majd feldolgozásuk után eseménnyel jelzi a modell a felsőbb réteg felé, hogy az adatok elkészültek. Ez az esemény csak mentés után váltódhat ki. A betöltött adatokat mentésig vagy felhasználó által bevitt frissítésig és újabb lekérdezésig nem töltjük újra.

- **SaveGraphAsync (String path)**

Ugyanúgy a felhasználó által megadott útvonalat és a táblát továbbítja a perzisztenciának. Módosításkor az adatok nem kerülnek automatikusan mentésre, ehhez a modellben definiált **SaveGraphAsync** metódus meghívására van szükség melyek a paraméterben megkapott adatot átadják a perzisztenciának mentési célból. Mentés után az elmentett adatkollekció újratöltődik a perzisztenciából.

- **ClearEdgeNodes ()**

A felhasználó által kiválasztott csúcsok nullázása.

- **StackEdges ()**

Ez a függvény ellenőrzi, hogy a felhasználó által kiválasztott csúcsok között létezik-e már él.

- **StackedEdges ()**

Egy irányított gráf irányítatlanná váltásánál ellenőriznünk kell, hogy vannak-e olyan csúcsok, amik között két él is húzódik.

- **DeleteStackedEdges ()**

Ha a fent említett függvény igaz lesz, akkor a felhasználó felé jelezzük, hogy törlődni fog a nagyobb súllyal rendelkező él.

- **HasClickedOnNode (Double x, Double y)** és
  HasClickedOnEdge (Double x, Double y)

A Nézettől kapott koordinátákat ellenőrzi, hogy megtalálja-e csúcsok vagy élek között.

- **GetNodeAt (Double x, Double y)** és

  GetEdgeAt (Double x, Double y)

A Nézettől kapott koordináták megnézi, hogy ténylegesen van-e ilyen csúcs, és ha megtalálta, akkor azt visszaadja.

- **OverlapsNodeorEdges (Point p)**

A program figyeli, hogy a felhasználó által kattintott területen van-e már valami. Ha talált, akkor a program nem engedi, hogy eltakarja azt a mezőt egy másik csúccsal.

- **GetEdgeDistance (Int32 _dis)**

Ezzel a függvénnyel kapja meg a modell nézet egy él súlyát.

- **GetDistance (Point p1, Point p2)**

Ebben a függvényben a felhasználó által kiválasztott két csúcs közötti középpont számolása történik, ami átadásra kerül egy él elkészítésekor, mivel ezen a koordinátán lesz egy él súlyának a **TextBlock**-a.

- **AssignEndNodes (double x, double y)**

Két csúcs kiválasztásánál fut ez a függvény. Ellenőrzésre kerül a kapott koordináta csúcs-e, és ezután elmentésre kerül a készülő él egyik végpontjaként.

- **CreateNode (Point p)** és

  CreateEdge (Node node1, Node node2, Int32 distance)

A csúcsok, és élek betöltődésekor csak a háttérben futó tulajdonságok töltődnek fel, a kirajzolás nem itt történik meg.

- **directedComboBoxSelectionChanged ()**

Irányítottság váltásánál ez a függvény fog lefutni, ami opcionálisan akár törölheti az összes élt.

- **Clear ()**

Ez a függvény letöröl mindent a felületről.

- **Restart ()**

Egy algoritmus futása után, a csúcsok és élek egyik tulajdonsága megváltozik, amivel lehet ellenőrizni, hogy járt-e már ott a folyamat. Ebben a függvényben ezt a tulajdonságot változtatjuk, hogy újra lehessen futtatni egy algoritmust.

- **returnButtonClicked ()** és
  forwardButtonClicked ()

Az ismétlés és visszavonás folyamatok végrehajtására, egy segédváltozót vettem fel, amivel el lehet dönteni, hogy mi volt az utolsó felhelyezett vagy törölt elem, és a megfelelő komponenset visszahelyezni vagy eltüntetni a felületről.

Mind a három algoritmust külön függvényben tárolom, és egy külön változó segítségével döntöm el, hogy melyiket is választotta ki a felhasználó, és az alapján futtatom a megfelelőt.

- **FindMinDistancePath (Node start, Node end)** és
  FindAllMinDistancePath (Node nsource)

A Dijkstra és a Bellman Ford algoritmusok egy kiválasztott csúcsból egy másik kiválasztott csúcsba vagy az összes többi csúcsba keresi meg a legrövidebb utat. Ezek a függvények a **combobox** indexe alapján eldöntik, hogy melyik algoritmus fusson végig a gráfon.

4.3.1.2.2 DBalgorithmMinDistanceAlgo osztály

- **BellmanFord (Graph graph, int source, int end = -1)** és
  BellmanFordFIFO (Graph graph, int source, int end = -1) és
  Dijkstra (Graph graph, int source, int verticesCount, int end = -1)

A legrövidebb utak három algoritmusának implementálása. Az opcionális end paraméterrel döntjük el, hogy az algoritmus az egyik csúcsból az összes többi csúcsba keresi az utat vagy egy választott csúcsba.

- **MinimumDistance (int[] distance, bool[] shortestPathTreeSet, int verticesCount)**

Egy segédfüggvény a Dijkstra algoritmushoz, ami megadja, hogy melyik csúcs van a legkisebb távolságra, és még nem jártunk ott.

- **SearchPath (ref GraphTable Table, Graph graph, int[] parent, Double[] distance, int source, int verticesCount, int end = -1)**

Ha az algoritmus sikeresen lefutott, akkor a szülő (parent) tömb segítségével visszakeressük az algoritmus útját a gráfban.

4.3.1.3 Nézet réteg

A **nézetek** a felhasználói felületet megjelenítő osztályokat foglalják magukba. A program minden lapjához (ablakához) tartozik egy **XAML** és egy **C#** kód is. A *XAML* kódban egy *HTML*-hez hasonló nyelven van leírva az ablak megjelenése, a komponensek helye, mérete, színe és hogy hogyan lehet velük interakcióba lépni. Az ablakok működését a lapokhoz tartozó *C#* kódban lehet kifejteni. A nézetek névtere az **DBalgorithm.View**.

Az alkalmazás indulásáért és a különböző alkalmazás futási státuszokért a beépített *Application* osztályból öröklődő **App** osztály valósítja meg. Az osztály konstruktora hozza létre a nézeteket, ez a belépési pontja a programnak. A nézet a legfelső rétege az alkalmazásomnak ezért a nézet feladata a többi réteg példányosítása, inicializálása majd ezek összekötése. A rétegek osztálypéldányait privát mezőkben tárolja el (**_dataAccess**, **_model,_dBalgorithmMinDistanceAlgo**). Az *Application* osztály rendelkezik egy **App_Startup** eseménykezelő metódussal mely az alkalmazás indításakor fut le. A függvény lefutása után, az alkalmazás betöltődése közben még mielőtt megjelenne a felhasználói felület példányosítjuk az alsóbb rétegeket. Az egyes rétegek konstruktor paraméterként kapják meg az alsóbb rétegek példányait ezért alulról felfelé haladó sorrendben kell példányosítani és inicializálni a komponenseket.

Két ablakunk van az alkalmazásunkban, a fájlkezelő ablakokon kívül amit a Save és a Load metódusok hívnak meg. A fő ablakunk (**MainWindow**) valamint egy segédablak egy él súlyának megadásához(**DistanceWindow**).

A nézet réteg osztályai, és egymástól való függésük a következő képen látható:

4.3.1.3.1 MainWindow osztály

- **MainWindow ()**

Ennek az ablaknak a konstruktor lefutásakor helyezi el a képernyőn az objektumokat, és tölti be az ikonunkat.

- **Window_Loaded (object sender, RoutedEventArgs e)**

Az ablak elemeinek elhelyezése után, a load függvényben példányosítjuk az alsóbb rétegeket a nézet számára. Először a perzisztencia osztályt, mert erre szüksége van a modell osztályunknak. Itt hozzuk létre egy **Timer**-t, aminek a Tick eventjének

segítségével tesszünk lenyomhatóvá a visszavonás és ismétlés gombokat. Valamint itt kapják meg az értékeiket a **combobox-**ok, és hogy melyik indexen is állnak.

- **drawingCanvas_MouseLeftButtonUp (object sender, MouseButtonEventArgs e)**

A nézet legnagyobb függvénye a Canvas felületén lenyomott bal gomb, itt egymás után ellenőrzések futnak le a modellből meghívva, mivel a bal gomb lenyomásakor felülírhatunk egy súlyt, törölhetünk a felületről egy élt vagy csúcsot, hozzáadhatunk egy élt két csúcs kijelölése után, vagy egy üres helyre kattintva egy újabb csúcsot.

- **drawingCanvas_MouseRightButtonDown**
  (object sender, MouseButtonEventArgs e)

A jobb gombbal tudunk áthelyezni egy elemet a felületen. Ez a függvény kétfelé van osztva, mert lenyomáskor ellenőrizzük, hogy egy csúcs vagy él pozícióját szeretnénk módosítani.

- **drawingCanvas_MouseRightButtonUp (object sender, MouseButtonEventArgs e)**

A jobb oldali gomb elengedésekor nézzük a kurzor pozícióját, és az általunk áthelyezni kívánt elem oda fog kerülni.

Az egér gombjainak eseményei után, a menüsor gombjainak eseményei következnek.

- **clearBtn_Click (object sender, RoutedEventArgs e)**

Balról kezdve a legelső gombbal készíthetünk, egy új gráfot. Ez a függvény letöröli az egész felületet, így kezdhetjük újra a rajzolást.

- **restartBtn_Click (object sender, RoutedEventArgs e)**

Az algoritmus visszavonása gombbal újra színezi a gráfot, ezzel elérhetővé téve, egy új algoritmus futtatását.

- **saveButton_Click (object sender, RoutedEventArgs e)** és
  loadButton_Click (object sender, RoutedEventArgs e)

A mentés és betöltés gombok lenyomásakor megjelenik egy fájlkezelő felület, az egyszerűbb navigálás miatt, hogy a felhasználó maga tudja eldönteni, hova is szeretné elmenteni a gráfját, majd onnan visszatölteni.

- **del_Click (object sender, RoutedEventArgs e)**

A törlés gomb lenyomása után ki kell választanunk egy csúcsot vagy egy élet, ezután ez törlésre kerül. Természetesen ilyenkor a függvény ellenőrzi, hogyha a kiválasztott elem egy csúcs, akkor ahhoz milyen élek tartoztak, és ezek az élek is törlésre kerülnek.

- **exitButton_Click (object sender, RoutedEventArgs e)**

A kilépés gombra kattintás után megkérdezzük a felhasználót, hogy biztos-e a döntésében, és ennek a válaszadása utána történik csak meg a bezárás.

- **findMinDistanceBtn_Click (object sender, RoutedEventArgs e)** és
  findAllMinDistanceBtn_Click (object sender, RoutedEventArgs e)

Kétféle gombot hoztam létre a legrövidebb utak megtalálására. Az egyikkel csak egy csúcsot kell kiválasztanunk, és abból a csúcsból megtalálja az összes többi csúcsba a legrövidebb utat. A másikkal két csúcsot kell kiválasztanunk, és akkor azok között keresi meg a legrövidebb utat.

- **PaintMinDistancePath (Node start, Node end)** és
  PaintAllMinDistancePath (Node nsource)

A kétféle gombnak, kétféle rajzolása van. Amikor egy csúcsból az összes csúcsba keressük a legrövidebb utakat, akkor a rajzoláson kívül létrehoz egy táblázatot is amit feltölt a kezdőcsúcstól az összes többi csúcshoz tartozó súlyokkal. Két csúcs esetén a státuszsorban helyezem el a legrövidebb útnak a mennyiségét. A rajzolás az algoritmus futása közben történik a háttérben. Ahogy megtalálja a legrövidebb utat, azon visszamegy a kezdőcsúcshoz, és eközben változtatja a színét a gráfnak.

- **returnButton_Click (object sender, RoutedEventArgs e)** és
  forwardButton_Click (object sender, RoutedEventArgs e)

Visszavonás gomb lenyomásával, a legutoljára felrajzolt csúcs vagy él eltűnik a képernyőről, és elhelyezem a háttérben a szükséges adatokkal. Ha a felhasználó mégis úgy gondolja, hogy szüksége van arra az elemre, akkor vissza tudjam állítani az ismétlés gombbal.

- **distanceList_MouseDoubleClick (object sender, MouseButtonEventArgs e)**

A képernyőn elhelyeztem egy **listbox-**ot is, és ebben sorolom fel az éleknek a súlyait. Azért ha már egy bonyolult gráfon dolgozunk, akkor nem tudjuk biztosra, hogy melyik élnek a súlyát akarjuk felülírni, akkor ennek a **listbox-**nak az eseményével megtehetjük ezt.

- **createNodeBtn_Click (object sender, RoutedEventArgs e)** és
  createEdgeBtn_Click (object sender, RoutedEventArgs e)

A felrajzoláson kívül választhatjuk a billentyűzetről való bevitelt is. Miután a felhasználó megadta a szükséges adatokat, a program felrajzol neki egy csúcsot vagy élt.

- **create_PreviewTextInput (object sender, TextCompositionEventArgs e)**

Billentyűzetről való bevitelnél ellenőrizzük, hogy csak számok kerüljenek a beviteli mezőbe.

- **deledge_Click (object sender, RoutedEventArgs e)**

Ez a gomb aktiválja azt a lehetőséget, hogy minden irányítottság váltásnál törlődjön az összes él, vagy csak változzon meg az élek irányítottsága.

- **PaintNode (Node node)**

Ez a függvény hívódik meg egy csúcs rajzolásakor. Egy csúcsot a c#-ba beépített **Ellipse** típussal valósítjuk meg, amire felteszünk egy szövegdobozt, amivel jelezzük az egyedi azonosítóját.

- **PaintUnDirecetedEdge (Edge edge)** és
  PaintDirecetedEdge (Edge edge)

Ez a függvény fut le egy él rajzolásakor. Egy élt a c#-ba implementált **Line** típussal valósítjuk meg, aminek a közepére felteszünk egy szövegdobozt, és ezzel jelezzük az él súlyát. Valamint amikor áthelyezzük egy él középpontját, akkor oda egy töréspont kerül, és onnantól két vonalból fog állni az él.

- **DrawLinkArrow (Edge edge)**

A programnyelvben megtalálható Line típust nem lehetett olyan paraméterrel megadni, hogy az irányított gráfnak megfelelő nyíl alakot öltjön. Ezért egy segédfüggvényt hoztam létre, amivel egy háromszöget rajzolok a vonal végére.

- **PaintAllNodes()** és
  PaintAllEdges()

Végigmegy az összes élen vagy csúcson, és a beállításoknak megfelelően felrajzolja őket a képernyőre.

- **directedComboBox_SelectionChanged**
  (object sender, SelectionChangedEventArgs e)

Ez a függvény kezeli az irányítottság váltást. Ha megmaradnak az élek váltáskor, akkor ilyenkor végig megyünk az összes élen, és módosítjuk az irányítottság szerint.

- **wMinDistanceComboBox_SelectionChanged (object sender,**
  SelectionChangedEventArgs e)

A legrövidebb utat kereső algoritmusok kiválasztására egy **combobox-**ot hoztam létre, és ebben a függvényben kerül ellenőrzésre a Dijkstra algoritmus választásánál, hogy ne tartalmazzon a gráf negatív élet.

4.3.1.3.2 DistanceDialog osztály

Egy új él készítésekor, vagy egy meglévő él súlyának felülírásakor jelenik meg a nézetnek a másik ablaka. Ebben az ablakban adhatsz meg egy számot, amit az OK gomb lenyomása után felvisz a felületre egy **TextBlock**-al.

- **okBtn_Click(object sender, RoutedEventArgs e)**

Ezen beviteli mezőn olyan ellenőrzések vannak megírva, hogy addig nem tűnik el, amíg megfelelő adatot nem viszünk be, vagy ki nem lépünk belőle.

4.3.2 Extra Funkciók

A szakdolgozatom témabejelentőjébe beleírt funkciókon kívül az alkalmazás támogat extra funkciókat, amiket a Modell rétegbe ágyaztam be. Többek között vissza lehet vonni az utoljára felrajzolt lépést a gráfon. Ilyenkor törlődik a felületről, de csak eltárolom egy másik listába, ahonnan bármikor vissza tudom hozni. Ezért is van olyan funkció is, hogy ismétlés. Ilyenkor, ha mégis úgy gondoljuk, hogy szükségünk van arra a lépésre, akkor vissza lehet tenni a felületre azt is. Ezeknek a funkcióknak az adatai minden egyes lépés után törlődnek. A következő ilyen funkció a törlés, ilyenkor ahova kattintunk a felületen, ott először ellenőrzi a program, hogy mi is van ott. Például, ha egy csúcs, akkor a csúcsba menő összes élet töröli, és a hozzájuk tartozó adatok is. A következő az Irányítottság váltásnál az összes él törlése menüpont. Ezt a funkciót azért tettem bele, hogy aki jobban szeretné újrakezdeni a munkáját, amikor irányítottságot vált, annak van rá lehetősége. Ekkor csak csúcsok maradnak a képernyőn, és az élek nem. Ezen kívül a felületen megtalálható egy **ListBox** is, ez egy lista ahova az élek sorszámát teszem be a hozzátartozó súlyokkal. Az egyszerűbb felhasználás miatt döntöttem úgy, hogy beleteszem, mivel ilyenkor nem kell a felületre kattintani, hogyha szeretnénk egy élnek új súlyt megadni, hanem itt kiválasztjuk, és a megfelelő adatbevitel után már az új súllyal rendelkezik a gráf. Létrehoztam beviteli mezőket a csúcsok koordinátáinak megadására. Ezeken megfelelő ellenőrzés fut, hogy ne tudjanak a rajzoló felületen kívülre tenni semmit. Az élek felrajzolásához csak meg kell adni a kezdő, és végcsúcsot, valamint az él súlyát, és el is készül.

4.4. Tesztelés

Az alkalmazás teszteléséhez nagy segítséget nyújt a Visual Studio fejlesztőkörnyezet. Lehetőségünk van létrehozni egy speciális **UnitTest** projektet melyben definiált teszt osztállyal teszt metódusokat futtathatunk a rétegek stabilitásának ellenőrzéséhez. A projektemben létrehoztam az **DBalgorithm.DBalgorithmTest** névtérben az **DBalgorithmModelTest** tesztosztályt, ami a modell réteget teszteli. Az osztályt szükséges ellátni a **[TestClass]** attribútummal. A modell **[TestInitialize]** attribútummal ellátott **Initialize** függvénye példányosítja a modellt és a modell létrehozásához szükséges alsóbb rétegeket.

Az alábbi tesztesetek kerültek megvalósításra:

- **ClearEdgeNodeTest**: Létrehoz két pontot, amikből Csúcsokat készít, majd amikor az élek csúcsaival is egyenlővé tette, akkor meghívom a modellben lévő függvényt, és ellenőrzöm, hogy jól futott-e le.
- **CreateEdgeTest**: Létrehoz két csúcsot, majd ezek között létrehoz egy élet.
- **RestartTest**: Felrajzolunk egy gráfot, és ezután töröljük.
- **ReturnTest**: Ellenőrzöm, hogy egy visszavonásnál ténylegesen belekerülnek-e a háttérben futó listákba az elemek.
- **ForwardTest**: Ellenőrzöm, hogy egy ismétlésnél ténylegesen belekerülnek-e az előtérben futó listákba az elemek.
- **ClearTest**: Létrehoz két csúcsot bizonyos pontokba, majd közöttük egy élet, megnézi, hogy ténylegesen léteznek-e majd meghívja a Modell Clear függvényét, és ekkor ellenőrizzük, hogy eltűntek-e.
- **DirectedComboboxSelectionChangedTest**: Létrehozok két csúcsot, és közöttük egy élt, majd amikor meghívom, ezt a függvényt megnézem, hogy törlődik-e az él a felületről.
- **GetEdgeDistanceTest**: Ellenőrzi, hogy egy létrehozott élnek a súlyát megkapja-e a modellben lévő változó
- **OverlapsNodeTest**: Megnézi, hogy a megadott helyen van-e él, létrehoz ott egyet, és utána is ellenőrzi.

- **GetNodeAtTest**: Létrehozok a felületre egy csúcsot, és megnézem, hogy a létrehozott csúcsot kapom-e vissza a modellben lévő függvény alapján.
- **HasClickedOnNodeTest**: Létrehoz egy csúcsot, és meghívja a modellben lévő függvényt, amivel ellenőrzi, hogy ténylegesen arra kattintott-e a felhasználó.
- **AssingEndNodeTest**: Létrehozunk két csúcsot a felületen, és azokat értékül adjuk, mint él végpontok, és ellenőrizzük, hogy meg is kapták-e
- **GetDistanceTest**: Létrehozunk az két csúcsot és egy élt közöttük, és ellenőrizzük az él középpontját.
- **GetEdgeAtTest**: Létrehoznak két csúcsot és egy élt közöttük, és megnézzük, hogy visszakapjuk-e a modellben lévő függvény alapján.
- **HasClickedOnEdgeTest**: Létrehozunk két csúcsot, és közöttük egy élt, ezután megnézzük, hogy a modellben lévő függvény igazat ad-e vissza a meghívott pozícióra.
- **GraphStackEdgeTest**: Létrehozunk egy gráfot, és ellenőrizzük, hogy a felhasználó által kiválasztott csúcsok között van-e már él.
- **GraphNOStackedEdgeTest**: Létrehozunk egy gráfot, és ellenőrizzük, hogy van-e két olyan csúcs, amik között két él is fut.
- **GraphStackedandDeleteEdgeTest**: Az előző teszt mintájára, ha előfordulnak ilyen csúcsok, akkor töröljük közöttük a nagyobb súlyú élt.
- **DijkstraTest**: Elkészítek gráfokat, és futtatom rajtuk a Dijkstra algoritmust sikeresen.
- **BellmanFordOKTest**: Elkészítek gráfokat, és futtatom rajtuk a Bellman-Ford algoritmust sikeresen.
- **BellmanFordCircleTest**: Elkészítek gráfokat, és futtatom rajtuk a Bellman-Ford algoritmust, ami a startcsúcsból elérhető negatív kört talál.
- **BellmanFordFIFOOKTest:** Elkészítek gráfokat, és futtatom rajtuk a Bellman-Ford FIFO algoritmust sikeresen.
- **BellmanFordFIFOCircleTest**: Elkészítek gráfokat, és futtatom rajtuk a Bellman-Ford algoritmust, ami a startcsúcsból elérhető negatív kört talál.

4.5. Fejlesztési lehetőségek

Az alkalmazás habár már hónapok óta megbízhatóan működik és megvalósítja a kitűzött célokat, további funkciókkal és lehetőségekkel bővíthető.

- Adatbázis cseréje SQL alapú adatbázisra.
- Felhő alapú adattárolás
- A modell osztályait ellátni jól definiált interfészekkel
- Több algoritmus futtathatósága
- Negatív kör kirajzolása piros színnel
- Többnyelvűség
- A legrövidebb utak megtalálásának animálása
- Alkalmazás fejlesztése Universal Windows Platform-on (UWP), hogy mobil, és tablet felületen, valamint más operációs rendszeren is futtathatóvá váljon.
- Telefonon, számítógépen vagy tableten lévő térkép képek importálása az alkalmazásba, amire automatikusan rajzolna gráfot, és lehetne futtatni az algoritmusokat.

5. Összegzés

Az alkalmazásom egytől harminc csúcsig terjedő gráfok megrajzolására, és ezeknek mentésére és visszatöltésére ad lehetőséget, amiken futtathatjuk a három általam implementált algoritmust. A felhasználó mindenféle nehézségek nélkül, pár kattintással meg tudja rajzolni a gráfját, amin kétféleképpen is futtathatja az algoritmusokat. Ha van egy bizonyos útvonal, akkor azt is megkeresheti, és csak az jelenik meg, de futtathatja egy csúcsból az összes többi csúcsba vezető legrövidebb utat is. A megrajzolt vagy módosított gráfoknak az adatbázisba való elmentésével és későbbi visszatöltésével sok időt és munkát takaríthatunk meg.

Az alkalmazásom Windows Presentation Foundation-ben készült, azaz Windows operációs rendszert futtató eszközökön lehet használni. A célplatform Windows 10, mivel ezen teszteltem legtöbbet. Az adatokat lokálisan, szöveges állományból felépített adatbázisban tárolja. A megfelelő interfészeknek köszönhetően ez később cserélhető SQL alapú adatbázisra vagy felhőtárolásra.

Az alkalmazásomat hónapok óta tesztelem, és remélem megfelelő segítséget fog nyújtani a felhasználók számára.

6. Függelék

6.1. Az algoritmusok fejlesztőinek önéletrajzai

6.1.1. Edsger Wybe Dijkstra

Edsger Wybe Dijkstra 1930-ban született a hollandiai Rotterdamban. Szülei elismerten jó végzettségű értelmiségiek voltak, édesapja kémikus, édesanyja matematikus volt. 1942-ben, 12 éves korában bekerült egy igen magas színvonalú gimnáziumba, az Erasminium Gimnáziumba, ahova kivételes tehetségek jártak. Dijkstra sok különböző tárgyat tanult, mint például: görög, latin, francia, német és angol nyelv, valamint biológia, matematika és kémia.

1945-ben még jogi pályára készült, hogy utána képviselőként dolgozzon. Azonban abból kifolyólag, hogy kémiából, biológiából és matematikából jó eredményei voltak, úgy döntött, hogy Leindeni egyetemen folytatja tanulmányait és elméleti pszichológiát hallgat. 1951 nyarán, a cambridge-i egyetemen ismerkedett meg először a programozással. 1952 márciusától részmunkaidőben dolgozott egy amszterdami matematikai központnál, ekkor kezdte el igazán érdekelni az informatika. Amilyen gyorsan csak lehetett befejezte a pszichológiai tanulmányait és elkezdett hódolni ennek az új szenvedélyének.

Miután 1957-ben megházasodott, folytatta a munkáját a matematikai központban, miközben az 1970-es évek elején az egyesül államokbeli Borroughs Corporation kutatási tagja is volt. 1972-ben megkapta az ACM Turing Díjat, 1974-ben az AFIPS Harry Good Memorial Díjat. Az 1980-as évek elején a texasi Austinban költözött, majd 1984-ben állást is kapott az Informatikai Tudományok Egyetemén, ahol 69 éves koráig dolgozott. 1999-ben lett professor emeritus. Rákban halt meg nueneni otthonában 2002. augusztus 6-án.

(Wikipedia, 2007)

6.1.2. Lester Randolph Ford, Richard Bellman

Ifjabb Lester Randolph Ford 1927. szeptember 23-án született. A „Network Flow” programozás egyik úttörője. Édesapja az idősebbik L.R. Ford, aki maga is elismert matematikus, a Farey sorozatokra adott egy bámulatos értelmezést. Ifjabb L.R. Ford nevéhez fűződik többek között a Ford-Fulkerson algoritmus is, amely a maximális folyam problémát oldja meg. (Singh Nayandeep, 2001)

Richard Ernest Bellman (1920. augusztus 26. – 1984. március 19.) alkalmazott matematikus volt, az a1953-as dinamikus programozás terén elért felfedezéséért volt méltán ünnepelt, valamint nevéhez köthető, sok a matematika más területén elért eredménye is.

New Yorkban született, ahol édesapja, John James Bellman egy élelmiszerüzletet vezetett a Bergen utcában a Prospest Park közelében Brooklynban. Bellman a középiskolában (New York), és matematikát hallgatott a brooklyni főiskolán, ahol 1941-ben szerzett BA diplomát, a későbbiekben pedig a Wisconsin-Madison Egyetemen megszerezte az MA diplomát is. A II. világháború idején a hadseregnél az Elméleti Pszichológia Részlegen dolgozott Los Alamosban. 1946-ban megszerezte a Ph.D titulust a Princetonon. A dél kaliforniai egyetem professzora volt, ösztöndíjas kutatója az amerikai Tudományok és Művészetek Akadémiájának (1975), valamint tagja a nemzeti Mérnöki Akadémiának (1977). 1979-ben az IEEE Becsület Medállal tüntették ki „A döntési eljárások és az ellenőrző rendszerek területén elért eredményéért, különösen a dinamikus programozásban alkotottakért.” Legfőbb műve a Bellman-egyenlet (Wikipedia, 2009)

6.1.3. Robert Endre Tarján

Robert Endre Tarján 1948. április 30-án született. Amerikai számítógépes tudós és matematikus. Ő a kitalálója több gráf algoritmusnak, többek között a Tarján off-line legkisebb közös ősök algoritmusnak, és a társ-feltalálója a ferde fáknak és a Fibonacci kupacnak. Tarján jelenleg a James S. McDonnel kiváló egyetem professzora. Pomonában született. Apja gyermekpszichiáter egy állami kórházban. Tarján sok sci-fi könyvet olvasott, ezért csillagász akart lenni, de elkezdte érdekelni a matematika, miután elolvasta Martin Gardner egyik könyvét. Jó tanára miatt általános iskolában még jobban elkezdte érdekelni a matematika. Középiskolai éveiben Tarján munkát kapott az IBM-nél. Ekkor dolgozott valódi számítógépekkel, miközben csillagászatot tanult. Ezután BA diplomát kapott matematikából, a Kaliforniai technológia intézetben. A Stanfordon folytatta tanulmányait, ahol megszerezte mesterdiplomáját, Informatikai szakon, 1971-ben, majd a Doktori végzettségét, 1972-ben, ugyanitt. Ezen az egyetemen Robert Floyd és Donald Knuth felügyelte munkásságát. Tarján azért választotta a számítástechnikát, mint területet, mert azt sejtette, hogy a CS a matematika egy gyakorlati alkalmazási területe. Tarján a Princetoni Egyetemen tanít 1985 óta. Emellett akadémia pozícióban volt a Corelli Egyetemen, a Kaliforniai egyetemen, a Stanford egyetemen, és a New York-i egyetemen, valamint a NEC kutatóintézet egyik munkatársa volt. 2014 októberében csatlakozott az Intertrust-hoz, technológiai vezetőkutatóként. Néhány közismert algoritmusa a Tarján-féle erősen összefüggő komponensek algoritmus és ő volt a medián a mediánban lineáris idejű kiválasztási algoritmus öt társszerzőjének egyike.

(Wikipedia, 2010)

7. Irodalomjegyzék

- [1] Matthew MacDonald: Windows Presentation Foundation in .NET 4.5, Apress, 2012, 9781484214503
- [2] Illés Zoltán: Programozás C# nyelven, Jedlik Oktatási Stúdió, 2008, 9638762948
- https://en.wikipedia.org/wiki/Dijkstra's_algorithm 2016.10.28.
- https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm 2016.10.28.
- http://aszt.inf.elte.hu/~asvanyi/ad/Algoritmusok%20es%20adatszerkezetek%202%20javitott%202016.06.18.pdf 2016.11.05.
- https://www.codeproject.com/articles/15354/dragging-elements-in-a-canvas 2016.12.07.
- https://msdn.microsoft.com/en-us/library/system.windows.controls.menuitem(v=vs.110).aspx 2016.12.05.
- https://msdn.microsoft.com/en-us/library/bb203924.aspx 2016.12.05.
- http://www.c-sharpcorner.com/uploadfile/mahesh/listbox-in-wpf/ 2016.11.05.
- https://www.dotnetperls.com/combobox-wpf 2016.11.04.
- https://msdn.microsoft.com/en-us/magazine/jj991977.aspx 2016.11.10.
- https://hu.wikipedia.org/wiki/T%C3%B6bbr%C3%A9teg%C5%B1_architecht%C3%BAra#R.C3.A9tegek 2016.12.03.
- https://msdn.microsoft.com/en-us/library/ms747393(v=vs.110).aspx 2016.10.13.