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

Igényvezérelt heurisztikaszámítás informált kereséssel támogatott absztrakció alapú modellellenőrzésben

TDK dolgozat

Készítette:

Vörös Asztrik

Konzulens:

Szekeres Dániel

2023

Kulturális és Innovációs Minisztérium

Új Nemzeti Kiválóság Program

Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal

Tartalomjegyzék

Kivonat **i**

Abstract **iii**

1. Bevezetés **1**

2. Háttérismeretek **4**
2.1. Formális verifikáció 4
2.2. Vezérlési folyam automata 4
2.3. Állapottér 6
2.4. Állapottér absztrakció 6
2.5. Keresési algoritmusok 7
2.6. Absztrakt elérhetőségi gráf 9
2.7. Ellenpélda-vezérelt absztrakció finomítás 10

3. Hierarchikus A* algoritmus **12**
3.1. A* beépítése a CEGAR hurokba 12
3.2. Megismert távolságok beállítása ARG-ben 14
3.3. Hierarchikus A* keresés teljes ARG kifejtéssel 15
3.4. Hierarchikus A* keresés részlegesen igény szerinti ARG kifejtéssel 15
3.5. Hierarchikus A* keresés heurisztika csökkentéssel 16
3.6. Hierarchikus A* keresés teljesen igény szerinti ARG kifejtéssel 17

4. Architektúra **20**
4.1. CEGAR hurok 20
4.2. Hierarchikus A* Abstractor 20
4.3. Hierarchikus A* specifikus ARG 21
4.4. Távolságot leíró komponens 22
4.5. A* keresést leíró komponens 23
4.6. CEGAR iterációk tárolása 23
4.7. Hierarchikus A* változatok támogatása 24
4.8. Hibakeresés támogatása 25

5. Mérések **27**
5.1. Megvalósítás és mérési környezet 27
5.2. XCFA formális modelleken végzett mérések eredményei 28
5.3. XSTS formális modelleken végzett mérések eredményei 30
5.3.1. EXPL_PRED_COMBINED absztrakció 30
5.3.2. PRED_CART és PRED_SPLIT absztrakció 34
5.3.3. Explicit változó absztrakció 37

6. Összefoglalás és jövőbeli tervek **39**

Köszönetnyilvánítás **41**

Irodalomjegyzék **42**

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.

Korábbi kutatásom során a korábbi iterációkban található információk későbbi iterációkban történő felhasználását dolgoztam ki az úgynevezett Hierarchikus A* algoritmus keretében, mely ezen információkat használja ki, illetve bizonyítottam, hogy a legrövidebb ellenpéldát fogja biztosítani. Az algoritmusnak több változatát is kidolgoztam, melyek az A* heurisztika pontosságában és számítási igényében térnek el. Ezen változatokban azonban még maradtak kiaknázatlan optimalizációs lehetőségek.

Ezen munka során a már elkészült heurisztika számítási módszerek kerülnek kiegészítésre a Teljesen igény szerint történő Hierarchikus A* változat kidolgozásával és annak helyességbizonyításával. A már kidolgozott Igény szerinti változat hátránya, hogy nem veszi figyelembe a többi kifejtésre jelölt, ismert heurisztikával rendelkező csúcsnak a he-

urisztikáját, ami lehetőséget adna arra, hogy a korábbi iterációkban történő kereséseket megszakítsuk, amint annak során bebizonyosodik, hogy a legközelebbi hibás állapot távolságértéke nagyobb lesz egy, a jelenlegi iterációban ismert heurisztikával szemben. A Teljesen igény szerinti változat prototípus változatát implementálom a nyílt forráskódú Theta modellellenőrző-keretrendszerbe, illetve annak hatékonyságát szoftverek és mérnöki modellek verifikálásának benchmarkolásával kiértékelem, összehasonlítva a Thetában megtalálható keresési stratégiákkal, illetve a korábbi Hierarchikus A* változatokkal.

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.

During my earlier research, I have developed the Hierarchical A* algorithm which can use information from earlier iterations in order to support subsequent iterations and proved that the algorithm provides the shortest counterexample. I devised multiple versions of this algorithm, which differ in the precision of the A* heuristic and its computational demand. However there are still optimization possibilities that have not been exploited in my earlier work.

In this work I am going to extend the Hierarchical A* algorithm family introduced in my previous paper with the Fully On-Demand Hierarchical A* version and with a correctness proof. The disadvantage of the already developed On-demand Hierarchical A* version is that it doesn't take into account the other expansion candidate nodes' heuristics which would open the possibility to pause a search when it becomes evident that the distance to the closest target node will be greater than a known heuristic in the current iteration. I implemented a prototype version of the Fully On-demand Hierarchical A* in the open source Theta model-checking framework and then evaluated its effectiveness by benchmarking on software and engineering models against the built-in search strategies in Theta and against the Hierarchical A* versions implemented earlier.

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 [9].

Amennyiben a követelmény sérül, a modellellenőrző algoritmus által nyújtott ellenpélda segítségként tud szolgálni [12] 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 [7, 17].

A CEGAR algoritmusok szakirodalmában találhatunk példát mélységi és szélességi keresés alkalmazására, illetve számos más prioritás alapú keresési stratégiára, melyek a modell előzetes elemzésén alapulnak [9]. 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.

Korábbi munkám során ([1]) egy hatékonyabb és robusztusabb keresési stratégiát dolgoztam 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. Ezzel támogatva a mérnököket nem csak a helyesség bizonyításában, hanem a hatékonyabb hibakeresésben is a fókuszáltabb ellenpéldák nyújtásával.

Ennek eredményeként született meg a Hierarchikus A* alapú algoritmus, ami egy - a CEGAR hurkon belül található - intelligens keresési stratégia. Az algoritmus a jelenlegi absztrakt állapottér A* alapú bejárásához az előző absztrakciókat felhasználva számít egy heurisztikát. Az algoritmusnak több változata is elkészült, amelyek különböző stratégiák mentén dolgoznak. Az algoritmus kidolgozásán kívül az algoritmus helyességét is bizonyítottam.

Jelen munkám során a célom a Részlegesen igény szerinti ARG kifejtéssel történő Hierarchikus A* változat hatékonyságának javítása volt. Ennek során a korábbi ARG-k csúcsait fejtjük tovább, hogy azok távolságértékeiből heurisztikát származtassunk. Azonban ahhoz, hogy a jelenlegi iterációban történő A* keresés során eldöntsük a még fel nem dolgozott csúcsok közül melyiket kell először feldolgoznunk, ahhoz nem szükséges, hogy mindegyik csúcsra pontosan ismert legyen a heurisztika. A Teljesen igény szerinti ARG kifejtéssel történő Hierarchikus A* változat ezért az A* keresés során használt prioritási sorba helyezés előtt nem határozza meg pontosan a heurisztikát. Ehelyett az abból való kivétel során a legkisebb teljes költségű csúcs meghatározásának érdekében, az ahhoz szükséges csúcsok heurisztikáját a szükséges mértékben részben kiszámítja, ezzel egy alsóbecslést meghatározva.

Az új Hierarchikus A* algoritmust a nyílt forráskódú Theta verifikációs keretrendszerben[1] implementálásra kerültek a korábbi munkám során lefejlesztett változatok mellé, így bárki számára elérhetővé tettem a munkám eredményeit[2].

Emellett megvizsgáltam a javasolt új algoritmus gyakorlati alkalmazhatóságát is: a javasolt algoritmus változatnak kiértékeltem a hatékonyságát, és összehasonlítottam az elterjedten használt keresési algoritmusok, illetve a korábbi munkám során kidolgozott Hierarchikus A* változatok hatékonyságával különböző benchmark készleteken.

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 állapottereket a keresés hatékonyabbá tételéhez. Ezután a korábbi és a mostani munkám során kidolgozott Hierarchikus A* változatok kerülnek ismertetésre, illetve azok céljai, előnyeikkel és hátrányaikkal.

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

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

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

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

Kapcsolódó munkák Munkám újdonsága, a korábbi munkám során kifejlesztett absztrakcióval kombinált informált keresési stratégiákat kiegészítettem egy újabb változattal. Ezen a területen javarészt az egyszerűbb keresések (szélességi és mélységi keresés), vagy statikusan heurisztikát számolt keresési algoritmusok elérhetőek el [8, 6, 14]. A korábbi iterációban lévő távolságértéket heurisztikaként való felhasználásáról is készült már munka ([16]).

De a heurisztika számítás történhet akár egy absztraktabb probléma megoldásával [13]. Ezekkel kapcsolatban az elmúlt években született egy nagyon alapos összefoglaló [9], 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. Ezen ismeretek az előző évi TDK-m során kerültek bemutatásra [1].

2.1. Formális verifikáció

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

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

2.2. Vezérlési folyam automata

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

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

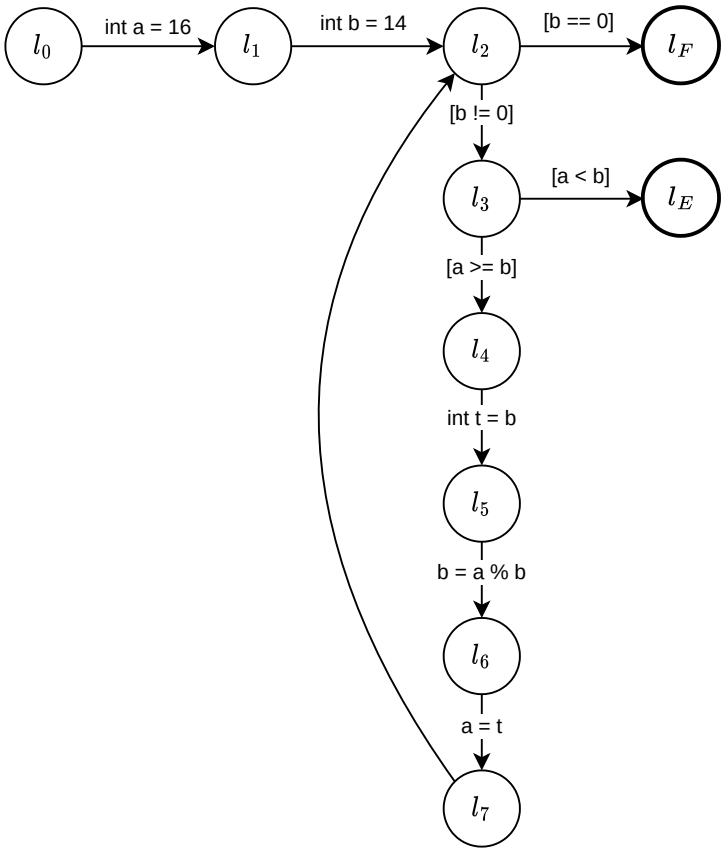
- L: Helyek halmaza, amelyek az egyes elemi programkód utasítások végrehajtás előtti strukturális állapotokat írják le, gráfunkban csúcsokkal reprezentálva. Speciális helyek a program végét és elejét jelző helyek.
- V: Programkódban szereplő változók halmaza, $V = v_1, \ldots, v_N$.

- 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, $Ops = Actions \cup Guards$.
- $Actions$: Olyan elemi utasítások halmaza, melyek lefutása megváltoztatja V egyes elemeinek értékét.
- $Guards$: Olyan elemi utasítások halmaza, melyek V jelenlegi értékei alapján logikai értéket vesznek fel. Az adott $e \in E$ élet csak akkor járhatjuk be, ha V jelenlegi értékei alapján igazra értékelődik ki az adott $guard \in Guards$.
- l_0: Kiinduló hely, mely a program elejét jelzi, innen indulnak a lefutásaink. ▪

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

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

2.1. ábra. C kód



2.2. 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 operációja op-val egyezik meg. ▪

2.4. Állapottér absztrakció

A formális verifikáció során vizsgált programok lehetnek determinisztikusak és nemdeterminisztikusak. Egy egyszerű forrása lehet például a nemdeterminizmusnak a felhasználói bemenet. A nemdeterminisztikus programok formális ellenőrzése könnyen túl sok futásidővel vagy memóriával járhatnak, hiszen a lehetséges lefutások száma exponenciálisan nő nemdeterminisztikus műveletenként. Például c db nemdeterminisztikus 32 bites egész szám esetén $(2^{32})^c$ féle lefutás lehetséges. Ezt a jelenséget állapottér-robbanásnak nevezzük.

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

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

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

Absztrakt állapotok esetén is megkülönböztetjük a hibás állapotokat.

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

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

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

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

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

Az absztrakció egy másik fontos tulajdonsága, hogy a lehetséges lefutásokat konzervatívan felülbecsli, azaz úgy enged meg más nem konkretizálható lefutásokat, hogy az összes konkrét lefutást is megengedi [5].

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

2.5. Keresési algoritmusok

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

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

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

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

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

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

Továbbá vezessük be a következő segédfüggvényeket:

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

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

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

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

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

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

Ezek után már definiálhatjuk a keresési algoritmusokhoz kapcsolódó fogalmakat.

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

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

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

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

A keresési algoritmusokat informáltságuk szerint nem informált és informáltként csoportosíthatjuk.

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

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

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

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

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

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

Tétel 1 (A* optimalitás gráf alapú keresés esetén). [10, 11] Ha a heurisztikánk konzisztens és nincsenek negatív körök a gráfban, akkor az A* keresési algoritmus által visszaadott út a legközelebbi célállapotban végződő út lesz. ▪

2.6. Absztrakt elérhetőségi gráf

Az *absztrakt elérhetőségi gráf (Abstract Reachability Graph, ARG)* [3] az absztrakt állapotterek (absztrakt) lefutásait gráfként reprezentáló tömör adatstruktúra, RT-hez hasonló módon.

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

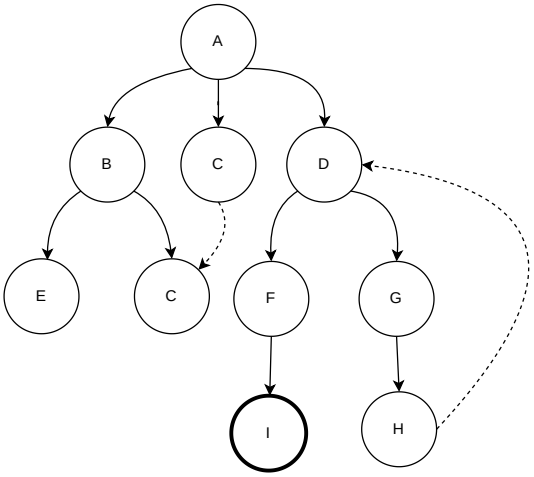
- N: Gráfban lévő csúcsok halmaza. Speciális eleme a gyökér csúcs, melynek állapota a kezdőállapot.
- $L : N \to S_{absztrakt}$: Az adott csúcshoz tartozó absztrakt állapotot megadó címkézés.
- $E \subseteq N \times Ops \times \{1\} \times N$: Gráfban lévő irányított élek halamaza.

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

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

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

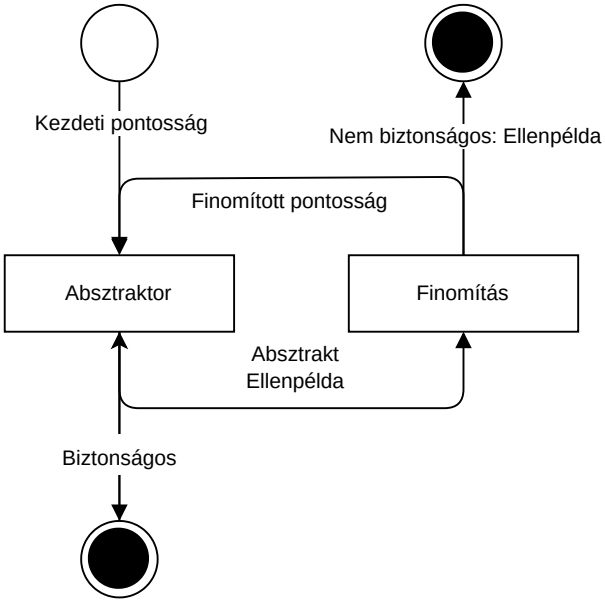
állapota, ha az konzervatívan felülbecsli az elérhető lefutásokat, azaz K kiinduló csúcs és V végcsúcs esetén $L(K) \preceq L(V)$. Mivel csak optimalizációs funkciót látnak el, amiatt a legrövidebb ellenpélda hosszába nem fognak beleszámítani, ezért a fedőéleket 0 súlyozással látjuk el.



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

2.7. Ellenpélda-vezérelt absztrakció finomítás

Az absztrakció tulajdonságaira építve az *ellenpélda-vezérelt absztrakció finomítást (Counterexample-guided abstraction refinement, CEGAR)* [5] egy absztrakció alapú formális verifikációs algoritmust valósít meg. Az algoritmus felhasználja a már megismert ARG-t, illetve egy kereső algoritmust annak bejárása érdekében. Szintén szükségünk van egy állapottér absztrakcióra, ami pontosan meghatározza annak pontosságának számítását.



2.4. ábra. A CEGAR hurok

Az algoritmus elején kiindulunk egy kezdeti pontosságból.

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

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

3. fejezet

Hierarchikus A* algoritmus

Ebben a fejezetben a kontextus teljessége érdekében bemutatom a korábbi munkám során már kifejlesztett és bemutatott Hierarchikus A* változatokat, majd az ezen munkám során kifejlesztett Teljesen igény szerinti Hierarchikus A* változatot.

Az egyes tételekhez tartozó bizonyítások megtalálhatók a korábbi munkámban [1], ahol a nem Teljesen igény szerinti változatokhoz tételek és definíciók már bemutatásra kerültek.

3.1. A* beépítése a CEGAR hurokba

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

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

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

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

Alapértelmezetten azonban struktúratartó providert fogunk használni, amelynek okát láthatjuk majd a 3.1. tételben.

Definíció 18 (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 szülő 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 $\forall n_B \in N_B : L_A(provider(n_B)) \succeq L_B(n_B) \wedge \exists e_A \in e_A(provider(p_B)) : c(e_A) =$

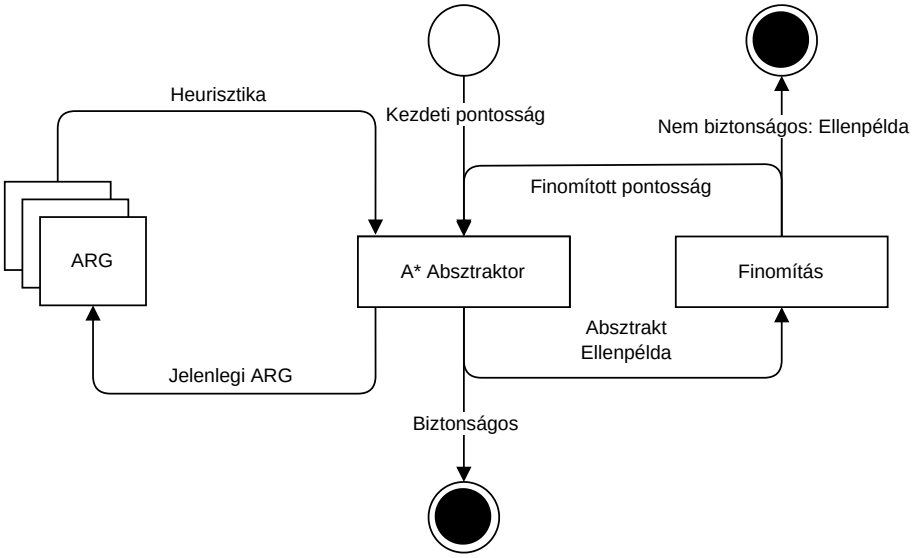
$provider(n_B)$. Gyökér csúcs esetén a provider A gyökere lesz. Struktúratartó provider esetén biztosítanunk kell, hogy $provider(p_B)$ rendelkezésre álljon, illetve hogy ki legyen fejtve. ▪

A struktúratartó providernek az előnye az, hogy az általa elérhetővé tett A-beli távolságfüggvény értékek konzisztens heurisztikát alkotnak, ha a fedőélek behúzását csak akkor engedjük meg, ha az nem sérti a konzisztens heurisztika által megkövetelt feltételeket. Ezzel a módosítással kimondhatjuk és bizonyíthatjuk a következő tételt.

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

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 a későbbi iteráció számára, hogy az 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 konzisztens heurisztikaként felhasználjuk. Az ARG-ket az adott iterációban végrehajtott keresés végeztével tároljuk el az absztraktorban. A már látott CEGAR hurkot leíró ábra (2.4) a következőre módosul:



3.1. ábra. A* beépítve a CEGAR hurokba

A következő szekciókban az ARG kifejtésének mértékében különböző Hierarchikus A* változatokat kerülnek bemutatásra és megvizsgálásra, melyek a Teljesen igény szerinti változat kivételével a korábbi munkám során már bemutatásra kerültek:

- Hierarchikus A* keresés teljes ARG kifejtéssel
- Hierarchikus A* keresés részlegesen igény szerinti ARG kifejtéssel
- Hierarchikus A* keresés heurisztika csökkentéssel
- Hierarchikus A* keresés teljesen igény szerinti ARG kifejtéssel

3.2. Megismert távolságok beállítása ARG-ben

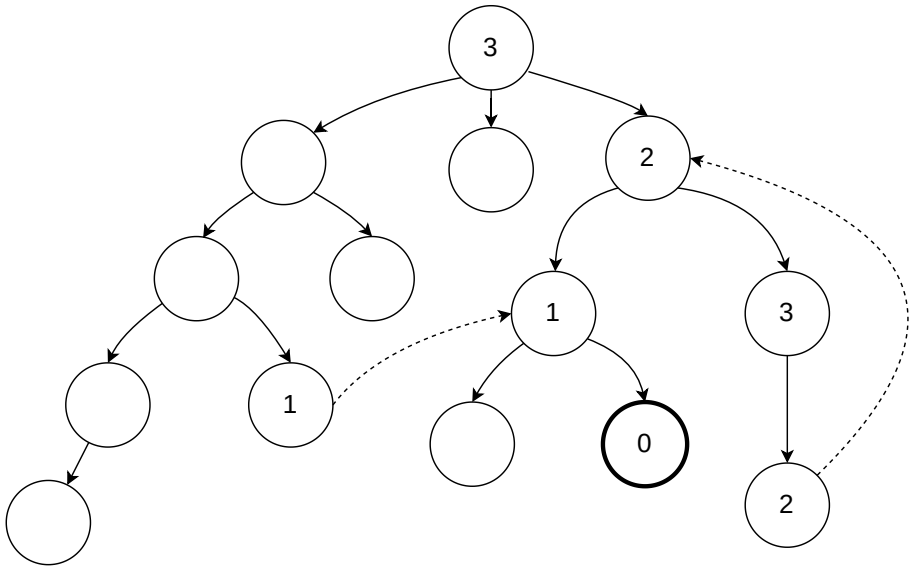
Az ARG-k esetében figyelembe kell venni, hogy a fedőélek olyan csúcs között keletkezhetnek, ahol az egyik csúcs absztraktabb egy másiknál, illetve céljuk hogy elkerüljék a konkrétabb csúcsból történő részgráf kifejtést. Ilyenkor a részgráf absztraktabb verziója a fedő csúcs részgráfja lesz, hiszen az 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 célállapotot, akkor csak az oda vezető útnak a távolsággal még nem rendelkező csúcsaira ismerjük meg a legrövidebb távolságot. Azonban ha egy csúcs egyedüli gyereke egy csúcs, mely ennek az útnak a része, akkor ezen csúcsnak is kizárásos alapon beállíthatjuk a távolságértékét.

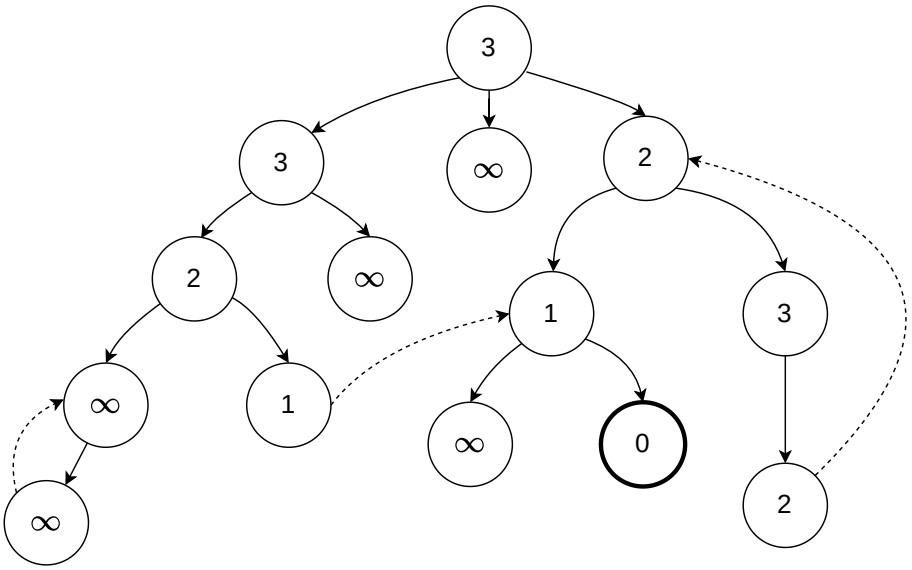
Az előbb leírt folyamatot meg tudjuk általánosan is fogalmazni azon csúcsokra, mely nem elemei az legrövidebb útnak. Minden olyan esetben, ha egy csúcs gyerekének megtudjuk a távolságfüggvény értékét és minden más gyerekének már ismert a távolságfüggvény értéke, akkor a csúcsnak is kiszámíthatjuk a távolságfüggvény értékét: $d(n) = \min_{e \in e(n)} d(c(e)) + 1$.

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 ennek a keresésnek a végeztével még nem kerül meghatározásra.



(a) A vastag körrel jelölt célcsúcstól az oda vezető legrövidebb úton, illetve a korábban vett képlet segítségével további csúcsokon a távolság beállítása.



(b) A távolság nélküli levelek és a végtelen heurisztikájú csúcsok végtelenre állítása, majd a képlet alkalmazásával további csúcsok távolságának beállítása.

3.2. ábra. A már látott ARG-n a távolságok beállítása a fentebb megismert módszer segítségével.

3.3. Hierarchikus A* keresés teljes ARG kifejtéssel

Az első módszer arra, hogy biztosítsuk egy ARG esetén a következő iterációban lévő ARG által igényelt távolságfüggvény értékek rendelkezésére állását, hogy 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 tárgyaltak alapján nem tudunk 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 célcsúcsot már ki van 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 még az absztrakt állapottér mérete is végtelen lehet a használt absztrakciótól függően, ezért ilyen esetekben nem tudjuk használni ezt az algoritmust.

3.4. Hierarchikus A* keresés 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 csak akkor kerülnek kiszámításra, amikor azokra a következő iterációban történő A* keresés során kifejtésre kerülnek.

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

Mivel n csúcs p szülőjének már ismert kell legyen a heurisztikája, hiszen a már kifejtett csúcsok részt vettek egy A* keresésben, ezért a célcsúcs kivételével a csúcs biztosan ki van fejtve, tehát $provider(n)$ is biztosan létezik. Tehát célcsúcs esetében a $provider(n)$ meghatározásához a csúcsot ki kell fejtenünk.

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 $provider(n_2)$, 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. Ebben az esetben nem kell újra bejárnunk a hozzá tartozó részgráfot, hanem elég lekorlátozni a keresés mélységét $k \coloneqq g(n) + d(n_2)$-re hiszen n_2-ből $d(n_2)$ alatt elérhető egy célcsúcs, míg n_2-ig $g(n)$ mélységgel juthatunk el. Ezen k korlátot a csúcsok $f(n)$ értékeivel szemben állítjuk, hiszen azok egy alsó becslést adnak a hozzájuk legközelebbi célcsúcstól való távolságra. Ha nem találunk ezen k korláton belül más célcsúcsot, akkor n csúcsból kiindulva a 3.2. szekcióban megismert módszerrel beállíthatjuk az egyes csúcsok távolságait, ahol az ismert távolságú csúcs a célcsúcs szerepét veszi fel.

Előnye ennek a módszernek, hogy elkerüljük az ARG teljesen kifejtését, ami akár végtelen is lehet. Azonban mivel egy korábbi iterációban lévő ARG-ben a különböző csúcsok távolságszámításához mindig egy új A* keresést indítunk a kérdéses csúcstól, ezért az egymást követő keresések során többször is meglátogathatjuk ugyanazt a csúcsot, ha a korábbi meglátogatás során történt keresés nem tudom beállítani számára 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 heurisztika csökkentéssel

A korábbi módszerek a keresés támogatása érdekében további kifejtéseket hajtottak végre a korábbi iterációjú ARG-ken. A csökkentéssel történű heurisztika számítás ezt 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, akkor n szülőjének heurisz-

tikájának eggyel csökkentet változatát használjuk, kivéve ha az negatív érték lenne, hiszen a 0 egy mindig igaz alsó becslés, mely pontosabb a negatív értékeknél. A csökkentésnek az oka, hogy biztosítsuk a heurisztika konzisztencia tulajdonságát

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

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

2. 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
return max(h(p_n) − 1, 0)
```

A módszer előnye, hogy a heurisztikát konstans időben és könnyen elő tudjuk állítani, mivel nem foglalkozunk a korábbi ARG-k továbbfejtésével, azokban nem végzünk további keresést. Azonban, ha korábbi ARG-ben kevés csúcsra ismerjük a távolságfüggvénybeli értékét, akkor sok helyen kell alkalmaznunk ezt a módszert, ami pontatlan becslést adhat, illetve kellő számú csökkentés után 0 lesz az értéke, ami a BFS kereséssel egyezik meg.

3.6. Hierarchikus A* keresés teljesen igény szerinti ARG kifejtéssel

Ezen munkám kontribúciója a Teljesen igény szerinti Hierarchikus A* változat, mely 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 19 (Pontos heurisztika).
Adott n csúcs esetén $d(provider(n))$ értéke. ▪

Definíció 20 (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 talált 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 az ehhez

szükséges csúcsok heurisztikáját 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 lb értéket nem adja. A nem pontos heurisztikával rendelkező csúcsok prioritási sorban való rendezésekor az $f(n)$ értékhez a nem pontos heurisztikát használja fel, 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 becslés: 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, az 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 elér 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 pontosnál 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 csúcsot a pontos becslés értékéig pontosítjuk, míg a másodikat pontosítása során találunk előbb egy célcsúcsot, akkor a fordított sorrendben végrehajtott pontosítás során az első csúcs pontosítását kihagyhattuk 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. Ezzel vagy a jelenlegi esetet kapjuk vissza, vagy megkapjuk a következő esetet.
   - Ha a két elem egyező alsóbecslésű, akkor az elsőt eggyel pontosítjuk. Ezzel vagy újra az 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 legkisebb pontos becslés értékét el nem éri az azt megelőző összes nem pontos becsléses elem, vagy ha ez egyik elem pontos lesz, mindkét esetben megkapva az első esetet. Utóbbi eset fog akkor is bekövetkezni, ha a prioritási sorban nincs pontos becsléses elem.

   Az első pontos becslés 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.

3. 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
return $h(provider(n))$ ▷ Kezdeti alsó becslésnek felhasználható a $provider(n)$ heurisztikája

4. 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 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() ▷ Jobb rekurzív hívás
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()
end if

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.

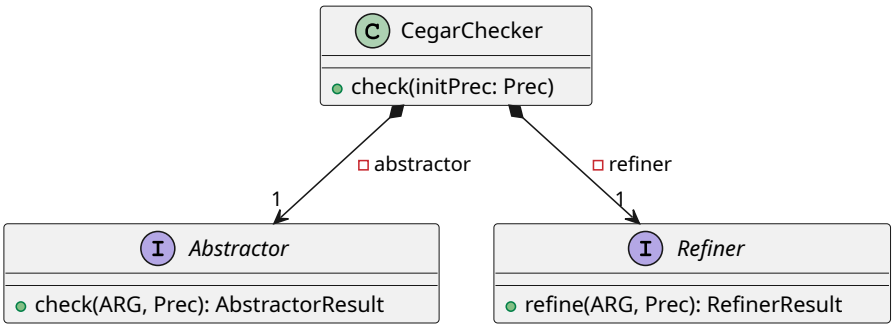
4. fejezet

Architektúra

A korábbi munkám során készített prototípus megvalósításom a Theta modellellenőrző keretrendszerben készült, az azon végzett mérések megmutatták a Hierarchikus A* algoritmuscsalád kompetetivitását, emiatt kidolgoztam egy stabil architektúrát az algoritmuscsalád Thetába integrálásához, melyre építve könnyen lehet új változatokat bevezetni. Ezen architektúrára építettem az új Teljesen igényszerinti változatot is. Ebben a fejezetben az implementáció során kialakított architektúrát fogom bemutatni.

4.1. CEGAR hurok

A CEGAR hurok magas szintű működését (2.4. ábra) megvalósító Thetabeli osztály a *CegarChecker*. Ez tartalmaz egy *Refiner* implementációt, mely az absztrakció pontosságának (*Prec*) finomításáért és az ARG visszavágásáért felel. Az absztrakt állapotteret leíró gráf (ARG) kifejtését pedig egy *Abstractor* implementáció végzi el.

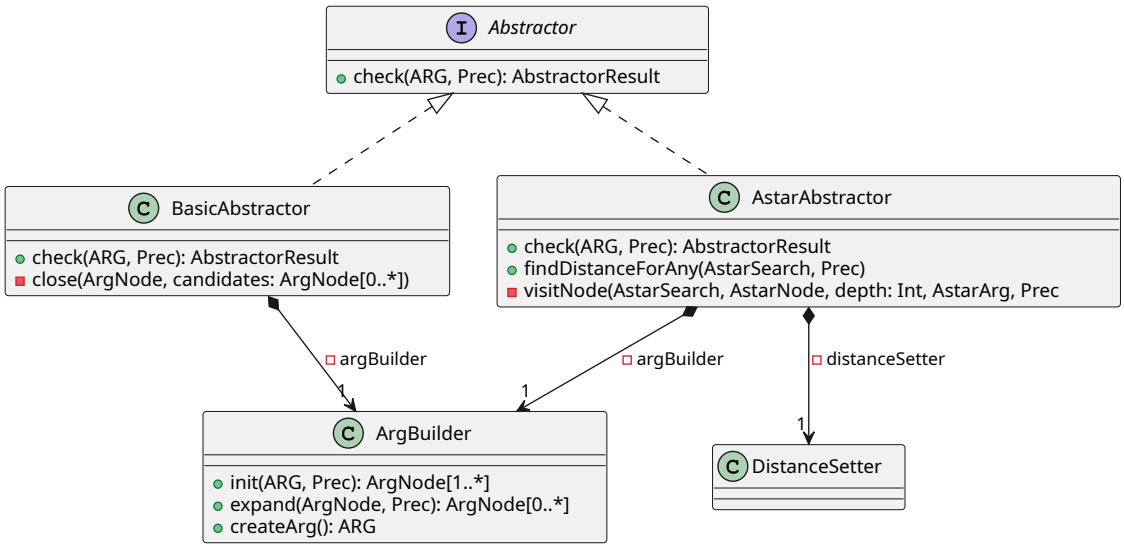


4.1. ábra

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.



4.2. ábra

Az *Abstractor* interface által megkövetelt *check* függvény a gyökér csúcsra meghívja a *findDistanceForAny* metódust, mely azért felelős, hogy kifejtse a gráfot. A gráf kifejtés leállítását a keresést leíró *AstarSearch* osztályra, míg a keresés végeztével a távolságok beállítását a *DistanceSetter*-re bízza. Lefutásának végeztével a gyökér csúcs távolságértékét vizsgálva megállapítható, hogy a gyökér csúcs elért-e hibás csúcsot. A csúcsok heurisztikájára az *AstarSearch*-nek van szüksége, így annak kiszámítása ott történik.

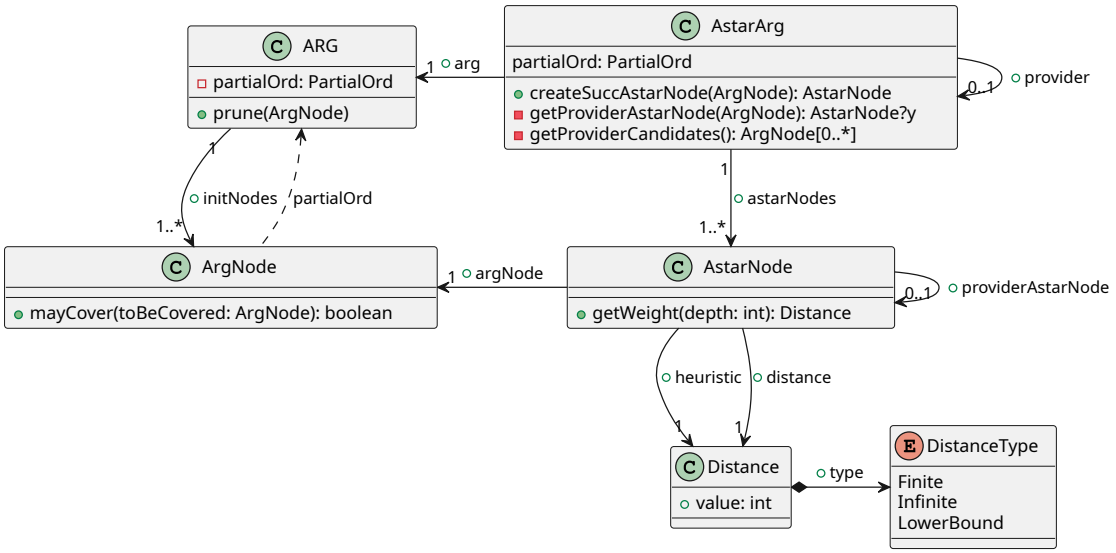
Az *ArgBuilder* komponens feladata az ARG csúcsainak létrehozása, így például kifejtéskor a benne található *expand* metódus kerül felhasználásra.

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, mely nincs hatással a keretrendszer többi részére. Ennek hátránya, hogy a tartalmazott osztályhoz a hozzáférés közvetett.



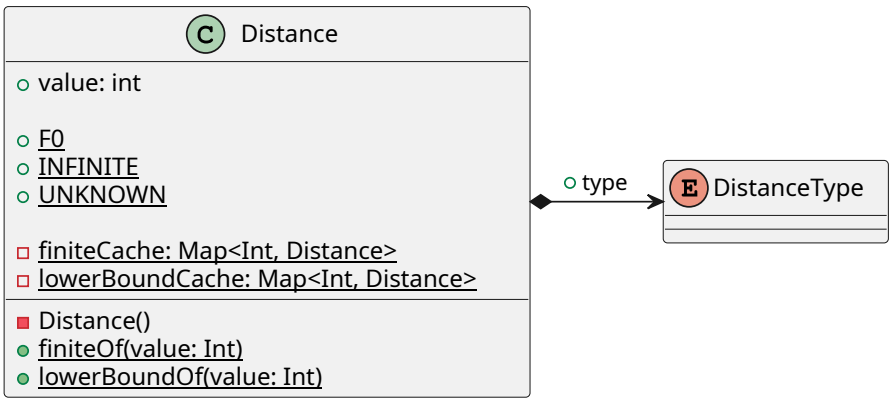
4.3. ábra. Adapter minta alkalmazása

Az *AstarArg* osztály *getProviderAstarNode* metódusa állapítja meg a hozzá tartozó korábbi iterációban lévő provider csúcsot, a 18. definícióban leírtak szerint, felhasználva a részbenrendezést (*partialOrd*), amely képes eldönteni, hogy egy adott absztrakt konkrét állapotainak részhalmazát írja-e le egy másik absztrakt állapot.

Az *AstarArg* egyes csúcsait reprezentáló *AstarNode* *getWeight* függvénye az A* keresés során a csúcsok sorrendezéséhez van használva (2.5. képlet). Fontos kiemelni, hogy ehhez szükség van egy mélység értékre, mely keresésenként eltérő lehet ugyanarra a csúcsra.

4.4. Távolságot leíró komponens

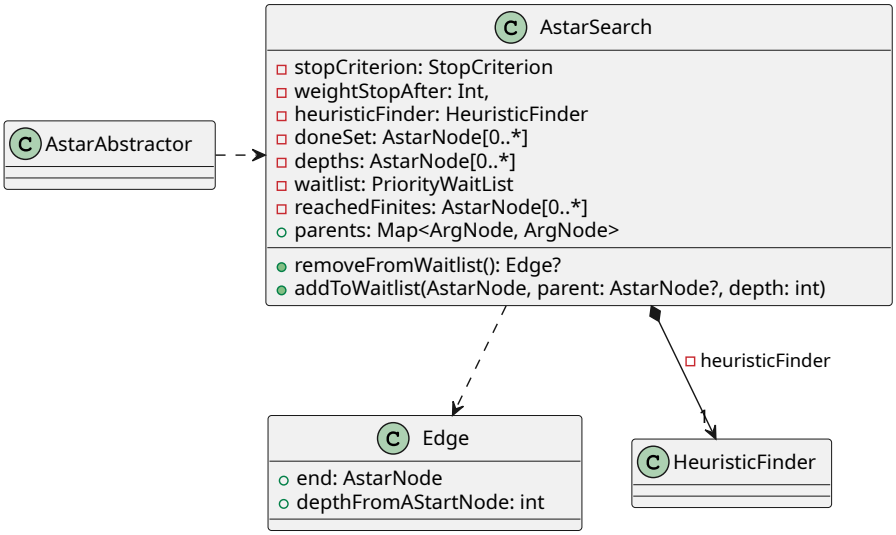
Mivel az ugyanazt reprezentáló távolság (*Distance*) példányok többször is felhasználásra kerülhetnek különböző csúcsoknál, ezért memóriahasználat szempontjából előnyös a gyakran használt értékeket csak egyszer létrehozni. Ennek érdekében az egyféle értékkel rendelkező típusokat statikus tagként, míg a végtelen lehetséges értékkel rendelkező *Finite* és *LowerBound* típusúakat statikus factory mintával 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 a példányt.



4.5. ábra

Amikor elérünk egy új csúcsot és azt berakjuk a prioritási sorba a $addToWaitlist$ segítségével, akkor meghatározásra kerül annak a csúcsnak a heurisztikája a keresési stratégiának megfelelően. 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 az algoritmus.

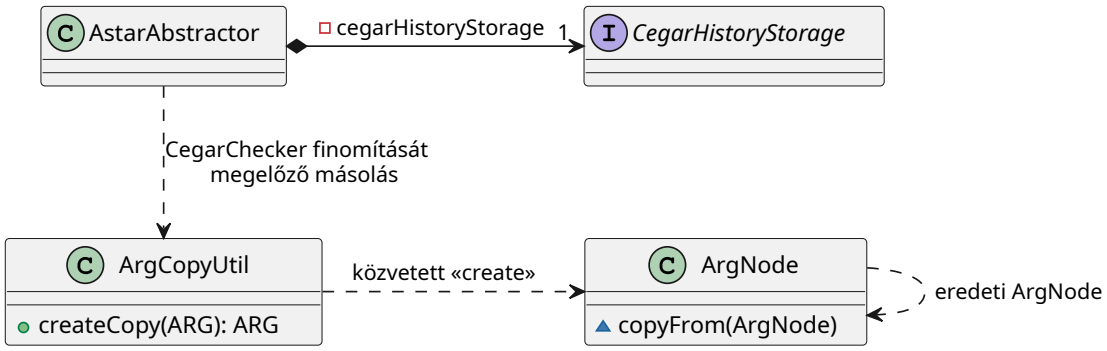
Korábbi iterációban indított keresés esetén elérhetünk már ismert, korlátos távolságértékű csúcsokhoz. Ilyenkor, hogy ezekhez a csúcsokhoz tartozó részgráfokat ne járjuk be újra, a csúcsot berakjuk a prioritási sorba, azonban itt a csúcsnak az ismert távolságát használjuk fel heurisztikának. Majd amikor az soron következőként kikerül a sorból, akkor ahogy egy célcsúcsot, ezt is eltároljuk a $reachedFinites$-ban.

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

A komponens többi adatváltozója megfeleltethető az A* keresés során használt adatváltozóknak.

4.6. CEGAR iterációk tárolása

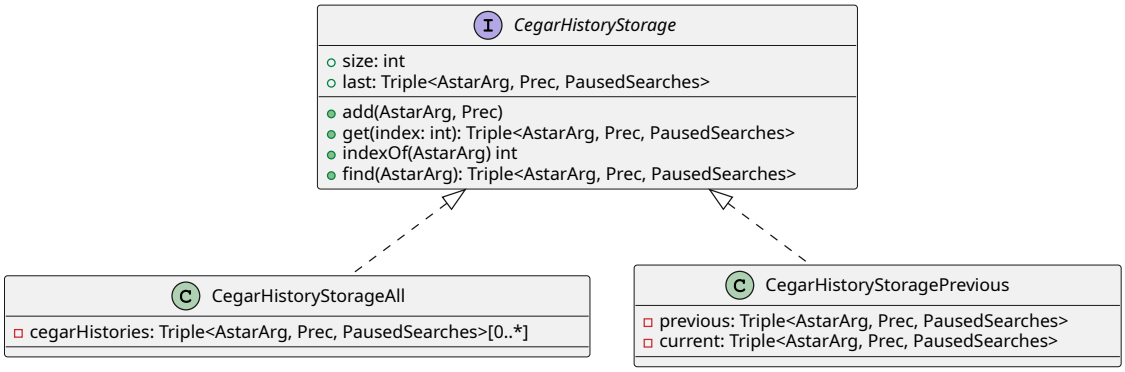
Mivel a CEGAR iterációk végén elveszik az akkori ARG, illetve absztrakciós pontosság ($Prec$), ezért szükséges előtte egy másolatot készíteni róluk és eltárolni őket, hogy tudjunk a korábbi iterációban megtalálható távolságértékeket heurisztikaként felhasználni. Ehhez egy, a $CegarHistoryStorage$-nak megfelelő implementációt használhatunk. Ez szintén tárolja a megszakított kereséseket is, mely a Teljesen igény szerinti változat használ fel.



4.6. ábra

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 a kifejtettség állapotot a Theta keretrendszerben csak az *ArgBuilder* tudja kezelni.

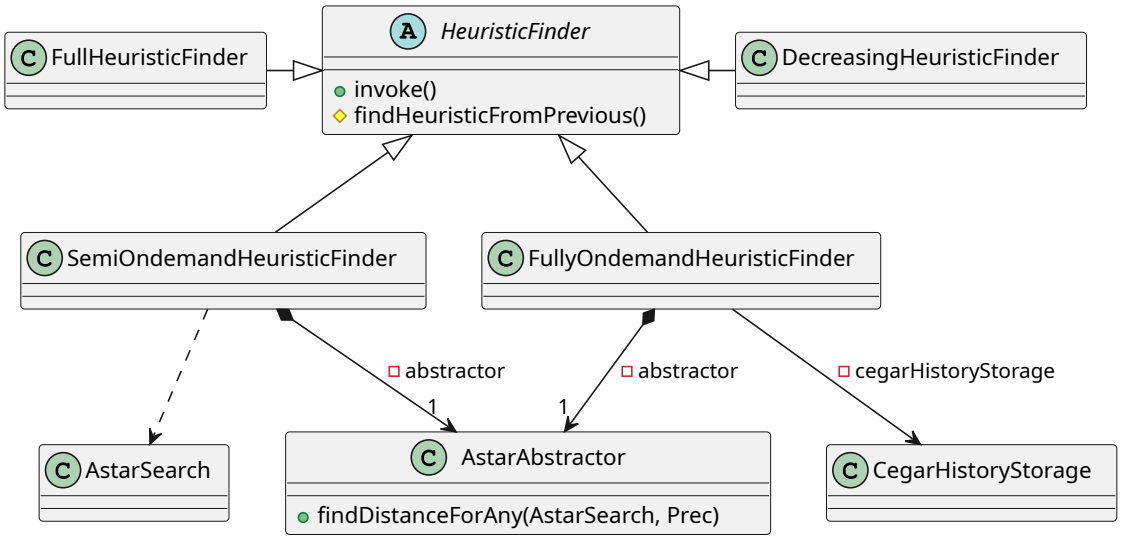
Mivel a Teljes ARG kifejtéses, illetve a Csökkentéses Hierarchikus A* változat esetén csak a jelenlegit megelőző iterációt használjuk fel, ezért a *CegarHistoryStorage*-ból készült egy olyan implementáció is, mely a többi iterációt nem tárolja el. Ez az iterációnként egyre nagyobb méretű *ARG*-k miatt egy fontos tárhely igény optimalizáció tud lenni.



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 mintát használtam a korábban is látott *HeuristicFinder*-nél.

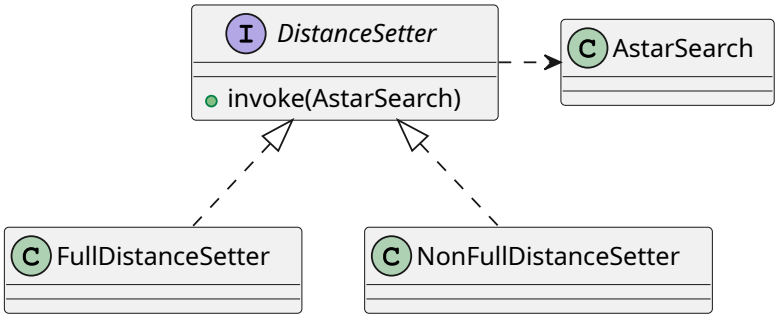


4.8. ábra

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 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.

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, vagy annak megszakítása után.



4.9. ábra

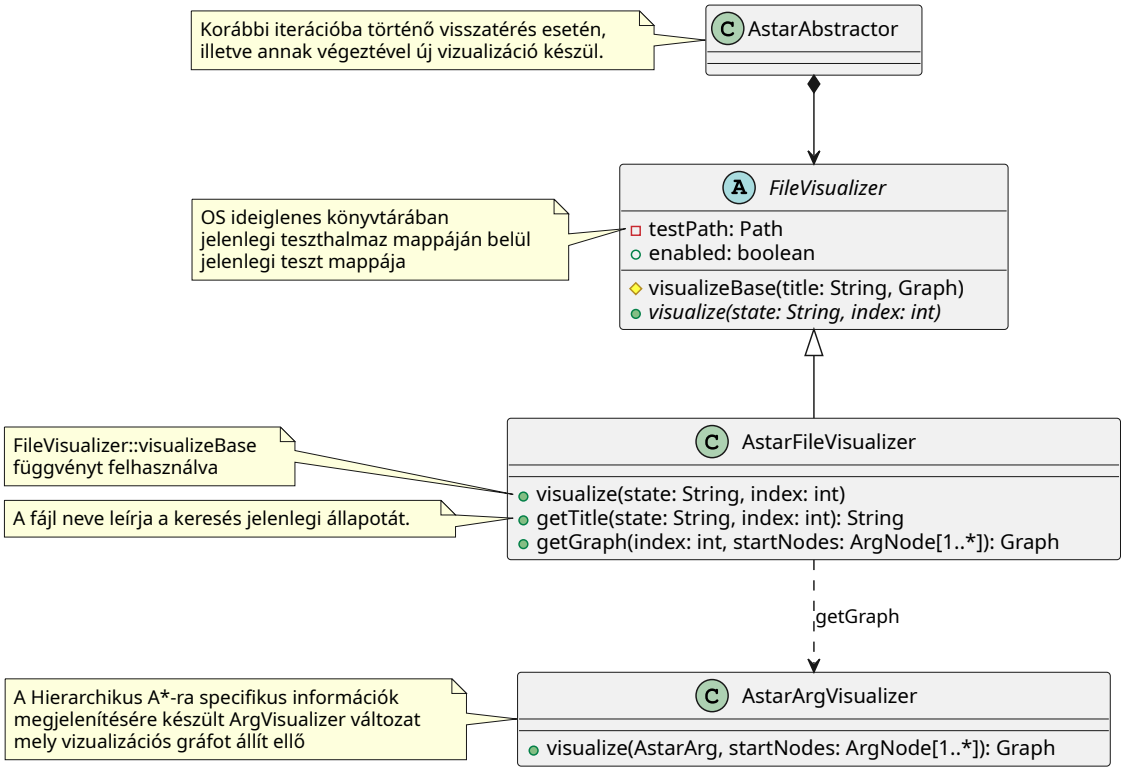
4.8. Hibakeresés támogatása

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* [1] 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.

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



4.10. ábra

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

5. fejezet

Mérések

Ez a fejezet a dolgozatban bemutatott Teljesen igény szerinti Hierarchikus A* teljesítményének kiértékelését fogja bemutatni és az azok eredményeiből levont következtetéseket tárgyalja.

5.1. Megvalósítás és mérési környezet

A Teljesen igény szerinti Hierarchikus A* algoritmust a nyílt forráskódú moduláris Theta[1] modellellenőrző keretrendszerbe fejlesztettem, mely számos moduláris elemet tartalmaz, mint például az absztrakciók, a finomítási eljárások vagy a keresési stratégiák. A Theta forkolt változata tartalmazza a korábbi munkám során lefejlesztett Hierarchikus A* változatokat.

A mérésekhez használt bemeneteket a Software Verification Competition [2] (SV-Comp) benchmark részhalmaza, illetve az FTSRG kutatócsoport által biztosított, többek között ipari partnerektől származó belső XSTS modellek [15] alkotják. A mérések végrehajtásához a BenchExec [4] került felhasználásra, mely segít a felhasznált erőforrások megbízható mérésében.

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

A Teljesen igény szerinti Hierarchikus A* változat összehasonlításra kerültek a Thetába már beépített BFS és ERR keresési algoritmusokkal és a korábbi Hierarchikus A* változatokkal. Az ERR keresés CFA struktúra alapján számít heurisztikát, a jelenlegi hely és a legközelebbi hibás hely távolsága alapján, nem véve figyelembe az elágazási feltételeket. A BFS biztosítani tudja számunkra a legrövidebb ellenpéldát a Hierarchikus A*-hoz hasonlóan. Az ERR keresés a Hierarchikus A*-hoz hasonlóan heurisztikát használ, azonban azzal ellentétben nem finomítja azt 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

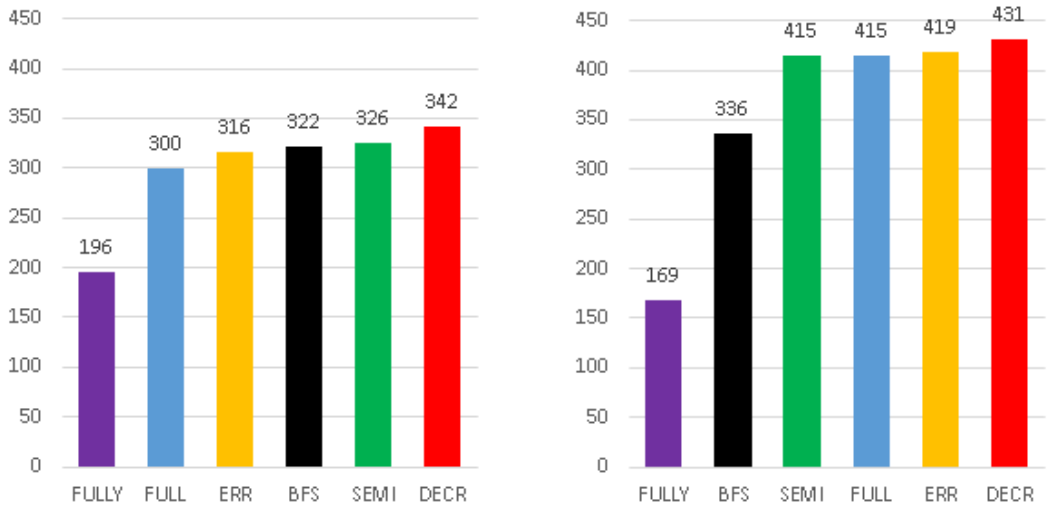
5.2. XCFA formális modelleken végzett mérések eredményei

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.

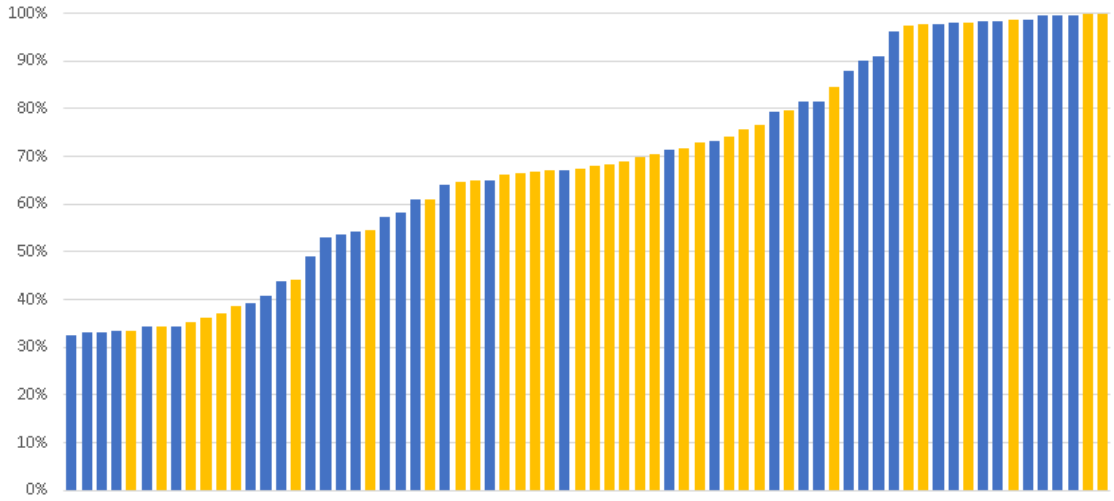
Látható, hogy a Teljesen igény szerinti változat nagy mértékben rosszabbul teljesít a beépített, illetve a korábbi Hierarchikus A* változatokkal szemben mindkét konfigurációban. Ebből arra a következtetésre juthatunk, hogy a keresések gyakori megszakítása nagyobb teljesítményromlást okoz, szemben az 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 Teljesen-, illetve a Részlegesen igény szerinti keresés is sikeresen tudott verifikálni:

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

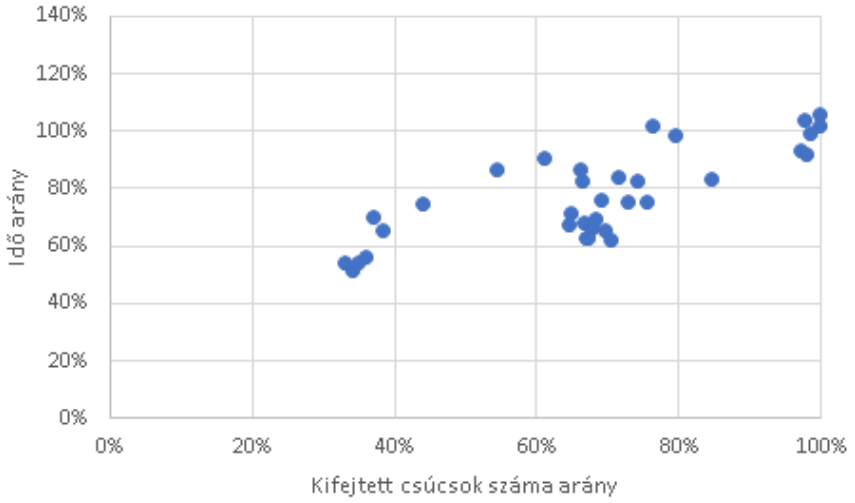
Látható, hogy kevés esetben tudott valójában csökkenést okozni a kifejtések számában a Teljesen igény szerinti változat, mely szintén hozzájárul a 5.1.-ben látott teljesítményromláshoz.

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

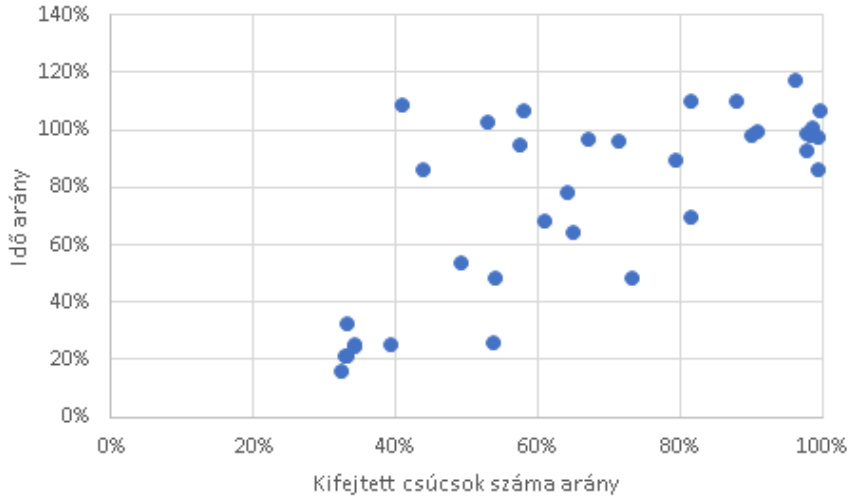


5.2. ábra. Sárga színnel az 1-es, míg kék színnel a 2-es számú konfigurációk esetében történt csúcs kifejtés csökkenés a Teljesen igény szerinti változatban a Részlegesen igény szerintihez képest

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 Részlegesen igény szerintivel 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.

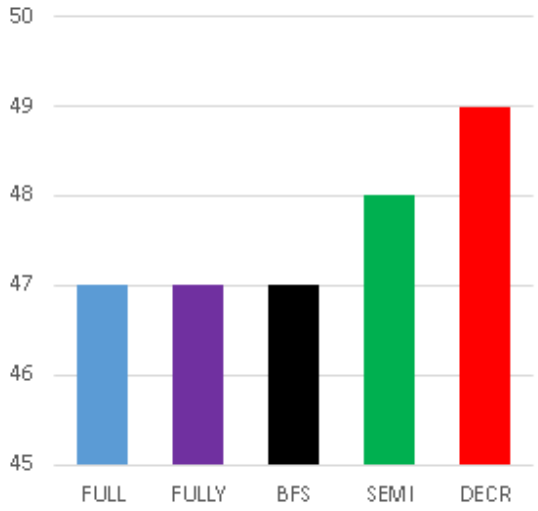
Az XCFA mérésekhez képest nagyobb számban mért konfigurációk miatt az eredményeket absztrakt domainenként elkülönítve mutatják be a következő részek. Az explicit (EXPL) és kombinált explicit-predikátum (EXPL_PRED_COMBINED) domainek esetében új konfigurációs paraméter a kifejtett gyerekek maximális száma, 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 Csökkentéses változat eggyel több modellt volt képes verifikálni, így ez kerül bemutatásra:



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

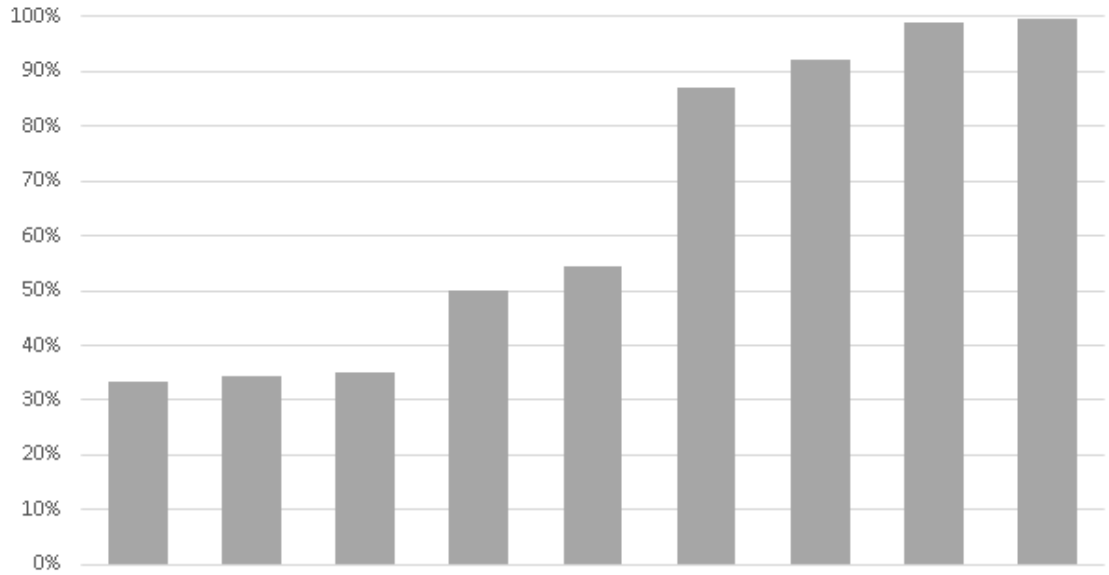
Ezen mérés során az egyes keresések között nem tapasztalható nagyobb eltérés, bár a Teljesen igény szerinti itt is kevesebb modell verifikációjára volt képest az időkorlát mellett.

Keresési stratégiától független paraméterezés és modell párokat vizsgálva, ahol a Teljesen-, illetve a Részlegesen igény szerinti 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%

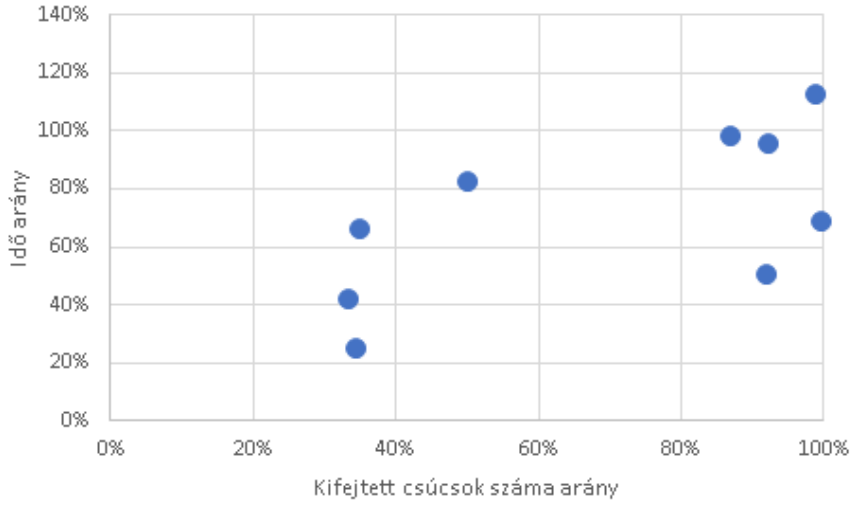
Hasonlóan az XCFA-hoz itt is kevés esetben volt képest kifejtés csökkenést elérni a Teljesen igény szerinti változat.

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.

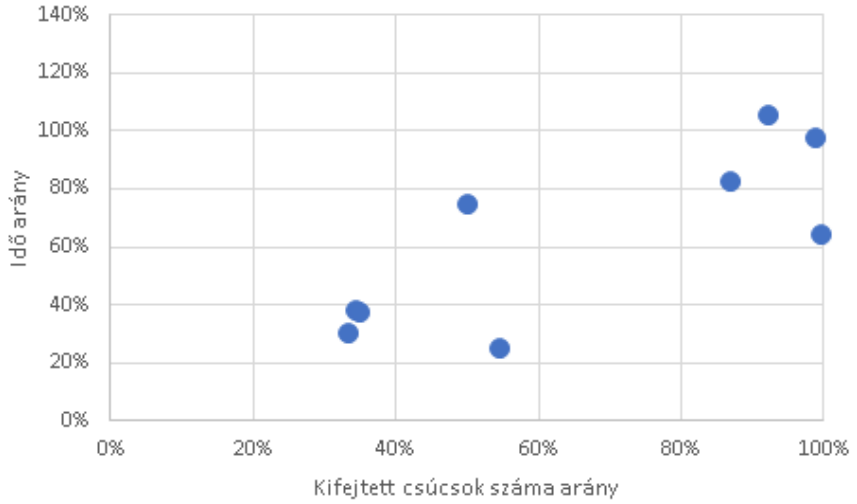


5.5. ábra. Csúcs kifejtés csökkenés a Teljesen igény szerinti változatban a Részlegesen igény szerintihez képest

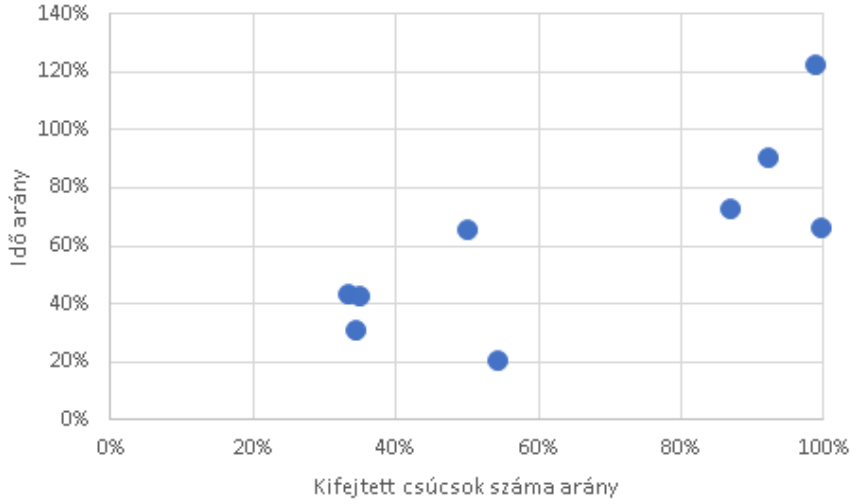
Azonban a futásidő arányoknál nem voltak azonosak az értékek, így a kifejtett csúcsok számának arányához társított verifikációval töltött idő arányát bemutató scatter plot ábra a 4 konfiguráción külön kerül bemutatásra:



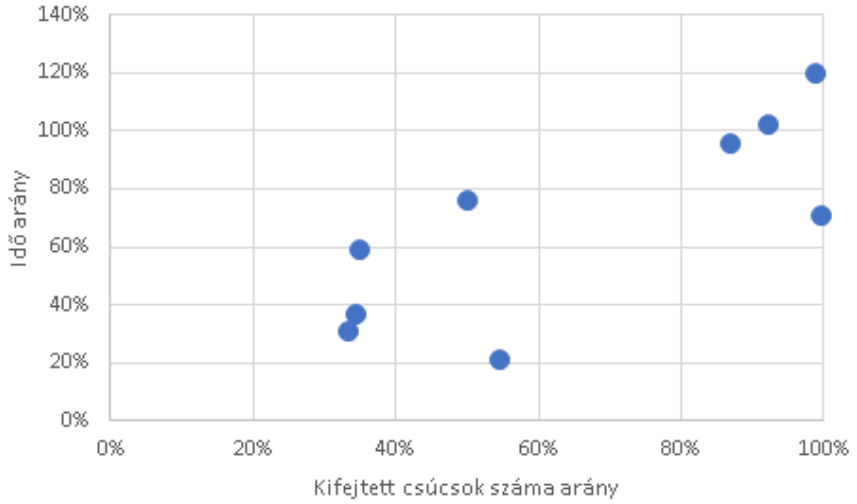
(a) 1. konfiguráció



(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

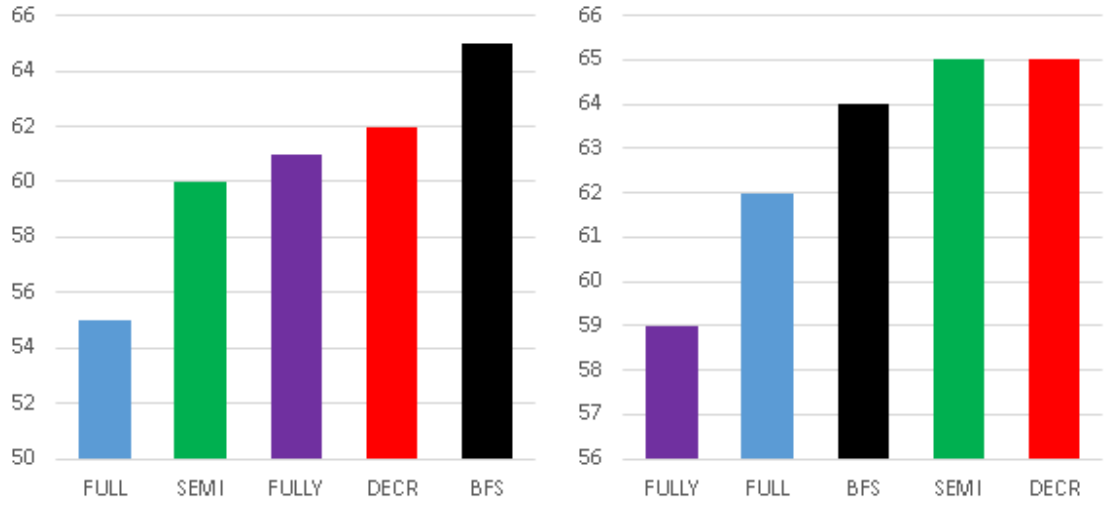
Az 1. konfiguráció esetében nem mutatható ki összefüggés a kifejtett csúcsok számának aránya és a futásidő aránya között.

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 Teljesen igény szerint 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.

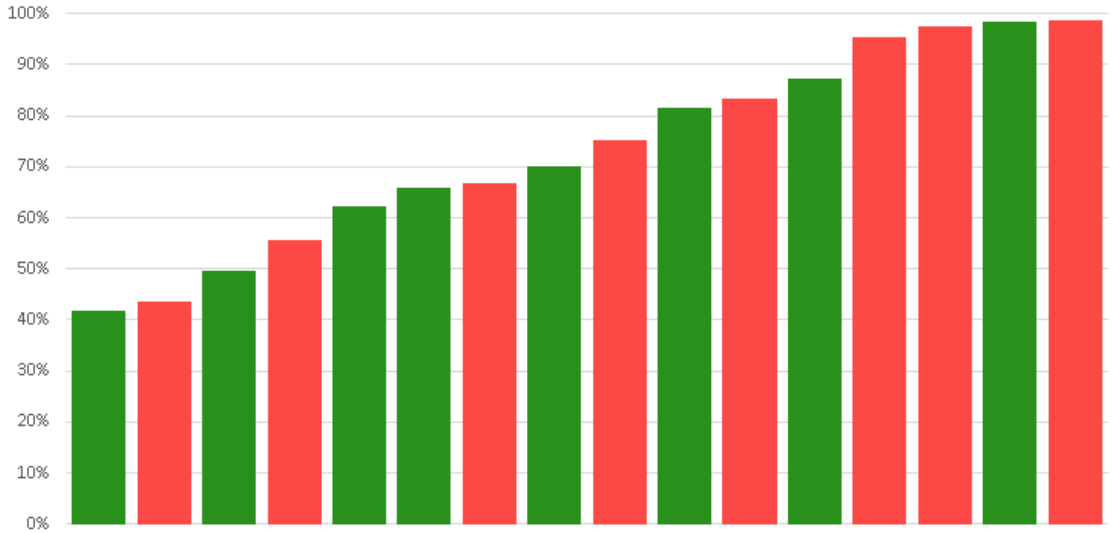
Bár a 2. konfigurációban az eddig megszokott eredmény tapasztalható, azonban az 1. konfiguráció esetében a Teljesen igény szerinti képes volt felvenni a versenyt a Részlegesen igény szerint változattal.

Keresési stratégiától független paraméterezés és modell párokat vizsgálva, ahol a Teljesen-, illetve a Részlegesen igény szerinti 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%

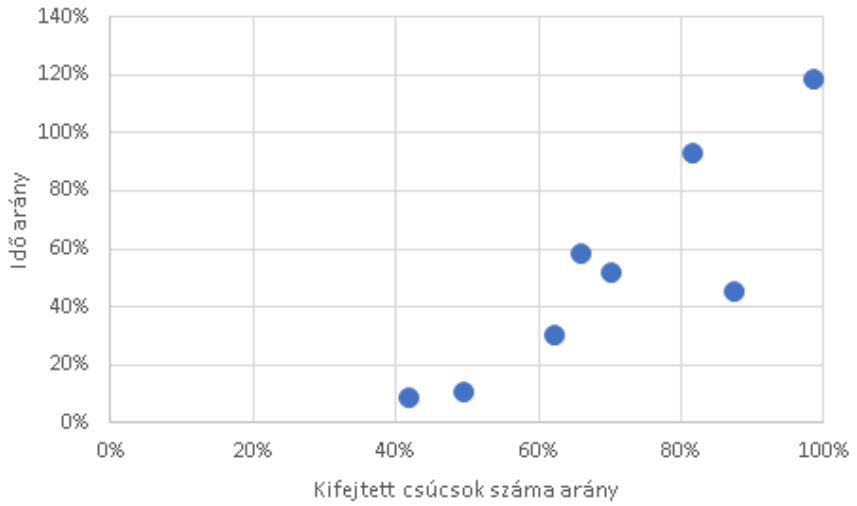
Hasonlóan az XCFA-hoz itt is kevés esetben volt képest kifejtés csökkenést elérni a Teljesen igény szerinti változat.

A kifejtés csökkenés mértékét részletesebben a következő diagram mutatja be.

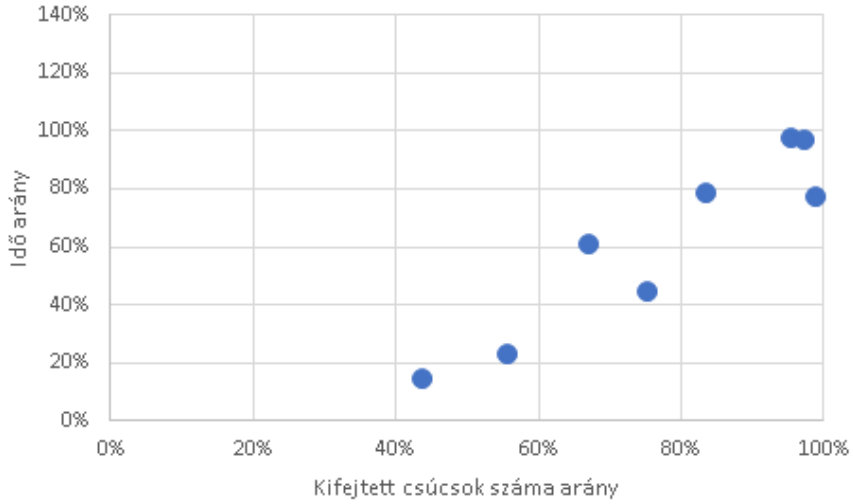


5.8. ábra. Zöld színnel az 1., míg prios színnel a 2. konfigurációk esetében történt csúcs kifejtés csökkenés a Teljesen igény szerinti változatban a Részlegesen igény szerintihez képest

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



(a) 1. konfiguráció



(b) 2. konfiguráció

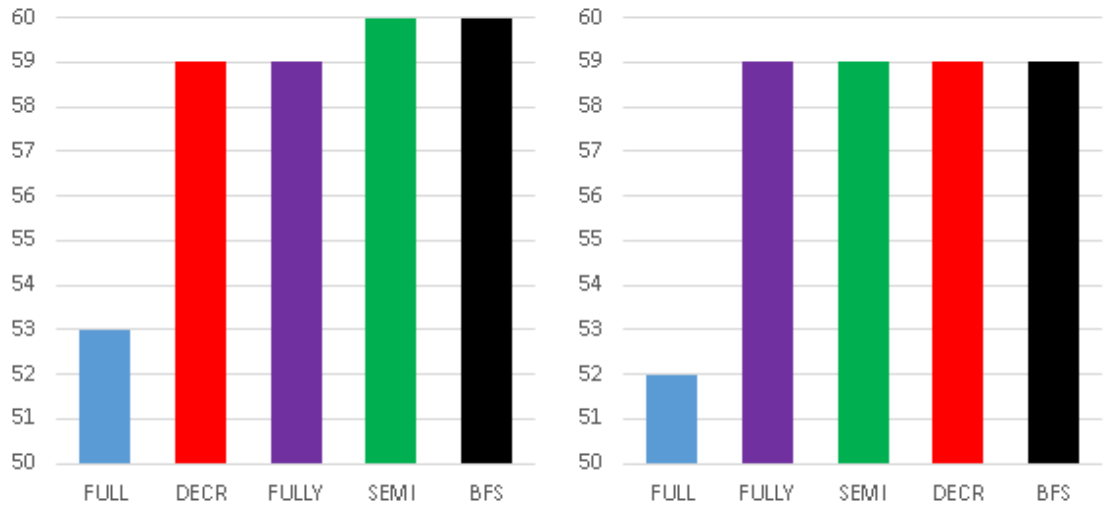
5.9. ábra. A kifejtett csúcsok számának arányához társított verifikációval töltött idő arányát bemutató scatter plot ábrák

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 Teljesen-, illetve a Részlegesen igény szerinti 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 Teljesen igény szerinti 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 a korábbi munkám során kifejlesztett Hierarchikus A* kereséssel támogatott CEGAR megközelítést egészítettem ki. Célom az volt, hogy a Részlegesen igény szerinti Hierarchikus A* változatot hatékonyabbá tegyem az által, hogy csökkentem a kifejtett csúcsok számát, mely befolyásolja a verifikáció gyorsaságát. Az új változat a Teljesen igény szerinti Hierarchikus A*. Ezen új algoritmust a korábbi Hierarchikus A* változatokkal együtt bemutattam a 3. fejezetben.

Mivel ezen algoritmus optimalizációja a Részlegesen igény szerinti változatnak, oly módon, hogy annak alapvető működését nem befolyásolja, így automatikusan ezen változat is 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, ugyanúgy képes a legrövidebb út 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 algoritmus teljesítménymérésének eredményeit. A mérési eredmények alapján ezen változat nem volt képest felvenni a versenyt jelen implementáció mellett a korábbi változatokkal, az egyes állapottérbejárások közötti váltás többletköltségét nem volt képes ellensúlyozni a kifejtendő csúcsok számának csökkentése.

Összefoglalva a dolgozat a következő főbb kontribúcióimat mutatta be:

- Elméleti eredményem a korábbi munkám során kidolgozott A* alapú hierarchikus keresési algoritmus család kibővítése a Teljesen igény szerinti Hierarchikus A* változattal, mely a Részlegesen igény szerinti változathoz képest tovább csökkenti a kifejtett csúcsok számát.

- Gyakorlati eredményként a korábbi munkám során elkészített prototípus változatot átdolgoztam egy stabil architektúrává, hogy a jövőben az mergelésre kerülhessen, illetve megkönnyítse újabb Hierarchikus A*-al való kiegészítését. Mindezt az egyetemen fejlesztett nyílt-forráskódú Theta modellellenőrző keretrendszerbe végeztem.

- Szintén gyakorlati eredményként az elkészült algoritmust integráltam az új architektúrába és elérhetővé tettem azt egy fork formájában[1].

- Publikus és ipari partnerektől eredő benchmark készleteken vizsgáltam az algoritmus hatékonyságát.

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

A jövőben fontos lesz az implementáció alapos áttekintése, mivel egyes esetek során a Teljesen igény szerinti változat több csúcsot fejtett ki a Részlegesen igény szerinti változattal szemben, ami az elmélet szerint nem fordulhatna elő. Ezen esetek a mérési eredmények megjelenítése előtt kiszűrésre kerültek. A korábbi állapotterekre való visszatérés megvalósításában hatékonysági problémára utal, hogy azonos számú kifejtett csúcs esetén a Teljesen igény szerinti változat egyes esetekben gyorsabban tudott verifikációkat végrehajtani.

Másik fontos feladat 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 kutatásom pénzügyi támogatását az ÚNKP, Kulturális és Innovációs Minisztérium, illetve a Nemzeti Kutatási, Fejlesztési és Innovációs Alap biztosította.

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. http://tdk.bme.hu/VIK/DownloadPaper/Informalt-keresesi-strategiak-absztrakcio.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.