Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Informált keresési stratégiák absztrakció alapú modellellenőrzésben

TDK dolgozat

Készítette:

Vörös Asztrik

Konzulens:

Szekeres Dániel
dr. Vörös András
dr. Molnár Vince

2022

Tartalomjegyzék

Kivonat i

Abstract ii

1. Bevezetés 1

2. Háttérismeretek 4
- 2.1. Formális verifikáció . . . 4
- 2.2. Vezérlési folyam automata . . . 4
- 2.3. Állapottér . . . 5
- 2.4. Állapottér absztrakció . . . 5
- 2.5. Keresési algoritmusok . . . 7
- 2.6. Absztrakt elérhetőségi gráf . . . 9
- 2.7. Ellenpélda-vezérelt absztrakció finomítás . . . 10

3. Hierarchikus A* algoritmus 12
- 3.1. A* beépítése a CEGAR hurokba . . . 12
- 3.2. Megismert távolságok beállítása ARG-ben . . . 15
- 3.3. Hierarchikus A* keresés teljes ARG kifejtéssel . . . 16
- 3.4. Hierarchikus A* keresés igény szerinti ARG kifejtéssel . . . 16
- 3.5. Hierarchikus A* keresés legközelebbi információval rendelkező providerrel . 17
- 3.6. Hierarchikus A* keresés heurisztika csökkentéssel . . . 18

4. Mérések 21
- 4.1. Megvalósítás és mérési környezet . . . 21
- 4.2. XCFA formális modelleken végzett mérések eredményei . . . 22
- 4.3. XSTS formális modelleken végzett mérések eredményei . . . 25
  - 4.3.1. EXPL_PRED_COMBINED absztrakció . . . 25
  - 4.3.2. PRED_CART és PRED_SPLIT absztrakció . . . 29
  - 4.3.3. Explicit változó absztrakció . . . 32

5. Összefoglalás és jövőbeli tervek 34

Köszönetnyilvánítás 36

Irodalomjegyzék 37

Kivonat

Az élet egyre több területén látnak el szoftveralapú megoldások kritikus funkciókat. Az autókban nagy számú beágyazott rendszer felügyeli a fékezést vagy éppen a kormányzást, a vasúti rendszerekben is egyre több mechanikai megoldást és funkctiót ültetnek át beágyazott szoftverre, de ugyanígy nem nehéz elképzelni, hogy egy modern légi járművön mennyire sok szoftver fut. A beágyazott rendszerek széles körű elterjedés azonban új problémákat is hoz magával, hiszen ezen rendszerek helyes működésének biztosítása egyre nehezebb feladat. A tesztelés egy hatékony eszköz hibák feltárására, azonban a szoftveralapú funkciók helyességének bizonyítására nem alkalmas. A modellellenőrzés módszere alkalmas arra, hogy automatizáltan bizonyítsuk a helyességét egy számítógépes alkalmazásnak, vagy felderítsük annak hibáit. A modellellenőrzés során bejárjuk a szoftver állapotterét és lehetséges hibákat keresünk benne.

Az állapottér mérete gyakran a program/rendszer leírásának méretében exponenciálisan nő, vagy akár végtelen is lehet, ha komplex adatokat használunk a programunkban. Egy hatékony megoldás a keresési feladat komplexitásának csökkentésére az absztrakció: az eredetileg vizsgált rendszer viselkedését konzervatívan közelítjük egy absztrakt modell létrehozásával. Az Ellenpélda Vezérelt Absztrakció Finomítás (Counterexaple Guided Abstraction Refinement - CEGAR) egy absztrakció alapú módszer, ami iteratívan finomítja az absztrakciót a modellellenőrzés során, amíg el nem jut a helyesség bizonyításáig vagy egy valódi ellenpélda megtalálásáig. Ennek a módszernek a hatékonysága nagyban függ az alkalmazott absztrakcióktól és az állapottér bejárása során alkalmazott keresési stratégiáktól.

Amennyiben a követelmény sérül, a modellellenőrző algoritmus által nyújtott ellenpélda segítségként tud szolgálni szoftveres hibakereséskor vagy rendszerterv javítása során. Azonban elengedhetetlen, hogy az ellenpélda a probléma valós gyökerére mutasson rá, ezért az ellenőrzést végző mérnökök számára a legrövidebb, legtömörebb ellenpélda szükséges.

A CEGAR algoritmusok szakirodalmában találhatunk példát mélységi és szélességi keresés alkalmazására, illetve számos más prioritás alapú stratégiára, melyek a modell előzetes elemzésén alapulnak. Viszont a keresés az absztrakciós séma minden iterációja során újraindul, így elveszítjük az előző keresések eredményeit, így az előző keresés során gyűjtött információ is elveszik.

Ezen munka célja, hogy a korábbi fázisok állapottérbejárása során gyűjtött információt kihasználva a finomabb állapottér bejárásakor informált keresést alkalmazva javítsunk a CEGAR alapú modellellenőrző algoritmusokon, mindeközben az ellenpélda méretének minimalizálását is biztosítva.

A fő kontribúciónk az újszerű Hierarchikus A* algoritmus, ami egy - a CEGAR hurkon belül található - intelligens keresési stratégia. Az algoritmus a jelenlegi absztrakt állapottér A* alapú bejárásához az előző absztrakciókat felhasználva számít egy heurisztikát. A javasolt algoritmus több különböző változatának kiértékeljük a hatékonyságát, és összehasonlítjuk az elterjedten használt keresési stratégiákkal. A méréseket a nyílt forráskódú Theta modellellenőrző-keretrendszerbe beépített prototípus implementációnkon hajtjuk végre.

Abstract

Model checking is an automated method to prove the correctness of or find errors in computer based applications. The core of model checking is an intelligent search for possible error states in the state space of the system.

The state space often grows exponentially in the size of the program/system description, or can be even infinite in the presence of complex data. Abstraction is an efficient technique to reduce the complexity of the search problem: we construct an abstract model that conservatively approximates the behaviors of the original system under analysis. Counterexample guided abstraction refinement (CEGAR) is an abstraction-based method which iteratively refines the abstraction during model checking until a proof of correctness or a counterexample is found. The efficiency of the method is heavily dependent on the applied abstractions and search strategies during state space exploration.

In practical applications, the output of model checking is often not only a verdict about correctness, but also a counterexample if the requirement is violated. This counterexample can serve as a hint for debugging software or improving the system design. However, it is crucial to focus the counterexample to the real root cause of error, so verification engineers require counterexamples of minimum size.

Depth first and breadth first search have already been applied in the literature of CEGAR algorithms, along with several priority-based strategies based on preliminary analysis of the model. However, the search is basically restarted in each iteration of the abstraction scheme, so we lose the results of the former search procedures, leading to a significant overhead.

In this work, we aim to improve the efficiency of CEGAR-based model checking algorithms by exploiting the information stored in coarser abstractions when exploring the state space of a finer one, while also ensuring that the size of the final counterexample is minimized.

Our main contribution is the novel Hierarchical A* algorithm, an intelligent search strategy inside the CEGAR loop. The algorithm utilizes previous abstractions to compute heuristics for the A*-based exploration of the current abstract state space. We evaluate the efficiency of multiple variants of the proposed algorithm and compare them to the standard search strategies. The numerical experiments are conducted using our prototype implementation in the open-source Theta model checking framework.

1. fejezet

Bevezetés

Az elmúlt időszakban hatalmas fejlődésen ment keresztül a technológia, ami által könnyen elérhetővé váltak a különböző beágyazott rendszerek. Ennek eredményeként az élet egyre több területén látnak el szoftveralapú beágyazott megoldások kritikus funkciókat. Az autókban nagy számú beágyazott rendszer felügyeli a fékezést vagy éppen a kormányzást, illetve a vasúti rendszerekben is egyre több mechanikai megoldást és funkciót ültetnek át beágyazott szoftverekre, de ugyanígy nem nehéz elképzelni, hogy egy modern légi járművön mennyi funkciót szoftveres megoldások látnak el. A beágyazott rendszerek széles körű elterjedése a kritikus rendszerekben azonban új problémákat is hoz magával, hiszen ezen rendszerek helyes működésének biztosítása egyre fontosabb, azonban a komplexitásuk növekedése miatt ez egy egyre nehezedő feladat. A tesztelés egy hatékony eszköz hibák feltárására, azonban a szoftveralapú funkciók helyességének teljeskörű bizonyítására nem alkalmas. Ha matematikailag be akarjuk bizonyítani a funkciók helyességét, akkor formálisan kell azokat verifikálnunk.

A modellellenőrzés egy automatikus módszer a beágyazott szoftveralapú rendszerek helyességének bizonyítására, amely komplex matematikai bizonyító algoritmusokat használ a szoftver lehetséges viselkedéseinek a felderítésére és a helyesség belátásra, vagy amennyiben hibás a rendszer, akkor a hibák felderítésére. A modellellenőrzés során a szoftver és a helyességet meghatározó követelmény matematikai (formális) modelljét készítjük el először, majd a formális modell által generált állapotteret különböző redukciós és kereső algoritmusokkal derítjük fel követelményt sértő állapotok megtalálása érdekében.

Azonban az állapottér mérete gyakran a program/rendszer leírásának méretében exponenciálisan nő, vagy akár végtelen is lehet, ha komplex adatokat használunk a programunkban. Egy hatékony megoldás a keresési feladat komplexitásának csökkentésére az absztrakció: az eredetileg vizsgált rendszer viselkedését konzervatívan közelítjük egy absztrakt modell létrehozásával. Az Ellenpélda Vezérelt Absztrakció Finomítás (Counterexample Guided Abstraction Refinement - CEGAR) egy absztrakció alapú módszer, amely iteratívan finomítja az absztrakciót a modellellenőrzés során, amíg el nem jut a helyesség bizonyításáig vagy egy valódi ellenpélda megtalálásáig. Ennek a módszernek a hatékonysága nagyban függ az alkalmazott absztrakcióktól és az állapottér bejárása során alkalmazott keresési stratégiáktól[8].

Amennyiben a követelmény sérül, a modellellenőrző algoritmus által nyújtott ellenpélda segítségként tud szolgálni[11] szoftveres hibakereséskor vagy rendszerterv javítása során. Azonban elengedhetetlen, hogy az ellenpélda a probléma valós gyökerére mutasson rá, ezért az ellenőrzést végző mérnökök számára a legrövidebb, legtömörebb ellenpélda szükséges[6, 15].

A CEGAR algoritmusok szakirodalmában találhatunk példát mélységi és szélességi keresés alkalmazására, illetve számos más prioritás alapú keresési stratégiára, melyek a modell előzetes elemzésén alapulnak[8]. Viszont a keresés az absztrakciós séma minden

iterációja során újraindul, így elveszítjük az előző keresések eredményeit, így az előző keresés során gyűjtött információ is elveszik.

Munkám célja, hogy egy hatékonyabb és robusztusabb keresési stratégiát dolgozzak ki, amely a CEGAR korábbi (absztraktabb) iterációinak állapottérbejárása során gyűjtött információt kihasználva a finomabb állapottér bejárásakor informált keresést alkalmazva javítani tud a CEGAR alapú modellellenőrző algoritmusokon, mindeközben az ellenpélda méretének minimalizálását is biztosítva. Célom a munkámmal támogatni a mérnököket nem csak a helyesség bizonyításában, hanem a hatékonyabb hibakeresésben is a fókuszáltabb ellenpéldák nyújtásával.

Munkám eredménye az újszerű Hierarchikus A* alapú algoritmus, ami egy - a CEGAR hurkon belül található - intelligens keresési stratégia. Az algoritmus a jelenlegi absztrakt állapottér A* alapú bejárásához az előző absztrakciókat felhasználva számít egy heurisztikát. Az algoritmusnak több változata is elkészült, amelyek különböző stratégiák mentén dolgoznak. Elméleti eredményem az algoritmus kidolgozásán kívül az algoritmus helyességének bizonyítása. Az algoritmusok a nyílt forráskódú Theta verifikációs keretrendszerben[1] kerültek implementálásra, így bárki számára elérhetővé tettem a munkám eredményeit[2].

Emellett megvizsgáltam a javasolt új algoritmus család gyakorlati alkalmazhatóságát is: a javasolt algoritmus változatoknak kiértékeltem a hatékonyságát, és összehasonlítottam az elterjedten használt keresési algoritmusok hatékonyságával különböző benchmark készleteken. Az új megközelítés skálázódás és futásidő szempontjából is kompetitív a meglévő megoldásokkal, miközben garantáltan képes a legrövidebb ellenpéldát nyújtani a mérnökök számára.

Kapcsolódó munkák Munkám újdonsága, hogy komplex, informált keresési stratégiákat kombináltam absztrakció alapú megközelítésekkel. Ezen a területen egyszerűbb keresések (szélességi és mélységi keresés), vagy statikusan heurisztikát számolt keresési algoritmusok voltak csak korábban elérhetőek[7, 5, 13]. De a heurisztika számítás történhet akár egy absztraktabb probléma megoldásásval[12]. Ezekkel kapcsolatban az elmúlt években született egy nagyon alapos összefoglaló[8], amelynek továbbfejlesztéseként értelmezhető a munkám.

Komplex keresési algoritmusokat általában explicit modellellenőrzőkben használtak ezen a területen, mivel ott rendelkezésre áll az állapottér explicit gráf reprezentációja, az állapottér nincs szimbolikusan elkódolva, mint a CEGAR-alapú megoldások esetén.

Munkám alapvetően ezen korábbi kutatások tapasztalataira épít. A dolgozat elkövetkező részei a következőképpen épülnek fel:

- Háttérismeretek, 2. fejezet: Ebben a fejezetben a munkámhoz kapcsolódó fogalmakat fogom részletezni, amelyre építek az algoritmus ismertetése és helyességének bizonyítása során. Mivel az algoritmus a CEGAR hurokba épül bele, ezért az ahhoz kapcsolodó fogalmak ismertetésére is sor kerül.

- Hierarchikus A* algoritmus, 3. fejezet: Ennek során kerül ismertetésre, hogy az A* algoritmus hogyan használja fel a korábbi iterációjú állapottereket a keresés hatékonyabbá tételéhez. Ezután a Hierarchikus A* változatok kerülnek ismertetésre, illetve azok céljai, előnyeikkel és hátrányaikkal.

- Mérések, 4. fejezet: A mérési környezet és tesztadatok meghatározása, majd a Hierarchikus A* változatok és a területen már elterjedt keresési algoritmusok, a mérés által meghatározott hatékonyságaiknak összehasonlítása.

[1]Theta verifikációs keretrendszer: `github.com/ftsrg/theta`

[2]Munkám forkolt Thetába beépítve, mely későbbiekben mergelésre kerül: `github.com/asztrikx/theta/tree/astar`

- Összefoglalás és jövőbeli tervek, 5. fejezet: A dolgozatban foglaltak összefoglalása, illetve bemutatja a munka folytatásának tervezett irányait.

2. fejezet

Háttérismeretek

Ebben a fejezetben azokat a fogalmakat, algoritmusokat és adatszerkezeteket mutatom be, melyeket az általam elkészített munka felhasznál vagy annak tárgyalása során előkerülnek.

2.1. Formális verifikáció

Szoftverfejlesztési folyamatok során a hibák kiszűrésére bevált módszer a tesztesetek írása. Azonban ezek csak egyes lefutásokat vizsgálnak meg, így nem garantálják, hogy a tesztesetek által ellenőrzött követelmény minden lefutás esetén teljesül. Biztonságkritikus beágyazott rendszerek fejlesztésekor azonban elengedhetetlen, hogy biztosítani tudjuk a rendszer helyes működését, mivel annak hiánya súlyos következményekkel járhat, akár emberi életeket is veszélyeztethet.

Formális verifikációt alkalmazva matematikailag be tudjuk bizonyítani, hogy nincs olyan lehetséges állapotsorozat, azaz *lefutás*, amely sértené az ellenőrzés céljául adott követelményt. Ehhez szükségünk van formális modellekre és formális követelményekre. A formális modellek a rendszerünket reprezentálják a formális verifikálást végző modellellenőrző számára értelmezhető módon. A formális követelmények megadják, hogy milyen állításoknak kell igaznak lenniük a rendszer egy állapotára, azaz kijelölik a helyes és hibás állapotokat. A formális verifikáció kimenete vagy a rendszer helyességének a ténye, vagy a követelmény sérülése esetén az ahhoz tartozó lefutás, amellyel eljutottunk a követelményt sértő állapotba. A követelményt sértő lefutás megadásának célja, hogy az ellenőrzést végző mérnökök azt megvizsgálva ki tudják deríteni a hiba okát. Mivel a hiba előfordulásához nem hozzájáruló állapotok nehezítik az ellenpélda értelmezését és feldolgozását, ezért célszerű minimalizálni a követelményt sértő lefutás hosszát.

2.2. Vezérlési folyam automata

Szoftver formális verifikálása során gyakran *vezérlési folyam automata (Control Flow Automaton, CFA)* formális modellt generálunk a forráskódból, és annak struktúráját felhasználva végezzük el az ellenőrzést.

Definíció 1 (CFA). A CFA egy (L, V, D, E, l_0) gráf

- L: Helyek halmaza, amelyek az egyes elemi programkód utasítások végrehajtás előtti struktúrális állapotokat írják le, gráfunkban csúcsokkal reprezentálva. Speciális helyek a program végét és elejét jelző helyek.
- V: Programkódban szereplő változók halmaza, $V = v_1, \ldots, v_N$.

- D: Változók értelmezési tartományainak halmaza, $D = d_1, \ldots, d_N$, ahol d_i v_i-nek az értelmezési tartománya.
- E: Helyek közötti átmenetek halmaza, $E \subseteq L \times Ops \times L$, gráfunkban élekkel reprezentálva, $Ops = Actions \cup Guards$.
- $Actions$: Olyan elemi utasítások halmaza, melyek lefutása megváltoztatja V egyes elemeinek értékét.
- $Guards$: Olyan elemi utasítások halmaza, melyek V jelenlegi értékei alapján logikai értéket vesznek fel. Az adott $e \in E$ élet csak akkor járhatjuk be, ha V jelenlegi értékei alapján igazra értékelődik ki az adott $guard \in Guards$.
- l_0: Kiinduló hely, mely a program elejét jelzi, innen indulnak a lefutásaink. ▪

Tekintsük például az alábbi C kódot (2.1), aminek a formális modelljét CFA-ban adjuk meg (2.2), illetve formális követelmény legyen az, hogy az assert utasításokban található kifejezések igazra értékelődjenek ki.

```
int a = 16;
int b = 14;
while (b != 0) {
    assert(a >= b);
    int t = b;
    b = a % b;
    a = t;
}
```

2.1. ábra. C kód

2.3. Állapottér

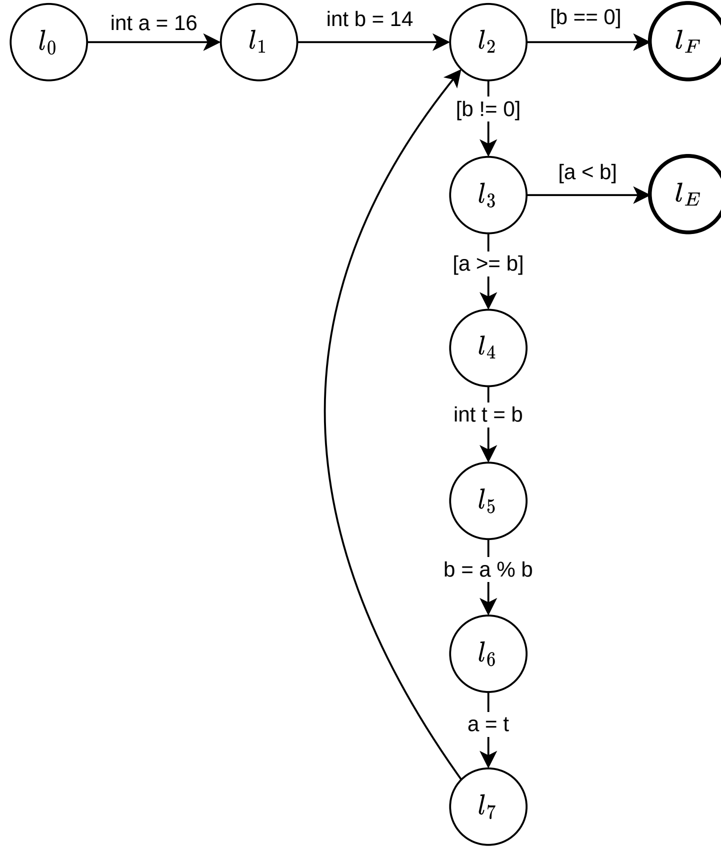
Az adott formális modell egyes lefutásai során az állapottérben közlekedünk, melyet a formális modell határoz meg. Az adott állapotokról a formális követelményt felhasználva megállapíthatjuk, hogy az *hibás állapot*-e.

Definíció 2 (Állapottér). Az állapottér egy (S, t) 2-es.

- S: Állapotok halmaza. CFA esetén $S = L \times d_1 \times \cdots \times d_{|V|}$, azaz a jelenlegi hely és változók értékei határozzák meg. Speciális eleme a kezdeti állapot, melyben a hely megegyezik a kezdeti hellyel, illetve a változók valamilyen alapértelmezett értékkel.
- $t : S \times Ops \rightarrow S$: Állapotok közötti átmenet függvény. $op \in Actions$ esetén a következő állapot változóinak értéke az utasításnak megfelelően megváltoznak, míg $op \in Guards$ esetén a változók értékei nem változnak. CFA esetén $t(s, op)$ rákövetkező állapot helye a E-nek azon elemének véghelye ahol a kiinduló hely s helyével egyezik meg, illetve operációja op-val egyezik meg. ▪

2.4. Állapottér absztrakció

A formális verifikáció során vizsgált programok lehetnek determinisztikusak és nemdeterminisztikusak. Egy egyszerű forrása lehet például a nemdeterminizmusnak a felhasználói bemenet. A nemdeterminisztikus programok formális ellenőrzése könnyen túl sok futásidővel vagy memóriával járhatnak, hiszen a lehetséges lefutások száma exponenciálisan nő



2.2. ábra. C kódhoz tartozó CFA

nemdeterminisztikus műveletenként. Például c db nemdeterminisztikus 32 bites egész szám esetén $(2^{32})^c$ féle lefutás lehetséges. Ezt a jelenséget állapottér-robbanásnak nevezzük.

Hatékony módszer lehet az állapottér méretének csökkentésére az absztrakció, mely során csak az ellenőrzés szempontjából fontos információkat tartjuk meg, így egy egyszerűbb, kompaktabb állapottér reprezentációt hozhatunk létre.

Definíció 3 (Állapottér absztrakció). Az állapottér absztrakció egy $(S, \preceq, \Pi, t)$ 4-es

- S: Absztrakt állapotok halmaza. Egy absztrakt állapotnak megfeleltethető több nem absztrakt (*konkrét*) állapot. Két speciális eleme a $\top$, mely minden konkrét állapotot reprezentál, illetve a $\bot$, amelyik egyet se.
- $k : S \times 2^{S_{konkrét}}$: Konkretizáló függvény, mely egy absztrakt állapothoz megadja az általa reprezentált konkrét állapotokat.
- Π: Pontosság, amely az absztrakció által megtartott információt írja le. A konkrét objektum, ami a pontosságot leírja, az alkalmazott absztrakció típusától függ.
- $t : S \times Ops \rightarrow 2^S$: Átmenet függvény, amely a jelenlegi absztrakt állapot és egy operátor segítségével a pontosságot figyelembevéve meghatározza a rákövetkező absztrakt állapotokat. $t(s, op) \supseteq \bigcup_{k \in k(s)} t_{konkrét}(k, op)$, azaz op esetén egy absztrakt állapotból kimenő élek szuperhalmaza a konkrét állapotaiból op esetén kimenő élek uniójának.

▪

Absztrakt állapotok esetén is megkülönböztetjük a hibás állapotokat.

Definíció 4 (Hibás absztrakt állapot). Egy S absztrakt állapotot hibásnak nevezünk, ha $\exists s \in k(S) : s$ sérti a követelményt. ▪

Az absztrakt állapottér definíciója alapján megadhatunk egy részbenrendezést az absztrakt állapotok között.

Definíció 5 (Absztrakt állapotok részbenrendezése). $\preceq \subseteq S \times S$: $A \preceq B$, akkor eleme a relációnak ha $k(A) \subseteq k(B)$. Minden $A \in S$-re igaz, hogy $A \preceq \top$ és $\bot \preceq A$. ▪

Absztrakciót alkalmazva kisebb állapotteret hozhatunk létre, mellyel az adott formális verifikáció akkor is megoldható lehet, ha a konkrét állapottér kezelhetetlenül nagy vagy végtelen.

Definíció 6 (Absztrakt lefutás konkretizálása). n hosszú $L_{absztrakt} \in (S_{absztrakt})^n$ absztrakt lefutás konkretizálása alatt egy olyan $L_{konkrét} \in (S_{konkrét})^n$ állapotsorozatot értünk, ahol $\forall i \in 1 \dots n : L^i_{konkrét} \in k(L^i_{absztrakt})$ és $\forall i \in 1 \dots (n-1) : (L^i_{konkrét}, L^{i+1}_{konkrét}) \in E_{konkrét}$. ▪

Az absztrakció egy másik fontos tulajdonsága, hogy a lehetséges lefutásokat konzervatívan felülbecsli, azaz úgy enged meg más nem konkretizálható lefutásokat, hogy az összes konkrét lefutást is megengedi[4].

Több információt megtartó pontosságot választva *finomítjuk* az absztrakciót, mely explicit változó, illetve predikátum absztrakció esetén a pontosságot leíró halmazhoz való új elem hozzáadását jelenti. Ekkor az absztraktabb állapottérben az egy állapottal leírt konkrét állapotok nem feltétlen reprezentálhatók ugyanúgy egy állapottal a finomabb állapottérben, hanem az új pontosságnak megfelelően akár több állapot is reprezentálja a konkrét állapotokat.

2.5. Keresési algoritmusok

A legkisebb méretű hibás lefutás megadásához az állapottérben meg kell találnunk a kezdőállapothoz legközelebbi hibás állapotot. Ezt a feladatot keresési algoritmusokkal tudjuk megoldani.

Mivel a keresési problémák általában gráfokon vannak definiálva, ezért hozzunk létre az állapottérnek megfelelő gráfot.

Definíció 7 (Elérhetőségi gráf (Állapottér gráf)). Az *elérhetőségi gráf (Reachability Tree, RT)* egy (N, L, E) 3-as.

- N: Gráfban lévő csúcsok halmaza. Speciális eleme a gyökér csúcs, melynek állapota a kezdőállapot, illetve azok a csúcsok melynek az állapota a formális követelmény által hibás állapotnak van megjelölve, a későbbiekben csak *célcsúcsok*.
- $L : N \to S$: Az adott csúcshoz tartozó állapotot megadó címkézés.
- $E \subseteq N \times Ops \times \{1\} \times N$: Gráfban lévő irányított élek halamaza, melyeknek a súlya, azaz a harmadik komponense azonos, hiszen az állapottérben nem teszünk különbséget az egyet átmenetek között.

  $$E = \{(n, op, 1, m) \mid n \in N, op \in Ops, s \in t(L(n), op), m \in N, L(m) = s\}$$

  , azaz két csúcs között akkor megy él adott *op*-val, ha a kiinduló csúcs állapotából *op*-t végrehajtva, az átmeneti függvény szerint érkezhetünk a végcsúcshoz állapotába.

Továbbá vezessük be a következő segédfüggvényeket:

- $e : N \to 2^E$: Adott csúcsból kimenő élek halmaza: $e(n) = \{e \mid e \in E \land e_1 = n\}$.

- $w : E \to \mathbb{Z}$: Adott él súlyát leíró komponensét adja meg: $w(e) = e_3$.
- $c : E \to N$: Adott él végcsúcsát adja meg: $c(e) = e_4$. ∎

A RT faként ábrázolja a lefutásokat, mégpedig úgy, hogy a lefutások azonos állapotokkal kezdődő részét azonos csúcsokkal reprezentálja, ezzel elkerülve az ismétlődő állapotsorozatokat, illetve hatékonyabbá téve a lefutások bejárását, hiszen így az azonos állapotokat elég egyszer meglátogatni.

Vezessünk be pár gráfhoz kapcsolódó fogalmat, hogy később könnyebben tudjunk hivatkozni annak egyes részeire.

Definíció 8 (Két csúcs távolsága). Egy olyan minimális hosszú út hossza, melynek első élének kezdeti csúcsa megegyezik az első csúccsal, míg az utolsó élének végcsúcsa megegyezik a második csúccsal. ∎

Definíció 9 (Távolságfüggvény). A távolságfüggvény $d : N \to \overline{\mathbb{N}}$ megadja, hogy egy csúcs milyen messze van a legközelebbi célcsúcstól, azaz minden célcsúcstól való távolság minimuma. ∎

Ezek után már definiálhatjuk a keresési algoritmusokhoz kapcsolódó fogalmakat.

Definíció 10 (Keresési probléma). Az adott gráf bejárása a meghatározott célcsúcsok egy részének megtalálása érdekében kijelölt kezdeti csúcsoktól kiindulva. A mi esetünkben ezt egy kezdeti csúcsra és egy megtalálandó célállapotra egyszerűsítjük. Így ha a gráfbejárás megtalál a célállapotok közül egyet, akkor a keresési probléma kimenete az oda vezető út, egyébként pedig a tény, hogy nem elérhető a gráfban célcsúcs. ∎

Mivel a teljes állapottér tárolása és előállítása jelentős memóriát és időt vehet igénybe, ezért annak csak a szükséges részét állítjuk elő. Ezáltal lesznek olyan állapotok, amiknek a rákövetkező állapotai még nem lesznek elérhetőek, azaz az állapot *kifejtetlen* marad. Ezáltal az állapottér gráf is bővül csúcsokkal és élekkel a keresés során, kifejtve a *kifejtetlen csúcsok*at.

Definíció 11 (Keresési stratégia). A keresési stratégia a kifejtetlen csúcsok közül valamilyen algoritmus mentén kiválasztja, hogy melyik kerüljön kifejtésre. A keresési algoritmus futásidejét a megfelelő keresési stratégia megválasztása határozza meg. ∎

Definíció 12 (Keresési algoritmus). A keresési algoritmus egy keresési stratégiát iteratívan alkalmazva megoldja az adott keresési problémát. ∎

A keresési algoritmusokat informáltságuk szerint nem informált és informáltként csoportosíthatjuk.

Definíció 13 (Nem informált keresési algoritmusok). Olyan keresési algoritmusok, melyeknek a célcsúccsal kapcsolatos információjuk kimerül abban, hogy egy csúcs célcsúcsnak minősül-e, így a keresési stratégiák ennek hiányában döntenek. ∎

Nem informált keresési algoritmusok közé tartozik a *szélességi keresés (Breadth First Search, BFS)*, mely az csúcsokat egyenletesen fejti ki, azaz a már elért, kifejtetlen csúcsok a kiinduló csúcsoktól való távolságuk (*mélységük*) növekvő sorrendje alapján. Ha BFS-t futtatva megtalálunk egy célcsúcsot, akkor az oda vezető út minden csúcsához a talált célcsúcs lesz a legközelebb, ezáltal távolságfüggvény ezen csúcsokhoz tartozó értéke ismert lesz.

Ezzel szemben az A* egy olyan *informált keresési algoritmus*, amely felhasznál egy $h : N \to \mathbb{N}$ becslő függvényt (*heurisztikus függvény*, *heurisztika*), mely becslést ad a távolságfüggvény értékeire. A kifejtendő csúcsok sorrendezése a rajtuk kiértékelt $f = g + h$ függvény értékeinek növekvő sorrendje szerint történik, ahol $g : N \to \mathbb{N}$ a csúcsok mélységét (kiinduló csúcsoktól való távolságát) adja meg, f pedig annak a legrövidebb út hosszának a becslése, amely a kiinduló csúcsból indul és egy célcsúcsban ér véget érintve az adott csúcsot. Fontos megemlíteni, hogy $\forall n \in N : h(n) = 0$ esetén f csak a mélységtől függ, ami megegyezik a BFS kereséssel.

A heurisztikus függvény megválasztása tetszőleges lehet, azonban célunk formális verifikáció során, hogy a legközelebbi célállapotot találjuk meg, ezzel biztosítva a legrövidebb ellenpéldát.

Definíció 14 (Elfogadható heurisztika). Egy heurisztika akkor elfogadható, ha $\forall n : h(n) \leq d(n)$ ▪

Definíció 15 (Konzisztens heurisztika). Egy heurisztika akkor konzisztens, ha $\forall n \forall e \in e(n) : h(n) - h(c(e)) \leq w(e)$ és minden m célállapotra $h(m) = 0$. ▪

Tétel 1 (A* optimalitás gráf alapú keresés esetén). [9, 10] Ha a heurisztikánk konzisztens és nincsenek negatív körök a gráfban, akkor az A* keresési algoritmus által visszaadott út a legközelebbi célállapotban végződő út lesz. ▪

2.6. Absztrakt elérhetőségi gráf

Az *absztrakt elérhetőségi gráf (Abstract Reachability Graph, ARG)* [2] az absztrakt állapotterek (absztrakt) lefutásait gráfként reprezentáló tömör adatstruktúra, RT-hez hasonló módon.

Definíció 16 (ARG). Az ARG egy speciális gráf, (N, L, E, C) 4-es.

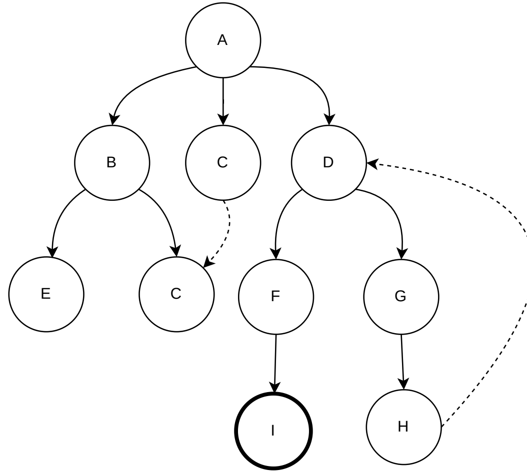
- N: Gráfban lévő csúcsok halmaza. Speciális eleme a gyökér csúcs, melynek állapota a kezdőállapot.
- $L : N \to S_{absztrakt}$: Az adott csúcshoz tartozó absztrakt állapotot megadó címkézés.
- $E \subseteq N \times Ops \times \{1\} \times N$: Gráfban lévő irányított élek halamaza.

  $$E = \{(n, op, 1, m) \mid n \in N, op \in Ops, s \in t_{absztrakt}(L(n), op), m \in N, L(m) = s\},$$

  azaz két csúcs között akkor megy él adott *op*-val, ha a kiinduló csúcs állapotából *op*-t végrehajtva, az átmeneti függvény szerint érkezhetünk a végcsúcshoz állapotába.
- $C \subseteq N \times \{0\} \times N$: Fedőélhalmaz, mely speciális irányított éleket tartalmaz. Egy N_1 és N_2 csúcs között akkor *lehet* felvenni fedőélet, ha $L(A) \preceq L(B)$ és N_1 nincs még kifejtve. ▪

Fedőéleket abban az esetben *vehetünk* fel, ha egy adott lefutás tartalmaz egy olyan absztrakt állapotot, mely eleme egy másik lefutásnak. Mivel azonos absztrakt állapotokból azonos lefutások érhetőek el, ezért az egyik absztrakt állapotból elérhető részgráfot törölhetjük és helyére egy kimenő fedőélet rakhatunk, aminek végcsúcsa megegyezik a másik absztrakt állapotot tároló csúccsal, ezzel megszűntetve a redundanciát, mégis megőrizve az állapotból elérhető lefutásokat. Mivel az absztrakt állapottér konzervatívan felülbecsli a lehetséges lefutásokat, így egy absztrakt állapot akkor is lehet a fedőél végcsúcsának

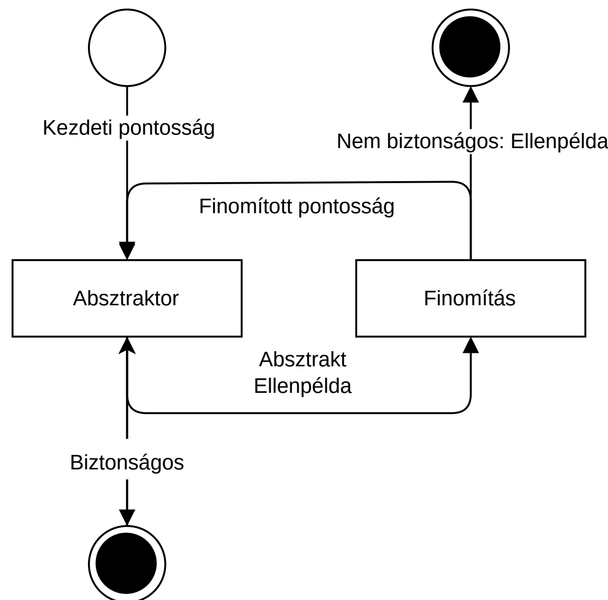
állapota, ha az konzervatívan felülbecsli az elérhető lefutásokat, azaz K kiinduló csúcs és V végcsúcs esetén $L(K) \preceq L(V)$. Mivel csak optimalizációs funkciót látnak el, amiatt a legrövidebb ellenpélda hosszába nem fognak beleszámítani, ezért a fedőéleket 0 súlyozással látjuk el.



2.3. ábra. Egy példa ARG. A fedőéleket szaggatott vonalakkal különböztetjük meg a normál élektől. Adott n csúcsok belsejébe írt betű hivatott reprezentálni a $L(n)$-t. A fedőélek alapján látható, hogy $C \preceq C$ és $H \preceq D$.

2.7. Ellenpélda-vezérelt absztrakció finomítás

Az absztrakció tulajdonságaira építve az *ellenpélda-vezérelt absztrakció finomítást (Counterexample-guided abstraction refinement, CEGAR)* [4] egy absztrakció alapú formális verifikációs algoritmust valósít meg. Az algoritmus felhasználja a már megismert ARG-t, illetve egy kereső algoritmust annak bejárása érdekében. Szintén szükségünk van egy állapottér absztrakcióra, ami pontosan meghatározza annak pontosságának számítását.



2.4. ábra. A CEGAR hurok

Az algoritmus elején kiindulunk egy kezdeti pontosságból.

Ezután belépünk az iteratív lépésbe. Az absztraktorban elindítunk egy keresést, valamilyen keresési algoritmus segítségével, ami hibás állapotot reprezentáló csúcs megtalálása esetén visszatér a lefutással, mint absztrakt ellenpélda. Mivel az absztrakt állapottér felülbecsli a lehetséges lefutásokat, ezért nem biztos, hogy a konkrét állapottérben is létezik az absztrakt ellenpéldához tartozó lefutás. Ezt a finomítási lépésben tudjuk eldönteni például úgy, hogy a lefutást logikai kifejezésekké alakítjuk, majd bemenetként egy *Satisfiability Modulo Theories (SMT)* megoldónak átadjuk, ami eldönti annak kielégíthetőségét, a mi esetünkben a lefutás konkretizálhatóságát. Ha az ellenpélda nem konkretizálható, akkor finomítunk a pontosságon azaz az absztrakció mértékét csökkentjük azért, hogy az így keletkező finomított absztrakciójú ARG-ben ne jussunk ugyanarra a nem konkretizálható lefutásra. Ezzel pedig kezdődhet újra az iteratív lépés, amit *CEGAR hurok*nak hívunk.

A modell helyességéről két eset alapján döntünk. Ha egy ARG-ben a keresési algoritmus nem talál hibás állapotot, akkor a modellünk helyes, hiszen ha a konkrét lefutások között létezne hibás lefutás, akkor a felülbecsült lefutások között is léteznie kell. A modellünkről pedig akkor mondhatjuk ki, hogy nem helyes, ha az absztrakt ellenpéldát sikeresen tudja a finomító konkretizálni.

3. fejezet

Hierarchikus A* algoritmus

Ebben a fejezetben mutatom be a munkám során kifejlesztett új algoritmusokat, azaz a hierarchikus keresési stratégiát a CEGAR hurokban. Emellett bizonyítom a bemutatott algoritmusok helyességét is.

3.1. A* beépítése a CEGAR hurokba

Az A* keresés CEGAR hurokba való beépítéséhez szükséges valamilyen heurisztika, mely becslést ad a legközelebbi hibaállapottól való távolságról az adott ARG-ben. Mivel célunk, hogy legrövidebb ellenpéldát találjuk meg, ezért konzisztens heurisztikát kell alkalmaznunk.

Tétel 2. Egy adott ARG távolságfüggvénye használható konzisztens heurisztikaként az ARG-ben. ▪

Bizonyítás. Indirekt tegyük fel, hogy nem konzisztens. Ekkor vagy $\exists m\ h(m) \neq 0$, ahol m célcsúcs, vagy $\exists n \exists e \in e(n) : d(n) - d(c(e)) > w(e)$. Mivel a távolságfüggvény értéke célcsúcsban 0, hiszen önmaga elérhető 0 távolságon belül, ezért csak az utóbbi eset lehetséges. Azt átrendezve: $d(n) > w(e) + d(c(e))$, azaz n csúcs egy gyerekén át elérhető egy célcsúcs, melynek távolsága kevesebb mint az $d(n)$, ami szintén nem lehet igaz, hiszen ellentmond a távolságfüggvény definíciójának. ▪

Azonban ha ismernénk az adott ARG távolságfüggvényt, akkor nem lenne szükség keresésre, hiszen pontosan tudnánk, mely éleken kell haladni ahhoz, hogy a legrövidebb célcsúcshoz vezető úton menjünk végig. Viszont egy absztraktabb, korábbi iterációjú ARG távolságfüggvénye rendelkezésünkre áll*hat*. Ahhoz azonban, hogy ezt fel tudjuk használni, be kell vezetnünk egy új fogalmat, mellyel kapcsolatot teremtünk egy i. és egy $(i+1)$. iterációjú ARG között.

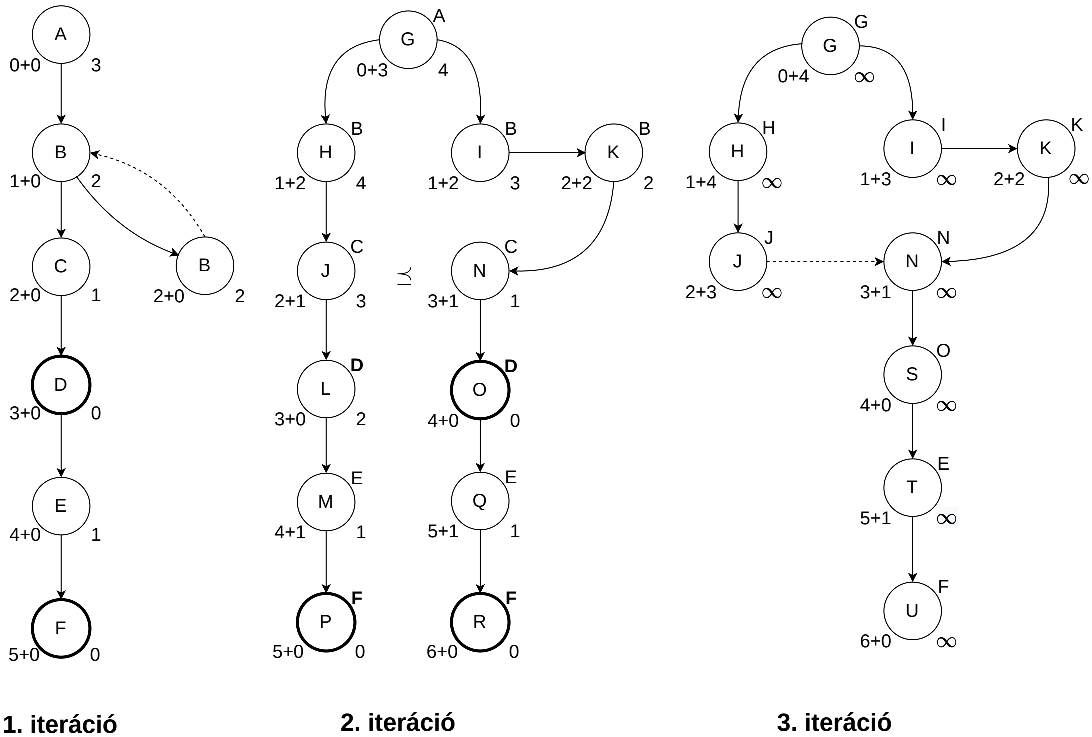
Definíció 17 (Tetszőleges provider). Jelölje A és B két egymást követő iterációban lévő ARG-t. Ekkor B szolgáltató függvénye $provider : N_B \to N_A$ egy tetszőleges függvény, amely B csúcsairól A csúcsaira képez le, illetve teljesíti a következő feltételt: $\forall n_b \in N_B : L_A(provider(n_B)) \succeq L_B(n_B)$, azaz egy olyan A-beli csúcs, melynek az állapota $\succeq$ a jelenlegi csúcs állapota.

Alapértelmezetten azonban struktúratartó providert fogunk használni, amelynek okát láthatjuk majd a 2 tételben.

Definíció 18 (Struktúratartó provider). Tetszőleges provider meghatározása esetén az összes korábbi iterációjú ARG-beli csúcs között keresünk, azonban ez $\mathcal{O}(|N|)$ futásidővel jár. Ennél hatékonyabb megoldás ha egy p_B szülővel rendelkező n_B csúcsnak a

providerét a szülő providerének gyerekei között keressük, azaz teljesülnie kell, hogy $\forall n_B \in N_B : L_A(provider(n_B)) \succeq L_B(n_B) \land \exists e_A \in e_A(provider(p_B)) : c(e_A) = provider(n_B)$. Gyökér csúcs esetén a provider A gyökere lesz. Ebben az esetben biztosítanunk kell, hogy $provider(p_B)$ rendelkezésre álljon, illetve ki legyen fejtve. ▪

A struktúratartó providernek az előnye az, hogy az általa elérhetővé tett A-beli távolságfüggvény értékek konzisztens heurisztikát alkotnak, ha a fedőélek behúzását korlátozzuk. Mivel a fedő élek csak optimalizációs céllal jönnek létre az adott lefutáshoz, így az él két csúcsának heurisztikája közötti különbség tetszőleges lehet. A fedőélek esetében a konzisztencia feltétel $h(p_B) - h(n_B) \leq 0$, vagyis $h(p_B) \leq h(n_B)$. Az alábbi ábrán egy ellenpélda található, ahol a fedőélek konzisztenciája nem teljesül, ha azt egyéb korlátozás nélkül vehetjük fel. A behúzott fedőél két csúcsának heurisztikája $h(p_B) = 3$ és $h(n_B) = 1$, ami nem konzisztens hiszen $3 \nleq 1$.



3.1. ábra. Az ábrán 3 különböző ARG iterációja látható. Az abc betűivel leírt állapotú csúcsok bal alsó sarkában $g + h$ található, a jobb alsó sarkában d, a jobb felső sarkában pedig a csúcs providere, illetve a célcsúcsok kiemelten szerepelnek. Ahogy az ábrán jelezve is van $L(J) \preceq L(N)$

Ezért az ARG kifejtése során a fedőél behúzását csak akkor engedjük meg, ha az nem sérti a konzisztens heurisztika által megkövetelt feltételeket. Ezzel a módosítással kimondhatjuk és bizonyíthatjuk a következő tételt.

Tétel 3. Egy absztraktabb ARG távolságfüggvénye alkalmas konzisztens heurisztikának egy finomabb ARG-ben a struktúratartó provider számítás segítségével, azaz legyen $h_B(n_B) = d_A(provider(n_B))$. ▪

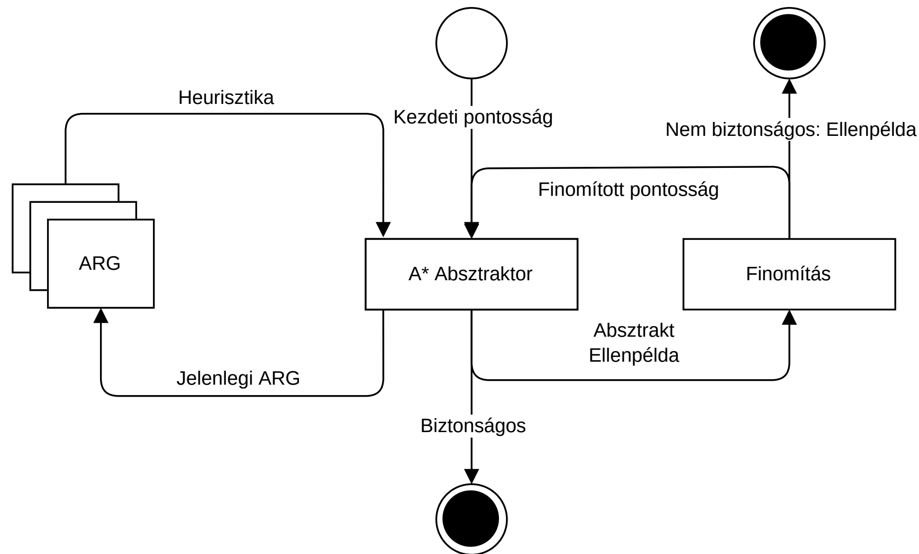
Bizonyítás. Ahhoz, hogy egy heurisztika konzisztens legyen, minden m_B célcsúcsra $h(m_B) = 0$ kell, hogy teljesüljön, illetve $\forall n_B \forall e_B \in e(n_B) : h(n_B) - h(c(e_B)) \leq w(e_B)$ kell, hogy igaz legyen.

Mivel $m_A \coloneqq provider(m_B) \succeq m_B$, ezért ha m_B célcsúcs volt, akkor annak absztrakt állapota tartalmazott követelményt sértő konkrét állapotot, így m_A absztrakt állapotának is tartalmaznia kell, tehát az is célállapot. Viszont ekkor $d_A(m_A) = 0$ kell, hogy teljesüljön, amiből következik, hogy $h_B(m_B) = 0$ teljesül, hiszen $h_B(m_B) = d_A(provider(m_B)) = d_A(m_A)$. Ezzel beláttuk, hogy az első követelmény teljesül ahhoz, hogy konzisztens heurisztikánk legyen.

A második követelmény teljesülésének ellenőrzését elég normál élekre belátnunk, hiszen fedőéleket csak akkor húzunk be, ha az nem sérti a heurisztika konzisztenciájára vonatkozó követelményeket. Tetszőleges normál él esetén legyen a kiinduló csúcs p_B és a végcsúcs n_B, ekkor $h(p_B) - h(n_B) \leq 1$ kell, hogy teljesüljön, hiszen távolságszámítás során a normál éleket egységnyi súlynak tekintjük. Behelyettesítve a heurisztika definícióját a következőt kapjuk: $d_A(provider(p_B)) - d_A(provider(n_B)) \leq 1$. A provider számítása miatt tudhatjuk, hogy a szülő és a gyerek providerei is szülő gyerek viszonyban állnak, ahol a gyerekek távolságfüggvénybeli értéke legfeljebb 1-gyel kevesebbek a szülőéhez képest, hiszen ellenkező esetben a távolságfüggvény nem lenne helyes. Tehát $d_A(provider(p_B)) - d_A(provider(n_B))$ legnagyobb értéke 1 lehet, így teljesül a követelmény. ∎

Mivel a későbbi iterációk számára az A* kereséssel szeretnénk előállítani a távolságfüggvényt, ezért nem csak az utolsó iteráció számára kell tudnunk konzisztens heurisztikát választani, amelynek a bekövetkeztét nem is ismerjük előre. Mivel az első iterációban nincs korábbi iteráció, ezért a távolságfüggvényen alapuló konzisztens heurisztika se áll rendelkezésünkre. Az A* keresési algoritmust vizsgálva megemlítettük, hogy $h(n) = 0$ heurisztikát választva BFS keresést kapjuk vissza, ami szintén képes előállítani a távolságfüggvényt a későbbi iterációknak, így első iterációban ezt a heurisztikát fogjuk használni.

A már megismert CEGAR hurok így kibővül egy ARG tárolóval, amiben a jelenlegi ARG-hez képest korábbi iterációban létrejött ARG-ket tároljuk. Innen minden absztraktor futtatáskor elérhető a jelenlegi ARG-hez a leírt módon a heurisztika, illetve ide eltároljuk az ARG-t ha végeztünk az abban való kereséssel. A már látott CEGAR hurkot leíró ábra 2.4 a következőre módosul:



3.2. ábra. A* beépítve a CEGAR hurokba

A következő szekciókban az ARG kifejtésének mértékében különböző Hierarchikus A* változatokat vezetünk be és vizsgálunk meg:

- Hierarchikus A* keresés teljes ARG kifejtéssel

- Hierarchikus A* keresés igény szerinti ARG kifejtéssel
- Hierarchikus A* keresés legközelebbi információval rendelkező providerrel
- Hierarchikus A* keresés heurisztika csökkentéssel

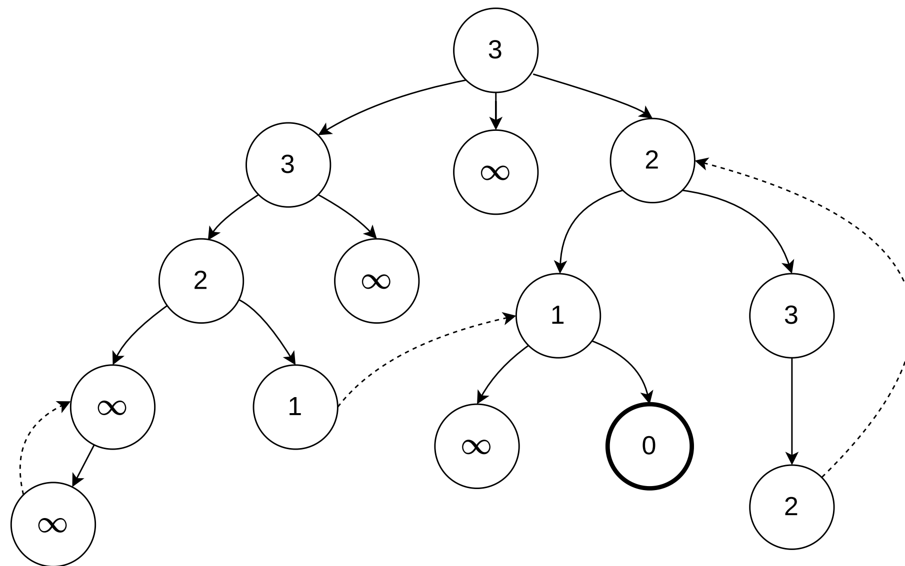
3.2. Megismert távolságok beállítása ARG-ben

BFS-nél említettük 2.5, hogy egy tetszőleges gráf esetén, ha csak az első célállapotig tart a keresés, akkor csak az oda vezető útnak a csúcsaira ismerjük meg a legrövidebb távolságot. Azonban az ARG egy speciális gráf, hiszen a fedőéleket leszámítva egy fa, illetve egy csúcs vagy le van fedve, vagy ki van fejtve.

Az utóbbi információt felhasználva tudhatjuk, hogy a vizsgált legrövidebb útba, ha belefed egy csúcs, akkor az máshonnan nem érhet el hibás állapotot, hiszen egyetlen kimenő éle a fedőél. (Normál, bemenő él csak egy lehet a legrövidebb úton, az is a szülőből indul ki, hiszen az ARG fedőélektől eltekintve fa.) Emiatt a fedőél kiinduló csúcsának is beállíthatjuk a távolságfüggvény értékét, ami meg fog egyezni a végcsúcsbeli értékkel, hiszen a fedőél 0 hosszúságú.

Az előbb leírt folyamatot meg tudjuk általánosan is fogalmazni azon csúcsokra, mely nem elemei az legrövidebb útnak. Minden olyan esetben, ha egy csúcs gyerekének megtudjuk a távolságfüggvény értékét és minden más gyerekének már ismert a távolságfüggvény értéke, akkor a csúcsnak is kiszámíthatjuk a távolságfüggvény értékét: $d(n) = \min_{e \in e(n)} d(c(e)) + 1$. Azonban vannak olyan esetek, amikor egy gyerek távolságának megismeréséhez szükséges, hogy a szülőnek a távolságát már ismerjük. Ez akkor fordulhat elő, ha a gyerek részgráfjának egy csúcsából kiinduló fedőél a kiinduló csúcs egy ősében végződik.

Az ilyen ágak hatékony felismerése és feloldása nem triviális. A teljes kifejtéssel történő A* keresés során szükséges ennek megoldása, azonban a többi esetben úgy tekintjük, hogy az ilyen szülők távolsága még ismeretlen, melyet az adott A* keresés a saját algoritmusával le fog tudni lekezelni.



3.3. ábra. A már látott ARG kitöltve az egy célcsúcs által megismert távolságokkal.

3.3. Hierarchikus A* keresés teljes ARG kifejtéssel

Az első módszer arra, hogy biztosítsuk egy ARG esetén a későbbi iterációban lévő ARG-k által igényelt távolságfüggvény értékek rendelkezésre állnak az az, hogy teljesen kifejtjük az adott ARG-t. Ekkor nem állunk meg egy célcsúcs elérésekor, hanem addig fejtjük ki a gráfot, amíg van kifejtetlen csúcs. Ennél a módszernél, ha minden olyan csúcs távolságfüggvénybeli értékét beállítjuk, aminek a távolsága ismert, akkor a kimaradt csúcsok biztosan nem érnek el célcsúcsot, azaz távolságfüggvényüket végtelenre állíthatjuk.

Az előző szekcióban azonban láthattuk, hogy nem tudunk minden ismerhető távolságú csúcsnak a távolságát meghatározni. Azonban az ott tárgyalt megoldás azt az esetet kezelte, amikor egy célcsúcsot találunk meg és az az által megismerhető távolságokat állítottuk be. Teljes kifejtés esetén azonban az összes célcsúcsot megtaláljuk, ezért számíthatjuk úgy is egy csúcs távolságát, hogy az milyen távol van a hozzá legközelebbi célcsúcstól. Ezt a célcsúcsokból kiinduló BFS bejárással tudjuk megoldani, hiszen ha van út egy csúcs és bármelyik célcsúcs között, akkor a BFS bejárás meg fogja határozni a legkisebb távolságot hozzájuk tetszőleges gráf esetén, tehát az ősbe végződő fedőél esetén is.

Ennek a módszernek előnye, hogy egyszerűen implementálható, és egyszerű a heurisztika kiszámítása. Azonban költséges lehet minden csúcsot kifejteni minden iteráció során, annak ellenére, hogy absztrakt állapotteret járunk be. Bizonyos esetekben még az absztrakt állapottér mérete is végtelen lehet a használt absztrakciótól függően, ezért ilyen esetekben nem tudjuk használni ezt az algoritmust.

3.4. Hierarchikus A* keresés igény szerinti ARG kifejtéssel

Az igény szerinti A* keresés azt a tényt használja ki, hogy egy A* bejárás során nem feltétlen minden absztraktabb csúcs távolságfüggvényének értékét használjuk fel. Emiatt azokat csak akkor számolja ki, amikor azokra egy későbbi A* keresés során a keresési stratégiának szüksége van.

Az algoritmus során csak addig fejtünk ki egy gráfot, amíg el nem érünk egy célcsúcsot. Egy iterációban történő A* keresés során, ha egy n csúcshoz szükségünk van $h(n)$-re (és nem az első iterációban vagyunk), de a $d(provider(n))$ nem ismert, akkor az előző iterációban lévő $provider(n)$-től indítunk egy másik A* keresést az első megtalált célcsúcsig, hiszen ekkor már $d(provider(n))$ ismert lesz.

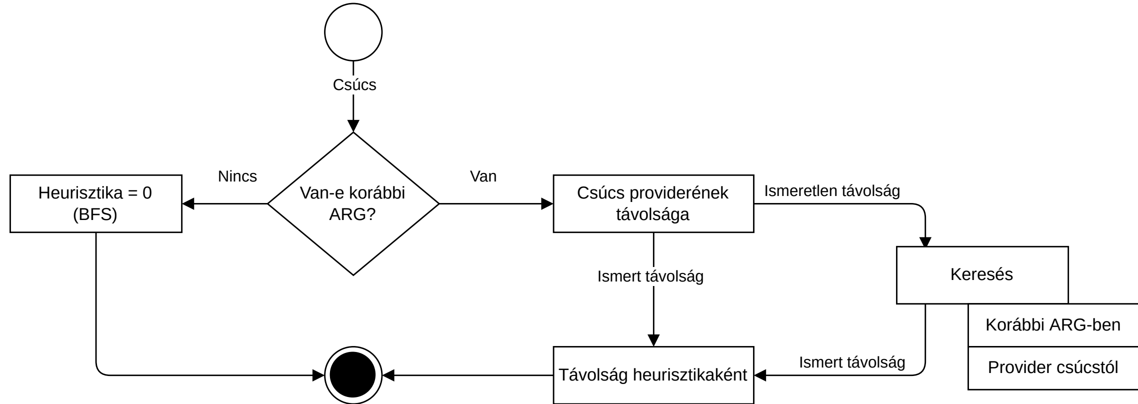
Mivel n csúcs p szülőjének már ismert kell legyen a heurisztikája, ezért egy A* keresés során meglátogattuk már a csúcsot. Ez a célcsúcs kivételével azt is jelenti, hogy a csúcs biztosan ki van fejtve, tehát $provider(n)$ is biztosan létezik. Ha módosítjuk a keresést úgy, hogy a megtalált célcsúcsok is kifejtésre kerüljenek, akkor mindig létezik $provider(n)$, így elegendő azzal foglalkozni, hogy $d(provider(n))$ ismert-e.

Fontos megjegyezni, hogy miközben a $provider(n)$-től indított keresés zajlik, szintén találhatunk egy n_2 csúcsot, melynek heurisztikája nem ismert, mely egészen addig előfordulhat amíg el nem érünk az első iterációhoz, ahol minden csúcs heurisztikája ismert ($h(n) = 0$).

Amikor a jelenlegi iterációs A* kereséstől eltérő keresést hajtunk végre n-től, akkor a keresés során találkozhatunk olyan n_2 állapottal, hogy $d(n_2)$ már ismert. Ebben az esetben lekorlátozhatjuk a keresés mélységét $k := g(n) + d(n_2)$-re hiszen ezáltal tudjuk, hogy van elérhető célcsúcs ilyen távolsággal, tehát $d(n) \leq k$. Ha nem találunk ezen k korláton belül más célcsúcsot, akkor n csúcs $d(n)$ értéke alapján beállíthatjuk az n-et elérő csúcsok távolságfüggvénybeli értékét.

Előnye ennek a módszernek, hogy nem fejtjük ki teljesen az ARG-t, ami akár végtelen is lehet. Azonban mivel egy korábbi iterációban lévő ARG-ben a különböző csúcsok távolságszámításához mindig egy új A* keresést indítunk a kérdéses csúcstól, ezért az új-

raindított keresések során többször is elérhetjük ugyanazt a csúcsot, és meglátogathatjuk annak szomszédait, ha a keresések során megtalált célcsúcsok alapján nem tudjuk beállítani annak távolságát. Viszont megfelelő implementáció mellett a csúcsok újrameglátogatása nagysárendileg kisebb költség, mint egy csúcs kifejtésének költsége.



3.4. ábra. A heurisztika előállításának módja

3.5. Hierarchikus A* keresés legközelebbi információval rendelkező providerrel

Ennek a módszernek a célja, hogy az igény szerinti kifejtés során előforduló korábbi iterációban található ARG-k kifejtése nélkül határozzunk meg heurisztikát. Azonban akkor, ha egy adott n csúcs szülőjének providere nem lett kifejtve, akkor n-nek nem létezik providere. Szintén, ha létezik $provider(n)$, de $d(provider(n))$ ismeretlen, akkor nem indíthatunk egy keresést $provider(n)$-ből hiszen az a korábbi ARG kifejtésével járna.

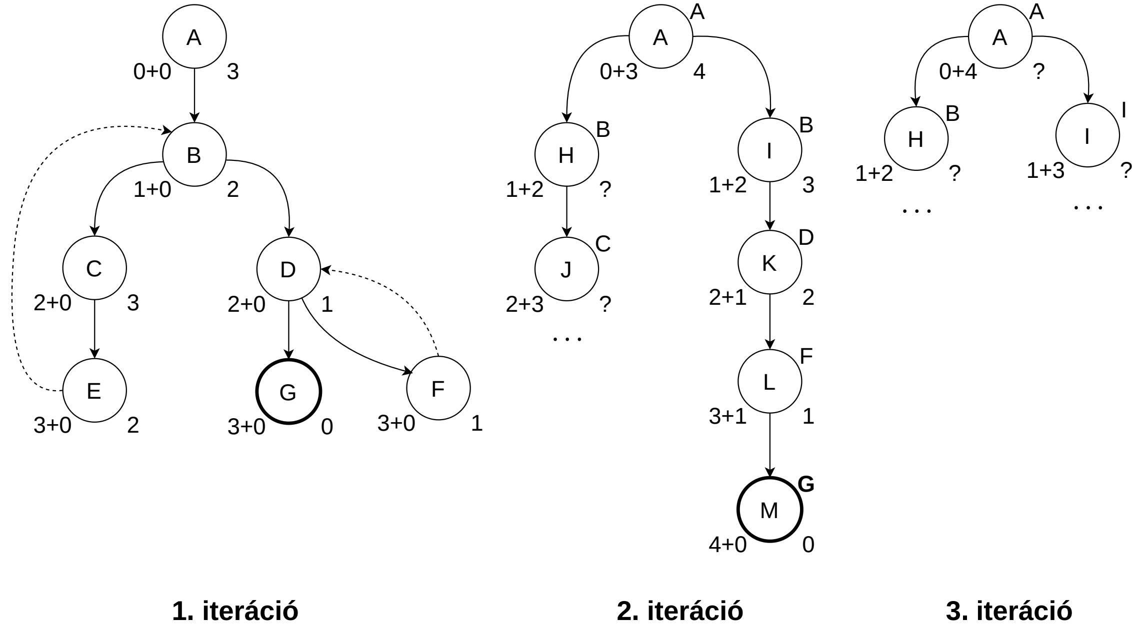
Ezt a provider meghatározásának módosításával oldhatjuk meg.

Definíció 19 (Legközelebbi információval rendelkező provider). A legközelebbi információval rendelkező provider meghatározása n csúcsra iteratívan történik n szülőjéből indulva. Először megvizsgáljuk, hogy az adott szülőnek providere létezik-e. Ha nem, akkor megáll az iteráció. Ezután azt vizsgáljuk meg, hogy létezik-e a providerének egy $n' \succeq n$ gyereke, melyre a távolságfüggvény értéke ismert. Ha igen, akkor azt állítjuk be heurisztikának, egyébként pedig az eljárást újrakezdjük a jelenlegi szülőtől. ▪

Az egyik problémája az új provider meghatározásnak, hogy ha egy csúcs egy gyerekének providere korábbi iterációbeli ARG-ben található, mint a csúcs providere, akkor gyerek providerének távolságfüggvénybeli értéke tetszőlegesen kisebb lehet mint a csúcsé. Azonban ez a heurisztika konzisztenciáját sértheti, hiszen azt a távolságfüggényből számoljuk. Az alábbi ábrán egy példát láthatunk, ahol az új provider definíciót használva nem konzisztens heurisztikát kapunk.

Egy másik probléma akkor fordul elő, ha nem találunk providert egy csúcsnak, hiszen ilyenkor heurisztikánk sem lesz. Bár erre az esetre láthatunk megoldást a Hierarchikus A* keresés heurisztika csökkentéssel szekcióban.

Az említett problémák miatt ez a változat nem került implementálásra. Helyette a korábbi iterációjú ARG-k továbbfejtését a következő szekcióban tárgyalt módon kerüljük el.



3.5. ábra. Az ábrán 3 különböző ARG iterációja látható. Az abc betűivel leírt állapotú csúcsok bal alsó sarkában $g + h$ található, a jobb alsó sarkában d, a jobb felső sarkában pedig a csúcs providere, illetve a célcsúcsok kiemelten szerepelnek. Az "..."-tal jelzett csúcsok még nem kerültek kifejtésre. Az utolsó iterációban látható, hogy H csúcsnak az előző iterációból megfelelő providert választani, így azt az első iterációból választottunk. Emiatt viszont A és H között nem konzisztens a heurisztika.

3.6. Hierarchikus A* keresés heurisztika csökkentéssel

Ez a módszer a legközelebbi információval rendelkező providert használó módszerhez hasonlóan a korábbi ARG-k kifejtését hivatott elkerülni. Ha bármilyen, az előbbiekben látott ok miatt nem tudunk a korábbi iterációjú ARG-ből n csúcsnak heurisztikát szerezni, akkor egyszerűen az ő p szülőjének heurisztikáját eggyel csökkentjük (kivéve, ha az már alapból 0), és azt állítjuk be heurisztikának.

Tétel 4. Az csökkentéssel meghatározott h' heurisztikák alulról becsülik a távolságfüggvény által meghatározott h' heurisztikákat. ▪

Bizonyítás. Jelölje n a gyerek csúcsot és p a szülő csúcsot. $provider(n)$ és $provider(p)$ jelölje azokat az előző ARG-beli csúcsokat, melyek kellő kifejtés esetén léteznének, illetve $d(provider(n))$ és $d(provider(p))$ a távolságfüggvénybeli értéküket. Ezek az értékek lehetnek végtelenek is, azonban nem fejtjük ki azon csúcsokat, melyeknek heurisztikája végtelen, hiszen ha az absztrakt állapottérben sem érhető el célcsúcs, akkor a konkrétban sem lesz, így azzal az esettel nem kell foglalkoznunk, ha $d(provider(p))$ végtelen.

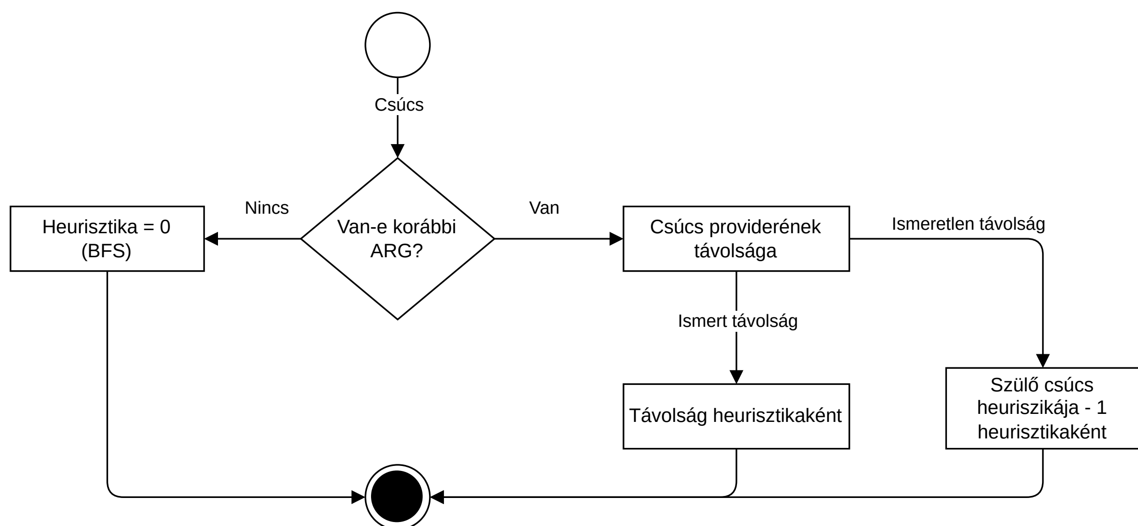
Mivel $provider(n)$ és $provider(p)$ vagy normál- vagy fedőéllel vannak összekötve, ezért vizsgáljuk külön ezeket az eseteket. Normál él esetén a távolságfüggvény tulajdonsága miatt $d(provider(n)) \geq d(provider(p)) - 1$, ami a heurisztika definíciókat figyelembe véve $h(n) = d(provider(n)) \geq d(provider(p)) - 1 = h'(n)$, tehát teljesül, hogy $h(n) \geq h'(n)$. Fedő él esetén $d(provider(n)) = d(provider(p)$, ahogy azt a Megismert távolságok beállítása ARG-ben szekcióban láttuk. A heurisztika definíciókat figyelembe véve $h(n) = d(provider(n)) = d(provider(p) \geq d(provider(p)) - 1 = h'(n)$, tehát ebben az esetben is teljesül. ▪

Tétel 5. A nem meghatározható heurisztikák esetén a szülő heurisztikáját eggyel csökkentve konzisztens heurisztikát kapunk a már ismert heurisztikákkal. ∎

Bizonyítás. A konzisztencia 1. feltételének teljesülését ellenőrizzük éltípusok szerint. A fedőéleket közötti konzisztenciát nem kell figyelembe vennünk, hiszen fedőélet csak konzisztenciát kielégítő csúcsok között veszünk fel. Normál éleket vizsgálva 4 eset lehetséges aszerint, hogy az kiinduló- és végcsúcsnak a heurisztikáját melyik heurisztika számítás alapján határoztuk meg. Jelölje K a kezdő csúcsot és V a végcsúcsot.

- K és V heurisztikája távolságfüggvény alapján: Ezt már beláttuk, hogy konzisztens lásd 2 tétel.
- K távolságfüggvény alapján vagy csökkentéssel és V csökkentéssel: $h(K) - h'(V) \leq 1$, ahol $h'(V) = h(K) - 1$, tehát $h(K) - (h(K) - 1) \leq 1$ kell teljesüljön, ami igaz.
- K csökkentéssel és V távolságfüggvény alapján: Ez az eset akkor fordulhat elő, ha a legközelebb található célcsúcshoz tartozó út fedőélen keresztül érte el a V csúcsot, hiszen normál éllel csak K-n keresztül érhette volna el, és akkor K heurisztikáját távolságfüggvény alapján határoztuk volna meg. A bizonyításhoz felhasználjuk az előbb belátott 4 tételt. Ehhez előbb rendezzük át azt a mindig teljesülő egyenlőtlenséget, melyet vizsgálnánk, ha mindkét csúcs heurisztikája távolságfüggvény alapján számítódna: $h(K) \leq 1 + h(V)$. Majd használjuk fel, hogy a csökkentéses heurisztika alulbecsüli a távolságfüggvény alapú heurisztikát: $h'(K) \leq h(K) \leq 1 + h(V)$. Azon tény miatt, hogy $h(K)$ értéke mindig teljesíti a jobb oldali egyenlőtlenséget, ezért $h'(K) \leq 1 + h(V)$ is mindig teljesül. Ezt visszarendezve megkapjuk a bizonyítandó egyenlőtlenséget: $h'(K) - h(V) \leq 1$.

A konzisztencia 2. feltételének bizonyítását heurisztika számítási módszerek szerint vizsgáljuk külön. Távolságfüggvényen alapuló heurisztikánál már láttuk, hogy m célállapotoknák $h(m) = 0$ a 2. bizonyításban. Mivel a csökkentésen alapuló heurisztikák alulbecsülik a távolságfüggvényen alapulókat, illetve annak értéke legalább nulla, ezért $h'(m)$ szintén 0 m célállapotokra. ∎



3.6. ábra. A heurisztika előállításának módja

A módszer előnye, hogy a heurisztikát konstans időben és könnyen elő tudjuk állítani, tehát nem foglalkozunk a korábbi ARG-k továbbfejtésével, azokban nem végzünk további

keresést. Azonban, ha korábbi ARG-ben kevés csúcsra ismerjük a távolságfüggvénybeli értékét, akkor sok helyen kell alkalmaznunk ezt a módszert, ami pontatlan becslést adhat, illetve kellő számú csökkentés után 0 lesz az értéke, ami a BFS kereséssel egyezik meg.

4. fejezet

Mérések

Ez a fejezet a dolgozatban javasolt Hierarchikus A* algoritmus különböző változatainak teljesítményének kiértékelését célzó méréseket és az azok eredményeiből levont következtetéseket tárgyalja.

4.1. Megvalósítás és mérési környezet

A Hierarchikus A* algoritmus változatokat a nyílt forráskódú moduláris Theta[1] modellellenőrző keretrendszerbe fejlesztettem, mely számos moduláris elemet tartalmaz, melyeket felhasználva új modellellenőrző algoritmusok implementálhatók. Ilyenek például az absztraktciók, vagy a finomítási eljárások. Emiatt a fejlesztésem egy új absztraktor (lásd 3.2), illetve keresési algoritmus megvalósításából állt.

A mérésekhez használt bemeneteket egyik részről a Software Verification Competition[1] (SV-Comp) benchmark halmaza, másik részről az egyetem által biztosított, többek között ipari partnerektől származó belső XSTS modellek[14] alkották. A mérések végrehajtásához a BenchExec[3] eszközt használtam, mely segít a felhasznált erőforrások megbízható mérésében.

A Theta modellellenőrző keretrendszer sok különböző konfigurációs paraméterrel rendelkezik. Mivel a méréseim célja a Hierarchikus A* keresés hatásának vizsgálata volt, emiatt a többi konfigurációs paramétereket a Thetán végzett korábbi mérések szerint jól teljesítő konfigurációknak megfelelően állítottam be (nem volt cél a teljes konfigurációs tér lefedése).

A mérés fő célja, hogy a Hierarchikus A* változatok hatékonyságát a Thetába már beépített BFS és ERR keresési algoritmusok hatékonyságával összevethető legyen. Az ERR keresés CFA struktúra alapján számít heurisztikát, a jelenlegi hely és a hibás hely távolsága alapján. Míg a BFS keresés biztosítani tudja számunkra a legrövidebb ellenpéldát, addig az ERR keresés heurisztikát használ, mely tulajdonságokat a Hierarchikus A* változatok ötvöznek.

Az egyes tesztesetekhez 15 perc időkorlát volt rendelve, így a következő szekciókban a sikeresen formálisan verifikált modellek számát (*megoldott tesztesetek száma*), illetve a BenchExec által generált quantile plot ábrákat fogom bemutatni, melyeken csak az időkorláton belül lefutott tesztesetek fognak szerepelni. A quantile plot ábrák az x tengelyen egységes térközönként a megoldott modellek száma található, míg az y tengelyen az eltelt idő található logaritmikus skálán. A quantile plot ábrák segítségével könnyen összehasonlíthatóak az egyes konfigurációk hatékonysága. Ha egy konfiguráció több tesztesetet old meg egy másikhoz képest, akkor annak görbéje nagyobb x értékeken is felvesz értékeket (az ábra "távolabbra elér"), illetve a tesztesetek számában való skálázódás is leolvasható.

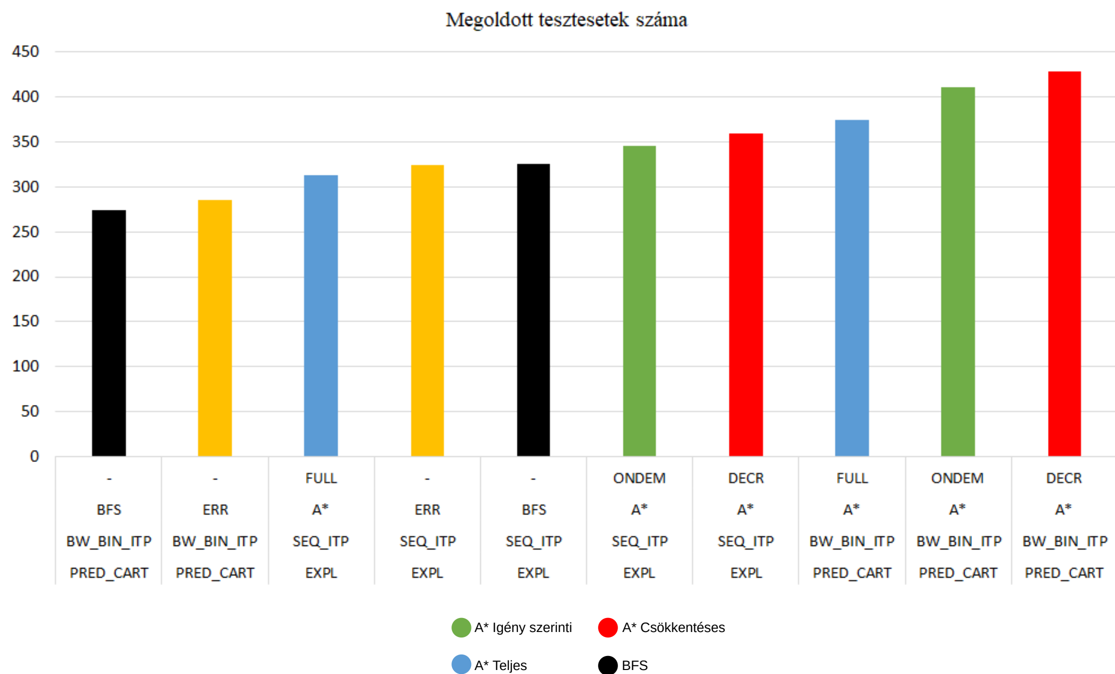
[1] Theta forráskódja: github.com/ftsrg/theta

4.2. XCFA formális modelleken végzett mérések eredményei

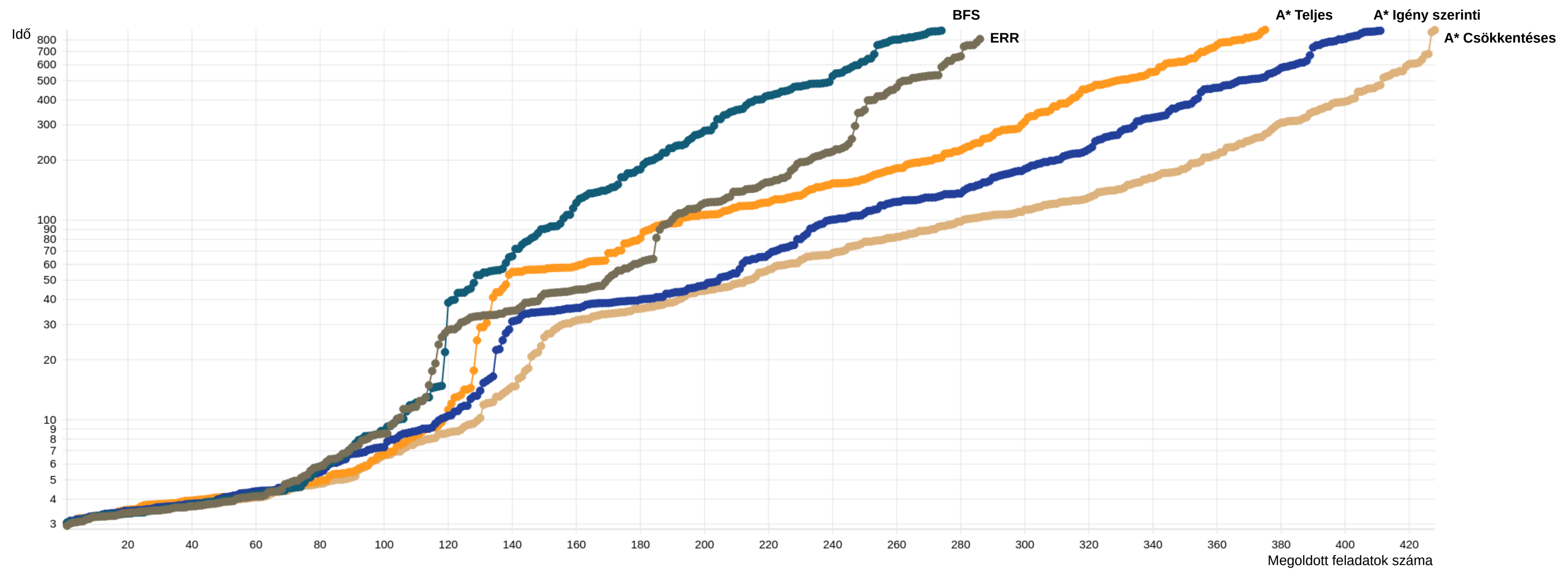
Az SV-Comp benchmark halmaz C programjaiból származtatott kibővített CFA (XCFA) formális modelleken a keresési algoritmusok hatékonyságát az alábbi konfigurációkkal mértem. Az alábbi táblázat egy sora részletezi egy konfigurációból a számunka érdekes konfigurációs elemeket, melyek minden keresési algoritmussal együtt alkotnak egy-egy konfigurációt.

#	Absztrakció	Finomítás
1	PRED_CART	BW_BIN_ITP
2	EXPL	SEQ_ITP

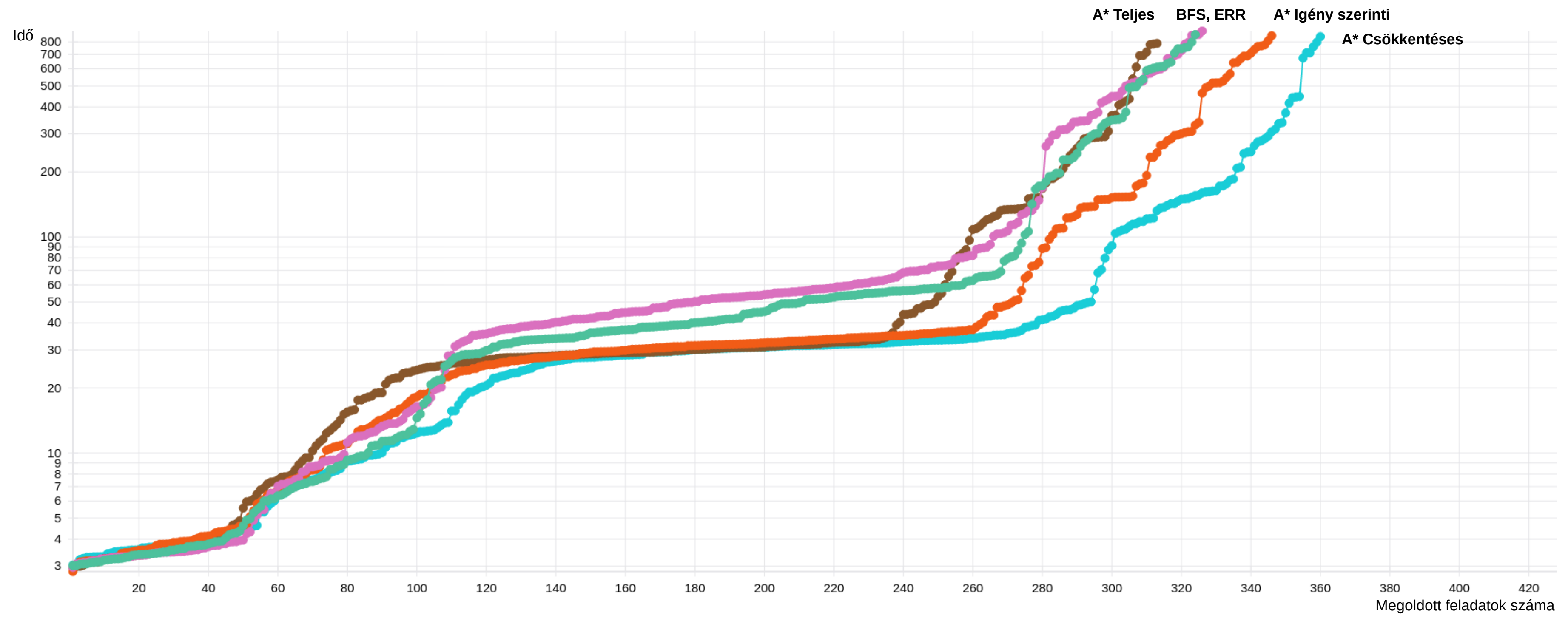
Az alább található keresési algoritmusokat konfiguránciónként összehasonlító ábrát áttekintve láthatjuk, hogy szinte minden esetben a Hierarchikus A* keresések több tesztesetet oldottak meg a jelenleg Thetába beépített BFS és ERR keresésekhez képest.



4.1. ábra. Az egyes konfigurációk által időkorláton belül megoldott modellek száma.



4.2. ábra. 1. konfiguráció quantile plot ábrája.



4.3. ábra. 2. konfiguráció quantile plot ábrája.

4.3. XSTS formális modelleken végzett mérések eredményei

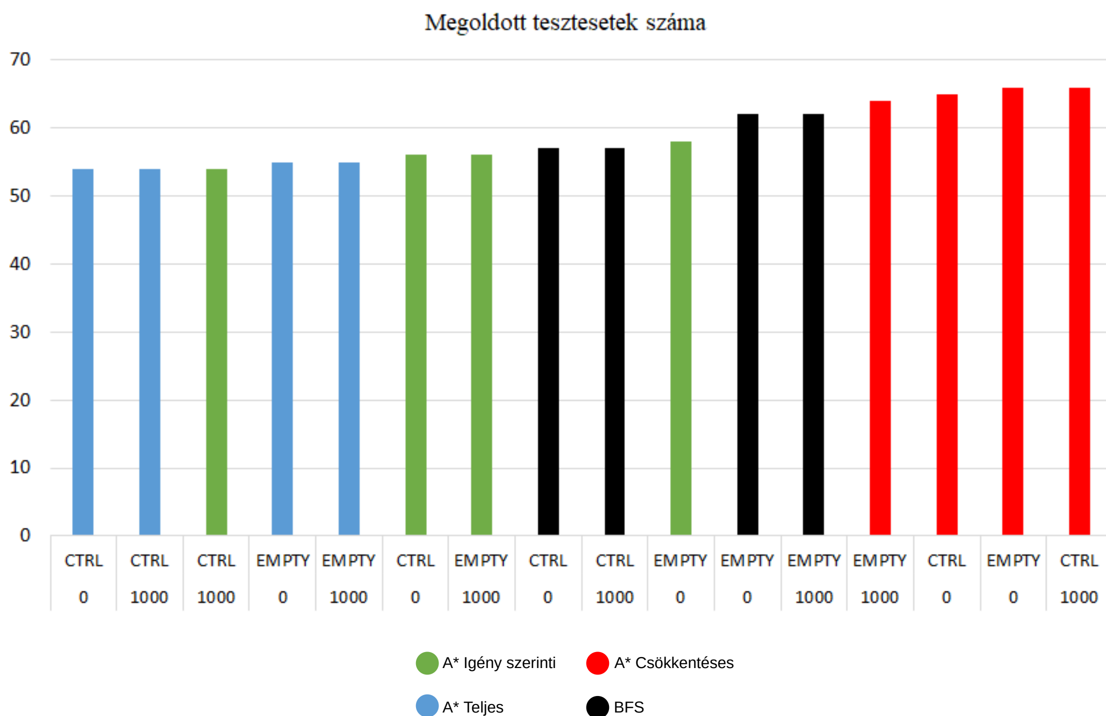
Az alábbiakban a Theta XSTS modellekből álló benchmark halmazán történt mérések eredményeinek bemutatása látható. Ezen formális modellek egy része különböző magasabb szintű állapot-alapú mérnöki modellekből származik, míg mások kézzel készültek tesztelési és teljesítménymérési céllal.

Az XCFA mérésekkel ellentétben a nagymennyiségű mért konfiguráció miatt az eredményeket absztrakt doménenként elkülönítve mutatják be a következő részek. Az explicit és kombinált explicit-predikátum domének esetében új konfigurációs paraméter a kifejtett gyerekek maximális száma, melynek célja, hogy elkerülje a végtelen méretű ARG-k létrehozását. Egy csúcs kifejtésekor legfeljebb ennyi új csúcsot hozunk létre, amennyiben ennél több lenne, akkor a megfelelő változó értékét ismeretlenre állítjuk az absztrakt állapotban. Másik új konfigurációs paraméter a kezdeti pontosság: XSTS modellekben egyes változók megjelölhetőek kontroll változóként, melyek értékét egyes esetekben érdemes explicit számon tartani már a kezdeti absztrakcióban is.

4.3.1. EXPL_PRED_COMBINED absztrakció

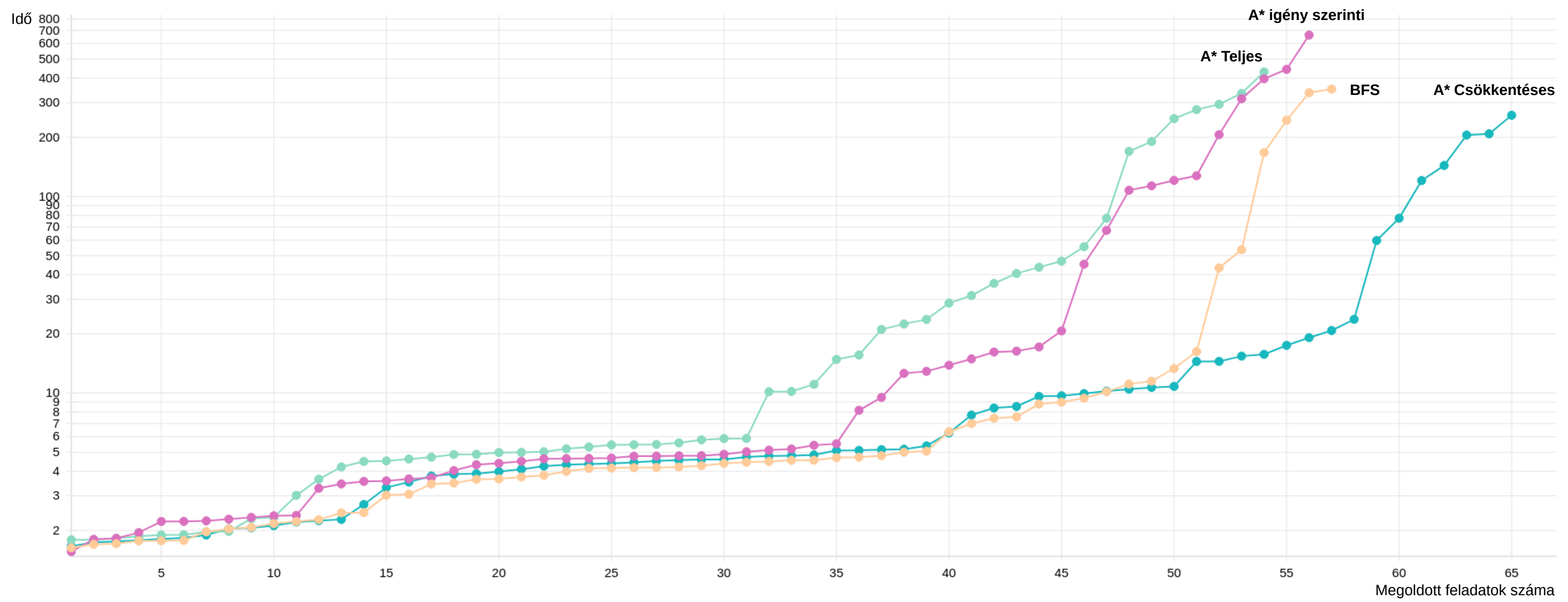
Az alábbi táblázatban található konfigurációkon kimérve a keresési algoritmusok hatékonyságát azt láthatjuk, hogy míg a csökkentéses Hierarchikus A* keresés jobban teljesített a BFS-nél, addig a többi változat rosszabbul. Azonban a mérés során beállított időkorlát mellett a megoldott tesztesetek száma a keresések között nem mutat nagy eltérést.

#	Finomítás	Kifejtett gyerekek száma	Kezdeti pontosság
1	SEQ_ITP	∞	CTRL
2	SEQ_ITP	∞	EMPTY
3	SEQ_ITP	1000	CTRL
4	SEQ_ITP	1000	EMPTY

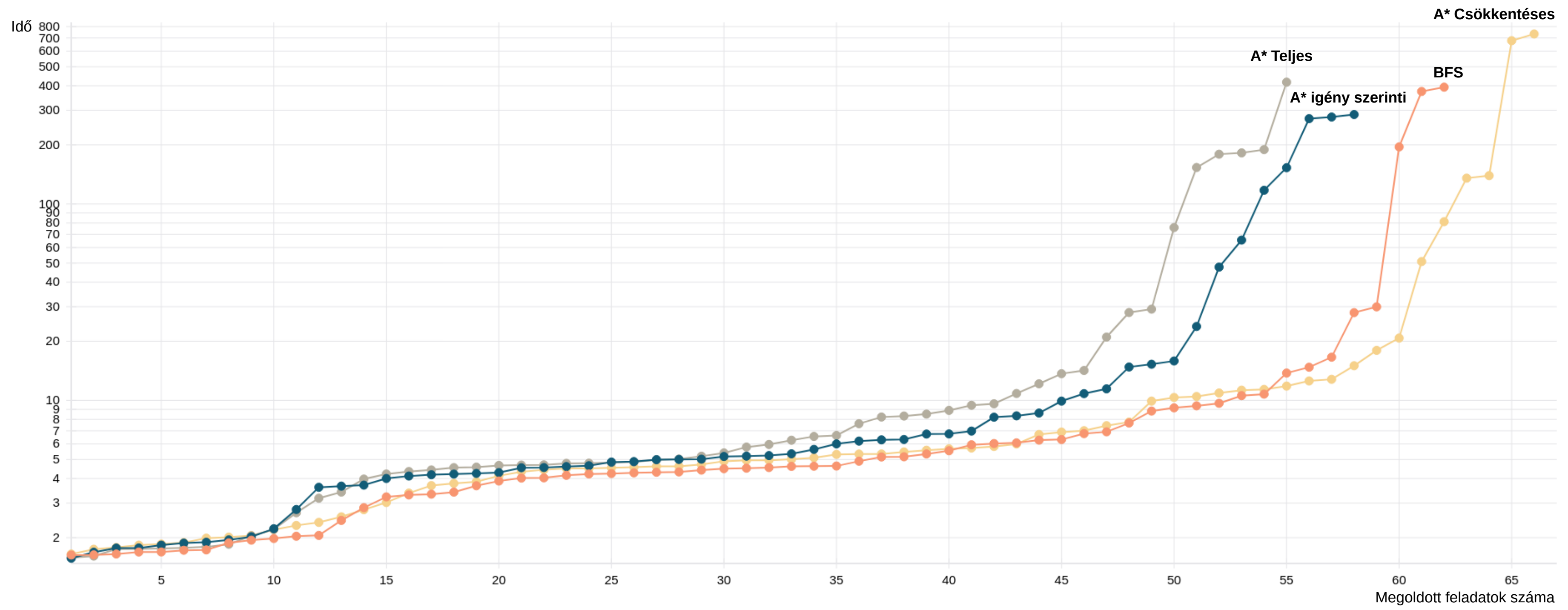


4.4. ábra. Az egyes konfigurációk által időkorláton belül megoldott modellek száma. Az x tengely 1. sorában a kezdeti pontosság, míg a 2. sorában a kifejtett gyerekek száma látható.

A következő ábrákon az áttekinthetőség érdekében a kifejtett gyerekek számából csak a csak végtelen beállításúakat ábrázoltuk.



4.5. ábra. 1. konfiguráció quantile plot ábrája.



4.6. ábra. 2. konfiguráció quantile plot ábrája.

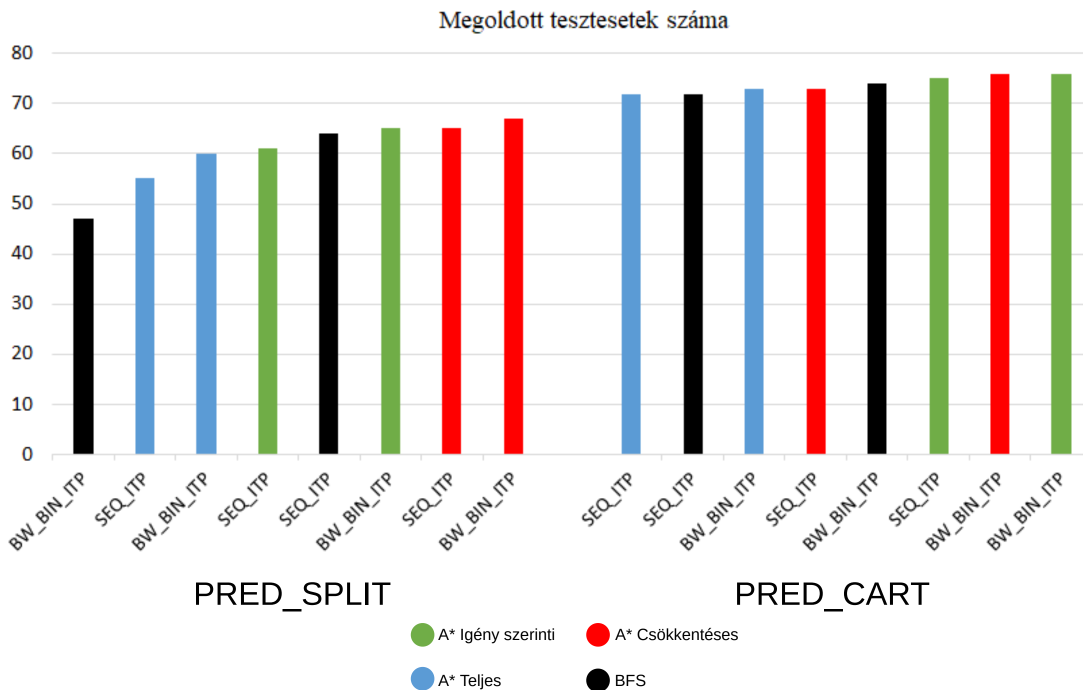
4.3.2. PRED_CART és PRED_SPLIT absztrakció

Hasonlóan az eddigiekhez a keresési algoritmusokat kimérve az alábbi konfigurációkon a 4.7 ábrán találjuk a megoldott tesztesetek számát.

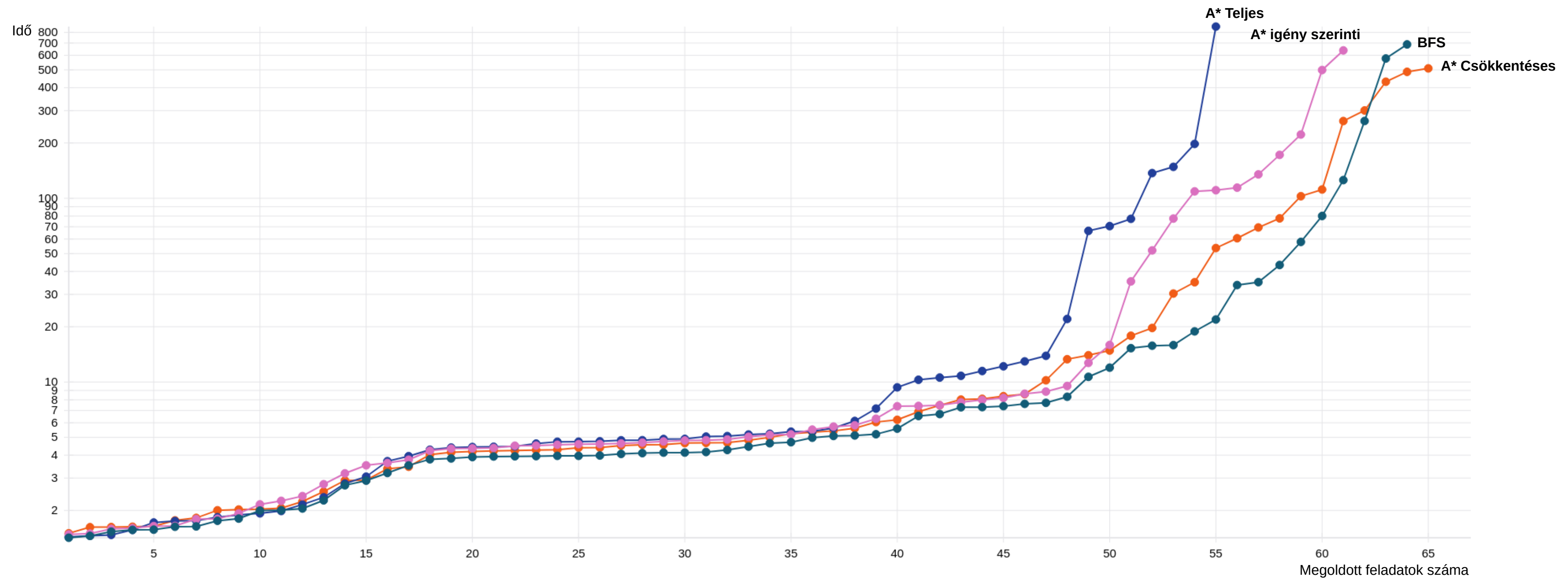
#	Absztrakció	Finomítás	Kifejtett gyerekek száma	Kezdeti pontosság
1	PRED_SPLIT	SEQ_ITP	∞	EMPTY
2	PRED_SPLIT	BW_BIN_ITP	∞	EMPTY
3	PRED_CART	SEQ_ITP	∞	EMPTY
4	PRED_CART	BW_BIN_ITP	∞	EMPTY

PRED_SPLIT absztrakciót vizsgálva azt láthatjuk, hogy BW_BIN_ITP finomítás esetén az összes Hierarchikus A* keresés jobban teljesített a BFS-nél, míg SEQ_ITP finomítást használva csak a csökkentéses Hierarchikus A* teljesít valamivel jobban.

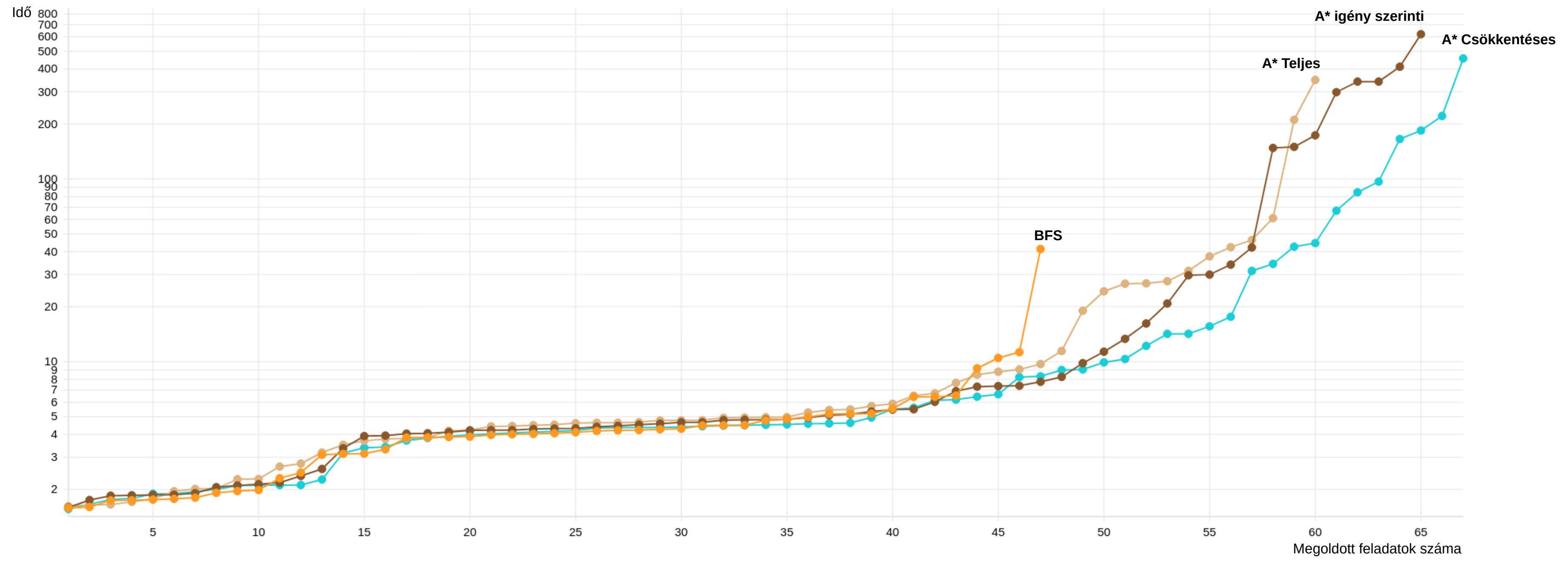
PRED_CART absztrakció esetén a mérést nem befolyásolta a keresési stratégia megválasztása.



4.7. ábra. Az egyes konfigurációk által időkorláton belül megoldott modellek száma. Az x tengely 1. sorában az alkalmazott finomítási eljárás található, míg a másodikban a két alkalmazott absztrakció.



4.8. ábra. 1. konfiguráció quantile plot ábrája.



4.9. ábra. 2. konfiguráció quantile plot ábrája.

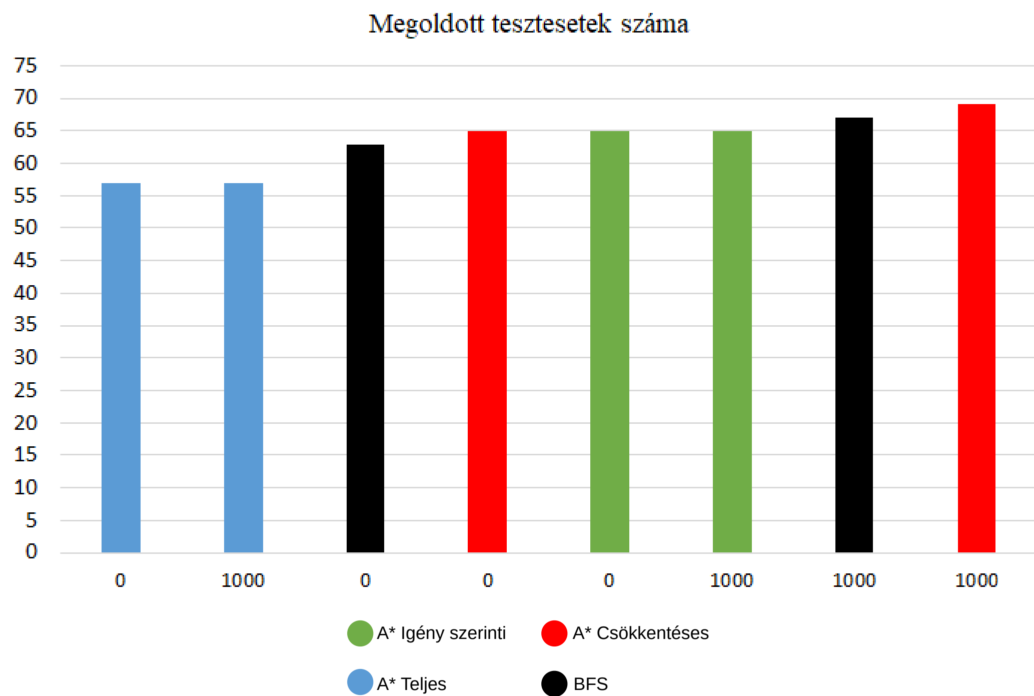
4.3.3. Explicit változó absztrakció

Explicit változó absztrakción végrehajtott konfigurációk:

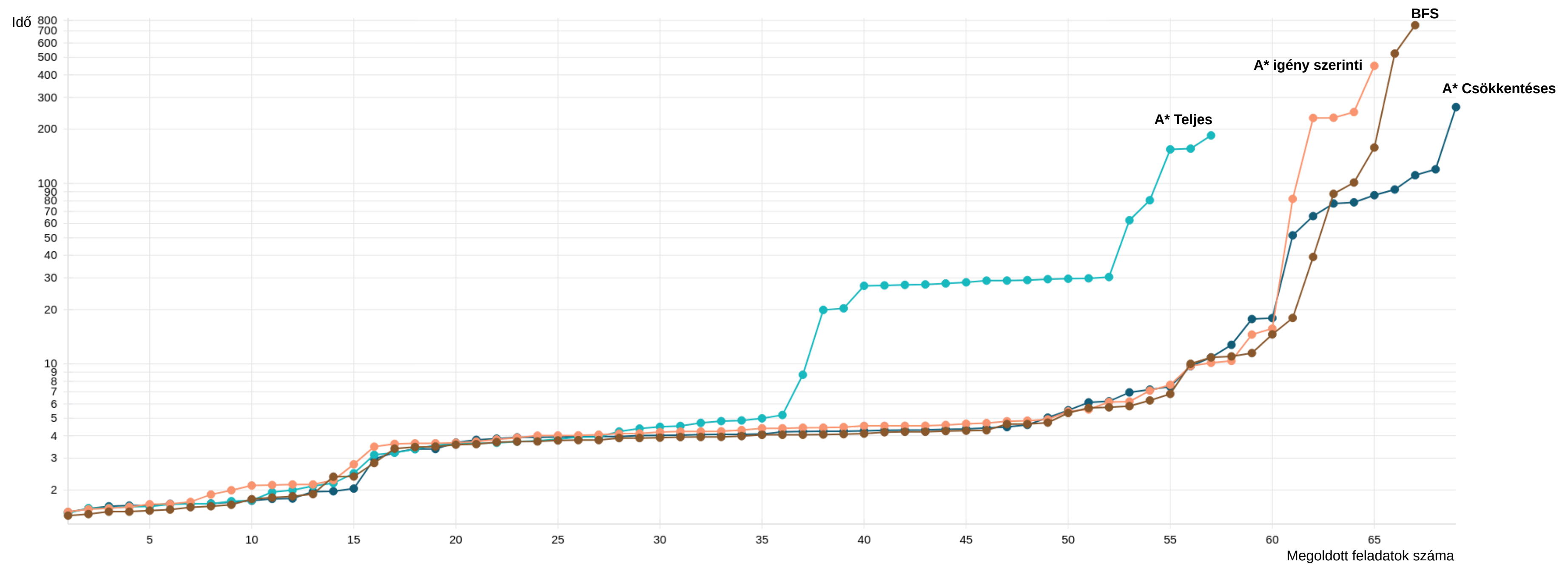
#	Absztrakció	Finomítás	Kifejtett gyerekek száma	Kezdeti pontosság
1	EXPL	SEQ_ITP	1000	EMPTY
2	EXPL	SEQ_ITP	∞	EMPTY

A mérések eredményéből látható, hogy bár a csökkentésen alapuló Hierarchikus A* keresés jobban teljesít a BFS-nél, azonban ez a különbség nem számottevő, ahogy ezt EXPL_PRED_COMBINED absztrakció esetén is láthattuk.

Mivel az 1. és 2. konfiguráció hasonló eredményeket mutatott, ezért az áttekinthetőség érdekében a quantile plot ábrán csak az 1. konfiguráció van ábrázolva.



4.10. ábra. Az egyes konfigurációk által időkorláton belül megoldott modellek száma. Az x tengelyen a kifejtett gyerekek száma található.



4.11. ábra. 1. konfiguráció quantile plot ábrája.

5. fejezet

Összefoglalás és jövőbeli tervek

Ezen dolgozat célja az absztrakció alapú formális verifikáció hatékonyabbá tétele volt, melyet a biztonságkritikus szoftveralapú rendszerek egyre nagyobb térnyerése motivált.

Munkám során egy hatékony absztrakcióalapú algoritmust, a CEGAR megközelítést fejlesztettem tovább. A CEGAR algoritmus iteratívan finomítja az absztrakciót és így próbál bizonyítást találni a szoftverek helyességére, vagy éppen ellenpéldát mutatni és rámutatni a szoftver hibáira. A CEGAR algoritmust a múltban többféle egyszerű, nem informált kereséssel integrálták, azaz csak az aktuális állapottér reprezentációból nyerhető információkat használták. Munkám során egy olyan keresési stratégiát dolgoztam ki, amely a heurisztikát a korábban bejárt állapottér reprezentációkból nyeri, ezáltal lehetővé téve az aktuális absztrakción való hatékonyabb keresést. Ezen algoritmus különböző változatait mutatta be a 3. fejezet.

Az így nyert algoritmus ugyan több állapottér reprezentációt is tárol, de ezáltal az informált keresés képes az aktuális, legfinomabb reprezentáción hamarabb konklúzióra jutni. Emellett fontos előnye a megközelítésnek, hogy robusztusabb, hiszen garantáltan mindig a legrövidebb aktuális ellenpéldát találja meg, így a finomítás során várhatóan a hibás absztrakció valós okaként szolgáló információt fogja felhasználni a CEGAR hurok a következő finomítás során. Emellett, amennyiben hibás a szoftverünk, az algoritmus képes a legrövidebb út megtalálására, amely csak a minimálisan szükséges lépéseket tartalmazza, így a mérnökök számára megkönnyíti a hibakereséśé folyamatát.

Az elméleti tárgyalást követően a 4. fejezet bemutatta a javasolt algoritmus teljesítménymérésének eredményeit. A mérési eredmények alapján kijelenthető, hogy a javasolt algoritmus kompetitív, alkalmazása sok esetben növelte az ellenőrzésre adott időlimit alatt ellenőrizhető modellek számát.

Összefoglalva a dolgozat a következő főbb kontribúcióimat mutatta be:

- Elméleti eredményem az új, A* alapú hierarchikus keresési algoritmus család kidolgozása a CEGAR algoritmus finomítási lépésének a támogatására. Ezáltal sikerült egy robusztus megoldást kidolgozni a keresés támogatására, továbbá a hibába vezető rövid, tömör ellenpélda valódi segítséget nyújt a mérnököknek. Az új algoritmusok helyességét megvizsgáltam és bizonyítottam.

- Gyakorlati eredményként az elkészült algoritmusokat integráltam az egyetemen fejlesztett nyílt-forráskódú Theta modellellenőrző keretrendszerbe és elérhetővé tettem bárki számára.

- Publikus és ipari partnerektől eredő benchmark készleteken vizsgáltam az algoritmusaim hatékonyságát. A mérések során meggyőződtem, hogy a megoldásom versenyképes alternatívása a jelenleg elérhető algoritmusoknak és robusztus, jól skálázódó megoldást nyújt a mérnökök számára.

A jövőben fontos lesz a mérések még alaposabb elvégzése, amely során igyekszem majd még több valós, ipari esettanulmányból eredő modellen kiértékelni a megoldásomat, ugyanis várhatóan ott még jobban elő tud jönni az okos keresési algoritmusok hatékonysága. Emellett ipari rendszerek esetén különösen fontos a mérnökök támogatása, tehát a rövid, fókuszált ellenpéldák szolgáltatása.

Az algoritmusban több optimalizálási lehetőség is rejlik még. Ilyen például az igény szerinti változat hatékonyabbá tétele azzal, hogy korábbi ARG-re visszatéréskor azt csak addig fejtjük ki, amíg meg nem bizonyosodtunk róla, hogy a már ismert heurisztikájú csúcsokhoz alacsonyabb heurisztika tartozik. Ezeket a jövőben szeretném megvalósítani, a javasolt algoritmus gyakorlati hasznosságát tovább növelve.

Köszönetnyilvánítás

Szeretném megköszönni konzulenseimnek a segítséget, mind az fejlesztés során, mind a TDK megvalósításában.

Külön szeretnék köszönetet mondani Szekeres Dánielnek a sok útmutatásért a feladat kivitelezése során.

Irodalomjegyzék

[1] Dirk Beyer: Progress on software verification: SV-COMP 2022. In Dana Fisman – Grigore Rosu (szerk.): *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022*, Lecture Notes in Computer Science konferenciasorozat, 13244. köt. 2022, Springer, 375–402. p. URL `https://doi.org/10.1007/978-3-030-99527-0_20`.

[2] Dirk Beyer – Thomas A. Henzinger – Ranjit Jhala – Rupak Majumdar: The software model checker blast. *Int. J. Softw. Tools Technol. Transf.*, 9. évf. (2007) 5-6. sz., 505–525. p. URL `https://doi.org/10.1007/s10009-007-0044-z`.

[3] Dirk Beyer – Stefan Löwe – Philipp Wendler: Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21. évf. (2019) 1. sz., 1–29. p. URL `https://doi.org/10.1007/s10009-017-0469-y`.

[4] Edmund M. Clarke – Orna Grumberg – Somesh Jha – Yuan Lu – Helmut Veith: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50. évf. (2003) 5. sz., 752–794. p. URL `https://doi.org/10.1145/876638.876643`.

[5] Stefan Edelkamp – Alberto Lluch-Lafuente – Stefan Leue: Directed explicit model checking with HSF-SPIN. In Matthew B. Dwyer (szerk.): *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings*, Lecture Notes in Computer Science konferenciasorozat, 2057. köt. 2001, Springer, 57–79. p. URL `https://doi.org/10.1007/3-540-45139-0_5`.

[6] Paul Gastin – Pierre Moro – Marc Zeitoun: Minimization of counterexamples in SPIN. In Susanne Graf – Laurent Mounier (szerk.): *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, Lecture Notes in Computer Science konferenciasorozat, 2989. köt. 2004, Springer, 92–108. p. URL `https://doi.org/10.1007/978-3-540-24732-6_7`.

[7] Alex Groce – Willem Visser: Heuristics for model checking java programs. *Int. J. Softw. Tools Technol. Transf.*, 6. évf. (2004) 4. sz., 260–276. p. URL `https://doi.org/10.1007/s10009-003-0130-9`.

[8] Ákos Hajdu – Zoltán Micskei: Efficient strategies for cegar-based model checking. *J. Autom. Reason.*, 64. évf. (2020) 6. sz., 1051–1091. p. URL `https://doi.org/10.1007/s10817-019-09535-x`.

[9] Peter E. Hart – Nils J. Nilsson – Bertram Raphael: A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4. évf. (1968) 2. sz., 100–107. p. URL `https://doi.org/10.1109/TSSC.1968.300136`.

[10] Peter E. Hart – Nils J. Nilsson – Bertram Raphael: Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Newsl.*, 37. évf. (1972), 28–29. p. URL `https://doi.org/10.1145/1056777.1056779`.

[11] Arut Prakash Kaleeswaran – Arne Nordmann – Thomas Vogel – Lars Grunske: A systematic literature review on counterexample explanation. *Inf. Softw. Technol.*, 145. évf. (2022), 106800. p. URL https://doi.org/10.1016/j.infsof.2021.106800.

[12] Sebastian Kupferschmid – Klaus Dräger – Jörg Hoffmann – Bernd Finkbeiner – Henning Dierks – Andreas Podelski – Gerd Behrmann: Uppaal/dmc- abstraction-based heuristics for directed model checking. In Orna Grumberg – Michael Huth (szerk.): *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007*, Lecture Notes in Computer Science konferenciasorozat, 4424. köt. 2007, Springer, 679–682. p.
URL https://doi.org/10.1007/978-3-540-71209-1_52.

[13] Jun Maeoka – Yoshinori Tanabe – Fuyuki Ishikawa: Depth-first heuristic search for software model checking. In Roger Lee (szerk.): *Computer and Information Science 2015* (konferenciaanyag). Cham, 2016, Springer International Publishing, 75–96. p.

[14] Mondok Milán: Mérnöki modellek formális verifikációja kiterjesztett szimbolikus tranzíciós rendszerek segítségével. 2020. URL https://diplomaterv.vik.bme.hu/hu/Theses/Mernoki-modellek-formalis-verifikacioja.

[15] Viktor Schuppan – Armin Biere: Shortest counterexamples for symbolic model checking of LTL with past. In Nicolas Halbwachs – Lenore D. Zuck (szerk.): *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005*, Lecture Notes in Computer Science konferenciasorozat, 3440. köt. 2005, Springer, 493–509. p.
URL https://doi.org/10.1007/978-3-540-31980-1_32.