HORNYÁK OLIVÉR

# Esettanulmány szoftverek hasonlóságának vizsgálatára

A tanulmány egy konkrét példán keresztül mutatja be, hogyan lehet szoftverek hasonlóságát elemezni. Kiindulásként az elvégzendő feladathoz két szoftverfejlesztési projekt teljes forráskódja állt rendelkezésre. A fejlesztői dokumentáció hiányos volt, a vizsgálathoz nem nyújtott támpontot. A vizsgálat célja az volt, hogy megtalálja azokat a mérőszámokat, amelyek objektív módon írják le a hasonlóságot, illetve *különbséget. Az adott feladat ismertetése mellett a tanulmány áttekinti azokat a módszereket és algoritmusokat, amelyek a szakirodalomban ismertek. A tipikus felhasználási területek a plágiumdetektálás,* a rosszindulatú kódrészletek felismerése *és a* dekompiláció.

**Kulcsszavak:** *forráskód-hasonlóság, plágiumdetektálás, dekompiláció, esettanulmány*

**Szerzői információ**

**Hornyák Olivér,** Miskolci Egyetem

https://orcid.org/ 0000-0003-0989-6109

**Így hivatkozzon erre a cikkre:**

Hornyák Olivér. „Esettanulmány szoftverek hasonlóságának vizsgálatára”.

*Információs Társadalom* XXIII, 1. szám (2023): 100–116.

*https://dx.doi.org/10.22503/inftars.XXIII.2023.1.6*

*A folyóiratban közölt művek*
*a Creative Commons Nevezd meg! – Ne add el! – Így add tovább! 4.0*
*Nemzetközi Licenc feltételeinek megfelelően használhatók.*

OLIVÉR HORNYÁK

# A case study for detecting software similarity

This paper presents a specific case study to determine the similarity / difference of two software projects. The investigation was based on the source code which was available for the analysis. The main goal of the investigation was to identify the metric that can be used as a forensic evident the case. Besides presenting the case study the paper overviews the methods and algorithms from the literature. Typical areas are plagiarism detection, malicious code detection, decompilation.

**Keywords:** *source code similarity, plagiarism detection, decompilation*

*All materials*
*published in this journal are licenced*
*as CC-by-nc-nd 4.0*

## Bevezetés

A szerző informatikai igazságügyi szakértő és egy konkrét szakértői feladat inspirálta a cikk megírására. A tanulmány szándékosan nem közöl olyan adatokat, amelyből a feladat megbízója visszakereshető lenne.

A feladat két szoftverfejlesztési projekt hasonlóságának vizsgálata volt. A megbízót hatósági eljárás alá vonták, és ennek keretein belül volt szükséges annak a vizsgálata, hogy egy kifejlesztett ügyviteli szoftver mennyiben különbözik egy másik vállalkozás szoftverétől. A megbízó vállalkozása egy olyan üzletágban tevékenykedett, ahol különböző cégek kerültek kapcsolatba egymással, melyek szövevényes módon voltak egymás alvállalkozói, tulajdonosai, utódai. A vállalkozói hálózat működését a hatóságok is vizsgálni kezdték, így vált szükségessé az igazságügyi szakértői felkérés.

Az igazságügyi szakértői feladat tárgya a vállalkozás munkájához használt egyedi fejlesztésű ügyviteli szoftver vizsgálata volt.

Az *ügyviteli szoftver* elnevezést tágan kell értelmezni: a napi ügymenet támogatásától a készletnyilvántartásig, a feladatok kiosztásától azok végrehajtásának ellenőrzéséig, a működés optimalizálásáig voltak vizsgálandó funkciók. A hatóságok gyanúja szerint az egyik cég által birtokolt szoftver megegyezett a másik cég által birtokolttal. Ugyan a megbízó ezt nem ismerte el, a szoftver alapötlete azonban valóban azokból az időkből ered, amikor az eredeti szoftver tulajdonosa és a megbízó partneri viszonyban álltak egymással. De amikor a (gazdasági) élet külön utakra sodorta őket, a megbízó saját fejlesztésbe kezdett, és a saját ötleteit is megvalósítva önálló szoftverfejlesztést végeztetett. A szakértői feladat az volt, hogy ezt az állítását alátámassza vagy cáfolja a szakvéleményben.

## Szakmai motiváció

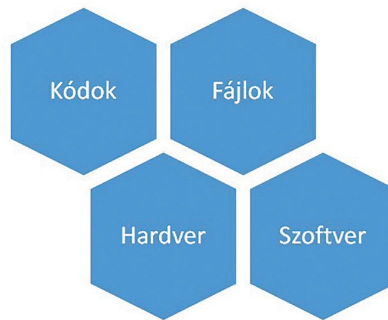
Bár az időben jelenlegi tudásunk szerint nem lehet utazni, az igazságügyi szakértői feladat nehézségét mutatja, hogy mégis úgy kell a múltbéli eseményekről megállapításokat tenni, hogy az tükrözze a dolgok akkori állapotát, a folyamatok alakulását. Az informatikában van ez alól egy hatalmas kivétel: maga az internet, amelynek archívuma bármikor fellelhető – például az *archive.org* weboldalon. Jelen ügyben a hatóságoknál eljáró szakreferens (azaz informatikában jártas, de a szakértői névjegyzékben nem szereplő szakember) a cégek nevén lévő domain címekre keresett rá a vizsgált időszakban, és ott megtalálta az ügyviteli szoftvert. Az ott talált, a vizsgálandó szoftverben is meglévő ötvennégy egyező forráskód fájlnév alapján állította, hogy a projektek megegyeznek. A fájlnevek egyezősége valóban jelezheti, hogy a két projekt forráskódjai között átfedés van, de ez az egyezés mértékét számszerűen lemérni képes módszerként nem elfogadható. A megfelelő módszer megtalálása érdekes kutatási feladat.

## Szakértői szemle

A szakértői vizsgálat céljának eléréséhez rendelkezésemre állt a második projekt működési leírása, a forráskódok, és módomban állt megvizsgálni a még működő rendszert

is. Az első projekt kapcsán csak az adatbázisban elmentett törzsadatok felsorolása volt hozzáférhető. Az előzetes álláspontom az volt, hogy pusztán fájlnevek egyezőségére vagy különbözőségére nem lehet hitelt érdemlő megállapítást megfogalmazni.

A projektek futhatnának teljesen eltérő hardver- vagy szoftverkörnyezetben. Össze is állítottam az előzetes vizsgálati szempontrendszeremet, lásd 1. ábra. Könnyedén magyarázható lenne a projektek különbözősége, ha az egyik például egy natív mobil, a másik pedig egy natív desktop alkalmazás lenne. Természetesen léteznek multiplatform technológiák is, amelyek alkalmazása esetén a vizsgálat még bonyolultabb lett volna. Az eltérő programozási nyelv alkalmazása is egyértelmű jele annak, hogy a projektek nem megegyezőek. Az esettanulmányban vizsgált hardver- és szoftverkörnyezet mindkét projekt esetén ugyanaz volt. A fájlnevek tekintetében – mint a szakreferensi véleményből kiderült – voltak egyezések.




*1. ábra:* A szoftverfejlesztési projektek hasonlóságát alátámasztó tényezők (saját szerkesztés)

Az egyező fájlnevek önmagukban nem tekinthetők bizonyító erejűnek a projektek vizsgálatakor. A fájlokban lévő forráskódot mindenképpen meg kell vizsgálni ahhoz, hogy meg lehessen állapítani az egyezőség mértékét.

A szakértői véleményt úgy kellett elkészítenem, hogy az abban megfogalmazott állítások ellenőrizhetőek és reprodukálhatóak legyenek, de a forráskódfájlokat sem a szakvéleményhez, sem máshová – például e tanulmányhoz – nem csatolhattam, hiszen erre a fejlesztők nem adtak engedélyt. A vizsgálat első lépésenként ezért minden egyes fájlnak előállítottam az MD5 hash kódját (Perlin és Pudipeddi 2016), és azt mentettem el a szakvélemény mellékleteként. Ezáltal később is ellenőrizhető, hogy az analízisre átadott fájl tartalma megváltozott-e a vizsgálatkori állapotához képest.

## Vizsgálat

### *Funkcionális különbségek*

A funkcionalitás hasonlóságának mértékének meghatározásával nem foglalkoztam. Például egy mobiltelefonomon lévő számológép app és egy laptopon futó számoló-

gép alkalmazás funkcióit tekintve közel 100%-ban megegyezik, mégis teljesen különálló alkalmazásnak tekinthetjük azokat.

Korábbi munkáim során (Hornyák és Sáfrány 2009; Hornyák 2019) már foglalkoztam azzal a kérdéssel, hogyan lehet a szoftverprojekteket funkcionális hasonlóságuk alapján csoportosítani, azaz megtalálni, mely projektek formálnak funkcionálisan hasonló egységeket. Az ott kidolgozott Genetic Algorithm Clustering (Hornyák és Sáfrány 2010) erre a vizsgálatra nem alkalmazható.

Informatikai szempontból érdekes vizsgálati perspektíva volt a menütérképek összevetése. A menütérképet a működő rendszerek szemrevételezése során rögzítettem. Abból a feltevésből indultam ki, hogy az eltérő, illetve bővebb funkcionalitás szükségszerűen a felhasználói interfészen is megjelenik, és ennek szembetűnő jele lesz a menük számának a különbözősége.

Megvizsgáltam továbbá, hogy tartalmaz-e a projekt egységteszteket. Ezek – ha jól vannak kidolgozva – jól lefedik a projekt által megvalósított funkciókat. Egyfajta projektdokumentációnak is tekinthetők, és a tesztesetek számából következtetést lehet levonni a projektre vonatkozóan. Sajnos a forráskódok egységteszteket nem tartalmaztak.

### *Strukturális különbségek*

A szakértői vizsgálat során fontos szempont, hogy a megállapításokat konkrét, megfogható, mérhető, ellenőrizhető adatok támasszák alá. A szoftverek vizsgálatánál a szoftver metrikákra (Fenton és Bieman 2014) lehet támaszkodni. Az első metrika, amit lefuttattam, a kódsorok számának meghatározása. Az 1. táblázat celláiban az első helyen az első projekt, az elválasztójel után a második projekt adatai láthatók.

| ***Nyelv*** | ***Fájlok*** | ***Üres sorok*** | ***Megjegyzés sorok száma*** | ***Kód*** |
|---|---|---|---|---|
| JavaScript | 15 / 291 | 3647 / 16510 | 7370 / 24504 | 16941 / 74535 |
| C# | 163 / 686 | 1266 / 7057 | 2003 / 10517 | 5147 / 31370 |
| CSS | 11 / 23 | 1689 / 3132 | 279 / 1169 | 13500 /30521 |
| ASP.NET | 55 / 168 | 187 / 1265 | 0 / 10 | 1079 / 9628 |
| Razor | 1 / 39 | 3 / 547 | 0 / 97 | 13 / 5055 |
| LESS | 25/ 25 | 314 / 314 | 355 / 355 | 2531 / 2531 |
| SQL | 1 / 5 | 71 / 342 | 80 / 396 | 493 / 2403 |
| HTML | 0 / 5 | 0 / 56 | 0 / 8 | 0 / 384 |
| JSON | 0 / 3 | 0 / 3 | 0 / 0 | 0 / 366 |
| XML | 0 / 1 | 0/ 0 | 0 / 0 | 0 / 75 |
| Összesen | 270 /1346 | 6994 / 29226 | 10087 / 37056 | 39704 /156868 |

*1. táblázat:* Sorok száma metrika (saját szerkesztés)

A kódsorok számának vizsgálatakor jól látszik, hogy a második projekt körülbelül négyszer akkora méretű, mint az első. Ez azt jelenti, hogy ha a második projekt teljes egészében azonos is lenne, akkor is legfeljebb 25 % egyezést lehetne kimutatni.

Az is látható a táblázatban, hogy mindkét projekt C#- és ASP.NET-, valamint JavaScript-kódot, a megjelenítéshez CSS- és LESS-technológiát, az adatbázis kezeléshez SQL-t használ.

A LESS-kódoknál különösen nagymértékű az egyezés, de a technológia egyezése miatt a többi fájltípust is további elemzésnek kell alávetni.

A JSON-, XML- és a HTML-fájlok viszont csak a második projektben jelennek meg, de mindösszesen 8 fájlról van szó, az összes fáj mintegy fél százalékáról, ez tehát nem jelent szignifikáns eltérést.

## *A funkciópontanalízis-alapú módszer*

A fájlok darabszámának összevetésénél sokkal pontosabb módszer a funkciópont-elemzésen alapuló eljárás. A módszert eredetileg Albrecht (1979) dolgozta ki. Ennek továbbfejlesztéseként tekinthetünk Symons (1991) munkájára, illetve az International Function Point Users Group – Nemzetközi Funkciópont Felhasználói Csoport – (IFPUG 1994) tevékenységére. A funkciópont-analízis a szoftver termékek funkcionális méretének meghatározására szolgál. Legtöbbször a szoftver költségének (erőforrás-ráfordítás igényének) becslésére szolgál, de a felhasználási területeihez tartozik a karbantartási költségek, a fejlesztési produktivitás meghatározása, funkcionalitás-összehasonlítás, hibasűrűség-mérés, kockázatelemzés.

A korrigálatlan funkciópont (Unadjusted Function Points, UFP) kiszámításának a képlete:

$$UFP = N_i \cdot W_i \cdot N_c \cdot W_c \cdot N_o \cdot W_0$$

ahol:

$N_i$ a bemenettípusú mezők száma,

$W_i$ a bemenet súlya,

$N_o$ a kimenettípusú mezők száma,

$W^o$ a kimenet súlya,

$N_c$ az információfeldolgozó entitások száma,

$W_c$ az információfeldolgozás súlya.

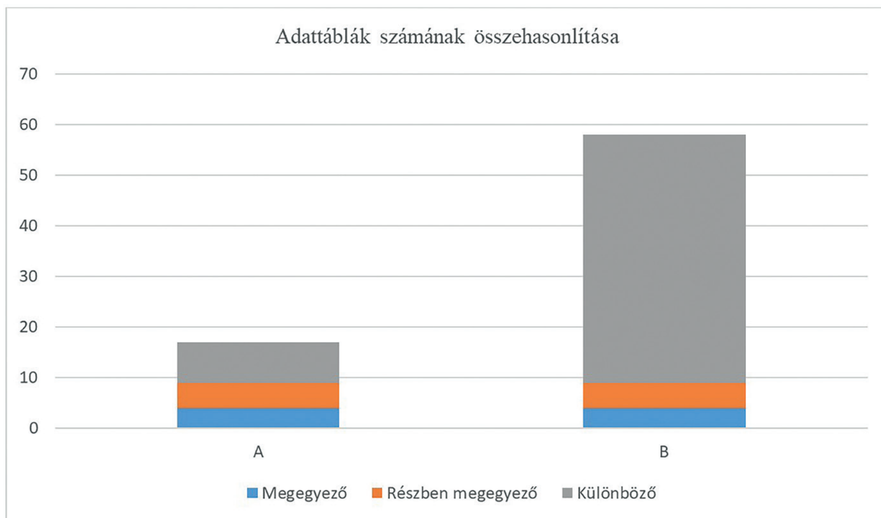
Az alkalmazandó súlyokra példákat találunk Molnár (2003) monográfiájában.

A fájlok darabszámát tekintve akkora volt az eltérés, hogy a funkcióanalízis módszerét nem alkalmaztam.

## *Az adatstruktúrák és adatrekordok összehasonlítása*

A rendelkezésre álló dokumentumok alapján a törzsadatokat össze tudtam vetni egymással. Ezekben nagymértékű volt az átfedés, de a második projekt bővebb adatállományt tudott kezelni.

Az SQL kiterjesztésű fájlok között megtaláltam azt az SQL-szkriptet, amely az adattáblák létrehozását végzi el. Mindkét szkript MS SQL szerver szintaktikát használt. Az első projekt 17 adattáblát hozott létre, a második pedig 58-at. A két projektben négy adattábla volt teljesen megegyező, azaz az adatmezők neve és típusa is azonos volt. Voltak további azonos nevű adattáblák is, amelyekben egymással megegyező nevű és típusú adatmezőket is találtam, de a két adattábla szerkezetileg nem volt azonos, valami módosult bennük. A véletlen egyezőség kizárható volt. Az látható volt, hogy a két projekt adatmodelljének kidolgozásakor az első projekt alapjául szolgált a másodiknak, azonban azt a második projekt átalakította, továbbfejlesztette és kiegészítette. Az adattáblák számát tekintve a második projekt pontosan négyszer akkora méretű volt, mint az első.




*2. ábra:* Adattáblák számának összehasonlítása (saját szerkesztés)

## *Egyező fájlok*

Mivel mindkét projekt forráskódfájl MD5 hashét legeneráltam, ezért egy egyszerű kereséssel meg tudtam határozni, hogy vannak-e teljesen azonos fájlok a projektfájlok között. Összesen 224 darab teljes fájlegyezőséget találtam. Ahogy az 1. táblázat alapján sejthető volt, a .less kiterjesztésű fájlok mindegyike megegyezett a két projektben. A LESS-fájlok (Leaner Style Sheets) a CSS-fájlok (Cascading Style Sheets) kiegészítéseként funkcionálnak. Vannak benne változók, osztályok, operátorok, függvények, névterek – csupa hasznos dolog. A projektekben szereplő LESS-fájlokat megtaláltam a GitHub repozitóriumában, egy adminisztrátorfelület forráskódjának a fájljai voltak. Tehát mindkét projekt ugyanazt a harmadik féltől származó kódot használta. Továbbvizsgálva az egyező fájlokat, ugyanehhez a GitHub repozitóriumhoz tartozó képfájlokat találtam a teljesen megegyező fájlok listájában.

Ezek után – az egyező fájlokra rákeresve – szinte mindegyikről kiderült, hogy megtalálhatók a GitHubon. A projektek fejlesztésénél természetes jelenség az, hogy harmadik fél kódja változtatás nélkül kerül be a projekt fájljai közé.

### *Keretrendszerek vizsgálata*

Mindkét forráskód használta a jQuery JavaScript-könyvtárat. Azonban az egyik verziója 1.2.2., a másiké 1.10.3. volt. A két verzió kibocsájtása között közel 5 és fél év telt el. Ebből is arra következtettem, hogy a második projekt technológiailag hasonló az elsőhöz, de annak továbbfejlesztett változata.

### *A nem teljesen egyező fájlok vizsgálata*

A vizsgálat legnagyobb kihívást jelentő része annak a megállapítása, hogy a két projekt további, nem teljesen azonos fájljai vajon mutatnak-e hasonlóságot, és ha igen, mekkorát. Ne feledjük, a kódsorok száma (azaz a projekt mérete) alapján 25%-nál nagyobb egyezés nem is adódhat, hiszen az első projekt forráskódjainak száma csupán negyede a másodiknak.

Egyetemi oktatóként gyakran előfordult, hogy a hallgatói feladatok beadásakor olyan érzésem támadt: mintha az adott kódot már láttam volna más hallgatói feladat részeként. Éppen ezért, kollégáimmal kidolgoztunk egy olyan környezetet, amelyben a hallgatók számára egyedi programozási feladatokat generálunk, és azokat automatikusan értékeljük ki (Király, Nehéz és Hornyák 2017). Az egyedi feladatkiírás és az automatikus kiértékelés a hallgatót önálló feladatmegoldásra készteti: egyedi feladatra várhatóan egyedi feladatmegoldás érkezik majd. Mindazonáltal a szoftverek hasonlóságának vizsgálata egy másik feladat, amely a tárgyalt szakértői vélemény készítésében is visszaköszönt. A következő fejezetben áttekintem az irodalomban fellelhető algoritmusokat, eszközöket, módszereket.

## Szoftverek hasonlóságát kereső algoritmusok áttekintése

A hasonlóságkereső algoritmusok két osztálya ismert: strukturális összehasonlításon és attribútum-összehasonlításon alapuló algoritmusok (Heon és Murvihill 2015). A struktúrákat összehasonlító algoritmusok tokeneket keresnek az összehasonlítandó kódban. A token a forráskód egy adott része, egy tokenizáló algoritmus állítja elő azt. Két alosztálya ismert az algoritmusoknak: a vektor távolságon alapuló algoritmusok (azaz két tokensorozat közti eltérést valamilyen metrika fejezi ki) és az ujjlenyomaton alapuló algoritmusok (azaz egyedi karakterisztikák jelenlétét vizsgáljuk). Az attribútumalapú összehasonlításkor a forráskód valamilyen profilját állítjuk elő és elemezzük: például az egyedi tokenek számát, a szavak számát, kódsorok átlagos hosszát stb.

Parker és Hamblen (1989) áttekintette, hogy milyen módosítások fordulhatnak elő a forráskódokban, a 3. ábra ezeket mutatja be. Az ábra közepén a teljesen válto-

zatlan kód áll, ezt veszi körbe az átdolgozott üzleti logika. A módosítás mértékének és a felismerésük bonyolultságának növekvő sorrendjében a lehetséges változtatások:

- nincs változtatás,
- csak megjegyzések / üres sorok változtak,
- azonosítók változtak,
- változók definiálásának helyzete változott,
- az eljárások változtak,
- az utasítások változtak,
- átdolgozott üzleti logika.

*3. ábra:* A lehetséges kódmódosítások (Parker and Hamblen 1989 alapján)

A modern szoftverfejlesztésben komoly szerepe van a refaktorálásnak (Fowler 2018), és a fejlesztői környezetek nagy mértékben támogatják is azt. Egészen az L4 szintig a program átalakítása csupán formális refaktorálással elvégezhető.

L1 szint: A megjegyzéseket a fordító vagy interpreter figyelmen kívül hagyja, azok a kód futását amúgy sem befolyásolják. Az L2 szinthez tartozik például az azonosítók átnevezése. Az L3 szintre jó példa a változók deklarálásának áthelyezése a függvények törzsébe, vagy kiszervezése egy header fájlba. Az L4 szinthez tartozik például az eljárások kiemelése, áthelyezése. Az L5 szinten található az utasítások megváltoztatása: például egy *if-then-else* helyett a *feltétel?kód1:kód2* operátor használata, vagy az i = i +1 helyett az i++ használata. Ezek már nehezebben felismerhető változások. Az üzleti logika megváltoztatása a legmagasabb, L6 szinten van, például sorba rendező algoritmusok kicserélése esetén az egyezőség felismerése szinte lehetetlen feladat.

Az informatika korai szakaszában bevezetett metrika (Halstead 1977) alapján lehetett megbecsülni egy szoftver volumenét ($V$) és az kidolgozásához szükséges erőforrásokat ($E$).

$n_1$ = egyedi operátorok száma,

$n_2$ = egyedi operandusok száma,
$N_1$ = összes operátorok száma,
$N_2$ = összes operandusok száma.

Ezekből kiszámolható:

$V=(N_1 + N_2) \log_2 (n_1 + n_2)$ *és*

$E=(n_1 N_2 (N_1 + N_2) \log2 (n_1 + n_2))/2n_2$.

A szoftver plagizálását ezek a mérőszámok egyértelműen azonban nem jelzik, de a keresést orientálhatják olyan módon, hogy a hasonló forráskódokra a kiszámolt mérőszámok is hasonlóak.

Attribútum halmazokat vizsgált Faidhi és Robinson (1987). Ezek:

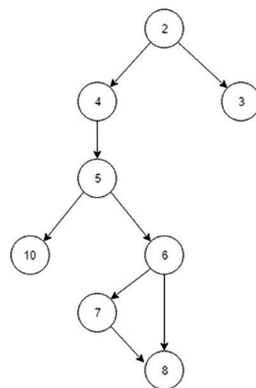
$m_1$ = programsorok hossza karakterekben,
$m_2$ = megjegyzések száma,
$m_3$ = behúzások száma,
$m_4$ = üres sorok száma.
$m_5$ = függvények átlagos hossza,
$m_6$ = kulcsszavak száma,
$m_7$ = átlagos változó hossz,
$m_8$ = sorok átlagos szóköz száma,
$m_9$ = címkék és goto utasítások száma,
$m_{10}$ = azonosítók változékonysága,
$m_{11}$= a vezérlésfolyam gráf régióinak száma,
$m_{12}$-$m_{14}$ = a vezérlésfolyam gráf egyes elemeinek a száma,
$m_{15}$-$m_{18}$ = a különböző típusú kifejezések aránya az összes kifejezés számához képest,
$m_{19}$ = a program puritánságának mérőszáma: üres utasítások száma, felesleges blokkszervező utasítások száma, felesleges zárójelek száma, deklarált, de nem használt változók száma,
$m_{20}$ = program modulok kódhosszának aránya a teljes kódhoz,
$m_{21}$ = program modulok száma,
$m_{22}$ = feltételes utasítások számának aránya az összes utasításhoz,
$m_{23}$ = ciklusképző utasítások számának aránya az összes utasításhoz,
$m_{24}$ = az utasítások száma.

Azt gondolom, hogy ezeknek a mérőszámoknak a kiszámítása mindkét projektre csupán jelezheti a hasonlóságot, de a plagizálás tényét ez sem mutatja ki megcáfolhatatlan módon.

## *Vezérlésfolyamgráf-alapú módszerek*

A vezérlésfolyam-gráf (Muchnick és Jones 1991) a program futásának egy grafikus reprezentációja. A csomópontokban utasítások vagy kód blokkok vannak, a lefutás menetét pedig a csomópontokat összekötő irányított görbék jelzik, a nyíl irányát követve látható az utasítások sorrendje. A következő ábra egy példát mutat arra, hogy egy N elemű egészekből álló tömb legnagyobb elemét hogyan tudjuk megkeresni.

```
1  int max (int vector[], int n) {
 2        if (n > MAX_ITEM_COUNT)
 3              return ERROR;
 4        int max = INT_MIN, i = 0;
 5        while (i < n) {
 6              if (vector[i] > max)
 7                    max = vector[i];
 8              i = i + 1;
 9              }
10        return max;
11        }
```




*4. ábra:* Legnagyobb elem keresésének vezérlésfolyam-gráfja (saját szerkesztés)

Ismert olyan eljárás (Yuan et al. 2018), amely a vezérlésfolyam-gráfok összevetésén alapul. A módszernek a következő hátrányai vannak:

A kódot le kell tudni futtatni. Ehhez szükséges a megfelelő szoftverfejlesztői licenc, valamint a szoftver lefordításához, telepítéséhez a – megfelelő verziójú – függőségek telepítése.

- A kódot módosítani kell, hogy a vezérlésfolyam-gráfot fel lehessen építeni: log bejegyzések szükségesek a gráf előállításához.
- Ha fel tudtuk építeni mindkét program vezérlésfolyam gráfját, akkor a keresési feladat matematikai értelemben izomorf gráfkeresési probléma, ami önmagában N-P komplett feladat. Matematikailag megfogalmazva: legyen *p1* az első program vezérlésfolyam-gráfja, *p2* a második program vezérlésfolyam-gráfja; keressük azt, hogy a *p1* és *p2* gráf között van-e bijektív struktúratartó leképzés.
- Részleges egyezés keresése esetén a matematikai probléma még összetettebb: keressük a *p1* gráf részgráfjai között azokat, amelyek a *p2* gráf részgráfjaival izomorfak.

Könnyen belátható, hogy ezt a módszer jelen vizsgálathoz nem tudtam alkalmazni.

## Anyajegyalapú módszerek

Miles és Colberg (2003) bevezette a szoftver „anyajegy” (birthmark) fogalmát. A definíciójuk szerint a dinamikus anyajegy *p* és *q* programokat vizsgálja.

Legyen *I* valamely inputja a programnak.

Jelölje ≡cp a másolás relációt.

Az *f(p, I)* a *p* program valamely attribútumainak halmaza.

Ezt definíció szerint dinamikus anyajegynek tekintjük, ha

*f(p, I)* csak a *p* programtól és az *I* inputtól függ, mástól nem:

$$p \equiv_{cp} q \therefore f(p, I) = f(q, I).$$

Ha a vizsgálatból kihagyjuk az *I* inputot, és az egyezőséget anélkül vizsgáljuk, akkor statikus anyajegy vizsgálatról beszélünk.

Könnyen belátható, hogy ha a *p* program egyik transzformáltja *p’*, akkor az anyajegyek megegyeznek:

$$f(p, I) = f(p', I).$$

A jól megválasztott anyajegyek azt is garantálják, hogy ha egy *p* és *q* program specifikációja azonos, de a programok külön-külön kerültek kifejlesztésre, akkor

$$f(p, I) \neq f(p', I).$$

Két különböző, az API-hívásokon alapuló anyajegyet definiálhatunk:

1. API-hívások sorrendje. A Windows operációs rendszeren futó programok API-hívásokat hajtanak végre. Ezek sorrendje egyedi. Ha a sorrend megváltozik, akkor az a funkcionalitás változását is jelzi. Bizonyos programok csak néhány fajta API-t használnak, de még ebben az esetben is különböző a hívások sorrendje. Formálisan megfogalmazva: legyen *p* a vizsgált program, amely *I* inputot fogad. Legyen *W* a vizsgálatba bevont API-hívások halmaza. Az EXESEQ(*p, I*) anyajegy azt a (*w1, w2, …wn*) API hívássorozatot jelenti, ahol $wi \in W$, $1 < i < n$. Azaz a vizsgálatba nem bevont API-hívásokat nem vesszük figyelembe.
2. Az API-hívások frekvenciája. Egy program szemantikus átírása esetén az API-hívások sorrendjének megváltoztatása mellett az API-hívások száma változatlan maradhat, de ezek frekvenciája jól jellemzi a programot. Formálisan leírva: legyen *p* a vizsgált program, amely *I* inputot fogad. Jelentse *ki* a *wi* az API-hívás függvényének a nevét, és *ai* a *ki* előfordulásainak számát *EXESEQ*(*p, I*)-ben. Az anyajegy jelölése EXEFREQ(*p, I*) és a *((k1, a1), (k2, a2), …(kn, an))* sorrendet értjük alatta.

A módszer használatához az vizsgálatba bevont hívásokat kellene begyűjteni és elemezni, amihez az operációs rendszer nem ad hozzáférést. Az egyik ismert módszer (Tamada et al. 2004) parazita DLL-fájlokat használ. A függvényhívási pointer tábla átírásával a hívást a parazita DLL-fájlokba irányítják. Ezek az anyajegy adatokat lementik, majd továbbítják a hívást az eredeti DLL-fájlok felé. A módszer hátránya, hogy:

- a lefordított kódot kell futtatni;
- el kell készíteni a parazita DLL-fájlokat;
- csak Windows operációs rendszeren működik;
- az EXESEQ-anyajegynél a részleges egyezés is vizsgálható, de az EXEFREQ esetén nem;
- víruskereső szoftverek blokkolhatják.

A módszer előnye, hogy az automatikus programtranszformációval (például a forráskód összezavarásával, nehezen olvashatóvá tételével) szemben robusztus. Az API-hívások egy része lecserélhető, így a forráskód módosításával az anyajegyek megváltoznak. Vannak azonban olyan alapvető API-hívások, amelyek nem megkerülhetők. Ha találunk ilyen API-híváshalmazt, akkor azok egyértelmű és egyedi lenyomatát adnák a kód futásának.

További anyajegytípusokat mutat be és kategorizálja az anyajegyeket Zeng et al. (2012):

- Utasításalapú anyajegyek. Az utasítások sorrendje jól tükrözi a program végrehajtását, így az ezen alapuló anyajegyek használatának van létjogosultsága. Erről a típusú hasonlóságkereséséről már tárgyaltunk.
- API-alapú anyajegyek. A szabványos API-könyvtár hívásainak vizsgálatán ala-

puló módszer. A megvalósítása nagyon komplex.
- Gráfalapú anyajegyek. A szoftvertervezés, a szoftverfejlesztés, az algoritmusok dokumentálása többfajta gráftípust ismer és használ. A vezérlésfolyam-gráfot már ismertettük, de létezik függvényhívási gráf, függőségi gráfok, öröklődési gráfok. Ezekre is hasonló megállapítások vonatkoznak, mint amit a vezérlésfolyam-gráfnál tettünk.

### *A futtatható állomány összevetésén alapuló módszerek*

Az összehasonlítási módszerek egy külön csoportja a bináris állományok összehasonlításán alapul. Ennek a vizsgálatnak a forrása a gépi kódú program, amelyet vissza kell fejteni assembly nyelvű programmá. Erre ismert algoritmusok vannak, lásd (Vigna 2007), a két legfontosabb: a Linear sweep (instrukciók lineáris feldolgozása az elsőtől az utolsóig) és a Recursive Traversal (rekurzív bejárás a hívások sorrendjében). Java és C++ nyelvek visszafordítását tárgyalja (Vinciguerra et al. 2003), a futás időben, dinamikusan interpretált nyelvek visszafejtésével foglalkozik (Bernstein 2018).

Az assembly nyelv röviden az alábbi formalizmussal írható fel:

assembly = {cím; opcode; operandus1; ... ; operandusn}

A teljes assembly nyelv leírása a Backus–Naur formalizmus alapján pedig (Matveev 2014) alapján:

```
program ::= line [„\n” program]
line ::= „” | label | cmd [arguments]
arguments ::= argument [„,” argument]
label ::= string „:”
cmd ::= *VM’s command mnemonic*
arguments ::= argument [„,” arguments]
argument ::= immutable | variable | address | register
immutable ::= „$” („0b” bdigit {bdigit} | „0o” odigit {odigit}
          | [„0d”] digit {digit} | „0x” xdigit {xdigit} | „0c” (ascii | escape))
variable ::= string
address ::= „$” string
register ::= „%” (digit | „a” | „b” | „AC” | „BP” | „SP” | „PC”)
string ::= char {char}
char ::= letter | digit | „_”
letter ::= „a” | ... | „z” | „A” | ... | „Z”
xdigit ::= digit | „a” | ... | „f”
digit ::= „0” | ... | „9”
odigit ::= „0” | ... | „7”
bdigit ::= „0” | „1”
escape ::= „\” („s” | „n” | „t” | „v” | „b” | „r” | „f” | „a” | „\” | „0”)
ascii ::= *ASCII-symbol*
```

A program lefutásának menetét az assembler kód alapján azért nehéz elemezni, mert az utasítást nehéz elválasztani az adattól. További nehézséget jelent, hogy a kódot gyakran összezavarják (obfuszkálják) – ezeket tárgyalja Linn és Debray (2003).

A zavaró tényezők közé tartoznak még a fordítóprogram különböző kapcsolói is, amellyel valamilyen platformra lehet optimalizálni a kódot. Ezek teljesen megegyező forráskód esetén is különböző bináris futtatható állományt állítanak elő. Ennek a hatását vizsgálta (Yuan et al. 2018). Kimutatták számszerűleg, mekkora mértékű változtatást eredményez csak a fordítási folyamat önmagában, azaz ugyanazon a kód fordításakor más-más fordítási opciók mellett mekkora lesz az assembler kódok különbsége. Cesare és Yang (2012) cikkében a bináris fájlok dekompilálásakor a következő feladatokat fogalmazza meg:

- pointerek analízise,
- feltételek kiküszöbölése,
- változók rekonstrukciója,
- globális változók felismerése,
- eljárások paramétereinek rekonstrukciója,
- vezérlésfolyam struktúrák rekonstrukciója,
- típusok rekonstrukciója.

## Az egyes módszerek összehasonlítása

A kódsorok számán alapuló metrika csak az azonos programozási nyelven kidolgozott fájlok összehasonlítására használható, a pontatlansága nagy, de a módszer gyors és egyszerű. A nagyon eltérő fájl-, és kódsorszámok azonban jól jelzik a projektek különbözőségét.

A funkciópont analízisen alapuló technika eltérő programozási nyelvekre is alkalmazható. A súlyok megfelelő meghatározása nagyobb tapasztalatot igényel. Kevésbé könnyű kijátszani, mint a kódsorok számának meghatározását. Az egyező fájlok keresése esetén minden fájlt, minden fájllal össze kell hasonlítani tartalom szempontjából. A keresést felgyorsíthatja egy a fájlra jellemző hash kód generálása (MD5 hash, vagy az SHA-algoritmusok egyike). Azonban ez a lehető legkönnyebben becsapható módszer. A generált MD5 hash-eket egy szóköz, egy megmásított sorvége jel könnyedén megváltoztatja. A szándékolt megtévesztés ellen nem használható. A részleges egyezés (például továbbfejlesztés) ezzel a módszerrel nem kimutatható.

Az adatbázisok struktúrájának hasonlóságát jelen esettanulmányhoz jól lehetett vizsgálni, általánosan azonban elmondható, hogy a relációs adatbázisok nagy mozgásteret adnak a struktúrák szándékolt megváltoztatásának. Megfelelő tapasztalat birtokában a módszer könnyen kijátszható: adatmezők új táblába szervezhetők, és külső kulcsokkal az eredeti táblákhoz kapcsolhatók, adattípusok részben megváltoztathatók, stb. Az egyezőség vizsgálat a táblákat létrehozó SQL-utasítások forráskódján elvégezhető. Jelen esettanulmány nem tárt fel általánosítható megoldást a feladatra, és a szakirodalomban sem sikerült alkalmazható módszert találni erre.

Kifejtettem olyan módszereket is, amelyeket végül a konkrét vizsgálathoz nem tudtam felhasználni. A vezérlésfolyam-alapú módszer hátránya, hogy képesnek kell lenni lefordítani a projekteket, amelyhez megfelelő fejlesztőkörnyezet, licenszek és dokumentáció szükséges. A módszer pontos, de a vizsgálat nagyon időigényes, kijátszásához az algoritmusokat teljesen újra kell tervezni, és ez gyakorlatilag a szoftver

újraírásának, önálló szellemi terméknek tekinthető. Az anyajegyalapú módszerek csak a Windows operációs rendszeren futó alkalmazások esetében működnek, megfelelően robosztusak. Az elméleti megközelítés figyelemreméltó, de gyakorlati alkalmazhatóságát meg lehet kérdőjelezni. A futtatható állomány vizsgálatán alapuló módszerek könnyedén kijátszhatóak, a szándékos összezavarás ellen nem védenek, gyakorlati jelentőségük akkor van, ha semmi más nem áll rendelkezésre, csak a futtatható bináris állomány.

## Összefoglalás

Ez az esettanulmány a szoftverek forráskódjának plagizációját, engedély nélküli felhasználását, illetve ennek kimutathatóságát vizsgálta. A felületes vizsgálat, például az egyező forráskód-fájlnevek felismerése önmagában alkalmatlan arra, hogy bizonyító erejű legyen két szoftver azonosságát illetően. Ehhez mindenképpen szükséges a forráskódok tartalmának vizsgálata. Kijelenthető, hogy a szoftverek forráskódjának analízisével a szoftverek hasonlósága megállapítható.

A jelen tanulmányban vizsgált esetben mind a forráskódsorok száma, mind az adatbázisok tábláinak mérete négyszeres mértékben tért el. Megállapítható volt, hogy az egyik projekt a másik projekt jelentős mértékű továbbfejlesztése. A felhasznált third party könyvtárak között is nagy verziókülönbség volt, amely szintén ezt támasztotta alá. A megrendelő kérdésére egyértelmű választ lehetett adni: a szakreferens megállapítása téves volt, a két ügyviteli rendszer olyan mértékben eltért egymástól, hogy a későbbi rendszert is önálló szellemi alkotásnak lehet tekinteni.

### Irodalom


Albrecht, Allan J. „Measuring application development productivity." In *Proceedings. of IBM Application Development Joint SHARE/GUIDE Symposium*, 83–92. Monterey, CA: 1979.

Bernstein, Rocky. „Decompilation at Runtime and the Design of a Decompiler for Dynamic Interpreted Languages." Utolsó hozzáférés: 2022. június 09. https://pdfs.semanticscholar.org/9dc4/022f12152c913990e403d2e3fef267662a8c.pdf

Cesare, Silvio és Yang Xiang. *Software similarity and classification.* London: Springer Science & Business Media, 2012. https://doi.org/10.1007/978-1-4471-2909-7

Faidhi, Jinan A. W. és Stuart K. Robinson. „An empirical approach for detecting program similarity and plagiarism within a university programming environment." *Computers & Education* 11, no. 1 (1987): 11–19. https://doi.org/10.1016/0360-1315(87)90042-X

Fenton, Norman és James Bieman. *Software metrics: a rigorous and practical approach.* Boca Raton, FL: CRC Press, 2014.

Fowler, Martin. *Refactoring: improving the design of existing code.* Boston: Addison-Wesley Professional, 2018.

Github, Matveev, Oleg. „Assembler:: Extended Backus Naur Form." Utolsó hozzáférés 2022. június 10.
https://github.com/Lincor/VM/wiki/Assembler::-Extended-Backus-Naur-Form

Halstead, Maurice Howard. *Elements of software science*. New York: Elsevier, 1977.
https://dl.acm.org/doi/10.5555/540137

Heon, Matthew és Dolan Murvihill. *Program similarity detection with checksims*. Worcester: Worcester Polytechnic Institute, 2015.
https://core.ac.uk/download/pdf/212993801.pdf

Hornyák Olivér és Sáfrány Gábor. „Group technology for automated generation of machine controller code." In *5th International Symposium on Applied Computational Intelligence and Informatics*, 17–21. Timişoara, Romania: IEEE, 2009.
https://doi.org/10.1109/SACI.2009.5136234

Hornyák Olivér és Sáfrány Gábor. „Models and methods to detect similarity of Manufacturing machines." *Hungarian Journal of Industry and Chemistry* 38, no. 2 (2010): 149–153.

Hornyák Olivér. „Grouping and analyzing PLC source code for smart manufacturing." In *Solutions for Sustainable Development: Proceedings of the 1st International Conference on Engineering Solutions for Sustainable Development (ICESSD 2019)*, Miskolc: CRC Press, 2019.
https://doi.org/10.1201/9780367824037

Király Sándor, Nehéz Károly és Hornyák Olivér. „Some aspects of grading Java code submissions in MOOCs." *Research in Learning Technology* no. 25 (2017).
https://doi.org/10.25304/rlt.v25.1945

Linn, Cullen és Debray Saumya. „Obfuscation of executable code to improve resistance to static disassembly." In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, 290–299. New York, NY: Association for Computing Machinery, 2003.
https://doi.org/10.1145/948109.948149

Marthaler, Valerie és Michigan Clarkston (eds). *IFPUG. Function Point Counting Practices Manual, Release 4.2.1*. Princeton Junction: International Function Point Research Group, 2005.
https://epmc2.monsite-orange.fr/file/d6ab0a1755c60de1840c1337f50b64d2.pdf

Molnár Bálint. *Funkciópont elemzés a gyakorlatban.* Budapest: MTA Információtechnológiai Alapítvány, 2003.
https://doi.org/10.13140/RG.2.2.12580.68484

Muchnick, Steven S. és Neil D. Jones. *Program flow analysis: Theory and applications*. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

Myles, Ginger és Christian S. Collberg. „Detecting Software Theft via Whole Program Path Birthmarks." In *Information Security*, 404–415. Berlin: Springer, 2004.
https://doi.org/10.1007/978-3-540-30144-8_34

Parker, Alan és James O. Hamblen. „Computer algorithms for plagiarism detection." *IEEE Transactions on Education* 32, no. 2 (1989): 94–99.
https://doi.org/10.1109/13.28038

Perlin, Eric C. és Ravisankar V. Pudipeddi. *„Efficient file hash identifier computation.*" U.S. Patent No. 9424266. 2007.
https://patents.google.com/patent/US9424266

Symons, Charles R. *Software sizing and estimating: Mk II FPA (Function Point Analysis)*. Hoboken, New Jersey: John Wiley & Sons Inc., 1991.

Tamada, Haruaki, Keiji Okamoto, Masahide Nakamura, Akito Monden és Ken-Ichi Matsumoto. „Dynamic software birthmarks to detect the theft of windows applications.” In *International Symposium on Future Software Technology (ISFST 2004)*, 1–6. Xian,China: ISFST 2004 Program Committee, 2004.

Vinciguerra, Lori, Linda Wills, Nidhi Kejriwal, Paul Martino és Ralp Vinciguerra. „An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java.” In *10th Working Conference on Reverse Engineering (WCRE) 2003 Proceedings*, 2003.
https://doi.org/10.1109/WCRE.2003.1287233

Vigna, Giovanni. „Static Disassembly and Code Analysis.” In Christodorescu, Mihai, Somesh Jha, Douglas Maughan, Dawn Song és Cliff Wang (Szerkesztők). *Malware Detection. Advances in Information Security*, 19–43. New York, NY: Springer, 2007.
https://doi.org/10.1007/978-0-387-44599-1_2

Yuan, Baoguo, Junfeng Wang, Zhiyang Fang és Li Qi. „A New Software Birthmark based on Weight Sequences of Dynamic Control Flow Graph for Plagiarism Detection.” *The Computer Journal* 61, no. 8 (2018): 1202–1215.
http://dx.doi.org/10.1155/2015/579390

Zeng, Ying, FenLin Liu, Luo XianYang és Lian ShiGuo. „Abstract interpretation-based semantic framework for software birthmark.” *Computers & Security* 31, no. 4 (2012): 377–390. https://doi.org/10.1016/j.cose.2012.03.004