Válaszok **Patricza András az MTA doktora** 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.

PA1: Mik az értekezés fő célkitűzéseinek legfontosabb szoftvertechnológiai kontextusai?

Az értekezés a szoftverfejlesztési folyamat hibakeresési és szoftverkarbantartási fázisaiban felmerülő feladatok támogatására nyújt újszerű megoldásokat. Az ismertetett módszerek konkrétan az alábbi módokon támogatják a fejlesztési munkát. A dinamikus programszeletelés a hibakeresés során abban nyújt segítséget, hogy amikor egy programponton hibát észlelünk futtatáskor, meg tudjuk határozni azokat a részeit a programnak, amelyek befolyásolták a hibás viselkedést, ezáltal csökkentve a keresési teret. Egy ilyen megoldás konkrét debugger eszközökbe kézenfekvő módon beilleszthető (a [7] publikációban a GDB környezetbe való integrálást ismertettük). A dinamikus függvénycsatolás is hasonló: egy programpont lehetséges hatásait adja meg a program többi részére, amit például a szelektív regressziós tesztelésnél vagy általában a karbantartási (kód javítása, átalakítása, megértése, stb.) munkáknál lehet hasznosítani.

A függőségi klaszterek gyakorlati kockázatainak, következményeinek része az, hogy kimutatható a szoftverkomponensek hibára való hajlamossága és a klaszterezettség közötti összefüggés, ami konkrét refaktorálási lépéseket irányozhat elő (*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*). Továbbá, a [25] publikációban empirikusan igazoltuk, hogy a nagy változtatási hatáshalmaz okozója a nagy függőségi klaszter lehet.

Az automatikus hibalokalizációs megoldások beleillenek a modern szoftvertechnológiai irányzatokba, hiszen az egyre nagyobb és komplexebb rendszerek tesztelése, karbantartása, javítása hagyományos eszközökkel egyre nehezebb. A korszerű fejlett mesterséges intelligencia megoldások integrálása az általunk kutatott statisztikai megközelítéssel tovább erősítheti ezt az irányt. Például, gépi tanulással megtanulható a spektrum-mátrix mintázata a hibára hajlamos kódelemek tekintetében és ez segítheti a statisztikai módszert. Nagy nyelvi modellek használata pedig külső kontextus információt adhat például a hibaleírások, a kód, a kommentek vagy a verziókövetőben szereplő adatok alapján. Mindkét terület aktívan kutatott, a szakirodalom biztató eredményekről számol be.

Ami a célkitűzésnek megfelelő programok méretét és jellegét illeti, 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 (ellentétben a legtöbb ismert egyéb technológiai megoldással). 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 heterogén architektúrájú rendszerek kezeléséhez, valamint az esetleges fejlesztőkörnyezetekbe és folyamatos integrációs környezetekbe való integráláshoz.

Ugyanakkor, az is elmondható, hogy az értekezésben javasolt megoldások a korábban ismert módszereket kiegészítik, nem kívánják minden szempontból meghaladni azokat. Például a statikus kódelemzés (statikus szeletelés, függőségelemzés) a sérülékenység vizsgálat, minőségfelmérés, architektúra rekonstrukció területeken továbbra is alkalmasabb, mint a dinamikus vizsgálat. Továbbá, az automatikus hibakeresésnek és hibalokalizációnak más megközelítései is ismertek, amelyek egyes szituációban alkalmasabbak, pl. a szöveges hibaleírások vagy a verziókövetési adatbázisokból történő adatbányászat (többek között mesterséges intelligencia algoritmusokkal) jó kiegészítése lehet a tesztelés statisztikai vizsgálatán alapú módszerünknek.

PA2: A dinamikus függvénycsatolás (DFC) jelentős futási idejének, illetve tárbonyolultságának kezelésére milyen javítási lehetőségek vannak? Felmerül, hogy a ritka adatszerkezetek speciális technikái alkalmazhatók-e a tárbonyolultság csökkentésére?

A dinamikus függvénycsatolás számításának alapja egy nagyon könnyűsúlyú dinamikus elemzés, amely a programvégrehajtás során csak a függvények belépési és kilépési eseményeit rögzíti. Ez alapján egy verem-alapú adatszerkezettel meghatározhatók a szükséges információk. Elvi szinten a módszer igényli a dinamikus hívási fa meghatározását, aminek mérete (csúcsainak száma, szélessége, mélysége) a végrehajtás hosszától és szerkezetétől függ, és ezt a programlogika határozza meg. Ez viszont a program méretével nem áll közvetlen összefüggésben. Maga a DFC metrika kiszámítása és tárolása már függ a program mérettől, hiszen minden függvénypárra meg kell azt határozni.

A gyakorlatban azonban nem szükséges a teljes hívási fát tárolni, ahogy azt a [3] publikációban ismertetjük. Több algoritmust is megadunk, amelyek különböző tár és idő bonyolultságokkal rendelkeznek, és a fa helyett egyszerűbb és kisebb adatszerkezeteken alapulnak (tömbök és vermek). Ezek különböző felhasználási célokra lehetnek alkalmasak attól függően, hogy egyszerre több DFC metrika kiszámítására van-e szükség vagy csak egyre, valamint, hogy a figyelembe vehető távolságot maximalizáljuk-e valamilyen rögzített értékre. Ha kis távolságokban maximalizáljuk az elemzést, akkor az algoritmus tár és idő igénye jelentősen javítható. Tovább optimalizálható a módszer, ha a metrika egyszerűsített változatát alkalmazzuk, amelyben csak a közvetlen hívások vannak elemezve. Empirikus méréseinkkel azt igazoltuk, hogy már 1-3 távolság esetén jó közelítést ad a metrika, így gyakorlati szempontból kihasználhatók ezek a javítási lehetőségek.

A ritka adatszerkezetek speciális technikái véleményem szerint ebben az esetben nem alkalmazhatók, mivel az alapvető szerkezetünk a dinamikus hívási fa, amelyek más szempontból speciálisak. Erre sokkal inkább jellemző az azonos részfák több példányban való megjelenése, mivel intuitíven egy program végrehajtásakor ugyanazon programrészek többször is hívódnak. Az így kapott részfák megosztva is tárolhatók egy irányított aciklikus gráfban, de ahogy az előző bekezdésben említem, gyakorlati esetekben nem is szükséges a fa tárolása, hanem helyette tömörebb adatszerkezetek is alkalmazhatók.

PA3: Van-e olyan gráfelméleti modell, amelyik a 3.3.2 alfejezetben bemutatott dinamikus konvergenciát (fedés és pontosság a függvénycsatolás maximális távolságának függvényében) elméletben is alátámasztja?

A módszerünkben d-vel jelöljük azt a maximális távolságot, ami két függvény előfordulás között fennállhat a DFC metrika szerint ahhoz, hogy azokat csatoltnak tekintsük. A teljes hívási fát tekintve az univerzumnak, egy rögzített d esetében annak lényegében egy részfájára korlátozódik a vizsgálat, pontosan arra, ami csak a “szorosabb” csatolást mutatja a kódelemek között. Ha a d értéket 1-ről indítva fokozatosan növeljük, egyre bővülő kapcsolatokat kapunk, míg végül eljutunk a maximális függőségi relációhoz (amit az “utána végrehajtott” nagyon megengedő, minden lehetséges kapcsolatot tartalmazó, de pontatlan reláció határoz meg).

Mivel a d növelésével a függőségi kapcsolatok csak bővülnek (az unió új elemekkel bővülhet, de elemek nem kerülhetnek ki), egy idő után a növekedés lelassul, végül eléri a szaturációt. A kérdéses kísérletben azt vizsgáltuk, hogy mely d értéknél következik ez be (a fedés/recall vizsgálata). Ez a jelenség egy jellemző halmazlefedési folyamat, amikor kezdetben sok új elem kerül lefedésre, míg később egyre több már lefedett elem kerül újra kiválasztásra. Erre léteznek matematikai elméleti modellek, egy ilyen a szubmoduláris függvények területe, ami gráfokra is kiterjeszthető. Lovász Lászlónak is vannak eredményei a területen, aki a konvex függvényekkel kapcsolatban vizsgálta a szubmoduláris függvényeket, valamint gráfok lefedési kérdéseivel is foglalkozott.

Megjegyzendő, hogy a pontosság (amit a pontos dinamikus szeletek által meghatározott kapcsolatokhoz mérünk) fordított jellegű konvergenciát mutat, ugyanis a DFC algoritmus hamis függőségeket is behoz. Ahogy bővítjük a függőségi relációt, egyre több ilyen hamis kapcsolat kerül be, viszont ez is egy idő után stabilizálódik, mivel a kiválasztható hamis kapcsolatok száma idővel lecsökken.

PA4: Hogyan viselkedik a dinamikus függvénycsatolás algoritmus akkor, ha a rendszerben globális változókon keresztül csatolt egyébként konkurens függvények vannak, hiszen ezek hívási és változóhivatkozási kapcsolata véletlenszerű lehet?

Ez a módszer az eljáráshívások sorrendiségén és távolságán alapul, amit konkrét programvégrehajtások feldolgozásával számítunk. Az eredeti módszer egyszálú végrehajtást feltételez, és a csatolási kapcsolatokat egy-egy szálra lehet meghatározni. Az így kapott eredményeket ugyanakkor lehetne aggregálni az egész rendszerre valamilyen súlyozott átlag megoldással. A konkurens módon végrehajtott függvények minden bizonnyal zajt eredményeznének a számításban, így azokat valamilyen módon kezelni kellene.

Az alap heurisztika általánosítható más paradigmákra és környezetekre, akár többszálú, konkurens végrehajtásra is. A módszernek az a kiindulási gondolata, hogy amikor két számítás (függvényhívás) időben, azaz a végrehajtási szekvencia során, közel van egymáshoz, akkor nagyobb a valószínűsége annak, hogy kapcsolódnak egymáshoz. Párhuzamos számítás esetén ennek a feltételezésnek nincs alapja, így más megközelítést kellene alkalmazni. Egyrészt, egy többszálú program esetében a szálak létrehozásának körülményeit lehetne vizsgálni (pl. melyik eljárás hozza létre melyik szálat), és ez alapján meghatározni az egyes szálak közötti csatolási kapcsolatokat. Másrészt, mivel a különböző párhuzamos számítási egységek jellemzően közös adatterületeken keresztül kommunikálnak egymással, a módszert ki lehetne terjeszteni úgy, hogy a közös adatelérések között határozunk meg egy távolsági mértéket, és azzal számoljuk tovább a csatolást. Például, az adat írás és olvasás műveleteket végző eljárások közötti csatolást az adott műveletek sorrendisége és időben egymástól való távolsága alapján határozzuk meg.

PA5: Az értekezésben bemutatott módszerek gráfelméleti reprezentációja segíthet-e a hatékony algoritmusok kidolgozásában?

Az értekezésben bemutatott legtöbb módszer alapvető adatszerkezete gráfként is értelmezhető, de megjegyzendő, hogy tulajdonképpen csak a függőségi klaszteréknél használtunk tényleges gráfokat a függőségi kapcsolatokhoz, a legtöbb módszernél okos adatszerkezetekkel elkerültük a gráfok és fák tárolását. A dinamikus szeletelésnél függőségi halmazokat, a dinamikus függvénycsatolásnál speciális lokális adatszerkezeteket használtunk. A spektrum-alapú hibalokalizáció alap adatszerkezete a spektrum-mátrix, ami kezelhető gráfként, de ez nem jellemző. A kapcsolódó függvényhívási láncok és gyakoriságok esetében a láncok dinamikus hívási faként reprezentálhatók, de a módszerünkhoz mindig csak az aktuális láncok ideiglenes tárolására van szükség.

Bár a többi területnél is elképzelhető a gráfelméleti reprezentáció és elméleti eredmények alkalmazása, a továbbiakban a függőségi klaszterek és a linchpin elemek azonosítása esetében vizsgálom meg a lehetséges megközelítések alkalmazását.

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.

A gráf centralitás módszerek 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 bonyolultságú (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 gráf vágópontok meghatározására szolgáló mélységi keresés alapú algoritmus is alkalmazható lenne esetünkben 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) is alkalmazhatók lennének a függőségi gráfok esetében, 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*).

Lovász László gráflimesz elmélete is kötődik a témához, melyet a centralitási módszerekhez is alkalmaztak (*Avella-Medina et al. "Centrality Measures for Graphons: Accounting for Uncertainty in Networks," in IEEE Transactions on Network Science and Engineering, vol. 7, no. 1, pp. 520-537, 2020*), de a ritka gráfokra vonatkozó kapcsolódó eredmények is megvizsgálhatók a problémakörünk tekintetében.

PA6: Van-e realitása annak, hogy az egyes tesztek eredményeit nemcsak bináris Go/NoGo értékükkel vegyük figyelembe, hanem a tényleges hibajelzéseket vagy súlyosságukat is használja a függvényhívási láncok számításán alapuló hibalokalizációs algoritmus?

Az elemi spektrum-alapú hibalokalizációs algoritmusok egyik nagy hiányossága, hogy a progremelemekkel kapcsolatosan semmilyen kontextus információt nem használnak, csak azok tesztek általi fedését. Az értekezésben bemutatott mindegyik kapcsolódó módszer ezt a hiányzó kontextust pótolja valamilyen módon. A függvényhívási láncok esetében a hívási hely jelenti ezt a kiegészítő információt, a hívási frekvenciák esetében a hívási pontok különbözősége, a programszeletelés alapú módszernél pedig a tényleges függőségek. Megjegyzendő viszont, hogy sok egyéb kontextus információval bővíthető a folyamat (*Szatmári Attila: Enhancing Spectrum-Based Fault Localization Using Contextual Knowledge, Doctoral thesis (PhD), University of Szeged, 2025*).

A hibajelzések különböző adatai remekül kiegészíthetnék az általunk javasolt módszereket, beleértve a függvényhívási láncok módszerét. Ilyenek lehetnek maga a bejelentés szövege, a kapcsolódó fejlesztői megjegyzések, a metaadatok, például súlyosság, nyomonkövethetőség a követelményekhez, a forráskódhoz vagy a tesztesethez, stb. Egy lehetséges módja a kiegészítésnek az lenne, hogy a hibajelzés nyomonkövethetőségét felhasználva a kapcsolódó kódelemhez és/vagy tesztesethez, a súlyossággal lehetne skálázni a kapcsolódó hívási láncok vagy tesztesetek fontosságát a hibalokalizációkor.

Korszerű mesterséges intelligencia megoldásokkal ezen információk egy része akár automatikusan is kinyerhető, például gépi tanulással lehetne az egyes kódelemek hibára való hajlamosságát becsülni, de akár a nagy nyelvi modellek is segíthetnek a forráskód és a kapcsolódó kommentek elemzése alapján.

PA7: A függvényhívási frekvenciák számításán alapuló hibalokalizációs algoritmusnál a szokásos fedési metrikák fényében hogyan alakul a hibalokalizációs folyamat? Jelentene-e például különbséget az, hogy a tesztkészlettel csomópont vagy útfedést célozd meg annak tervezője?

A spektrum-alapú hibalokalizáció legelterjedtebb alap lefedettségi adatszerkezete a tesztesetenkénti utasítás lefedettség, azaz a spektrum mátrix a tesztesetek és a programban lévő utasítások lefedettségi adatait

tartalmazza (ez az ún. találati / hit-based mátrix). Utasítások helyett magasabb szintű programozási egységek, például függvények is gyakran képezik a spektrum-mátrix kódvektorait. Kutatásaink egy részében függvény szintű elemzést alkalmaztunk.

Érdekes azonban észrevenni, hogy a hibalokalizáció kutatás kezdeti éveiben számos egyéb lefedettség típussal definiálták a spektrumot és végeztek velük kutatásokat [76, 115, 132]. Valószínűleg a találati spektrum egyszerű számítási módja, valamint a tár és idő takarékosság miatt az újabb kutatásokban az egyéb módszerekre már nem jut annyi figyelem.

Másféle lefedettség típus alkalmazása alapvetően azt jelenti, hogy a spektrum mátrix oszlopai (amelyek a kódot reprezentálják) a lefedettség alapját képező kódelemet írják le. Például egy döntési lefedettségnél az oszlopok a kódban lévő egyes elágazásokat (alapblokkokat) jelentik, útvonal lefedettségnél a különböző útvonalakat. Látható, hogy ez jelentős számítási és tárolási kapacitás növekedést jelentene. Ráadásul, útvonal lefedettségnél el kell dönteni, hogy milyen típusú útvonal lefedettséggel dolgozunk, hiszen általános esetben korlátlan számú különböző útvonal lehetséges egy programban (a ciklusok miatt), ezért valamilyen korlátozásos megoldást kell választani, például a ciklusok ismétlési számát korlátozni vagy komplexitás alapú lefedettséget alkalmazni (ilyen például a McCabe lefedettség).

Mindazonáltal, a kérdésben szereplő gyakoriság alapú módszerünk szerintem általánosítható lenne másféle lefedettség típusokra. Például, döntési és útvonal lefedettség esetén azt néznénk, hogy a vezérlési folyam (CFG) szerint hány különböző helyről érkezünk az adott döntéshez vagy útvonalhoz.

Egy másik lehetséges adaptáció szerint a mátrix kódelemei az utasítások lennének, de a gyakoriságot a vezérlési folyam egyes különböző útvonalai szerint határoznánk meg, tehát ha egy utasítást több különböző helyről is el tudunk érni, az nagyobb súlyt kapna. Ez a megoldás véleményem szerint praktikusan megoldható lenne jó tár és idő bonyolultsággal, és várakozásaim szerint érdekes eredményeket lehetne elérni a segítségével.

PA8: A függőségi klaszterekhez kapcsolódó linchpin elemeknél a gyakorlat számára is érdekes lenne, hogy a beazonosított részeknek mi a funkcionalitása?

A kutatásaink során esettanulmány jelleggel megnéztük a benchmark programjaink egy részénél a beazonosított linchpin elemek szerepét az adott programban. Például a [10] publikáció 5.1 fejezetében kitérünk arra, hogy a magas klaszterizációs metrikákkal rendelkező programoknál (13 darab) a három legnagyobb klaszterizáció csökkentést elérő linchpin jelölt függvénynek mi a neve és milyen funkcionalitással bír. Az esetek döntő többségében a beazonosított függvények valamilyen központi szerepet játszanak az adott programban (pl. *main*, *exec_command*, *HandleCommand*, compress programnál a *compress* függvény), ami igazolja a függőségi klaszter szemantikáját és a linchpin azonosítási módszer helyességét. Kutatásunknak nem volt célja ezt a kérdést szisztematikusan vizsgálni, így az egyelőre nem mondható meg, hogy milyen pontossági aránnyal működik a módszerünk. További érdekes kérdés lehet, hogy a beazonosított programelemek refaktorálása lehetséges-e annak érdekében, hogy a függőségi klaszter lecsökkenjen vagy megszűnjön.

PA9: Mennyiben tekinthetők az értekezés egyes részeiben benchmarkként használt szoftver elemek reprezentatívnak?

A dinamikus programszeleteléssel kapcsolatos első kísérleteket kisméretű C programrészleteken végeztük (a 1999-es publikációnk [14] néhány száz soros kódra hivatkozik). Ezután már valódi, 1-21 ezer soros C programokkal kísérleteztünk a [1,2] cikkekben. Ezek jellemzően olyan programok, amelyeket abban az időszakban (2000-es évek eleje) gyakran használtak a hasonló empirikus kutatásokban. A módszerünk implementációját Java programokra hasonló méretű, de ugyancsak valódi programokon teszteltük (a [28]-as publikációban 1-17 ezer soros kódok szerepelnek). A dinamikus függvénycsatolással kapcsolatos kísérleteinket hasonló méretű (1-2 ezer soros), de valódi Java nyelvű nyílt forráskódú szoftvereken végeztük.

A függőségi klaszterekkel kapcsolatos kutatásaink során többféle benchmarkot használtunk, de ezek is mind valódi nyílt forráskódú szoftverek. Egyrészt, használtunk egy 29 darab közepes méretű C programból álló halmazt, amelyben a programok sorainak száma néhány ezertől 200 ezerig terjed. Másrészt, két valódi ipari alkalmazás is tárgya volt a vizsgálatoknak, a WebKit böngészőmotor, ami közel 2 millió sor C++ nyelvű kódból áll, valamint a

GCC fordítóprogram C nyelvű komponensei, amik közel 4 millió sort tesznek ki. Úgy vélem, hogy e benchmark programok együttesen C/C++ technológiai környezetet tekintve reprezentatívak.

A dolgozat további részeiben a spektrum-alapú hibalokalizációs kísérletek zömét a Defects4J nevű standard benchmarkon végeztük. A kapcsolódó szakirodalomban ez az egyik leggyakrabban alkalmazott programadatbázis, mivel a programok mellett megfelelő mennyiségű tesztesetet, valamint valódi hibákat (a kapcsolódó hibajavításokkal) tartalmaz. Az alapvető programok valódi nyílt forráskódú szoftverek, melyek között vannak kisebbek, közepes méretűek, de nagyok is. Mindegyik programhoz több, kézzel validált valódi hiba is tartozik (tehát, nem mesterségesen injektált, mint sok más hasonló adatbázisban). Minden hiba a program két verziójával van leírva, a hibás verzióval és a javítottal, ami a hibalokalizációs kísérletekhez is megfelelő.

Benchmark verziótól függően, a Defects4J 6-17 programból áll (amiatt, hogy ezek a programok különböző Java verziókat támogatnak és a kísérleteink egyéb korlátai miatt az egyes publikációkban változó összetételű a konkrét szoftverek halmaza). A programok méretei 10-100 ezer sorig terjednek, a tesztesetek száma 100 és 15000 között, és a teljes benchmark 835 hibát tartalmaz (6-174 hiba programonként). Mivel a programok, a tesztesetek és a hibák is valódi szoftverekből származnak, méretük pedig nem triviális, ez a benchmark is reprezentatívnak tekinthető ebben a technológiai környezetben.

PA10: A szoftver területen számos komplexitási metrika létezik. Lehet-e ezeket figyelembe venni az értekezés egyes részeiben a becsléseknél?

Az értekezés célkitűzéseit szem előtt tartva, a szoftverekre vonatkozó komplexitás metrikák az alábbi módokon segíthetnek. Mivel több javasolt módszer is a hibák keresését, az okok feltárását és a javítást hivatott segíteni (dinamikus programszeletelés, függőségi klaszterek, hibalokalizáció), ezeket a tevékenységeket leginkább a kognitív komplexitást leíró metrikák segíthetik, például a vezérlési szerkezetek mélységét, adatfolyamok bonyolultságát, forráskód olvashatóságot figyelembe vevő metrikák. A klasszikus strukturális komplexitás metrikák, mint például a McCabe-féle ciklomatikus komplexitás, ami a vezérlési folyamokat írja le, vagy a Halsted-féle, információtartalmat tükröző metrikák az értekezés több területén is hasznosak lehetnek, mivel ezek a hibakeresés és karbantartás különböző tevékenységeinél is használhatók.

A magasabb (tervezési és architekturális) szintű komplexitás metrikák, például csatolás és kohézió véleményem szerint inkább a függőségi klaszterek vizsgálatánál segíthetnek, mivel azok nem lokális összefüggésekre mutatnak rá, hanem sokkal inkább globális jellemzőkre. Az egész rendszerre vonatkozó metrikák, például a karbantarthatósági index több metrikának az aggregálásából állnak elő, így ezek inkább a hibára való hajlamosságot, a módosítás nehézségét, a programmegértés nehézségét tükrözik.

Az említett metrikák egyrészt beépíthetők lennének az egyes algoritmusokba valamilyen súlyozott kiegészítő attribútumként, amivel például a hibára hajlamos elemek előrébb kerülhetnek a vizsgálat során. Másrészt, az egyes módszerek által szolgáltatott elemek (leggyanúsabb hibás elem, függőségi klasztert összetartó linchpin elem, leginkább csatolt függvény, stb.) komplexitás metrikájával lehetne további becsléseket adni a szoftverfolyamat következő lépéseire vonatkozóan, például a javítás és refaktorálás költségére. További érdekes kísérlet lenne megvizsgálni, hogy az algoritmusok által szolgáltatott elemek milyen korrelációt mutatnak a komplexitás metrikájukkal. Például, igaz-e az, hogy a hibás elem vagy a linchpin jellemzően magas komplexitás metrikával bír?

PA11: Az alkalmazott módszerek a tesztelés melyik fázisában (modul, integrációs tesztelés) hatékonyak vagy hatékonyabbak?

A tesztkészlet sok javasolt módszernél alapvető bemenő paraméter, és az algoritmusok tervezésénél valamint a kiértékelésnél a leginkább alkalmas teszt típusok és teszt szintek tesztesetei lettek használva. Ugyan a legtöbb módszernél explicite nem követeljük meg a tesztek jellegét, de az algoritmusok sikerességét mégiscsak meghatározzák azok.

Ez a kérdés a függőségi klaszterek témáját kivéve az értekezésben ismertetett minden módszert érinti, de leginkább a spektrum-alapú hibalokalizációt. Dinamikus elemzés lévén, ahol a programkódok összefüggéseit vizsgáljuk, általánosan elmondható, hogy alapvetően fehérdoboz teszteléshez állnak közel a módszereink, és az

alacsonyabb tesztelési szinteket célozzák úgy, mint az egységtesztelés, esetleg a (szürkedoboz) integrációs tesztelés. Jó példa erre a kódlefedettség mérő eszközök vizsgálata, ahol a kérdéses eszközök alapvető felhasználási területe a fehérdoboz egységtesztelés. A dinamikus programszeletelés és függvénycsatolás esetében nincs különösebb elvárás a tesztek jellegére, így ezeken a területeken a kérdés kevésbé releváns.

A spektrum-alapú hibalokalizációs módszerekhez is leginkább az egységtesztek alkalmasak, és a szoftverfejlesztési folyamat e szintjén legvalószínűbb az alkalmazásuk. Ezt tovább erősíti az a tény is, hogy a hibalokalizáció jellemzően fejlesztői tevékenység a debugging keretén belül, és az egységtesztelés legtöbbször a fejlesztő tesztelési feladatköre.

PA12: Származtathatóak-e olyan tervezési kritériumok a teszt tervezés számára, amelyek betartása esetén a bemutatott módszerek hatékonysága nő?

Az előző kérdés kapcsán rögzíthető, hogy alapvetően az egységtesztek tervezését érdemes vizsgálni.

A spektrum-alapú hibalokalizáció sikerességét alapvetően meghatározza a spektrumot előállító tesztek jellege, mennyisége és minősége, és ez érvényes a klasszikus módszerekre, de az általunk javasolt továbbfejlesztett algoritmusokra is. A tesztkészlet minőségét sokféleképpen lehet vizsgálni, ugyanakkor arra vonatkozóan viszonylag kevés kutatási eredmény létezik, hogy melyik minőségi szempont milyen mértékben és milyen körülmények között befolyásolja a módszerek hatékonyságát. Az általános kódlefedettség egy tesztkészlet vonatkozásában egy nagyon egyszerű aggregált mérőszám, amiben a tesztesetek lefedettségei összesítve szerepelnek, így önmagában nehezen látható az, hogy a lefedettség milyen mértékben befolyásolja a spektrum-alapú módszerek hatékonyságát. Triviális, hogy azok a kódelemek, amelyeket egy teszteset sem fed le, kiesnek az algoritmusok látóköréből (az általunk javasolt módszerek esetében is), gyakorlatilag a spektrum mátrix ezen elemekre vonatkozó oszlopai törölhetők. A magas kódlefedettség tehát alapvető elvárás.

Ha csak a bináris spektrum mátrix struktúráját, a lefedettségi információ “mintázatait” tekintjük, egy másik fontos jellemző kerül előtérbe, ami nagyobb mértékben befolyásolja a hibalokalizáció sikerességét. Nevezetesen, az egyes kódelemekre vonatkozó lefedettségi vektorok diverzitása. Ha sok, azonos lefedettségi mintázatú elem található a spektrumban, akkor (mivel ezek nem különböztethetők meg egymástól) romlik a hatékonyság. Az ideális spektrumban kevés azonos lefedettségi vektor van. Erre vonatkozólag léteznek vizsgálatok, amik igazolják, hogy a diverzitás valamilyen metrikával kifejezve jól előrejelezheti a hibalokalizáció sikerességét átlagos esetben (*Horváth et al.: Test Suite Evaluation using Code Coverage Based Metrics. In Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST 2015), pages 46-60, 2015*; *Abreu et al.: On the accuracy of spectrum-based fault localization. In Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007, pp. 89–98*).

Ez a tényező az általunk javasolt módszerekre is hatással van, de itt további szempontok is előtérbe kerülnek. A hívási láncok esetében azok a kedvező tesztkészletek, amelyeknél a diverzitás nem csak a lefedettségben mutatkozik meg, hanem abban is, hogy egy-egy teszteset mennyire különböző eljárásokat és eljárás sorozatokat hív meg, másszóval hívási struktúra szerint redundánsak-e a tesztek, vagy kellően különbözőek. A hívási gyakoriságok esetében nagyon hasonló a kérdés: mennyire különböző hívási kontextusokból juthatunk el egy-egy kódelemhez: ha több helyről, akkor az javítja a lokalizációs képességeket.

Legújabb kutatásunkban vizsgáltuk a tesztesetek “egységteszt jellegét”, vagyis, hogy a tesztesetek mennyire izoláltan tesztelnek csak egy-egy eljárást a többi elszigetelésével vagy sem (*Szatmári et al: Influence of Pure and Unit-Like Tests on SBFL Effectiveness: An Empirical Study. In Proceedings of the 2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pages 392-399, 2025*). Ideális esetben, amikor minden teszteset ilyen jellegű, akkor a hibalokalizáció szinte triviális. Viszont, érdekes kérdés, hogy egy tényleges tesztkészlet esetében milyen arányban kell, hogy legyenek az ilyen ideális egységtesztek a sikeres lokalizáláshoz.

Az értekezésben szereplő dinamikus programszelet alapú spektrum mátrix esetében [26] is alapvető jelentősége van a tesztesetek struktúrájának és minőségének. Az előzőekben vázolt elvárásokon túl itt még azt a szempontot is figyelembe kell venni, hogy a programszeletelés kiindulási pontja (a szeletelési kritérium) mennyire könnyen származtatható a teszteset kódjából. Egy szabályos egységteszt az *assert* utasítások segítségével ellenőrzi a

kapott eredményt az elvárthoz képest, így az assertek megléte és értelmezhetősége kulcsfontosságú. A gyakorlatban az egységtesztek írásakor sokszor nem tartják be az erre vonatkozó javaslatokat, és például egy tesztben több assert-et használnak vagy assert helyett más megoldást alkalmaznak az eredmény ellenőrzésére. További probléma az egyéb egységteszt írási elvek be nem tartása, ilyenek az izoláció és szeparáció a többi komponenstől. A tesztelési szakirodalom sok ajánlást tesz a megfelelő minőségű egységtesztek készítéséhez és az ún. “test smell”-ek kerüléséhez. Ezekhez a mi módszerünk kapcsán az említett assert-ek meglétét és szabályosságát tenném hozzá.

PA13: Az ilyen komplexitású problémák esetén az utolsó időben robbanásszerűen terjed a gépi tanulás alapú módszerek használata. Lehet-e ezeknek ennél a problémakörnél érdemi szerepük?

A gépi tanulás alapú módszerek, valamint a nagy nyelvi modellek új lendületet adtak az értekezésben bemutatott kutatási területeken belül is.

A programszeletelés, valamint a függőség és csatolás elemzés területén leginkább a különböző kontextus információknak (természetes nyelvű dokumentációk, hiba-adatok, fejlesztői szöveges üzenetek, stb.) gépi tanulással való integrálása látható a forráskóddal együttesen. Ezek a kiegészítő információk olyan tudást kódolhatnak, amelyeket gépi tanulással megtanulva különböző predikciók adhatók, amik pontosabbá és hatékonyabbá tehetik a folyamatot. Többek között az alábbiakra van példa az irodalomban: kódelem osztályozása aszerint, hogy a szeletben benne van-e vagy sem, függőségek súlyozása fontosság szerint, szeletben lévő elemek rangsorolása. Fontos azonban megjegyezni, hogy a programszeletelés és egyéb függőségelemzés klasszikus algoritmusai bizonyos korlátok között determinisztikus és megbízható (sound) eredményt adnak, de a mesterséges intelligencia által adott eredmények esetében nem lehet ilyen garanciákat adni.

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*). A nagy nyelvi modellek segítségével pontosabban, szándék és kontextus alapon lehet lekérdezni a szeleteket, ami konkrét alkalmazási területhez, a program logikájára és egyéb aspektusaira is vonatkoztatható, és nem pusztán szintaktikai alapon, mint a hagyományos esetben.

A másik terület, melyben nagy szerepe van jelenleg is a mesterséges intelligenciának, ami a jövőben várhatóan tovább fog növekedni, az a spektrum-alapú hibalokalizáció. Többféle megoldással lehet találkozni az irodalomban, többek között a hibarangsor tanulása (*Le et al.: A Learning-to-Rank Based Fault Localization Approach using Likely Invariants. In Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016, 177-188*), formulák kombinálása (*Xuan and Monperrus: Learning to Combine Multiple Ranking Metrics for Fault Localization. In IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 191-200*), lefedettségi mátrix és hibainformáció mélytanulási modellel való megtanulása, majd a hiba predikciója.

Utóbbi területen mi is végeztünk kutatást, melynek keretében arra voltunk kíváncsiak, hogy mennyire adnak stabil eredményeket a mélytanulás alapú hibalokalizációs megoldások, ha ugyanazon inputon többször is megtanítjuk a modellt. Eredményeink azt igazolták, hogy az irodalomban fellelhető megoldások még nem teljesen kiforrottak, hiszen sok esetben nem megbízható a modellek által adott eredmény [12].

A legújabb irányzat a mesterséges intellienciával támogatott hibalokalizációban a több ágenses nagy nyelvi modellek alkalmazása a folyamatban (pl. *Rafi et al.: A Multi-Agent Approach to Fault Localization via Graph-Based Retrieval and Reflexion, https://arxiv.org/abs/2409.13642, 2025*). Az alap ötlet itt az, hogy különböző LLM ágenseket alkalmaznak specifikus feladatokra, amelyek különböző adatforrásokból dolgoznak: kontextus kinyerő, debug támogató, kód átnéző, stb., majd ezek összetett promptolásával kaphatjuk meg a leginkább releváns választ a lokalizációs kérdésünkre. Mivel több forrásból dolgozik, nagyon jó hatékonyságot tud elérni ez a módszer, de természetesen a mesterséges intelligencia megoldások hátrányai ezt is terhelik.

PA14: Melyek az értekezésben bemutatott módszerek jövőbeli lehetséges kutatási irányai?

Az értekezésben bemutatott módszerek kísérleti környezetben, de legtöbb esetben valós alkalmazásokon lettek vizsgálva. Az eredmények közvetlen hasznosítása a módszerek olyan irányú továbbfejlesztése révén történhet, amelyek közelebb viszik azokat ipari méretű és valós körülmények között történő alkalmazáshoz. A teljesség igénye nélkül, az alábbi problémákat kellene megoldani ehhez a legtöbb tézispont tekintetében:

- 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 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.

A függőségi klaszterek területén meglátásom szerint a jelenleg legfontosabb kutatási irány az lenne, hogy jobban megértsük, milyen körülmények között jelentenek problémát a klaszterek a szoftverekben. Jelenleg képesek vagyunk azokat kimutatni, osztályozni, számszerűen jellemezni, de nagy a bizonytalanság abban a tekintetben, hogy mikor érdemes a klasztereket refaktorálni és mikor nem jelentenek veszélyt a karbantartási folyamatokra. A linchpin elemek azonosítására további hatékony és általánosítható algoritmusok kidolgozására lenne szükség, mint ahogy az sem teljesen világos, hogy milyen esetekben lehet egyetlen ilyen elemre számítani vagy több kódrész lehet együttesen felelős a nagy klaszterezettségért.

A spektrum-alapú hibalokalizációt véleményem szerint már nagyon sok megközelítésből vizsgálták, és bár az előző kérdésnél említett mesterséges intelligencia alapú továbbfejlesztések javulást hoznak a pontosságban, az

a meglátásom, hogy nem célravezető az ilyen jellegű kutatásokat folytatni (amelyeknek egyetlen célja a pontosság további javítása), mivel a kisebb mértékű javulásnak nagyon nagy számítási és tárolási költségei vannak.

A napi gyakorlatban történő alkalmazásnak szerintem inkább az a legnagyobb akadálya, hogy a fejlesztői és tesztelői környezetekben nem elérhetők ezek az eszközök, így nem valószínű azok befogadása a fejlesztők részéről. Az lenne a cél, hogy elfogadható pontosságot tudjanak produkálni a módszerek, viszont hatékonyan legyenek futtathatók és könnyen elérhetők a meglévő környezeteken keresztül. 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