Válaszok **Kovács Attila egyetemi tanár** kérdéseire az MTA doktori értekezésemhez

Tisztelt Tanár Úr!

Köszönöm a bírálatot az MTA doktori értekezésemhez. Az általános kritikai megjegyzéseket a jövőbeli kutatásaim során igyekszem figyelembe venni. A bírálatban foglalt kérdésekre az alábbiakban válaszolok. A könnyebb követhetőség érdekében a kérdéseket sorszámmal azonosítom.

KA1: A dinamikus szeletek kiszámítása során mekkora előnnyel jár a függőségi gráf helyett egyéb adatstruktúrák használata? Vannak-e erről valós ipari méretű statisztikák?

A dinamikus programszeletek számításához az egyes programelemek (pl. utasítások) végrehajtáskori előfordulásai közötti összefüggéseket követjük, szemben a statikus szeleteléssel, amikor magukat a programelemeket vizsgáljuk a függőségek meghatározásához. Ez lényegi különbség, mivel míg statikus esetben az elemzés elemi objektumainak száma a program méretének függvénye, addig dinamikus esetben ugyanez a végrehajtás hosszától (végrehajtott lépések számától) függ. A függőségi gráf alapú módszereknél így a gráf méretének nincs más korlátja, mint a programvégrehajtás hossza, ami nem praktikus. Az irodalomban léteznek ugyan olyan heurisztikus megoldások, amelyek a dinamikus függőségi gráfot csak részlegesen építik fel, vagy az ismétlődő részeket újra felhasználva tárolják, de ez az algoritmus család alapvetően nem skálázható valós programokra.

Ugyan a dinamikus függőségi gráf minden információt tartalmaz a programvégrehajtás teljes történetéről, de erre ritkán van szükség a gyakorlati alkalmazásoknál. Az általunk javasolt adatszerkezetek használatával a teljes történet elveszik, és mindig csak a legutolsó állapotra vonatkozó információk elérhetők, de a gyakorlati alkalmazásoknál (pl. debugging) ez elegendő. A szükséges tár így már nem függ a végrehajtás méretétől, ami korlátlan, csak a program által adott pillanatban használt élő értékkel rendelkező változók számától és a függőségek mennyiségétől. Az implementációinkat valós programokra is sikeresen alkalmaztuk, illetve egy valódi, széles körben használt debugging környezetben (GDB) is megvalósítottuk, igazolva ezzel a gyakorlati használhatóságot [7].

KA2: Vajon a különböző architektúra típusok esetén mit mutat a dinamikus csatolás (DFC) metrika?

Ez a metrika az eljáráshívások sorrendiségén és távolságán alapul, amit konkrét programvégrehajtások feldolgozásával számítunk. Az eredeti módszer egyszálú végrehajtási architektúrát feltételez, de a hívási kapcsolatok tetszőleges nyelvű és implementációjú eljárásra (függvényre, metódusra) értelmezhetők, procedurális programozási paradigmát feltételezve.

Komplexebb architektúránál vagy többszálú végrehajtásnál a módszer kiterjesztésére lenne szükség azt a kiindulási gondolatot követve, hogy amikor két számítás (függvényhívás) időben, azaz a végrehajtási szekvencia során, közel van egymáshoz, akkor nagyobb a valószínűsége annak, hogy kapcsolódnak egymáshoz.

Arra vonatkozólag nem végeztünk kísérleteket, hogy a különböző technológiáknak, speciális architektúráknak van-e érdemi hatása az említett “közelség” számszerű értékére, de véleményem szerint a DFC metrikát leginkább a programlogika és a megvalósítás miként befolyásolja. Várhatóan, azonos architektúra szerint fejlesztett különböző programok jellemző DFC metrikái sem lennének összehasonlíthatók.

KA3: Vannak-e mérések arról, hogy milyen mértékben támogatja a klaszterizációs metrika a karbantartók és fejlesztők döntéseit, például refaktorálás vagy újratervezés során?

A függőségi klaszterekkel kapcsolatos kutatások jelenleg még mindig viszonylag korai fázisban vannak: kezdetben a klaszterezettség fogalmát, felismerési, osztályozási módját vizsgálták, utána a linchpin elemek azonosításával foglalkoztak, és csak ezután került terítékre a klaszterek gyakorlati kockázatainak, következményeinek vizsgálata. Utóbbi témához tartozik például az, hogy a szoftverkomponensek hibára való hajlamossága és a klaszterezettség között van-e kimutatható összefüggés. Yang és társai statisztikailag szignifikáns pozitív korrelációt mutattak ki ebben a kérdésben (*Yang et al.: "An empirical study on dependence clusters for effort-aware fault-proneness prediction", in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pages 296–307, 2016*).

Egy kisebb kutatásban mi is foglalkoztunk a különböző klaszterizációs metrikák és a szoftverminőség kapcsolatával, és kimutattuk, hogy sok esetben látható összefüggés az egyes magas szintű strukturális szoftverminőség metrikákkal (*Csaba et al.: "Relating Clusterization Measures and Software Quality", in Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), pages 345-348, 2013*).

A [25] publikációban empirikusan igazoltuk, hogy a nagy változtatás hatáshalmaz okozója a nagy függőségi klaszter lehet, ezért a szoftverkarbantartás és evolúciós folyamatokat alapvetően befolyásolják ezek a szerkezetek.

Mindez azt igazolja, hogy mivel a kérdésben szereplő fejlesztői tevékenységek (refaktorálás, újratervezés) nagymértékben függnek a karbantartás alatt álló szoftver aktuális minőségétől, a függőségi klaszterek ismerete, lehetőség szerinti javítása fontos lehet e tevékenységek során. Ugyanakkor, erre vonatkozó konkrét empirikus mérésekről nincs tudomásom.

KA4: Hogyan viszonyul a dolgozatban bemutatott heurisztikus linchpin azonosítási megoldás a függőségi klaszterekben az alábbi gráfelméleti módszerekhez: betweenness centrality, degree centrality, articulation points (vágópontok), Laplace-mátrix és spektrális centralitás?

Kutatásunkban a szeparátor halmazok gráfelméleti fogalmának kapcsolatát vizsgáltuk a linchpin azonosítás problémájával ([10]-as publikáció 5.4 fejezete). Azt vizsgáltuk, hogy milyen feltételek mellett lehet egy gráfnak (esetünkben függőségi gráfnak) kis szeparátor csúcshalmaza. Abból a hipotézisből indultunk ki, hogy a függőségi gráfok nem véletlenszerűek és valószínűsíthetően nem expander gráfok (ritka, de mégis sűrűn összekapcsolt gráfok), így kevés számú csúcs törlésével legtöbbször csökkenhet a klaszterezettség (*Kawarabayashi and Reed: "A Separator Theorem in Minor-Closed Classes," 2010 IEEE 51st Annual Symposium on Foundations of Computer Science, 2010, pp. 153-162*). Empirikus kísérleteket végeztünk, amelyben az első 10 leggyanúsabb elem együttes eltávolításának hatását vizsgáltuk, és azt kaptuk, hogy sok esetben az első három elem eltávolítása elegendő volt a klaszterezettség jelentős javításához.

Az említett centralitás módszerek valóban kapcsolódnak a problémánkhoz (*Freeman: Centrality in social networks conceptual clarification, Social Networks, Volume 1, Issue 3, 1978, Pages 215-239*), de a legtöbb algoritmus a csúcsok centralitásának kiszámolására általános esetben nagy idő komplexitású (pl. closeness, betweenness esetében köbös a csúcsok számával). Speciális típusú gráfok esetében, pl. ritka gráfokhoz léteznek hatékonyabb algoritmusok ugyan, de az általunk megadott heurisztikus módszer egyszerűsége ellenére (függvényekből kimenő hívások száma - NOI metrika) nagyon jó eredményeket mutatott a kézi eredményekhez és metrikákhoz képest különböző programok esetében: magas klaszterezettség esetén 92% volt az egyezés. Ez a módszer leginkább a degree centrality gráfelméleti módszernek felel meg, amely idő komplexitása az élek számával lineáris.

A vágópontok meghatározására szolgáló mélységi keresés alapú algoritmus lineáris idejű és valóban alkalmazható lenne bizonyos mértékig (*Hopcroft and Tarjan: Algorithm 447: efficient algorithms for graph manipulation. Communications of The ACM, 16(6), 372–378, 1973*). Segítségével beazonosíthatók lennének azok a csúcsok (program elemek), amelyek biztos, hogy valamilyen szintig csökkenthetik a klaszterezettséget, mivel a függőségi gráf több összefüggő komponensre esne szét eltávolításukkal. Véleményem szerint ennek a

módszernek a mi esetünkben két limitációja van: egyrészt, várhatóan kevés ilyen csúcs található valós programokban, másrészt, az, hogy melyik csúcs eltávolításával esik szét a gráf a legnagyobb mértékben egy további számítási lépés lenne.

A spektrális centralitás módszerek a hálózatelméletből (pl. PageRank, sajátvektor, Laplace mátrix) valóban alkalmazhatók lennének a függőségi gráfok esetében is, melyeknek az az előnyük is meglenne, hogy nem csak lokális információt szolgáltatnak, hanem a teljes hálózat (gráf) kapcsán is. Időkomplexitás tekintetében ezek az algoritmusok viszont kevésbé hatékonyak: a csúcsok számával lineárisak, de négyzetesen függnek a gráf legnagyobb fokszámától (*Qi et al.: Laplacian centrality: A new centrality measure for weighted networks. Information Sciences, 2012, 194:240-253*).

KA5: A statikus gráf alapján nem mindig derül ki a valós futásidejű centralitás a függőségi klaszterekben, ezért a statikus és dinamikus metrikák kombinációja javasolt. Vannak-e ezirányú kísérleti eredmények, tervez-e ilyet?

A függőségi klaszterek és a kapcsolódó linchpin elemek és centralitási kérdések kutatása tetszőleges típusú függőségek esetén értelmezhető kérdések. A klaszter fogalma és meghatározási módjai függetlenek attól, hogy az alapvető függőségi halmazok statikus, dinamikus, programszelet vagy más típusú kapcsolatok alapján lettek-e meghatározva. Kísérleteinket zömében statikus függőségi gráfokon végeztük, de elvárásaim szerint dinamikus vagy egyéb, hibrid függőségek esetén is kirajzolódnának a klaszterek és lenne értelme a linchpin és centralitás vizsgálatoknak.

A [28] publikációnkban az ún. uniós szeleteket vizsgáltuk, amik dinamikus szeletek uniójából számíthatók ugyanazon szeletelési kritériumra több végrehajtás során, tehát ez egyfajta hibrid statikus-dinamikus függőség. Kísérleteink egyik mellék eredménye az volt, hogy az így kapott függőségi halmazokat vizsgálva, itt is tapasztalhatók voltak a függőségi klaszterek, ami igazolja azt a feltételezést, hogy dinamikus függőségekkel is érdemes folytatni a klaszterekkel kapcsolatos kutatásokat.

KA6: Vannak-e ismeretek arról, hogy a javasolt eljárás hívási láncokat és eljárás hívási gyakoriságokat használó spektrum-alapú hibalokalizációs technikáknak az adat és vezérlés bonyolultságot tekintve milyenek a skálázódási tulajdonságai?

Az eljárás hívási láncok és eljárás hívási gyakoriságok alapú módszerek alap entitásai a programfuttatás során érintett függvények, metódusok. A szükséges adatszerkezetek felépítése dinamikusan, a programfutással párhuzamosan történik, és ehhez mindössze az eljárások belépési és kilépési eseményeire van szükség. Mivel ez dinamikus elemzés, a (statikus értelemben) lehetséges vezérlési szerkezetek ismeretére nincs szükség, a hívási kapcsolatok felépíthetők egy verem-alapú adatszerkezettel az említett belépési és kilépési információk alapján. Adatok közötti összefüggéseket ugyancsak nem szükséges elemezni és tárolni.

A skálázódást meghatározó tényezők alapvetően a program mérete (eljárások számában), a tesztesetek száma, valamint a hívási láncok hossza és a különböző láncok darabszáma (ezek a hívási verem egyes állapotai). Utóbbi két tényező a program és a tesztek által megvalósított logikától függ, a program és a tesztek mérete, valamint az adat és vezérlés bonyolultsága közvetlenül nem függ össze velük. Más szóval, például, egy bonyolult vezérlési szerkezet eredményezhet kevés és rövid hívási láncot, de sok különböző és hosszú lánc is létrejöhet aránylag egyszerű struktúrájú és adatszerkezetű kód révén is.

Az algoritmusainkat a Defects4J hibaadatbázison és kód benchmarkon vizsgáltuk, amelyben programonként néhány száztól nyolcezer eljárás van, több ezer, akár tizenötezer tesztesettel. A hívási láncok hosszai tipikusan 3-15 között mozogtak, de volt kiugró 1000 hosszúságú lánc is. A különböző láncok száma pár ezertől hárommillióig terjedt. A jelenlegi nem optimalizált implementációkkal ezeket a programokat tudtuk kezelni a kísérleteinkben, de természetesen valós, ipari jellegű felhasználásnál további optimalizálásra lenne szükség. Ilyen lehetne például a láncok optimalizált tárolása az azonos prefixű láncok megosztott kezelésével. A tényleges hívási láncok egyik jellemzője ugyanis, hogy gyakran osztoznak azonos hívási sorozatokon például a program belépési pontjától vagy a tesztesettől számítva, és egy adott ponttól kezdve térnek csak el egymástól, amely tulajdonság remekül felhasználható az optimalizálásra.

KA7: Melyek a programszeletelés lehetséges jövőbeni kutatási irányai?

A programszeletelés fogalmát már az 1980-as években bevezették, és kijelenthető, hogy a legtöbb alap algoritmust a következő 1-2 évtizedben fejlesztették ki. A 2000-es évek elejéig nagy mennyiségű publikáció jelent meg, melyekben különböző szeletelési fajtákat, algoritmusokat, alkalmazásokat, egyes nyelvekre való adaptációkat és eszközöket mutattak be. Ipari alkalmazásba viszonylag kevés eszköz került: leginkább a sérülékenység és szoftverminőség vizsgálatot és programmegértést támogató eszközök integráltak szeletelési algoritmusokat.

Legtöbb alap algoritmus alapvető tár és idő komplexitási kihívások miatt nem került gyakorlati alkalmazásra, így a kutatók a különböző közelítő, heurisztikus megoldásokkal kezdtek el foglalkozni. Az ún. megfigyelés alapú szeletelés (*Binkley et al.: ORBS and the limits of static slicing, 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, pp. 1-10*) egy biztató iránynak tűnik, amelynek az a lényege, hogy az algoritmus nem végez komplex statikus vagy dinamikus elemzést, hanem a programkód szisztematikus szűkítésével és újrafuttatásával megfigyeli, hogy az eredeti programhoz képest megváltozott-e a viselkedése. Amennyiben nem, úgy tovább szűkíti a programot iteratívan egy bizonyos korlátig. Ez természetesen nagy mennyiségű számítási kapacitást igényel, de párhuzamosítható és manapság ez a számítási kapacitás már könnyebben elérhető, mint a korábbi években. A módszernek az az előnye is megvan, hogy tetszőleges technológiájú, heterogén architektúrák esetében is használható.

A programszeletelés további lehetséges jövőbeli irányait nehéz megjósolni, de bizonyos területek jól láthatók (ld. például *Gallagher and Kozaitis: Program Slicing: A Brief Retrospective, in IEEE Transactions on Software Engineering, vol. 51, no. 03, 2025, pp. 720-724*). Várható, hogy egyéb, közelítő és heurisztikus megoldások kerülnek kidolgozásra, amelyek a mai nagy és heterogén rendszerek kezelésére is alkalmasak lehetnek. Az általunk elkezdett hatékony dinamikus elemzés is ezt az irányt képviseli. Új számítási és programozási paradigmák kezelése is előtérbe kerülhet, így például a különböző szkript nyelvek, menedzselt nyelvek, modell-alapú fejlesztés, heterogén rendszerek, de akár a kvantumszámítás területe is lendületet adhat a programszeletelési algoritmusok kutatásának. A modern fejlesztési folyamatok – beleértve a DevOps-ot, elosztott és felhő alapú számítást, mobil környezeteket és a generatív mesterséges intelligenciával támogatott programozást – további kihívásokat jelentenek a programszeletelés gyakorlati alkalmazása tekintetében.

Végül, természetesen a mesterséges intelligencia területén elért korszerű eredmények is integrálhatók a programszeletelési megoldásokba. Itt leginkább a különböző kontextus információknak (természetes nyelvű dokumentációk, hiba-adatok, fejlesztői szöveges üzenetek, stb.) gépi tanulással való integrálása várható a forráskóddal együttesen. Illetve, a nagy nyelvi modellek alkalmazására a programszeletelés területén már most komoly eredmények láthatók (például *Shahandashti et al.: Program Slicing in the Era of Large Language Models, in 2025 IEEE 49th Annual Computers, Software, and Applications Conference, pp. 1224-1233*).

KA8: Melyek a dolgozatban bemutatott módszerek alkalmazásának lehetséges buktatói nagyméretű modern alkalmazásokon?

A dolgozatban bemutatott módszerek többségét azzal a céllal terveztük, hogy valódi, nagyméretű programok esetében is használhatóak legyenek, köszönve a takarékosan tárolt adatoknak és az elvi szinten hatékony algoritmusoknak. A legtöbb empirikus kiértékelés valódi, közepes vagy nagy méretű szoftveren történt. Ez természetesen még nem jelenti azt, hogy a bemutatott módszerek és eszközök továbbfejlesztés nelkül a napi gyakorlatban azonnal alkalmazhatóak lennének. További vizsgálatokra és fejlesztésekre lenne szükség ipari méretű és valós körülmények között végzett esettanulmányokhoz.

A teljesség igénye nélkül, az alábbi problémákat kellene megoldani egy ilyen esettanulmányhoz:

- modern programozási nyelvi verziók és többnyelvű, heterogén architektúrák támogatása,
- speciális architektúrák kezelése, például felhőalapú, mikroszolgáltatások, mobil, beágyazott,
- az algoritmusok kiterjesztése klasszikus procedurális és objektum orientált paradigmákon túlra, például dinamikus és szkriptnyelvek kezelése,
- párhuzamos és elosztott számítás hatékony és biztonságos kezelése,

- implementáció optimalizálása, beleértve például a dinamikus elemzés (programszeletelés, hívási láncok, stb.) programvégrehajtással párhuzamosan történő végzése,
- fejlesztőkörnyezetekbe és folyamatos integrációs környezetekbe való integrálás.

A dolgozatomban is említést teszek az eddigi eredményeinkről a módszerek valós környezetben való alkalmazhatósági elemzéséről, például a hibalokalizációs eszközök fejlesztői környezetbe való integrálásáról [22, 23, 27] és egyéb kapcsolódó kihívásról [15, 17, 20, 34].

Szeged, 2026. január 19.

Dr. Beszédes Árpád, habilitált egyetemi docens