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

Igényvezérelt heurisztikaszámítás informált kereséssel támogatott absztrakció alapú modellellenőrzésben

Szakdolgozat

Készítette
Vörös Asztrik

Konzulens
Szekeres Dániel

Kulturális és Innovációs Minisztérium

Új Nemzeti Kiválóság Program

Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal

2023. december 7.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Vörös Asztrik*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2023. december 7.

Vörös Asztrik
hallgató

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 funkciót 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 megtaláljuk annak hibáit. A modellellenőrzés lényege a lehetséges hibaállapotok megkeresése a rendszer állapotterében, egy intelligens keresés segítségével.

Az állapottér mérete gyakran a program/rendszer leírásának méretében exponenciálisan nő, vagy akár végtelen is lehet összetett adat esetén. 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 (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.

Egyes területeken gyakran nem csak a helyességről alkotott ítélet, hanem egy ellenpélda is a modellellenőrzés kimenete, ha a követelmény sérül. Az 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 legkisebb méretű 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, ami a jelenlegi keresésben sok többlet számítást okoz.

Ezen munka célja, hogy a korábbi iterációk 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 kontribúció 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 kiértékelésre kerül a hatékonysága, és összehasonlításra kerülnek az elterjedten használt keresési stratégiákkal. A méréseket a

nyílt forráskódú Theta modellellenőrző-keretrendszerbe beépített implementáción kerülnek végrehajtásra szoftverek és mérnöki modellek verifikálásának benchmarkolásáva.

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 by benchmarking on software and engineering models.

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 [10].

Amennyiben a követelmény sérül, a modellellenőrző algoritmus által nyújtott ellenpélda segítségként tud szolgálni [13] 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 [8, 18].

A CEGAR algoritmusok szakirodalmában találhatunk példát mélységi és szélességi keresés alkalmazására, illetve számos más prioritás alapú keresési stratégiára, melyek a modell előzetes elemzésén alapulnak [10]. 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 absztrakciót 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] [19] 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.

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 során. Mivel az algoritmus a CEGAR hurokba épül bele, ezért az ahhoz kapcsolódó fogalmak ismertetésére is sor kerül.

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

- Architektúra, 4. fejezet: A korábban ismertetett Hierarchikus A* változatok implementációjához tartozó, további Hierarchikus A* változatokkal bővíthető architektúra kerül ismertetésre.

- Mérések, 5. fejezet: A mérési környezet és tesztadatok meghatározása, majd a Hierarchikus A* változatok és a területen már elterjedt keresési algoritmusok hatékonyságainak összehasonlítása.

- Összefoglalás és jövőbeli tervek, 6. fejezet: A dolgozatban foglaltak összefoglalása, illetve bemutatja a munka folytatásának tervezett irányait.

A munkám a korábbi TDK-imra épülnek [1] [2], illetve további ábrákkal kiegészítésre kerültek.

Kapcsolódó munkák Munkám újdonsága, a korábbi munkám során kifejlesztett absztrakcióval kombinált informált keresési stratégiákat kiegészítettem egy újabb változattal.

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

[2] Munkám forkolt Thetába beépítve: `github.com/asztrikx/theta/tree/astar`

Ezen a területen javarészt az egyszerűbb keresések (szélességi és mélységi keresés), vagy statikusan heurisztikát számolt keresési algoritmusok érhetőek el [9, 7, 15]. A korábbi iterációban lévő távolságértéket heurisztikaként való felhasználásról is készült már munka [17].

De a heurisztika számítás történhet akár egy absztraktabb probléma megoldásával [14]. Ezekkel kapcsolatban az elmúlt években született egy nagyon alapos összefoglaló [10], 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.

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 Automata, CFA)* formális modellt generálunk a forráskódból, és annak struktúráját felhasználva végezzük el az ellenőrzést.

Definíció 1 (CFA / Control Flow Automata). A CFA egy (L, V, D, E, l_0, l_F) gráf

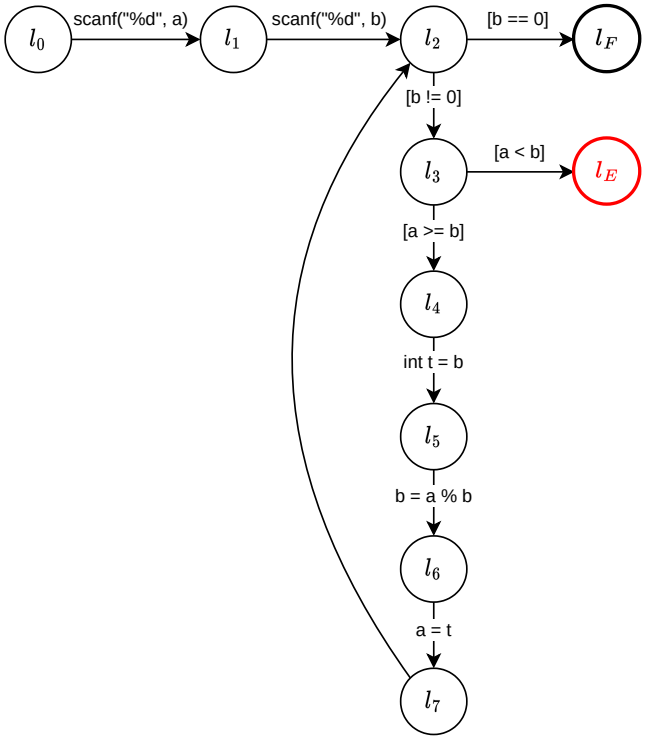
- L: Helyek halmaza, amelyek az egyes elemi programkód utasítások végrehajtás előtti, illetve utáni strukturális állapotokat írják le, gráfunkban csúcsokkal reprezentálva. Speciális helyek a program végét és elejét jelző helyek.
- V: Programkódban szereplő változók halmaza, $V = \{v_1, \ldots, v_N\}$.

- 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, ahol az 1. komponensből indul az él, illetve $Ops = Actions \cup Guards$.
- $Actions$: 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.
- l_F: Végső hely, mely a program végét jelzi, itt érnek véget 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 utasításban található kifejezés igazra értékelődjön ki. Ez esetben a CFA-ban kijelölhető egy hely (l_E) melyet elérve sérül a követelmény.

```
scanf("%d", &a);
scanf("%d", &b);
while (b != 0) {
    assert(a >= b);
    int t = b;
    b = a % b;
    a = t;
}
```

2.1. ábra. C kód



2.2. ábra. C kódhoz tartozó CFA

2.3. Állapottér

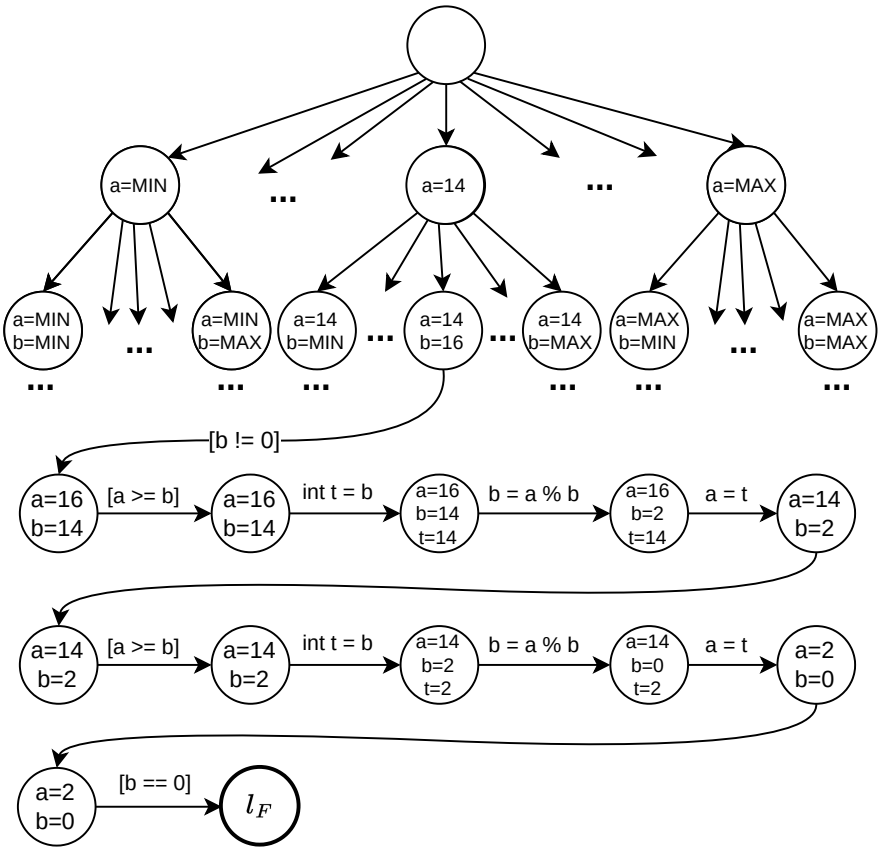
Az adott formális modell egyes lefutásai során az állapottérben közlekedünk, melyet a formális modell határoz meg. Az adott állapotokról a formális követelményt felhasználva megállapíthatjuk, hogy az *hibás állapot*-e.

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.
- $t : S \times Ops \to S$: 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 utasítása 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 nemdeterminisztikus műveletenként. Például c db nemdeterminisztikus 32 bites egész szám esetén $(2^{32})^c$ féle lefutás lehetséges. Ezt a jelenséget állapottér-robbanásnak nevezzük.



2.3. ábra. C kódhoz tartozó állapottér egy részlete

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, k, Π, 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.
- $k : S \to 2^{S_{konkrét}}$: 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.
- $t : S \times Ops \times \Pi \to 2^S$: Átmenet függvény, amely a jelenlegi absztrakt állapot és egy utasítás segítségével a pontosságot figyelembevéve meghatározza a rákövetkező absztrakt állapotokat. $t(s, op) \supseteq \bigcup_{k \in k(s)} t_{konkrét}(k, op)$, azaz op utasítás esetén egy absztrakt állapotból kimenő élek megegyeznek az s-hez tartozó konkrét állapotokból kimenő, op utasítású élek uniójának szuperhalmazával. ▪

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 $k(A) \subseteq k(B)$, azaz B absztraktabb A-nál. 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). Egy n hosszú

$$L_{\text{absztrakt}} \in S^1_{\text{absztrakt}} \times op^1 \times S^2_{\text{absztrakt}} \times \cdots \times S^n_{\text{absztrakt}}$$

absztrakt lefutás konkretizálása alatt egy olyan

$$L_{\text{konkrét}} \in S^1_{\text{konkrét}} \times op^1 \times S^2_{\text{konkrét}} \times \cdots \times S^n_{\text{konkrét}}$$

állapotsorozatot értünk, ahol

$$\forall i \in 1..n : L^i_{\text{konkrét}} \in k(L^i_{\text{absztrakt}})$$

és

$$\forall i \in 1..(n-1) : (L^i_{\text{konkrét}}, op^i, L^{i+1}_{\text{konkrét}}) \in E_{\text{konkrét}}$$ ▪

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 [6].

Több információt megtartó pontosságot választva *finomítjuk* az absztrakciót, mely explicit változó, illetve predikátum absztrakció esetén a pontosságot leíró halmazhoz való új elem hozzáadását jelenti. Az explicit változós absztrakció során az egyes változó értékeit, míg predikátum absztrakció során csak a változókra vonatkozó predikátumokat követünk. 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 ugyanazon 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). (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. Az élek az 1. komponensből indulnak.

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

  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: $e(n) = \{e \mid e \in E \wedge e_1 = n\}$.
- $w : E \to \mathbb{Z}$: Adott él súlyát leíró komponensét adja meg: $w(e) = e_3$.
- $c : E \to N$: Adott él végcsúcsát adja meg: $c(e) = e_4$. ▪

Az Elérhetőségi gráf 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 csúcsokat elég egyszer meglátogatni.

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

Definíció 8 (Csúcs gyereke). A csúcsból 1 élen keresztül elérhető csúcsok halmaza, azaz egy adott n csúcs esetén:

$$\{e_4 \mid e \in E \wedge e_1 = n\}$$ ▪

Definíció 9 (Csúcs szülője). Az adott n csúcsban végződő élek kiinduló csúcsainak halmaza. Ez Elérhetőségi gráf esetén egy csúcsot jelent:

$$\{p_n\} = \{e_1 \mid e \in E \wedge e_4 = n \wedge e_3 = 1\}$$ ▪

Definíció 10 (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 11 (Távolságfüggvény). A távolságfüggvény $d : N \to \overline{\mathbb{N}}$ megadja, hogy az adott csúcs a legközelebbi célcsúcstól milyen távol van, azaz minden célcsúcstól való távolság minimumával egyezik meg. ▪

Ezek után már definiálhatjuk a keresési algoritmusokhoz kapcsolódó fogalmakat.

Definíció 12 (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 célcsúcsra egyszerűsítjük. Így ha a gráfbejárás megtalál a célcsúcsok közül egyet, akkor a keresési probléma kimenete az oda vezető út, egyébként pedig a tény, hogy nem elérhető a gráfban célcsúcs. ▪

A teljes állapottér előállítása és tárolása jelentős időt és memóriát vehet igénybe, ezért az csak igény szerint történik meg. Ez gráf reprezentáció esetén azt jelenti, hogy annak egy adott állapotában lehetnek olyan csúcsok, melyeknek az optimalizáció érdekében éppen még nem ismertek a gyerekei, de egy későbbi állapotban már azok lesznek, ha azokra közben szükség lett. Az ilyen csúcsokat **kifejtetlen csúcs**oknak hívjuk. A folyamatot, mely előállítja egy kifejtetlen csúcs esetében a rákövetkező csúcsokat, **kifejtés**nek nevezzük.

Definíció 13 (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 14 (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 15 (Nem informált keresési algoritmusok). Olyan keresési algoritmusok, melyeknek a célcsúccsal kapcsolatos információjuk kimerül abban, hogy egy csúcs célcsúcsnak minősül-e. ▪

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:

$$f = g + h \tag{2.1}$$

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

Definíció 17 (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). Ha a heurisztikánk konzisztens és nincsenek negatív körök a gráfban, akkor az A* keresési algoritmus által visszaadott út a legközelebbi célállapotban végződő út lesz. [11, 12] ▪

2.6. Absztrakt elérhetőségi gráf

Az *absztrakt elérhetőségi gráf (Abstract Reachability Graph, ARG)* [4] az absztrakt állapotterek (absztrakt) lefutásait gráfként reprezentáló tömör adatstruktúra, a nem absztrakt Elérhetőségi gráfhoz hasonló módon.

Definíció 18 (ARG / Absztrakt elérhetőségi gráf). (N, L, E, C) 4-es.

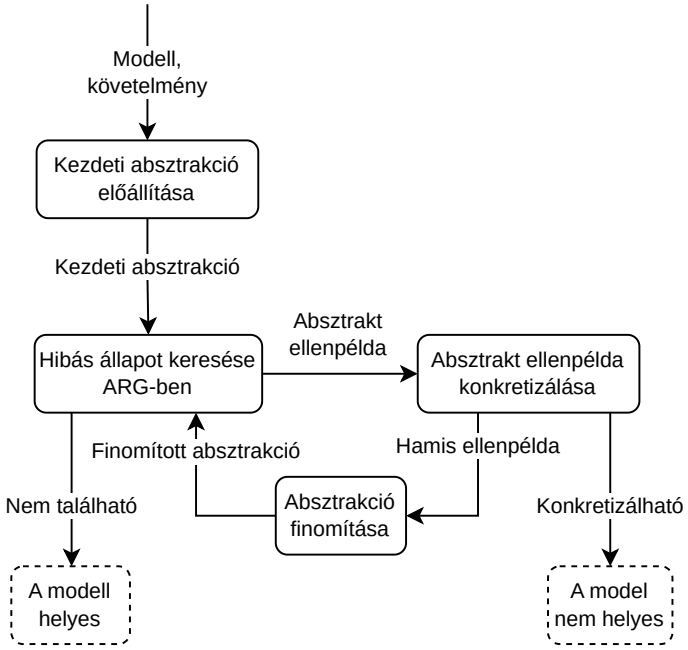
- N: Gráfban lévő csúcsok halmaza. Speciális eleme a gyökér csúcs, melynek állapota a kezdőállapot.
- $L : N \to S_{absztrakt}$: Az adott csúcshoz tartozó absztrakt állapotot megadó címkézés.
- $E \subseteq N \times Ops \times \{1\} \times N$: Gráfban lévő irányított élek halamaza.

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

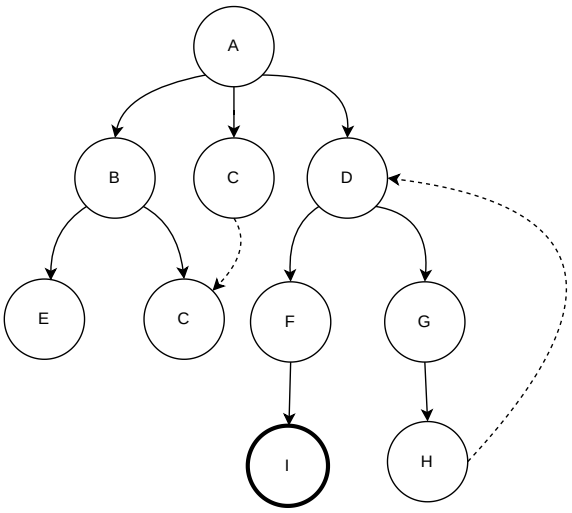
  azaz két csúcs között akkor megy él adott *op*-val, ha a kiinduló csúcs állapotából *op*-t végrehajtva, az átmeneti függvény szerint érkezhetünk a végcsúcshoz állapotába.
- $C \subseteq N \times \{0\} \times N$: Fedőélhalmaz, mely speciális irányított éleket tartalmaz. Az élek az 1. komponensből indulnak. Egy n_1 és n_2 csúcs között akkor *lehet* felvenni n_1-ből kiinduló fedőélet, ha $L(n_1) \preceq L(n_2)$ és n_1 nincs még kifejtve vagy lefedve. ▪

Fedőélek olyan csúcs között *keletkezhetnek*, ahol az egyik csúcs absztraktabb egy másiknál ($A \preceq B$), illetve céljuk hogy elkerüljék a konkrétabb csúcsból történő részgráf kifejtést. Ilyenkor a finomabb csúcs részgráfja az absztraktabb fedő csúcs részgráfja lesz, hiszen az tartalmaz minden lehetséges utat, ami a finomabb csúcsból lett volna elérhető a konzervatív absztrakciónak köszönhetően. Ez azonban azt jelenti, hogy ezek az élek a konkrét modellben nem szerepelnek, csak optimaliációs céllal jönnek létre, ezáltal azok 0 súllyal rendelkeznek.

Definíció 19 (Csúcs keresési szülője). Az adott keresés ezen csúcs egy kimenő élén át érte el legkorábban az adott csúcsot. Ez az él lehet fedő él is, szemben a korábbi szülő definícióval (9). ▪



2.5. ábra. CEGAR, illetve a benne található CEGAR hurok



2.4. 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)* [6] 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.

Az algoritmus a formális verifikáció két bemenetét, a modellt és a követelményt felhasználva létrehoz egy kezdeti absztrakciót.

Ezután belépünk a *CEGAR hurok*ba. A jelenlegi absztrakcióhoz tartozó ARG-ben elindítunk egy keresést, valamilyen keresési algoritmus segítségével, ami egy célcsúcs megtalálása esetén visszatér a hozzá vezető úttal, 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. Ez az absztrakt ellenpélda konkretizálhatóságának eldöntésével állapítható meg 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 absztrakt ellenpélda nem konkretizálható, akkor finomítunk az absztrakción, azaz az absztrakció mértékét csökkentjük azért, hogy az így keletkező finomított absztrakciójú ARG-ben ne találhassuk meg ugyanazt a nem konkretizálható lefutást újra. Ezzel az új absztrakcióval újra végrehajthatjuk a CEGAR hurkot.

Ha egy adott absztrakciójú ARG-ben a keresési algoritmus nem talál célcsúcsot, 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 tudjuk konkretizálni, mely esetben ezen konkretizált ellenpélda lesz az kimeneti ellenpélda.

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* keresési algoritmus CEGAR hurokba való beépítéséhez szükséges valamilyen heurisztika, mely becslést ad a legközelebbi célcsúcstól 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 távolság 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 a $d(n)$, ami szintén nem lehet igaz, hiszen ellentmond a távolság tulajdonságának. ▪

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, eggyel 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 20 (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 a részbenrendezésben nagyobb helyen áll a B-beli csúcs állapotához képest. ▪

Ehelyett azonban struktúratartó providert fogunk használni, amelynek okát láthatjuk majd a 3. tételben.

Definíció 21 (Struktúratartó provider). Tetszőleges provider meghatározása esetén az összes korábbi iterációjú ARG-beli csúcs között keresünk. Ennél hatékonyabb megoldás, ha egy p_B szülővel rendelkező n_B csúcsnak a providerét a p_B providerének gyerekei

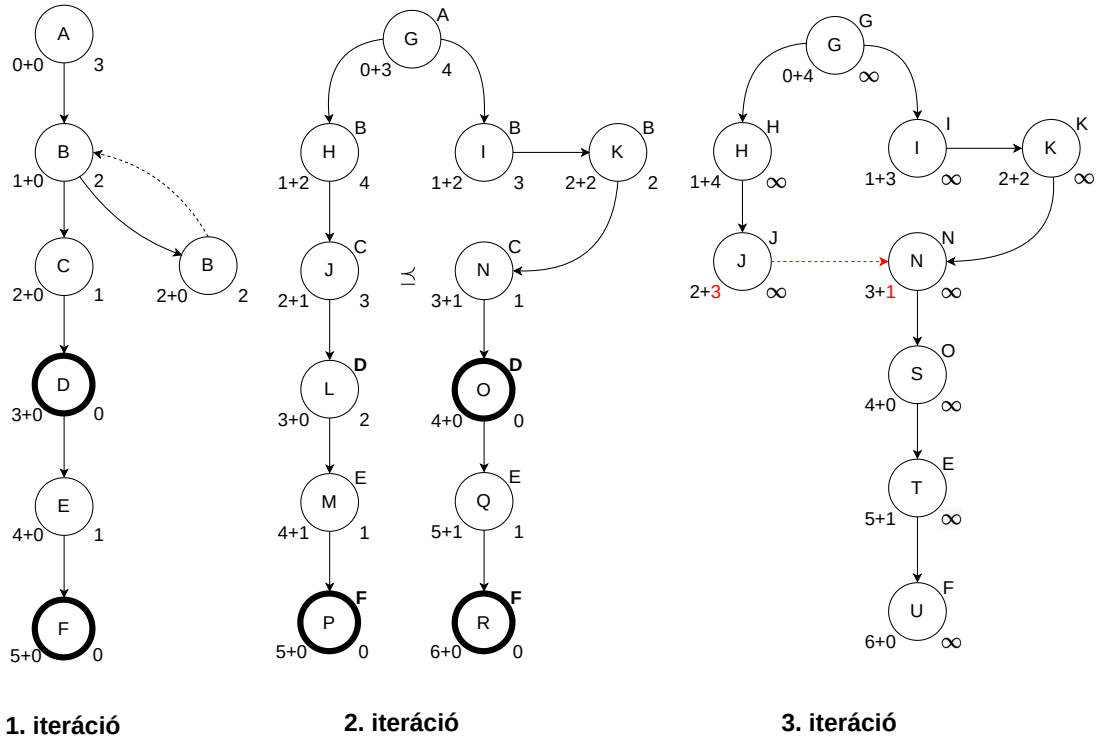
között keressük. Ehhez az kell, hogy mindig létezzen ilyen gyerek, ami teljesül hiszen egy absztraktabb állapot felülbecsli a konkrétabb állapotból elérhető lehetséges lefutásokat, tehát:

$$\forall n_B \in N_B : L_A(provider(n_B)) \succeq L_B(n_B) \wedge \exists e_A \in e_A(provider(p_B)) : c(e_A) = provider(n_B).$$

B gyökér csúcsának providere A gyökere lesz. ▪

Mindkét provider esetében biztosítanunk kell, hogy a korábbi ARG eléggé ki legyen fejtve, hogy létezzen provider. Azonban ez Struktúratartó provider esetén egyszerűen megoldható azzal, hogy $provider(p_B)$-t kifejtjük, ha még nincsen.

A Struktúratartó providernek másik előnye az, hogy az általa elérhetővé tett A-beli távolság értékek konzisztens heurisztikát alkotnak, ha a fedőélek behúzását korlátozzuk. (Ez a megkötés a Tetszőleges provider esetén is fent kell álljon, hiszen Tetszőleges provider esetén is lehetnek a providerek struktúratartóak.) 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.



3.1. ábra. Az ábrán 3 különböző iteráció ARG-je 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 is jelezve van $L(J) \preceq L(N)$. A 3. iterációban behúzott fedőél két csúcsának heurisztikája $h(J) = h(p_B) = 3$ és $h(N) = h(n_B) = 1$, ami nem konzisztens hiszen $h(p_B) \not\leq h(n_B)$.

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, mely nem csak két egymást követő iterációbeli ARG-kre igaz.

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

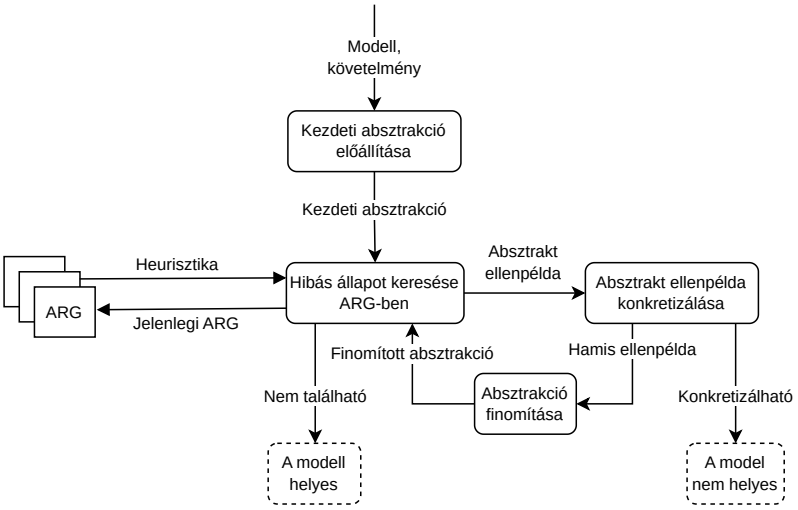
Bizonyítás. Ahhoz, hogy egy heurisztika konzisztens legyen, minden m_B célcsúcsra $h(m_B) = 0$ kell, hogy teljesüljön, illetve $\forall n_B \forall e_B \in e(n_B) : h(n_B) - h(c(e_B)) \leq w(e_B)$ kell, hogy igaz legyen.

Ha m_B célcsúcs volt, akkor annak absztrakt állapota hibás volt, így mivel $m_A \coloneqq provider(m_B) \succeq m_B$, ezért m_A absztrakt állapota is hibás, tehát m_A is célcsúcs. 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: $d_A(provider(p_B)) - d_A(provider(n_B)) \leq 1$. Mivel struktúratartó providert használunk, ezért a szülő és a gyerek providerei is szülő gyerek viszonyban állnak, ahol a gyerekek távolság értéke legfeljebb 1-gyel kevesebbek a szülőéhez képest, hiszen ellenkező esetben a távolság nem lenne helyes a szülő providerére nézve. Tehát $d_A(provider(p_B)) - d_A(provider(n_B))$ legnagyobb értéke 1 lehet, így teljesül a második követelmény is. ▪

A *konzisztens heurisztikát használó* A* keresés segítségével a csúcsok számára megállapíthatjuk a távolságértékeket, melyek a következő iteráció számára konzisztens heurisztikaként szolgálnak. Az első iteráció esetében, tekintve hogy nincs korábbi iteráció amely heurisztikát biztosítson, $h(n) = 0$ heurisztikát használunk minden csúcsra. Ennek során az A* keresés csak a mélységet veszi figyelembe, amely a Szélességi kereséssel egyezik meg, ezáltal ez is képes előállítani a távolságfüggvényt, hogy azt a későbbi iteráció konzisztens heurisztikaként használja.

A már megismert CEGAR hurok így kibővül egy ARG tárolóval, amiben egy adott ARG-hez megkaphatjuk az előtte lévő iterációban használt ARG-t, hogy annak távolságértékeit heurisztikaként felhasználjuk. Az ARG-ket az adott iterációban végrehajtott keresés végeztével tároljuk el. A már látott CEGAR hurkot leíró ábra (2.5) a következőre módosul:



3.2. ábra. A* beépítve a CEGAR hurokba

A következő szekciókban az ARG kifejtésének mértékében, illetve a heurisztika pontosságában eltérő Hierarchikus A* változatokat kerülnek bemutatásra és megvizsgálásra:

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

3.2. Megismert távolságok beállítása ARG-ben

Az ARG-k esetében figyelembe kell venni, hogy a fedőélek olyan csúcs között keletkezhetnek, ahol az egyik csúcs absztraktabb egy másiknál, illetve céljuk, hogy elkerüljék a konkrétabb csúcsból történő részgráf kifejtést. Ilyenkor a részgráf absztraktabb verziója a fedő csúcs részgráfja lesz, hiszen az tartalmaz minden lehetséges utat, ami a konkrétabb állapotból lett volna elérhető. Ez azonban azt jelenti, hogy ezek az élek a konkrét modellben nem szerepelnek, ezáltal azokat 0 súllyal kell figyelembe venni a távolság meghatározásakor.

A* keresés esetében, ha találunk egy célcsúcsot, akkor csak az oda vezető útnak a távolsággal még nem rendelkező csúcsaira ismerjük meg a legrövidebb távolságot. Azonban ha egy csúcs egyedüli gyereke egy olyan csúcs, mely ennek az útnak a része, akkor ezen csúcsnak is kizárásos alapon beállíthatjuk a távolságértékét.

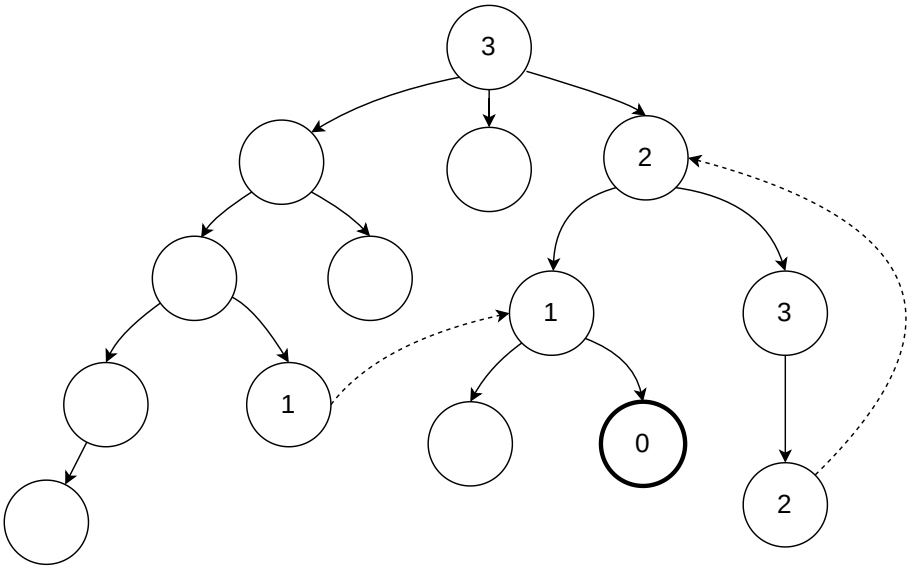
Az előbb leírt folyamatot meg tudjuk általánosan is fogalmazni azon csúcsokra, melyek nem elemei az legrövidebb útnak. Minden olyan esetben, ha egy csúcs gyerekének megtudjuk a távolság értékét és minden más gyerekének már ismert a távolság értéke, akkor a csúcsnak is kiszámíthatjuk a távolság értékét:

$$d(n) = \min_{e \in e(n)} d(c(e)) + w(e) \tag{3.1}$$

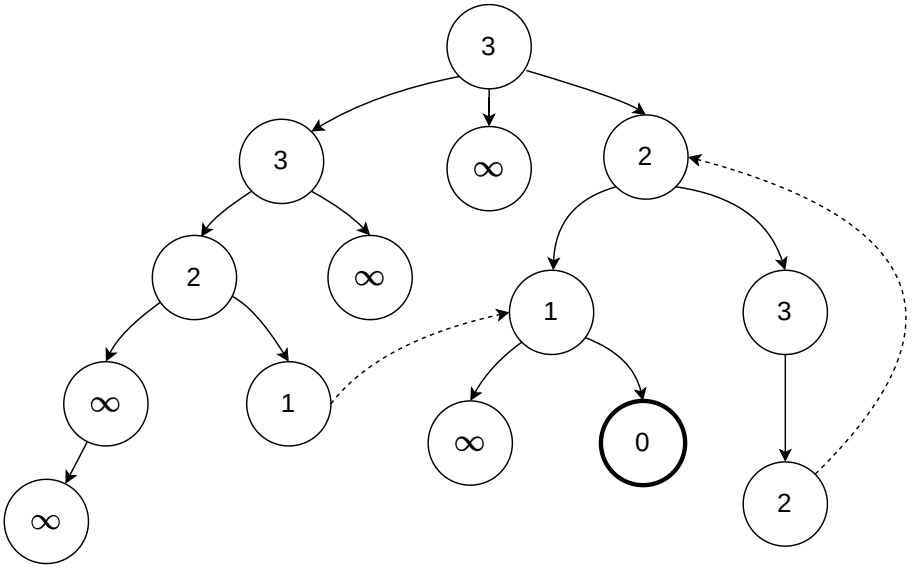
Végtelen minimum érték esetén $d(n) = \infty$. Abban az esetben ha n nincs lefedve, akkor $\forall e \in e(n) : w(e) = 1$, amúgy $w(e) = 0$ és a minimum számítás egyetlen gyereken történik.

Szintén, ha egy csúcsnak végtelen a heurisztikája vagy kifejtve nincs több gyereke (és nem célcsúcs), akkor annak távolságértékét végtelenre állíthatjuk, majd a fentebb megfogalmazott képletet felhasználva a csúcsnak az őseire is meghatározhatunk végtelen távolságot.

Ez a megoldás azonban nem tudja minden csúcs távolságértékét meghatározni, ami kiszámítható lenne. Például ha egy adott csúcs egyik gyerekének leszármazottja be van fedve a csúcs egy ősébe, akkor annak a csúcsnak távolságértéke az adott keresésnek a végeztével még nem kerül meghatározásra ezen módszer által.



(a) A gyökércsúcstól a kiemelt célcsúcsig tartó legrövidebb út csúcsainak, illetve a 3.1. képlet segítségével további csúcsok távolságának meghatározása.



(b) A távolság nélküli levelek és a végtelen heurisztikájú csúcsok végtelenre állítása, majd a 3.1. képlettel további csúcsok távolságának meghatározása.

3.3. ábra. Egy ARG-n a távolságok beállítása a fentebb megismert módszer segítségével.

3.3. Hierarchikus A* keresés teljes ARG kifejtéssel

Az első változat során teljesen kifejtjük az adott ARG-t. Ekkor nem állunk meg egy célcsúcs elérésekor, hanem addig fejtjük ki a gráfot, amíg van kifejtetlen csúcs.

Az előző szekcióban tárgyalt módszer nem tudja minden ismerhető távolságú csúcsnak a távolságát meghatározni. Azonban az ott tárgyalt megoldás azt az esetet vizsgálta, amikor egy (vagy valahány) célcsúcsot találunk meg és az az által megismerhető távolságokat állítottuk be. Teljes kifejtés esetén azonban az ARG-ben található összes célcsúcs ki van már fejtve, ami által minden csúcs számára meg tudjuk határozni a távolságértéket. Ezt a célcsúcsokból kiinduló módosított BFS bejárással (mely a fedőélek 0 súlyára tekintettel van) 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. Azon csúcsok, amelyek az ARG semelyik célcsúcsából nem voltak elérhetőek, azok távolságát végtelenre állíthatjuk.

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 az absztrakt állapottér mérete végtelen is lehet a használt absztrakciótól függően, ezért ilyen esetekben nem tudjuk használni ezt az algoritmust.

3.4. Hierarchikus A* keresés részlegesen igény szerinti ARG kifejtéssel

A részlegesen igény szerinti A* keresés azt a tényt használja ki, hogy egy A* bejárás során nem feltétlen fejtünk ki minden csúcsot, így azok heurisztikáját, azaz a hozzájuk tartozó absztrakt csúcsok távolságértékét nem szükséges meghatározni. Emiatt ezen módszer során azok csak akkor kerülnek kiszámításra, amikor azokra a következő iterációban történő A* keresés során szükség van.

Az algoritmus során csak addig fejtünk ki egy gráfot, amíg el nem érünk egy célcsúcsot. Ha egy adott iterációban történő A* keresés során 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, mely végeztével $d(provider(n))$ ismerté válik.

Mivel a már kifejtett csúcsok részt vettek egy A* keresésben, ezért a keresések által vizsgált n csúcsok p szülőjének már ismert kell legyen a heurisztikája, tehát $provider(p)$ is biztosan létezik. Ezzel teljesül a 21. definíció után elvárt követelmény.

Fontos megjegyezni, hogy miközben a $provider(n)$-től indított keresés zajlik, szintén találhatunk egy n_2 csúcsot, melynek heurisztikája nem ismert, aminek meghatározásakor az ahhoz tartozó $provider(n_2)$-től, még eggyel korábbi iterációbeli csúcstól kell keresést indítani. Ez a rekurzív folyamat egészen addig előfordulhat, amíg el nem érünk az első iterációhoz, ahol minden csúcs heurisztikája ismert ($h(n) = 0$).

Szintén találkozhatunk olyan n_2 csúccsal, aminek a távolságértéke már ismert és nem végtelen. Ebben az esetben nem kell újra bejárnunk a hozzá tartozó részgráfot, hanem elég lekorlátozni a keresés mélységét

$$k \coloneqq g(n_2) + d(n_2) \tag{3.2}$$

értékre, hiszen a gyökércsúcstól n_2-n át elérhető egy célcsúcs. Ezen k korlátot a csúcsok f-beli értékeivel szemben állítjuk, hiszen azok egy alsó becslést adnak a gyökércsúcshoz legközelebbi célcsúcstól való távolságra az adott csúcsot érintve. Ha nem találunk ezen k

korláton belül más célcsúcsot, akkor n_2 csúcsból kiindulva a 3.2. szekcióban megismert módszerrel beállíthatjuk az egyes csúcsok távolságait, ahol n_2 a célcsúcs szerepét veszi fel.

Előnye ennek a módszernek, hogy elkerüljük az ARG teljesen kifejtését, ami akár végtelen is lehet. Azonban mivel ismeretlen heurisztika esetén a korábbi iterációban az adott csúcshoz tartozó csúcstól mindig új A* keresést indítunk, ezért az egymást követő keresések során többször is meglátogathatjuk ugyanazon csúcsokat, ha a korábbi keresések nem tudták meghatározni számukra a távolságértéket. Azonban a csúcsok újra meglátogatása sokkal kisebb költségű lehet, mint egy csúcs kifejtésének költsége.

1. Algorithm n heurisztikájának előállítása

```
if Nincs korábbi ARG then
    return 0                                          ▷ BFS
end if
if d(provider(n)) ismert then
    return d(provider(n))
end if
Keresés(provider(n))          ▷ d(provider(n)) ezáltal ismert lesz
return d(provider(n))
```

3.5. Hierarchikus A* keresés teljesen igény szerinti ARG kifejtéssel

A Teljesen igény szerinti Hierarchikus A* változat a Részlegesen igény szerinti változatot fejleszti tovább a kifejtett csúcsok számának csökkentésében.

Az A* keresés során a konzisztens heurisztika használatának célja, hogy a prioritási sorból kikerült csúcs és a hozzátartozó távolság esetén tudhassuk, hogy a csúcsot az adott távolsággal érhetjük el legkorábban. A prioritási sorból a legkisebb $f(n) = g(n) + h(n)$ értékkel rendelkező csúcsok kerülnek ki.

További csúcsok kifejtését kerülhetjük el, ha a csúcsok heurisztikáját csak akkor számítjuk ki pontosan, vagy csak akkor pontosítjuk, ha arra szükség van, hogy a prioritási sornak meghatározzuk a legkisebb elemét.

Definíció 22 (Pontos heurisztika).
Adott n csúcs esetén $d(provider(n))$ értéke. ▪

Definíció 23 (Nem pontos heurisztika / Heuriszika alsó becslése *lb*).
Adott n csúcs esetén $lb \leq d(provider(n))$ alsó becslés. ▪

Ilyen lb alsó becslést úgy kaphatunk, ha egy célcsúcsot még nem elért keresést megszakítunk egy olyan állapotban, ahol a prioritási sor legkisebb elemének $f(n)$ értéke megegyezik lb-vel.

A Teljesen igény szerinti ARG kifejtés tehát a csúcsokra nem határozza meg a heurisztikát a prioritási sorba rakás előtt, hanem a prioritási sorból való kivétel során a csúcsok heurisztikáját addig pontosítja, amíg a legkisebb elem meg nem határozható. Egy adott csúcs heurisztikájának pontosítását a provider csúcs távolságának pontosításával teszi meg, mely a tőle indított félbehagyott keresés folytatását jelenti, míg az a várt, pontosított, nagyobb alsóbecslésű lb értéket nem adja. A nem pontos heurisztikával rendelkező csúcsok prioritási sorba való rendezésekor a nem pontos heurisztikát használja fel $f(n)$ értékként, így valójában az $f(n)$ érték is egy alsó becslést ad a pontos $f(n)$ értékkel szemben. A prioritási sorból való kivétel során a következő esetek fordulhatnak elő:

1. A sor (egyik) legkisebb eleme pontos: Ennél kisebb $f(n)$ értékű elem nem létezhet a sorban, hiszen abban az alsó becslések és a pontos értékek is legalább ekkorák, így ez az elem a legkisebb.

2. Az első elem alsó becsléses, a prioritási sor egy elemű: Nincs más elem, így olyan sincs akinek a pontos $f(n)$ értéke kisebb lehetne, így nem szükséges a pontos heurisztikát meghatározni, ez lesz a legkisebb elem.

3. Az első elem alsó becsléses, míg a második pontos: Mivel nem tudhatjuk, hogy az alsó becslés pontos értéke kisebb vagy nagyobb lesz-e a pontosnál, így az ahhoz tartozó keresést egészen addig kell folytatnunk, míg vagy az az által előállított pontosított alsó becslés nem éri el (vagy lépi túl) a második csúcs pontos értékét vagy amíg a keresés el nem ér egy célcsúcsot. Az előbbi esetben az eddigi első elem legalább akkora lesz mint a második pontos elem, így visszakapjuk az első esetet. Hasonlóan az utóbbi esetben a nem pontos heurisztika pontossá alakul, amely kisebb vagy egyenlő kell legyen a második elemnél a fentebb leírtak alapján, így szintén visszakapjuk az első esetet.

4. Az első és a második elem is alsó becsléses: Bár az összes első pontos elem előtt lévő alsó becsléses értéket lehetne egyesével a 3. eset szerint kezelni, azonban azok bármikor elérhetnek egy célcsúcsot, ami miatt sok felesleges csúcs kerülhetne kifejtésre. Példának okáért az első csúcs a legkisebb pontos elem értékéig pontosul, míg a második pontosítása során találunk előbb egy célcsúcsot, akkor a fordított sorrendben végrehajtott pontosítás során az első csúcs pontosítását a legkisebb pontos elem értékéig elkerülhettük volna. Emiatt a következő eljárásokat fogjuk végrehajtani, hogy csökkentsük a feleslegesen kifejtett csúcsok számát:

   - Ha az első elem kisebb alsó becslést ad a másodikhoz képest, akkor egészen addig pontosítjuk, amíg legalább akkora nem lesz. Ha nem találunk célcsúcsot közben, akkor vagy megkapjuk a következő esetet vagy a jelenlegi esetet kapjuk vissza. Utóbbi azért lehetséges, mert a pontosítás nem feltétlen egyesével növeli az alsó becslést, így a kisebb alsó becslésű elemből lehet a nagyobb alsó becslésű például.
   - Ha a két elem egyező alsóbecslésű, akkor az elsőt eggyel pontosítjuk (mely akár nagyobb pontosítást is eredményezhez, ahogy azt láttuk az előbbiekben). Ezzel vagy újra a mostani eset fog fennállni, hiszen lehet más elem is még az eredetileg második elem értékével, vagy pedig az előző esethez jutunk.

   Ezt az eljárást egészen addig hajtjuk végre, amíg a prioritási sorban a legkisebb pontos elem értékét el nem éri az azt megelőző összes alsó becsléses elem, vagy amíg ez egyik elem pontos nem lesz, mindkét esetben megkapva az első esetet. Utóbbi eset fog akkor is bekövetkezni, ha a prioritási sorban egyáltalán nincs pontos elem.

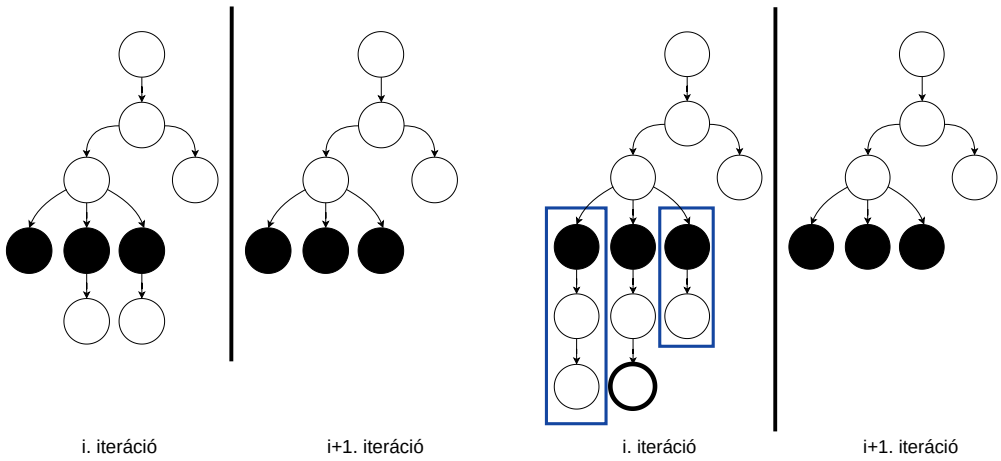
   Az első pontos elem értékének meghatározásához szükséges lenne a prioritási sorok elemeinek növekvő sorrendben való végig iterálására, ami nem hatékony, ezért a rendezést kiegészítjük azzal, hogy két azonos f érték esetén a pontos f értékeket soroljuk előbbre, ami által a korábban felsorolt esetek során a pontos értékű elemek mindig a sor elején lesznek, amikor azok relevánsak.

2. Algorithm n heurisztikájának előállítása sorbahelyezéskor

if Nincs korábbi ARG **then**
  return 0 ▷ BFS
end if
if $d(provider(n))$ ismert **then**
  return $d(provider(n))$
end if
return $h(provider(n))$ ▷ $d(provider(n))$ kezdeti alsó becslésnek felhasználható az arra vonatkozó heurisztika

3. Algorithm Prioritási sorból elem kivétele: remove()

$e \leftarrow sor$ ▷ Azonos e_f értékek esetén a pontos előbb szerepel
$n = e_{\text{csúcs}}$
if e_f pontos **then**
  return e
end if
if sor üres **then**
  return e ▷ nem feltétlen pontos
end if
if első$(sor)_f$ pontos **then** ▷ Sor első elemének lekérdezése, annak kivétele nélkül
  KeresésFolytatása($provider(n)$, első$(sor)_f$) ▷ e keresés folytatása első elem f értékéig
  $sor \leftarrow e$ ▷ e sorba berendezése
  return remove() ▷ Első elem pontos lesz
end if
if első$(sor)_f$ nem pontos **then** ▷ Legkisebb pontos érték ennél nagyobb (ha létezik)
  KeresésFolytatása($provider(n)$, első$(sor)_f + 1$)
  $sor \leftarrow e$
  return remove() ▷ Jobb rekurzív hívás
end if



(a) A 3 csúcs közül meg kell találni a legkisebb f értékűt, azaz ebben az esetben a korábbi iterációban a legkisebb távolságú providert

(b) Részlegesen igény szerinti változatban mindhárom részgráf kifejtésre került volna a legközelebbi célcsúcsig

3.4. ábra. Egy eset, ahol kevesebb csúcs került kifejtésre a Részlegesen igény szerinti változathoz képest

Ezen módszer előnye, hogy csökkentheti a kifejtendő csúcsok számát, mely a verifikációhoz szükséges idő jelentős részét teszi ki. Azonban a keresések gyakori megszakítása, illetve váltogatása többletköltséget jelenthet a futásidőre nézve.

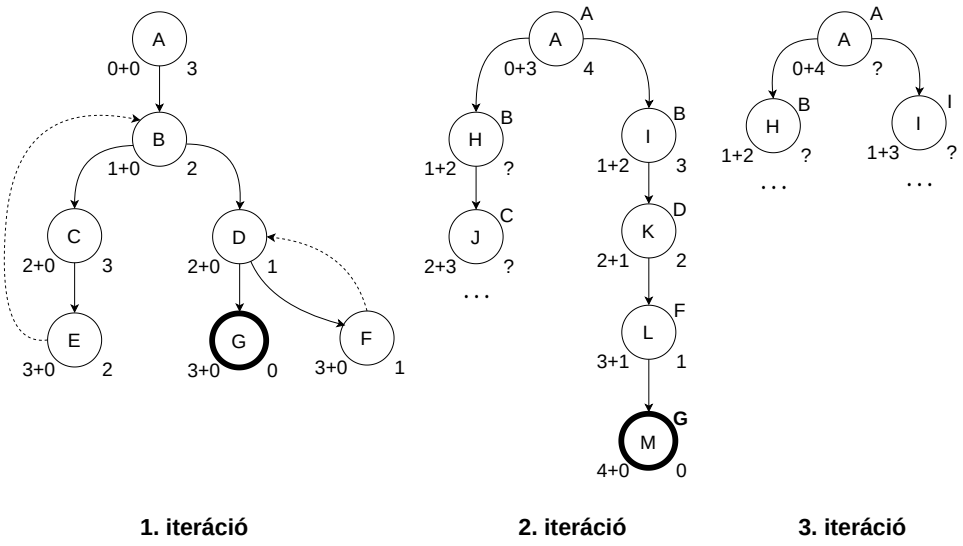
3.6. Hierarchikus A* keresés legközelebbi információval rendelkező providerrel

Ennek a módszernek a célja, hogy az igény szerinti kifejtés során előforduló korábbi iterációban található ARG-k továbbfejtése 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 továbbfejtésével járna.

Ezt a provider meghatározásának módosításával oldhatjuk meg.

Definíció 24 (Legközelebbi információval rendelkező provider). A legközelebbi információval rendelkező provider meghatározása n csúcsra iteratív eljárással történik n szülőjéből indulva. Először megvizsgáljuk, hogy az adott szülőnek providere létezik-e. Ha nem, akkor megáll az eljárás és nem lesz a csúcsnak providere. Ezután azt vizsgáljuk meg, hogy létezik-e a providerének egy $n' \succeq n$ gyereke, melyre a távolság értéke ismert. Ha igen, akkor azt állítjuk be heurisztikának, egyébként pedig az eljárást újrakezdjük a jelenlegi szülőtől. ▪

Az egyik problémája ennek az eljárásnak, hogy ha egy szülő gyerekének providere korábbi ARG-ben található, mint a szülő providere, akkor a gyerek providerének távolság értéke tetszőlegesen kisebb lehet mint a szülőé. 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.



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. A "..." jelzések felett található csúcsok még nem kerültek kifejtésre.

Az utolsó iterációban látható, hogy H csúcsnak az előző (2.) iterációból nem tudunk megfelelő providert választani, mivel a nála absztraktabb 2. iterációbeli H csúcs távolságértéke ismeretlen. Emiatt, követve az előbbiekben ismertetett eljárást, a providert az azt megelőző 1. iterációból választjuk, mely a B lesz. Azonban látható, hogy ezáltal a 3. iterációban az A csúcs heurisztikája 4 lesz, míg a H csúcsé 2, mely sérti a konzisztens heurisztikával szemben támasztott követelményt.

Egy másik probléma akkor fordul elő, ha nem találunk providert egy csúcsnak, hiszen ilyenkor heurisztikánk sem lesz. Bár erre az esetre láthatunk megoldást a 3.7. 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.

3.7. Hierarchikus A* keresés heurisztika csökkentéssel

A csökkentéssel történő heurisztika számítás a legközelebbi információval rendelkező providert használó változathoz hasonlóan a korábbi iterációjú ARG-kben történő további kifejtéseket hivatott elkerülni azzal, hogy csak a megismert távolságokat alapján határoz meg heurisztikát.

Amennyiben n esetében ismert $d(provider(n))$, akkor a korábbi heurisztikákkal egyezően ezt használjuk heurisztikának. Azonban ha az nem ismert, vagy $provider(n)$-t nem tudjuk meghatározni (mivel az ARG nincs kifejtve eléggé), akkor p_n heurisztikájának eggyel csökkentet változatát használjuk, kivéve ha az negatív érték lenne, hiszen a 0 egy mindig igaz alsó becslés, mely pontosabb a negatív értékeknél. Ezen heurisztika származtatás a heurisztika konzisztencia tulajdonságának megőrzéséből következik.

$$h'(n) = \begin{cases} max(h(p_n) - 1, 0) & \text{ha } h(p_n) = d(provider(n)) \text{ ismert} \\ max(h'(p_n) - 1, 0) & \end{cases} \tag{3.3}$$

Tétel 4. A csökkentéssel meghatározott h' heurisztikák alulról becslik a távolság érték alapú h heurisztikákat. ▪

Bizonyítás. Jelöljön n egy tetszőleges csúcsot és p_n annak szülő csúcsát. Definíció szerint ezek közé nem tartoznak a fedőélek csúcsai (9. definíció), de azokat nem is szükséges vizsgálnunk, hiszen a csökkentés nem fedőélen keresztül történik (3.3. képlet). $provider(n)$ és $provider(p_n)$ jelölje azon (struktúratartó) providereket, illetve $d(provider(n))$ és $d(provider(p_n))$ azon távolság értékeket, melyek kellő kifejtés esetén léteznének.

Mivel a gyökér csúcsnak biztosan ismert a heurisztikája, ezért minden h' heurisztikával rendelkező csúcs egyik felmenője rendelkezik h heurisztikával. A tételt erre alapozva teljes indukcióval látjuk be. A csökkentés definícióban (3.3. képlet) bár szerepel egy maximum képzés is, azonban mivel h távolság alapú, így ha h' 0 értéket vesz fel, akkor biztosan alulról becsli a nem negatív távolság értékeket. Emiatt a következőkben a maximum képzés másik argumentumát vizsgáljuk csak:

- A kiinduló esetben p_n heurisztikája h, míg n-é h' alapú. A távolság tulajdonsága alapján tudjuk, hogy:

  $$d(provider(p_n)) \leq d(provider(n)) + 1,$$

  amit átrendezve a bal oldalon h', míg jobb oldalon h definícióját kapjuk meg:

  $$\begin{aligned} d(provider(p_n)) - 1 &\leq d(provider(n)) \\ h(p_n) - 1 &\leq h(n) \\ h'(n) &\leq h(n) \end{aligned}$$

  Tehát h' ebben az esetben alulról becsli h-t.

- A többi esetben p_n és n is h' alapú heurisztikával rendelkezik. Hasonlóan az átrendezett távolság tulajdonságot alapul véve, majd a definíciókat felhasználva:

  $$\begin{aligned} d(provider(p_n)) - 1 &\leq d(provider(n)) \\ h(p_n) - 1 &\leq h(n) \\ h(p_n) &\leq h(n) + 1 \end{aligned}$$

  Majd a szülőre feltételezve, hogy teljesül rá a bizonyítandó tulajdonság:

  $$\begin{aligned} h'(p_n) &\leq h(p_n) &&\leq h(n) + 1 \\ h'(p_n) - 1 &\leq h(p_n) - 1 &&\leq h(n) \end{aligned}$$

  Ekkor láthatjuk, hogy a bal oldalon h' definíciója található:

  $$h'(n) \leq h(p_n) - 1 \leq h(n)$$

  Amiből már látható, hogy ha szülőre teljesül a tulajdonság akkor a gyerekre is.

Ezzel teljes indukcióval beláttuk a tulajdonságot.

Habár a d értékek lehetnek végtelenek is, azonban a rákövetkező iterációban kiszűrjük a végtelen heurisztikájú csúcsokat, így ezt az esetet nem kell vizsgálni, hiszen abból sose történik csökkentés. ▪

Tétel 5. Az csökkentéssel meghatározott h' heurisztikák konzisztens heurisztikát alkotnak a távolság alapú h heurisztikákkal együtt. ▪

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.

- p_n és n heurisztikája h alapján: Ezt már beláttuk, hogy konzisztens lásd 2. tétel.
- p_n heurisztikája h és n-é h' alapján: Kiindulva a konzisztencia definíciójából (17. definíció), majd ekvivalens átalakításokat végezve, felhasználva hogy itt $h'(n) = h(p_n) - 1$:

$$\begin{aligned} h(p_n) - h'(n) &\leq 1 \\ h(p_n) - (h(p_n) - 1) &\leq 1 \\ 1 &\leq 1 \end{aligned}$$

- p_n és n heurisztikája h' alapján: Az előzőhöz hasonlóan, azonban most $h'(n) = h'(p_n) - 1$:

$$\begin{aligned} h'(p_n) - h'(n) &\leq 1 \\ h'(p_n) - (h'(p_n) - 1) &\leq 1 \\ 1 &\leq 1 \end{aligned}$$

- p_n h' és n h alapján: Ez az eset nem fordulhat elő, lásd 17. definíciót.

A konzisztencia 2. feltételének bizonyítását heurisztika számítási módszerek szerint vizsgáljuk külön:

- Távolságon alapuló heurisztikánál már láttuk, hogy egy m célcsúcs esetén $h(m) = 0$ a 2. bizonyításban.
- Csökkentésen alapuló heurisztikák esetében, mivel azok alulbecslik a távolságon alapulókat ($h'(m) \leq h(m)$), illetve azok értéke legalább nulla ($h'(m) \geq 0$), ezért $h(m) = 0$-t felhasználva beláthatjuk, hogy $h'(m) = 0$. ▪

4. Algorithm n heurisztikájának előállítása

```
if Nincs korábbi ARG then
    return 0                                          ▷ BFS
end if
if d(provider(n)) ismert then                         ▷ Gyökér csúcs esetén ez mindig teljesül
    return d(provider(n))
end if
return max(h(p_n) − 1, 0)                             ▷ Szülő heurisztikája már ismert ekkor
```

A módszer előnye, hogy a heurisztikát konstans időben és könnyen elő tudja állítani, mivel nem foglalkozik a korábbi ARG-k továbbfejtésével, azokban nem végez további keresést. Azonban, ha korábbi ARG-ben kevés csúcsra ismerjük a távolság értékét, akkor sok helyen kell alkalmaznunk ezt a módszert, ami pontatlan becslést adhat. Kellő számú csökkentés után a heurisztikák értéke 0 lesz, ami a BFS-sel egyezik meg.

4. fejezet

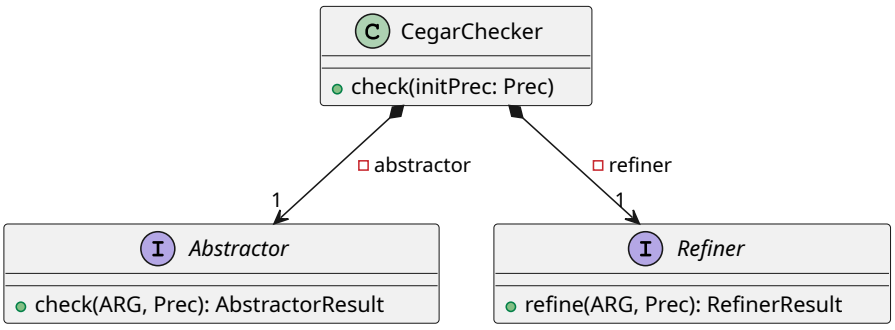
Architektúra

A korábban bemutatott Hierarchikus A* változatok számára kidolgoztam egy stabil architektúrát az algoritmuscsalád Thetába[1] [19] integrálásához. Ezen architektúra lehetővé teszi, hogy azt könnyen kiegészítsük jövőbeli Hiearchikus A* változatokkal.

A következő szekciókban az architektúrához tartozó UML ábrák, illetve az azokhoz tartozó magyarázatok kerülnek bemutatásra.

4.1. CEGAR hurok

A CEGAR hurok magas szintű működését (2.5. ábra) megvalósító Thetabeli osztály a *CegarChecker*. Ez tartalmaz egy *Refiner* implementációt, mely az absztrakció pontosságának (*Prec*) finomításáért felel. Az absztrakt állapotteret leíró gráf (ARG) kifejtését pedig egy *Abstractor* implementáció végzi el.



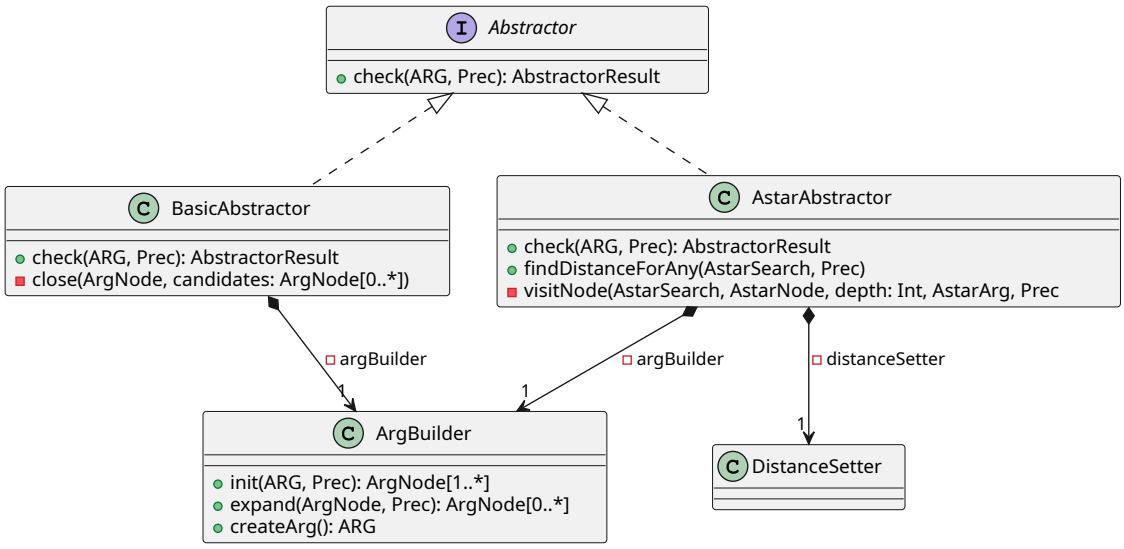
4.1. ábra

A Hierarchikus A* változatok ezt a már meglévő CEGAR hurok megvalósítást használják.

4.2. Hierarchikus A* Abstractor

Mivel a gráf kifejtésének a módját az *Abstractor* kezeli, így a Hierarchikus A* egy saját *Abstractor* implementációként lett megvalósítva.

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



4.2. ábra

Az *Abstractor* interface által megkövetelt *check* függvény a gyökér csúcsra meghívja a $findDistanceForAny$ metódust, mely azért felelős, hogy kifejtse a gráfot, amíg nem talál célcsúcsot. Az *Abstractor* a gráfkifejtés leállítását a keresést leíró $AstarSearch$ osztályra, míg a keresés végeztével történő távolságok beállítását a $DistanceSetter$-re bízza. Lefutásának végeztével a gyökér csúcs távolságértékét vizsgálva megállapítható, hogy a gyökér csúcs elért-e célcsúcsot. A csúcsok heurisztikájára az $AstarSearch$-nek van szüksége, így annak kiszámításának kezelése ott történik.

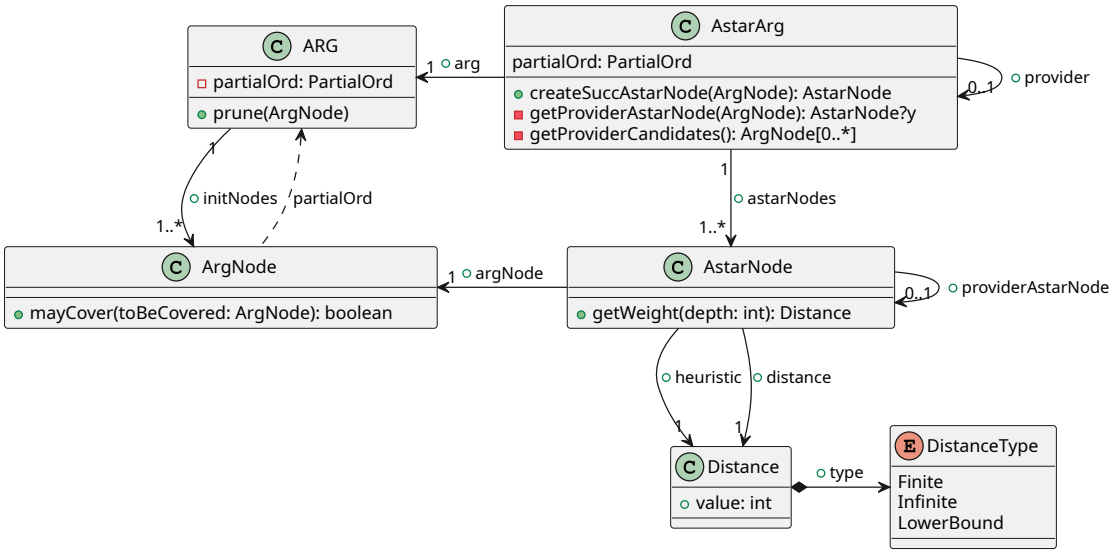
Az $ArgBuilder$ komponens feladata az ARG csúcsainak létrehozása, így például kifejtéskor a benne található *expand* metódus kerül felhasználásra.

4.3. Hierarchikus A* specifikus ARG

A Hierarchikus A* működéséhez szükséges az ARG-kben a távolságértéket tárolni. Mivel a Theta lényege, hogy egy moduláris keretrendszert biztosítson formális verifikációra, ezért a jelenlegi ARG-t reprezentáló osztály A* specifikus dolgokkal való módosítása nem lehetséges.

Mivel a jelenlegi ARG implementációt akarjuk specializálni annak módosítása nélkül, ezért az első megvizsgált megoldási lehetőség az öröklés volt. Azonban ehhez a keretrendszerben túl sok módosítást kellett volna végrehajtani, ami kihatott volna sok más komponensre. Illetve ennek meglépése esetén a Java generikus rendszer tulajdonságai miatt sok művelet esetén (például: A* típusú gyerek csúcsok elérése) új generikus típus felvétele nélkül (mely a keretrendszer számos komponensére kihatott volna) castolásra lett volna szükség.

Ehelyett a Hierarchikus A*-hoz tartozó ARG reprezentációja adapter mintával lett implementálva ($AstarArg$), mely nincs hatással a keretrendszer többi részére. Ennek hátránya, hogy a tartalmazott osztályhoz a hozzáférés közvetett, illetve az ARG-kre a keretrendszer által alkalmazott műveleteket egyes esetekben az $AstarArg$-okra is végre kellett hajtani külön lépésben.



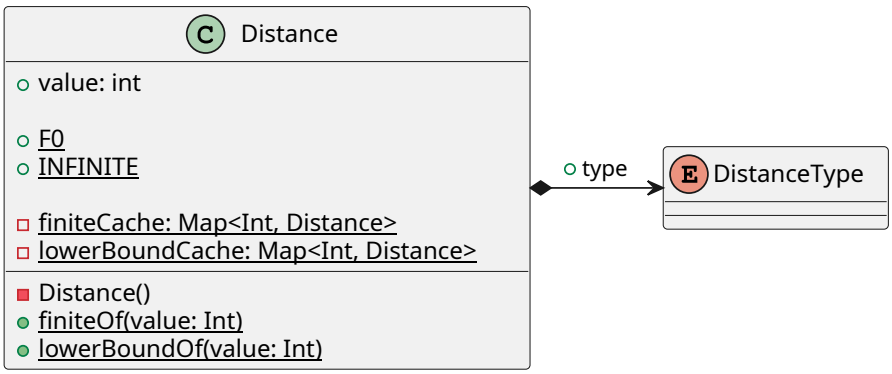
4.3. ábra. Adapter minta alkalmazása

Az *AstarArg* osztály *getProviderAstarNode* metódusa állapítja meg a hozzá tartozó korábbi iterációban lévő provider csúcsot, a 21. definícióban leírtak szerint, felhasználva az absztrakció részbenrendezést (*partialOrd*).

Az *AstarArg* egyes csúcsait reprezentáló *AstarNode* *getWeight* függvénye az A* keresés során a csúcsok sorrendezéséhez van használva (2.1. képlet). Fontos kiemelni, hogy ehhez szükség van egy mélység értékre, mely keresésenként eltérő lehet ugyanarra a csúcsra.

4.4. Távolságot leíró komponens

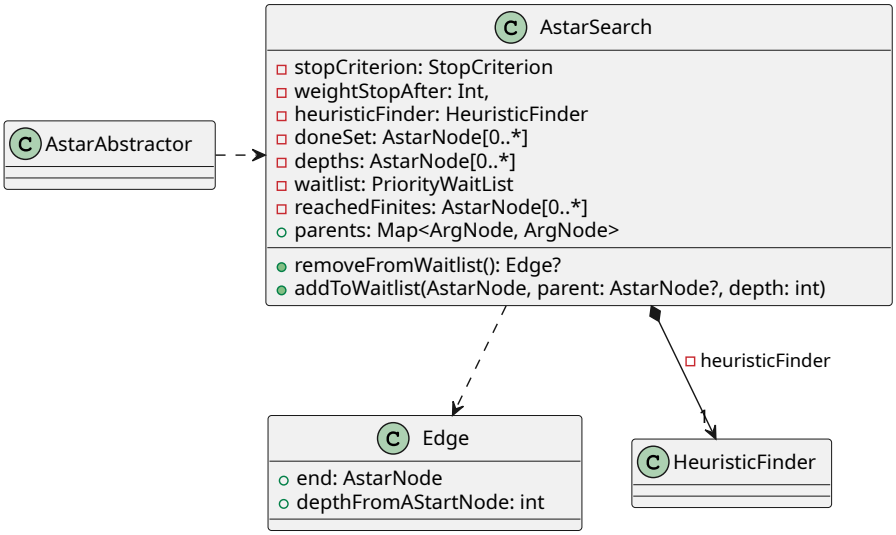
Mivel az ugyanazt reprezentáló távolság (*Distance*) példányok többször is felhasználásra kerülhetnek különböző csúcsoknál, ezért memóriahasználat szempontjából előnyös ezen értékeket csak egyszer létrehozni. Ennek érdekében az egyféle értékkel rendelkező típusokat statikus tagként (végtelen, pontosan 0 távolságú), míg a végtelen lehetséges értékkel rendelkező *Finite* (pontos, nem végtelen) és *LowerBound* (alsó becsléses) típusúakat statikus factory mintával, illetve cacheléssel kínálja ki az osztály. Ehhez biztosítani kellett, hogy az osztály immutable legyen.



4.4. ábra. Statikus factory cacheléssel

4.5. A* keresést leíró komponens

Az adott A* keresés állapotát leíró komponens (*AstarSearch*) szétválasztásra került az *Abstractor* komponenstől, hiszen más feladatot látnak el. Ennek kódszervezésen kívül az is az előnye, hogy az egyes kereséseket meg tudjuk szakítani, majd folytatni, ha tároljuk, majd visszaállítjuk az adott *AstarSearch* példányt.



4.5. ábra

Amikor elérünk egy új csúcsot és azt berakjuk a prioritási sorba az *addToWaitlist* segítségével, akkor meghatározásra kerül annak a csúcsnak a heurisztikája a keresési stratégiának megfelelő *HeuristicFinder* segítségével. Igény szerinti kifejtés esetén ez az Abstractorban található *findDistanceForAny* hívással járhat a csúcs providerétől indítva, hiszen heurisztikának a korábbi iterációbeli provider csúcs távolságértékét használja fel ezen algoritmus változat.

Korábbi iterációban indított keresés esetén elérhetünk már ismert, korlátos távolság értékű csúcsokhoz. Ilyenkor, hogy ezekhez a csúcsokhoz tartozó részgráfokat ne járjuk be újra, a csúcsot berakjuk a prioritási sorba, azonban itt a csúcsnak az ismert távolságát használjuk fel heurisztikának. Majd amikor az soron következőként kikerül a sorból, akkor ahogy egy célcsúcsot, ezt is eltároljuk a *reachedFinites*-ban. Amikor egy csúcsot a *reachedFinites*-ba helyezünk, akkor az *AstarSearch* osztály értesül arról, hogy elértük a következő legközelebbi célcsúcsot. Ezzel a módszerrel egyszerűen megvalósítottuk a 3.2-ben bemutatott keresési korlátot.

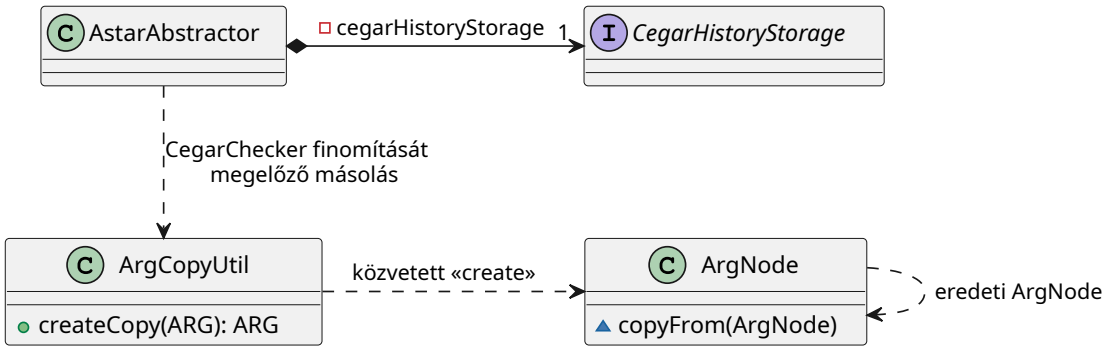
A *stopCriterion* a *reachFinites* alapján tudja eldönteni, hogy elég célcsúcsot elért-e a keresés annak befejezéséhez. Ez Teljes ARG kifejtéses változat esetén sose okozza a keresés leállítását. A többi esetben ez a Theta paraméterezésétől függ egy adott ARG-ben a gyökér csúcstól indított keresés során, míg amikor egy korábbi iterációban keresünk, mindig leállunk amint nem üres a *reachedFinites*.

A komponens többi adatváltozója megfeleltethető egy tipikus A* keresés implementáció során használt adatváltozóknak.

4.6. CEGAR iterációk tárolása

Mivel a CEGAR iterációk végén elveszik az akkori ARG, illetve absztrakciós pontosság (*Prec*), ezért szükséges előtte egy másolatot készíteni róluk és eltárolni őket, hogy tudjunk

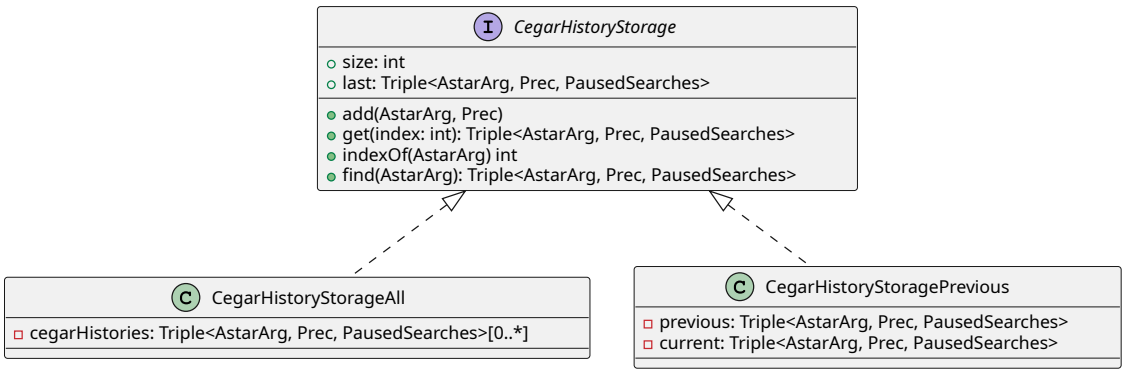
a korábbi iterációban megtalálható távolságértékeket heurisztikaként felhasználni, illetve azokban újabb kereséseket indítani. Ehhez egy, a *CegarHistoryStorage*-nak megfelelő implementációt használhatunk. Ez szintén tárolja a megszakított kereséseket is, mely a Teljesen igény szerinti változat használ fel.



4.6. ábra

Mivel a Thetában egyetlen ARG példányon hajtódik végre a CEGAR hurok, ezért minden iteráció végeztével le kell másolni az adott ARG-t. A másolást az *ArgCopyUtil* végzi el, ami a csúcsok létrehozása után a csúcsokon meghívja a *copyFrom* függvényt, paraméterül átadva az eredeti csúcsot. Erre azért van szükség, mert az *ArgNode* osztály egyes állapotait (például a kifejtettséget) a Theta keretrendszerben csak az adott csomagbeli osztályok tudják módosítani.

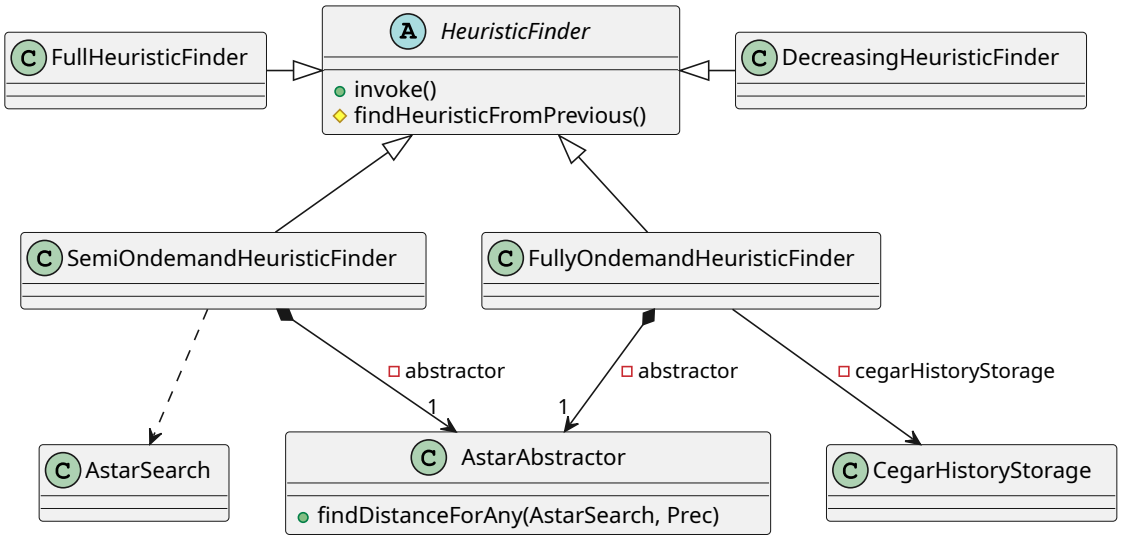
Mivel a Teljes ARG kifejtéses, illetve a Csökkentéses Hierarchikus A* változat esetén csak a jelenlegit megelőző iterációt használjuk fel, ezért a *CegarHistoryStorage*-ból készült egy olyan implementáció is, mely a többi iterációt nem tárolja el. Ez az iterációnként egyre nagyobb méretű *ARG*-k miatt egy fontos tárhely igény optimalizáció tud lenni. Itt fontos, hogy az *AstarArg*-ben a providerekre mutató referenciákat is töröljük, hogy a korábbi ARG-kre ne legyen referencia.



4.7. ábra

4.7. Hierarchikus A* változatok támogatása

A különböző Hiearchikus A* változatok számára szükséges, hogy a heurisztika megállapításakor az ahhoz tartozó megvalósítás hajtódjon végre. Ehhez Strategy minta lett használva a korábban is látott *HeuristicFinder*-nél.



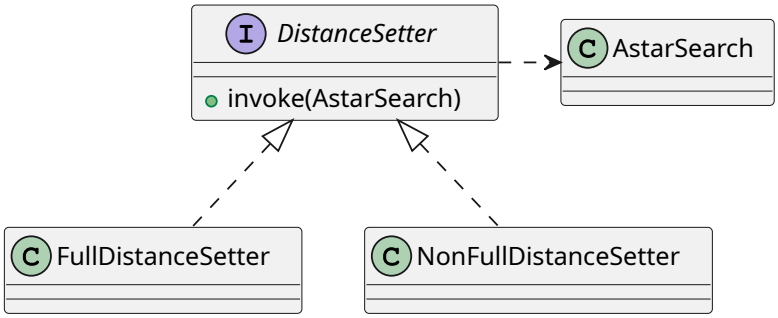
4.8. ábra

Ennek során az ősosztály lekezeli azon helyzeteket, mely a Hierarchikus A* megvalósításoktól függetlenek, mint például, hogy $d(provider(n))$ ismert, vagy hogy nincs előző iteráció. Ha ezután sincs elérhető heurisztika, akkor a $findHeuristicFromPrevious$ hívásával a $HeuristicFinder$ leszármazottak megállapítják a hozzájuk tartozó Hierarchikus A*-nak megfelelően a heurisztikát.

Teljesen-, illetve Részlegesen igény szerinti változat esetében ($Fully-$, $SemiOndemand$) ez azt jelenti, hogy a providertől keresést kell indítanunk, vagy folytatnunk kell azt, így ezen leszármazottak elérik az $AstarAbstractor$-t. Részlegesen igény szerinti változat esetében ehhez létre kell hoznunk egy új $AstarSearch$-t, míg Teljesen igény szerinti esetében a korábban leállított kereséseket kell felhasználnunk, melyet a $CegarHistoryStorage$ tartalmaz.

Teljes ARG kifejtéses változat esetében ($Full$) a $findHeuristicFromPrevious$-nak nem szabad meghívódnia, míg a Csökkentéses változat esetében ($Decreasing$) ez egy egyszerű műveletet végzést jelent (3.3. képlet).

Mivel Teljesen kifejtett ARG esetén az összes csúcsra tudunk mondani távolságértéket, ezért a távolságbeállító esetében is Strategy mintát alkalmaztam, mely a Hierarchikus A* változattól függően állítja be a távolságokat egy keresés végeztével.



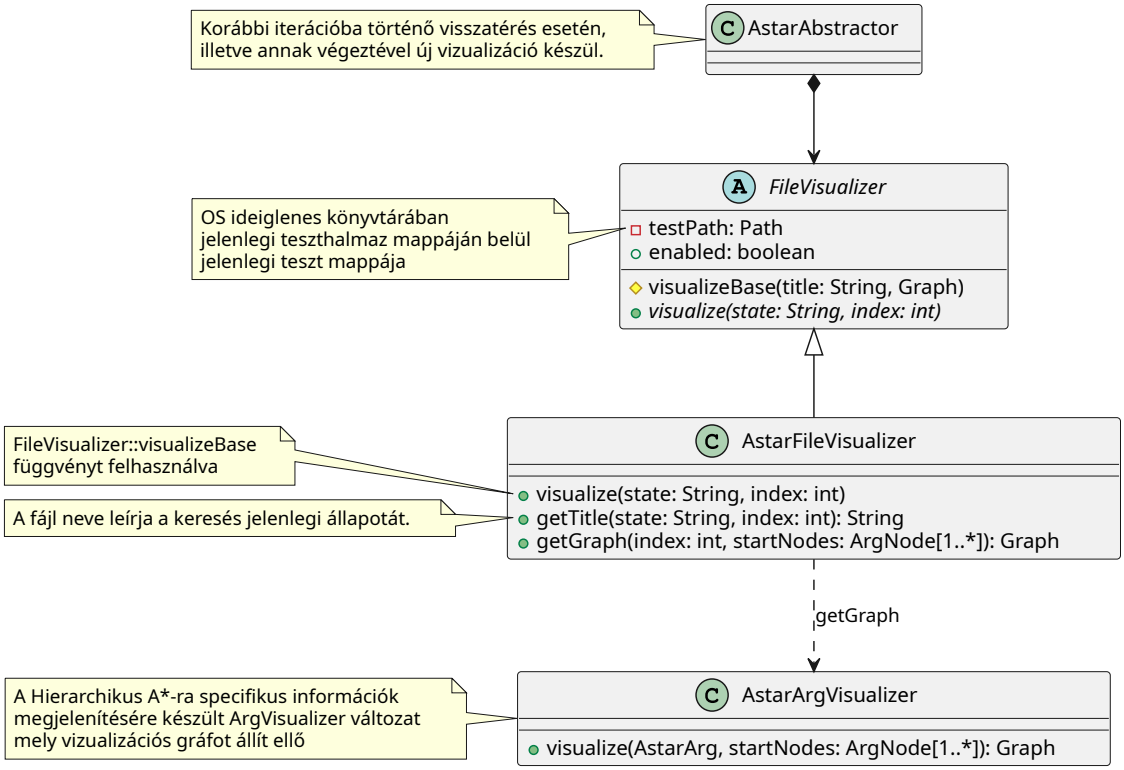
4.9. ábra

4.8. Hibakeresés támogatása

Az Igény szerinti Hierarchikus A* változat esetében szükséges, hogy könnyen áttekinthetőek legyenek a rekurzív működés során történt változások. Ezért a Thetában megtalálható *ArgVisualizer* osztály, mely elősegíti a *Graphviz* [2] vizualizációt, kiegészítésre került a Hierarchikus A* komponensekhez tartozó információk kiírásával (*AstarArgVisualizer*). Emellett hasznos volt egy olyan komponens létrehozása, mely ezeket a vizualizációkat fájlba is írja strukturált módon több teszteset futtatása esetén.

Az *AstarAbstractor* a fájlvizualizációs komponens segítésével a korábbi iterációkba visszatérés előtt, illetve annak végeztével is létrehoz egy fájlvizualizációt, ha az engedélyezve van (*enabled*). Alapértelmezetten ez a funkció le van tiltva, hiszen a nagy fájlműveletek jelentősen lassíthatják a verifikáció végrehajtását. A létrehozott fájlok nevei a fájlvizualizációs komponensnek átadott állapotleíró szövegek alapján kerülnek meghatározásra, melyek jelzik, hogy jelenleg melyik iterációban hajtunk végre keresést, illetve abban mi történt éppen.

A beépített ARG vizualizáció kiegészítésre került a gyerek nélküli, de kifejtett csúcsok, illetve a korábbi iterációból finomítás után fennmaradt csúcsok jelzésével (utóbbi nem került a dolgozat során tárgyalásra). Szintén kiegészültek a csúcsok a hozzájuk tartozó A* csúcsok leírásával (távolság, heurisztika, provider), illetve a keresésben szereplő csúcsok esetén az azok keresésbeli adataival (mélység, listába berakáskori heurisztika).



4.10. ábra

[2] Gráf leírást vizualizáló szoftver: `https://gitlab.com/graphviz/graphviz`

5. fejezet

Mérések

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

5.1. Megvalósítás és mérési környezet

A Hierarchikus A* algoritmus változatokat a nyílt forráskódú moduláris Theta[1] [19] modellellenőrző keretrendszerbe fejlesztettem, mely számos moduláris elemet tartalmaz, mint például az absztrakciók, a finomítási eljárások vagy a keresési stratégiák. Az algoritmushoz szükséges bővítéseket a 4. fejezet mutatja be részletesebben.

A mérésekhez használt bemeneteket a Software Verification Competition [3] (SV-Comp) benchmark részhalmaza, illetve az FTSRG kutatócsoport által biztosított, többek között ipari partnerektől származó belső XSTS modellek [16] alkotják. A mérések végrehajtásához a BenchExec [5] került felhasználásra, mely segít a felhasznált erőforrások megbízható mérésében.

A Theta modellellenőrző keretrendszer sok különböző konfigurációs paraméterrel rendelkezik, azonban a mérés célja a Hierarchikus A* változatok kiértékelése volt, ezért azoknak azon paraméterkombinációja került felhasználásra (a keresési algoritmusoktól eltekintve), melyek a Thetán végzett korábbi mérések tapasztalatai alapján hatékonyan teljesítenek.

A Hierarchikus A* változatok összehasonlításra kerültek a Thetába már beépített BFS és ERR keresési algoritmusokkal. A BFS (Szélességi keresés) biztosítani tudja számunkra a legrövidebb ellenpéldát a Hierarchikus A*-hoz hasonlóan. Az ERR keresés hasonlóan tudja ezt biztosítani, illetve az a CFA struktúra alapján számít heurisztikát, a jelenlegi hely és a legközelebbi hibás hely távolsága alapján, nem véve figyelembe az elágazási feltételeket. Azonban a Hierarchikus A*-gal ellentétben ez nem finomítja a heurisztikát a futás során dinamikusan.

Mivel a modellek verifikálása sok ideig is eltarthat, ezért azokhoz 15 perces időkorlát volt rendelve, ez idő alatt kellett tudniuk az egyes paraméterkombinációknak az adott modellekkel szemben támasztott követelmény teljesülését eldönteni. Amennyiben a verifikáció túllépi az időkorlátot, úgy a verifikáció sikertelennek számít.

Az egyesek mérések bemutatása során a következő rövidítéseket használom:

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

Rövidítés	Feloldás
FULL	Teljes kifejtéssel történő változat
SEMI	Részlegesen igény szerinti változat
FULLY	Teljesen igény szerinti változat
DECR	Csökkentéssel történő változat

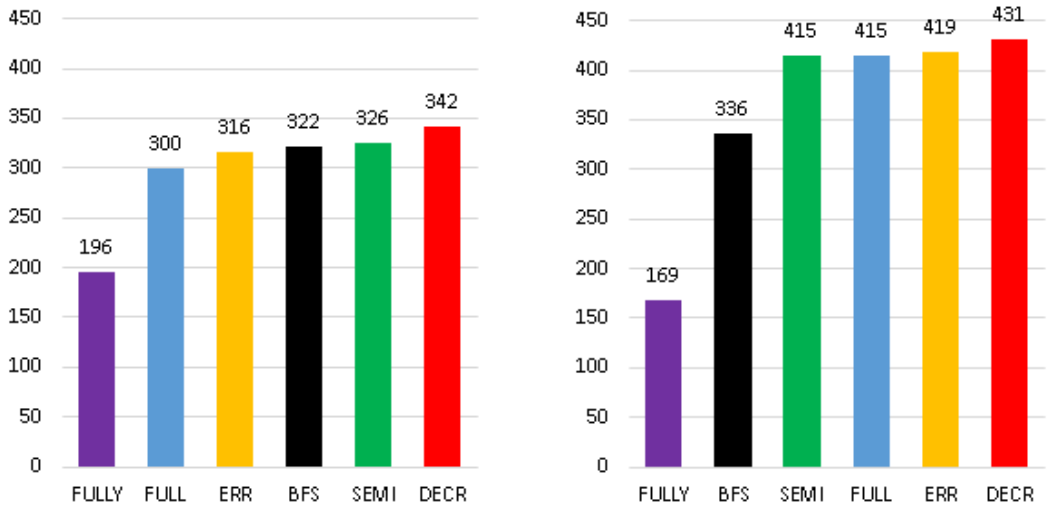
5.2. XCFA formális modelleken végzett mérések eredményei

Ezen szekció az SV-Comp benchmark halmaz C programjaiból származtatott kibővített CFA (XCFA) formális modelleken történő verifikáció eredményeit mutatja be.

A mérés során felhasznált paraméterkombinációkat az alábbi táblázat mutatja be soronként. Ezen paraméterkombinációk kiegészültek az összes Hierarchikus A* változattal, illetve a Thetába beépített BFS, illetve ERR kereséssel, összesen $2 * 6$ paraméterezést létrehozva.

#	Absztrakció	Finomítás
1	EXPL	SEQ_ITP
2	PRED_CART	BW_BIN_ITP

Ezen paraméterkombinációk mindegyikével megkísérlésre kerültek a korábban említett modellek időkorláton belüli verifikálása.



(a) 1. konfiguráció **(b)** 2. konfiguráció

5.1. ábra. Az egyes keresési stratégiák által sikeresen verifikált modellek száma.

Látható, hogy az 1. konfigurációban mindkét beépített keresési stratégiánál több verifikációt volt képes megoldani a SEMI és a DECR változat az adott időkorlát mellett. A 2. konfigurációban ezzel szemben csak a DECR volt képest jobban teljesíteni, azonban a BFS-nél ebben az esetben a FULL is jobban tudott teljesíteni.

Szintén az is látható, hogy a FULLY nagy mértékben rosszabbul teljesít a beépített, illetve a többi Hierarchikus A* változatokkal szemben mindkét konfigurációban. Ebből arra a következtetésre juthatunk, hogy a keresések gyakori megszakítása nagyobb teljesítményromlást okoz, szemben az az által elkerült kifejtések okozta teljesítménynövekménnyel.

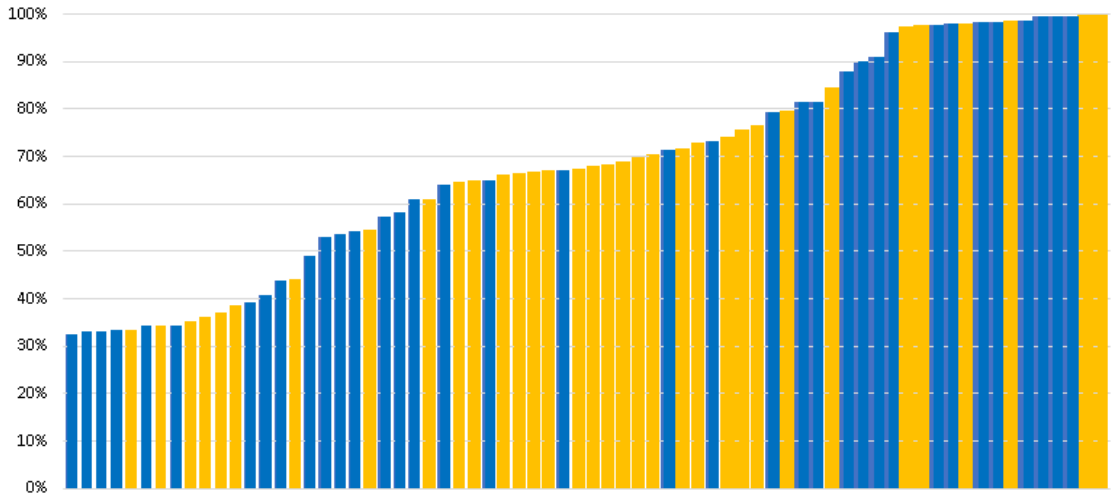
Kiegészítve a Thetát az adott verifikáció során történt csúcs kifejtési metrikával, a következő adatokat kapjuk azon keresési stratégiától független paraméterezés és modell

párokat vizsgálva, amelyekkel együtt a SEMI, illetve a FULLY is sikeresen tudott verifikálni:

Konfiguráció	Összesen	FULLY kevesebb csúcsot fejtett ki	Medián kifejtés csökkenés
1.	189	34 (18%)	68%
2.	158	36 (23%)	66%

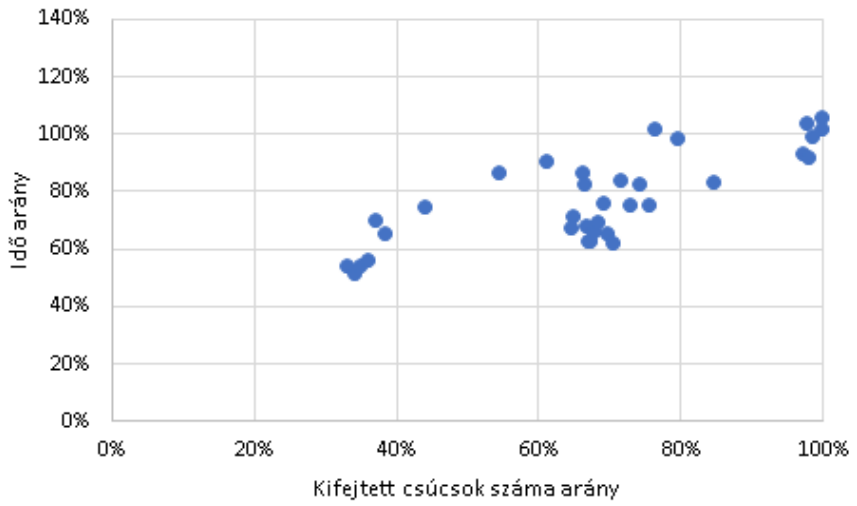
Látható, hogy kevés esetben tudott valójában csökkenést okozni a kifejtések számában a FULLY változat, mely szintén hozzájárul a 5.1.-ben látott teljesítményromláshoz.

A kifejtés csökkenés mértékét részletesebben a következő diagram mutatja be ezen esetekre:

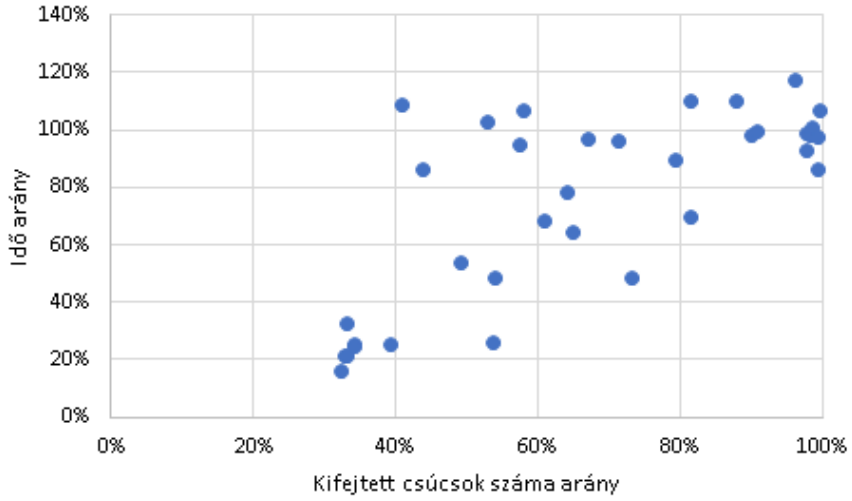


5.2. ábra. Sárga színnel az 1-es, míg kék színnel a 2-es számú konfigurációk esetében történt csúcs kifejtés csökkenés a FULLY változatban a SEMI-hez képest

Az előző értékekhez társítva a verifikációval töltött időt a következő scatter plot ábrákat kapjuk:



(a) 1. konfiguráció



(b) 2. konfiguráció

5.3. ábra. A kifejtett csúcsok számának arányához társított verifikációval töltött idő arányát bemutató scatter plot ábrák

Az 1. konfiguráció esetében elmondható, hogy bár a futásidő javul a kifejtés csökkentése által, azonban ez az összefüggés nem lineáris.

Azonban a 2. konfiguráció esetében megfigyelhető, hogy a 40% és 60% arányú kifejtés arány esetén is előfordulhat, hogy a futásidő nem javul, sőt valamivel rosszabb is lesz a SEMI-vel szemben. Ezen esetektől eltekintve itt közel jól látható a várt összefüggés.

5.3. XSTS formális modelleken végzett mérések eredményei

Az alábbiakban az FTSRG kutatócsoport által biztosított XSTS modellek bizonyos konfigurációkon való verifikációja kerül bemutatásra. Ezen formális modellek egy része különböző magasabb szintű állapot-alapú mérnöki modellekből származik, míg mások kézzel készültek tesztelési és teljesítménymérési céllal.

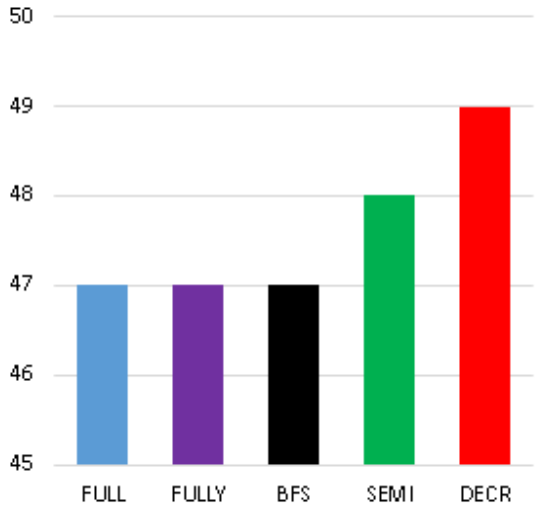
Az XCFA mérésekhez képest nagyobb számban mért konfigurációk miatt az eredményeket absztrakt domainenként elkülönítve mutatják be a következő részek. Az explicit (EXPL) és kombinált explicit-predikátum (EXPL_PRED_COMBINED) domainek esetében új konfigurációs paraméter a kifejtett gyerekek maximális száma, 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.

5.3.1. EXPL_PRED_COMBINED absztrakció

A mérés során felhasznált konfigurációk:

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

A sikeresen verifikált esetek a különböző konfiguráción nem mutatnak eltérést, a 2. konfiguráció kivételével, mely esetében a DECR változat eggyel több modellt volt képes verifikálni, így ez kerül bemutatásra:



5.4. ábra. Az egyes keresési stratégiák által sikeresen verifikált modellek száma.

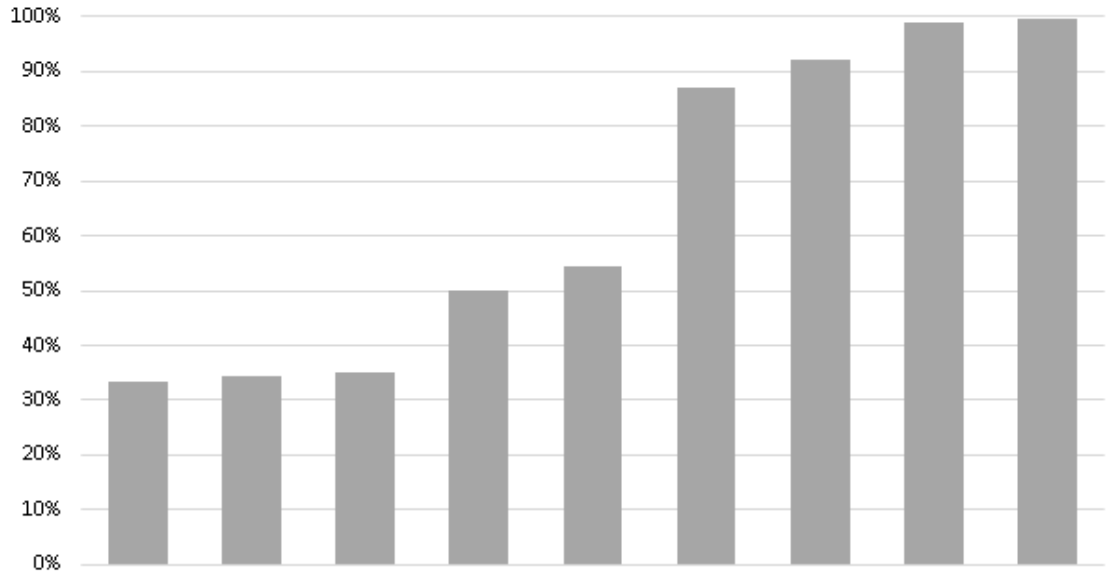
Ezen mérés során az egyes keresések között nem tapasztalható nagyobb eltérés, azonban SEMI és a DECR változat képes volt több verifikációt végrehajtani, míg a többi változat nem teljesített rosszabbul a BFS-hez képest.

Keresési stratégiától független paraméterezés és modell párokat vizsgálva, ahol a FULLY, illetve a SEMI változat is sikeresen képes volt verifikálni:

Konfiguráció	Összesen	FULLY kevesebb csúcsot fejtett ki	Medián kifejtés csökkenés
1.	45	9 (20%)	87%
2.	45	9 (20%)	55%
3.	45	9 (20%)	54%
4.	45	9 (20%)	55%

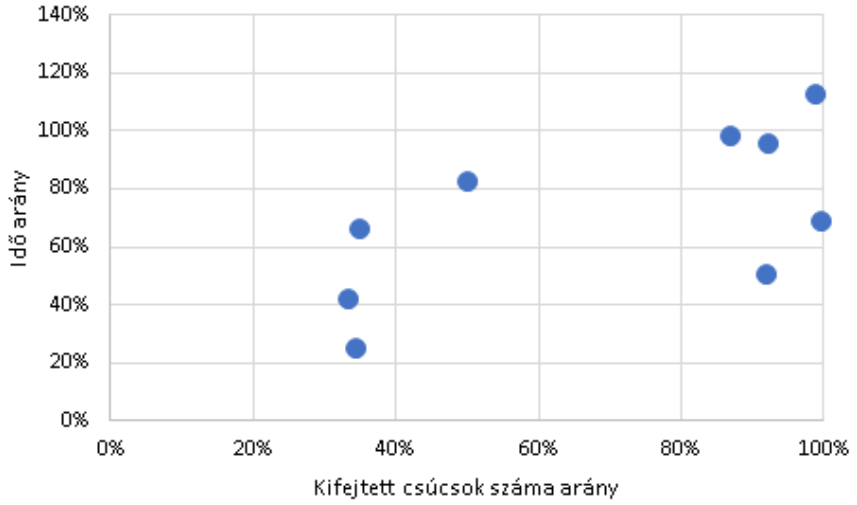
Hasonlóan az XCFA-hoz itt is kevés esetben volt képest kifejtés csökkenést elérni a FULLY változat.

A kifejtés csökkenés mértékét részletesebben a következő diagram mutatja be. Mivel mindegyik konfiguráció során ugyanazon mértékben csökkentették a kifejtett csúcsok számát, így azok egy konfiguráció eredményeinek ábrázolásán keresztül kerülnek bemutatásra.

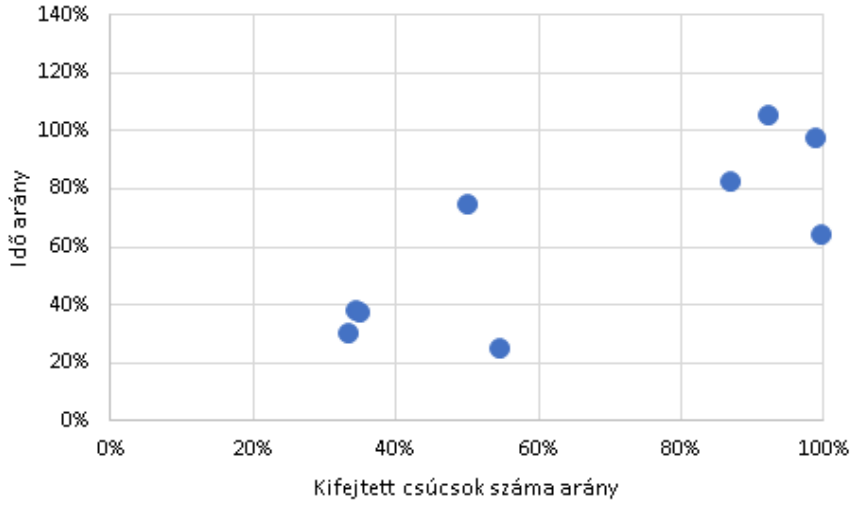


5.5. ábra. Csúcs kifejtés csökkenés a FULLY változatban a SEMI-hez képest

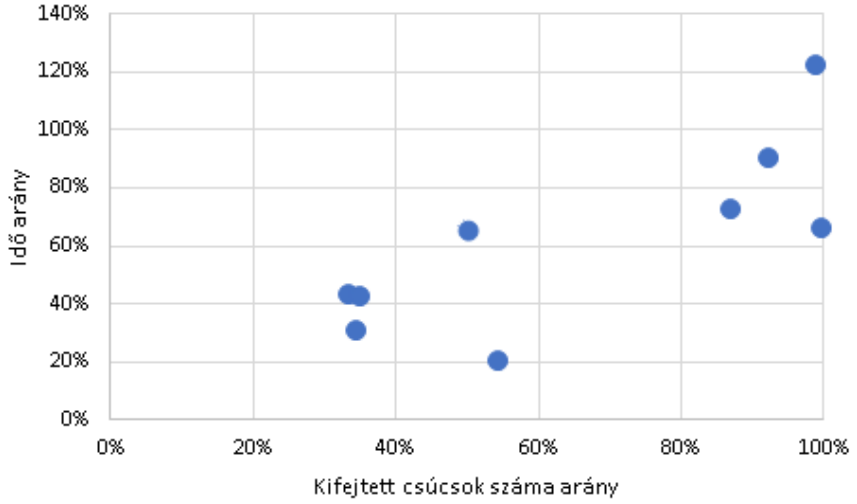
Azonban a futásidő arányoknál nem voltak azonosak az értékek, így a kifejtett csúcsok számának arányához társított verifikációval töltött idő arányát bemutató scatter plot ábra a 4 konfiguráción külön kerül bemutatásra:



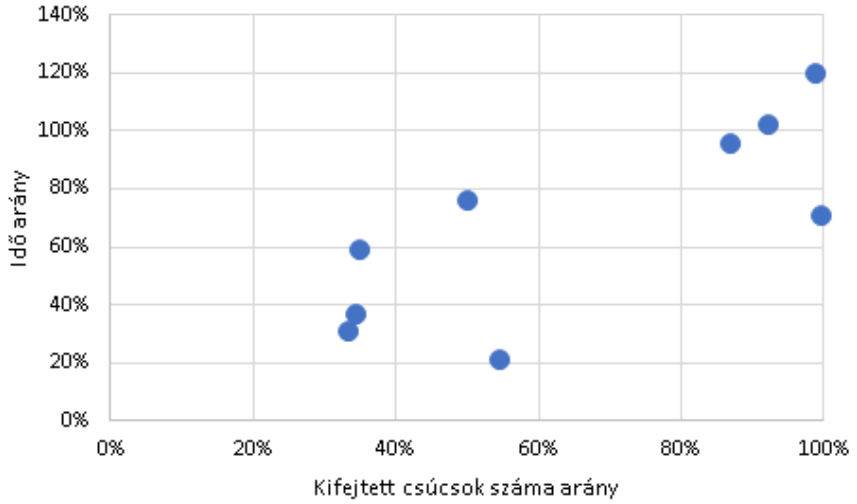
(a) 1. konfiguráció



(b) 2. konfiguráció



(c) 3. konfiguráció



(d) 4. konfiguráció

5.6. ábra. A kifejtett csúcsok számának arányához társított verifikációval töltött idő arányát bemutató scatter plot ábrák

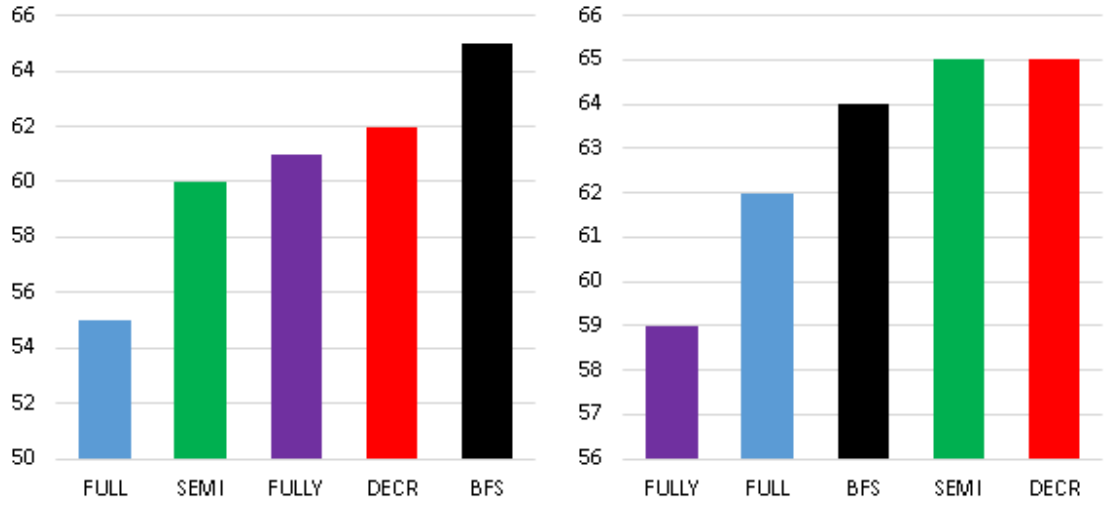
Az 1. konfiguráció esetében nem mutatható ki összefüggés a kifejtett csúcsok számának aránya és a futásidő aránya között.

5.3.2. PRED_CART és PRED_SPLIT absztrakció

A mérés során felhasznált konfigurációk:

#	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

A 3. és 4. PRED_CART absztrakciójú konfigurációkon belül nem volt eltérés a keresési algoritmusok között a sikeresen verifikált modellek számában (71, illetve 72 darab), illetve nem volt olyan eset, ahol a FULLY változat képes lett volna csökkenteni a kifejtett csúcsok számán, így a következő diagramok során ezek a konfigurációk nem kerülnek ábrázolásra.



(a) 1. konfiguráció **(b)** 2. konfiguráció

5.7. ábra. Az egyes keresési stratégiák által sikeresen verifikált modellek száma.

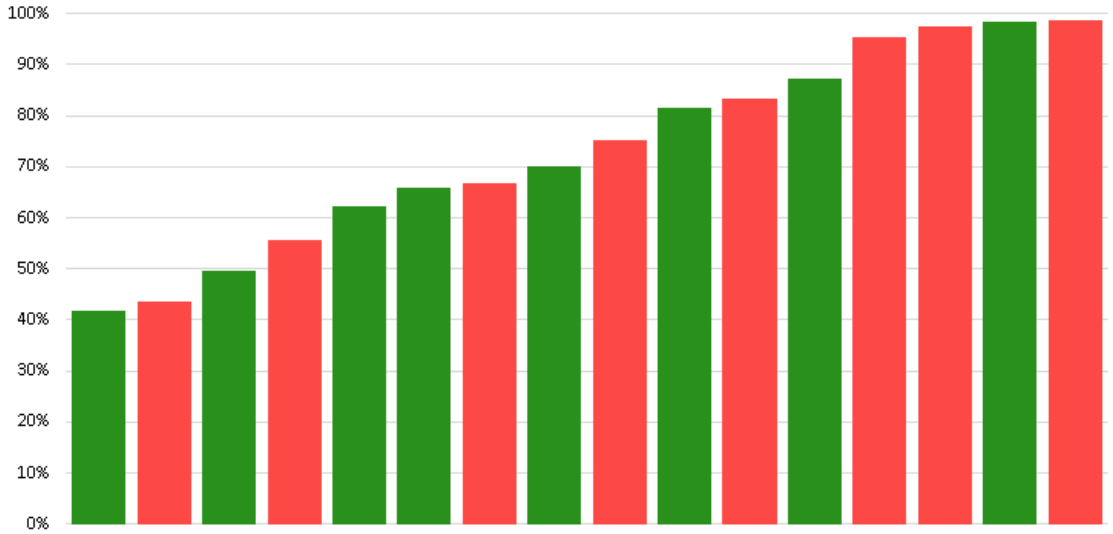
Bár a 2. konfigurációban az eddig megszokott eredmény tapasztalható, azonban az 1. konfiguráció esetében a FULLY képes volt felvenni a versenyt a SEMI változattal. Ettől függetlenül azonban az 1. változatban a Hierarchikus A* változatok rosszabbul teljesítettek a BFS-sel szemben.

Keresési stratégiától független paraméterezés és modell párokat vizsgálva, ahol a FULLY, illetve a SEMI változat is sikeresen képes volt verifikálni:

Konfiguráció	Összesen	FULLY kevesebb csúcsot fejtett ki	Medián kifejtés csökkenés
1.	49	8 (16%)	68%
2.	55	8 (15%)	79%

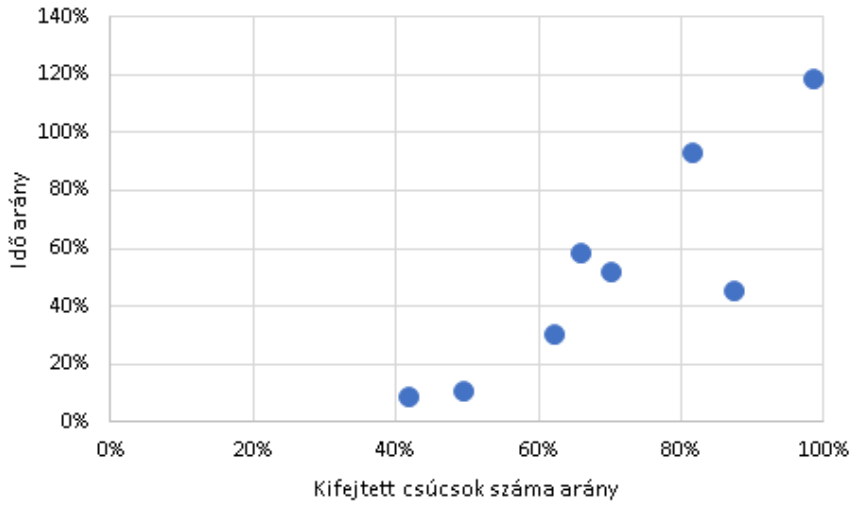
Hasonlóan az XCFA-hoz itt is kevés esetben volt képest kifejtés csökkenést elérni a FULLY változat.

A kifejtés csökkenés mértékét részletesebben a következő diagram mutatja be.

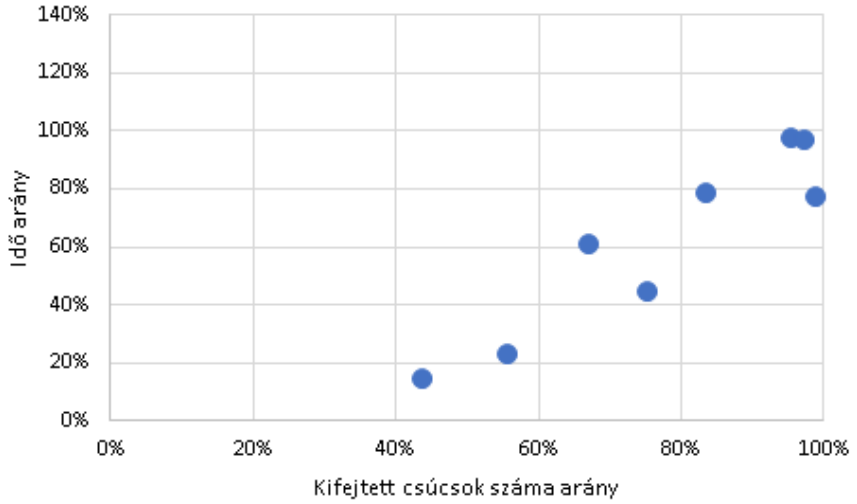


5.8. ábra. Zöld színnel az 1., míg prios színnel a 2. konfigurációk esetében történt csúcs kifejtés csökkenés a FULLY változatban a SEMI-hez képest

Az előző értékekhez társítva a verifikációval töltött időt a következő scatter plot ábrákat kapjuk:



(a) 1. konfiguráció



(b) 2. konfiguráció

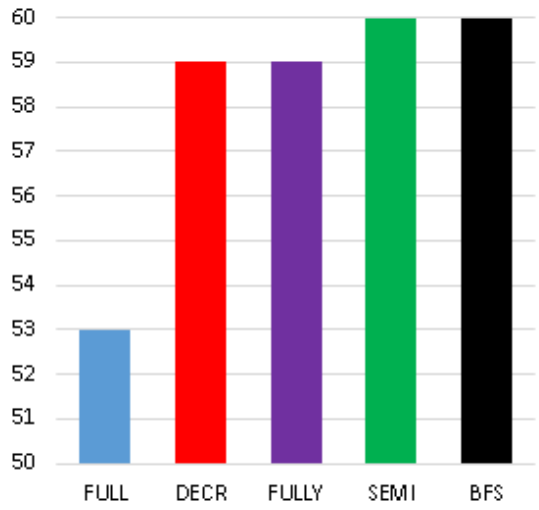
5.9. ábra. A kifejtett csúcsok számának arányához társított verifikációval töltött idő arányát bemutató scatter plot ábrák

Látható az elvárt összefüggés mindkét esetben, a kifejtett csúcsok csökkentése a futásidőt is csökkenti, méghozzá lineárisan.

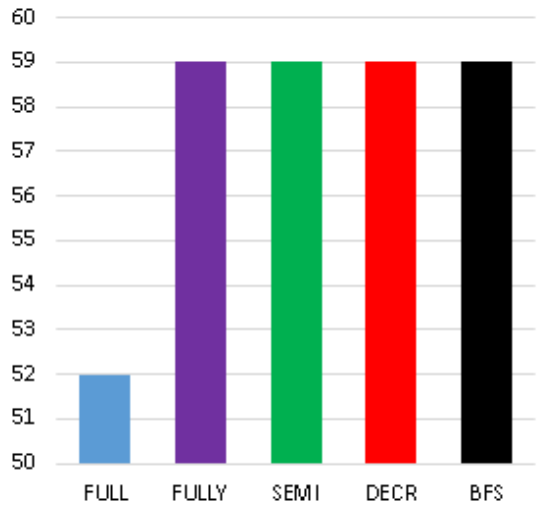
5.3.3. Explicit változó absztrakció

A mérés során felhasznált konfigurációk:

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



(a) 1. konfiguráció



(b) 2. konfiguráció

5.10. ábra. Az egyes keresési stratégiák által sikeresen verifikált modellek száma.

A két konfiguráció között kis eltérés található, ezen kívül a korábbi absztrakcióknál látott eredmények tapasztalhatóak itt is.

Keresési stratégiától független paraméterezés és modell párokat vizsgálva, ahol a FULLY, illetve a SEMI változat is sikeresen képes volt verifikálni:

Konfiguráció	Összesen	FULLY kevesebb csúcsot fejtett ki	Medián kifejtés csökkenés
1.	58	2 (3%)	69%
2.	58	2 (3%)	69%

Ellentétben az eddigi közel 20%-os arányhoz, itt a FULLY változat csak az esetek 3%-ban hozott javulást kifejtett csúcsok számában.

Mivel a 3% 2 esetet jelent, így ezen absztrakt domaint nem lehet tovább elemezni.

6. 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 reprezentációból 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 robusztus, 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 ellenpélda megtalálására, amely csak a minimálisan szükséges lépéseket tartalmazza, így a mérnökök számára megkönnyíti a hibakeresésé folyamatát.

Az elméleti tárgyalást követően a 5. fejezet bemutatta a javasolt algoritmusok teljesítménymérésének eredményeit. A mérési eredmények alapján kijelenthető, hogy a javasolt algoritmuscsalád 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 kidolgoztam egy stabil architektúrát az egyetemen fejlesztett nyílt-forráskódú Theta modellellenőrző keretrendszerbe [1]. Ezen architektúra célja, hogy könnyűvé tegye az újabb Hierarchikus A* változatokkal való kiegészítését.

- Publikus és ipari partnerektől eredő benchmark készleteken vizsgáltam az algoritmusaim hatékonyságát. A mérések során meggyőződtem, hogy a megoldásom verseny-

[1] Fork elérhetősége: `https://github.com/asztrikx/theta/tree/astar`

képes alternatívája a jelenleg elérhető algoritmusoknak és robusztus, jól skálázódó megoldást nyújt a mérnökök számára.

A jövőben fontos feladat lesz megvizsgálni a Theta több ellenpélda alapú finomítóját, mivel a jelenlegi beállítások mellett az csak 1 hamis ellenpélda útvonalat képes eltávolítani a gráfból, ami által a gráfban a többi megtalált célcsúcs megmarad. Az Hierarchikus A* változatok esetében ilyenkor az elkövetkező iterációkban, amíg a megmaradt célcsúcsok nem távolítódnak el, addig a Hierarchikus A* gyorsan megtalálja a hibás csúcsokat, hiszen azok távolsága pontosan ismert. Ezt az előnyt azonban ki kell szűrni, hiszen az ez által jelentett verifikáció gyorsulás nem a Hierarchikus A* keresési stratégiához köthető, így torzítja a mérés eredményeit.

Köszönetnyilvánítás

Szeretnék köszönetet mondani a konzulensemnek, Szekeres Dánielnek a sok útmutatásért a feladat kivitelezése során.

A Kulturális és Innovációs Minisztérium ÚNKP-23-1-I-BME-34 kódszámú Új Nemzeti Kiválóság Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.

A 2019-1.3.1-KK-2019-00004. számú projekt a Kulturális és Innovációs Minisztérium Nemzeti Kutatási Fejlesztési és Innovációs Alapból nyújtott támogatásával, a 2019-1.3.1-KK pályázati program finanszírozásában valósult meg.

Irodalomjegyzék

[1] Vörös Asztrik: Informált keresési stratégiák absztrakció alapú modellellenőrzésben. 2022, Tudományos Diákköri Konferencia, BME. https://tdk.bme.hu/VIK/DownloadPaper/Informalt-keresesi-strategiak-absztrakcio.

[2] Vörös Asztrik: Igényvezérelt heurisztikaszámítás informált kereséssel támogatott absztrakció alapú modellellenőrzésben. 2023, Tudományos Diákköri Konferencia, BME. https://tdk.bme.hu/VIK/DownloadPaper/Informalt-keresesi-strategiak-absztrakcio1.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] Kairong Qian: *Formal symbolic verification using heuristic search and abstraction techniques*. PhD értekezés (University of New South Wales, Sydney, Australia). 2006.
URL http://handle.unsw.edu.au/1959.4/25703.

[18] 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.

[19] Tamás Tóth – Ákos Hajdu – András Vörös – Zoltán Micskei – István Majzik: Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart – Georg Weissenbacher (szerk.): *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design* (konferenciaanyag). 2017, 176–179. p. ISBN 978-0-9835678-7-5.