Matematikai és Informatikai Intézet

Gráfelméleti megoldó-algoritmusok automatikus tesztelése C# nyelven írt alkalmazással

Készítette
Dancsó Sándor
Programtervező informatikus

Témavezető
Troll Ede Mátyás
Tanársegéd

Eger, 2020

Tartalomjegyzék

Bevezetés

Olyan nincs, hogy nincs

> „Hamilton kör probléma megoldására nincsen polinomiális időbonyolultságú algoritmus."
>
> (Dr. Kovásznai Gergely)

Jelen dolgozat témája egy gráfelméleti problémák megoldására és megoldó algoritmusok tesztelésére szolgáló asztali alkalmazás. A szoftver megvalósításának célja kettős. Egyrészt egy könnyen kezelhető grafikus felület kialakítása gráfok egyszerű megjelenítésére, szerkesztésére, azokon különböző számítások végzésére, másrészt lehetőség biztosítása a képességeiket próbára tenni kívánó fejlesztőtársaknak a saját maguk által megálmodott és implementált gráfelméleti megoldó-algoritmusaik kipróbálására, automatikus tesztelésére és más megoldókkal való összehasonlítására.

A fenti idézet csak egy egyszerű, tanórán elhangzott mondat, ám témaválasztásomnak – ha nem is közvetlenül, de – ez volt a kiváltó oka. Akkoriban még minimális ismeretem sem volt bonyolultság elméletről, sőt talán még azt sem tudtam, hogy létezik ez a terület. Szkeptikus énem – mint oly sokszor – a felszínre tört, és napokra (talán hetekre) átvette az irányítást.

Több algoritmus ötlete is összeállt a fejemben, mielőtt hazaértem volna a szóban forgó óráról, de később az asztalnál ülve felettébb nehezen megvalósítható és igencsak időigényes volt, hogy próbára tegyem őket. A papíron történő rajzolás és számolás lehetőségét egy elszámolásokkal teleírt spirálfüzet után elvetettem, és a második lehetőségként felmerülő táblázatkezelőben való próbálkozás is elég rugalmatlannak és lassúnak bizonyult.

Mások hagyatéka

A kezdeti kudarcokat követően koncepció váltás következett, és a kutatásra helyeztem a hangsúlyt. Első körben azt fürkésztem, hogy akadt-e már valaki, akinek sikerült rácáfolnia a fenti állításra. A világháló – mint már megannyi esetben – megfelelő területnek bizonyult, hogy átfésüljem. Nem merném kijelenteni, hogy dúskálhattam a

témában érdemlegesen kutatást végzők tényközlő leírásaiban, főleg nem világmegváltó publikációkban, viszont könnyedén megfogalmazhattam három kategóriát, melybe besorolhatjuk a Hamilton-kör problémával foglalkozókat.

Első körben említeném a „meg nem értett zsenik” csoportját. Ők váltig állítják, hogy mindent elsöprő módszerük van, mellyel valós időben találnak Hamilton-kört bármilyen gráfban, de olyan bonyolult az elméletük, és olyan hosszú a matematikai leírás, illetve bizonyítás, hogy azt még senki sem értette meg, így nem is teszik közzé, hanem belenyugodnak, hogy a módszer nem állhat a világ szolgálatába.

Másodsorban megemlíthetjük a „brute force rejtegetők” osztályát. Ők valódi megoldásokkal állnak elő, melyet nem rejtenek véka alá, de a módszerükben ilyen, vagy olyan formában mindig fellelhető egy apró probléma. A történet mindig azzal végződik, hogy szisztematikusan próbálkozunk, míg meg nem leljük azt a Hamilton-kört.

És végül említést érdemel a „valódi kutatók” maroknyi serege. Ők valóban kutatnak, és alkotnak. Belátják, hogy nem lehetséges a problémát polinomiális időben általánosan megoldó algoritmus felírása, ám (főleg) mintakeresésen alapuló módszereket dolgoznak ki, és fejlesztenek tovább, melyek adott korlátok között hatékonyan működnek. Megfelelő hozzáférés hiányában módszereikből és eredményeikből csak töredékeket láthatunk sajnos.

Ezen harmadik csoportot illetően az általam fellelt eredmények közül egyet említenék, melyhez elérést a [4] alatt találunk. A szóban forgó módszer az 5000 és annál kevesebb csomóponttal bíró gráfok körében képes hatékonyan Hamilton-kört találni. A hatékonyság számszerűsített értékéről, valamint a megoldó algoritmus időbonyolultságáról és -igényéről konkrét adat nem áll rendelkezésre.

Szoftveres támogatottság

A megoldó algoritmusok utáni kutatásom kettős eredménnyel zárult. Egyrészt alapvető ismereteket szereztem a bonyolultság elméletében, és beláttam, hogy mégis lehet némi igazság a fejezetet indító idézetben, másrészt megerősítést nyert bennem az a tény, hogy a nem egzakt módszerek is lehetnek eredményesek. Korábbi mérnöki munkásságom során is alkalmaztam olyan módszereket, melyek matematikailag és fizikailag megalapozatlanok voltak, ám a gyakorlatban sok-sok éve, évtizede megbízhatóan működnek. Miért is lenne másképp ez a tudomány egyéb területein?

Továbbra is rendíthetetlen volt a fejemben kavargó algoritmusokba vetett bizalmam. Tudtam, hogy valamelyikük sikeresen fog működni, sőt az 5000 csomópontos határt is könnyedén maga mögött hagyja, ám kipróbálásuk a gyakorlatban még mindig nem volt megoldott. Újabb kutatásba kezdtem.

Találtam néhány alkalmazást, mind asztalit, mind online felületűt, mellyel gráfokat definiálhatok, és az adott gráfban megoldást kereshetek adott problémákra (többek

között Hamilton-kör problémára), azonban mindig ugyanazokkal a negatívumokkal kellett, hogy szembetaláljam magam:

- A gráfok definiálásának rugalmatlansága. Vagy valami rögzített formátumú, szöveges definícióra volt lehetőségem, mely a végtelenségig kényelmetlen és átláthatatlan a vizuális megjelenítéshez szokott szemnek, vagy létezett grafikus felület, de annak lehetőségei korlátozottak és felettébb barátságtalanok voltak a felhasználó számára.

- Alapértelmezett permutációs elven működő algoritmusok szolgáltatták a megoldásokat, így 15 csomópont esetében akadt időm egy frissítő italra, 20 csomópontnál pedig az unokáim felnevelésére is. És ami közös volt minden fellelt szoftverben, hogy ezen beton-stabilan beégetett megoldók helyett nem állt módomban saját szubrutinnal próbálkozni.

A dolgozat célja

Ezen a ponton döntöttem úgy, hogy egy időre pihentetem az algoritmusaim tesztelését, és új célokat tűzök ki magamnak, melyeket a következő pontokban fogalmaztam meg:

- Elkészítek egy alkalmazást, mely a próbálkozó szellemű társaimnak lehetőséget biztosít saját gráfelméleti megoldó-algoritmusaik tesztelésére a következő pontok szerint.

- A program felületet biztosít gráfok definiálásához. A felület mindenképp grafikus, rugalmas és felhasználóbarát kell, hogy legyen, és biztosítania kell minden alapvető funkciót, ami gráfok jellemzését, szerkeszthetőségét szolgálja. Szintén minimális felületi követelmény több gráf párhuzamos kezelése, valamint a rajtuk végezhető alapvető fájlműveletek garantálása.

- Az alkalmazás ismeri a leggyakoribb gráfelméleti problémákat, és lehetőséget nyújt a felhasználónak, hogy megoldási módszert definiáljon a problémák valamelyikére.

- A felhasználó kipróbálhatja a maga által írt megoldó algoritmust, melyhez két módot is biztosítani kell. Egyrészt a grafikus felületen megrajzolt gráfokon lehessen tesztelni a megoldót, és vizualizálni a kapott eredményt, másrészt az automatikus megoldó-tesztelés lehetőségét is meg kell adni.

- A program kell, hogy tartalmazzon alapértelmezett megoldókat az általa ismert problémákra. A beépített megoldók a klasszikus algoritmusok szerint működnek, és kettős célt szolgálnak. Egyrészt esetlegesen felmerülő, egyszerűen felvázolható,

konkrét problémákra biztos (ám nem feltétlenül gyors) megoldást garantálnak, másrészt összehasonlítási alapot nyújtanak a saját megoldóink tesztelésekor.

– Az előző pont folyományaként definiálni kell egy módszert és szempontrendszert, mely alapján két, azonos probléma megoldását célzó algoritmus hatékonysága összehasonlíthatóvá válik.

– Az alkalmazás megfelelő dokumentáltságát biztosítani kell mind felhasználói, mind fejlesztői, mind pedig telepítési szempontból.

– Végső célom a fentiek megvalósítását követően egy olyan megoldó implementálása, mely gyorsabban és szélesebb skálán talál Hamilton-kört, mint a jelenleg ismert (vagy ismeretlen) algoritmusok.

1. fejezet

Alternatívák és döntések

1.1. Nyelv és technológia

Mint minden, kötöttségektől mentes igény esetében, elsődlegesen afelől kellett döntenem, hogy milyen platformra és milyen nyelven készüljön el az alkalmazás.

Matematikai jellegű problémáról lévén szó, mindenképp megfontolandónak tartottam valamely 4. generációs, matematikára specializálódott nyelv (például MatLab) alkalmazását, azonban több hátránnyal járt volna ez a megoldás, mint előnnyel.

- Bár könnyedén lehetne ezeken a nyelveken megoldó algoritmusokat implementálni és kipróbálni, az automatikus tesztelés lehetőségét nem tartom megvalósíthatónak.

- A számítások kimenetét kiválóan lehet vizuálisan szemléltetni, viszont a bemenetek grafikus megadása nem megoldott, így gráfok definiálására kizárólag a scriptelés marad, ami a felhasználói élmény teljes megsemmisülését vonja maga után.

- A szóban forgó programok, programnyelvek vagy nem érhetők el jelentős anyagi befektetés nélkül, vagy – ami az ingyenes helyettesítő megoldásokat illeti – aluldokumentáltak, kevésbé támogatottak.

- A mai vezető paradigma, az objektum orientáltság elveit alacsony szinten kezelik, sokkal inkább alkalmasak ezen nyelvek a procedurális programozásra, mely jelen körülmények között kevésbé célszerű választás a probléma megoldására és hatékony, jól áttekinthető implementáció kialakítására.

- A kitűzött feladat összetettségét figyelembe véve nem tartom szerencsésnek egy ekkora projekt keretein belül megismerkedni egy új nyelvvel és környezettel, mert a fejlesztés közben főleg a túlélésre kerülne a hangsúly, azaz az elsődleges feladat minden pillanatban az adott apró részletprobléma megoldása lenne bármilyen

áron, és majdhogynem lehetetlen feladat lenne magára az architektúrára és a kifejező, olvasható implementációra való koncentrálás.

A kiemelkedően magas számú matematikai könyvtár, a széleskörű közösségi támogatottság, a gazdag dokumentáció-készlet és a platformfüggetlenség mind a Python nyelv mint potenciális választás mellett szólnak. További pozitívumként említhető, hogy magasszintű, az OOP elveit teljes mértékben támogató nyelvről van szó, melyhez már készíthetők grafikus felhasználói felületek is. A Python esetében kizárólag a nyelv és a fejlesztői környezetek ismeretének hiánya az a negatívum, ami miatt ez a lehetőség elvetésre került, hisz így elkerülhető az, hogy a valódi problémára szánt idő és energia az ismerkedés és tanulás bonyodalmaira forduljon.

A Python esetén említettekhez teljesen hasonló pozitívumokat tudtam felvonultatni a Java esetében, mely az egyik legesélyesebb jelölt volt a nyelvválasztás során. A Java alapszintű ismeretének birtokában nem találtam lehetetlennek a kitűzött probléma megvalósítását (akár egy androidos mobilalkalmazás keretein belül sem), azonban választásom mégis a C#-ra esett. Legfőbb indokom a döntésem mellett az volt, hogy mind munkám során, mind szabadidős projektjeimben ezt a nyelvet alkalmazom, ezen nyelvhez kapcsolódó technológiákkal ismerkedem, ezen nyelv területén halmoztam fel olyan ismeretanyagot, mely segítségemre lehet bármely kitűzött probléma megvalósításában anélkül, hogy technológiai részletkérdések felkutatására kellene helyeznem a hangsúlyt.

Az alkalmazott technológia tekintetében elsőként egy MVC-s webalkalmazás megvalósításának lehetősége merült fel ASP.Net Core alkalmazásával. A nyelvválasztás során az egyetlen, amit sajnáltam feladni, az a Python és a Java által biztosított platformfüggetlenség. Az ASP.Net Core lehetett volna a megfelelő technológia, mely mégis megadja a lehetőséget, hogy mind a szerver oldali kód elfuthasson bármilyen rendszeren, mind a felhasználóknak biztosítva legyen a kliens oldali kapcsolat a kedvenc böngészőjében, bármilyen platformra is legyen az telepítve. Több grafikus alkalmazást készítettem már, melynek felületét a webböngésző biztosította JavaScript-ben megírt sorok által, így a feladat ezen része - vagyis gráfok grafikus ábrázolása és szerkesztése - sem lett volna idegen és meglepetésekkel teli számomra, ám a további, mélyebb átgondolás során a következő, a webes implementáció ellen szóló okokból mégis egy asztali alkalmazás elkészítése mellett döntöttem:

Egyik elsődleges célkitűzésem az, hogy a felhasználó által implementált megoldót automatikus tesztnek lehessen alávetni, melyhez szükséges az azt tartalmazó szerelvény szerveroldalra való feltöltése, és az abban rejlő kódsorok futtatása. Ez fontos biztonsági problémákat vet fel, és annak ellenőrzése, hogy a felhasználói megoldó semmilyen rossz szándékú algoritmust nem tartalmaz, igen nehezen kivitelezhető, valószínűleg nagyságrenddel komolyabb feladatot rejt magában, mint az eredeti célkitűzéseim, melyek között egyetlen biztonságtechnikai pont sem szerepel.

Amennyiben eltekintünk a biztonságtól, és jóhiszeműen megbízunk a felhasználókban, másik, szintén a megoldó-teszteléssel kapcsolatos, sokkal inkább technikai problémák is felvetődnek. A tesztelés menetének technikai kidolgozása nélkül is gyanítható, hogy egy megoldó képességeinek automatikus vizsgálata idő- és erőforrásigényes. Amennyiben egyszerre nagy számú felhasználó veszi igénybe a szolgáltatást, olyan számítási kapacitásra van szükség, amely komoly anyagi befektetést igényel, ám mégsem garancia arra, hogy az összes felhasználói igény megfelelően kielégítésre kerül. Amennyiben korlátozzuk az egyidejűleg megengedett felhasználói tesztek számát, és egy sorba helyezzük el az új tesztelési igényeket, a megjósolhatatlan várakozási idő eltántorítja a felhasználót az alkalmazás használatától.

Szintén a szerver aktuális terheltségéből adódó probléma, hogy az erőforrások több, vagy kevesebb számítási feladat közötti megosztása miatt egy adott megoldási folyamat vagy lassabban, vagy gyorsabban fut le, márpedig a megoldási idő figyelembe vétele a megoldó tesztelés egy fontos tényezője, melyet semmiképp sem befolyásolhat külső tényező, mint például az alkalmazást egy időben terhelő felhasználók száma.

Végül miképp a C# nyelven írt asztali szoftver megírására szűkült a kör, már csak azt kellett eldönteni, hogy WindowsForms, vagy WPF keretein belül készüljön el az alkalmazás. Lényegében személy szerint ezt már kevésbé fontos döntésnek tartottam, mindkét technológia kiválóan alkalmas a feladat kivitelezésére a maga előnyeivel és hátrányaival. Munkám során a WPF technológiai fogásait sajátítottam el, míg szabadidős projektjeimet főként WindowsForms-os alkalmazások formájában készítem folyamatosan feltérképezve a benne rejlő lehetőségeket. A döntésben meghatározó tényezők közül mindössze a következő kettőt emelném ki:

- WPF esetén a felületi logika és egyéni control-ok létrehozásának XAML kód általi kivitelezése, valamint a grafikus elemek – azaz a megrajzolt gráfok – nyilvántartása és kezelése jelenthet feleslegesen befektetendő többlet időt és energiát, míg WindowsForms-ban a rajzolásra és control-ok definiálására tökéletesen kiforrott, sok évre, évtizedre visszatekintő megoldások vannak, melyeket jómagam is nap mint nap alkalmazok.

- Az asztali alkalmazásokat a jobb átláthatóság és a teljes tesztelhetőség jegyében általában MVVM minta szerint alakítom ki. Jelen esetben sem szeretnék ettől eltérni. Míg a WPF a kötések által ezen minta kialakítását tökéletesen támogatja, WindowsForms-ban ezen mintát a gyakorlatban elég nehéz elérni, és sokszor csak elbonyolított felületi háttérkódhoz vezet.

Mielőtt kérdésként felmerülne, hogy miért nem egyértelmű az újabb, modernebb WPF-es technológia alkalmazása a már-már feledésbe merült WindowsForms-zal szemben, megjegyezném, hogy személy szerint a WPF-nek sem jósolok túl sok időt. Fény-

korán már bőven túl van, így a technológia kiválasztásánál ezt nem éreztem fontos szempontnak.

A választásom végül a WindowsForms-ra esett. A felmerülő MVVM problémát a 2.2. fejezetben említett saját készítésű library oldotta meg. Szintén megjegyzendő, hogy az MVVM mintának köszönhetően a jövőre tekintve fennáll a lehetőség, hogy minimális időráfordítással a felület WPF-esre cseréljem.

1.2. Kiválasztott problémák

Fontos kérdésnek tartottam, hogy a gyakorlatilag felsorolhatatlanul sok ma ismert (vagy kevésbé ismert) gráfelméleti probléma közül melyek kerüljenek bele az alkalmazásba. Minden egyes probléma esetén szükség van – a célkitűzéseimből kiindulva – egy beépített megoldó algoritmusra, egy problémadefinícióra, mely alapján egyéni megoldó implementációk születhetnek, egy, az adott probléma jegyeit tartalmazó gráfok véletlenszerű előállítását elvégző mechanizmusra, valamint egy, a mások által írt megoldókat tesztelő rendszerre. A felsorolt tényezők általában problématípusonként kisebb-nagyobb mértékben eltérők, ezért megvalósításuk minden egyes probléma esetén jelentős idő- és energiabefektetéssel jár.

A kiválasztott gráfproblémák, melyeket az előzetes számításaim szerint mindenképp szeretnék, hogy a jelen dolgozat keretein belül helyet kapjanak a programban a következők:

- Hamilton-út és Hamilton-kör. Az eredeti kiinduló problémámnak megfelelően a Hamilton problémák természetesen első körben válnak az alkalmazás részévé. A probléma megjelenik mind út, mind kör formájában irányított és irányítatlan gráfok tekintetében egyaránt.

- Euler problémák. Mondhatni, hogy a Hamilton problémák testvéréről lévén szó, az Euler problémák is természetesen bekerülnek a programba az előző esetben említett 4 formában, azaz mind út, mind kör formájában, mind irányított, mind irányítatlan esetekre.

- Maximális átfolyás problémája. A probléma természetesen csak irányított gráfokon értelmezhető.

- Legkisebb költségű feszítőfa problémája. Vizsgálható a probléma irányított gráfok esetén is, ám első körben az irányítatlan verzió kerül a programba.

- Legrövidebb út problémája. Jelen esetben szintén csak irányítatlan gráfok tekintetében kerül ez a probléma vizsgálatra.

A felsorolt problémákat tartom személy szerint a legtöbbek által ismertnek, így a program első verziójában ezeket szeretném, ha helyet kapnának. A felsorolás a későbbiekben természetesen bővíthető.

2. fejezet

A megvalósítás

Jelen fejezetben ismertetem, miképp láttam neki a feladatnak, milyen nehézségeim adódtak, azokat hogyan küzdöttem le, továbbá kitérek a megvalósítás azon részleteire, melyek véleményem szerintem külön említést érdemelnek.

2.1. Módszertan, környezet

A projektet mindenképpen kontrollált keretek között szándékoztam végigvinni, ezért a következő előfeltételeket biztosítottam a tervezés és implementáció előtt. A projekt teljeskörű menedzselésének elősegítésére a Microsoft Team Foundation Server 2018-as verzióját (TFS) telepítettem a számítógépemre, mely számos feladatkör ellátását biztosította számomra.

Elhatározásom szerint a kitűzött feladatokat – a paradox hangzása ellenére – egyszemélyes SCRUM csapatként végeztem el. Minden tennivalót apró részegységekre bontottam, melyeket bepontoztam, és több (2-3 hetes) iteráció keretében oldottam meg őket. Ezzel egyrészt garantáltam a magam számára, hogy nem fogok az időből kifutni, másrészt biztos lehettem abban is, hogy egy jól nyomon követhető folyamat során készül majd el az alkalmazásom. Az ehhez szükséges feltételek a TFS biztosította számomra, mindent megadott, ami egy ilyen (egyszemélyes) csapatmunkában szükséges lehet. Rendelkezésemre állt a különböző backlogok, a sprintek és egyéb előfeltételek definiálásának lehetősége. A beépített GIT rendszerének köszönhetően a verziókövetés lehetősége is adott volt a teljes fejlesztési folyamat alatt.

Az automatizálás jegyében a projekthez egy build scriptet írtam Cake Build rendszerben[1], melyet a TFS-ben készített build definícióban hívtam meg. A script segítségével minden pull request esetén mielőtt a friss kód bekerülhetett a rendszerbe, a program fordítása automatikusan végbement, és az összes unit- és integrációs teszt lefutott. Bármilyen hiba esetén a pull request visszautasításra, ellenkező esetben pe-

[1] A Cake Build rendszerről az [5] alatt olvashatunk.

dig elfogadásra került. A build eredményeit, részleteit a TFS-ben bármikor részletesen megtekinthettem.

A TFS további nagy előnye volt számomra, hogy ha fejlesztés közben a program bármely részében rendellenességet észleltem, azonnal hibajegyet tudtam feladni, amit alkalom adtán behúzhattam az aktuális iterációba, így az nem merült feledésbe. Megfelelő beállítások mellett a TFS a Visual Studioval tökéletesen együttműködik ezzel is megkönnyítvén a fejlesztési és a dokumentációs feladatokat. A TFS rendszerről néhány demonstratív célzatú képet találhatunk az E függelékben.

2.2. NuGet csomagok

A program megírásához leginkább saját kódokat készítettem, ám ahol azt célszerűnek láttam, külső könyvtárakat alkalmaztam. Ezeket a következő NuGet csomagok formájában adtam a projektemhez mint függőségeket:

2.2.1. Külső csomagok

NUnit és NUnit3TestAdapter. A legelterjedtebb teszt-keretrendszer .Net-es körökben, mely sokkal egyszerűbb egységtesztelést biztosít, mint a Visual Studio beépített tesztrendszere, valamint kifejezőbb tesztleírást tesz lehetővé a változatos metódusai által.

Moq. Széles körben elterjedt C#-os mock keretrendszer, mely szintén a tesztelés folyamatát könnyíti meg. Széles körben biztosít mockolási lehetőségeket megkímélvén a felhasználót a fake-osztályok létrehozása nyújtotta felesleges nyűgöktől.

2.2.2. Saját csomagok

Saját korábbi és aktuális munkáimat is felhasználtam a projekt készítésekor, melyeket szintén NuGet csomagok formájában referáltam be a jelen munkában. A csomagok nyilvánosan nincsenek közzétéve, azonban a [6] alatt található linken elérhetők.

AsanyadEncoder. Saját fejlesztésű titkosító rendszer, melyet már több alkalommal használtam. A modern titkosító rendszerekkel ellentétben ennek a titkosításnak nem a kulcson, hanem az algoritmuson alapul az ereje. Üzenetküldésre természetesen nem alkalmas, de lokális használatra kiválóan üzemel. Bármilyen szöveget decimális számjegyek sorává konvertál, melyek közül egynek a megváltoztatása is az eredeti üzenet megsemmisülését vonja maga után. Érdekessége, hogy ugyanarra a bemenetre minden alkalommal más kimenetet generál, viszont az mindig egyértelműen visszafejthető. A csomagot a megoldó-tesztek eredményeinek elmentésekor alkalmazom, hogy azok ne legyenek meghamisíthatók.

GrapherDevelopersLibrary. A feladat szempontjából a legfontosabb csomag, mely egy interfész-gyűjteményt tartalmaz. Ezek a publikus felületek egyrészt gráfdefiníciók, másrészt olyan megoldó interfészek, melyeket implementálva és dll-be fordítva a jelen dolgozat keretein belül megvalósított program képes felismerni, alkalmazni, és automatikus tesztnek alávetni.

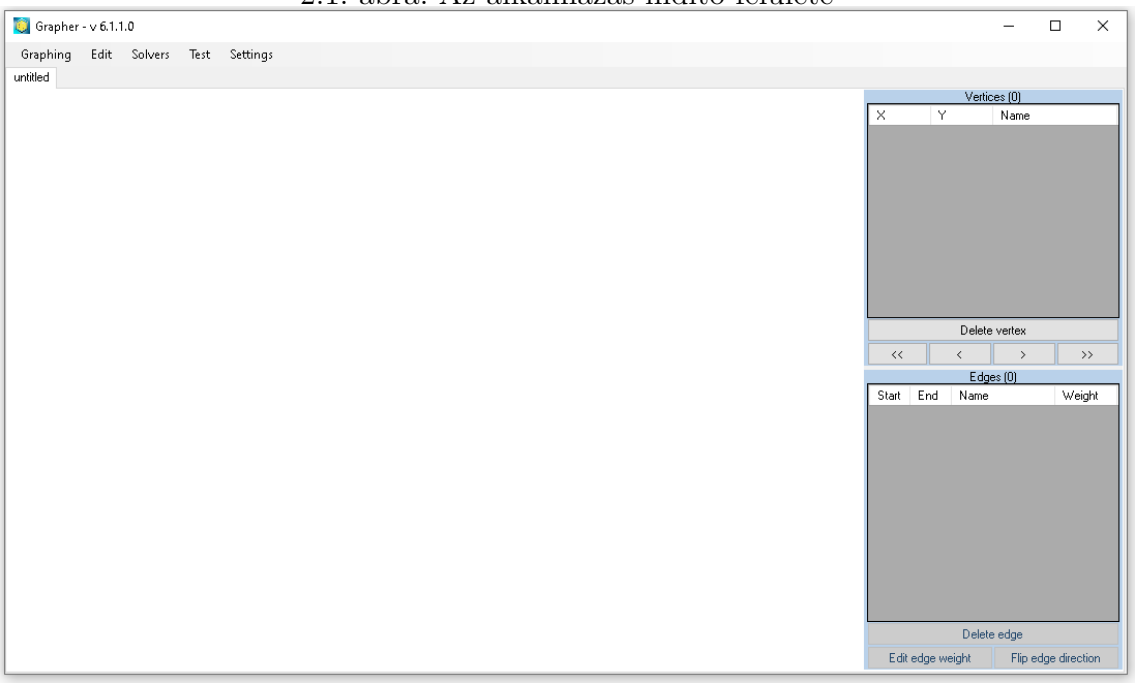
WindowsFormsBinder. A dolgozattal egy időben induló, azzal párhuzamosan fejlődő csomagról van szó, melyet a jelen szoftver aktívan alkalmaz ugyan, mégis sokkal inkább tekintek rá külön projektként, mint a dolgozat szerves részére. Ezen csomag nyújt megoldást a bevezetésben említett MVVM problémára. Lényegében az egyetlen publikus metódusa a változatosan paraméterezhető Binds metódus, melyet bármilyen WindowsForms-os control-on meghívhatunk imitálva ezáltal a WPF adta XAML kódba ágyazott kötéskialakításokat. A kötések a WPF-es technológiával ellentétben nem fordítási, hanem futásidőben kerülnek kialakításra, azonban ennek én mint fejlesztő csak minimális hátrányát láttam, a program felhasználója pedig egyáltalán nem érzékeli ezt. A csomag hátteréről minimális információként csak a kötések kialakításának elvét tartom szükségesnek megemlíteni, melynek mibenléte az összekötött control-ok és property-k egymás eseményeire való reflection általi feliratkoztatásában rejlik.

2.3. Főfolyamatok

Az elkészített alkalmazás elindításakor a 2.1. ábrán látható felület fogadja a felhasználót. Itt alapvetően egy rajzoló felületet láthatunk, viszont a fenti szalagmenün, illetve billentyű-kombinációk által a többi funkcionalitás is elérhető, melyek – a teljesség igénye nélkül – a következők:

- További rajzfelületeket nyithatunk, melyeken korlátlan számban, egymástól függetlenül szerkeszthetünk gráfokat egy felhasználóbarát környezetben.
- A megrajzolt gráfjainkon a programba belefoglalt gráfelméleti problémákra kereshetünk megoldást akár a beépített megoldók segítségével, akár külső implementáció hozzáadásával. A külső megoldókat természetesen eltárolja az alkalmazás, és a következő használatkor azok automatikusan rendelkezésre állnak.
- Külső könyvtárban található megoldókat (de akár beépítetteket is) lehetőségünk van automatikus tesztnek alávetni, melyhez több tesztelési paraméter beállítása is módunkban áll. A megoldó-tesztelés elméleti hátterének magyarázatát a felhasználó felé az ezzel kapcsolatos Súgó közvetíti, míg mi a 2.6. szakaszban olvashatunk róla. A teszteredményeket a program grafikus formában jeleníti meg, mely kiváló vizuális lehetőséget nyújt különböző megoldó algoritmusok összevetésére.

2.1. ábra. Az alkalmazás indító felülete



2.3.1. Felületek

Az alkalmazás 8 (, vagy felfogástól függően 10) különböző grafikus felhasználói felületet tartalmaz a felhasználóval történő interakciót segítendő. Ezek részletezésére itt nem kerül sor, egyrészt az A függelékben található felhasználói útmutatóban láthatók alkalmazásuk részletes leírásával együtt, másrészt javasolt őket élesben birtokba venni a program kipróbálása során.

2.3.2. MVVM és tesztek

Az előző pontban említett felületek mindegyikéhez egy-egy ViewModel van kötve az MVVM minta értelmében. A ViewModel-ek a felület minden információját publikus property-kben tárolják, és a felületről kezdeményezhető minden interakciót command-okon formájában hajtanak végre. Ezen kialakításnak köszönhetően több előnyre is szert tettem.

- A program egészét tudom programatikusan használni felhasználói felület megnyitása nélkül, így akár teljes, automatizált főfolyamat-tesztek is futtathatók.
- A felületek nélküli alkalmazhatóság folyományaként természetesen lehetőségem nyílik a jövőben teljesen új arculatot adni az alkalmazásnak pusztán a ViewModelek interfészeinek ismeretében.
- A ViewModelek mint az alkalmazás főfolyamatait leíró egységek implementációi teljes tesztlefedettség mellett készülhettek el (leginkább TDD által, néha utólagos

teszteléssel), mely megadja a lehetőséget, hogy bármikor változtathassak a kódon felesleges rizikó vállalása nélkül. Az alkalmazás főfolyamatainak tesztlefedettségét a program jelen állapotában bő 600 unit- és integrációs teszt biztosítja.

Az MVVM egyetlen negatívumaként azt tudom megemlíteni, hogy összetettebb felhasználói felületekhez igen nagy méretű és sok funkciót ellátó ViewModel-ek tartoznak, amely sehogyan sem egyeztethető össze az egy felelősség elvével, bár ennek még sosem találkoztam hátulütőjével. Természetesen a funkcionalitások mentén egy ViewModel széttagolható több kisebb egységre, ám gyakorlati tapasztalatom alapján ez több fejtörést okoz, mint amennyi hasznot hoz.

Az MVVM mintáról és WPF-es megvalósításáról részletesen a [2]-ben olvashatunk.

2.4. ObjectFactory

Az elkészített alkalmazásban igen fontos elemnek tartom az ObjectFactory névre keresztelt osztályt, mely sok nehézségtől mentett meg. Főként a tesztelhetőség és karbantarthatóság érdekében a ViewModel-ek konstruktorban kapják meg minden függőségüket, melyeknek nem ritkán egész sorát fel lehet sorakoztatni. Ezek értelmében egy-egy osztály már hajlamos átesni a ló túloldalára, és több soros konstruktorokat felvonultatni rengeteg objektum beinjektálásával, ami a fejlesztői szemnek sem kellemes látvány már.

Miképp a felhasználó interakciói nyomán az egyik felület nyílik a másikból, úgy a ViewModel-ek is egymást példányosítják, így a szülőnek már a létrehozott gyermeke függőségeit is meg kell kapnia, hogy továbbadhassa a gyermek példányosításakor, mely szintén csak „feleslegesen” bővíti a konstruktorok paraméterlistáját.[2]

Ezen problémára jelent megoldást az ObjectFactory osztály. Az ObjectFactory az alkalmazás indításakor kerül példányosításra, és az alkalmazás fő ViewModel-je ezt az egyetlen objektumot kapja meg konstruktorban. A további ViewModel-ek függőségei szintén redukálhatóvá váltak erre az egy egyszerű objektumra, melyet a szülőtől megkapnak, és az esetlegesen példányosított gyermekeiknek ők is továbbadhatnak.

Az ObjectFactory feladatköre egyszerű. Ha valamilyen objektum egy példányára van szükségünk, akkor tőle kérjük ezt el. Valójában interfész típusokat igénylünk tőle, de a megfelelő implementáció egy példányát adja vissza a céljaink szerint felparaméterezve. Ha valamely osztályból csak egy példányra van szükség a program egész élete során, akkor mindig ugyanazt a letárolt példányt adja vissza mintegy singleton jelleggel.

[2] Jelen kontextusban a szülő-gyermek kapcsolat alatt ne az öröklődés elvében megfogalmazott ős-leszármazott kapcsolatot értsük, hanem szülőre mint példányosító, gyermekre pedig mint példányosított osztályra tekintsünk!

Ha valamely interfészre új implementációt készítenék performancia-javítás céljából, vagy egyéb más okból, akor kizárólag az ObjectFactory-ban kellene módosítást eszközölnöm, mert ez az egyetlen hely, ahol példányosításra kerül. Egy kis túlzással ugyan, de elmondható, hogy a jelen alkalmazás esetében az ObjectFactory egyfajta DI-konténer szerepét tölti be, de mindenképp igaz, hogy a program inicializációs bedrótozásait végzi el.

Maga az ObjectFactory, és az általa visszaadott objektumok is interfészek mögé vannak rejtve, így mock-olásuk megoldott, mely jelentős könnyedséget eredményezett az egység- és integrációs tesztek megírásában.

2.5. Megoldók

Az alkalmazás egyik célkitűzése, hogy beépített, alapértelmezett megoldók által biztosítsa a felhasználóban felmerülő és a gráfok nyelvén megfogalmazott problémáinak a megoldását, és ennek tükrében az 1.2. pontban felsorolt gráfelméleti problémákra nyújtson megoldó algoritmusokat. A beépített megoldók ismert algoritmusok alapján működnek, melyek biztos megoldást garantálnak, ám ez nem feltétlenül társul hatékonysággal.

A hatékonyság hiánya kimondottan az Euler és Hamilton problémák esetében igaz, hisz ezek egyszerű brute-force algoritmusok, így a gráf méreteinek növekedésével a megoldás időigénye faktoriális mértékben növekszik. Az élek (illetve a csomópontok) permutációit veszik sorra, és ellenőrzik, hogy az adott permutációban azok megvalósítható bejárást jelentenek-e.

A többi probléma esetén a beépített megoldók elméleti hátterét, működési elvét a a [3]-ből merítem, implementációját pedig saját elgondolásom szerint valósítom meg.

Maximális átfolyás problémájára a Ford–Fulkerson algoritmus nyújt megoldást, legkisebb költségű feszítőfa előállítását a Kruskal algoritmus implementációja biztosítja, míg a két csomópont közötti legrövidebb út felkutatása a Dijkstra algoritmus által történik.

A jobb karbantarthatóság és könnyebb módosíthatóság érdekében a hasonló jellegű problémák (vagyis az Euler és Hamilton problémák 4-4 variánsa) esetében természetesen kihasználtam a öröklődés mint objektum-orientált alapelv adta lehetőségeket, így a megoldó algoritmusok jelentős része egy absztrakt ősosztályban került implementálásra, és a gyermekosztályok csak a kód azon minimális részét tartalmazzák, melyekben eltérés jelentkezik egymáshoz képest. Ezen stratégia pattern szerinti implementáció az alkalmazás kódjában több helyen is fellelhető, hisz nem egyszer nyújtott segítséget a fejlesztés során.

2.6. Megoldó tesztelés és az új metrika

Jelen szakaszban arra szeretnék kitérni, hogy miként sikerült elérnem a projekt és a dolgozat fő célját, avagy miként valósítottam meg megoldók automatikus tesztelését, és miként definiáltam összehasonlíthatósági alapot két, ugyanazon gráfelméleti problémára készített megoldó algoritmus számára. A témát főként elméleti síkon közelíteném meg, mert – bár az implementáció terén az alkalmazás ezen modulja bizonyult a legfőbb kihívásnak és a legtöbb bonyodalom forrásának – személyes véleményem szerint a projekt jelen pontján az elméleti háttér kidolgozása képvisel nagyobb értéket, és talán a dolgozat legfontosabb részeként tekinthetünk rá.

2.6.1. Mi a sikeres megoldás, mi a tesztpéldány?

2.1. Definíció (Sikeres megoldás)**.** Tekintsünk adottnak

- egy problémát,
- egy gráfot, melyre az adott probléma értelmezhető és megoldható, és
- egy algoritmust, melynek célja, hogy az adott gráfban megoldást találjon az adott problémára.

A megoldási folyamatot akkor nevezem sikeresnek, ha az algoritmus a gráfban értelmezett problémára egy előre meghatározott időn belül megoldást talál.

2.2. Definíció (Tesztpéldány)**.** Az említett, egy adott gráfon egy adott megoldóval egy adott problémára irányuló egyszeri megoldási kísérletként leírható folyamatot nevezem tesztesemény-példánynak, vagy röviden tesztpéldánynak.

2.6.2. A tesztgráf leírása

A bevezetőben említett példa esetéből is érzékelhető, hogy egy megoldót leginkább azzal jellemezhetünk, hogy mekkora gráfméretig ad sikeres megoldást[3] az adott probléma esetében.

„300 csomópontig stabilan működik az algoritmusom.” – hangozhatna el egy megoldó jellemzése során, de ha kicsit megpróbálunk mélyebben belegondolni a jelentésébe, a mondat eléggé problémásnak hangzik a többedik átgondolásra.

Gondolatmenetem az elkövetkezendőkben szeretném egy példa mentén kifejteni, melyhez vegyük alapul a Hamilton-kör problémát (az egyszerűség kedvéért irányítatlan gráfok esetében).

A szóban forgó 300 csomópontos gráf hány élet tartalmaz? Amennyiben 300-at, és a gráfban található Hamilton-kör, a megoldótól nem nagy teljesítmény, hogy azt

[3] A sikeres megoldás jelentését a 2.6.1 szakasz írja le.

megtalálja egy előre definiált rövid időn belül. Viszont ha az adott gráfban 40000 él található, és azok közül találja meg az algoritmus azt a 300-at, melyek a Hamilton-kört alkotják és mindezt ugyanilyen hatékonyság mellett, akkor az jobban felkelti az érdeklődésünket.

Ezek alapján megállapítást nyer, hogy adott probléma megoldását szolgáltató algoritmus esetén azon gráfokat, melyeken az algoritmus tesztelve lett, legalább két adattal kell jellemeznünk, melyek alapesetben értelemszerűen – a fenti példából kiindulva – lehetnek a csomópont- és az élszámok. Természetesen figyelembe vehetnénk több leíró adatot is, például hogy bizonyos nevezetes minták előfordulnak-e az adott gráfban, vagy hogy a gráf valamely ismert gráfcsaládba tartozik-e, de jelen tanulmány keretein belül szorítkozzunk az él- és csomópontszámok általi jellemzésre.

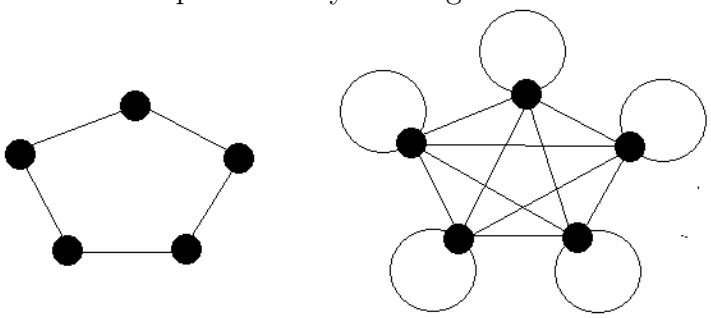
2.6.3. A leírás hibája

Egy adott tesztpéldány esetében a kimenetel jellemzése legegyszerűbben egy predikátum formájában adható meg: Találtunk, avagy nem találtunk megoldást adott időn belül. Egy megoldó algoritmus eredményességét az eddigiek figyelembe vételével a következőképp szemléltethetjük hatékonyan: A Descartes-féle síkbeli derékszögű koordinátarendszer tengelyeihez él-, illetve csomópontszámokat rendelünk, így az első síknegyed minden (pozitív egész koordinátákkal rendelkező) pontja egy-egy tesztpéldányt jelent, ahol a megoldás kimenetele két értékkel (0 és 1 számokkal, piros és zöld színekkel, vagy egyéb módon) megjeleníthető. Ez a fajta ábrázolásmód hatékony és szemléletes lehet a megoldó algoritmus jellemzésére, vagy több algoritmus hatékonyságának összehasonlítására.

Az ábrázolás azonban egy hibát rejt magában. A Hamilton-kör problémánál mint példánál maradva, kijelenthetjük, hogy a megoldó olyan gráfot vár bemenetként, amiben nincsenek felesleges, duplikált élek, vagy ha vannak is, első dolga, hogy ezeket figyelmen kívül hagyja, mint ahogyan a GPS alkalmazások útvonal-tervezésekor sem játszik szerepet, hogy a 200 km hosszú autópálya-szakaszon a három sáv közül melyiken haladunk. Hamilton-kör probléma esetén a V csomópontos irányítatlan gráfban minimum $E_{min} = V$ és maximum $E_{max} = \frac{V(V+1)}{2}$ egymástól különböző él található. $V = 5$ esetén a 2.2. ábrán látható, hogy ezt hogyan is értelmezhetjük szemléletesen.

$V = 5$ esetében $E_{min} = 5$, és $E_{max} = 15$, míg egy nagyobb csomópontszám, például $V = 300$ esetén $E_{min} = 300$ és $E_{max} = 45150$, vagyis látható, hogy ha az egyik tengelyen haladunk pozitív irányba a csomópontok számának növelésével, akkor az egyes értékekhez a másik tengelyen értelmezett élszám-tartományok csak minimális mértékben, vagy egyáltalán nincsenek fedésben, sőt a példa alapján nagyságrendi eltérések várhatók. Amennyiben ábrázolni szeretnénk az $V = 300$ csomóponttal és $E = 300 \div 45150$ éllel rendelkező gráfokon elvégzett tesztek eredményeit, az $V = 5$ csomópontszámú grá-

2.2. ábra. 5 csomópontos irányítatlan gráfok Hamilton-körrel



fokhoz tartozó eredmények az élek számától függetlenül láthatatlanná zsugorodnának a felületen. Valamelyest megoldást jelenthet erre a problémára a koordináta-rendszer logaritmikus skálázása, ám ez sem oldaná meg azon problémát, hogy a síknegyed egy sávban kerülne csak lefedésre a megoldások eredményei által.

2.6.4. A hatékony metrika

Vezessünk be két új fogalmat, melyek által szintén egy gráf méretét adhatjuk meg. Vegyünk egy, a gráfot és a rajta vizsgált problémát együttesen jellemző adatot, a Hamilton-kör probléma eseténél maradva legyen ez a csomópontszám (amely mindenképpen megegyezik a Hamilton-kört alkotó élek számával, azaz látható, hogy tényleg probléma-specifikus adatról van szó), és nevezzük ezt alapméretnek, jele pedig legyen B, azaz $B = V$. Másodsorban vegyük az ehhez tartozó lehetséges minimális és maximális élszámokat (E_{min} és E_{max}), és definiáljunk egy szaturáció (vagy telítettség) elnevezésű jellemző mértéket (és jelöljük így: S), melyre igaz, hogy E_{min} esetén $S = 0$, E_{max} esetén pedig $S = 1$. Egyéb $E_{min} < E < E_{max}$ esetében S értékének meghatározása lineáris interpolációval történik, azaz

$$S = \frac{2(E - B)}{B(B - 1)} \tag{2.1}$$

A későbbiekben számunkra sokkal fontosabb lesz az élek száma, mely az összefüggésből kifejezve a következő:

$$E = \left[B \left(1 + (B - 1)\frac{S}{2} \right) \right] \tag{2.2}$$

Az összefüggésben fontos az egészre csonkolás, hiszen egy gráfban az élek száma csakis nemnegatív egész értéket vehet fel.

Az alapméret és a szaturáció bevezetésének haszna és ereje abban áll, hogy jelen esetben is két mérőszámmal jellemezzük a tesztgráfunkat, azonban bármely $B > 0$ alapméretű gráf esetén a másik jellemző méret, azaz a telítettség értéke 0 és 1 közötti tartományban marad, így bármely tesztgráfon kapott eredményünk ugyanolyan látha-

tósággal és helyfoglalással jeleníthető meg a derékszögű koordináta-rendszerben, mint egy tetszőlegesen választott másik gráf eredménye, a teszteredményeink ábrázolására szánt terület lefedettsége pedig tökéletesen egyenletes lehet megfelelő paraméterek kiválasztása mellett.

Fontos megjegyezni, hogy az alapméret és a telítettség definíciója önkényes, és adott gráfelméleti problémához kötött. Amennyiben nem Hamilton-kör problémáról, hanem más megoldandó feladatról van szó, biztosan állíthatjuk, hogy a (2.1) és (2.2) összefüggések érvénytelenek lesznek, azonban más definíció mellett bármely gráfelméleti probléma esetén meghatározhatók hasonló előnyökkel járó mérőszámok. A jelen dolgozat keretén belül készített alkalmazás által ismert megoldandó problémák esetére az alapméret- és szaturáció definíciókat a D függelékben találhatjuk. A függelékben az is kiderül, hogy adott probléma esetén több definíció is létezhet, melyek között nem érdemes rangsort állítani, egyik sem jobb, vagy rosszabb, hogy melyiket alkalmazzuk, az pusztán döntés kérdése.

2.6.5. Tesztesetek és alesetek

2.3. Definíció (Aleset)**.** Teszt-alesetnek (röviden alesetnek) nevezem azon eseményt, mely során konkrétan adott B alapméretű és S telítettségű gráfokra megvizsgálom egy adott probléma megoldására irányuló algoritmus hatékonyságát. Egy aleset során több (n darab) azonos metrikával jellemzett gráfon is kipróbálhatom az algoritmust (emlékezzünk, ezeket neveztük tesztpéldányoknak), melyből k darab sikeres megoldás esetén a k/n értéket nevezem az aleset eredményének.

2.4. Definíció (Teszteset)**.** Tesztesetnek azon eseményt hívom, mely során adott B alapméretű gráfok által tesztelem a megoldót oly módon, hogy alesetek sorát vonultatom fel a telítettség 0-tól 1-ig történő folyamatos változtatása mellett. A teszteset eredményét a benne foglalt alesetek eredményeinek számtani közepeként értelmezem.

2.6.6. Generátorok

Az algoritmusok teszteléséhez használt gráfok természetesen nem előre definiáltak, a programban ezeket generátor osztályok állítják elő. Minden, a programban szereplő problémához egy-egy ilyen generátor osztály került implementálásra, mely garantáltan a bemenő adatoknak – azaz alapméretnek és telítettségnek – megfelelő, ám véletlenszerű gráfot képes legyártani úgy, hogy abban az adott problémára biztosan található megoldás.

A generátorok közös őstől származnak, és – mint megannyi mást, ami szintén a gráfprobléma jellegétől függ – factory példányosítja őket, így további probléma hozzáadása esetén a program könnyedén bővíthető a meglévő kódállomány változtatása nélkül.

2.6.7. Az algoritmus-tesztelés adatai és menete

A 2.6.5. és 2.6.1. pontokban definiált fogalmak birtokában már világosan leírható, miképp is történik egy megoldó algoritmus tesztelése. A felhasználótól egy adott probléma megoldása céljából írt algoritmust, azaz annak implementációját várja el az alkalmazás, valamint a következő négy tesztparaméter megadását:

t_{max}. Időkorlát, melyen belül az algoritmusnak megoldást kell találnia.

e_{min}. Minimális százalékos érték, mely fölött egy tesztesetet elfogadhatónak tekinthetünk.

n_s. Alesetek száma egy teszteseten belül, mely meghatározza az alesetekhez tartozó telítettség értékeket is.

n_i. Tesztpéldányok száma egy aleseten belül.

A fenti adatok ismeretében a megoldó tesztelése a következő lépések szerint történik: $B = 3$ alapmérettel felveszünk egy tesztesetet, azon belül pedig $S = 0$ telítettséggel egy alesetet. Példányosítunk egy megfelelő gráfgenerátort, és az aleseten belül n_i darab tesztpéldányt futtatunk, melyekhez a generátor szolgáltat megfelelő tesztgráfokat a (B, S) bemenő adatpárnak megfelelően. Minden tesztpéldány esetében három kimenetel lehetséges:

- Az algoritmus megtalálja a megoldást.
- Az algoritmus őszintén bevallja, hogy nem tudta a problémát megoldani.
- Az algoritmus túllépi a t_{max} időkorlátot.

Természetesen ezek közül csak az elsőt tekinthetjük sikeres tesztpéldánynak. Amennyiben mind az n_i darab tesztpéldány lefutott, az aleset véget ért, és eredménye a sikeres tesztpéldányok száma és n_i hányadosaként áll elő. Az aktuális szaturáció $1/(n_s - 1)$ értékkel történő növelésével egy újabb alesetet definiálunk, melyre a fentieket ismételjük meg. Mindezt addig tesszük, mígnem (B, 1) adatpárral jellemzett alesetet is késznek nem nyilvánítjuk.

Az alesetek eredményeinek átlagaként megkapjuk a teszteset eredményét, és kiértékeljük azt. Ha az eredmény e_{min} alatti érték, akkor a tesztnek vége, ellenkező esetben pedig B inkrementálása mellett $(B = 4)$ felveszünk egy újabb tesztesetet, melyben azonos módon járunk el, mint az előző teszteset során. A B érték növelése nem folytonos, hanem a 2.1. algoritmus szerint működik, mely a következő sorozatot generálja:

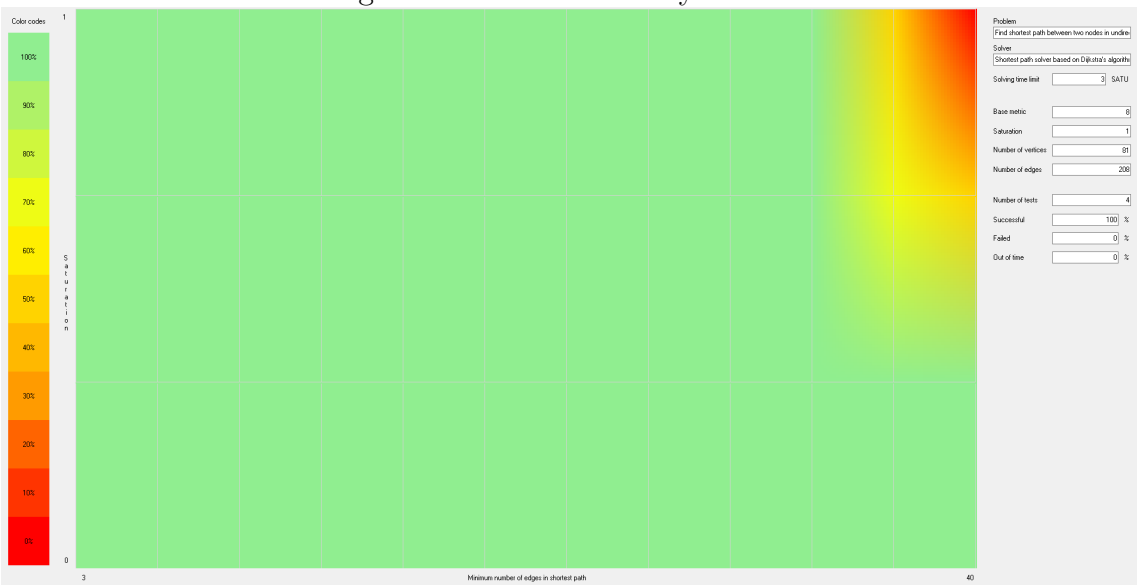
3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, ...

kód 2.1. Alapméret inkrementálása

```
1   ELJARAS INIT()
2       B<-3;
3       Binc<-1;
4   E_VEGE
5
6   ELJARAS INCREMENT()
7       B<-B+Binc;
8       HA B=10*Binc AKKOR
9           Binc<-10*Binc;
10      H_VEGE
11  E_VEGE
```

A megoldó tesztelésének végeztével, azaz mikor az utoljára vizsgált tesztesetünk eredménye már nem bizonyult megfelelőnek, akkor a kapott eredményeket grafikus formában tárjuk a felhasználó elé, ahogy az a 2.3. ábrán is látható. Ez a fajta megjelenítésmód a vizuális mivolta miatt egyrészt könnyebben befogadható és értelmezhető a felhasználó számára, mint egy hatalmas számhalmaz, másrészt pedig megfelelő alapot nyújt különböző algoritmusokon végzett tesztek eredményeinek összevetésében.

2.3. ábra. Megoldó tesztelés eredménye vizuális formában



A teszteredmények exportálására és importálására is lehetőséget biztosít az alkalmazás, melynek kettős haszna van. Egyrészt a szöveges formátumban tárolt teszteredmény könnyedén elküldhető egy másik fél számára, aki azt beolvasván ugyanazon vizuális megjelenítésben láthatja a minden fontos adatot tartalmazó eredményeket, mint mi, míg egy képernyőkép küldésével az összes mögöttes, számszerűsített információ elveszne. Másrészt pedig fontos a gyorsaság. Ha egy adott megoldóhoz tartozó, korábban elvégzett tesztek eredményeit szeretnénk újra elővenni, akkor csak be kell olvasnunk

azt egy korábbi exportból, hisz valósággal abszurd lenne, ha újra kellene futtatni az adott tesztet, mely esetenként akár órákban mérhető hosszúságúra is nyúlhat.

Az exportált eredményeket illetően szintén vállalhatatlan lenne, ha azokat mint szöveges dokumentumokat bárki – mintegy meghamisító szándékkal – átírhatná, és úgy küldené el egy másik fél számára. Ezen problémát megelőzendő az eredmények a 2.2. pontban említett AsanyadEncoder csomag által generált titkosított formában kerülnek kiírásra.

2.6.8. SATU

A tesztelésnek még egy kulcsfontosságú paramétere van, melyről ezidáig nem esett szó. Az 1.1. pontban, a lehetséges technológiák megfontolása során említésre került, hogy a tesztelt algoritmus adott gráfon végzett munkájának számítási időigénye nem függhet külső tényezőktől, amilyen például a szerver terheltsége egy esetleges webalkalmazás készítése esetén. Hasonló problémával találhatjuk szembe magunkat asztali alkalmazás keretein belül is, hiszen ugyanazon lépések elvégzésére más-más időigénye lehet különböző számítógépeknek a belső architektúrájuk függvényében. Hogy egy megoldó algoritmust hatékonynak találunk-e az elvégzett tesztek alapján semmiképp sem állhat, vagy bukhat azon, hogy milyen teljesítményű gépen vizsgáljuk azt.

A fenti probléma kiküszöbölése céljából vezettem be egy architektúra-független időegységet, mely a SATU (Solver Acceptance Time Unit) nevet kapta, és a következőképp definiálható:

2.5. Definíció (SATU)**.** Vegyünk egy dedikált tesztgráfot, melyet a 2.4. ábrán láthatunk. A gráf éleire írt számok jelen esetben nem súlyokat jelentenek, hanem azon sorrendet, amely szerint felvételre kerültek a gráf definiálása, avagy megrajzolása során. A csomópontok megadását illetően ezen sorrend maguk a csomópontok nevéből látható, és növekvő ábécé-rend szerint értendő.

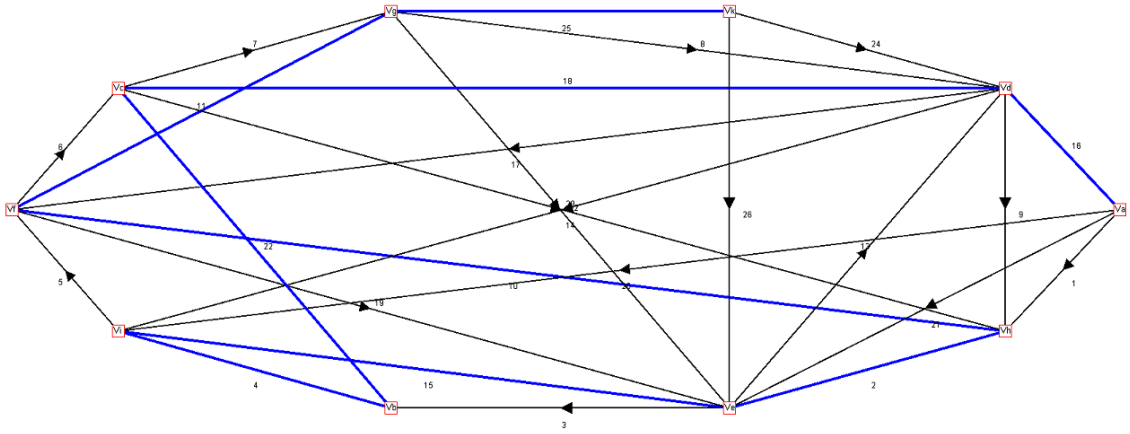
Vegyük az alkalmazás alapértelmezett (beépített), permutációs elvű, irányítatlan gráfokra implementált Hamilton-kör megoldóját.

Nevezzük 1 SATU-nak azon időtartamot, melyre az említett megoldónak szüksége van, hogy a tesztgráfban Hamilton-kört találjon.

A program használata során, mikor a felhasználó a megoldó automatikus tesztelésének paramétereit határozza meg, a t_{max} időkorlátot nem szekundumban, vagy milliszekundumban, hanem SATU-ban értelmezve kell megadnia. Ehhez elengedhetetlen, hogy a felhasználó tisztában legyen a SATU jelentésével, és a tesztelési folyamat elméleti hátterével. Ehhez nyújt segítséget a programba beépített Súgó, melyben éppen ezen fogalmak jelentésének magyarázatát lehet elolvasni.

A SATU aktuális konfiguráción vett értéke a program indításakor egy külön szálon kerül meghatározásra. A tesztgráfot 20 alkalommal adjuk át a megoldónak, és mérjük

2.4. ábra. Tesztgráf SATU méréséhez



a megoldáshoz szükséges időket, majd ezek átlagát tekintjük a SATU értékének. A felhasználó ebből a folyamatból csak annyit tapasztalhat, hogy a Teszt menüpont alatt néhány funkció inaktív a program indítását követő pár másodpercben, míg ez a mérés lefut.

Amennyiben feltételezhetjük, hogy a program teljes futásidejében az adott számítógépen ugyanazon erőforrások állnak rendelkezésre, azaz nem indít, vagy zár be a felhasználó nagy memória- és processzorigényű alkalmazásokat, a SATU értékét állandónak tekinthetjük.

A SATU-t alkalmazva, és a fent megfogalmazott feltételt betartva három különböző teljesítményű számítógépen futtatott, azonosan paraméterezett automatikus teszt ugyanazon megoldóra – minimális eltéréssel – ugyanazt a teszteredményt szolgáltatta, mely számomra elég meggyőző volt a SATU mint időegység létjogosultságáról és hasznáról az alkalmazásban.

Természetesen továbbra is fennáll a lehetősége annak, hogy a felhasználó időközben új alkalmazásokat indít, és elvonja az erőforrásokat a megoldó-tesztelőtől, de a jelen verzió ezt a lehetőséget nem veszi figyelembe. A 4.3. szakaszban említett továbbfejlesztési lehetőségek között említek megoldási javaslatokat ezen probléma kapcsán, melyeket érdemesnek tartok megfontolni, esetleg kivitelezni az alkalmazás egy későbbi verziójában.

2.7. Problémák és megoldások

Ezen szakaszban néhány, a fejlesztési folyamat közben felbukkanó, általam érdekesnek, vagy említésre méltónak tartott elméleti és implementációs fogást, esetlegesen felmerülő problémát és azokra eszközölt megoldást szeretnék bemutatni.

2.7.1. NameGenerator

Akár a manuálisan, grafikus úton definiált, akár a generátor osztályok által létrehozott gráfokat tekintjük, mind a csomópontok, mind az élek megkülönböztethetőségét biztosítani kell. Generált, felhasználó számára nem elérhető gráf esetében ez nem fontos tényező, hisz a csúcs- és élobjektumok különbözősége a referenciáik által automatikusan biztosított, ám a grafikus felületen kirajzolt gráfok tekintetében a felhasználónak igencsak fontos, hogy annak elemeire hivatkozhasson, ezért van minden él és csomópont egyedi névvel ellátva. A nevek automatikusan generáltak, megfelelő sorrendben követik egymást, és biztosan egyedik.

A neveket a NameGenerator osztály biztosítja melynek működése egy érdekes, kompozit-szerű mintát követ. A generátor egy karaktertömböt kap inicializáláskor, melyet mint ábécét használ, és van egy privát mezője, melynek típusa megegyezik a saját típusával. Ha kérünk tőle egy nevet, akkor visszaadja az ábécéje soron következő karakterét, majd ha annak a végére ér, újra az első karakterrel kezdi, viszont példányosítja a saját belső NameGenerator-át, és attól kér egy előnevet. Ahányszor körbeér az ábécén, mindig új előnevet kér a belső példánytól, mely azonos módon viselkedik, és szükség esetén újabb belső példányt hoz létre.

A megvalósítás előnye, hogy a generátor sosem fogy ki a nevekből, és nincs szükség előre definiált hosszokra, hisz először az adott ábécéből képezhető egy karakteres nevek kerülnek kiosztásra, majd a kétkarakteresek, és így tovább.

2.7.2. Timer

A megoldó-teszteléskor szükség van a megoldási idő mérésére. Egyrészt le kell állítani az aktuális tesztpéldányt, ha a megoldásra fordított idő már túllépte a t_{max} határt, és sikertelennek kell a tesztpéldányt minősíteni, másrészt ellenkező esetben, azaz az algoritmus sikeres lefutásakor szintén tudni kell, hogy a megoldási idő milyen hosszú volt. A .Net keretrendszerben található System.Timers.Timer osztály Elapsed eseménye kielégíti az első szükségletet, azaz a t_{max} idő elteltekor elsül, viszont tetszőleges pillanatban nem kérdezhetjük le tőle, hogy aktuálisan mennyi idő telt el az indítás óta. System.Diagnostics.Stopwatch osztály pedig pont a másik követelménynek tesz eleget, vagyis az indítás pillanatától eltelt időt bármikor visszaadja az ElapsedMilliseconds property által, viszont nem ad lehetőséget arra, hogy értesüljünk t_{max} idő elteltéről.

A probléma megoldására saját Timer osztályt implementáltam, mely nemes egyszerűséggel tart egy-egy példányt a fenti két osztály mindegyikéből, és együtt kezeli – azaz indítja el, és állítja meg – őket. A .Netes Timer példány Elapsed eseményére feliratkozva annak bekövetkeztére saját Elapsed eseményét triggereli, ElapsedMilliseconds property-je pedig visszaadja a StopWatch példány azonos nevű property-jéből kinyert értéket.

2.7.3. Hibakezelés és logolás

Ha egy réteggel a felhasználói felületek mögé tekintünk, láthatjuk, hogy a program használata alapvetően a ViewModel-ek property-jeinek történő értékátadásokkal, majd command-jainak végrehajtásával történik. A command-ok hívásakor bármi is mellékvágányra csúszna, az alkalmazás nem kezelt kivétel miatt leállna, így első körben az összes command belső logikáját try-catch blokkokba foglaltam, ahol a catch ág minden esetben ugyanazt a hibakezelő eljárást hívta meg. Ez a kódsorok számának felesleges növekedéséhez, és szépnek semmiképp sem nevezhető kódismétléshez vezetett.

A kód refaktorált verziójában a ViewModel-ek közös ősosztályában helyet kapott egy ExecuteAndHandle metódus, melynek érdekessége, hogy egy Action<object> delegétet vár paraméterként, és ugyanezen típussal tér vissza mégpedig a bemenő függvényreferencia hibakezelt változatával. Ezáltal egy szebb kialakítású, jobban formázott kódot értem el a ViewModel-ekben, ahol a command-ok definiálásakor azok bemenő Action<object> paramétereit úgy adtam meg, hogy az ExecuteAndHandle metóduson keresztülmenjenek. Try-catch blokkot így kizárólag egy helyen, az ősosztályban kellett alkalmaznom, és a hibakezelő eljárás is csak ugyanezen az egy helyen kerül meghívásra.

Maga a hibakezelő eljárás két feladatot végez el. Egyrészt lekezeli a kivételt, és megmutatja annak üzenetét a felhasználó számára, másrészt átadja a kivételt a logoló osztály LogError metódusának, mely az aktuális időpont rögzítése mellett egy logfájlba írja annak, és esetlegesen belecsomagolt belső kivételeknek az üzenetét rekurzív módon.

A logoló osztály implementálása természetesen a megszokott módon, singleton minta alkalmazásával történt.

2.7.4. Kommunikáció a felhasználóval

Azon esetekben, ahol az alkalmazásnak szüksége van egy, a felhasználó által közölt adatra, vagy válaszra, rendelkezésünkre állnak a WindowsForms által biztosított megoldások, mint a DialogBox-ok. A felhasználói reakció azonban egy olyan függőség a szoftverekben, mely a tesztelést megnehezíti, vagy inkább lehetetlenné teszi, így ezekkel a beépített megoldásokkal direkt módon nem lehetett élni.

A megoldást egy UserCommunicator nevű osztály jelenti, mely a háttérben ugyan ezen bevált technológiákat alkalmazza, kifelé viszont egy interfész mögé van rejtve, ezáltal mockolhatóvá válik, és a folyamatokat így teljes mértékben tesztelhetővé teszi.

A UserCommunicator-t felesleges minden alkalommal újra példányosítani, így ez is singleton minta alapján egyetlen statikus példányként van jelen a program élete során.

2.7.5. Szálkezelés

A program építése során több ponton is szükségem volt adott számítások mellékszálon való futtatására. A megoldó-tesztelés a leghosszabb időn át lejátszódó folyamatok egyike, és ennek során semmiképpen sem szabad megbénítani a felhasználói felületet, hisz egyrészt ez a lefagyott szoftver érzését keltené a felhasználóban, másrészt a tesztelés folyamatának aktuális állomásait pontosan a felületen kell megjeleníteni tudatván a felhasználóval, hogy éppen hol is tartunk a procedúrában. A felületen megtervezett gráfokon való megoldó algoritmus futtatásakor ugyanez a meglátás érvényes, sőt, a SATU mérésére szánt pár másodperces idő is már azon kategóriába esik, ahol indokolt a háttérszálon történő számítás.

A mellékszálak alkalmazása két problémát is generált a gyakorlatban. Egyrészt ezen háttérfolyamatok dolguk végeztével – vagy akár még az előtt, a folyamat során többször is – az eredményeiket közlik a főszál felé, mely azokat a felületen keresztül tudatja a felhasználóval. Ezt a nyelv sajnos nem támogatja, ugyanis a WindowsForms technológiában nem engedélyezett a felületi elemek tulajdonságainak módosítása (akár csak egy TextBox szövegének megváltoztatása), csakis a főszálból. A megoldást a control-ok Invoke metódusa jelenti, mely egy Action, vagy MethodInvoker delegétet vár paraméterül. A control Invoke metódusát bármely szálból meghívhatjuk, viszont a paraméterben kapott delegétet – melyben a végrehajtandó utasítás akár maga a control módosítása is lehet – már a főszálban futtatja.

Bár nem a dolgozat része, azért megjegyezném, hogy a fenti probléma a legnagyobb fejtörést a 2.2. szakaszban bemutatott WinFormsBinder csomag által biztosított Binds metódus hívásaikor jelentette, hisz a metódus felületi elemeket köt össze property-kkel, mely property-k számára nem garantálható, hogy csakis főszálban futó folyamatok adnak nekik értéket. A probléma a csomag megreformálását is magával vonta, és egy újabb publikus metódus került bele, a beszédes nevű BindsThreadSafe.

A háttérszálak használatából adódó másik probléma főképp külső megoldóval való számítás során bukkant fel. Amennyiben a megoldó algoritmus hibára fut, nem kezelt kivétel keletkezik, mely nem jut át a főszálra, nem kerül kezelésre, és ez a program leállásához vezet. Fontos volt megoldani, hogy ez ne így történjen, hisz a felhasználó ebből csak annyit érzékel, hogy a program megbízhatatlan, és nem a saját megoldójában keresi a hibát. Amennyiben Thread helyett Task példányt hozunk létre a háttérfolyamat futtatásához, annak ContinueWith metódusával megmondhatjuk, hogy milyen akció történjen a folyamat végén. Ezen akcióban definiálhatunk egy hibakezelést – mely az a jelen esetben egyszerűen a kapott kivétel főszálnak való továbbadása, hisz ott már van valódi hibakezelés – ehhez viszont szükséges a TaskContinuationOptions.OnlyOnFaulted flag felcsapása, mely által az akció kizárólag a háttérfolyamat hibás működése esetén lép életbe.

2.7.6. Permutátor

A permutációs elven működő beépített megoldók esetében szükség volt egy mechanizmusra, mely egy adott objektumokból (csomópontokból, vagy élekből) álló kollekciónak kérésre egy következő permutációját adja vissza. Ezt a funkciót az alkalmazásban egy generikus, Permutator nevű osztály látja el, mely példányosításakor egy n elemű kollekciót kap paraméterként, aminek a soron következő permutációját bármikor lekérhetjük a GetNextPermutation metódus által. Amennyiben már megkaptuk az összes lehetséges, $n!$ darab permutációt, következő kérésünk eredménye egy NoMorePermutation kivételt vált ki.

Az osztály a háttérben természetesen 0-tól $n-1$-ig terjedő számsort permutál, és ennek mint indexsorrendnek megfelelően adja vissza a generikus lista átsorrendezett elemeit. Viszont az aktuálisan következő indexsorrend előállítása komoly kihívást, és érdekes feladatot jelentett számomra. Kézenfekvőnek tűnhet Heap algoritmusának a használata, melyre könnyedén található C#-ban írt implementáció is, ám ez az algoritmus eredeti formájában visszaad minden egyes permutációt, nem csak a soron következőt, melyre éppen szükségünk van. Természetesen meg lehet vizsgálni őket egyesével, de ha már esetleg az elsők között megtaláljuk a nekünk megfelelő permutációt, akkor feleslegesen tároltunk el $n!$ darab n elemű kollekciót ($n = 50$ esetén például $3 \cdot 10^{64}$ darab 50 eleműt) a memóriában, ha egyáltalán van erre kapacitásunk. Szintén lehetséges megoldás lett volna az algoritmus átalakítása, hogy csak kérésre szolgáltassa a következő permutációt, de egyrészt korrektebb, másrészt érdekesebb feladatnak találtam egy saját permutációs algoritmus kidolgozását, melyet illetően végül sikerrel jártam, és a következőkben leírt elméleti lépések mentén valósítottam meg.

Induljunk ki egy n elemű kollekcióból, melyben az indexek 0-tól $n-1$-ig terjednek, és inicializáljuk az algoritmushoz szükséges adatszerkezeteinket a következőképp:

- Vegyünk fel egy n elemű P tömböt, melynek i-ik eleme i,
  ahol $0 <= i < n$, és $i \in \mathbb{Z}$. Például $n = 5$ esetén $P = \{0, 1, 2, 3, 4\}$.
- Vegyünk fel egy n elemű H tömböt, melynek i-ik eleme $(n - i - 1)!$,
  ahol $0 <= i < n$, és $i \in \mathbb{Z}$. Például $n = 5$ esetén $H = \{24, 6, 2, 1, 1\}$.
- Az összes lehetséges permutációk száma: $C_{max} = H[0] \cdot n$. Például $n = 5$ esetén $Cmax = 120$.
- Az aktuális permutáció sorszáma C_{akt}, melynek értéke kezdetben 0, az aktuális permutáció pedig maga a H tömb.

A változók inicializálása, a tömbök erőforráskímélő feltöltése mind implementációs részletkérdés, melyet jelen pillanatban nem érzek szükségesnek részletezni. A soron következő permutációt a 2.2. pszeudokódban leírt algoritmus szerint állítjuk elő, és ezután a P tömbben találjuk meg.

kód 2.2. Következő permutáció előállítása

```
1   ELJARAS GenerateNextPerm()
2       VALTOZOK
3           i: EGESZ
4       ALGORITMUS
5           HA Cakt = Cmax AKKOR
6               KI: 'No more permutation';
7           KULONBEN
8               HA Cakt <> 0 Akkor
9                   i <- IndexOfFirstDivisorOutOfH();
10                  ReorderPFromI(i);
11              H_VEGE;
12              Cakt <- Cakt + 1;
13              KI: P;
14          H_VEGE;
15  E_VEGE;
```

Az algoritmus belsejében hívott két segédmetódus megvalósítása jelen esetben szintén érdektelen (igény esetén a forráskódban megtekinthető egy implementáció), inkább jelentésük a fontos.

IndexOfFirstDivisorOutOfH. Visszaadja a H tömb első olyan elemének indexét, mely osztója C_{akt}-nak. Ezt az értéket tároljuk i-ben. Például $n = 5$ esetén $C_{akt} = = 12$ érték mellett a metódus visszatérési értéke 1, hiszen H tömb elemei közül a 6 az első, mely 12-nek osztója, és ennek (0 alapú) indexe a tömbben 1.

ReorderPFromI. A P tömb $i, i+1, \ldots n-1$ indexein lévő elemeket újrasorrendezzük aszerint, hogy a jelenlegi $P[i]$ értéktől nagyobb elemek közül a legkisebb lesz az új $P[i]$ érték, az $i+1, \ldots n-1$ indexekre pedig a maradék elemek kerülnek növekvő sorrendben.

Lévén, hogy az algoritmus nem fontos tényezője a jelen projektnek, helyességének matematikai alátámasztását ezen dolgozatban mellőzöm, ehelyett inkább rávilágítanék egy nagy hibájára. Amennyiben a permutálandó kollekció 20-nál több elemű, a H tömb elemei már 8 bájtos előjel nélküli egész típusban sem tárolhatók el. Természetesen Heap algoritmusa esetén sem várja ki senki, hogy az legenerálja a 10^{20+} nagyságrendű permutációt, de az elméleti lehetőség megvan rá, míg az én algoritmusom esetében nincs. Az alkalmazás írásakor ezt a hibát figyelmen kívül hagytam, mert egyrészt képtelenség annyit várni egy permutációs megoldóra, hogy bármilyen, a hibából eredő problémára fény derülhessen a felhasználó számára, másrészt a .Netes projektek alapbeállítások mellett egész típusú változókon végzett műveletek során elnyelik az aritmetikai túlcsordulást, így legrosszabb esetben is a H tömb kerül feltöltésre hamis értékekkel, melyek használatára viszont sosem kerül sor.

2.7.7. Reszponzív mentés

Az alkalmazás rajzolói felületére teljes reszponzivitás jellemző, azaz a rajztér és a rajta ábrázolt gráf is arányosan átméreteződik az ablak átméretezésekor. Az adott gráf csomópontjainak aktuális koordinátáit a rendszer tárolja, mentéskor kiírja őket egy fájlba, megnyitáskor beolvassa őket onnan, módosítás előtt pedig egy verembe helyezi őket lehetővé téve ezáltal a visszavonás műveletet. Ez két esetben jelentette probléma forrását. Ha az adott ablakméreten elmentett gráfot más alblakméret mellett olvasom be, akkor az torzult alakban fog megjelenni az új rajzfelületen, vagy akár ki is lóghat belőle. Ugyanezt a jelenséget eredményezi, ha egy visszavonás művelet kezdeményezése előtt átméretezem az ablakot és vele a rajzteret. A probléma oly módon került megoldásra, hogy a koordinátáknak sosem a pontos értékét tároltam el mentéskor, vagy verembe helyezéskor, hanem a rajzterület szélességéhez és magasságához viszonyított százalékos értéküket.

Ezzel kapcsolatosan pedig már csak egyetlen probléma merül fel, mégpedig az, hogy az MVVM minta alapelvei szerint a ViewModel nem tudhat semmit a View-ról, nem ismerhet arról konkrét adatokat, mint a rajzterület méretét, de ez már csak elvi ellentmondás, melyet a fentiek átfogalmazásával fel is oldhatunk. Amennyiben úgy fogalmazunk, hogy definiálunk maximálisan megadható X és Y koordináta értékeket a csomópontoknak, azt már a ViewModel-nek kötelessége ismerni. Ezen két újonnan definiált property értékét már tetszés szerint köthetjük a View bármely eleméhez, annak bármely tulajdonságához, így akár a rajzterület méreteihez is. Ezen kis kitérővel csak arra szeretnék rávilágítani, hogy igencsak érdekesnek találom, hogy a szoftverfejlesztés világában is nyithatunk kiskapukat a beton-stabilnak látszó szabályokon pusztán a megfogalmazásunk refaktorálásával.

2.7.8. Megoldó törlése

Miképp a felhasználó beolvashat saját megoldó algoritmust tartalmazó szerelvényt, arra is lehetőség nyílik, hogy ezen szerelvényt a rendszer eltárolja, és következő indításkor az algoritmus szerepeljen a választható megoldók listáján. Tekintettel arra az eshetőségre, hogy a felhasználó nem elégedett a beolvasott megoldóval, természetesen lehetőséget biztosítottam annak eltávolítására is. A listából való törlés nem jelentett gondot, az viszont komoly fejtörést okozott, hogy a megoldó fizikai kivetülését, azaz a .dll fájlt nem lehet futásidőben törölni, mert az már szerepel az alkalmazás által referált szerelvények listáján.

A problémára eszközölt megoldásom működik, viszont főként a megkerülő jellege miatt tartom említésre méltónak. A szerelvény az eltávolítás kezdeményezésekor nem kerül törlésre – lévén, hogy ez nem kivitelezhető – , hanem kap egy Dirty flag-et, a program többi modulja pedig az ekképpen megbélyegzett fájlokat úgy kezeli, mintha

nem is léteznének. A megjelölt szerelvények nevét a program konfigurációs fájlja is tartalmazza, így következő induláskor azok fizikálisan is törlésre kerülhetnek, mielőtt az alkalmazás referenciát kapna rájuk.

2.7.9. Settings

Biztosítottam, hogy az alkalmazás arculatát és bizonyos funkcionalitásait a felhasználó a saját szájíze szerint alakíthassa. A személyre szabható beállításokat tételesen nem sorol fel ezen szakaszban, azokról részletesebb leírást a az A függelék alatti felhasználói dokumentációban olvashatunk, vagy kipróbálhatjuk őket az alkalmazás Settings menüpontja alatt. A beállítások aktuális értékei a program konfigurációs fájljában kapnak helyet, olvasásukat és írásukat a SettingsManager osztály biztosítja.

2.8. Saját control-ok

Az alkalmazás fejlesztése közben kettő saját WindowsForms-os control-t – azaz felületi elemet – készítettem, melyek többszörösen felhasználásra kerültek az alkalmazás különböző részein, ezeket ismertetném a következő pontban.

2.8.1. GraphDisplayer

A gráfok grafikus létrehozását és szerkesztését megkönnyítő, a vizuális megjelenítés szívét és lelkét képező GraphDisplayer elem arculata a 2.5. ábrán látható. A control három fő egységre tagolható:

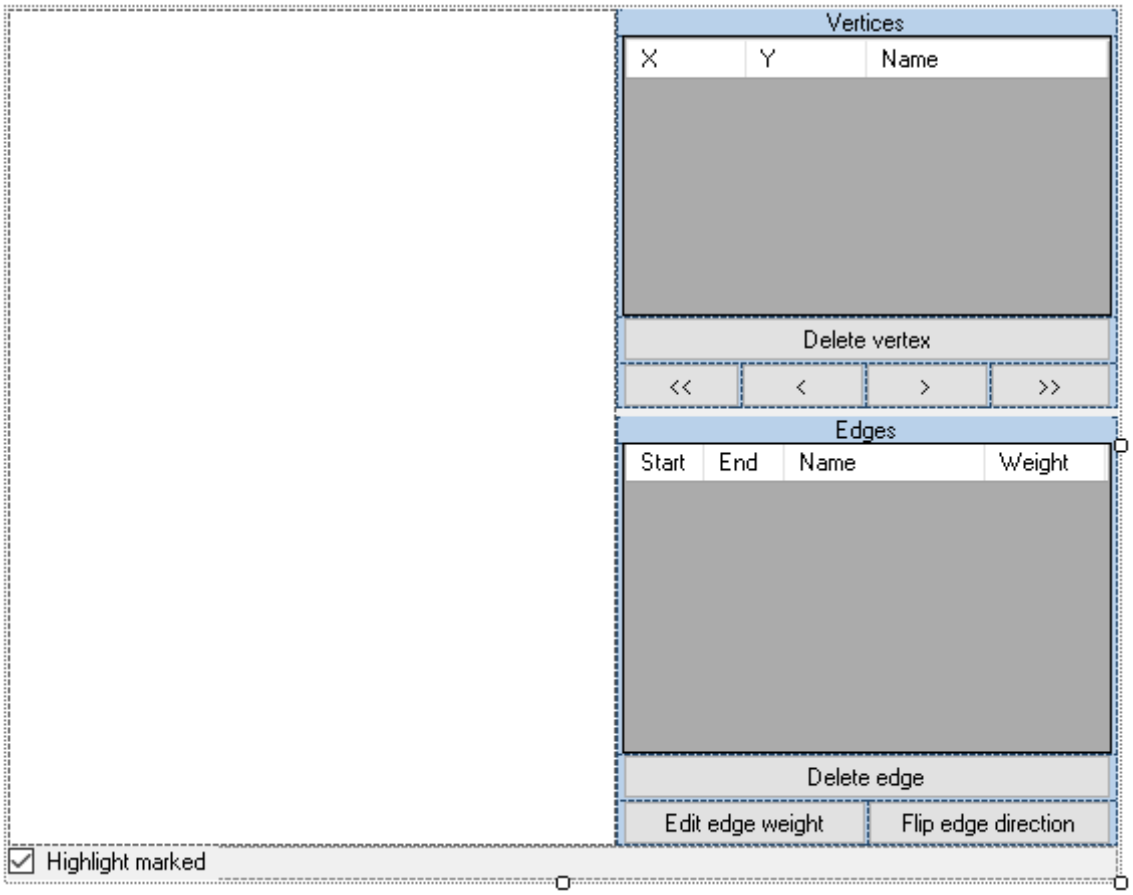
1. Az üres fehér rész a rajzterület, mely a gráfok vizuális megjelenítését teszi lehetővé.

2. A bal oldali táblázatok tartalmazzák az ábrázolt gráf csomópontjainak és éleinek részletes adatait, a hozzátartozó gombok pedig az ezeken végzett műveleteket kezdeményezik.

3. A control alján található blokk egyetlen jelölőnégyzetet tartalmaz, mellyel a gráf jelölt részeinek speciális módú ábrázolása kapcsolható ki, illetve be.

A 2. és 3. blokk mérete állandó, míg a rajzterület mérete automatikusan változik a control méreteit követve. Ezen két blokk láthatósága ki-bekapcsolható, ezzel is növelve a control felhasználási lehetőségeinek számát. Számítási eredmény megjelenítése esetében például nincsen semmi szükség arra, hogy a gráfot táblázatos formában is lássuk, vagy bármilyen műveletet kezdeményezzünk azon (ezért a 2. blokk megjelenítésére nincs szükség), viszont fontos, hogy adott legyen a lehetőség az eredmény kiemelésére, például a Hamilton-kört alkotó élek más színű megrajzolására, illetve ezen funkció

2.5. ábra. A GraphDisplayer control kinézete



ki-bekapcsolására (melyet a 3. blokk biztosít). Rajzoláskor pont fordított a helyzet. Szeretnénk látni a táblázatot, és adott objektumon műveletet végezni (2. blokk), a kiemelést pedig nem szeretnénk ki-bekapcsolgatni, így ezen alkalmazásmódban a 3. blokk megjelenítése felesleges.

A rajzterület és a táblázatok tökéletes összhangban vannak, az egyiknek a módosítása azonnal megjelenik a másikon is. A rajzterületen új pozícióba mozgatott csomópont koordináta-változása például a táblázatban is új formában jelenik meg, illetve fordítva. Az ábrázolt gráf mindig a táblázatban foglaltaknak megfelelő, viszont a táblázat tartalma nem módosítható felhasználó által. A táblázatok a control publikus property-jei, így tartalmuk programatikusan szerkeszthető, sőt köthető adott esetben ViewModel propertyk-hez. A jelen alkalmazás is ilyen formán alkalmazza őket.

A control-ban az is állítható, hogy a rajzterület interaktív-e, vagy sem. Amennyiben egy számítás (például legrövidebb út) eredményét szeretnénk ábrázolni, semmi szükség arra, hogy a rajzterület reagáljon bármilyen kattintásra, míg gráf rajzolása közben ez a legfontosabb elvárásunk a felülettől. Az interaktivitás kapcsán megemlítendő, hogy a gráfrajzolás körforgás szerű mechanizmus. A rajztéren való megfelelő kattintás nem hoz létre új csomópontot, vagy élet, hanem a megfelelő adatok paraméterbe való becsomagolásával elsüt egy eseményt arról, hogy igény van egy új csomópontra, vagy élre.

Erre a gráfot mint objektumot nyilvántartó és kezelő osztály feliratkozik, és hatására hozzáad egy új élet, vagy csomópontot a gráfhoz. A gráf él- és csomópont-kollekciói kötve vannak a control táblázataihoz, melyekben az új (vagy módosított) érték megjelenik, és a control belső szinkronja eredményezi, hogy a felületen is láthatóvá válik az új objektum. A controlon elvégezhető további minden műveletre is igaz ez a körkörös működési elv.

Három féle kiemelésmódra van lehetőségünk a control adottságai alapján:

- Nincs kiemelés. Ez esetben a 3 blokk jelölőnégyzetének állapota semmilyen hatással nincs a gráf megjelenésére.

- Kijelölés alapú kiemelés. A jelölőnégyzet bepipálásával azt érjük el, hogy más színben és vastagságban jelenik meg az aktuálisan kiválasztott él, vagy csomópont, melyen műveletet szeretnénk végezni. Gráf rajzolásakor ez a preferált mód.

- Eredmény alapú kiemelés. Kipipált jelölőnégyzet esetén azon objektumok jelennek meg más színnel és vastagsággal, melyeknek egy megoldás szempontjából kifejező erejük van, például a Hamilton-körben foglalt élek. Objektum alatt kell értenünk jelen esetben többek között a csómópontokat, éleket, irányítottságot jelző nyilakat, IsMarked és ResultValue értékeket[4].

A GraphDisplayer egy további érdekessége, hogy használható átméretező üzemmódban is, mely azt jelenti, hogy az éppen aktuálisan ábrázolt gráfot mindig arányosan átskálázva ábrázolja úgy, hogy az pont kitöltse a rajztér méreteit.

2.8.2. ResultDisplayer

Az alkalmazás egy másik fontos control-ja a ResultDisplayer, mely az automatikus megoldótesztek eredményeinek megjelenítéséért felelős. Szükséges tudnunk, hogy a teszteredményeket a tesztelőrendszer egy SolverTestResult DTO formájában állítja elő, a ResultDisplayer control-nak pedig van egy ilyen típusú property-je. Amint a property új értéket kap, az abban foglalt teszteredmények automatikusan megjelenítésre kerülnek a control felületén.

A 2.3 ábrán egy ilyen felületi elemet láthatunk működés közben. A rendelkezésre álló terület vízszintesen tesztesetekre, függőlegesen pedig alesetekre van tagolva, így az ábrán látható halvány szürke háló minden cellája egy-egy alesetet jelöl. Az egeret valamely cella fölé mozgatva, az adott aleset jellemző be- és kimeneti értékei jelennek meg a grafikus területtől jobbra található mezőkben.

[4] A gráfdefiníciós interfészekről, és ezeket implementáló domain-objektumokról bár illene, nem teszek említést a dolgozatban, így itt jegyezném meg, hogy az IsMarked és ResultValue mind a csomópontokat, mind az éleket jellemző tulajdonságok, melyek értékének beállítása a felhasználó által megírt megoldó algoritmusok legfőbb feladata.

Mint látható, a control a teszteredményeket színkódok – melyek magyarázata a control bal oldalán látható – formájában ábrázolja, és ez a vizuális megjelenítés nagy segítséget nyújt az eredmények kiértékelésében, és több eredmény összevetésében. A 11 lépcsős színskálában látható színek mind a control publikus property-jei, és az alkalmazás Settings menüjében lehetősége is nyílik a felhasználónak ezek módosítására, és egy kedveltebb színskála létrehozására.

2.8.3. Pattern-ellenesség

A GraphDisplayer control változatos lehetőségeket biztosító összetett működési mechanizmusa nagyon jól felvázolható egy állapotátmeneti gráffal, tervezése folyamán jómagam is ekképp jártam el. Ebből kiindulva adta magát a lehetőség, hogy a későbbi esetleges funkcionalitás-változások implementálását elősegítendő a háttérkódot state tervezési minta szerint írjam meg, és el is készült egy ilyen változat, azonban minden előnye ellenére nekem nem nyerte el a tetszésem, így visszatértem az elágazások sorát felvonultató és egymásba ágyazó megvalósításhoz.

Fontosnak tartom leírni döntésem okát, ugyanis nem első alkalom volt ez, hogy ezen minta alkalmazása kétségeket ébresztett bennem. Ellenszenvem egyik oka, hogy rengeteg adatot kell az állapotok között átadogatni. Természetesen ezek becsomagolhatók egy kontextus objektumba, de ezen becsomagolt adatok mindegyike publikus kell, hogy legyen, holott magán a sokállapotú objektumon kívül ezek senki másra nem tartoznak. Hasonlóan rossz érzést kelt bennem, hogy a szóban forgó objektumunk állapota mint tulajdonság szintén kívülről elérhető és módosítható bárki által.

Lehet, hogy ezen szakasz felesleges kitérőnek tűnik a dolgozatot olvasó szemében, számomra viszont fontos volt rávilágítani annak az okára, hogy a program talán legfontosabb felületi elemének miért úgy néz ki a háttérkódja, ahogyan kinéz.

3. fejezet

Saját Hamilton-kör megoldó

Nagy örömömre szolgálhatott volna, ha a dolgozatnak ez lenne a fő fejezete, és itt fejthetném ki az elért áttöréseim a Hamilton-kör probléma valós idejű megoldását illetően, de sajnos nem így alakult, és a dolgozat legrövidebb fejezetét olvashatjuk. A kiötlött módszereim ugyan villámgyorsnak bizonyultak, hatékonyságuk messze elmaradt a reméltől. A legjobb algoritmusom is körülbelül 20 csomópontos gráfokig tudott megoldást szolgáltatni, 20 és 30 csomópont között teljesen rapszodikus teszteredményeket produkált, ennél nagyobb gráfok esetén pedig már jóformán sosem működött.

A tesztelt algoritmusok részleteit, és elméleti hátterét jelen keretek között nem tartom említésre méltónak, viszont az eredményeim semmiképpen sem érzem kudarcnak, mert bár a ma használt megoldók performanciájának töredékét sem sikerült biztosítanom, a pillanatok alatti biztos számítás mellett elért 20 csomópontos határ, is közel duplája annak, mint amit az egzakt matematikára építkező permutációs megoldók egy emberélet alatt számítanak ki.

A projekt legfőbb célját ugyan nem sikerült elérnem, a ráfordított energia és idő töredékét sem sajnálom, hisz a megalkotott alkalmazás által már lehetőségem nyílik további ötleteim egyszerű kipróbálására, és esetlegesen társaim részére is tudom ezen lehetőséget biztosítani. Az eddigieknél hatékonyabb algoritmus kidolgozásának ötletét természetesen nem vetettem el, és tovább próbálkozom.

4. fejezet

Továbbfejlesztési lehetőségek

Az alkalmazás későbbi verzióira tekintve a következő szakaszokban foglalnám össze azon lehetőségeket, melyek megvalósítása által a program nagyobb értéket képviselhet. A pontok sorrendje semmilyen prioritást nem képvisel.

4.1. Nyelvesítés

Jelenleg a program felületi feliratai és hibaüzenetei egyaránt angol nyelven jelennek meg. Mindenképp pozitív hatást keltene, ha ez módosítható lenne. Mindehhez az egyik legfőbb feltétel adott a jelenlegi kialakításban, mivel az összes, felhasználó számára látható szöveg resource fájlból kerül megjelenítésre. Az, hogy melyik ilyen fájl tartalma legyen a kiválasztott, konfigurációs beállításban tárolható, egyetlen kódsor által kiolvasható az alkalmazás indításakor, viszont ehhez szükséges további resource fájlok létrehozása, melyhez maga a fordítás, valamint a kapott eredmények tüzetes átvizsgálása jelent idő- és energiaigényes folyamatot. Megfelelő mennyiségű szabadidő esetén természetesen hasznosnak ítélem további nyelvek hozzáadását az alkalmazáshoz.

4.2. További gráfproblémák

Szintén a program értékét növelő és az egyik legfontosabb távoli célkitűzésem, hogy további gráfelméleti problémákra (például hozzárendelési-, vagy izomorfia problémára) implementáljak beépített megoldókat, és automatikus tesztelési lehetőséget egyéni fejlesztésű algoritmusok számára. Ezen cél eléréséhez mindösszesen elegendő mennyiségű szabadidőre van szükség, az alkalmazás jelenlegi architektúrája minden további nélkül biztosítja az ez irányú továbbfejlesztés lehetőségét.

4.3. Performanciaváltozási problémák megoldása

A 2.6.8. szakasz végén említetteknek megfelelően az alkalmazásban nincsen még megoldás azon esetlegesen felmerülő problémára, hogy a programindításkor mért SATU értéke az idő folyamán változhat, miképp a felhasználó egyéb erőforrásigényes folyamatokat indít, vagy állít meg. Több megoldási lehetőség is fennáll, ezek közül néhány:

- Figyelmeztetjük a felhasználót a további alkalmazások indításának esetleges kockázataira.
- A SATU értékét minden megoldóteszt folyamata előtt (esetleg közben is többször) újramérjük.
- Figyeljük az operációs rendszer felől elérhető adatokat mind a processzek számát, mind az erőforrások kihasználtságát illetően, és szükség esetén mérjük újra a SATU-t, valamint lépünk interakcióba a felhasználóval.
- Külön alkalmazásmodul készítése, mely konfigurációs beállításoknak megfelelően lefoglal adott méretű erőforrást a program számára, melyet az egész futásidő alatt sajátjának tekint, nem ad át más alkalmazásnak, viszont nem is igényel ennél többet. Analógiaként gondolhatunk a virtuális gépek által lefoglalt erőforrásokra.

4.4. Új felület

Az alkalmazás grafikus felületei az 1.1. szakaszban leírtaknak megfelelően a WindowsForms által biztosított ablakok, viszont mindenképp célszerűnek tartom, hogy a későbbiekben kialakítsak WPF-es felületeket is, és összevessem a két technológiát a jelen alkalmazáson keresztül is. Nemcsak a külső változtatása lenne a cél, hanem esetlegesen önmagam megcáfolása is a XAML kódban való fejlesztés bonyodalmait illetően.

További lehetőség a container-megoldások tanulmányozása, feltérképezése, és általuk a már jól működő WindowsForms-os saját control-ok beépítése új WPF-es felületekbe.

4.5. Bőséges logolás

Míg jelenleg az alkalmazás csak a hibákat, és azoknak is csak az üzeneteit írja logfájlba, a jövőbeli verziókra nézve fontos elvárás, hogy a hatékonyabb hibanyomozás érdekében hiba esetén a teljes hívási verem kerüljön kiírásra. A hibák reprodukálhatósága érdekében szintén fontos lenne, hogy bármely command végrehajtása esetén maga az esemény jellege és a körülményeket leíró paraméterezése is váljék a log részévé.

4.6. Multipéldány tiltása, rugalmas fülek kialakítása

Az alkalmazás támogatja a több fülön (tabon) való munkát, ennek értelmében két példány indítása egyrészt felesleges, másrészt fel nem térképezett kockázatokat rejthet. A több példányos használat tesztelése is lehet cél, de sokkal előrelátóbbnak tartom ezen lehetőség tiltását, és helyette rugalmasabb tab-rendszer megvalósítását. A tabok átsorrendezése, esetlegesen új keretbe való kiemelése – két tab tartalmának egyszerre történő megtekintése, azok összehasonlítása céljából – ma már gyakran alapfunkciónak számít, és a felhasználó el is várja ezt a lehetőséget. Ha mindezt a legkevesebb energiabefektetés árán szeretnénk elérni, a megoldás jelen esetben is új WPF-es felület kialakításával kezdődne, hisz ez a technológia nagyobb támogatást nyújt a fülek rugalmas kezelése terén.

4.7. Jóhiszemű tesztelés feladása

Jelenleg a megoldótesztelő mechanizmus az úgynevezett jóhiszeműség elvein működik. Amennyiben a felhasználó megoldó algoritmusa azt állítja, hogy megoldást talált a tesztgráfban az adott problémára, akkor azt elfogadjuk, és a sikert mint adatot rögzítjük. Bár a felhasználó hamis adatokat szolgáltató megoldóval csak önmagát csapja be, mégis célszerű lenne ellenőrizni, hogy a megoldótól eredményként visszakapott gráf és a benne rejlő eredménydefiníciók valóban helyes megoldást jelentenek-e az adott probléma szempontjából, hisz a felhasználó akár tudtán kívül is követhet el olyan implementációs hibát, mely miatt a jónak vélt megoldás mégsem kielégítő.

4.8. Megoldóteszt párhuzamosítása

A megoldók tesztelési idejének rövidítése érdekében fontos továbbfejlesztést jelentene, ha több tesztpéldány, esetleg több aleset azonos időben külön szálakon futhatna. Ehhez mindenképpen szükség lenne a futtató számítógép architektúrájának, erőforrásainak feltérképezésére. Meg kell ismerni a rendelkezésre álló magok, szálak számát, a számítási kapacitásokat, a lefoglalható memóriaterületeket, és ezeket oly módon szétosztani, hogy az egy időben futó tesztelési folyamatok semmiképp sem legyenek hatással egymás eredményeire. Ez komoly optimalizációs folyamatot jelentene, mely megvalósításához jelenleg nem érzem, hogy elegendő ismeret birtokában lennék, így a probléma megoldásához vezető út első lépése mindenképp a technológiák, lehetőségek kutatása, azaz a tanulás.

4.9. Help kiterjesztése

A mostani programverzióban fellelhető Súgó fájlok kizárólag a tesztelési folyamat elméleti hátterét ismertetik, ezzel szemben elvárható fejlesztési igény lehet az alkalmazással szemben, hogy tartalmazza, vagy tegye elérhetővé – akár webes úton – a teljes felhasználói dokumentációt, és annak is a mindenkori aktuális verzióját.

4.10. Kényelmi funkciók

A felhasználói élmény fokozásának jegyében több kényelmi funkcióval is fel lehetne ruházni az alkalmazást, mind a felületek még barátságosabbá tételével, mind további funkcionalitások beépítésével. A sok említhető lehetőség közül jó példa a teszteredmények változatos kezelése, azaz lehetőség nyújtása az egy kattintás általi pillanatképkészítésre, vagy eredmények e-mail általi – akár automatikus – továbbítására. Szintén hasznos és kényelmes lehet egy beépített print funkció, mellyel a rendszer által ismert nyomtatók valamelyikére, vagy pdf-be küldhetnénk az aktuális tartalmakat.

4.11. További beállítások

További rugalmasságot biztosítana a Settings menüben található beállítások körének bővítése. Jelenleg néhány nagyon alapvető paraméter megadására, módosítására van a felhasználónak lehetősége, melyek számát szükséges lenne növelni. Természetesen a ló túloldalára sem tartom célszerűnek átesni, nem támogatom azon elméletet, mely alapján néhány alkalmazásban olyan pilótavizsgás beállítási felületek készülnek, hogy a rajtuk való kiigazodáshoz is külön kézikönyvet lehetne kiadni 2-3 kötetben.

4.12. Automatikus telepítőcsomag-készítés

Ezen fejlesztés inkább az alkalmazás elkészítésének folyamatára irányul, mintsem a program felhasználói szemmel való megközelítésére, de ettől függetlenül fontos lépés lenne. Jelenleg amikor a kód és annak mögöttes tartalma olyan állapotban van, hogy jónak látom egy újabb release tesztelését, akkor egy MSI Setup Project segítségével készítek telepítőcsomagot, melynek a belső szerkezetét a csomag bizonyos kötöttségei miatt harmadik féltől származó külső program által módosítanom kell. Szeretném elérni, hogy ez a folyamat automatikusan megtörténjen minden egyes elfogadott pull request esetén a build folyamat részeként, annak utolsó fázisában.

4.13. WinFormsBinder fejlesztése

Bár nem szerves része a dolgozatnak, ismét említeném a WinFormsBinder csomagot, mely jelenleg a dolgozat keretein belül készített alkalmazás kötési igényeit elégíti ki. A csomag sokkal több kötéstípus biztosítását lenne hivatott ellátni, és ezek megvalósítását szintén jövőbeli feladatomnak tartom akár egy opensource projekt keretein belül is.

4.14. Névválasztás

Egy olyan technikai részletkérdést céloz meg ezen fejlesztési lehetőség, melyet most, a dolgozat írása közben is meg tudok valósítani, és meg is teszem ezt. A lefejlesztett programra ezidáig úgy hivatkoztam, hogy „az alkalmazás”, azonban célszerűbb lenne, ha saját nevet kaphatna, és ennek tükrében az alkalmazást mostantól a következőképp hívják: Asanyad Grapher.

5. fejezet

Összefoglalás

A dolgozat alapproblémája az volt, hogy a jelenleg elérhető alkalmazások és szolgáltatások egyike sem teszi lehetővé, hogy adott gráfelméleti probléma megoldására készült algoritmusok működésének hatékonyságát manuálisan, vagy automatikusan tesztelhessük szoftveres környezetben.

Szintén a terület hiányosságaként említhettük, hogy különböző megoldó algoritmusok hatékonyságának összevetésére sincs lehetőségünk. Természetesen a tár- és időigények alapján történő összehasonlítás adott mint lehetőség, viszont ennél gyakorlatiasabb megoldás elérhetőségét is szükségesnek vélem.

Feltett szándékom volt, hogy hiánypótló megoldást biztosítsak ezen a területen, és lehetővé tegyem ez által a magam és mások számára gráfelméleti algoritmusaink tesztelését.

A dolgozat megvalósítása során olyan alkalmazást készítettem, és elméleti hátteret dolgoztam ki, melyek a fentieket lehetővé teszik.

Az Asanyad Grapher alkalmazás felhasználóbarát felületet nyújt, és sokrétű funkcionalitással bír, mind alkalmazhatóság, mind hatékonyság szempontjából megfelel a kitűzött kritériumoknak.

Elméleti síkon a legfőbb elért eredményként könyvelhetem el, hogy olyan leírási lehetőséget definiáltam gráfokra, mely lehetővé teszi egységes kezelésüket egy adott probléma szempontjából, és elősegíti a megoldó-tesztelés eredményeinek egyszerűen áttekinthető ábrázolását.

Szintén fontosnak tartom, hogy az alkalmazás a futtató számítógép architektúrájától függetlenül képes mérni, hogy milyen sebességgel működnek a megoldó algoritmusok.

Az eredeti célkitűzéseim – egy kivételével – úgy vélem lehető legjobb eredmény mellett teljesítettem. A kivételt képező cél – azaz saját megoldó algoritmus implementálása Hamilton-kör problémára – nem-teljesülését semmiképp sem tartom kudarcnak, hiszen a megvalósított program segítségével nyugodtan próbálkozhatok tovább az ideális megoldás kutatásával egy következő projekt keretein belül.

6. fejezet

Köszönetnyilvánítás

Ezen fejezetben megragadnám a lehetőséget, hogy külön köszönetet mondjak mindenkinek, aki bármilyen formában hozzájárult ahhoz, hogy a dolgozat elkészülhessen. A teljesség igénye nélkül a következő személyeket emelném ki a sokaság soraiból:

Családtagjaim. Köszönöm nekik, hogy elviselték gyakori távollétemet, otthon töltött időmben pedig a dolgozatírás teremtette elviselhetetlen természetemet!

Minden tanárom. Köszönet illeti őket, hisz mindahányan hozzájárultak a dolgozat elkészüléséhez. Név szerint kiemelném Troll Ede Mátyást a konzulensi munkája miatt, és Dr. Kovásznai Gergelyt a Bevezetésben idézett mondatáért.

Kollégáim. Köszönöm mindannyijuknak – kiemelten Sosterics Ádámnak – azt az elméleti és gyakorlati tudásanyagot, mely szintén nagyban hozzájárult az elkészített alkalmazás kialakításához!

Irodalomjegyzék

[1] Thomas H. Cormen, Charles E.Leiserson, Ronald L. Rivest: *Algoritmusok*, Műszaki Könyvkiadó, Budapest, 1999.

[2] Arnaud Weil: *Learn WPF MVVM – XAML, C# and the MVVM pattern: Be ready for coding away next week using WPF and MVVM*, Leanpub, Victoria, British Columbia, Canada, 2016.

[3] Douglas B. West: *Introduction to Graph Theory*, Pearson Education, Singapore, 2002.

[4] Flinders University: *Flinders Hamiltonian Cycle Project honlapja*, Link: http://www.flinders.edu.au/science_engineering/csem/research/programs/flinders-hamiltonian-cycle-project/

[5] *A Cake Build rendszer honlapja*, Link: https://cakebuild.net//

[6] Dancsó Sándor: *A Grapher alkalmazás és függőségeinek elérése*, Link: http://grapher.asanyad.xyz

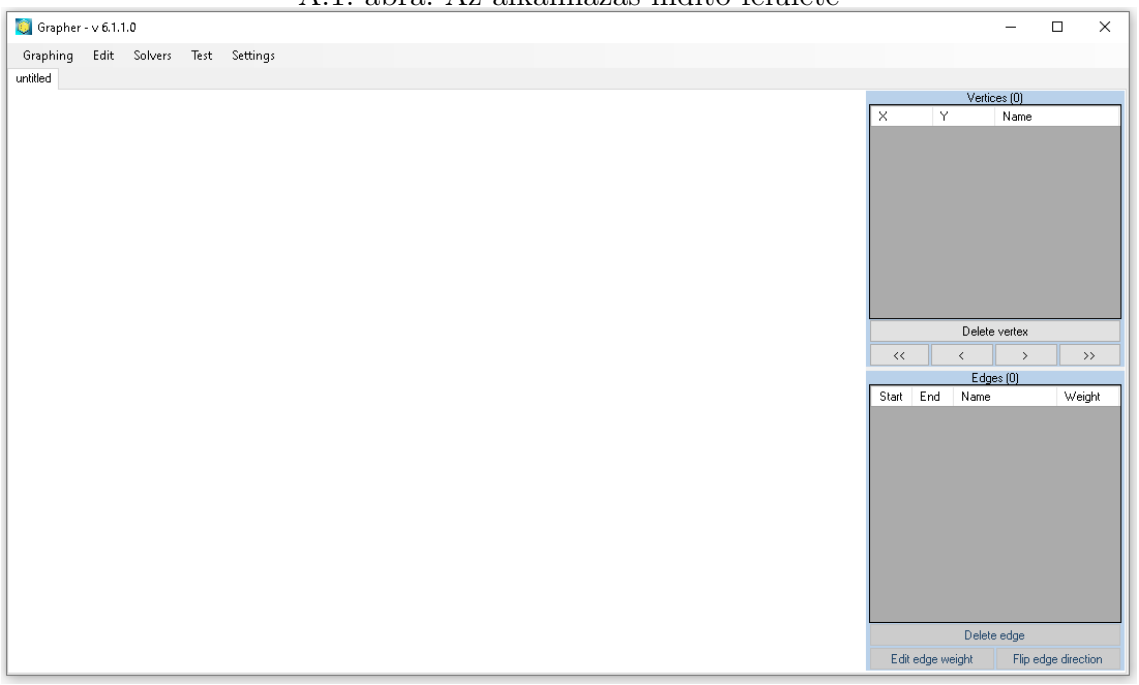
A függelék

Grapher – Felhasználói útmutató

A.1. Indítás, bezárás

A programot célszerű rendszergazdaként indítanunk, ellenkező esetben adott funkciók nem működnek megfelelően. Indítást követően a kiinduló felület (A.1. ábra) fogad minket.

A.1. ábra. Az alkalmazás indító felülete



A programot a „Ctrl + X” megnyomásával, vagy a „Graphing/Close Grapher” menüponttal zárhatjuk be. Ekkor a program megerősítést kér a szándékunkat illetően, figyelmeztet is, ha vannak mentetlen gráfjaink.

A.2. Fülek

Az indító képernyőn egy gráfszerkesztő felület található. Látható, hogy a szerkesztő egy fülön (tabon) helyezkedik el, ebből a fülből tetszőleges mennyiségű példányt nyithatunk meg, és rajtuk egymástól függetlenül rajzolhatunk gráfokat.

Új fület a „Graphing/New graph” menüpont által, a „Ctrl + N” billentyű-kombinációval, vagy a meglévő fülektől jobbra lévő területen duplán kattintva nyithatunk, az aktuális fület pedig a „Graphing/Close tab”, vagy a „Ctrl + W” zárja be. Fül bezárásakor, amennyiben van el-nem-mentett módosításunk, figyelmeztető üzenetet kapunk.

A.3. Szerkesztés

Csomópont hozzáadása. A rajzterület üres részén bal egérgombbal kattintva hozhatunk létre új csomópontot.

Él hozzáadása. Egy meglévő csomóponton a jobb egérgombot lenyomva, az egeret másik csomópontra mozgatva, és a gombot azon felengedve a két csomópont között egy élet hozhatunk létre. Az élen lévő nyíl az irányítottságot, az élre írt szám (alapesetben 1) pedig az él súlyát jelenti. Ha a kiinduló- és érkező csomópont azonos – vagyis jobb egérgombbal kattintunk egy csomóponton –, akkor önélet hozunk létre. Az önélen nincs nyíl, hisz nincs értelme irányítottságról beszélni.

Csomópont mozgatása. A rajzterületen lévő csomópontot a bal egérgomb lenyomva tartásával foghatjuk meg, és mozgathatjuk át a rajztér egy másik pontjára. A csomóponttal együtt mozognak a kapcsolódó élek is.

Objektum adatok. A csomópontokat és éleket jellemző adatokat a rajztér melletti „Vertices” illetve „Edges” táblázatokban láthatjuk. A táblázatok tartalma minden esetben összhangban van az általunk rajzolt gráffal.

Objektum kijelölés. További műveletek elvégzése érdekében szükségünk lehet egy csomópont, vagy egy él kijelölésére. Ehhez vagy a táblázatban kattintsunk az objektum sorára, vagy a rajzterületen magára az objektumra. A kijelölt objektum mind a táblázatban, mind a rajztéren eltérő módon kerül megjelenítésre, és a hozzá tartozó – táblázat alatti – akciógombok aktívak.

Objektum törlése. A kijelölt objektum (csomópont, vagy él) törlésére 3 lehetőség van:

- „Del” billentyű
- „Edit/Delete selection” menüpont
- „Delete vertex” és „Delete edge” gombok a táblázatok alatt

Csomópont törlése a hozzákapcsolódó élet eltávolítását is magával vonja.

Élfordítás. A kijelölt él irányítottságát a „Flip edge direction” gomb, a „Ctrl + SPACE” billentyűkombináció, vagy az „Edit/Change edge direction” menüpont által fordíthatjuk az ellenkezőjére.

Súlyozás. A kijelölt él alapértelmezetten 1 értékű súlyát az „Edit edge weight” gomb, az „Alt + SPACE” billentyűkombináció, vagy az „Edit/Edit edge weight” menüpont által módosíthatjuk. A felugró ablakban írjuk be az új – pozitív egész – értéket és az „OK” gomb megnyomásával nyugtázzuk.

Csomópont sorrend. Lehetőségünk van a csomópontok hozzáadási sorrendjét módosítani. Ennek a későbbiekben lesz fontos szerepe, mikor problémát szeretnénk megoldani a rajzolt gráfunkon. A kijelölt csomópont mozgatását a következő módokon végezhetjük el:

Mozgatás eggyel előrébb. „Edit/Move vertex up”, „Ctrl + ↑”, „<” gomb.

Mozgatás eggyel hátrébb. „Edit/Move vertex down”, „Ctrl + ↓”, „>” gomb.

Mozgatás legelőre. „Edit/Make vertex first”, „Ctrl + ←”, „«” gomb.

Mozgatás leghátra. „Edit/Make vertex last”, „Ctrl + →”, „»” gomb.

Visszavonás. A legutóbbi mentésünk óta tett bármely szerkesztési lépésünket visszavonhatjuk az „Edit/Undo”, vagy a „Ctrl + Z” segítségével.

A Grapher-t az A.2. ábrán tekinthetjük meg szerkesztő üzemmódban való alkalmazás közben.

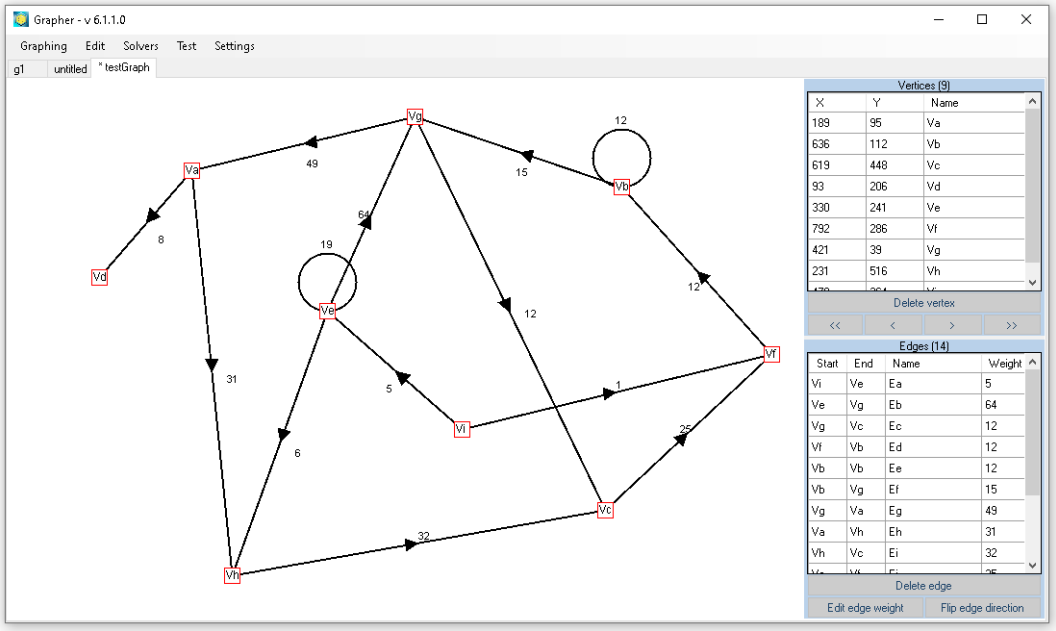
A.4. Mentés beolvasás

A rajzolt gráfot a „Ctrl + S” billentyű-kombinációval, vagy a „Graphing/Save graph” menüpont által menthetjük el. Amennyiben első alkalommal mentjük a gráfot, ki kell választanunk, hogy hová és milyen néven kerüljön a fájl. A mentést tartalmazó fájl .gdf kiterjesztésű.

A mentés más néven funkció eléréséhez a „Graphing/Save graph as”, vagy a „Ctrl + A” használandó.

Egy mentett .gdf fájlt a „Graphing/Open graph”, vagy a „Ctrl + O” által nyithatunk újra meg, és egy új fülön fog megjelenni.

A.2. ábra. Grapher – szerkesztés közben



A.5. Megoldók használata

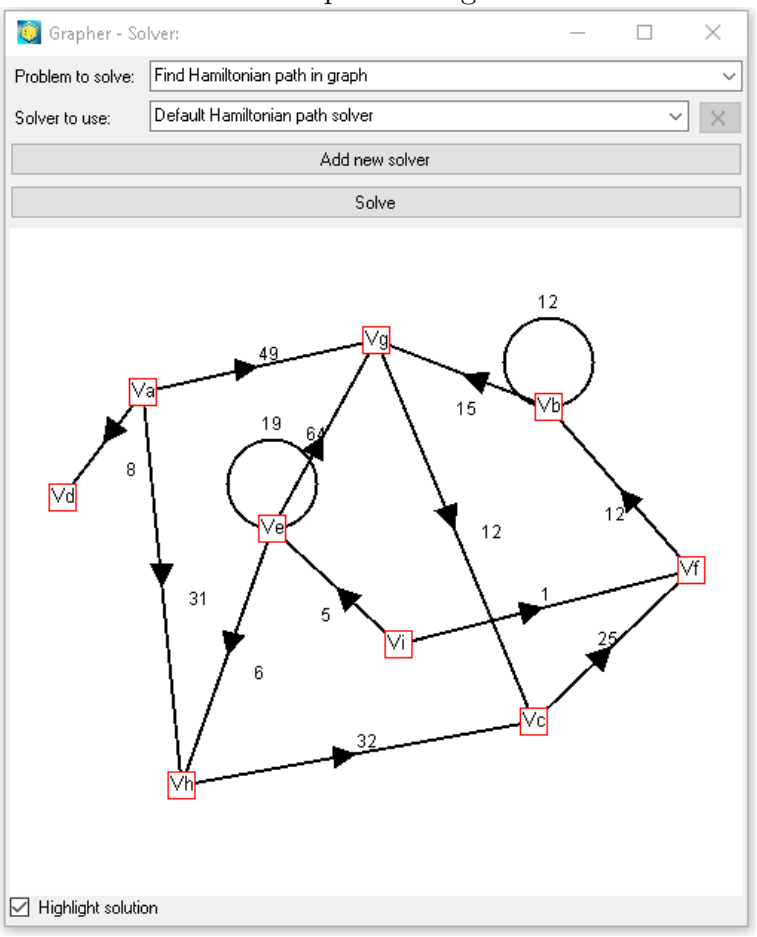
A szerkesztő felületen megrajzolt gráfunkon módunkban áll különböző számításokat végezni. A támogatott problémák, és a megoldásuk kezdeményezése a következőképp alakul:

- Irányítatlan Hamilton-út – „Ctrl + F1”, vagy „Solvers/Hamiltonian path”
- Irányított Hamilton-út – „Ctrl + F2”, vagy „Solvers/Hamiltonian path - Directed”
- Irányítatlan Hamilton-kör – „Ctrl + F3”, vagy „Solvers/Hamiltonian cycle”
- Irányított Hamilton-kör – „Ctrl + F4”, vagy „Solvers/Hamiltonian cycle - Directed”
- Irányítatlan Euler-út – „Ctrl + F5”, vagy „Solvers/Eulerian path”
- Irányított Euler-út – „Ctrl + F6”, vagy „Solvers/Eulerian path - Directed”
- Irányítatlan Euler-kör – „Ctrl + F7”, vagy „Solvers/Eulerian cycle”
- Irányított Euler-kör – „Ctrl + F8”, vagy „Solvers/Eulerian cycle - Directed”
- Legnagyobb átfolyás – „Ctrl + F9”, vagy „Solvers/Max Flow in flow network”
- Legolcsóbb feszítőfa – „Ctrl + F10”, vagy „Solvers/Minimum spanning tree”
- Legrövidebb út – „Ctrl + F11”, vagy „Solvers/Shortest path”

Bármelyiket is választjuk, megnyílik az A.3. ábrán látható megoldó felület, és a „Problem to solve” legördülő listában kiválasztásra kerül a megoldani kívánt probléma.

A „Ctrl + F12” billentyű-kombináció hatására, vagy „Solvers/Any” menüponton át is erre a felületre jutunk, viszont nem kerül kiválasztásra a megoldandó probléma.

A.3. ábra. Grapher – megoldó felület



A „Problem to solve” legördülő listából bármikor új problémát választhatunk. A „Solver to use” legördülő lista mindig az aktuálisan kiválasztott problémára irányuló, a rendszer által ismert megoldókat tartalmazza.

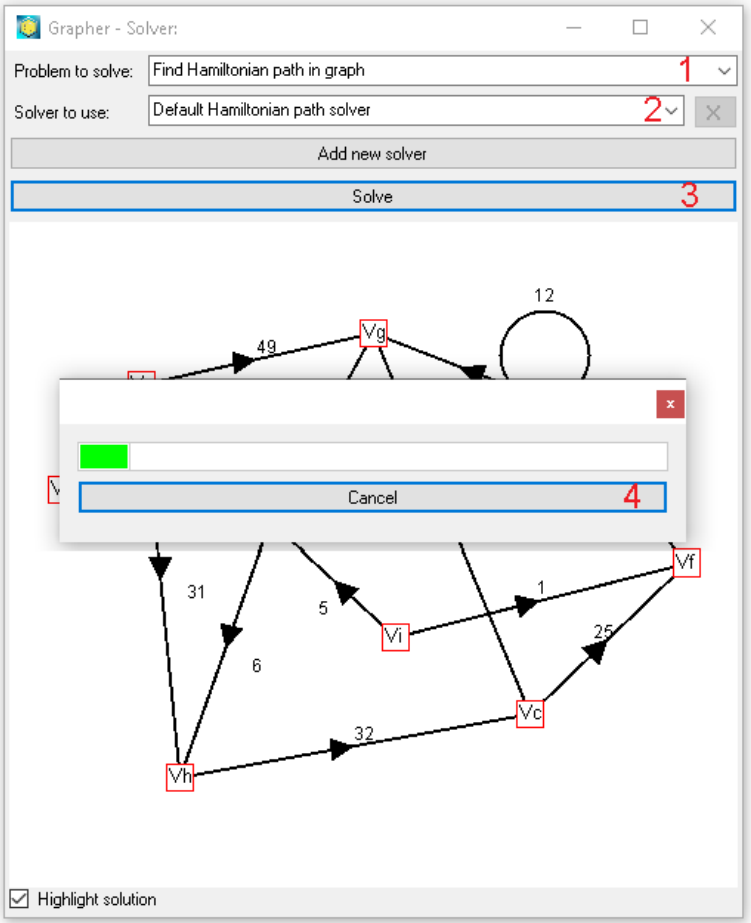
Ha problémát és megoldót is választottunk, a „Solve” gomb megnyomásával kezdeményezhetjük a megoldást, melynek aktuális állapotát a folyamatjelző felület mutatja. A megoldást a folyamatjelző „Cancel” gombjával állíthatjuk le igény esetén. Ezen megoldási és megállítási folyamatot szemlélteti az A.4. ábra.

Amennyiben hagyjuk a megoldót dolgozni, a feladat végeztével kedvező, vagy kedvezőtlen üzenettel jelzi a számítás végét (A.5 .ábra).

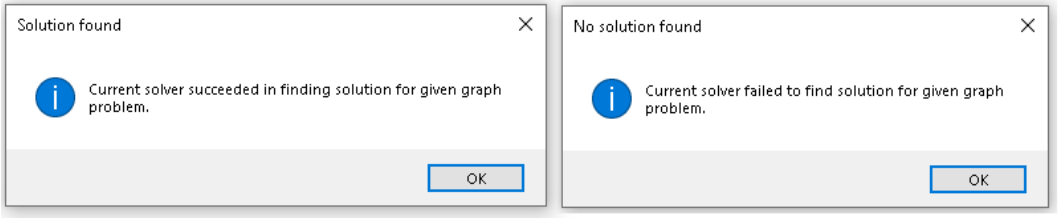
Sikeres folyamatot követően a megoldás megjelenik a megoldófelületen. Ennek a kiemelő stílusú nézetnek a ki- és bekapcsolását érhetjük el a felület alján található „Highlight solution” jelölőnégyzet állapotának változtatásával (A.6. ábra).

Amennyiben olyan megoldót szeretnénk használni, amely még nem része a program-

A.4. ábra. Grapher – megoldás menete



A.5. ábra. Grapher – megoldás kimenetele



nak, az „Add new solver” gomb megnyomására megnyíló fájlkereső dialógusablakban kitallózhatjuk a megoldót tartalmazó .dll fájlt, és a megoldást kezdeményezhetjük azzal is. A beolvasott megoldó ettől kezdve a program része lesz, és a továbbiakban mindig rendelkezésre áll.

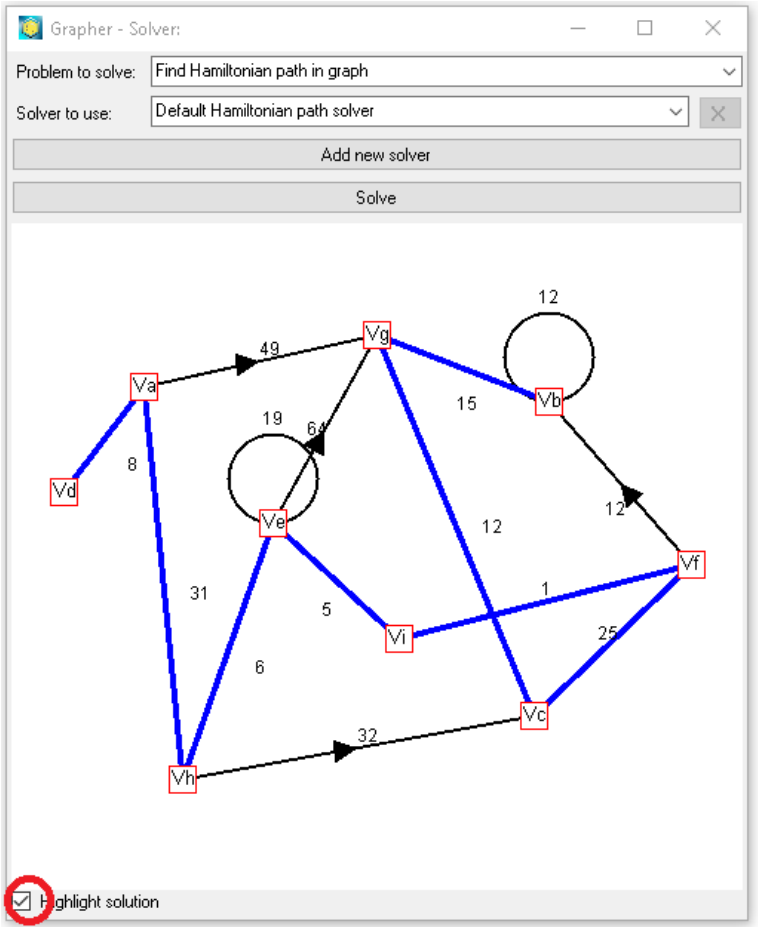
Hozzáadott megoldó törlését – amennyiben feleslegessé vált valamilyen okból – a neve melletti X gombra kattintva kezdeményezhetjük (A.7. ábra). Beépített megoldók esetén a gomb inaktív, azok nem törölhetők.

A.1. *Megjegyzés.* Saját megoldó algoritmus készítése programozói ismereteket igényel, menetének leírását a Grapher fejlesztői útmutatója tartalmazza.

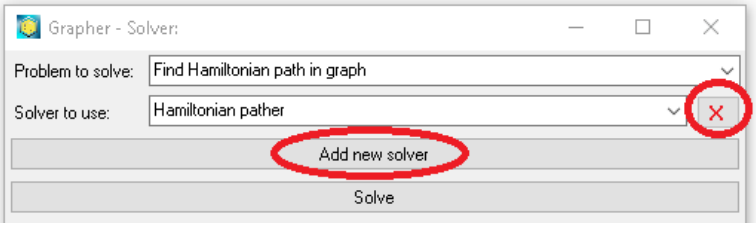
A.2. *Megjegyzés.* Fontos tudni, hogy amennyiben legnagyobb átfolyás, vagy legrövidebb út problémájára szeretnénk megoldást keresni, ezen problémák esetében mindig

van két kiemelt szerepű csomópont: Kiinduló/érkező, forrás/nyelő. A megoldók a gráfszerkesztésnél megadott első és utolsó csomópontokat tekintik ezen kiemelt csomópontoknak. Ezen okból fontos, hogy szerkesztéskor a csomópontok sorrendje módosítható az A.3. szakaszban leírtak szerint.

A.6. ábra. Grapher – megoldás kimenetele



A.7. ábra. Grapher – megoldó hozzáadása és törlése



A.6. Megoldók tesztelése

A Grapher-ben lehetőségünk nyílik gráfelméleti algoritmusok automatikus tesztelésére, melyet a következő módokon kezdeményezhetünk:

– A program által ismert megoldó tesztelése: „Test/Test solver included in library” menüpont, vagy „Alt + F1”.

– Új, saját megoldó tesztelése: „Test/Test new solver” menüpont, vagy „Alt + F2”.

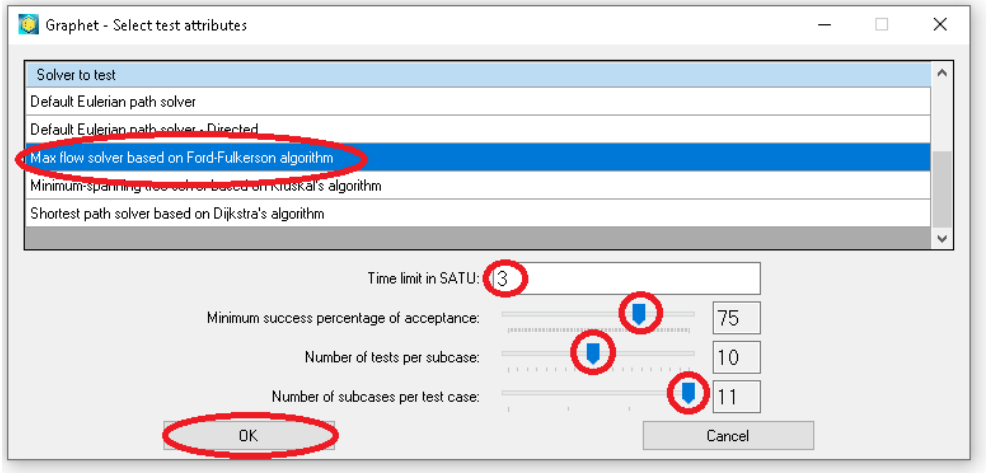
– Új, saját megoldó tesztelése, és hozzáadása a programhoz: „Test/Test new solverr and add it to library” menüpont, vagy „Alt + F3”.

Bármely opciót választjuk, az A.8 .ábrán látható felület nyílik meg, viszont az utóbbi két lehetőség esetén ezt megelőzi egy fájlkiválasztó dialógusablak, melyben a tesztelni kívánt algoritmust tartalmazó .dll fájlt kell kitallóznunk.

A felület tetején lévő listából tudjuk kiválasztani, hogy a program által ismert – vagy a kiválasztott szerelvényben található – megoldók közül melyikre szeretnénk automatikus tesztet futtatni, alatta pedig a tesztelési paramétereket állíthatjuk be. A teszt az „OK” gomb megnyomásával indul.

A.3. *Megjegyzés.* A tesztelési paraméterek és a tesztelés menetének megértéséhez javasolt a Grapher Súgójának elolvasása, melyet a „Test/Help” menü almenüpontjaiban érhetünk el.

A.8. ábra. Grapher – automatikus megoldóteszt paramétereinek megadása

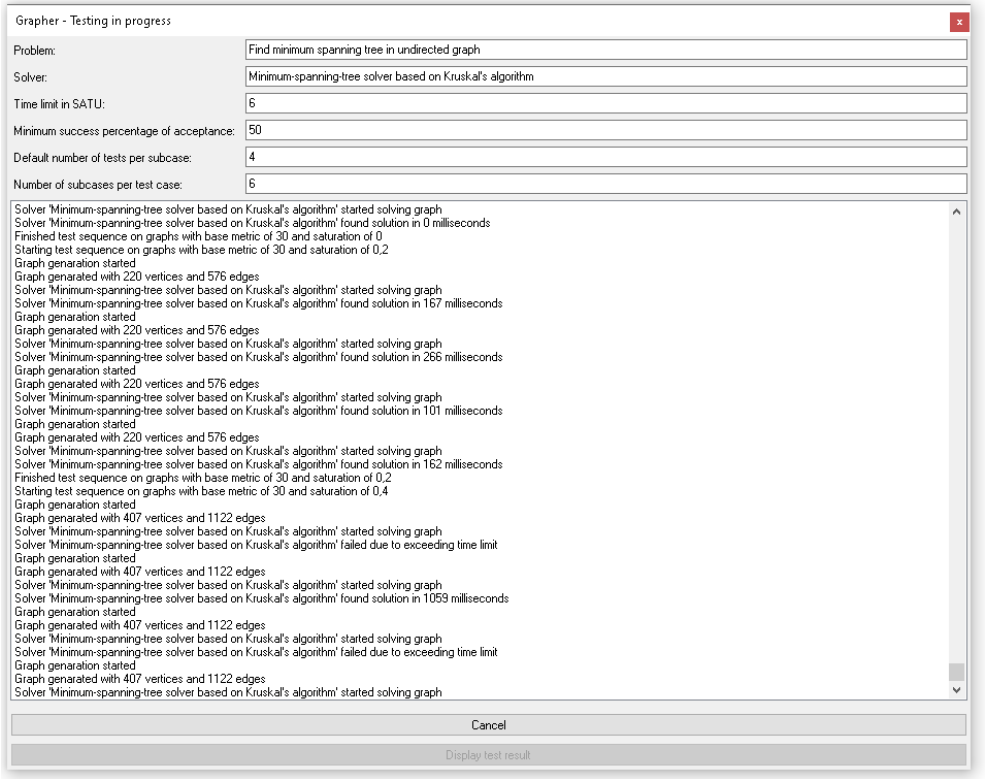


A teszt futása közben az A.9 .ábrán látható ablak ad folyamatos tájékoztatást. Az ablak felső részén a teszt jellemző adatai láthatók, alatta pedig az aktuálisan végzett tesztműveletet jeleníti meg a program.

A teszt bármikor megszakítható a „Cancel” gomb megnyomásával. A teszt befejeztét a „Display test result” gomb aktiválódása jelzi, melyet megnyomva a teszteredmények vizuális formában kerülnek megjelenítésre az A.10 .ábrán látható módon.

A felület halvány szürke rácsozásának minden cellája egy-egy tesztelési alesetnek felel meg, melynek adatai a jobb oldali mezőkben válnak láthatóvá, ha az egérmutatót a cella fölé mozgatjuk.

A.9. ábra. Grapher – automatikus megoldóteszt futás közben



A teszteredményt – annak minden adatával együtt – módunkban áll lokálisan elmenteni, melyet a „Test/Export test result” menüpont, vagy az „Alt + S” billentyűk által kezdeményezhetünk. A mentett .str fájlok „Test/Import test result”, vagy az „Alt + O” által olvashatók be.

A teszteredmények külön fülön találhatók, így egyszerre több is megnyitható belőlük. Az aktuális fület azonos módon zárhatjuk be, mint a szerkesztő fület („Graphing/Close tab”, vagy „Ctrl + W”).

A.4. *Megjegyzés.* A megoldó tesztelés folyamata hosszadalmas lehet, a futtató számítógép, a tesztelési paraméterek, és a tesztelt megoldó algoritmus hatékonyságának függvényében akár órákig is tarthat.

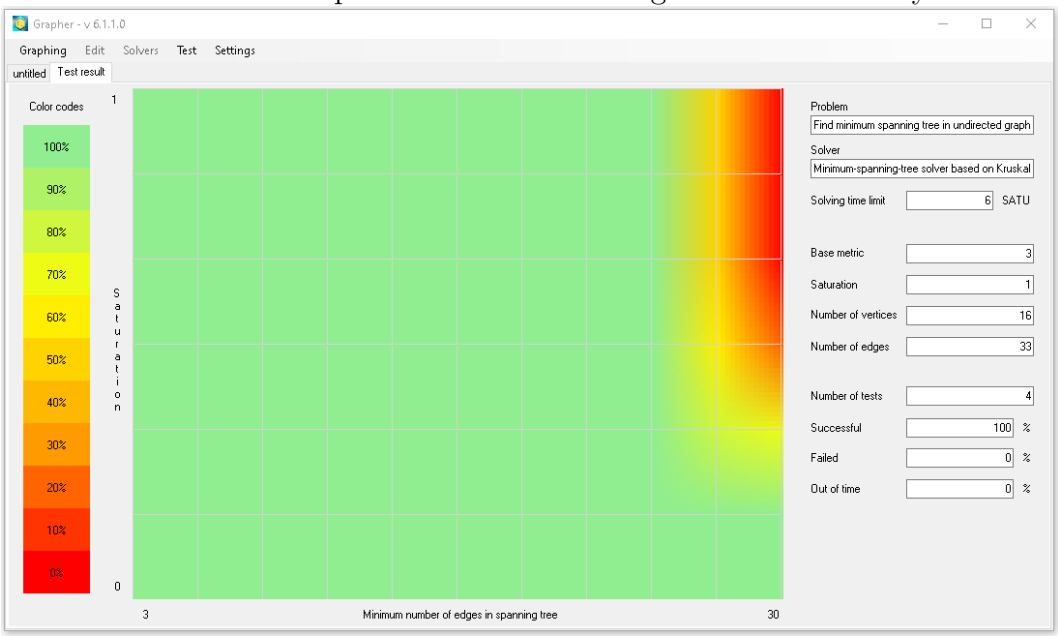
A.7. Beállítások

A „Settings” menüpontra kattintva az A.11. ábrán látható „beállítások” felület válik elérhetővé, ahol három fülön áll módunkban a Grapher jellemzőit módosítani.

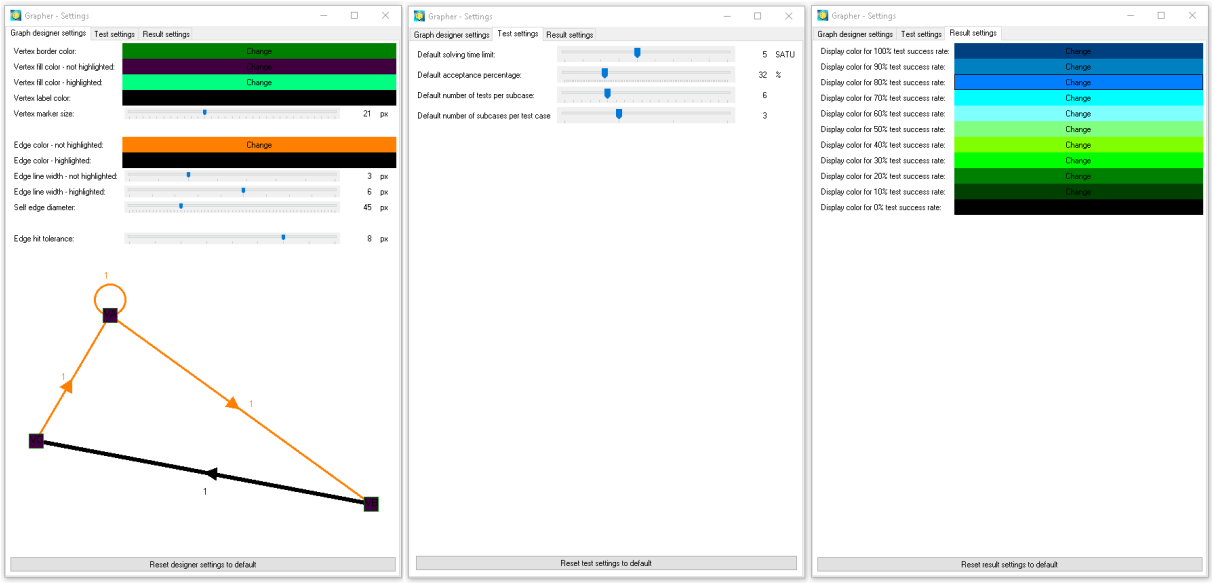
Az első fülön („Graph designer settings”) a gráfrajzolással kapcsolatos attribútumok módosíthatók, mint például megjelenítési színek, vastagságok, méretek, vagy az élekre kattintás sikerességének toleranciája.

A második fülön („Test settings”) a tesztparaméterek alapértelmezett értékeit változtathatjuk meg. Mikor az A.8. ábrán látható felület megnyílik automatikus megol-

A.10. ábra. Grapher – automatikus megoldóteszt eredménye



A.11. ábra. Grapher – beállítások felület fülei



dótesztelés kezdeményezésekor, akkor az ott található négy tesztparaméter a beállításokban megadott értékekre van hangolva alapértelmezetten.

A harmadik fülön („Result settings”) új színkódokat rendelhetünk a vizuális teszteredmények színskálájának 11 lépcsőfokához.

Bármely fülön is változtatunk értéket, az mentésre kerül, és azonnal kifejti hatását a programra. A fülek alján található „Reset ... settings to default” gombbal az adott fülön található attribútumok értékeit visszaállíthatjuk az eredeti értékekre.

B függelék

Telepítési útmutató

Jelen leírás a Grapher program használatbavétele előtti tennivalókat írja le.

B.1. Telepítés csomagból

A http://grapher.asanyad.xyz link által elérhető weboldalon kattintsunk az „Asanyad Grapher – telepítő”-re, és a felkínált setup.msi fájlt mentsük le, majd indítsuk el. A telepítővarázsló „Tovább-Tovább-Kész” módszerrel pillanatok alatt végigfuttatható. Az asztalon megjelenő parancsikon által el is indíthatjuk az alkalmazást.

A Grapher-t célszerű mindig adminisztrátorként indítani, ellenkező esetben bizonyos funkciók hibára futhatnak. Javasolt a parancsikonon tulajdonságaiban a „Futtatás rendszergazdaként” opció bekapcsolása, így a programot mindig rendszergazdaként fogjuk indítani.

B.2. Indítás forrásból

Amennyiben a Grapher-t nem telepített verzióból, hanem – a dolgozathoz csatolt – forráskódja által szeretnénk használni, és tanulmányozni, nyissuk azt meg Visual Studioban. Futtatás előtt a következő előfeltételek biztosítása szükséges még:

- A számítógépünk legyen Windows rendszerű, és legyen rá telepítve a .Net keretrendszer 4.7.2-es, vagy magasabb verziója.
- A http://grapher.asanyad.xyz link alatti weboldalról töltsük le az „AsanyadEncoder”, a „GrapherDevelopersLibrary” és a „WindowsFormsBinder” NuGet csomagokat, és manuálisan referáljuk be őket a Grapher solution-be.
- A „UserInterfaces” projektet állítsuk be indító projektnek.

Ezen lépéseket követően az állomány használatra és vizsgálatra kész állapotba kerül, Visual Sutudio-ból indítható.

C függelék

Fejlesztői útmutató

Jelen dokumentum azon fejlesztőknek hivatott segítséget nyújtani, akik saját megoldó algoritmusukat szeretnék a Grapher program segítségével kipróbálni és tesztnek alávetni valamely, a program által ismert gráfelméleti probléma tekintetében.

Hogy az algoritmusunkat megfelelő formában implementálhassuk, a következők szükségesek:

- Készítsünk egy új C# „Class Library” projektet.
- A http://grapher.asanyad.xyz link alatti weboldalról töltsük le a „GrapherDevelopersLibrary” NuGet csomagot, és referáljuk be a projektünkbe.

A letöltött csomag egy interfész-gyűjtemény, melynek elemei két csoportra bonthatók, a gráf-, és a megoldó-definíciókra. A csomag teljes dokumentációja, így az interfészek property-jeinek, metódusainak, eseményeinek jelentésének és használatának leírása elérhető a https://doc.grapherdevlib.asanyad.xyz link alatt.

C.1. Gráfdefiníciós interfészek

A következőkben felsorolt interfészek által definiált gráf objektumokat ismeri és kezeli a Grapher.

IVertex. Gráf csomópontok és rajtuk végezhető műveletek definíciója.

IEdge. Gráf élek és rajtuk végezhető műveletek definíciója.

IGraph. Gráfok és rajtuk végezhető műveletek definíciója.

Megoldó algoritmus megfelelő elkészítése mellett a gráfdefiníciós interfészek implementálására nincsen szükség.

C.2. Megoldó-definíciós interfészek

A legfőbb interfész az „IGraphSolver”. Metódusainak, eseményeinek leírását szükséges megtekinteni, ebből láthatjuk, hogy az általunk implementált megoldó osztály milyen elven kell, hogy működjön. Ezen interfészt implementálja további 11 interfész, melyek nem rendelkeznek újabb adatokkal, és csak nevükben térnek el egymástól.

Az „IGraphSolver” interfészt ne implementáljuk, hanem a 11 leszármazottja közül valamelyiket annak függvényében, hogy milyen gráfelméleti problémára nyújt megoldást az algoritmusunk. További fontos feltétel, hogy az implementált megoldó rendelkezzék egy paraméter nélküli konstruktorral. Az elkészült projektet .dll-be fordítva a Grapher felismeri azt. Az implementálható megoldóinterfészek rendre a következők:

IEulerianCycleSolver. Akkor implementáljuk ezen interfészt, ha Euler-kör problémára nyújtunk megoldást irányítatlan gráfokban.

IEulerianDirectedCycleSolver. Akkor implementáljuk ezen interfészt, ha Euler-kör problémára nyújtunk megoldást irányított gráfokban.

IEulerianPathSolver. Akkor implementáljuk ezen interfészt, ha Euler-út problémára nyújtunk megoldást irányítatlan gráfokban.

IEulerianDirectedPathSolver. Akkor implementáljuk ezen interfészt, ha Euler-út problémára nyújtunk megoldást irányított gráfokban.

IHamiltonianCycleSolver. Akkor implementáljuk ezen interfészt, ha Hamilton -kör problémára nyújtunk megoldást irányítatlan gráfokban.

IHamiltonianDirectedCycleSolver. Akkor implementáljuk ezen interfészt, ha Hamilton-kör problémára nyújtunk megoldást irányított gráfokban.

IHamiltonianPathSolver. Akkor implementáljuk ezen interfészt, ha Hamilton-út problémára nyújtunk megoldást irányítatlan gráfokban.

IHamiltonianDirectedPathSolver. Akkor implementáljuk ezen interfészt, ha Hamilton-út problémára nyújtunk megoldást irányított gráfokban.

IMaxFlowSolver. Akkor implementáljuk ezen interfészt, ha legnagyobb átfolyás problémára nyújtunk megoldást irányított gráfokban.

IMinimumSpanningTreeSolver. Akkor implementáljuk ezen interfészt, ha legkisebb költségű feszítőfa problémára nyújtunk megoldást irányítatlan gráfokban.

IShortestPathSolver. Akkor implementáljuk ezen interfészt, ha két csomópont közötti legrövidebb út problémára nyújtunk megoldást irányítatlan gráfokban.

D függelék

Alapméretek és telítettségek

A 2.6.4. szakaszban leírtaknak megfelelően jelen függelék tartalmazza a Grapher által ismert gráfelméleti problémák esetében miképp definiálom a gráfok méretének megadására használt alapméretet és telítettséget.

D.1. Hamilton problémák

Mind a négy Hamilton probléma esetén a gráf alapméretét a csomópontok száma jelenti.

D.1.1. Hamilton-út probléma irányítatlan gráfokban

Élek lehetséges legkevesebb száma B alapméret esetén:

$$E_{min} = B - 1$$

Legyen ekkor a szaturáció $S = 0$.

Élek lehetséges legmagasabb száma B alapméret esetén:

$$E_{max} = \frac{B(B+1)}{2}$$

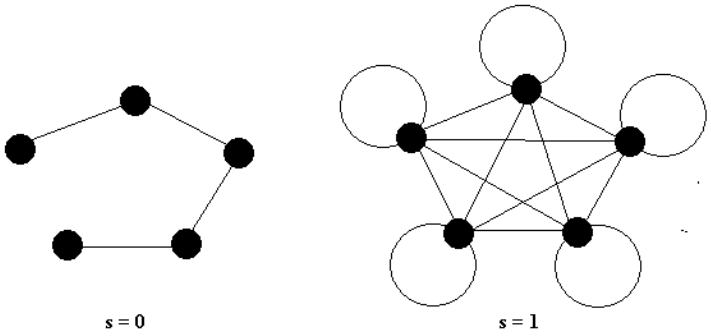
Legyen ekkor a szaturáció $S = 1$.

Ezáltal tetszőleges $0 \leq S \leq 1$ szaturáció mellett lineárisan interpolálva az élek száma a következőképp számítható:

$$E = \left[\frac{S(B^2 - B + 2)}{2} + B - 1 \right]$$

$N = 5$ alapméret esetére példaképp láthatunk $S = 0$ és $S = 1$ telítettségű gráfokat a D.1. ábrán.

D.1. ábra. Telítettség szélsőértékei Hamilton-út probléma esetén irányítatlan gráfokban $B = 5$ alapméret mellett



D.1.2. Hamilton-út probléma irányított gráfokban

Élek lehetséges legkevesebb száma B alapméret esetén:

$$E_{min} = B - 1$$

Legyen ekkor a szaturáció $S = 0$.

Élek lehetséges legmagasabb száma B alapméret esetén:

$$E_{max} = B^2$$

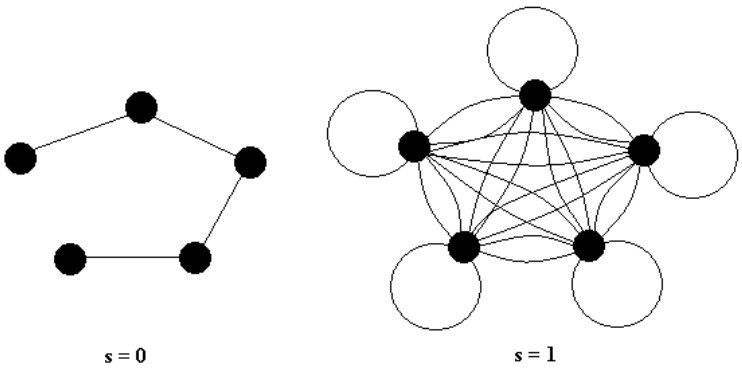
Legyen ekkor a szaturáció $S = 1$.

Ezáltal tetszőleges $0 \leq S \leq 1$ szaturáció mellett lineárisan interpolálva az élek száma a következőképp számítható:

$$E = \left[S(B^2 - B + 1) + B - 1\right]$$

$N = 5$ alapméret esetére példaképp láthatunk $S = 0$ és $S = 1$ telítettségű gráfokat a D.2. ábrán.

D.2. ábra. Telítettség szélsőértékei Hamilton-út probléma esetén irányított gráfokban $B = 5$ alapméret mellett



D.1.3. Hamilton-kör probléma irányítatlan gráfokban

Élek lehetséges legkevesebb száma B alapméret esetén:

$$E_{min} = B \tag{D.1}$$

Legyen ekkor a szaturáció $S = 0$.

Élek lehetséges legmagasabb száma B alapméret esetén:

$$E_{max} = \frac{B(B+1)}{2}$$

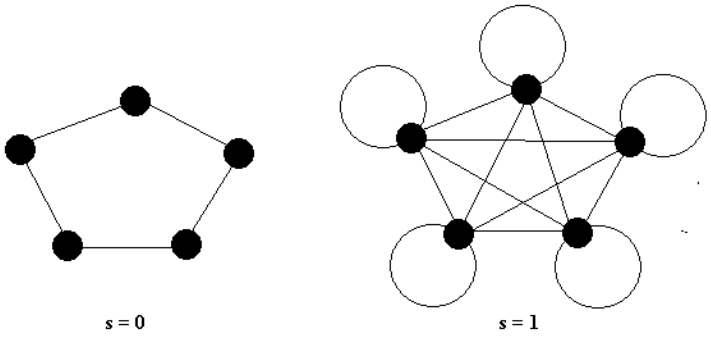
Legyen ekkor a szaturáció $S = 1$.

Ezáltal tetszőleges $0 \leq S \leq 1$ szaturáció mellett lineárisan interpolálva az élek száma a következőképp számítható:

$$E = \left[\frac{S(B^2 - B)}{2} + B \right]$$

$N = 5$ alapméret esetére példaképp láthatunk $S = 0$ és $S = 1$ telítettségű gráfokat a D.3. ábrán.

D.3. ábra. Telítettség szélsőértékei Hamilton-kör probléma esetén irányítatlan gráfokban $B = 5$ alapméret mellett



D.1.4. Hamilton-kör probléma irányított gráfokban

Élek lehetséges legkevesebb száma B alapméret esetén:

$$E_{min} = B$$

Legyen ekkor a szaturáció $S = 0$.

Élek lehetséges legmagasabb száma B alapméret esetén:

$$E_{max} = B^2$$

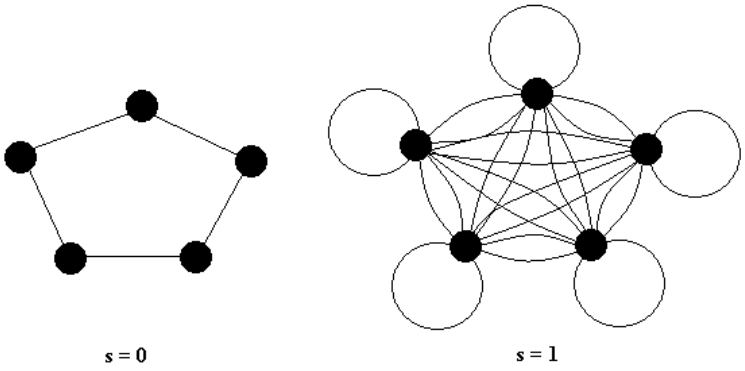
Legyen ekkor a szaturáció $S = 1$.

Ezáltal tetszőleges $0 \leq S \leq 1$ szaturáció mellett lineárisan interpolálva az élek száma a következőképp számítható:

$$E = \left[S(B^2 - B) + B\right]$$

$N = 5$ alapméret esetére példaképp láthatunk $S = 0$ és $S = 1$ telítettségű gráfokat a D.4. ábrán.

D.4. ábra. Telítettség szélsőértékei Hamilton-kör probléma esetén irányított gráfokban $B = 5$ alapméret mellett



D.2. Euler problémák

Mind a négy Euler probléma esetén a gráf alapméretét az élek száma jelenti. Figyeljük meg a következőkben, hogy ezen problémákra definiált metrikákban a szaturáció csökkenését két csomópont összeolvasztása idézi elő.

D.2.1. Euler-út probléma

Mind irányított, mind irányítatlan gráfok esetén igazak a következők. Csomópontok lehetséges legkevesebb száma B alapméret esetén:

$$N_{min} = 1$$

Legyen ekkor a szaturáció $S = 0$.

Csomópontok lehetséges legmagasabb száma B alapméret esetén:

$$N_{max} = B + 1$$

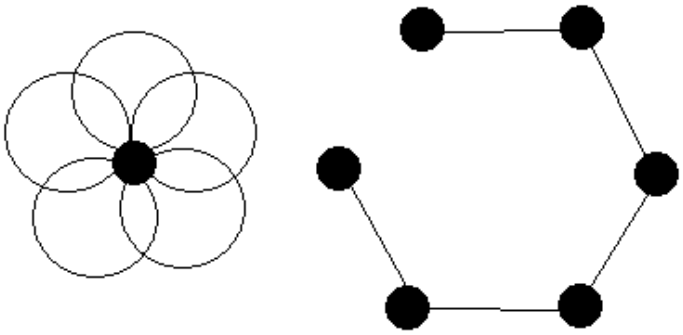
Legyen ekkor a szaturáció $S = 1$.

Ezáltal tetszőleges $0 \leq S \leq 1$ szaturáció mellett lineárisan interpolálva a csomópontok száma a következőképp számítható:

$$N = [1 + S \cdot B]$$

$N = 5$ alapméret esetére példaképp láthatunk $S = 0$ és $S = 1$ telítettségű gráfokat a D.5. ábrán.

D.5. ábra. Telítettség szélsőértékei Euler-út probléma esetén $B = 5$ alapméret mellett



D.2.2. Euler-kör probléma

Mind irányított, mind irányítatlan gráfok esetén igazak a következők. Csomópontok lehetséges legkevesebb száma B alapméret esetén:

$$N_{min} = 1$$

Legyen ekkor a szaturáció $S = 0$.

Csomópontok lehetséges legmagasabb száma B alapméret esetén:

$$N_{max} = B$$

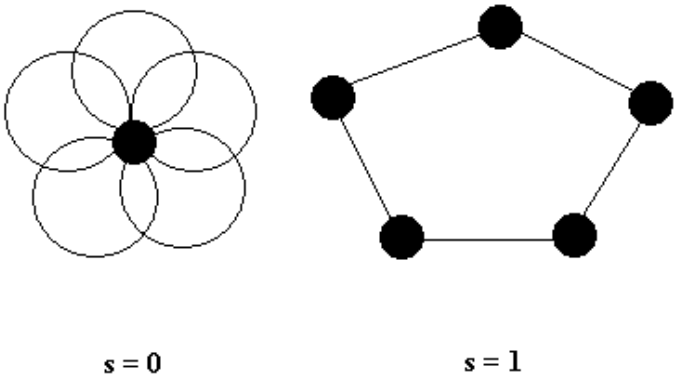
Legyen ekkor a szaturáció $S = 1$.

Ezáltal tetszőleges $0 \leq S \leq 1$ szaturáció mellett lineárisan interpolálva a csomópontok száma a következőképp számítható:

$$N = [1 + S \cdot (B - 1)]$$

$N = 5$ alapméret esetére példaképp láthatunk $S = 0$ és $S = 1$ telítettségű gráfokat a D.6. ábrán.

D.6. ábra. Telítettség szélsőértékei Euler-kör probléma esetén $B = 5$ alapméret mellett



D.3. További problémák

A Grapher további három gráfelméleti problémát ismer, melyek a következők.

- maximális átfolyás problémája
- legkisebb költségű feszítőfa problémája
- legrövidebb út problémája

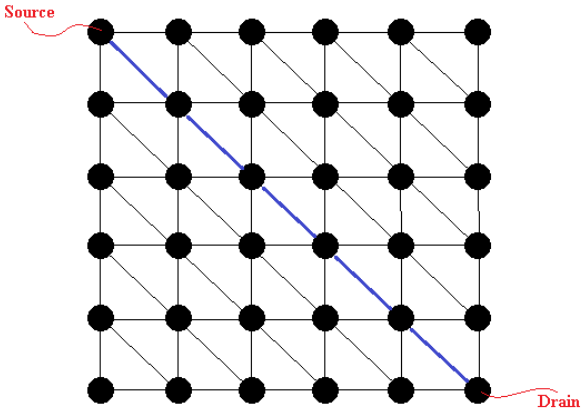
Ezek esetében kevésbé egyértelműen adja magát az alapméret és a telítettség definiálásának miként je, mint a Hamilton- és Euler problémák tárgyalásakor. Mivel mind a három probléma esetében nehéz megfogalmazni bármi nemű fix adatot, amit alapméretnek nevezhetnénk, kötöttségektől mentesen adhatunk definíciót. Ezáltal a közös kezelhetőség érdekében mindhárom probléma esetére a következők szerint definiálom a gráfokat leíró adatpárt:

- Vegyünk egy $K \times K$ darab négyzetből álló rácsot (függőleges és vízszintes rácsvonalakkal).
- Minden négyzetbe rajzoljuk be a bal felső sarokból induló átlóját.
- A vonalak adta minden metszéspontot tekintsünk mint csomópontot, a csomópontok között futó szakaszokat pedig mint éleket.

A felsorolt problémák közül kettő esetében vannak kitüntetett (kiinduló és érkező) csomópontok, legyenek ezek a rács bal felső és jobb alsó sarkában lévő csomópontok. Definiáljuk a gráf alapméretét ekkor mint az élek számát abban az útban, mely a lehető legkevesebb csomópontot érintve köti össze ezen két dedikált csomópontot, melynek konkrét értéke $B = K$. A fenti leírás jellemezte gráfra, és az alapméret definíciójára láthatunk szemléletes példát a D.7. ábrán $B = 5$ mellett.

Tekintsük a B alapmérethez társított $S = 1$ telítettségűnek azon gráfot, mely tartalmazza a fent vázolt gráf minden csomópontját és élét, $S = 0$ telítettségűnek pedig

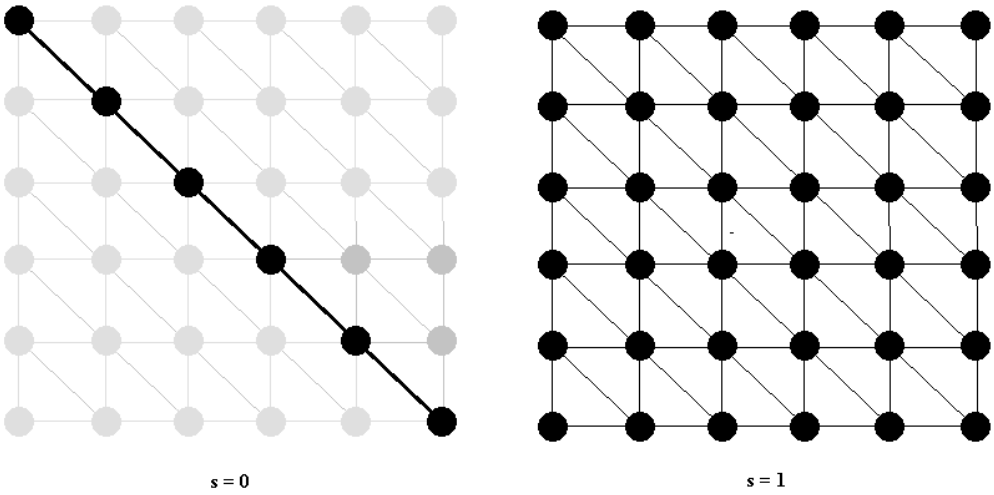
D.7. ábra. Speciális gráf topológia metrika bevezetéséhez



azt, melyben kizárólag a dedikált csomópontokat összekötő legrövidebb út élei és csomópontjai szerepelnek. Szemléletes példát láthatunk ezen szélsőértékekre a D.8. ábrán $B = 5$ alapméret esetére.

$S = 1$ szaturációtól indulva úgy haladhatunk fokozatosan $S = 0$ felé, hogy a bal alsó és jobb felső sarkokból indulva, egyre a minimum felé haladva törlünk éleket (és feleslegessé vált csomópontokat).

D.8. ábra. Telítettségi szélsőértékek $B = 5$ esetén



A gráf éleinek száma $S = 0$ telítettség esetén:

$$E_{min} = B$$

A gráf éleinek száma $S = 1$ telítettség esetén:

$$E_{max} = 3B^2 + 2B$$

Ezáltal tetszőleges $0 \leq S \leq 1$ szaturáció mellett lineárisan interpolálva az élek száma

a következőképp számítható:

$$E = \left[S(3B^2 + B) + B\right]$$

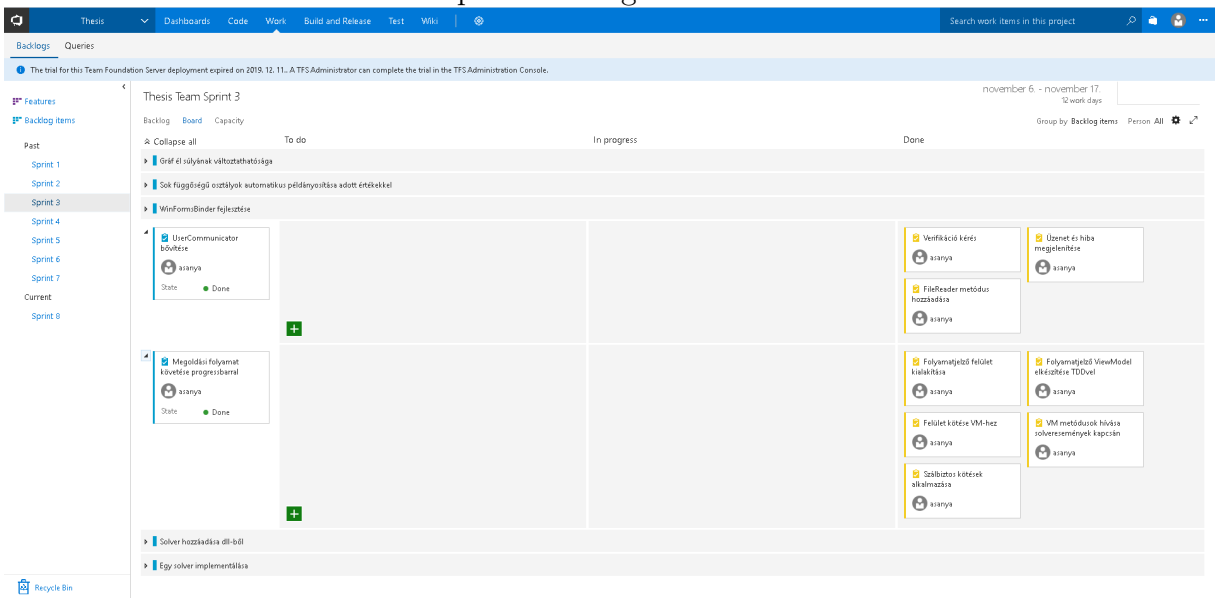
D.1. *Megjegyzés.* Maximális átfolyás problémája esetén szerepet kap a gráf éleinek irányítottsága, melyet a következőképp definiálok:

- Legyen a pozitív irány a balról jobbra vagy/és a fentről lefelé mutató irány, azaz azon irányok, melyek a kiinduló csomóponttól az érkezőhöz való közelebb jutást biztosítják.
- A kiinduló és érkező csomópontokat összekötő legrövidebb út élei mind pozitív irányúak.
- A gráf összes többi éle 0.75 valószínűséggel pozitív irányú.

A Grapher beépített gráfgenerátora ezen elveknek megfelelően készíti a megoldóteszteléshez a tesztgráfokat.

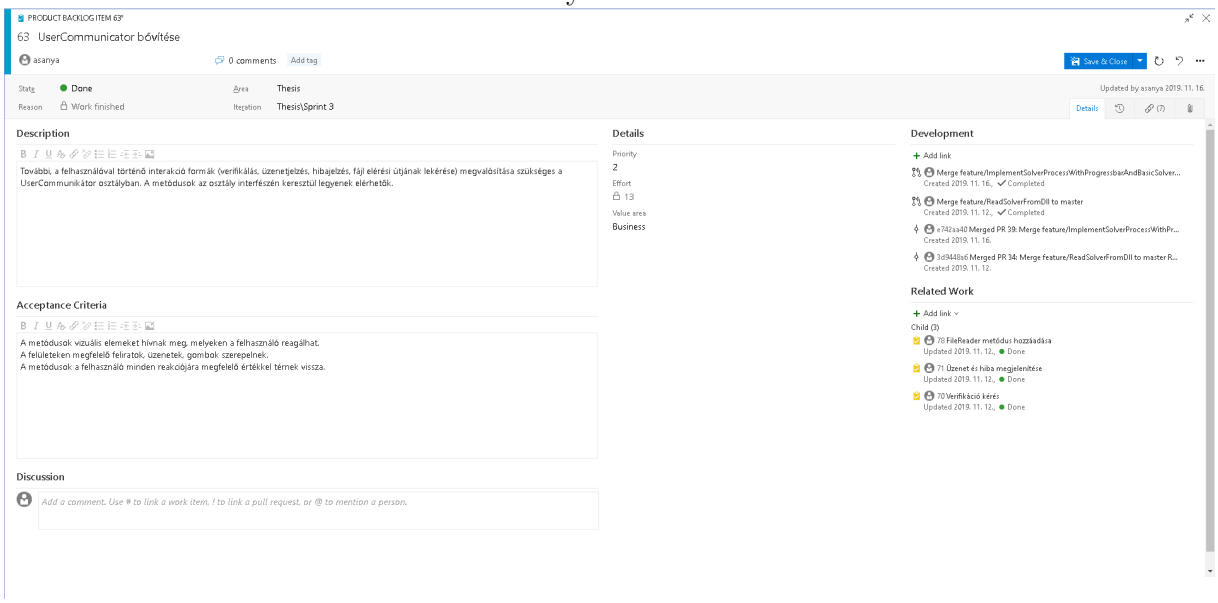
E függelék

TFS képernyőképek

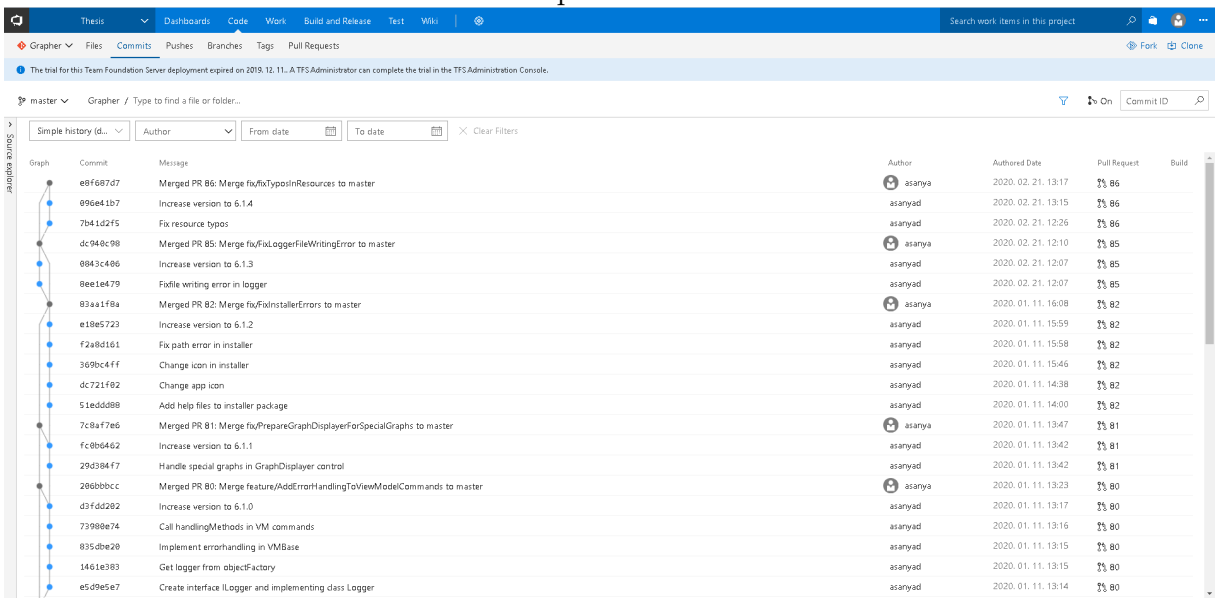


E.1. ábra. A sprint backlog arculata a TFS-ben

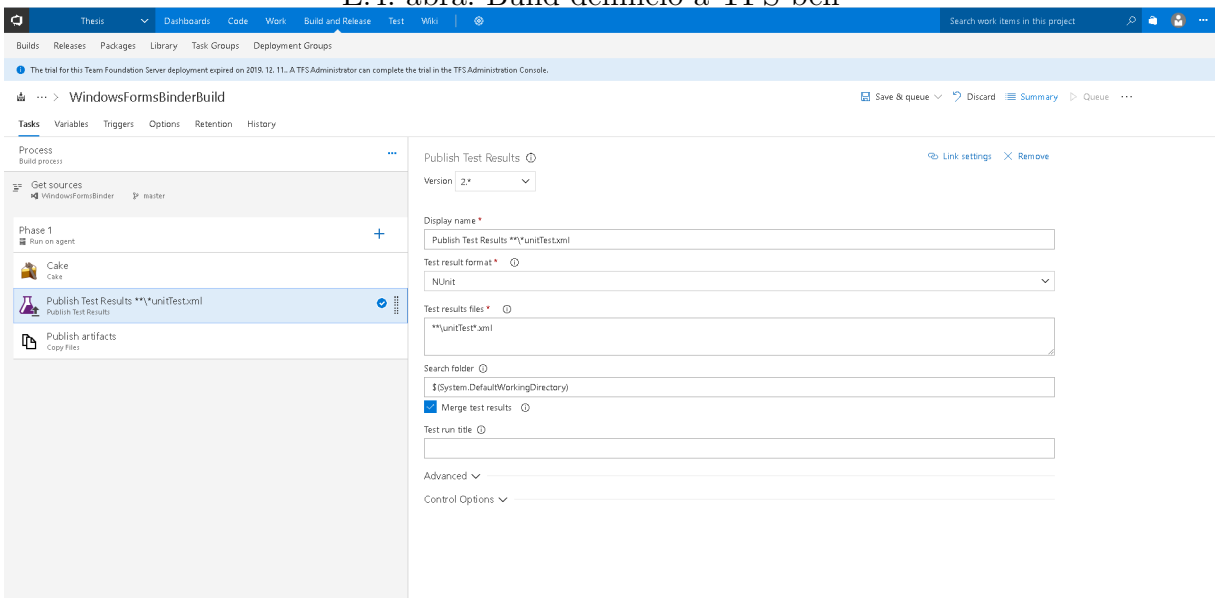
E.2. ábra. A userstory-k leíró oldala TFS-ben



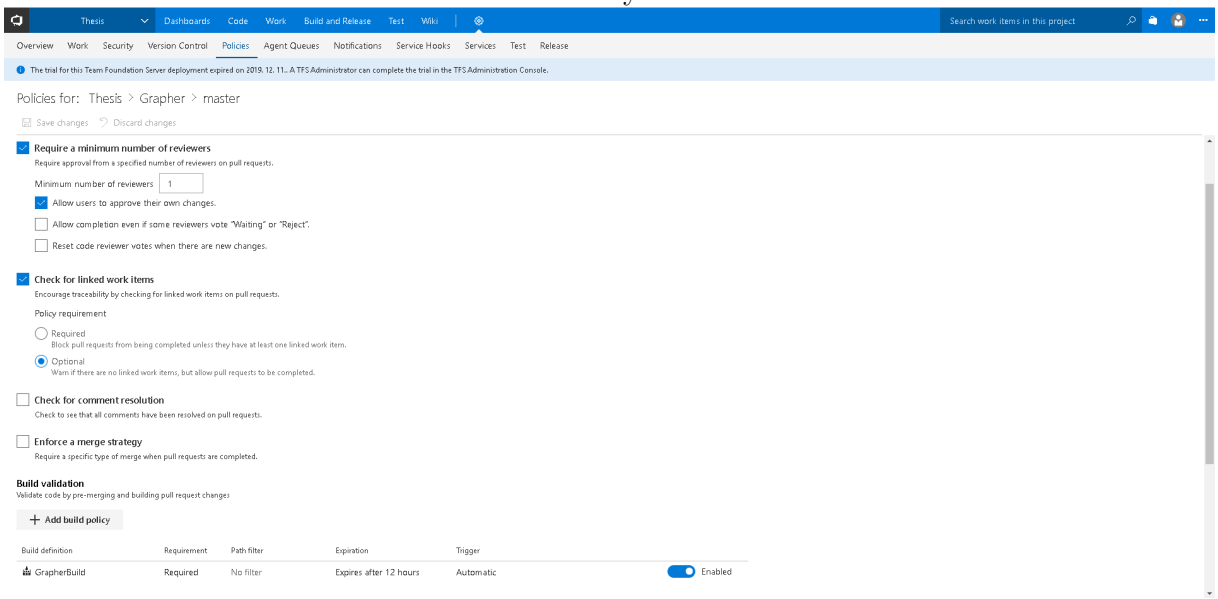
E.3. ábra. A TFS beépített verziókövető felülete



E.4. ábra. Build definíció a TFS-ben



E.5. ábra. Branch irányelvek a TFS-ben



NYILATKOZAT

Alulírott ....Dancsó Sándor....., büntetőjogi felelősségem tudatában kijelentem, hogy az általam benyújtott, Gráfelméleti megoldó-algoritmusok automatizálása tesztelése C# nyelven írt alkalmazással című szakdolgozat (diplomamunka) önálló szellemi termékem. Amennyiben mások munkáját felhasználtam, azokra megfelelően hivatkozom, beleértve a nyomtatott és az internetes forrásokat is.

Tudomásul veszem, hogy a szakdolgozat elektronikus példánya a védés után az Eszterházy Károly Egyetem könyvtárába kerül elhelyezésre, ahol a könyvtár olvasói hozzájuthatnak.

Kelt: Budapest 2020 év ....május... hó ....1.... nap.

.....
aláírás