Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Algoritmikus problémák bonyolultságának előrejelzése tömörítéssel

Papp Pál András

II. évfolyam, mérnök informatikus szak

Konzulens: Dr. Mann Zoltán Ádám

Számítástudományi és Információelméleti Tanszék

2011. október 27.

Tartalomjegyzék

1. Témaválasztás

1.1. Algoritmikus problémákról

A különféle algoritmikus problémák nem csak az informatika különböző szakterületein dolgozókat foglalkoztatják, de egyben a mindennapi életben is lépten-nyomon jelen vannak. Ha útvonaltervezőt használunk, órarendünket osztjuk be vagy épp egy zárthelyi során azt mérlegeljük, hogy mely feladatokat érdemes megoldanunk, akkor különböző híres algoritmikus problémákat oldunk meg, de hasonló problémák vannak jelen a számítógépes hálózatok működése során, áramkörök tervezésekor vagy egyes közgazdaságtani feladatok megoldása esetén is.

A bonyolultságelmélet a matematikának és az informatikának az a közös ága, amely a problémák nehézségének meghatározásával foglalkozik. Ez a szakterület a problémákat különböző bonyolultsági osztályokba sorolja; ezek közül talán a legismertebbek a P és NP osztályok. A P osztályba tartozó problémákra ismerünk polinomiális futásidejű algoritmust. Nagyobb kihívást jelentenek viszont az úgynevezett NP-nehéz problémák, amikre a mai napig nem tudunk hatékony, azaz polinomiális időben futó algoritmust; a kutatás ezek vizsgálatával foglakozik. Az NP-nehéz problémák egy azonos nehézségű problémákat tartalmazó, sokat vizsgált részhalmazát nevezzük NP-teljes problémáknak.

Bár ezekre a problémákra polinomiális futásidejű algoritmust találni mindmáig nem sikerült, rengeteg, különböző eredmény született már a témában. Gyakori az approximációs módszerek fejlesztése, amelyek megelégednek egy közel-optimális megoldás megtalálásával, vagy paraméterizációké, melyek többparaméteres algoritmusok esetén egyes paraméterek lefixálásánál hatékony működést eredményeznek. Szintén sok heurisztika került kidolgozásra a szóban forgó problémákra, amelyek az esetek nagy többségében elég jó valószínűséggel megtalálják a legjobb, vagy egy viszonylag jó megoldást. (Ezek egyike, a genetikus algoritmus részletesen bemutatásra kerül a dolgozat során.)

Ugyancsak sok módszer született az egzakt algoritmusok optimalizálására, probléma-specifikus speciális esetek vizsgálatával a várható futásidő drasztikus csökkentésére, de az a legrosszabb esetben így is exponenciális marad. Ennek köszönhetően NP-teljes esetekben kulcsfontosságú az egyes problémapéldányok bonyolultságának helyes megbecslése [1]. Amennyiben nagyjából tisztában vagyunk egy probléma nehézségével, akkor megbecsülhetjük, hogy körülbelül mennyi

ideig fog dolgozni vele egy egzakt algoritmus, így eldönthetjük, érdemes-e megvárni, hogy lefusson a feladaton, vagy ez feltehetően annyi időt vesz igénybe, amennyit nem tudunk erre áldozni, így használjunk inkább egy heurisztikát. A bonyolultságnak nem csak a helyes algoritmus kiválasztásakor, hanem több probléma több eszközön (például több processzor használatával) való feldolgozásakor is kiemelt szerepe van, ugyanis enélkül nem tudjuk a számítási kapacitás kihasználását optimalizálni; csak akkor hozhatunk megfelelő döntést arról, hogy mely problémák mely gépeken fussanak, ha meg tudjuk becsülni, hogy az adott problémák futása mennyi időt és számítási kapacitást vesz igénybe.

Világos tehát annak a fontossága, hogy az NP-teljes problémák nehézségét minél jobban előre jelezzük; hogy a probléma függvényében megállapítsunk egy értéket, amellyel jó becslést adhatunk egy egzakt algoritmus körülbelüli futásidejére vagy egy heurisztika várható eredményességére. A kutatás során azt vizsgáltam, hogy már ismert módszerekkel mennyire jól jósolhatók ezek a tulajdonságok, illetve hogy egy új módszer létrehozásával tudunk-e esetleg az eddigieknél jobb előrejelzést adni.

1.2. Számítástudományi megközelítés

A problémakört formálisan megközelítő bonyolultságelméletben természetesen van erre egy jól ismert módszer: az adott probléma nehézségét az alapján jellemzik, hogy a futásidő az input méretének milyen típusú függvénye. A már említett P típusú problémáknál ez a függvény polinomiális, míg az NP típusúaknál nem polinomiális fajtájú.

A számítástudományban a problémák nehézségét tehát gyakorlatilag kivétel nélkül az input mérete alapján szokás meghatározni. A megközelítés előnye, hogy a bonyolultság igen gyorsan és problémamentesen számítható. Azonban az már messze nem magától értetődő, hogy ez az elméleti becslés a gyakorlat szempontjából is a lehető legjobb. Bár az input mérete valóban sok információt tartalmaz a nehézségről, egyáltalán nem jellemzi azt pontosan; adott input mérettel rendelkező problémák bonyolultsága között hatalmas eltérés lehet. Ez annak köszönhető, hogy a bonyolultságelmélet az input méretéhez az adott méret esetén elképzelhető legnagyobb nehézséget rendeli, ezzel azt vizsgálva, hogy mennyire lehet bonyolult a probléma a legrosszabb esetben (ezt nevezzük *worst-case bonyolultságnak*). Ennek köszönhetően egy felső becslést kapunk a probléma nehézségére, azonban (az előző fejezetben a bonyolultságbecslés gyakorlati fontosságát illusztráló példákat megfigyelve) azt vehetjük észre, hogy nekünk hasznosabb lenne, ha a probléma bonyolultságát sokkal jobban be tudnánk határolni, mint az csupán egy felső korlát ismeretében

lehetővé válik. Több processzor közötti optimalizáció esetében nyilván nem egy felső határra, hanem a futásidő minél pontosabb ismeretére van szükség, az algoritmustípus választásánál pedig ugyancsak kellemetlen, ha egy pesszimista worst-case becslés miatt nem használunk egzakt algoritmust egy problémán (pedig az valójában lefutott volna rajta elfogadható idő alatt), hanem a bizonytalanabb heurisztika mellett döntünk. A számítástudomány klasszikus szemlélete tehát ideális az elméleti modell megalapozására, a gyakorlatban azonban sokszor lényegesen jobb lenne a szükséges futásidőre, illetve számítási kapacitásra egy pontosabb előrejelzési módszert alkalmazni.

Egy ilyen módszer megtalálásához érdemes alaposan utánanéznünk, hogy milyen különbséget találunk két olyan probléma között, amelyek ugyanolyan input mérettel rendelkeznek, de bonyolultság szempontjából lényeges különbségeket mutatnak. Ehhez építhetünk az NP-teljes problémákon lefolytatott rengeteg empirikus vizsgálat [1, 2] eredményére, felhasználva azokat általános következtetések levonására és szabályszerűségek keresésére.

Ezen tapasztalok egyik legszembetűnőbb jellegzetessége, hogy a valamilyen véletlenszerű módszerrel generált problémák általában sokkal nagyobb nehézségeket produkálnak, mint az ugyanekkora input méretű, különböző gyakorlati alkalmazási területekről összegyűjtött példányok. Az alkalmazási területekről szerzett, általában ember által létrehozott, mesterségesen generált problémák tehát valami okból sokkal könnyebbek a megfigyelt algoritmusok számára; található bennük valami (esetleg nehezen meghatározható) rendszer, struktúra, ami a problémák megoldását általában lényegesen könnyebbé teszi a megoldó programok számára. Az egzakt algoritmusok ennek segítségével hamarabb találnak megoldást, ha megoldás egyáltalán létezik, illetve amennyiben nem, akkor gyorsabban ki tudják zárni a keresési fa bizonyos részeit, ezzel ugyancsak kevesebb idő alatt jutva el a konklúzióhoz. Heurisztikák esetén ugyanez jobb eredményeket, nagyobb sikerességi százalékot eredményez, illetve itt is megnyilvánulhat kisebb futásidőben, ha ez az adott heurisztikánál lehetséges (például genetikus algoritmusoknak strukturált példányok esetén általában sokkal kevesebb generációra van szüksége egy optimális megoldás megtalálásához). A tapasztalat tehát azt mutatja, hogy az input mérete mellett a strukturáltság is lényeges tényező a bonyolultság kialakulásában.

Véletlen, illetve strukturált problémák összehasonlításával szintén több kutatás foglalkozik [3, 4].

1.3. Van jobb előrejelzés?

Természetesen merül fel a kérdés, hogy ha találnánk egy olyan mértéket, amely a vizsgált probléma strukturáltságát méri, akkor abból mennyivel adhatnánk jobb becslést a nehézségre.

A strukturális információ mérésére van egy magától értetődő és közismert módszer: a tömörítés. A tömörítés során egy adatsort a benne található rendszerességek, szabályosságok segítségével próbálunk az eredetinél kevesebb szimbólummal leírni. Minél több szabályosság található az adathalmazban, annál kisebbre tudjuk betömöríteni, így a kisebb tömörített méret strukturáltabb adathalmazt, a nagyobb szabálytalanabb, véletlenebb adatsort jelent.

Több korábbi munka is foglalkozik hasonló kérdésekkel; megmutatták, hogy heurisztikus algoritmusok sikerességét, futásidejét jól lehet becsülni lineáris regresszióval [1] vagy az entrópiához hasonló mennyiségekkel [4], de ezek egzakt algoritmusokat nem vizsgáltak. Szintén ismeretes több, gráfok struktúrájának különböző módon való megkeresésére vonatkozó eredmény [5, 6], ám nem elemezték, hogy ezek milyen kapcsolatban állnak a hozzájuk kötődő algoritmikus problémákkal.

Kutatásom során a következő kérdésekre kerestem a választ:

- **A probléma tömörített mérete jobban előrejelzi-e a bonyolultságot, mint a számítástudományban szinte kizárólagosan használt input-méret?**
- **Már ismert gráf tömörítő eljárások mennyire használhatók ilyen típusú előrejelzésre?**
- **Adható-e olyan algoritmus gráfok tömörítésére, mely még inkább alkalmas a bonyolultság előrejelzésére?**

Ehhez természetesen szükség van egy NP-teljes problémára, amelynek a bonyolultságát vizsgáljuk különböző esetekben. Erre remek választás a gráfszínezés, mivel egy egyszerű és közismert problémakör, amely köré már rengeteg különböző kutatás és vizsgálat épült, és igen változatos alkalmazási területeken fordul elő a gyakorlatban [7, 8]; ezekkel részletesebben foglalkozunk a következő részben.

Szükségünk van továbbá algoritmusokra, amelyekkel megkíséreljük a vizsgált problémákat megoldani, hogy megtudhassuk azok bonyolultságát; szerencsére gráfszínező algoritmusokból is rengeteg született az idők során; ezekből mi két különbözőt, egy erősen optimalizált egzakt algoritmust, valamint egy genetikus algoritmus típusú heurisztikát fogunk használni a problémákon

való futtatásra és a nehézség mérésére.

Ugyancsak nélkülözhetetlen a vizsgálathoz akkora mennyiségű gráf (mint gráfszínezéshez tartozó probléma), amely már statisztikailag megbízható adatokat szolgáltat. Szerencsére ilyenből szintén rengeteg áll rendelkezésre. Nekünk alapvetően két különböző fajtára van szükségünk: olyanokra, amelyeket strukturáltnak tekintünk, valamint olyanokra, amelyek minél kevesebb szabályosságot mutatnak, hisz ennek a két csoportnak az összehasonlításával kaphatunk választ a dolgozat témájaként kitűzött alapkérdésekre. A kutatás során a strukturált példányok regiszterallokációs problémák gráfjai lesznek; ezek nem csak egy sokat vizsgált alkalmazási területről származnak, hanem ráadásul gráfszínezésre visszavezethető probléma eredményeként jöttek létre [9]. A strukturálatlan gráfokat véletlen gráfok fogják reprezentálni, melyeket az először Erdős és Rényi által használt G(n, p) véletlen modellel [10] generálunk. Ezek a gráfok és algoritmusok a 2. fejezetben kerülnek részletes bemutatásra.

A dolgozat négy különböző tömörítési eljárás előrejelző-képességét vizsgálja meg és hasonlítja össze, melyek közül három külső programok használatára támaszkodik. Ezek:

- A szomszédossági mátrix zip archívumba való tömörítése
- A Minimum Description Length elv használatával való tömörítés
- A Subdue program használata, ami ismétlődő részstruktúrák keresésével tömörít egy gráfot

A tömörítési eljárások részletes leírása a 3. fejezetben olvasható.

A kutatás során a fő kihívás a negyedik vizsgált eljárás, egy saját tömörítőprogram megírása volt, amely lehetőleg mind a számítástudomány becslésénél, mind az előző három módszernél jobb predikciót ad egy gráf ismeretében a gráfon futó algoritmusok (így a gráfszínezés) bonyolultságáról. A program a Graph Structure Analyzer (GSA) nevet viseli, és tulajdonképpen egy genetikus algoritmus, melyben a gráf különböző leírásai fejlődnek, és a program ezek közül próbálja megtalálni a lehető legjobbat. Az egyes leírásokban a gráfot olyan (esetleg egymást átfedő) részegységek összességeként építjük fel, amelyeknek a gráfproblémák nehézségének kialakulásában gyakran kulcsszerepe van. Ilyenek például a teljes részgráfok, az üres részgráfok vagy az utak. A dolgozat részletesen leírja a GSA működésének alapelveit, és megmutatja, hogy az valóban alkalmas gráfok tömörítésére, valamint struktúrájuk megkeresésére, és így a gráfszínezés bonyolultságának előrejelzésére.

Végül a 4. fejezet a kutatás eredményeit mutatja be.

1.4. Gráfok színezése

Gráfnak nevezzük és $G(V,E)$-vel jelöljük *V csúcsok* és őket összekötő *E élek* összességét.

A pontok számára általában $|V(G)| = n$, az élekére $|E(G)| = e$ a megszokott jelölés. Amennyiben a gráf két pontja között fut él ($\{v_1, v_2\} \in E$), akkor a két csúcsot (v_1-et és v_2-t) *szomszédosnak* vagy *összekötöttnek* nevezzük. Ha egy él két végére ugyanaz a pont illeszkedik, akkor azt *hurokélnek*, ha két pont között több él is húzódik, akkor ezeket *többszörös* vagy *párhuzamos éleknek* hívjuk. Ha egy gráf nem tartalmaz hurokéleket és többszörös éleket, akkor *egyszerű gráfnak* nevezzük. A kutatás során csak egyszerű gráfokkal foglalkoztam.

Azt mondjuk továbbá, hogy a $G'(V',E')$ gráf a G gráfnak *részgráfja*, ha $V' \subset V$, $E' \subset E$ és $\{v_1, v_2\} \in E' \Rightarrow v_1, v_2 \in V'$.

Egy olyan gráfot, amelyben bármely két pont éllel van összekötve, *teljes gráfnak* hívunk. Hasonlóan, azt mondjuk, hogy egy gráf *üres gráf*, ha semelyik két pontja nincs éllel összekötve.

Útnak nevezzük csúcsok olyan sorozatát, amelyek mind különbözők, és közülük bármely egymást követő kettő szomszédos. A sorozat első és utolsó pontjai az út *végpontjai*.

Azt mondjuk, hogy a gráf *összefüggő*, ha bármely két pontja (mint végpontok) között fut út. Az összefüggőség, mint ekvivalenciareláció, ekvivalenciaosztályokra osztja a gráfot, amik maximális összefüggő részhalmazok. Ezeket a részgráfokat *komponensnek* nevezzük.

Az adott v_1 pontra illeszkedő élek számát a v_1 *fokszámának* nevezzük.

Ha algoritmikus problémákkal foglalkozunk, igen gyakran futunk bele gráfelméleti problémákba, mivel a legismertebb NP-teljes példányok között rengeteg az ilyen [11]. A gráfszínezés, a Hamilton-kör keresése, a legnagyobb klikk megkeresése, a legkisebb lefedő csúcshalmaz megkeresése, a leghosszabb út megkeresése vagy két gráf legnagyobb közös részgráfjának megtalálása csak néhány a leggyakrabban előkerülők közül. Ezért gyakori megközelítés, hogy a gráfok vizsgálatával próbálunk minél többet megtudni a szóban forgó problémákról.

Legyen adott egy gráf, amelynek a csúcsaihoz színeket kell rendelnünk úgy, hogy szomszédos csúcspárhoz nem rendelhetjük ugyanazt a színt. Annak a problémának az eldöntését, hogy ezt meg tudjuk-e tenni egy adott számú színnel, illetve egy megfelelő ilyen hozzárendelési konstrukciót hogyan tudunk megtalálni, gráfszínezési problémának nevezzük. A legkisebb olyan számot, amelyre

igaz, hogy a gráf színei kiszínezhetők a megadott feltételek szerint ennyi színnel, a gráf *kromatikus számának* nevezzük, jelölése általában χ-vel történik.

Mivel a kutatás által megfigyelt algoritmikus probléma a gráfszínezés, érdemes további fogalmakat tisztáznunk. A gráfszínezés mint eldöntési probléma azt jelenti, hogy adott egy G gráf és egy *k* egész szám, és azt akarjuk eldönteni, hogy kiszínezhető-e a gráf *k* színnel úgy, hogy semelyik szomszédos pontpár nem kapja ugyanazt a színt. Ha a gráfszínezésről mint optimalizálási problémáról beszélünk, akkor azt akarjuk megtudni, hogy mi az a legkisebb *k* szám, amelyre igaz, hogy a gráf színezhető *k* színnel (tehát χ értékét keressük).

Mint említettük, a gráfszínezés (az egy, illetve két színnel való színezhetőség megállapításának kivételével) NP-teljes probléma, azok közül is az egyik legjobban vizsgált [12, 13, 14]. Nagyon sok technikát fejlesztettek ki az egzakt algoritmusok futásidejének jelentős csökkentésére, és szintén rengeteg különböző heurisztikát írtak, melyek nagyon jó eséllyel találnak optimális vagy közel optimális megoldást a problémára [15, 16].

Fontos még megjegyeznünk, hogy két, különböző kromatikus számú gráf esetében nem teljesen világos, hogy mit tekintünk ugyanannak a problémának a két gráfon, ami igencsak lényeges lehet akkor, ha a két gráf bonyolultságát össze akarjuk hasonlítani. Első gondolatra talán logikusnak tűnhet, hogy egy adott *k* számra a két gráf *k* színnel való színezését hasonlítsuk össze. Az empirikus vizsgálatok azonban azt mutatják, hogy a gráfok színezése egyértelműen akkor a legnehezebb, ha a használt színek száma a kromatikus szám közelében van, és az ettől való távolodáskor drasztikusan csökken [16]; a színezés nehézsége tehát erősen függ a gráf χ értékétől. Felmerül ekkor az a logikusabb alternatíva, hogy a két gráf színezésének bonyolultságát úgy hasonlítsuk össze, hogy mindkettőnél a saját kromatikus számának megfelelő számú színt használunk, így mindkettőnél a legnagyobb bonyolultságot mutató esetet figyeljük. A kutatás során mindkét lehetőséget megvizsgáltam.

A gráfszínezés vizsgálata azért is kiemelten fontos az NP-nehéz problémák között, mert a gyakorlatban meglepően sokszor találunk erre visszavezethető feladatokat, ráadásul ezek a legkülönbözőbb szakterületekről kerülnek elő.

Ilyen alkalmazási terület például a regiszterallokáció [7, 9], mely során a változók felelnek meg a gráf csúcsainak, a processzor regiszterei pedig a használható színeknek. A kutatás során erről a problematikáról még részletesebb leírást adunk.

Általánosságban is elmondható, hogy leggyakrabban ütemezési feladatok során botlunk bele a gráfszínezésbe [8]. Adott elvégzendő feladatok egy halmaza (a gráf csúcsai), és különböző

időintervallumok (a színek), valamint olyan feladatpárok, amelyeket nem lehet ugyanazon intervallumban elvégezni (élek). Ekkor a gráf kromatikus száma épp az összes feladat elvégzéséhez szükséges minimális idő lesz.

Hasonlóan a gráfszínezésre vezethető vissza a frekvenciakiosztás problémája [17]. Ebben az esetben frekvenciákat rendelünk különböző eszközökhöz úgy, hogy célunk a kialakuló interferencia minimalizálása. A csúcsok ekkor az eszközöknek, a színek a frekvenciáknak felelnek meg.

A példák jól illusztrálják, hogy a gráfszínezéssel rengeteg különböző alkalmazási területen találkozhatunk, így az idők során az egyik legtöbbet kutatott, leginkább megismert NP-teljes problémává vált. Ennek köszönhetően az egyik legjobb választási lehetőség, ha olyan algoritmikus problémát keresünk, amely bonyolultságát a kutatás során vizsgálni akarjuk.

2. A kutatás eszközei

2.1. A BCAT rendszer

A kutatás során a használt algoritmusok és programok a Budapest Complexity Analysis Toolkit[18]-en (BCAT) keresztül futottak. A BCAT egy összetett keretrendszer, amely optimalizációs problémák és ezeken futó algoritmusok implementálására és értékelésére szolgál. Többféle véges matematikai vonatkozású algoritmust és adatstruktúrát ismer, melyekhez a rendszer moduláris felépítésének köszönhetően a felhasználó könnyen hozzáadhatja sajátjait.

Működése során a BCAT lefuttatja algoritmusok egy halmazát problémák egy halmazán. A felhasználónak csupán annyi a dolga, hogy (a konfigurációs fájlban) meghatározza a kívánt algoritmusokat és problémákat, s ezután a rendszer a kiválasztott algoritmusok mindegyikét lefuttatja a meghatározott problémák mindegyikén. A működéshez és az eredmények kiértékeléséhez szükséges egyéb teendőket (annak megállapítása, hogy az adott algoritmus futtatható-e az adott problémán; az algoritmusok által adott eredmények rendszerezett kiírása; a futásidő mérése; a futás alatti események naplózása, stb.) a BCAT szintén elvégzi.

Algoritmusokon és problémákon kívül a keretrendszer támogatja még az analyzerek implementálását. Az analyzerek működésükben nem különböznek az algoritmusoktól, csupán szemléletbeli különbség miatt alkotnak külön kategóriát. Mind egy algoritmus, mind egy analyzer egy adott problémán fut és annak függvényében ad vissza valamilyen eredményt, de míg előbbinek a probléma megoldása a feladata, az analyzerek célja valamilyen adatok összegyűjtése az input problémáról. Például míg egy algoritmus megállapítja, hogy van-e egy adott gráfban Hamilton-kör, addig egy analyzer célja lehet például a gráf átlagos fokszámának kiszámítása, hogy utána a felhasználó összefüggést kereshessen az átlagos fokszám és a Hamilton-kört kereső algoritmus eredményessége, futásideje között. Mivel a kutatás részét képző, általam írt GSA feladata szintén a gráf egy mértékének (az adott tömörítési módszerrel kapott tömörített méret) megállapítása, ez ugyancsak analyzerként került implementálásra.

A kutatásban a gráfok színezésére felhasznált két algoritmus [16] szintén a BCAT rendszerben lett leprogramozva, futásuk ezen keresztül történik.

2.2. A vizsgált gráfok

2.2.1. Regiszterallokációs gráfok

Regiszter allokációnak [7, 9] azt az optimalizációs problémát nevezzük, amikor egy célprogram nagyszámú változójához kell hozzárendelnünk a processzor néhány (ennél általában kisebb számosságú) regiszterét. Nagyon fontos, hogy minél több (lehetőleg az összes) változó regiszterben legyen tárolva, mert ellenkező esetben a RAM-ban kap csak helyet, ennek az elérése pedig nagyságrendekkel lassabb a regisztereknél. (A regiszterallokációs probléma gyakori rövidítése a RAP, amelyet én is használni fogok a dolgozat során.)

Ez az optimalizálás programírás során általában a programozó elől rejtett módon hajtódik végre, mégpedig a fordító által. A folyamat lényege abból a tényből következik, hogy általában nem hivatkozunk minden változóra egy időben, így azon változókat, amelyekre nincs egyszerre szükség, akár tárolhatjuk egyazon regiszterben is. Világos, hogy innentől a probléma ekvivalens a gráfszínezéssel: a gráf csúcsait a változók alkotják, és azon párok között fut él, amelyek egyszerre vannak használatban. Az egyes színek különböző regisztereknek felelnek meg. A változókhoz (csúcsokhoz) kell hozzárendelnünk a regisztereket (színeket) úgy, hogy két egy időben használt (szomszédos) csúcshoz nem rendelhetjük ugyanazt.

A kutatás során a strukturált gráfokat ilyen regiszterallokációs problémák során keletkező gráfok alkották. A felhasznált gráfok a Princeton egyetem honlapjáról származnak [19], ahol közel 27000 ilyen példány érhető el. Ezek változatos csúcs- és élszámmal rendelkeznek; a kutatáshoz használt kb. 60 darab mind az első 2000-ből lett válogatva.

A szóban forgó problémák mind 32 regiszteres gépekre voltak tervezve, amelyből azonban 11 különleges célokra van fenntartva, így a változók tárolásához 21 regiszter (a színezéshez 21 szín) áll rendelkezésünkre. Mindegyik probléma 21 változóval kezdődik, amelyek közül bármelyik kettő között interferencia van; ez a gráfunkban egy 21 méretű klikket jelent, ami a rendelkezésre álló 21 színnel egybevetve annyit jelent, hogy a használt regiszterallokációs gráfok mindegyikének pontosan 21 a kromatikus száma. Ezt a további változók követik, valamint az interferencia-viszonyok, amelyekből a gráf leírását megkapjuk.

A konkrét vizsgálatokhoz 34, 48, 58, 63, 78, 83, 89, 110, továbbá 122 és 152 pontú regiszterallokációs gráfokat használtam. Ezeknek a számoknak semmilyen kitüntetett jelentősége nincs, csupán olyan számok, amelyek nagyjából egyenletesen oszlanak el a vizsgálni kívánt tartományban, és ilyen csúcsszámú regiszterallokációs gráfokból állt éppen rendelkezésre elegendő

mennyiségű.

A 34, 48, 58 és 63 csúcsú gráfokból egyenként 7 darab, 78 és 83 csúcsúból 6 darab, 89 és 110 csúcsúból 5 darab gráfot vizsgáltam minden esetben. Mivel kevesebb állt rendelkezésre, a 122, illetve 152 csúcsúakból már csupán két darabot használtunk a kutatás során, de az ezekhez tartozó adatok is többé-kevésbé megbízhatónak látszottak, nagyjából beleillettek a mintákba, nem bizonyultak szélsőséges eseteknek.

Az eredmények kiértékelésekor az adott csúcsszámmal rendelkező példányokat minden esetben egy adatpontnak tekintettük. Ezt úgy kell érteni, hogy mind a tömörítési eljárásokat, mind az algoritmusokat lefuttattuk például az összes 58 pontú gráfon, majd úgy vettük, hogy egy 58 csúcsú gráfhoz a kapott tömörített méretek, illetve futásidők átlaga tartozik. Ezzel (mivel öt-hat gráf már elég változatos mintának tekinthető) kiküszöböltük az esetleges nagy szórásból adódó problémákat; ha egy 58 pontú gráfpéldány „mérési hiba”, például valamiért sokkal kisebb a tömörített mérete, mint az ennyi csúcsú regiszterallokációs gráfoknál szokásos, akkor is csak kicsit húzza lefelé az átlagot, ráadásul a különböző szélsőségek korrigálják egymást. Így az átlagszámítással jó közelítést kaphatunk az adott pontú gráfok általános tulajdonságaira.

2.2.2. Véletlen példányok

Az előbbi strukturált gráfokkal olyanokat akarunk összehasonlítani, amelyek strukturálatlanok, bármilyen szabályosságból viszonylag kevés található bennük. Ehhez valamilyen véletlen módszerrel generált példányokra lesz szükségünk.

Véletlen gráfok előállítására rengeteg különböző modell született, ezekből a BCAT többet is tud generálni. A kutatásokhoz ezek közül a legegyszerűbbet és leginkább magától értetődőt, az úgynevezett (n, p) modellt [10] választottam.

Az (n, p) modellben úgy generálunk n pontú egyszerű gráfot, hogy veszünk n pontot, és ezekből bármely kettőt a többitől függetlenül p valószínűséggel összekötjük. (Tehát például p=0 üres gráfot, p=1 teljes gráfot eredményez.) Mind az egyszerű megvalósíthatóság, mind a jól látható szabálytalanság emellett a módszer mellett szól.

Mivel a kutatás során azt akarjuk megállapítani, hogy a bonyolultságra nézve milyen hatásai vannak a gráf strukturáltságának, kulcsfontosságú, hogy a két vizsgált gráfcsoport csupán ebben az egy tényezőben mutasson jelentős különbségeket, ellenkező esetben ugyanis az eltéréseket okozhatják az egyéb különbségek is. Mind a gráf csúcsainak száma, mind az élek száma jelentős lehet egy probléma szempontjából, fontos, hogy olyan példányokat hozzunk létre, amelyek mind

csúcs-, mind élszámban közel állnak a vizsgált strukturált gráfokhoz.

Ehhez a következő módszert alkalmaztam: minden egyes vizsgált regiszterallokációs gráfhoz létrehoztam annak egy „véletlen megfelelőjét”. Az adott regiszterallokációs gráfhoz olyan véletlen gráfot alkottam, amely csúcsainak *n* száma megegyezik a regiszterallokációs gráf csúcsszámával, a *p* valószínűség értékét pedig úgy választottam, hogy a kapott gráfban az élek várható értéke (ami természetesen $\frac{p \cdot n \cdot (n-1)}{2}$) a RAP gráf éleinek számával egyezzen meg (tehát egy *n* pontú, *e* élű regiszterallokációs gráfhoz *p* paraméter értékére $\frac{2 \cdot e}{n \cdot (n-1)}$ adódik).

A kapott élszámokból megfigyelhető, hogy (főleg nagyobb csúcsszámú véletlen gráfok esetén) az így kapott élszám eltérése a RAP gráf élszámától csak néhány százalék, tehát valóban (strukturáltságot leszámítva) nagyon hasonló tulajdonságú gráfokat kaptunk.

2.3. A felhasznált algoritmusok

2.3.1. Egy egzakt algoritmus

Egzakt algoritmusoknak nevezzük azon algoritmusokat, melyek valamiképpen a keresési tér minden elemét megvizsgálják vagy kizárják. Ez eldöntési probléma esetén azt jelenti, hogy amennyiben létezik megoldás, akkor azt mindenképp megtalálják, ha pedig megoldás nem létezik, akkor ennek megállapítására is képesek; optimalizálási problémánál pedig azt eredményezi, hogy teljes bizonyossággal megtalálják a legjobb megoldást. Sajnos NP-teljes problémákról lévén szó, ezt csak úgy tudják megtenni, hogy az input méretének a futásidejük nem polinomiális függvénye, így nagyobb példányokra ritkán lehet alkalmazni őket [14, 15, 16].

Az egzakt algoritmusok egyik legelterjedtebb fajtája a Branch-And-Bound (*BB*) módszer, melyet általában akkor alkalmazunk, ha a feladatban adott objektumokhoz értékek egy sorát rendelhetjük. (Például gráfszínezés esetén a pontokhoz a színeket, míg egy SAT probléma esetén a változókhoz a {0, 1} elemeit.) Az algoritmus futásakor sorra vesszük az egyes objektumokat, és az éppen vizsgálthoz hozzárendelünk sorra minden lehetséges értéket. Amennyiben egy érték nem okoz ellentmondást a megoldási kísérletünkben (gráfszínezésnél például: nem keletkeznek azonos színű szomszédos csúcsok), akkor továbblépünk a következő objektumra. Ha viszont azt kapjuk, hogy az eddigi folyamatot már nem tudjuk jó megoldásra kiegészíteni (már hozzárendeltük ugyanazt a színt két szomszédos csúcshoz), akkor visszalépés (*backtrack*) történik; az utolsó hozzárendelést visszavonjuk, s az előző objektum vizsgálatát folytatjuk a következő hozzárendeléssel.

A kutatásom során a gráfok színezésére használt két algoritmus egyike a BCAT-ben már implementált GraphColouringBB, egy egzakt algoritmus, ami az előbbi technikát használja gráfok színezésére. Ennek köszönhetően persze exponenciális futásidővel rendelkezik, így kulcsfontosságú a futásidő csökkentése. A program több különböző módszert is alkalmaz erre, például heurisztikákat a vizsgálandó csúcsok és hozzárendelhető értékek közül való választás sorrendjére, amellyel várhatóan hamarabb megoldást találunk, ha létezik ilyen (ellenkező esetben úgyis mindet végig kell próbálnunk), vagy a színek szimmetriájának megtörését (amennyiben több szín van még, amit eddig egy csúcshoz sem használtunk, akkor a hozzárendelést ezek közül csak az egyikkel próbáljuk ki, mivel a színek között nincs különbség). A BB tehát egy jól optimalizált, ám így is exponenciális futásidejű algoritmus, amely mindig a legjobb megoldást találja meg.

A kutatás szempontjából kulcsfontosságú eldöntenünk, hogy mit tekintünk a probléma bonyolultságának. Szerencsére a GraphColouringBB esetében erre természetesen adódik a futás során történt backtrackek száma. Branch-And-Bound algoritmusokkal vizsgált problémák bonyolultságának mérésékor ez gyakori megközelítés, mivel (a futásidővel ellentétben) ez csupán a programtól függ, a használt eszköz teljesítményétől független. Minél nehezebb a probléma, annál többször jut zsákutcába a színezési folyamat, így annál nagyobb lesz a backtrackszám, vagyis jogosan várhatjuk, hogy jelzi számunkra a probléma nehézségét.

2.3.2. Egy heurisztika

Mivel az egzakt algoritmusokat exponenciális futásidejük miatt csak kisebb méretű bemeneteknél tudjuk alkalmazni, NP-teljes problémák esetén gyakori a heurisztikák használata. Heurisztikának nevezünk egy olyan algoritmust, amely, bár nem találja meg teljes bizonyossággal a problémára való legjobb megoldást, az esetek többségében egy ehhez igen közel lévő eredményt szolgáltat polinomiális időben [20].

A Branch-And-Bound mellett a másik gráfszínezésre felhasznált algoritmus egy heurisztika, típusa szerint egy genetikus algoritmus, amely a GraphColouringGA névre hallgat, és szintén a BCAT rendszerben van implementálva.

A GraphColouringGA szintén hatékony eszköz gráfok színezésére, mely többek között felhasználja a mohó színezést, valamint új egyedek létrehozásakor egy triviális hibák kijavítására szolgáló eljárást. Alapvetően elmondható róla, hogy amely problémákra az egzakt algoritmusunk kis bonyolultságot ad vissza, azokat általában a genetikus algoritmus is ki tudja színezni a kromatikus számuknak megfelelő számú színnel, és olyan példányokra is viszonylag jó megoldást (kromatikus szám fölött néhány színnel való színezést) talál, amelyeken a Branch-And-Bound

típusú programunk már nagyon sokáig fut.

Sajnos a GraphColouringGA esetében már egyáltalán nem magától értetődő, hogy hogyan mérjük le egy probléma nehézségét. Mivel a genetikus algoritmusok viselkedését jelentősen befolyásolhatja a véletlen, így használatukkor általában a program fő részét képző megoldáskeresést többször is lefuttatják, hogy csökkentsék a jó megoldások csupán balszerencse miatt való elmulasztásának esélyét. Ezt tudva jó ötletnek tűnhet az eldöntési problémánk nehézségét az alapján meghatározni, hogy *r* darab futtatásból hányszor sikerül a gráfot a rendelkezésre álló számú színnel kiszínezni, vagy hogy hány futtatás szükséges, mire az algoritmus talál egy megfelelő színezést. A gyakorlati tapasztalatok azonban azt mutatták, hogy a vizsgált problémákon a különböző futtatások között nem adódnak lényeges különbségek; a probléma nehézségétől függően általában vagy csak nagyon ritkán (0, 1, esetleg 2 futtatás alkalmával), vagy majdnem mindig (*r*-1-szer vagy *r*-szer) talál megoldást a heurisztikánk, így az „*r*-ből hányszor találunk megoldást” modell nem működőképes. Hasonlóan, ha egyáltalán megoldást talál, akkor ez általában már az első 2-3 futás során megtörténik, így a „hányadik futás alkalmával találunk megoldást” jellegű adatok között sem lesznek lényeges különbségek, vagyis a másik ötlet sem használható.

A kutatás során ezért a GA-hoz rendelt bonyolultságnak a futásidőt tekintettem. Az algoritmus ugyanis, amint megoldást talál, abbahagyja működését, így az, hogy ez mennyi idő elteltével következik be, jól jellemezheti a megoldást. (A futásidőket *Intel Core2 Duo* processzorral (2M Cache, 2.4 GHz) és 2 GB RAM-mal rendelkező gépen mértem, *Windows XP* alatt.)

Lényeges még megjegyeznünk, hogy az egzakt algoritmusunkkal szemben a gráfszínező GA nem egy determinisztikus program, így többszöri futás esetén más bonyolultságokat adhat. Ezért itt kiemelten fontos, hogy a felhasznált adatpontokat több probléma átlagolásával kaptuk, tompítva így az esetleges mérési hibák hatását.

3. Tömörítési eljárások

3.1. Tömörítés ZIP formátumban

Mivel a gráfban hordozott információ leírásának legegyszerűbb és leggyakoribb módja a szomszédossági mátrix, érdemes kipróbálnunk, hogy a szomszédossági mátrix tömörítésével milyen előrejelzéseket kaphatunk. Mivel a szomszédossági mátrix valójában bitek egy egyértelműen dekódolható sorozata, tömörítésünkhöz felhasználhatjuk mindazt a tudást, ami az évek során bitsorozatok témájában összegyűlt.

A kutatás során csak egyszerű irányítatlan gráfokkal foglalkoztam, így a teljes mátrix helyett elegendő csupán a főátló alatti (vagy feletti) részeket vizsgálnunk. Ezekből a gráf már egyértelműen visszafejthető, ráadásul redundanciát sem tartalmaznak. Ezen biteket egy tetszőleges, előre meghatározott sorrend szerint egymás után írva n csúcsú gráf esetén kapunk egy $\frac{n \cdot (n-1)}{2}$ hosszúságú bitsorozatot, amivel a gráfunkat reprezentálhatjuk.

A vizsgálatok során ezt a bitsorozatot írtam ki egy szövegfájlba, és ezt a fájlt tömörítettem ZIP kiterjesztésű archívumba. Az így kapott tömörített fájl méretét tekintettem a gráf tömörített méretének, és ebből próbáltam becslést adni a gráfon futtatott algoritmikus probléma bonyolultságára. Ettől a becsléstől jó eredményeket várhatunk annak ellenére, hogy nem gráfok tömörítésére fejlesztették ki, hiszen sok jellegzetes részstruktúrához (például egy teljes gráfhoz) tartozik szabályosságot mutató, így jól tömöríthető rész a szomszédossági mátrixban.

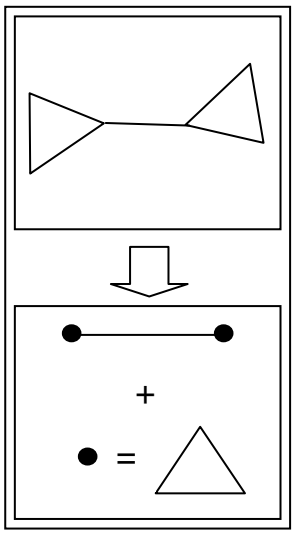
Érdemes megjegyeznünk, hogy a hasonló tömörítéseknek természetesen rengeteg formája ismert és használatos manapság. Ezek közül a ZIP népszerűsége és könnyű hozzáférhetősége miatt került kiválasztásra.

3.2. Subdue

A második tömörítési módszer a Subdue rendszer [6, 21] segítségével történt. A Subdue egy olyan program, amelynek alapvető funkciója gráfokban ismételten előforduló részgráfok hatékony megkeresése (hasonló tehát a zip-hez, ami ismétlődő részsorozatokat keres). Ezt a nekünk

szükségesnél némileg komplexebb szinten teszi: mind a csúcsokhoz, mind az élekhez megadhatók típusok, és két részgráfot csak akkor tekint azonosnak, ha az egyes csúcsok, illetve élek izomorf párjának típusa is megegyezik az eredetiével. Mi erről a képességről nem veszünk tudomást, az összes csúcsot, illetve élt ugyanolyan típusúnak tekintjük.

A tömörítést a Subdue úgy végzi, hogy a gráf egy (a tömörítés szempontjából optimális) részstruktúráját csúcsnak tekinti, és csupán egyszer írja le annak minden előfordulása helyett. Ezt a folyamatot illusztrálja az 1. ábra: az ezen látható hatpontú gráfot a Subdue tömörítheti például úgy, hogy legjobb részstruktúraként megtalálja a hárompontú teljes gráfot, majd a gráfot úgy írja le, hogy leírja az egészet a választott részstruktúra pontként való használatával (két ilyen részstruktúra egy éllel összekötve), valamint leírja a részstruktúrát is. Ezt a folyamatot akár több körön keresztül is folytatja, tehát előfordulhat, hogy például a harmadik iteráció során talált legjobb részstruktúra vegyesen tartalmaz csúcsokat, első iteráció során talált legjobb részstruktúrákat és második iteráció során talált részstruktúrákat, a teljes gráf leírása pedig a harmadik iteráció után ez előbbi három mellett a harmadik iteráció legjobb részstruktúrájából is tartalmazhat valamennyit. A program a részstruktúrák többszöri leírásának kikerülésével tehát méretet spórol, így tömörít gráfokat.

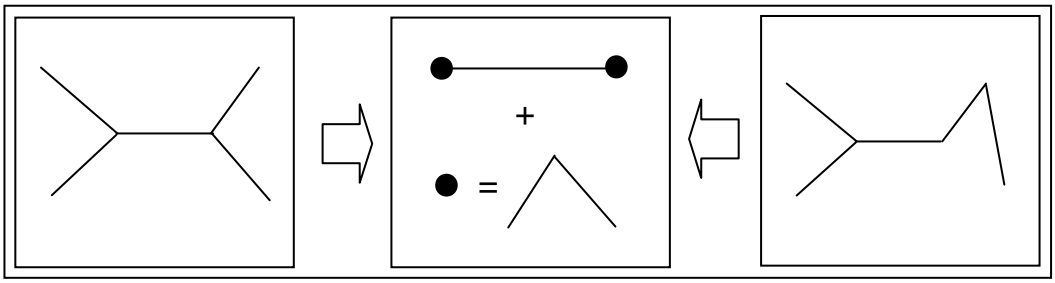


1. ábra: Subdue tömörítés illusztrációja

Sajnos a tömörítés eredményeként kapott gráf nehezen kezelhető, mivel különböző tulajdonságú csúcsok (csúcsok és részstruktúrák) vannak benne; nem egyértelmű, hogy hogyan állapítsuk meg a méretét. A Subdue tartalmaz egy beépített megoldást erre a problémára: kiszámítja az általa tömörített gráfok méretét a (következő részben tárgyalt) MDL segítségével. A kutatás során ezt a számot tekintettem a gráf Subdue-val tömörített méretének.

Fontos megjegyeznünk, hogy a többi tárgyalt eljárással szemben a Subdue által végzett tömörítés veszteséges, nem egyértelműen dekódolható. Ez annak köszönhető, hogy a kiválasztott részstruktúrákat a továbbiakban a program egy csúcsnak tekinti, így ha azon belül a csúcsok helyzete nem szimmetrikus, akkor nem tudhatjuk, hogy a részstruktúrába befutó él melyik csúcsban ér véget. Így megtörténhet, hogy különböző kiindulási gráfok esetén ugyanazt kapjuk tömörítési eredményként.

Ezt szemlélteti a 2. ábra. A Subdue a két oldalt lévő gráfot a „két él közös végponttal” részstruktúra megtalálásával ugyanabba a gráfba tömöríti, holott azok korántsem voltak izomorfak. (Valójában ez a jelenség a részstruktúra-választás módja miatt ezen a példán nem, csak nagyobb gráfok esetén következik be, azonban az ábra jól szemlélteti a jelenséget.)



2. ábra: A Subdue tömörítés veszteségessége

Amennyiben tömörítési képesség szempontjából akarnánk összehasonlítani a vizsgált eljárásokat, akkor ez persze problémát jelentene, célunk azonban a strukturális információ mennyiségét kifejező mérőszám keresése, ami jól előrejelzi a gráfon futó probléma bonyolultságát. Erre pedig a program megfelelő lehet; találhatjuk úgy, hogy a Subdue-val tömörített méret igen jól korrelál a vizsgált probléma nehézségével, annak ellenére, hogy a tömörítés veszteséges.

A kutatás mérései a Subdue 5.2.1-es verziójának Windows-kompatibilisre átírt, minimálisan módosított változatával (csupán felhasználót érintő átalakítások, például futásidő mérése) lettek végrehajtva.

3.3. Az MDL

Adatok tömörítésekor feltétlen meg kell említenünk a Minimum Description Length Principle-t [22], vagy röviden MDL-t.

Az MDL az adatleírás kérdésének formális, matematikai megközelítése; az úgynevezett modellválasztás problémájára a közelmúltban kifejlesztett módszer. A modellválasztás problémája alatt egy adathalmaz különböző leírásai közötti döntés feladatát értjük úgy, hogy ehhez csak korlátozott számú lépés áll rendelkezésünkre. Az MDL által erre adott megoldás azon a gondolaton alapszik, hogy egy adott adathalmaz bármely szabályossága felhasználható az adatok tömörítésére, (vagyis az eredetinél kevesebb szimbólummal való leírására) és minél több szabályosságot találunk, annál jobban tudunk tömöríteni.

Ezen gondolat formalizálásához szükségünk van egy leírási módszerre, egy formális nyelvre, amin az adat tulajdonságait meghatározzuk. Ehhez tetszőleges (persze Turing-teljes) nyelvet választhatunk, belátható, hogy az így kapott értékek egymástól csak additív konstansban térnek el. A nyelv birtokában már definiálhatjuk az adatsor úgynevezett Kolmogorov-bonyolultságát; ez a legrövidebb olyan program hossza, amely a szóban forgó adatsort a kimenetre írja. A Kolmogorov-bonyolultság már megoldás a formális problémánkra, csupán a gyakorlatban kevéssé használhatjuk,

ugyanis véges idő alatt bizonyíthatóan nem kiszámítható [23].

Az MDL az imént definiált fogalom és a megvalósíthatóság közötti kompromisszum. Ehhez a megengedett kódokat lekorlátozzuk úgy, hogy a legrövidebb kódhossz már számítható legyen, de még elég általános feltételeink legyenek ahhoz, hogy viszonylag sokféle szabályosság felismerésére képesek legyünk. Az így megmaradt leírások közül választjuk ki a lehetséges legjobbat, és ezt alkalmazzuk az adatok tömöríteni kívánt halmazán.

Látható, hogy az MDL-számítás önmagában is egy összetett probléma. Szerencsére a Subdue rendszer tartalmaz egy beépített MDL-számítót, amit így a kutatásban fel is használunk, mint a harmadik tömörítési eljárás által adott tömörített méret, amelyről el akarjuk dönteni, hogy milyen összefüggésben van az adatsoron futó probléma bonyolultságával.

3.4. Graph Structure Analyzer

3.4.1. Gráfok leírása

A negyedik tömörítési eljárás egy általam írt program, amely a genetikus algoritmusok működési elve szerint keresi gráfok minél rövidebb leírását. Az algoritmus a BCAT rendszerben lett implementálva C++ nyelven, és a Graph Structure Analyzer, vagy röviden GSA nevet viseli.

A GSA célja a vizsgált gráf leírása úgy, hogy a gráfot különböző részegységek összességeként építi föl. Ezeket a részegységeket *objektumnak* neveztem. A lehetséges objektumok kiválasztásánál a fő szerepet az játszotta, hogy olyan részegységek legyenek az építőkockák, amelyek algoritmikus problémák bonyolultságának meghatározásában gyakran fontos szerepet játszanak. A (kutatás által többé-kevésbé igazolt) feltételezés ugyanis az volt, hogy egy ilyen tömörítés esetén a tömörített méret jobban előrejelzi az adott gráfon futó algoritmikus problémák bonyolultságát, mint egyéb tömörítési eljárások.

A programban öt különböző objektumtípus van implementálva, amelyekből a gráf felépíthető:

- *Teljes gráf*, amely a gráfelméletben pontok olyan halmazát jelenti, amelyekből bármely kettő között fut él. Valójában itt nem teljesen erről van szó; adott pontok egy halmaza, és ezen felül néhány *kivételél*, amelyek nem illeszkednek a mintába. Az adott pontok közül bármelyik kettő között fut él, kivéve azokat, amelyek egy kivételél két végén helyezkednek el. A módosítás azért szükséges, mert általában igen ritkán találunk nagyobb méretű teljes

részgráfokat, ám olyanokat, amelyeket úgy kapunk, hogy teljes részgráfokból elhagyunk néhány élt, már lényegesen gyakrabban, és az ilyen struktúrák szintén fontosak számunkra. Teljes gráf típusú objektumnak nevezünk tehát egy $G(V,E)$ gráfot és élek egy K halmazát, amelyre $\{v_1,v_2\}\in E \Leftrightarrow \neg\{v_1,v_2\}\in K$.

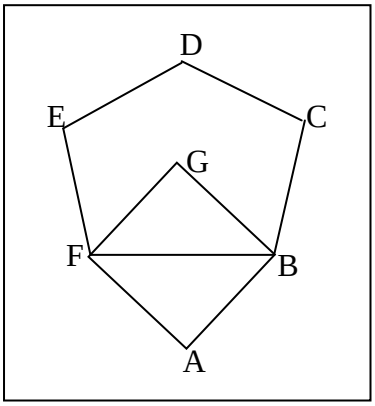
- *Üres gráf*, amely (hasonlóan az előzőhöz) általában olyan ponthalmazt jelent, amelyből semelyik két csúcs között sincs él. Akárcsak az előbb, itt is egy üres gráf típusú objektumon pontok egy halmazát és kivételélek egy halmazát értjük. Az adott pontokból csak azon kettes kombinációk között fut él, amelyeket kivételél köt össze. Formálisan tehát olyan $G(V,E)$ gráf és élek K halmaza, amire $\{v_1,v_2\}\in E \Leftrightarrow \{v_1,v_2\}\in K$.
- *Út*, amely alatt az út gráfelméleti definíciójának megfelelő részgráfot értünk.
- *Káosz*, amely alatt pontok olyan halmazát értjük, amelyet a szomszédossági mátrixukkal adunk meg. A pontok között futó élekre tehát nincs semmilyen megkötés, csupán a kapcsolatuk leírásának módjára. Az objektum bevezetését az motiválta, hogy kezelni tudjunk olyan kvázi-diszjunkt részegységeket, amelyek az előző kategóriák mindegyikétől távol állnak, tehát túl sok élük van ahhoz, hogy hatékony legyen őket üres gráf és kivételélek összességeként megadni, de túl kevés ahhoz, hogy teljes gráf és kivételélek összességeként legyen érdemes. Ha például van néhány pont, amelyek között körülbelül a lehetséges élek fele van behúzva, viszont a ponthalmaz külső pontokkal csak kevés él mentén érintkezik, akkor a ponthalmaz pontjai között futó éleket (*belső éleket*) szomszédossági mátrixszal érdemes megadni, viszont a halmazban lévő és külső pontok között futó éleket kis számuk miatt különálló élként érdemes megjegyeznünk.
- *Különálló csúcs* vagy *független csúcs*, egész egyszerűen egy csúcsot jelöl.

Az objektumokon kívül megengedünk még *további éleket* (vagy *különálló éleket*), amelyek valamely objektum egyik pontja és egy (általában) másik objektum valamely pontja között futnak, és külön szükséges megemlítenünk őket, mivel semelyik objektum nem tartalmazza azokat. Az ilyen éleket tehát a két végpontjuk objektumának sorszámával, illetve a kérdéses csúcs objektumon belüli sorszámával azonosítjuk.

Végül a gráfleírás tartalmaz *átfedéseket* (vagy *overlapeket*), amelyek azt fejezik ki, hogy adott objektumok adott pontjai valójában a gráfnak ugyanazt a csúcsát jelölik. Ez azért szükséges, mert sokkal kevésbé tudnánk tömöríteni, ha a gráfot az előző objektumok diszjunkt uniójára akarnánk bontani. Overlapek segítségével a több objektumban is jelen lévő pontot úgy kezelhetjük, mintha valójában több pont lenne, s csak a leírás dekódolásakor szükséges újra figyelembe vennünk, hogy a

pont egyes „példányairól” szóló tulajdonságok valójában egyazon pontra vonatkoznak.

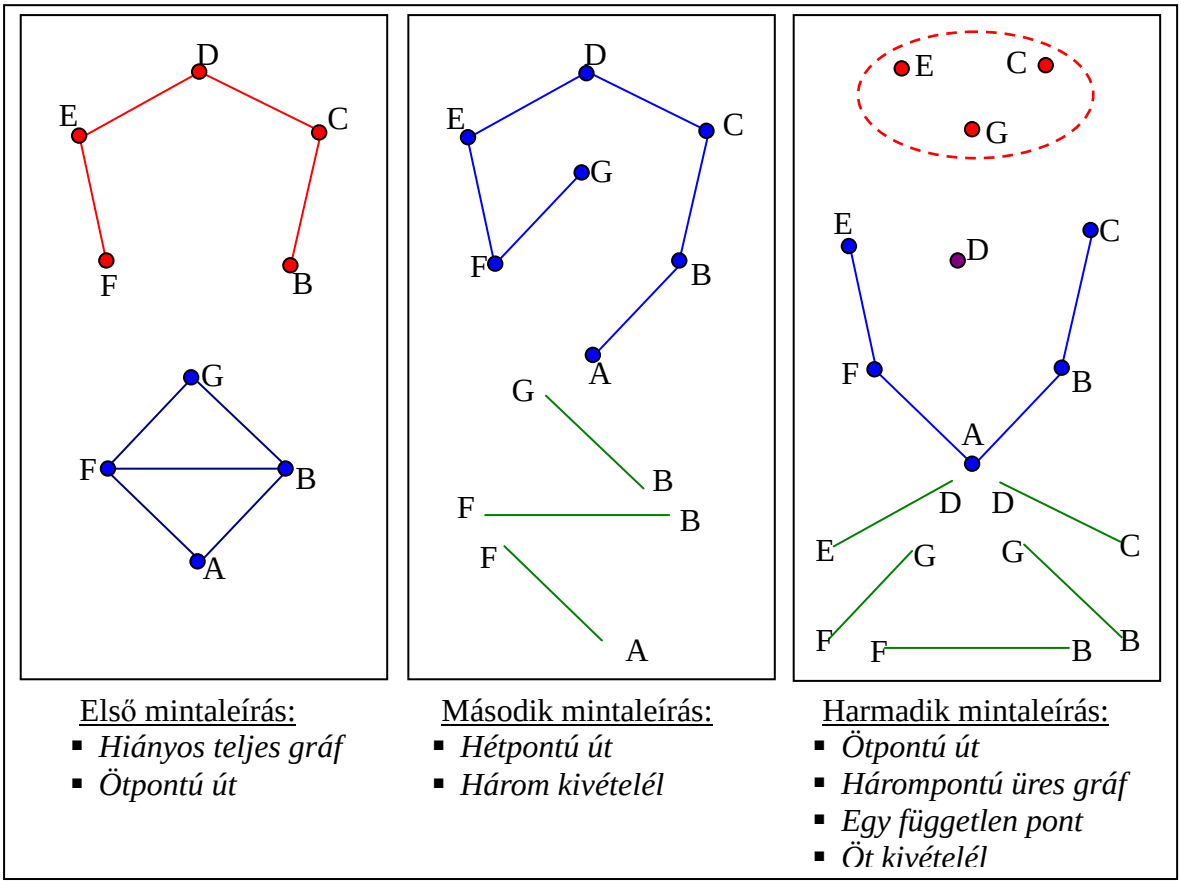
A *gráf leírásán* tehát objektumok, különálló élek és átfedések olyan halmazát értjük, melyek összessége a gráf összes pontját és élét előállítja, azonban nem tartalmaz olyan pontot vagy élt, amely a gráfban nem található meg. A leírásból a gráf (izomorfia szintjén) egyértelműen dekódolható. A fenti objektumok jellegzetes szerkezete miatt egy ilyen leírás általában sokkal szemléletesebb, mint a gráf szomszédossági mátrixa, bár a tömörített méret várható értéke is nagyobb, mint a szomszédossági mátrix mérete; problémabonyolultság előrejelzéséhez azonban a többi tömörítési eljárásnál akár sokkal eredményesebb lehet.



3. ábra: Kisméretű gráf a leírások illusztrációjára

Példaként nézzük meg a 3. ábrán látható hétpontú gráf néhány különböző leírását:

- Egy négypontú teljes gráf (*A*, *G*, *B*, *F*), annak első és második pontja között egy kivételéllel, egy ötpontú út (*B*, *C*, *D*, *E*, *F*), valamint két átfedés: az első objektum harmadik pontja a második objektum első pontjával, míg az első objektum negyedik pontja a második objektum ötödik pontjával egyezik meg.



4. ábra: A hétpontú példagráf három lehetséges leírásának felépítése

- Egy hétpontú út (*A*, *B*, *C*, *D*, *E*, *F*, *G*) és három további él (*AF*, *FB*, *GB*).
- Egy ötpontú út (*E*, *F*, *A*, *B*, *C*), egy hárompontú üres gráf (*G*, *C*, *E*), egy különálló pont (*D*) és öt további él (*FB*, *FG*, *GB*, *ED*, *EC*) az objektumok megfelelő csúcsai között. Továbbá lesz két átfedés, az üres gráf és az út két megfelelő csúcspárja (*E* és *C* pontok példányai) közt.

Érezhetjük, hogy ez utóbbi sokkal kevésbé szemléletes és egyben lényegesen bonyolultabb megoldás; ennek az érzésnek a formalizálása és számértékekre való lefordítása a később részletesen tárgyalt fitness-függvény feladata lesz.

3.4.2. A genetikus algoritmusokról

A genetikus algoritmus egy heurisztika, amelyet széles körben használnak keresési és optimalizációs problémák megoldására, a számítástudomány és a matematika területén kívül például a bioinformatikában, kémiában vagy a közgazdaságtanban is. Nevét onnan kapta, hogy alapötletét az evolúció működéséből meríti; futása során megoldásjelöltek halmaza fejlődik egy jobb megoldás felé [24, 25].

Adott a működés során egy, az optimalizációs probléma megoldásainak jelöltjeiből (*egyedekből*) álló halmaz (*populáció*), amit a futás kezdetén valamilyen véletlen módszerrel létrehozunk. Általában ennek a fejlődését vizsgáljuk valahány fázison (*generáción*) keresztül. A problémához szintén adottnak tételezünk fel egy függvényt (*fitness függvény*), amely minden egyedhez hozzárendeli annak a mérőszámát (*fitness*), hogy a szóban forgó megoldási jelölt „mennyire jó megoldás”, mennyire áll közel a célként kitűzött optimumhoz. Mindegy egyes generációban kiértékelésre kerül az összes egyed fitness értéke, majd ezen értékek alapján sztochasztikusan kiválasztásra kerül valamennyi, és ezek módosított formája alkotja a következő generáció populációját (*reprodukció*). A módosításnak alapvetően két jellemző típusa van, amikor egyetlen egyed módosításával hozzuk létre a következő generáció egyik egyedét (*mutáció*), illetve amikor több egyed felhasználásával alkotjuk meg azt (*rekombináció*). Az algoritmus általában akkor hagyja abba futását, ha egy adott generációszámot elérünk, vagy egy megfelelő fitness értékű egyed jön létre.

3.4.3. A leírások evolúciója

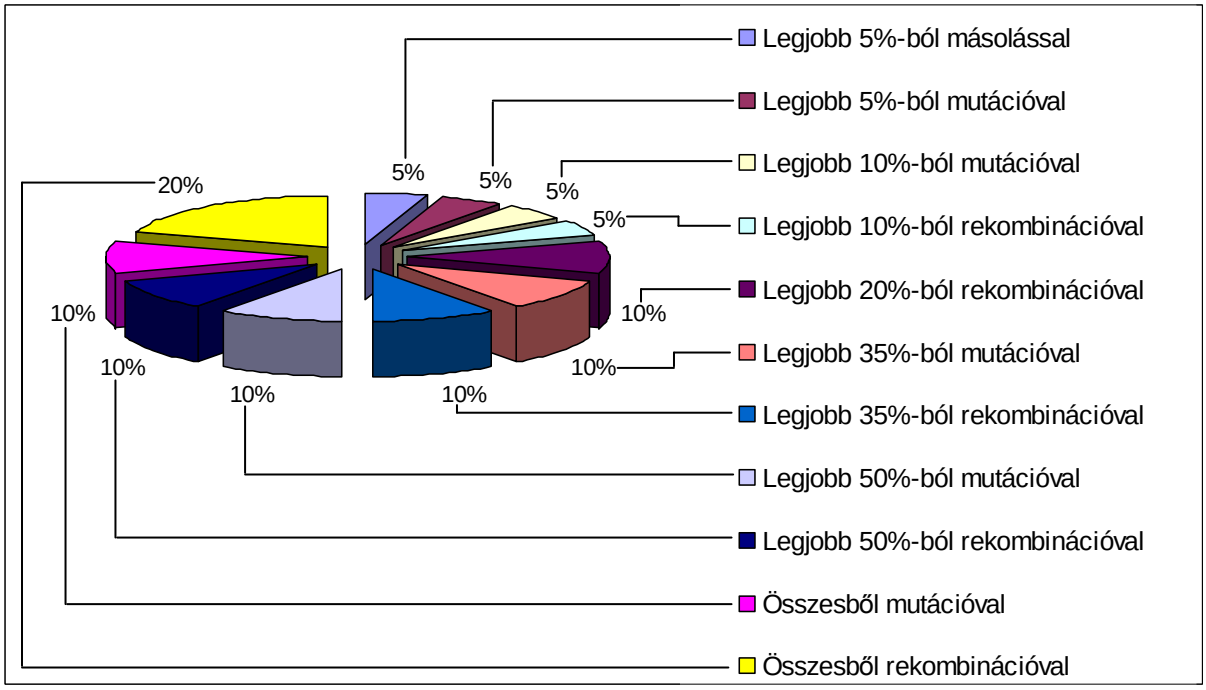
A GSA program egy genetikus algoritmus, ahol a populáció egyedeit a fenti leírások alkotják. A futás során ezek a leírások fejlődnek és változnak, egy, az optimálishoz minél közelebbi leírás megtalálásának reményében. A program fő paraméterei (generációk száma, populáció mérete,

időlimit) a BCAT konfigurációs fájljából beállíthatók. Szintén állítható a lefuttatások száma is, ami (mint korábban már említettük) a sztochasztikus folyamatok szórását hivatott csökkenteni.

Mivel a leírásokban semmilyen kapcsolat nincs a gráf különböző komponensei között, a GSA első dolga a gráfot komponenseire bontani, majd a komponensek mindegyikére meghívni a genetikus algoritmust. Ez hatékonyabbá teszi a működést, mivel a leírási rendszerünkben különböző komponensekben lévő pontok közötti mindenféle kapcsolat vizsgálata felesleges időtöltés volna.

Természetesen szükségünk van a genetikus algoritmusok összes problémaspecifikus feladatának specializációjára. Ezek közül a legfontosabb egy fitness függvény létrehozása, amely megállapítja, hogy az adott egyed (leírás) mennyire tömör, így mennyire jó számunkra. Ezt követően az evolúció kezdeti lépéseként létre kell hoznunk egyedek egy kezdeti sorozatát, amelyek majd fejlődni fognak. Végül pedig a reprodukció folyamatát kell specializálnunk.

Mivel jelen problémánk esetén tapasztalataim szerint az egyedek fitness értékeinek aránya és különbsége problémánként igen eltérő értéket mutat, a reprodukciós valószínűséget (a megoldási javaslatok továbböröklődésének esélyét) nem a fitness arányában, csupán a fitness értékek szerinti rendezés alapján határoztam meg. A GSA-ban a reprodukció folyamata több részből áll össze; ezek során a következő generációba hozunk létre egyedeket az előző generáció fitness szerint rendezett egyedeinek egy részéből másolással, mutációval vagy rekombinációval:



5. ábra: Az egyedek százalékos megoszlása új generáció létrehozásakor

Az ábrázolt esetekben a megadott halmazból az egyedek ugyan ugyanakkora valószínűséggel kerülnek kiválasztásra, mégis, mivel a jobb fitness értékű egyedek több halmazban is jelen vannak, az ő tulajdonságaik nagyobb eséllyel örökítődnek tovább a következő generációba.

3.4.4. A fitness függvény

A fitness függvény feladata megállapítani, hogy az adott egyed mennyire jó megoldási kísérlet. Ez helyettesíti a természetbeli evolúcióban előforduló erőforrásokért való versengést, „természetes szelekciót”. Mivel célunk a legrövidebb leírás megtalálása, a függvény azt adja vissza, hogy az adott leírás hány bitet foglal. Ez persze azt eredményezi, hogy a kisebb fitness érték fog fittebb, tömörebb, „életképesebb” leírást jelenteni, és így nagyobb valószínűséggel reprodukálódni.

A függvényhez létre kell hoznunk egy konvenciót, hogy egy egyedet hogyan írunk le, alakítunk egyértelműen dekódolható bináris sorozattá; ez az alábbi táblázatban látható. A bal oldalon különválasztott, értéket tartalmazó cellák egy ciklust jelölnek; például a teljes gráfok *t* száma után sorban mind a *t* darab teljes gráfra először megadjuk a gráf méretét, majd a benne rejlő kivételélek *k* számát, ezt pedig az objektum *k* darab kivételéle követi.

A táblázatban felhasznált *o* érték az objektumok száma, vagyis a teljes gráfok, üres gráfok, utak, káoszok és különálló pontok darabszámának összege.

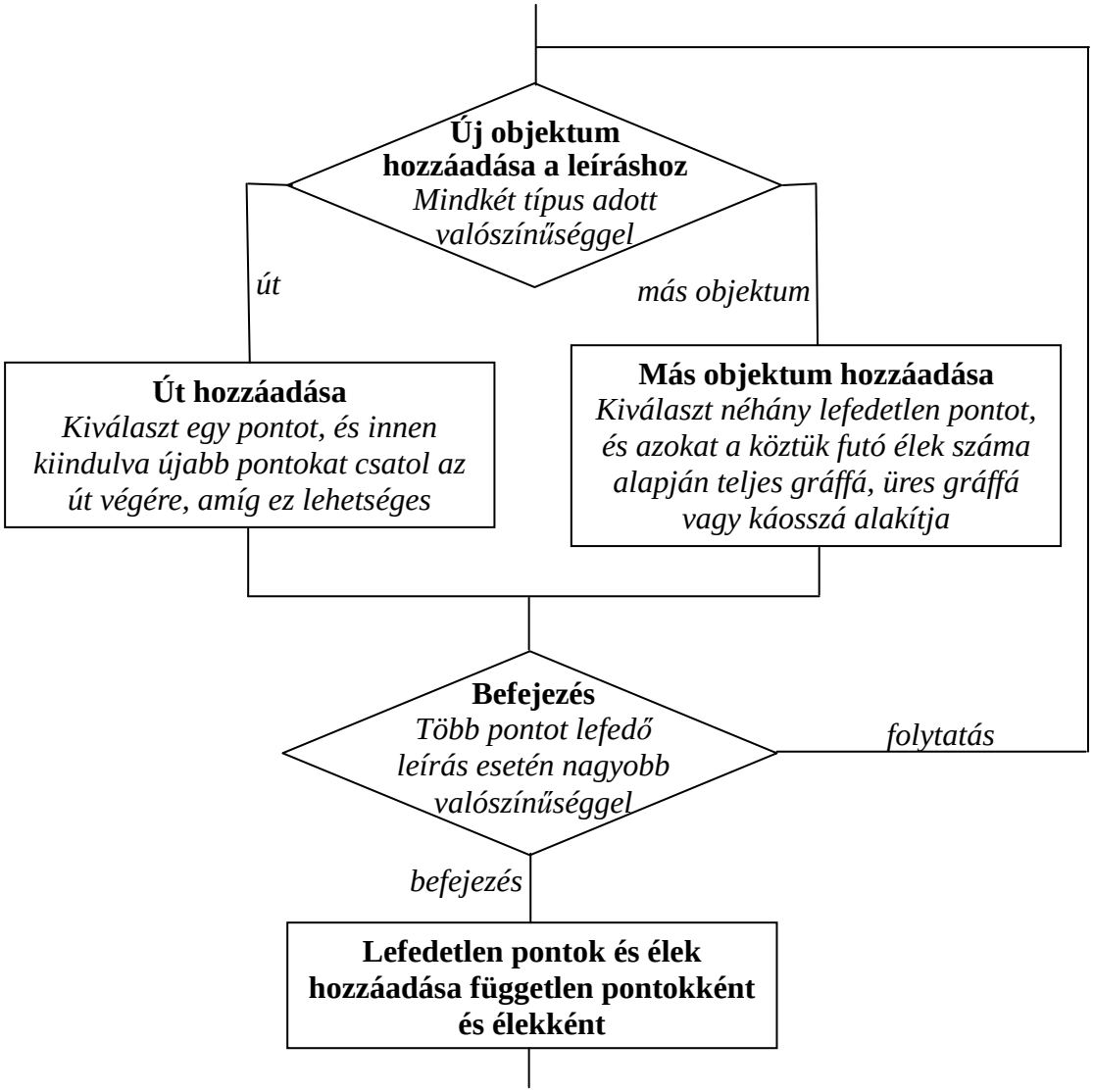
Jól megfigyelhető az egyértelmű dekódolhatóság, hiszen az elfoglalt bitek száma minden esetben egy már korábban megállapított paramétertől függ, vagyis minden adatmező kiolvasásakor már pontosan tudjuk, hogy az adott mező a következő hány bitet foglalja el. A bitsorozatból tehát egyértelműen képezhető az adott egyed, amely pedig (izomorfia szintjén) már egyértelműen meghatározza a vizsgált gráfot, így a tömörítés veszteségmentes.

Megnevezés			**Bitek száma**	**Megjegyzés**
A gráf csúcsainak száma (*n*)			12	Hogy ez elfér 12 biten, az csupán egy feltételezés, mely szükség esetén könnyen módosítható.
Teljes gráfok száma (*t*)			$\log_2 n$	A teljes gráf típusú objektumok száma.
t	Méret (*s*)		$\log_2 n$	Az adott teljes gráf csúcsainak száma.
	Kivételélek száma (*k*)		$\log_2\left(\frac{s\cdot(s-1)}{2}\right)$	A teljes gráf kivételéleinek száma.
	k	Kivételél két csúcsa	$2\cdot\log_2 s$	A teljes gráf azon két csúcsának sorszáma, amelyek között a kivételél fut.
Üres gráfok száma (*ü*)			$\log_2 n$	Az üres gráf típusú objektumok száma.
ü	Méret (*s*)		$\log_2 n$	Az adott üres gráf csúcsainak száma.
	Kivételélek száma (*k*)		$\log_2\left(\frac{s\cdot(s-1)}{2}\right)$	Az üres gráf kivételéleinek száma.
	k	Kivételél két csúcsa	$2\cdot\log_2 s$	Az üres gráf azon két csúcsának sorszáma, amelyek között a kivételél fut.
Utak száma (*u*)			$\log_2 n$	Az út típusú objektumok száma.
u	Méret		$\log_2 n$	Az út csúcsainak száma, avagy a hossza+1.
Káoszok száma (*c*)			$\log_2 n$	A káosz típusú objektumok száma.
c	Káosz mérete (*s*)		$\log_2 n$	Az adott káosz csúcsainak száma.
	Belső szerkezet		$\frac{s\cdot(s-1)}{2}$	Az adott pontok által kifeszített részgráf szomszédossági mátrixa.
Csúcsok száma			$\log_2 n$	Különálló pont típusú objektumok száma.
Élek száma (*e*)			$\log_2\left(\frac{n\cdot(n-1)}{2}\right)$	Az objektumok által le nem fedett élek száma.
e	Objektumok száma		$2\cdot\log_2 o$	Azon objektumok sorszáma, amelyek valamely csúcsában az adott él két vége végződik.
	Csúcsok száma		$2\cdot(\log_2 s$ *vagy 0)*	Az adott objektumok azon csúcsának sorszáma, amelyre az él illeszkedik. Amennyiben az objektum különálló pont típusú, akkor ez az adat felesleges, így kimarad.
Átfedések száma (*á*)			$\log_2 n$	A több helyen is előforduló („átfedett”) pontok száma.
á	Méret (*s*)		$\log_2 o$	Hány különböző objektumban fordul elő az adott csúcs.
	s	Objektum	$\log_2 o$	Az adott előfordulás melyik objektumban található.
		Csúcs	$\log_2 s$	Az előfordulásban a csúcs az objektum hányadik pontja.

3.4.5. A kezdeti populáció létrehozása

Bár általában nem szokott különösebb problémát jelenteni, itt a kezdeti populáció létrehozása volt az egyik legnagyobb kihívás. Egyedek olyan halmazát kellene létrehoznunk, amelyek többé-kevésbé jó fitness értékeket produkálnak; ha egy jó heurisztikával viszonylag fitt nulladik generációt kapunk, akkor rengeteg felesleges nemzedéket spórolhatunk meg, jelentősen javítva így algoritmusunk hatékonyságát.

Sajnos már az sem magától értetődő, hogy hogyan hozunk létre a semmiből nemtriviális (vagyis nem szinte csak pontokat és éleket vagy káoszt tartalmazó) gráfleírást. Mivel az objektumok méretnövekedése (tehát például nagyobb teljes gráf létrejötte egy kisebből) a gyakorlati tapasztalatok szerint igen megbízható folyamatnak tűnt, a kezdeti populáció kreálásánál végül sok kisméretű diszjunkt objektum létrehozása mellett döntöttem. Az erre szánt algoritmus egyszerűsített leírását folyamatábrán tudjuk jól illusztrálni.



6. ábra: A kezdeti populáció létrehozás leegyszerűsített folyamatábrája

3.4.6. Mutáció

Az evolúciós algoritmusok kulcsfontosságú motívuma a mutáció, mely során a következő generáció egyik egyedét hozzuk létre a jelenlegi generáció egyetlen egyedéből. Ez fenntartja a populáció változatosságát, és új megoldási lehetőségek felmerüléséhez vezet.

Az algoritmus hatékonyságának érdekében általában intelligens mutációt alkalmazunk. Az új egyedet nem teljesen véletlen változtatás eredményeként hozzuk létre, hanem a mutáció során viszonylag gyors heurisztikus módszerekkel megpróbáljuk kiszűrni az eleve halálra ítélt, rosszabb fitness értékű egyedek létrejöttét, és feltehetően életképesebb egyedet nagyobb valószínűséggel hozunk világra. Ezzel drasztikusan javítható az algoritmusunk hatékonysága, hiszen az életképtelen egyedek felvétele helyett a program már sok generációval hamarabb megtalálja a heurisztikák segítségével a (valószínűleg) jobb fitness-szel rendelkezőket.

A GSA a gráfleírások mutációinak sokféle verzióját ismeri, melyek alapvetően hat fő kategóriába vannak besorolva. Mutációnál e hat kategória valamelyike hívódik meg az egyedre (mindegyik egy adott valószínűséggel), majd az ezen belüli mutációk egyike lejátszódik rajta. Az egyes kategóriák meghívási valószínűsége, illetve az ezen belüli mutációk valószínűsége lényegesen befolyásolja az algoritmus eredményességét; optimális értékük problématípusonként, akár problémánként eltérő lehet. Ezek a valószínűségek a BCAT rendszer konfigurációs fájljában, az algoritmus paramétereként megadhatók. Az egyes kategóriák és mutációk a következők.

1. Objektum létrehozása:

- Egy élt átalakít kétpontú úttá. Olyan élt, amelynek egyik vége nem komplex objektumban, hanem pontban végződik, nagyobb eséllyel választ, két pont közötti élt még nagyobbal.
- Egy élt átalakít kétpontú teljes gráffá. Olyan élt, aminek egyik vége pontban végződik, nagyobb eséllyel választ, két pont között húzódó élt még nagyobbal.
- Két élt – melyek egyik végpontja közös – hárompontú úttá alakít. Különálló pontokat nagyobb eséllyel jelöl ki, mint összetett objektum által lefedett csúcsokat.
- Két élből – melyek egyik végpontja közös – (esetleg hiányos) hárompontú teljes gráfot hoz létre. Különálló pontokat nagyobb eséllyel választ ki.
- Négy különálló csúcsot egy új objektumba rak. A négy pont között húzódó belső élek száma határozza meg az objektum típusát. Kevés belső él esetén új üres gráfot, közepes mennyiségnél új káoszt, sok él esetében teljes gráfot kapunk.

2. Objektum törlése:

- Egy legfeljebb négy csúcsból álló objektumot különálló pontokká és élekké bont szét. Az objektumokat annál nagyobb valószínűséggel választja, minél több alkalommal (minél több különböző objektumban) vannak lefedve az objektum csúcsai.
- Végignézi az objektumokat, és valamekkora valószínűséggel töröl közülük minden olyat, amely pontjainak legalább 70%-át más objektum is lefedi. Az objektum lefedetlen pontjait és éleit különálló pontokkal és élekkel helyettesíti.

3. Objektum méretének változtatása:

- Egy teljes gráfhoz egy újabb pontot csatol. Minél több pontjával van összekötve egy csúcs a teljes gráfnak, annál nagyobb eséllyel kerül kiválasztásra.
- Egy üres gráfhoz egy újabb pontot csatol. Minél kevesebb pontjával van összekötve egy csúcs az üres gráfnak, annál nagyobb eséllyel választja.
- Egy út valamelyik végére csatol még egy csúcsot, amit az aktuális végcsúcshoz él köt.
- Egy káoszhoz csatol egy újabb csúcsot. Minél több pontjához kapcsolódik egy csúcs a káosznak, annál nagyobb valószínűséggel dönt mellette.
- Egy teljes gráfnak leveszi egy pontját. Minél kisebb egy pont belső fokszáma, annál nagyobb valószínűséggel választja.
- Egy üres gráfnak leveszi egy pontját. Minél nagyobb egy pont belső fokszáma, annál nagyobb valószínűséggel választja.
- Egy útnak leveszi valamelyik végpontját.
- Egy káosznak leveszi egy pontját. Minél kisebb a csúcs belső fokszáma, annál nagyobb eséllyel veszi ki a káoszból.

4. Objektum típusának megváltoztatása:

- Választ egy teljes gráfot. Ha éleinek számát egy adott határnál alacsonyabbnak találja, akkor a teljes gráfot káosszá alakítja át.
- Választ egy üres gráfot. Ha éleinek számát egy adott határnál magasabbnak találja, akkor az üres gráfot káosszá alakítja át.
- Választ egy káoszt. Ha a belső élek számát egy adott határnál magasabbnak találja, akkor teljes gráffá, ha egy másik határnál alacsonyabbnak, akkor üres gráffá alakítja át.

5. Két objektum összeolvasztása:

- Kiválaszt két teljes gráfot, és összeolvasztja őket egyetlenegy teljes gráffá. Minél több közös

csúcsa van két teljes gráfnak és minél több él fut a kettő között, annál nagyobb valószínűséggel választja az adott objektumpárt.

- Kiválaszt két üres gráfot, és összeolvasztja őket egyetlenegy üres gráffá. Minél több közös csúcsa van két üres gráfnak és minél kevesebb él fut közöttük, annál nagyobb valószínűséggel választja a két objektumot.
- Kiválaszt két utat, melyeknek van közös végpontja, és egy úttá egyesíti őket.

6. Egy objektum kettővé alakítása:

- Egy teljes gráfot kettéoszt két teljes gráfra. Ha egy pontot beoszt az egyik gráfba, akkor a vele nem összekötött, még nem beosztott csúcsokat a másik gráfba rakja. Ezzel a heurisztikával próbálja biztosítani, hogy a két kapott gráf között minél kevesebb él fusson.
- Egy üres gráfot két üres gráfra oszt szét. Hasonlóan, ha beoszt egy pontot az egyikbe, akkor a vele összekötött, még nem beosztott csúcsokat a másikba helyezi.
- Egy utat valamely pontjánál megtörve két új útra oszt.
- Teljes gráf néhány pontját káoszként leválasztja. Ehhez először választ egy pontot, (kisebb belső fokszámúakat nagyobb valószínűséggel) majd ezt és a teljes gráf összes pontját, ami nincs összekötve vele, leválasztja káoszként. Ezzel a heurisztikával próbálja elérni, hogy a teljes gráf minél kevesebb élt tartalmazó része alkossa az új káoszt.
- Üres gráf néhány pontját káoszként leválasztja. Ehhez először választ egy pontot, (nagyobb fokszámút nagyobb valószínűséggel) majd ezt és az üres gráf összes ezzel összekötött pontját leválasztja káoszként.
- Üres gráfból leválaszt egy utat. Ehhez először választ egy kiindulási pontot az objektumban (nagyobb fokszámút nagyobb eséllyel), majd a lehetséges folytatási élek közül mindig véletlenszerűen választva bővíti a lecsatolandó utat, amíg ez lehetséges.

3.4.7. Rekombináció

Rekombinációnak azt nevezzük, amikor kettő (vagy néha több) egyedből hozunk létre újabb egyedet; a legtöbb genetikus algoritmus ilyet is tartalmaz. Ez alapvetően arra szolgál, hogy a különböző egyedekben jelenlévő megoldásrészletek megjelenhessenek ugyanabban az egyedben.

A GSA a rekombináció során a két eredeti egyed objektumainak uniójából választ ki egy részhalmazt, ami az új egyed objektumait alkotja, majd ezt egészíti ki a még szükséges pontokkal és élekkel.

Természetesen itt is megpróbálunk egy heurisztikus módszert adni, ami rövid idő alatt is

viszonylag megbízható képet ad arról, hogy mely részhalmazok lehetnek potenciálisan elég jó megoldások, és melyek vezetnek nagyon rossz fitness értékű egyedekhez. Akárcsak a mutáció során, ezt itt is a kiválasztási valószínűségek manipulálásával tesszük. Tegyük fel, hogy néhány objektumot már kiválasztottunk; ekkor a következő kiválasztásánál a valószínűség meghatározásában két tényező játszik közre. Egyrészt az új objektum azon pontjai, amelyeket az eddig kiválasztott objektumok nem fednek le, növelik az új objektum kiválasztásának valószínűségét, hiszen ezzel a leírás több pontot fog lefedni, így teljesebb lesz. Másrészt az új objektum olyan pontjai, amelyeket már az eddig kiválasztottak is lefednek, csökkentik a kiválasztás esélyét, mivel ezek csak egyes csúcsok többszörös leírásához vezetnek, ami a hosszabb leírás miatt rosszabb fitness értéket eredményez.

3.4.8. Működés közben

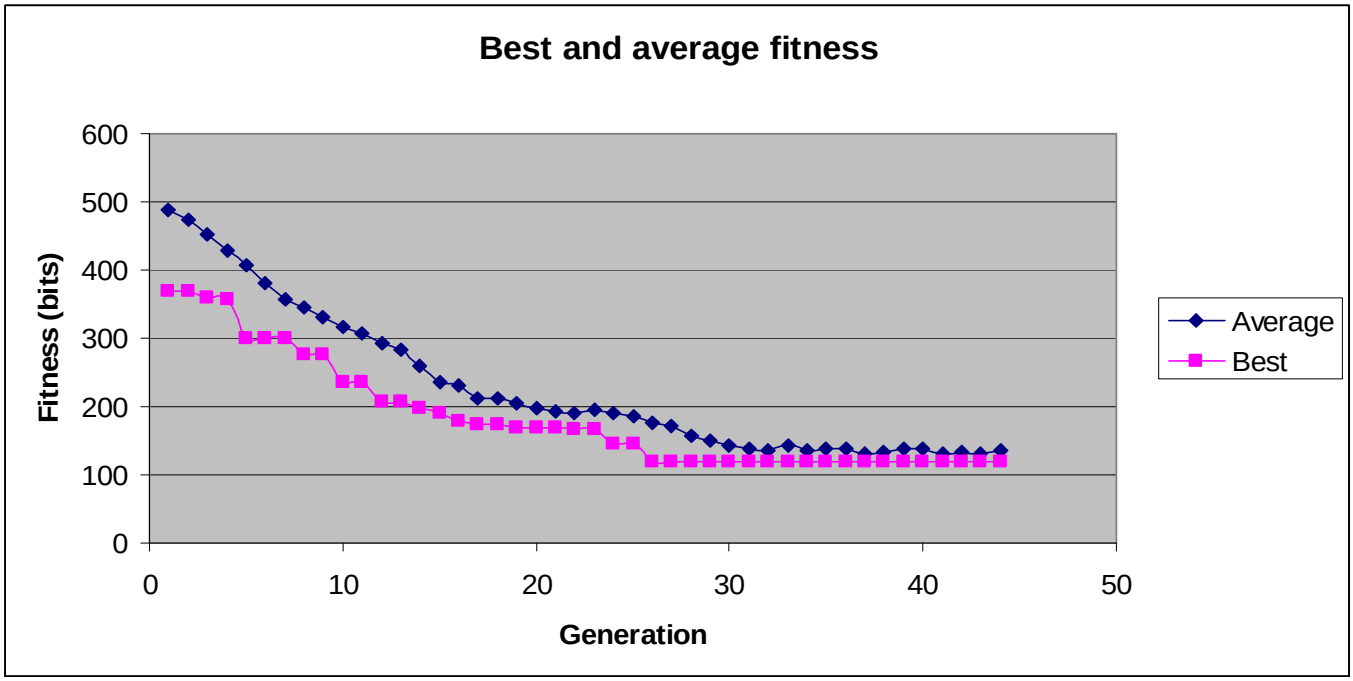
A program az elvártnak megfelelően működik, és általánosságban elmondható, hogy elég jó eredményeket ad. Az első tesztelésekhez olyan gráfok a legcélszerűbbek, melyek elég strukturáltak ahhoz, hogy ránézésre megállapítható legyen a legjobb leírásuk. Az ilyeneken a program megfelelően működött, a nyilvánvalóan legjobb megoldást találta meg.

A különböző mutációs eljárások lefuttatásának valószínűségét be lehet állítani viszonylag jól úgy is, hogy csupán a futási eredményt (a legjobb egyed fitness értékét) nézzük. Ez a tesztelés során meg is történt; az objektumok méretváltoztatására például nagyon nagy, míg a típusváltásra nagyon kicsi valószínűség adódott optimálisnak.

Sokat elárul a folyamatról, ha megfigyeljük, hogy az egyedek fejlődése hogyan zajlik generációról generációra, például ha megnézzük a legjobb egyed fitness értékét a generáció sorszámának függvényében. A program futási paramétereinek finomhangolására, főleg a mutáció-mennyiség (tehát hogy hány elemi mutációs lépéssel alakítsuk ki a kiválasztott egyedből a következő populáció egyik egyedét) helyes mértékének meghatározására is ez a legjobb módszer.

A tesztelés folyamán ekkor derült ki, hogy a mutáció kezdetben beállított mennyisége az optimálisnál sokkal kisebb volt, így a program nagyon könnyen megrekedt a keresési tér lokális minimumaiban, és nagyon kicsi esélye volt kijutni annak környezetéből. Ez nem csak sokkal gyengébb végeredményhez vezetett (mivel egy lokális minimum értéke akár nagyon távol is eshet a globális minimumétól), hanem az eredmények sokkal nagyobb szórásához is. Ez utóbbi annak köszönhető, hogy a kezdeti populáció hatása jelentősen befolyásolta az eredményt: a végkifejlet nagyrészt attól függött, hogy a kezdeti populáció egyedei (az erre szolgáló véletlen folyamaton keresztül) éppen melyik lokális minimum közelében jöttek létre, hiszen miután a populáció egy

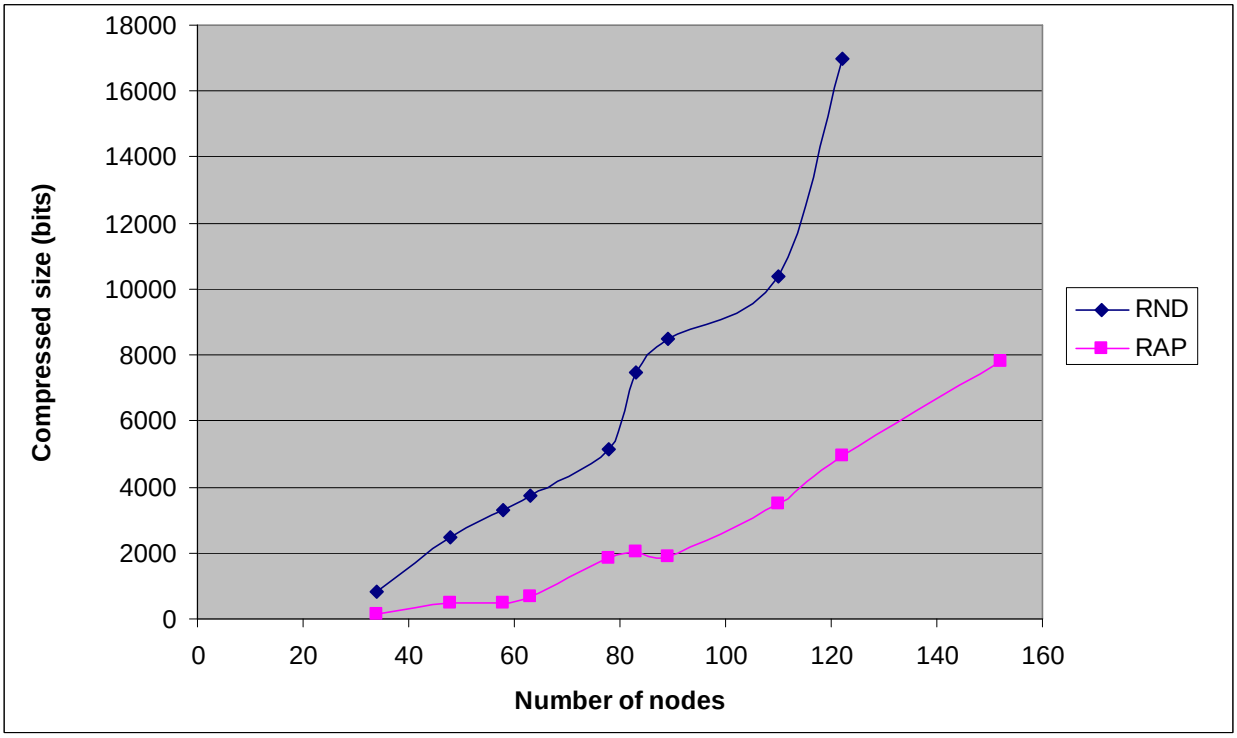
ilyen szélsőérték-pont közelébe került, utána nagyon kis eséllyel vagy csak rengeteg generáció múltán tudott távozni onnan. A mutációs lépésszám megnövelésével jelentős javulást tapasztalhattunk; nemcsak az elért eredmények lettek jobbak, de gyakorlatilag a legjobb megtalálható megoldást is negyedannyi generáció alatt megtalálta. (A „gyakorlatilag a legjobb megtalálható megoldás” alatt azt értjük, hogy bármeddig is futtatjuk a programot, ennél legfeljebb néhány százalékkal jobb megoldást talál.)



7. ábra: A legjobb és az átlagos fitness a generációszám függvényében

A 7. ábrán a generációszám függvényében látjuk a legjobb addig megtalált egyed fitness-ét, illetve az adott generációban a populáció átlagos fitness értékét egy igen strukturált 20 csúcsú gráfon. Az előbbi nyilván monoton csökken; a kisebb csökkenések általában kisebb mutációknak, a nagyobbak egy igen sikeres mutációnak vagy két különböző rész jó leírását tartalmazó egyedek rekombinációjának felel meg. Az átlagfitness nem feltétlenül monoton csökkenő; az ábrán nagyjából követi a legjobb fitness értékét; mikor az beáll a megtalált minimumra, az átlag néhány százalékkal fölötte ingadozik, hisz a mutációk ilyenkor általában rosszabb megoldások felé vezetnek, mivel céljuk a változatosság fenntartása, márpedig egy ilyen populációnál a (nagyrészt jó) megoldásokon való változtatás többnyire rosszabbakat eredményez.

Szintén fontos kérdés, hogy a megírt program mennyire érzékeli egy gráf strukturáltságát. Ehhez természetesen már külön kell megfigyelnünk a vizsgált RAP gráfok, illetve a véletlen példányok tömörített méretét. Az ábrázolt függvényen az adott csúcsszámú gráfokat egy adatpontnak tekintettük, és az ehhez tartozó fitness értéknek a gráfok futtatásakor kapott legjobb fitness értékek átlagát vettük.



8. ábra: RAP és véletlen gráfok GSA-val tömörített mérete a csúcsszám függvényében

A fenti ábrán a legjobb fitnesst (tehát a tömörített méretet) látjuk a csúcsszám függvényében regiszterallokációs, illetve véletlen gráfokra. Megfigyelhetjük, hogy az adott csúcsszámmal bíró regiszterallokációs gráfokhoz tartozó érték lényegesen kisebb az ugyanennyi pontú véletlen gráfokhoz tartozónál, a GSA tehát valóban felismeri gráfok struktúráját; így jó becslést adhat a probléma bonyolultságára.

Eddig csak elméleti megközelítésben vizsgáltuk a GSA-t, ám egy problémák bonyolultságának előrejelzésére szolgáló programmal szemben természetesen a minél gyorsabb futásidő is elvárás; kevés esetben tudunk hasznosan alkalmazni egy olyan, futásidő becslésére szolgáló eljárást, amely maga is igen sok számítási kapacitást igényel.

A GSA természetesen polinomiális időben fut, de jó eredmények eléréséhez sokszor mind a generációk számának (g), mind az egyedek számának (i) nagy értéket kell felvennie, amely nagy csúcsszámmal (n) rendelkező gráfnál már problémás lehet. Mivel egyes ritkán meghívott (kis valószínűségű) mutációs heurisztikák az elméletileg elképzelhető legrosszabb esetben $O(n^4)$ időben futnak, a bonyolultságelmélet klasszikus becslése szerint a probléma $O(g \cdot i \cdot n^4)$ nehézségű. A kutatás során vizsgált gráfokon azonban az volt a tapasztalat, hogy a futásidő körülbelül arányosnak volt tekinthető $g \cdot i \cdot n^2$–tel. A program 20-30 pontú példákon jelenleg néhány másodperc alatt fut le a szükséges generációszámmal, 60-70 pontú gráfokhoz körülbelül egy percre van szüksége.

4. Eredmények

4.1. Függvényábrázolás

A mért eredményeket célszerű lenne valamiképp egy grafikonon szemléltetnünk, ám először végig kell gondolnunk, hogy mit lenne érdemes ábrázolni minek a függvényében.

Egy vizsgált tömörítési eljárást akkor mondhatunk eredményesnek a problémabonyolultság előrejelzésének szempontjából, ha a tömörített méretből jó becsléssel megmondhatjuk a probléma bonyolultságát, mégpedig a probléma típusától függetlenül. Jelen esetben (két problématípussal) ez azt jelenti, hogy a *k* bitre tömörített strukturált gráfon futó probléma bonyolultságának minél közelebb kell lennie a *k* bitre tömörített strukturálatlan gráfhoz tartozó problémáéhoz. Vagy másképp, hogy egy adott bonyolultságú strukturált problémát körülbelül ugyanakkorára tömörít, mint egy ugyanilyen bonyolultságú strukturálatlan problémát. Ekkor mondható el, hogy a gráf strukturáltságából fakadó nagy különbséget kiküszöböltük, és a probléma bonyolultságára annak típusától (RAP vagy véletlen) függetlenül jó becslést tudunk adni.

Jó ötlet ekkor a bonyolultság ábrázolása a tömörített méret függvényében. A kérdéses függvényeket a vizsgált gráfokon mért (tömörített méret; bonyolultság) pontokra fogjuk illeszteni; mint azt a vizsgált gráfok című részben már említettem, az azonos csúcsszámú gráfokon kapott mérési eredményeket a mérési hibák elkerülésének érdekében kiátlagoljuk. Tehát ha például van 7 darab 58 csúcsú RAP gráfunk, melyek tömörített méretének átlaga 700 bit, a futásidejük átlaga pedig 1500 ms, akkor az ábrázoló függvény a 700-ban az 1500 értéket fogja felvenni. (Az átlagokat minden esetben egészre kerekítettem.)

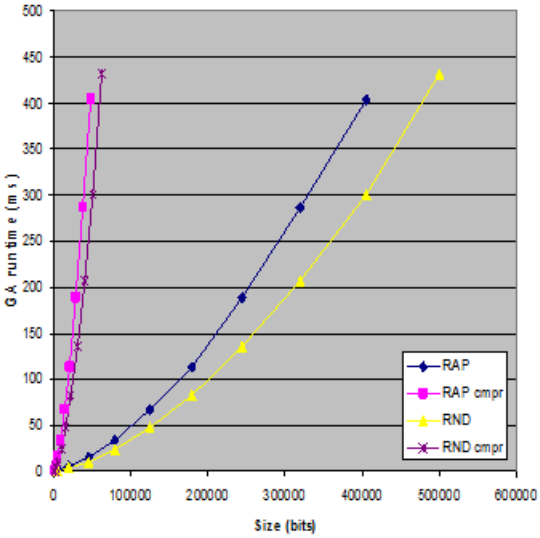
Ezután a mért eredményekből kapott pontokat összekötjük, feltételezve, hogy elég sűrűn helyezkednek el ahhoz, hogy a lineáris itt már jó közelítés legyen. Ezzel kaptunk egy „függvényt”, amely azt ábrázolja, hogy egy adott tömörített mérethez mekkora bonyolultság tartozik.

A szemléltető ábrákon külön függvényt készítünk a regiszterallokációs gráfok pontjain (*RAP cmpr*), illetve a véletlen problémákhoz tartozó pontokon (*RND cmpr*), mivel a két görbe (a feladat szempontjából nem ideális tömörítési eljárások esetén) adott esetben igen különböző lehet. Egy ábrán onnan láthatjuk, hogy a tömörítési eljárásunk jó előrejelzést szolgáltat, hogy a két görbe közel helyezkedik el egymáshoz (hiszen ekkor igaz, hogy azonos tömörített mérethez hasonló

bonyolultság tartozik).

Persze a „közel van egymáshoz” igencsak relatív, így nem ártana valamilyen viszonyítási alapot találnunk. Jó választás erre a szerepre az adatok eltárolásának természetes módja, a szomszédossági mátrix. Ha ennek a méretének függvényében ábrázoljuk a bonyolultságot, az jól jellemzi a számítástudomány fő irányvonalát, amely csak a csúcsszámból becsli a nehézséget, mivel ekkor egy adott csúcsszámú gráf szomszédossági mátrixának mérete független a gráf strukturáltságától. Adott tehát az ábráinkon még két függvény: az egyik (*RAP*) a regiszterallokációs gráfok, míg a másik (*RND*) a véletlen gráfok esetén ábrázolja a probléma bonyolultságát a szomszédossági-mátrix méretének függvényében. A tömörítésünk pedig akkor jobb predikció a bonyolultságelmélet klasszikus becslésénél, ha a *RAP cmpr* és *RND cmpr* függvények közelebb vannak egymáshoz, mint a *RAP* és *RND* függvények.

A kialakított módszer még egy apró módosításra szorul. Ha ugyanis egy tömörítés sokkal kisebb helyen tárolja a gráfot a szomszédossági mátrixánál, akkor előfordulhat, hogy a tömörített méretek az *x* tengely nagyon kis részébe lesznek „bezsúfolva”, így – mivel a tömörített és tömörítetlen méret a közös értelmezési tartománynak nagyon különböző részén van jelen a közös értelmezési tartománynak – összehasonlításuk igencsak nehézkessé válik (ld. 9. ábra: *Függvénypárok normálás nélkül*). Erre megoldás, ha a grafikonokat lenormáljuk; vagyis az *RND* és *RAP* függvények minden pontjához tartozó *x* (tömörítetlen méret) értéket elosztjuk pontjaik uniójából közül a legnagyobb *x* koordinátájú pont *x* koordinátájával, az *RND cmpr* és *RAP cmpr* függvénypárral pedig ugyanezt tesszük. Így mindkét függvénypár értelmezési tartománya a [0; 1] lesz, ami a köztük lévő különbségek összehasonlítását lényegesen egyszerűbbé teszi.



9. ábra: Függvénypárok normálás nélkül

Az eredmények szekció összes függvénye ezen alapelveket követve került ábrázolásra.

4.2. Mérőszám keresése

Bár az előbbi rendszerben kialakított ábrák jól szemléltetik egy előrejelzés minőségét, fontos lenne valamilyen mérőszámot is rendelnünk ehhez, mivel enélkül az eredmények precíz

kiértékelése igen nehéz.

Adatsorok közötti távolság mérésére több módszer is elképzelhető. Legtermészetesebbként talán a $\sum(a_i - b_i)^2$ formula merül fel, ám ez igen nehezen megvalósítható, mivel a tömörített méretek esetében a két adatsor adatpontjai különböző *x* értékeknél találhatók (attól függően, hogy az adott gráfokat épp mekkorára sikerült tömöríteni a vizsgált eljárással). Következő ötletként az $\int(a(x) - b(x))^2$ kerülhet elő, ám az adatpontok között lineáris közelítéssel készített függvényünk integrálása hosszadalmas feladat. Az adatpontjainkra próbálhatnánk polinomot illeszteni, azonban a gyakorlati tapasztalatok azt mutatták, a lineáris közelítések sokkal reálisabb függvényeket eredményeztek ezeknél a magasabb fokú polinomoknál.

Így az integrálásnál jobb megoldásnak tűnik az egyik adatsort (a lineáris közelítésben bízva) valóban függvénynek tekinteni, s a másik adatsor adatpontjainak x_i koordinátájához tartozó a_i értéket hasonlítani ennek a függvénynek az (esetleg egy közelítő szakaszon lévő) adott *x* koordinátához tartozó b_i értékéhez. Ekkor már használhatóvá válik a $\sum(a_i - b_i)^2$ képlet, ami egy jó mérőszáma a két függvény közötti eltérés mértékének. A kutatási eredmények kiértékelésekor is ezt használtam; minden függvénypár esetén az *RND* pontokhoz tartozó értékeket hasonlítottam a *RAP* függvény adott pontokba extrapolált értékeihez a $\sum(a_i - b_i)^2$ formulával. Ezt tekintjük tehát egy függvénypár távolságának; jelölése (*f* és *g* függvények esetén) legyen $d(f, g)$.

Ekkor egy tömörítési eljárás előrejelzésének minőségét a *RAP cmpr – RND cmpr* görbék távolságának, valamint a *RAP* és *RND* függvények távolságának aránya adja. *Előrejelzési aránynak* nevezzük tehát mostantól a következő arányt:

$$\frac{d(RAP\ cmpr, RND\ cmpr)}{d(RAP, RND)}$$

Az egynél nagyobb előrejelzési arányú függvények a számítástudományi csúcsszám becslésnél rosszabb becslést adnak, az egynél kisebb arányúak pedig annál jobb előrejelzéshez vezetnek, és annál jobbnak tekinthetünk egy predikciót, minél kisebb az előrejelzési aránya.

Fontos még megjegyeznünk, hogy a kutatás során a távolságmérést az értelmezési tartomány azon részén végeztük el, ahol mindkét függvény értelmezve volt, így a két-két függvénypár távolsága sokszor különböző számú pont alapján került meghatározásra. Persze az adott távolságértékeket ezután a pontok arányának megfelelően korrigáltuk, így ha $N(f, g)$ jelöli a távolságméréshez használt pontok számát, akkor az előrejelzési arány:

$$\frac{d(RAP\ cmpr, RND\ cmpr)}{d(RAP, RND)} \cdot \frac{N(RAP, RND)}{N(RAP\ cmpr, RND\ cmpr)} = \frac{\frac{d(RAP\ cmpr, RND\ cmpr)}{N(RAP\ cmpr, RND\ cmpr)}}{\frac{d(RAP, RND)}{N(RAP, RND)}}$$

4.3. A mért bonyolultságokról

A felhasznált gráfok egyes algoritmusokkal mért bonyolultságáról (vagyis a függvény alappontjainak *y* koordinátájáról) tudjuk, hogy nem függ az éppen vizsgált tömörítési eljárástól; az egyes tömörítésekhez tartozó függvények különbségét az fogja adni, hogy ezeket az értékeket mely pontokban fogják felvenni, mely *x* koordináta tartozik ezekhez az állandó *y*-okhoz a függvény alapjaiként funkcionáló referenciapontoknál. Érdemes tehát a kapott értékekről általánosságban is írnunk, mielőtt a konkrét eljárások eredményeinek vizsgálatát megkezdjük.

A használt algoritmus (GraphColouringBB vagy GraphColouringGA) és a színek száma (tehát azt a problémát elemezzük-e, amikor adott *k* szín áll rendelkezésünkre minden gráfhoz, vagy minden példányhoz a saját kromatikus számának megfelelő színt használunk) alapján négy különböző vizsgálati eset képzelhető el. Ezek mindegyikéről szólunk most néhány szót.

A regiszterallokációs gráfok felépítésekor említettük, hogy azok mindegyikére $\chi = 21$ teljesül. A véletlen példányok kromatikus számára (gyakorlatilag csúcsszámtól függetlenül) minden esetben az 5, 6, 7 egyikét, a legnagyobb gráfok esetén néha a 8 értéket figyelhettük meg [26].

I. A gráf színezése a kromatikus számának megfelelő számú színnel, egzakt algoritmussal

A legtermészetesebb (és így számunkra leglényegesebb) eset, mivel a színezés bonyolultsága szempontjából a gráfok színezésének legbonyolultabb esetei (a kromatikus számúak) tekinthetők ekvivalensnek egymással, így ezek összehasonlítása kiemelten fontos; továbbá ezen algoritmus futásidejének bonyolultságát fontosabb a felhasználások szempontjából tudni. Bár ebben az esetben nem egyértelmű, hogy az eldöntési vagy az optimalizálási problémát érdemes vizsgálnunk, a gyakorlatban azt tapasztalhatjuk, hogy a RAP gráfok 21 színnel már olyan könnyen színezhetők, hogy az algoritmus általában backtrack nélkül is boldogul vele; mivel az eldöntési probléma csupán ebből áll, így az ebből kapott adatok nehezen értelmezhetők. Az ilyen grafikonok tehát mindig az optimalizálási problémát (a kromatikus szám megkereséséhez szükséges backtrackszámot) fogják vizsgálni.

A regiszterallokációs problémák bonyolultsága ebben az esetben egy adott korlát alatt maradt; semelyik gráfhalmazhoz nem volt szükség 200-nál nagyobb átlagos backtrackszámra, ami (főleg a 100 pont fölötti gráfoknál) nagyon könnyű eseteket jelent, az algoritmusnak gyakorlatilag semmi problémája nem akadt ezeknél a példányoknál. Véletlen esetben azonban jól látszott a függvény exponenciális jellege; míg 60 pont körül 500 alatti értékeket kaptunk, száz pont körül már milliós backtackszámokat tapasztalhattunk. A drasztikus különbségek miatt ebben az esetben az *y* tengelyt mindig (tízes alapú) logaritmikus típusúra választottuk.

Mivel ez azt jelenti, hogy magas csúcsszámnál hihetetlenül nagy eltéréseket tapasztaltunk a RAP és a véletlen gráfok között, itt egy tömörítési eljárás csak úgy adhat jó eredményt, ha sikerül az összes vizsgált RAP gráfot olyan kicsire tömörítenie, hogy az ugyanekkorára tömörített szabálytalan gráfok még a viszonylag kis bonyolultságú példányok közül valók legyenek. Ellenkező esetben ugyanis egy adott tömörített mérethez hatalmas bonyolultságkülönbségek tartozhatnának, így csak nagyon rossz becslést adhatunk belőlük.

II. Egzakt algoritmus használata adott k számú színnel

Nem igazán volt vizsgálható: mivel a RAP gráfok kromatikus száma 21 volt, $k \geq 21$-nek teljesülnie kellett, ám a rengeteg intelligens heurisztikával ellátott Branch-And-Bound módszer számára a véletlen gráfok színezése a kromatikus számuknál legalább háromszor több színnel egyáltalán nem volt kihívás, az esetek nagy többségében 0 backtrackkel lefutott, így a bonyolultságok semmilyen összehasonlításra nem voltak alkalmasak. Ez persze nem azt jelenti, hogy a véletlen gráfok könnyebbnek bizonyultak volna, csak azt, hogy a színezést valóban a kromatikus számhoz viszonyítva érdemes vizsgálni, mert ellenkező esetben az eredményre komoly hatással van a vizsgált gráfok közötti, strukturáltság melletti másik lényeges különbség, a kromatikus szám nagy eltérése.

A másik két esetben a GraphColouringGA-val színezünk. Mivel ez a genetikus algoritmus kromatikus szám keresésére nem alkalmas, csak azt tudja vizsgálni, hogy a gráf kiszínezhető-e valahány színnel, itt mindig csak az eldöntési probléma elemzése jön szóba.

III. Színezés kromatikus számú színnel, genetikus algoritmussal

Itt is hatalmas bonyolultságkülönbségeket tapasztalunk. Míg a regiszterallokációs gráfokon egy másodperc alatti átlagos futásidőket kaptunk, addig a véletlen példányok nehézsége még az egzakt algoritmusoknál megfigyeltnél is sokkal nagyobb mértékben emelkedik. A 48 pontú gráfok

némelyike már egy percnél is bőven további futáshoz vezet, a 63 csúcsú véletleneknek pedig egyike sem fut le néhány perc alatt sem. Így az eset grafikonjain az *RND*, illetve *RND cmpr* függvényeket csak az első néhány gráfhalmazból kapott adatokra illesztettem, és ezeken belül is minden, 40 másodperc alatt le nem futtatható eredményt 40 másodpercnek tekintettem (holott ennél nyilván nagyságrendekkel nagyobb volt.) Ez az eset nem látszott semmi szabályosságot mutatni, így az egyes tömörítési eljárásoknál nem is sokat szólunk róla.

IV. A genetikus algoritmus használata adott k számú színnel

A RAP gráfok kromatikus száma miatt $k \geq 21$ lehet csak, és mivel ugyanezen gráfok színezése tapasztalat szerint 22 színnel már nagyon könnyű, $k=21$ tűnik az egyetlen lehetséges választásnak. Az algoritmus lefuttatása után azt tapasztaltam, hogy mind a RAP, mind a véletlen gráfok igen kicsi, 2 másodperc alatti futásidőket produkálnak. Ez még önmagában nem meglepő, hisz már tapasztaltuk, hogy 21 színnel mindkét gráfcsoportot igen könnyű színezni (az egzakt algoritmusnál, ami a RAP-okon 200 alatti backtrackszámokat ad, a véletleneken pedig általában 0-t), ám a részletes vizsgálat szerint nem csak erről van szó. Tulajdonképpen a genetikus algoritmus minden esetben néhány, általában már 0 generáció után talál egy megoldást; ez úgy lehetséges, hogy a GraphColouringGA a kezdeti populáción (amelynek már a létrehozásához is intelligens módszereket alkalmaz) lefuttat egy, a könnyen javítható hibák kiszűrésére szolgáló heurisztikát, ami ilyen könnyű esetekben már általában önmagában is képes a populáció rengeteg egyedéből legalább egy esetében jó megoldást generálni. A genetikus algoritmus futásidejének elemzésekor tehát ebben az esetben valójában ennek a heurisztikának a futásidejét mérjük, de ettől ezt még tekinthetjük eredménynek, hisz ez ugyanúgy egy (bár csak elég könnyű esetekben hatásos) heurisztukus megoldási módszer bonyolultságát adja.

Táblázatban összegezve a mért bonyolultságok alapvetően a következők:

Bonyolultság	**Egzakt algoritmus (*backtrackszám*)**	**Genetikus algoritmus (*futásidő*)**
RAP gráfok *21 szín = χ szín*	Könnyű példányok *(200 alatti értékek)*	Könnyű példányok *(1 másodperc alatti értékek)*
Véletlen gráfok *χ szín (5, 6, 7, 8)*	Nehéz példányok *(exponenciálisan nő; 60 pontnál 300, 100 pontnál már 1000000 körül)*	Nagyon nehezek *(már a 48 pontúakból is néhány, 60 pont felett pedig már mindegyik több percig fut)*
Véletlen gráfok *21 szín*	Vizsgálhatatlanul egyszerűek *(szinte mindig 0 backtrack)*	Könnyű példányok *(2 másodperc alatti értékek)*

A fejezet tapasztalatait tehát úgy foglalhatjuk össze, hogy a gráfok tulajdonságai, a köztük lévő

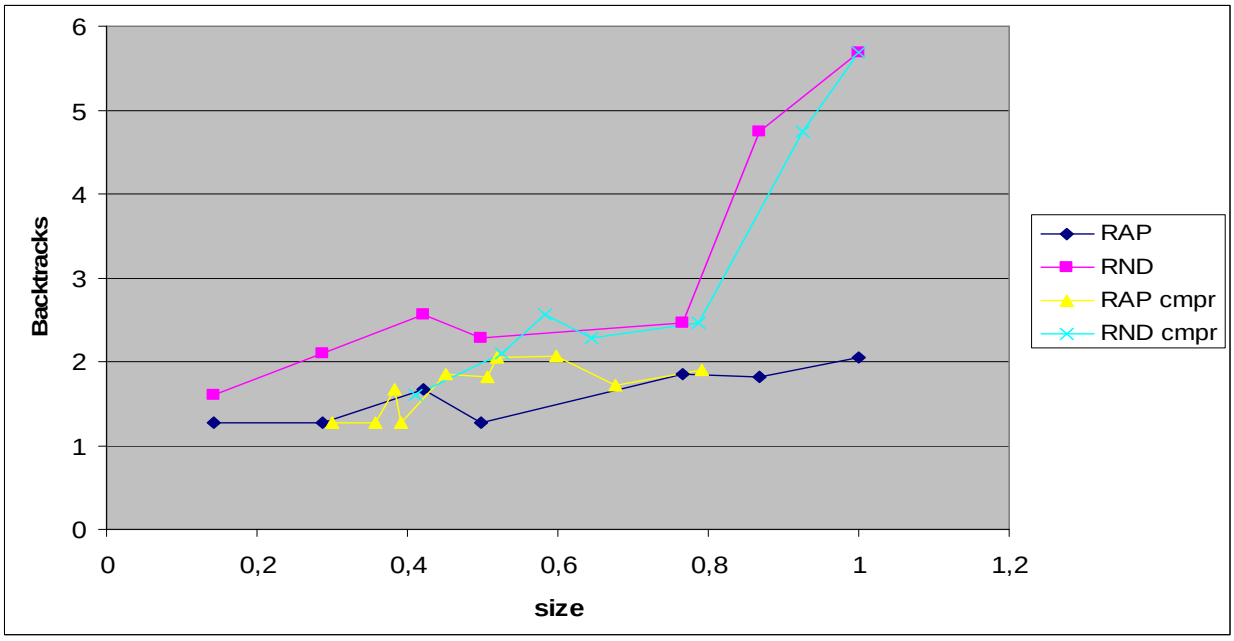
szélsőséges különbségek, illetve az algoritmusok jellemzői miatt csak néhány esetben tudjuk jól összehasonlítani a mért eredményeket. Az alábbi táblázat foglalja össze a problémákról szóló megállapításainkat.

Vizsgált probléma		**Egzakt algoritmus**	**Genetikus algoritmus**
χ szín	**Eldöntési probléma**	Nem vizsgálható *(RAP gráfokon túl könnyű, általában 0 backtrack)*	**Vizsgálható** *(A drasztikus különbségek miatt nem ad jó eredményt)*
	Optimalizálási probléma	**A legjobban vizsgálható**	A genetikus algoritmussal nem vizsgálható *(Hiszen optimalizálási probléma)*
21 szín	**Eldöntési probléma**	Nem vizsgálható *(Véletlen gráfokon túl könnyű, általában 0 backtrack, mert a színszám a χ sokszorosa)*	**Vizsgálható**

4.4. A zippelés

Egy mindennapi informatikában is rengetegszer használt módszernek nagyon hatékonynak kell lennie; nem meglepő hát, hogy tömörítés szempontjából a Zip bizonyult a legjobbnak a négy eljárás közül. Szinte kivétel nélkül minden gráfot 1000 bit alá tömörített, ami jelentős teljesítmény, ha belegondolunk, hogy a nagyobb gráfok szomszédossági mátrixának mérete eredetileg 6-7 ezer bit felett volt.

Az eljárás szintén hatalmas különbséget talált a RAP, illetve a véletlen példányok szomszédossági mátrixa között. Nagyon jó közelítéssel elmondható, hogy egy adott regiszterallokációs gráfot feleakkora méreten tudott leírni, mint a neki megfelelő véletlen gráfot.



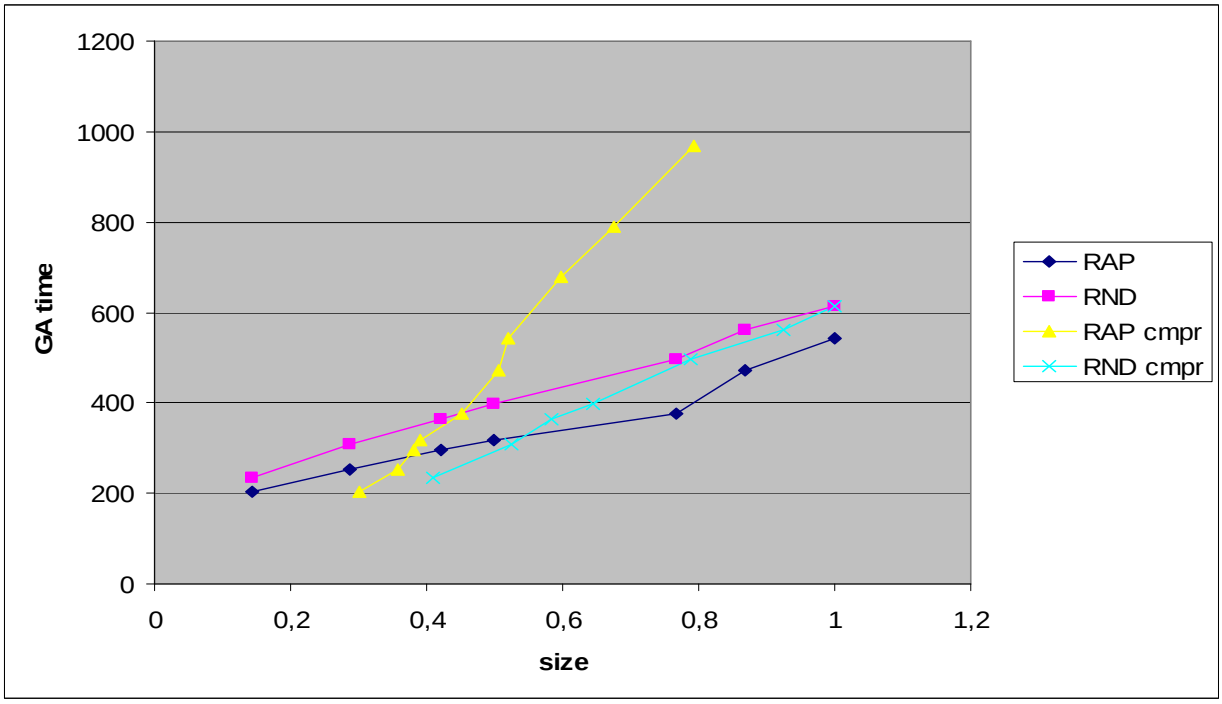
10. ábra: Az egzakt algoritmus backtrackszáma a zippelt méret függvényében. Az y tengely tízes logaritmikus.

Az egzakt algoritmus vizsgálatával kapott eredményeket a fenti ábra mutatja. Az előző fejezet megállapításai alapján ez a kromatikus szám megkereséséhez szükséges backtrackek számát tekinti bonyolultságnak; ezt látjuk a tömörített méret függvényében. Ez a grafikon egész bíztatónak tűnik; amely intervallumban mind a *RAP cmpr*, mind az *RND cmpr* függvények értelmezve vannak, ott jelentősen jobban követni látszanak egymást, így alapvetően kisebb távolságot mutatnak, mint a *RAP* és *RND* grafikonok. Mint megállapítottuk, ez a véletlen gráfok bonyolultságainak 80 csúcs körültől történő hirtelen növekedése mellett csak úgy lehetséges, ha a tömörítési eljárásunk még a nagy csúcsszámú RAP problémákról is felismeri, hogy ezek igen könnyűek csúcsszámukhoz képest. Ezt a zipnél nem is lehetne jobban illusztrálni; a tömörítés folyamán még a 152 pontú problémák tömörített mérete is csak akkora, mint véletlen gráfok közül a 78 pontúaké.

Előrejelzési arányra a teljes grafikonon egy nagyon jó eredmény (0.04) adódik. Felmerülhet ekkor a kérdés, hogy helyénvaló-e az előrejelzési arányt a függvénypárok teljes közös értelmezési tartomány mérni, ugyanis a drasztikus különbségért javarészt a 0.8 fölötti pontok felelősek; ha a négy függvényt az ennél kisebb értékekre nézzük, ott csupán 0.21-es arányt kapunk (ami még mindig elég jó). Ha meg akarjuk fogalmazni, hogy mitől ennyire lényeges az előrejelzési különbség a két intervallum között, a következőre jutunk: a [0, 0.8] intervallumon mind a csúcsszámból, mind a zippelt méretből többé-kevésbé elfogadható nehézségbecslést adhatunk. Azonban a teljes [0, 1]-en már előfordulhat, hogy a számítástudomány klasszikus becslésével hihetetlenül nagy eltéréseket kapunk (a 83 csúcsú RAP gráfok alig 65, az ugyanekkora véletlenek több, mint 50000 backtracket használtak). A 0.04-es arányszám kifejezi, hogy ez a zippelt mérettel nem történhet meg, mivel ez a tömörítés érzékelte, hogy még a 152 pontú RAP gráfok is csak a 78 pontú véletlenekkel vannak

hasonló nehézségi kategóriában.

A genetikus algoritmus színezésekor χ színnel az itt fellépő drasztikus különbségek miatt a futásidőre a zip nem ad túl jó becslést, az ehhez tartozó mérőszám a 0.75.



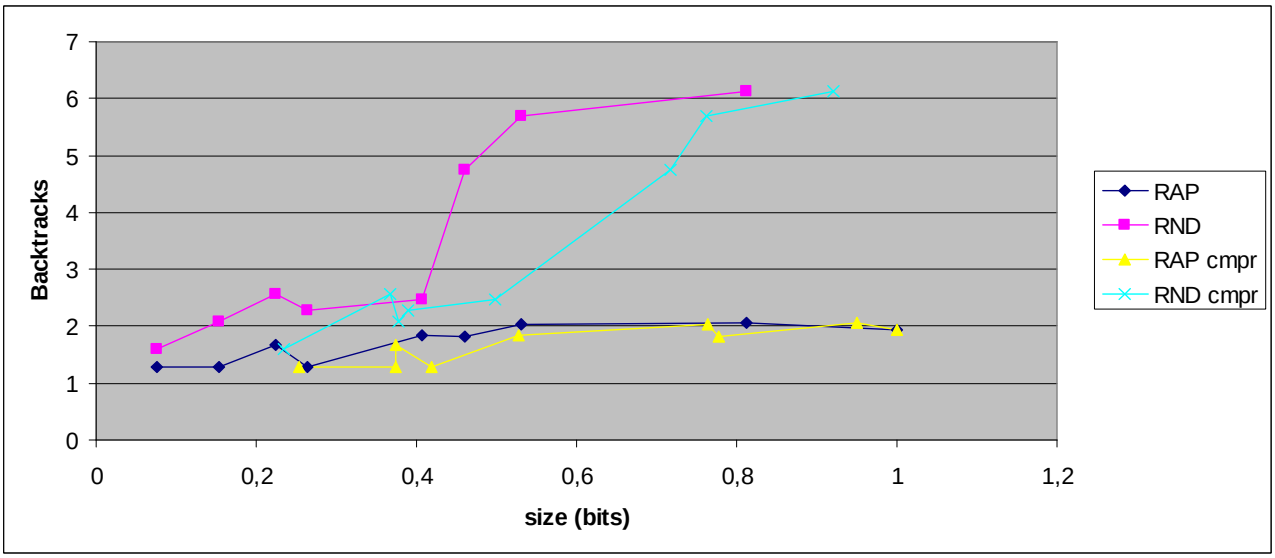
11. ábra: A 21 színt használó genetikus algoritmus futásideje ezredmásodpercben, a zippelt méret függvényében.

A harmadik vizsgálható eset a genetikus algoritmus futásidejének megfigyelése 21 színnel. Erre a zip előrejelzése egyáltalán nem jó, a futásidőknél ugyanis azt tapasztalhatjuk, hogy a véletlen gráfokon az algoritmus futásideje körülbelül 20%-kal nagyobb, mint a nekik megfelelő RAP példányokon. Mivel ez az arány a tömörített méretnél sokkal nagyobb (a véletlenek mérete körülbelül a RAP-ok kétszerese), a tömörített függvénypár nyílásszöge hatalmas lesz, így nagy csúcsszám esetén igen távol lesznek egymástól. Ez tehát nagyon rossz becslést jelent; előrejelzési aránynak 18.37 adódik, vagyis a klasszikus bonyolultságelméleti megközelítés több, mint 18-szor pontosabb predikciót szolgáltat.

4.5. Tömörítés Subdue-val

A négy vizsgált eljárás közül a Subdue tömörítése szolgáltatta a legkevesebb eredményt. A fő meglepetést ebben az esetben az okozta, hogy a strukturált és véletlen példányok közötti különbség alig volt hatással a tömörített méretre, sokszor a véletlen példányokat kisebb helyre tömörítette a megfelelő RAP-oknál.

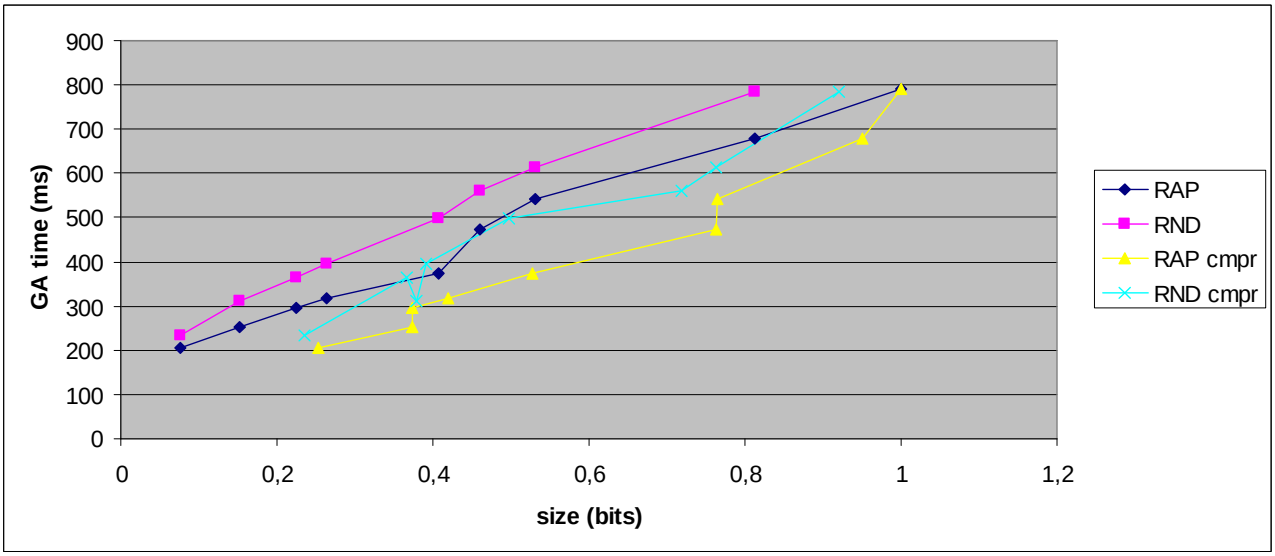
A lenti grafikon az egzakt algoritmuson mért adatok grafikonja. Láthatjuk, hogy a RAP és RND függvények nincsenek lényegesen távolabb egymástól, mint a tömörítettek, így nem számíthatunk túl jó előrejelzésre. Az előrejelzési arány 1.0; ez a tömörítés ennél az algoritmusnál pont annyira jó, mint a csúcsszámból adott becslés.



12. ábra: Az egzakt algoritmus backtrackszáma a Subdue-s tömörített méret függvényében. Az y tengely logaritmikus beosztású.

A genetikus algoritmus kromatikus számú színezése itt sem vezetett eredményre; a 0.73-es érték nem tér el lényegesen a zip esetén mért számtól.

Végül a harmadik vizsgálható esetünk, a genetikus algoritmus futásideje 21 színnel sem különösebben jó eredmény. Itt az eredeti, illetve a tömörített függvénypárok grafikonjai között körülbelül ugyanakkora, sőt, az eredetiek közt egy kicsivel kisebb távolságot tapasztalhatunk. A konkrét mérőszám értékére 1.23-at kapunk. A Subdue tömörítését tehát egyik esetben sem tudtuk sikeresen alkalmazni; ennek az lehet az oka, hogy az alapja az ismétlődő részstruktúrák keresése, amiknek viszont nincs lényeges hatásuk a gráfszínezés bonyolultságára.

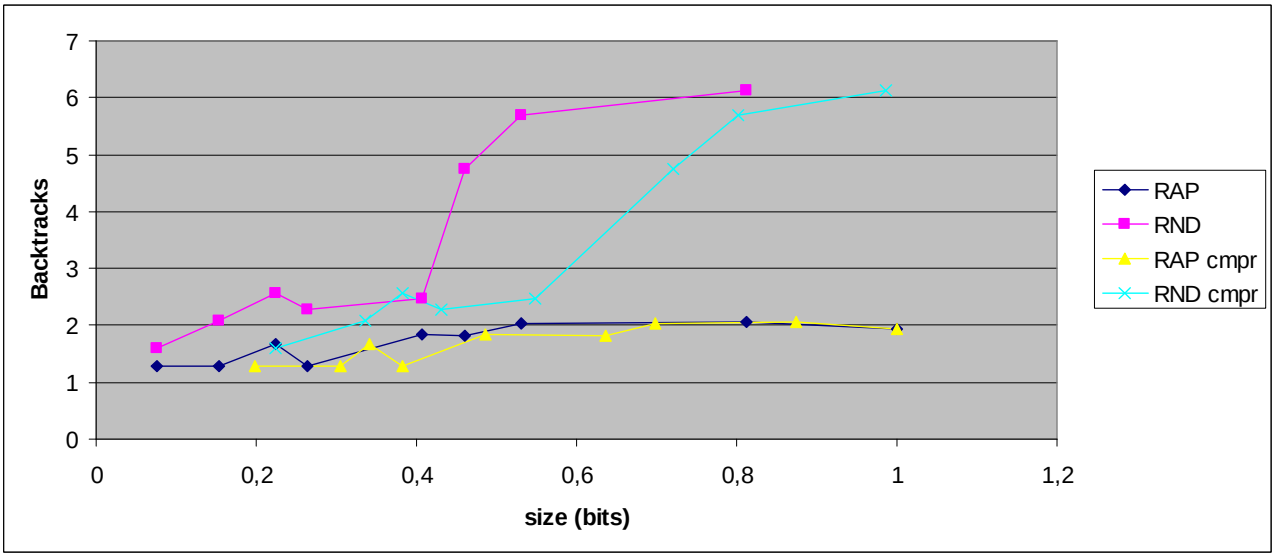


13. ábra: A genetikus algoritmus futásideje 21 színnel, a Subdue-s tömörített méret függvényeként

4.6. Az MDL

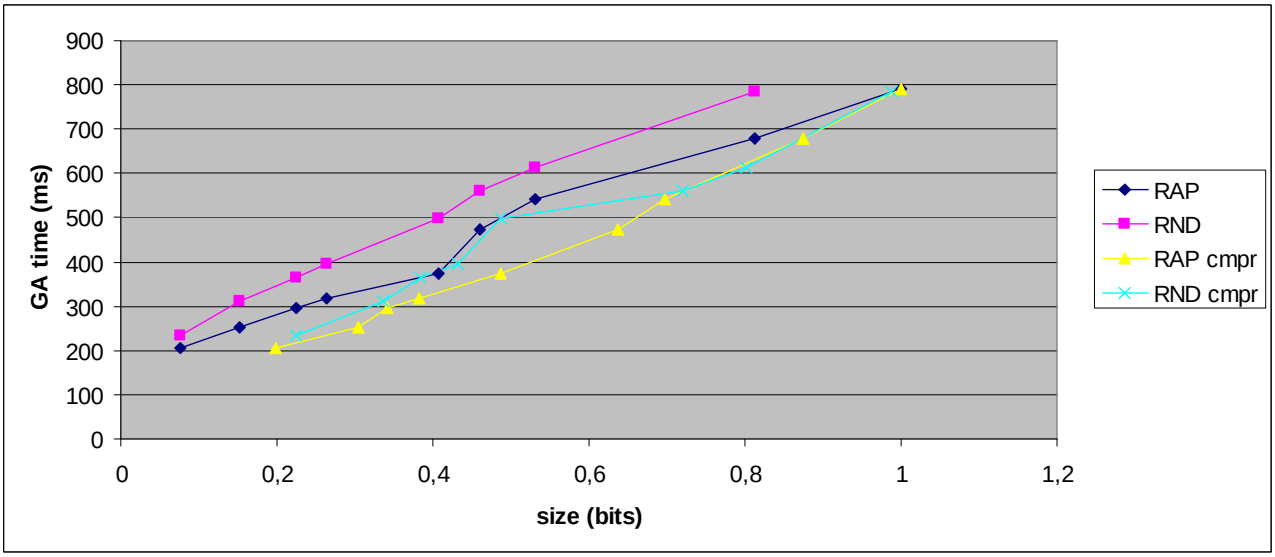
Tömörítés szempontjából az MDL meg sem közelítette a zip szintjét, de tulajdonképpen elfogadható tömörítést adott. Kis gráfoknál ugyan messze a szomszédossági mátrix mérete fölötti helyre volt szüksége, de nagyobb pontszámok esetén már mind a RAP, mind a véletlen példányokat a mátrix méreténél kisebbre sűrítette.

Az MDL ugyan érzékelte a strukturált és szabálytalan példányok közötti különbséget, de messze nem annyira, mint a zippelés. Míg ez utóbbinál kétszeres különbséget figyelhettünk meg az egymásnak megfelelő példányok között, addig az MDL esetén csupán 10-20%-os eltérés adódott. Ennek köszönhetően az egzakt algoritmus hatalmas bonyolultságkülönbségeit nem tudta követni, ám annál jobb (a vizsgáltak közül messze a legerősebb) becslést ad a genetikus algoritmus 21 színes színezésére, ahol a futásidőkben éppen ekkora eltérések tapasztalhatók a RAP és nekik megfelelő véletlen példányok között.



14. ábra: Backtrackszám logaritmikus skálán az MDL-lel tömörített méret függvényeként

A konkrét előrejelzési arányok: az egzakt esetében 0.99, a genetikus algoritmus kromatikus számú színnel 0.69, míg ugyanez 21 színnel 0.24 arányú. Ez alapján tehát azt mondhatjuk, hogy a használt külső programok közül az egzakt algoritmus backtrackszámát a zippeléssel, a genetikus algoritmus 21-színes futásidejét az MDL-lel tudtuk messze a legjobban előre jelezni.

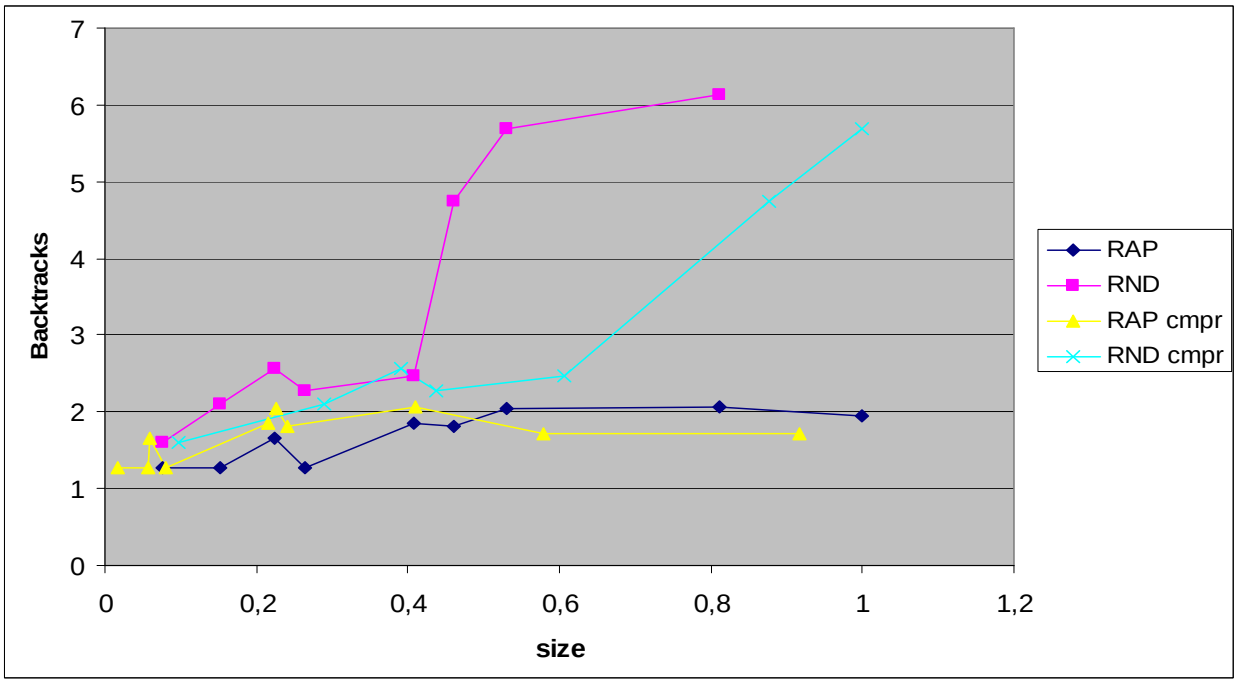


15. ábra: A 21 színes genetikus algoritmus futásideje az MDL által megadott méret függvényében

4.7. A GSA eredményei

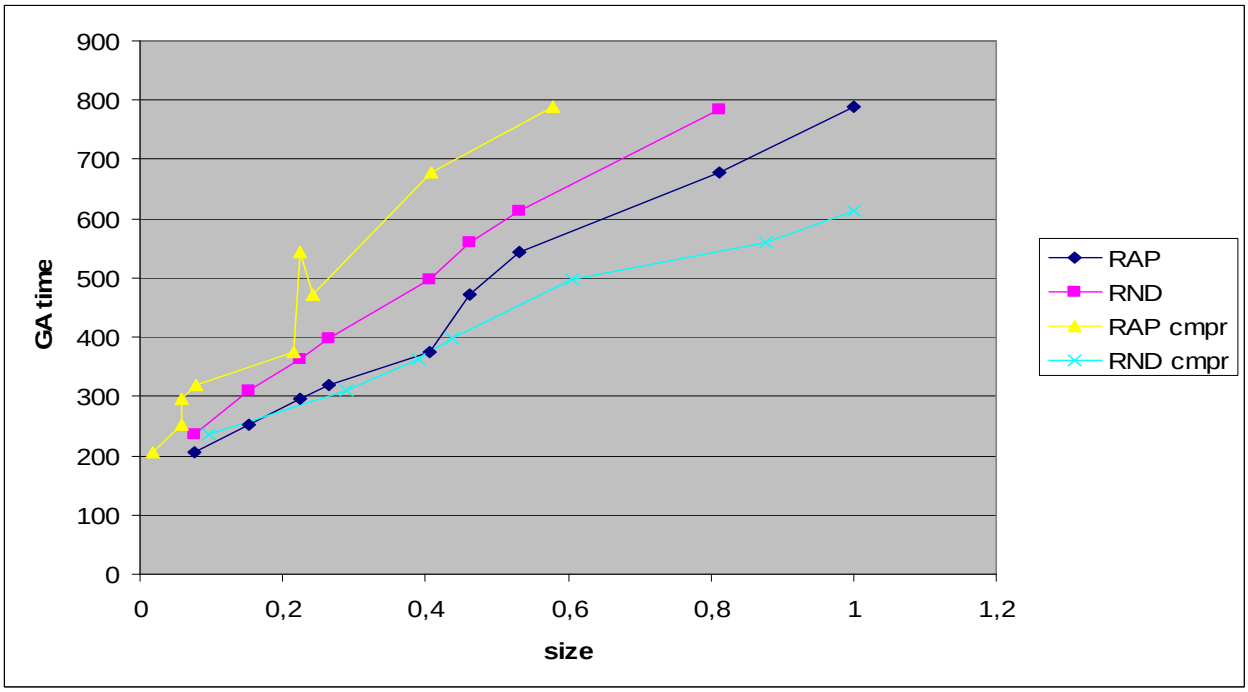
A GSA tömörítési képességeinek elemzésekor azt tapasztaljuk, hogy a regiszterallokációs példányokat jól tömöríti, a véletlenek esetén viszont általában csak a szomszédossági mátrix méreténél nagyobb leírást talál.

A négy tömörítési eljárás közül messze a GSA érzékeli legjobban a véletlen és strukturált példányok közötti különbséget. Az algoritmusunk a RAP példányokat szinte minden esetben a megfelelő szabálytalan példányoknál négyszer kisebb helyre tömörítette. Eddigi tapasztalataink alapján már sejthetjük, hogy ekkor a hatalmas nehézségkülönbségeket mutató BB-ra jó becslést fog adni, míg a sokkal kisebb eltéréseket adó genetikus színezésre elég rosszat.



16. ábra: Az egzakt algoritmus backtrackszáma a GSA-val tömörített méret függvényében. Az y tengely logaritmikus.

Valóban, láthatjuk, hogy utolsó *RAP cmpr* pont kivételével a tömörített grafikonok nagyon jól követik egymást, távolságuk nagyon kicsi: 0.29-es arányt kapunk a grafikonból. Fontos megemlítenünk, hogy ezért alapvetően az utolsó RAP pont felelős; ha csak a 150 pont alatti gráfokon tekintjük a grafikont, akkor becslési arányra hihetetlen jó, 0.02 adódik. Valóban, jól látszik, hogy addig a két *cmpr* grafikon nagyon közel mozog egymáshoz. Összesítve tehát, a zip jobb arányt produkált a GSA-nál, ám ez alapvetően azért történt, mert volt egy gráfhalmaz, amelyen a GSA elég rossz eredményt adott. Az algoritmus további fejlesztésének egyik célja az ilyen esetek számának minimalizálása. Ugyanakkor sikernek tekinthető, hogy a 150 pont alatti gráfpéldányokon a GSA adta a kutatás során mért legjobb, 0.02-es előrejelzési arányú becslést. (A zippelés a 150 pont alatti gráfokon 0.03-as arányt produkál.)



17. ábra: A genetikus algoritmus futásideje 21 színnel, ezredmásodpercben, a GSA-s méret függvényeként.

A genetikus algoritmus kromatikus számú színezésére itt sem kapunk túl erős becslést, 0.75-ös arány tartozik hozzá.

A vártnak megfelelően a genetikus algoritmus 21 színes színezésénél azt kapjuk, hogy a GSA előrejelzései sokkal rosszabbak a csúcsszámból adottnál. A tapasztalat tehát az, hogy az általunk írt program előrejelzési tulajdonságait tekintve nagyon hasonlít a zippeléshez. Valóban, az ehhez tartozó 2.91-es arány azt jelzi, hogy lényegesen rosszabb a számítástudomány becslésénél, habár még mindig nagyságrendekkel jobb, mint a zip ebben az esetben.

4.8. Összegző táblázat

A szemléltetés kedvéért érdemes a kapott előrejelzési arányokat egy táblázatban összefoglalnunk. A táblázat egyes cellái a függvénypárok közti távolságarányt mutatják százalékos formában, ahol a csúcsszámból adott becslést tekintjük 100%-nak. Az egyes cellák utolsó sorai az összes arány összevetését is tartalmazzák, eredeti–zippelt–subdue–MDL–GSA sorrendben.

Színek száma	**Egzakt algoritmus**	**Genetikus algoritmus**
χ szín	**Optimalizálási probléma** Zippelés: 4 : 100 Subdue: 100 : 100 MDL: 99 : 100 GSA: 29 : 100 (*150 alatt 2 : 100*) *100 : 4 : 100 : 99 : 29 (E:Z:S:M:G)*	**Eldöntési probléma** Zippelés: 75 : 100 Subdue: 73 : 100 MDL: 69 : 100 GSA: 75 : 100 *100 : 75 : 73 : 69 : 75 (E:Z:S:M:G)*
21 szín	*Nem vizsgálható, mivel a véletlen gráfok legtöbbjén 0 backtrackkel lefut*	**Eldöntési probléma** Zippelés: 1837 : 100 Subdue: 123 : 100 MDL: 24 : 100 GSA: 291 : 100 *100 : 1837 : 123 : 24 : 291 (E:Z:S:M:G)*

5. Összefoglalás

5.1. Eredmények

A dolgozatban azt vizsgáltuk, hogy adott problémák bonyolultságát jobban tudjuk-e becsülni a tömörített méretből, mint az eredeti input-méretből. Ehhez gráfszínezési problémák futásidő-igényét figyeltem meg egy egzakt algoritmussal és egy heurisztikával, mégpedig regiszterallokációs problémákból származó (strukturált) és véletlen példányokon.

A zip tömörítés esetében azt tapasztaltuk, hogy az egzakt algoritmus futásidejére a számítástudományi megközelítésnél lényegesen jobb előrejelzést adhatunk; ezen a területen ez teljesített legjobban a három külső program közül. Ugyanakkor a genetikus algoritmus futásidejének megjósolására ez a módszer nyilvánvalóan alkalmatlan volt.

A Subdue tömörítése semelyik algoritmus esetén nem adott igazán jó becslést.

Az MDL ugyan nem mutatott fel eredményeket az egzakt esetben, de a genetikus algoritmus esetén meglepően jó előrejelzést kaphattunk a segítségével. A heurisztika esetén tehát a vizsgált négy módszerből messze ő bizonyult legsikeresebbnek.

A kutatás során megírtam a GSA-t, amely a genetikus algoritmusok alapelveit követve megpróbálja szemléletes elemek összességeként felépíteni a gráfot, majd a legjobb megtalált leírás hosszát tekinti a gráf tömörített méretének. Az így kapott program az előrejelzések terén tulajdonságait tekintve igencsak hasonlított a zippeléshez; a heurisztikára lényegesen gyengébb becslést ad a számítástudományi szemléletűnél, de az egzakt algoritmus nehézségének becslésére (amely a kettő közül a lényegesebb) nagyon jól jelez előre, kisebb gráfok esetén legyőzve a zip tömörítést is.

Összességében tehát sikeresnek mondhatjuk a kutatást; a feltételezésnek megfelelően valóban összefüggést találtunk a tömörített méret és a bonyolultság között, ráadásul a különböző tömörítési eljárásokat sikerült különböző esetekben hasznosítanunk. Ugyancsak siker, hogy a kutatás céljából általunk írt algoritmus is kifejezetten hatékonynak bizonyult.

5.2. További munka

A kutatás során ugyan több különböző szempontból megvizsgáltuk a tárgyalt problémát, ám a téma még rengeteg felderítetlen lehetőséget rejt magában.

Érdemes lehet például utánanézni, hogy a dolgozat két gráftípusán felüli, más fajtájú példányok, mondjuk más alkalmazási területről összegyűjtöttek vagy más véletlen modellel létrehozottak hasonló módon sejtéseinket igazolják-e. Ugyancsak érdemes lenne a rendelkezésre álló gráfokon egy tetszőleges másik NP-teljes gráfprobléma (például a Hamilton-kör keresés) bonyolultságát vizsgálni, és megfigyelni, hogy az itt tapasztaltak mennyiben térnek el a gráfszínezésnél érzékelt jelenségektől.

A Graph Structure Analyzer is egy elég összetett rendszer ahhoz, hogy fejlesztése további lehetőségeket rejthessen. Érdemes lehet időt szánni annak kiderítésére, hogy egy jobb stratégia a kezdeti populáció kifejlesztésére, vagy újabb mutációs módszerek implementálása mennyit javíthat a predikciónk minőségén.

A GSA megírásakor az elsődleges szempont a könnyű átláthatóság volt, így ebből valamennyit feláldozva várhatóan a program futásidején szintén jelentős optimalizálást tudunk végezni. Mivel a programban igen sok különböző dologhoz szükséges $O(n^2)$ lépés, a nagyságrenden nem igazán tudunk már javítani, de az együttható csökkentésével még akár elérhetünk lényegi javulást.

Bár a tömörítéssel való előrejelzéshez, így a dolgozat témájához szorosan nem kapcsolódik, meg kell jegyeznünk, hogy jelen kutatásban egyáltalán nem használjuk ki teljes mértékben a GSA által nyújtott lehetőségeket. A program ugyanis nem csupán a vizsgált gráf tömörített méretét adja vissza, hanem megadja a legjobb megtalált leírást is; megkapjuk tehát egy gráf teljes felépítését, és ennek az információnak jelen kutatásunkban csak töredékét használjuk ki. A leírás rengeteg különböző tulajdonságáról (például nemtriviális objektumok száma, a teljes gráfok átlagos mérete, stb. [27]) szintén elképzelhető, hogy nagyon sokat elárulnak a probléma bonyolultságáról vagy akár egyéb aspektusairól. Ezek részletes vizsgálata a kutatás újabb távlatait nyithatja meg.

6. Irodalomjegyzék

[1] Frank Hutter, Youssef Hamadi, Holger H. Hoos, Kevin Leyton-Brown. Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms. In: *Lecture Notes in Computer Science - Volume 4204,* 2006.

[2] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, Lidror Troyanskyk. Determining computational complexity from characteristic `phase transitions'. In: *Nature 400,* 1999.

[3] Carla Gomes, Toby Walsh. Randomness and Structure. In: *Handbook of Constraint Programming, Elsevier,* 2006.

[4] Tad Hogg. Which Search Problems Are Random? In: *IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence,* 1998.

[5] Rudi Lutz. Evolving Good Hierarchical Decompositions of Complex Systems. In: *Journal of Systems Architecture: the EUROMICRO Journal - Special issue on evolutionary computing,* 2001.

[6] Diane J. Cook, Lawrence B. Holder. Substructure Discovery Using Minimum Description Length and Background Knowledge. In: *Computer Research Repository*, 1994.

[7] Preston Briggs, Keith D. Cooper, Linda Torczon. Improvements to graph coloring register allocation. In: *ACM Transactions on Programming Languages and Systems - Volume 16 Issue 3,* 1994.

[8] Nirbhay K. Mehta. The application of a graph coloring method to an examination scheduling problem. In: *Interfaces, 11,* 1981.

[9] Jens Palberg. Register allocation via coloring of chordal graphs. In: *Proceedings of the thirteenth Australasian symposium on Theory of computing - Volume 65,* 2007.

[10] Pál Erdős, Alfréd Rényi. On the evolution of random graphs. In: *MTA Matematikai Kutatóintézet Közl. 5,* 1960.

[11] Kurt Mehlhorn. Graph algorithms and NP-completeness. In: *Springer-Verlag,* 1984.

[12] Zoltán Ádám Mann, Anikó Szajkó. Determining the expected runtime of exact graph coloring. In: *Mini-conference on Applied Theoretical Computer Science,* 2010.

[13] Zoltán Ádám Mann, Anikó Szajkó. Improved bounds on the complexity of graph coloring. In: *Advances in the Theory of Computing,* 2010.

[14] C. Lucet, F. Mendes, A. Moukrim. An exact method for graph colouring. In: *Computers and Operations Research - Volume 33 Issue 8,* 2006.

[15] Rémi Monasson. On the Analysis of Backtrack Procedures for the Colouring of Random Graphs. *In: Complex Networks, Springer,* 2004.

[16] Tamás Szép, Zoltán Ádám Mann. Graph Colouring: the more colors, the better? In: *IEEE 11th International Symposium on Computational Intelligence and Informatics,* 2010.

[17] Zoltán Ádám Mann, Anikó Szajkó. Frekvencia allokáció számítástudományi megközelítésben. In: *Hálózattervezés és szimuláció,* 2010.

[18] Zoltán Ádám Mann, Tamás Szép. BCAT: A framework for analyzing the complexity of algorithms. In: *IEEE 8th International Symposium on Intelligent Systems and Informatics,* 2010.

[19] At *http://www.cs.princeton.edu/~appel/graphdata*

[20] Henry Kautz, Ashish Sabharwal, Bart Selman. Incomplete Algorithms. In: *The Handbook of Satisfiability – Chapter 6, IOS Press,* 2009.

[21] Diane J. Cook, Lawrence B. Holder. Graph-Based Data Mining. In: *IEEE Intelligent Systems - Volume 15 Issue 2,* 2000.

[22] Peter Grünwald. A Tutorial Introduction to the Minimum Description Length Principle. In: *Advances in Minimum Description Length: Theory and Applications, MIT Press,* 2005.

[23] Lajos Rónyai, Gábor Ivanyos Ivanyos, Réka Szabó. Algoritmusok. In: *Typotex,* 2008.

[24] John H. Holland. Genetic Algorithms. In: *Scientific American,* 1992.

[25] Melanie Mitchell. An Introduction to Genetic Algorithms. In: *MIT Press,* 1998.

[26] Béla Bollobás. The chromatic number of random graphs. In: *Combinatorica, 8,* 1988.

[27] Zoltán Ádám Mann, Tamás Szép. A gráfszínezés bonyolultságának és a gráf statisztikus tulajdonságainak összefüggései. In: *Intelligens Rendszerek,* 2009.