Multidiszciplináris tudományok, 10. kötet. (2020) 2 sz. pp.105-118 **https://doi.org/10.35925/j.multi.2020.2.14**

Automatikus vizuális tesztelés HTML oldalakhoz

Szilvási Attila
hallgató, Miskolci Egyetem, Informatikai Intézet
3515 Miskolc, Miskolc-Egyetemváros, e-mail: szilvsiattila6@gmail.com

Szacsuri Norbert
hallgató, Miskolci Egyetem, Informatikai Intézet
3515 Miskolc, Miskolc-Egyetemváros, e-mail: szacskesz@gmail.com

Szöllősi Dániel
hallgató, Miskolci Egyetem, Informatikai Intézet
3515 Miskolc, Miskolc-Egyetemváros, e-mail: szollosi.daniel0@gmail.com

Szőke Attila
hallgató, Miskolci Egyetem, Informatikai Intézet
3515 Miskolc, Miskolc-Egyetemváros, e-mail: clownface68@gmail.com

Sólyom Tamás
hallgató, Miskolci Egyetem, Informatikai Intézet
3515 Miskolc, Miskolc-Egyetemváros, e-mail: tsolyom@freemail.hu

Kiss Áron
hallgató, Miskolci Egyetem, Informatikai Intézet
3515 Miskolc, Miskolc-Egyetemváros, e-mail: aron.kiss01@gmail.com

Hornyák Olivér
egyetemi docens, Miskolci Egyetem, Informatikai Intézet
Cím: 3515 Miskolc, Miskolc-Egyetemváros, e-mail:oliver.hornyak@iit.uni-miskolc.hu

Absztrakt

A vizuális tesztelés webes alkalmazások körében egyre szükségszerűbb a változó igények és a változatos platformok miatt. Ennek egyik módja a bemutatott automatizált regressziós tesztelés. Ehhez meg kell értenünk a vizuális tesztelés mikéntjét. Hagyományosan, manuális módszerekkel végezve különböző böngészőben és platformokon vizsgáljuk alkalmazásunk megjelenését és hasonlítjuk össze a megjelenített tartalmat. Ez azonban így rengeteg monoton munkavégzést jelent, illetve magában hordozza az emberi tévesztés lehetőségét, amely rontja a megbízhatóságot és a konzisztenciát. További hátrány a lassúság. Ha figyelembe vesszük, hogy egy-egy webes alkalmazás frontendje akár hetente kaphat frissítést, a manuális tesztelés nem lehet megoldás. Így ezen a területen sokat nyerhetünk az automatizálással, hiszen az egyes böngészők eredményének összehasonlítása konzisztens lesz, mentes az emberi hibáktól. A tesztkörnyezet konfigurálása sok időt vehet igénybe, de ára többszörösen megtérülhet. Azonban ezen eszközök általi tesztelés is órákat vehet igénybe, hiszen minden

böngészőben meg kell nyitni az alkalmazást és képet kell készíteni róla, mely időigénye az alkalmazás méretével arányosan nő. Továbbá a dinamikus tartalmakkal rendelkező oldalak – híroldalak – esetén fals hibákat kaphatunk, így ezen esetekben további konfigurációs beállításokra van szükség.

Kulcsszavak: *vizuális regresszió tesztelés, szoftverfejlesztés*

Abstract

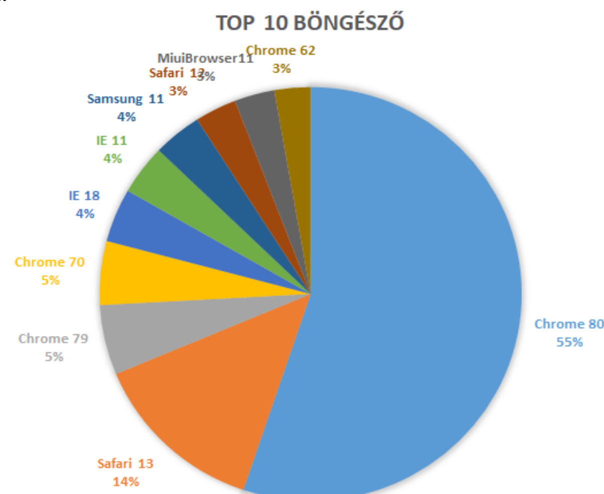
Web applications displayed on heterogeneous platforms and have a frequently changing life cycle. There is a high demand for their visual testing. This paper presents an approach for automated visual regression testing. The basic concepts of visual testing are discussed. The traditional, manual testing checks the displayed content of various browsers and platforms. This is a huge amount of monotonous work, which often results in human mistakes thus reducing reliability and consistency. Another disadvantage of the manual testing is that it is slow. Certain web applications have weekly updates, manual testing is not a feasible option for them. Automation would gain us consistency and be free from human errors. Obviously, the configuration of the test environment takes some effort but that could return on long term. The paper investigates testing time and reliability of automated visual regression testing in a selected environment.

Keywords: *visual regression testing, software engineering*

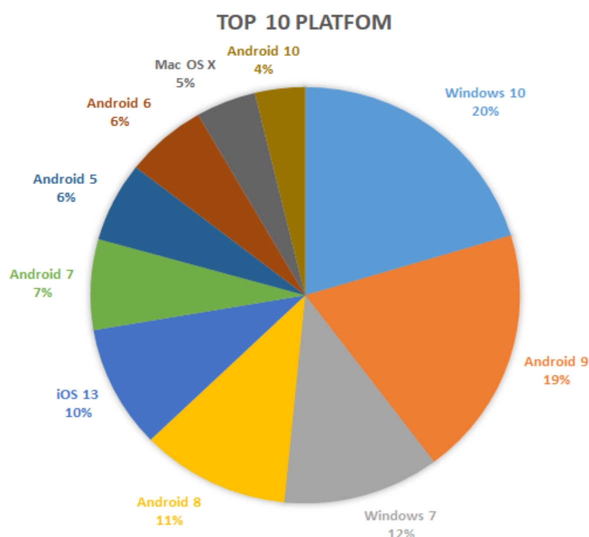
1. Bevezetés

A vizuális tesztelés azt vizsgálja, hogy az alkalmazás a kívánt módon jelenik-e meg a felhasználó számára. A mai világban, a HTML világában a webfejlesztők különböző operációs rendszereken, különböző böngészőkben megjelenített tartalmakat fejlesztenek ki. [1] rendszeresen közöl statisztikákat a böngészők elterjedtségéről. A cikk írásának idején a böngészők piaci részesedése: Chrome 59.3%, Safari 12.3%, Internet Explorer & Edge 9.1%, Firefox 4.5%, Opera 3.0%.

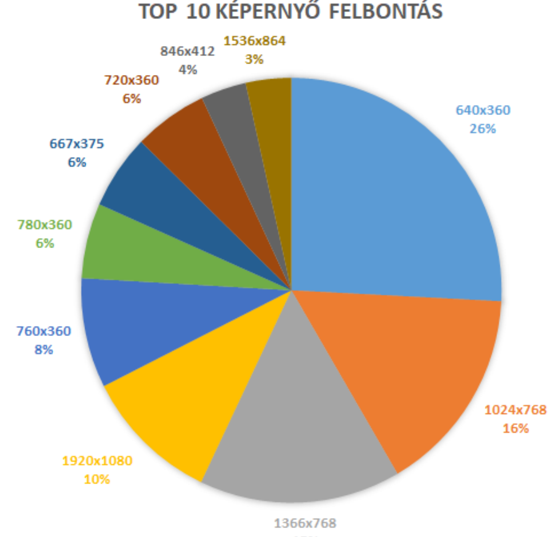
Általános szoftverfejlesztői gyakorlat szerint a legalább 5 %-os részesedést elérő böngészők támogatása elvárt. A következő ábrák (1. ábra., 2. ábra., 3. ábra.) a platformok piaci eloszlását mutatják [2] adatai alapján:



1. ábra. *A 10 legnépszerűbb böngésző.*



2. ábra. *A 10 legnépszerűbb operációs rendszer.*



3. ábra. *A 10 legnépszerűbb képernyőfelbontás.*

A HTML és a CSS szabványok segítik a frontend fejlesztőket abban, hogy az általuk fejlesztett szoftver ugyanazzal a kóddal futtatva bárhol működő legyen. A szoftveres minőség biztosításban az ide vonatkozó a faktort nevezik hordozhatóságnak.

4. ábra. *Szoftveres minőségi jellemzők.*

A 4. ábrán látható szoftveres minőségi jellemzőket az alábbiakban foglalhatjuk össze [6]:

a) Funkcionalitás

Működési funkciókkal és azok meghatározott tulajdonságaival összefüggő attribútumok halmaza. A működési funkciók megadott vagy értelemszerű igényeket elégítenek ki. A funkcionalitás minőségjellemző a következő részjellemzőkre bontható:

- Célnak való megfelelés;
- Helyes működés;
- Más rendszerekkel való együttműködési képesség;
- Illeszkedés szakterületi szabványokhoz, megállapodásokhoz;
- Védelem a jogosulatlan (véletlen vagy szándékos) hozzáférések ellen.

b) Megbízhatóság

A szoftver azon attribútumainak halmaza, amelyek utalnak a szoftver adott feltételek mellett, adott időtartamon keresztül fenntartható, rendeltetésének megfelelő működési szintjére.

A szoftver más termékekkel ellentétben nem kopik, nem öregszik. A megbízhatóság korlátait a mindenkori követelmények, a tervezés és az implementáció hiányosságai és nem az eltelt idő szabja meg. A megbízhatóság minőségjellemző például a következő részjellemzőkre bontható:

- Érettség, azaz hiba-előfordulás gyakoriság;
- Hibatűrő képesség;
- Helyreállíthatóság.

c) Használhatóság

A szoftver használatához szükséges erőfeszítésekkel, valamint a megadott vagy értelemszerűen számbavehető felhasználók egyéni megítélésével kapcsolatos attribútumok halmaza.

A felhasználók lehetnek interaktív szoftver felhasználók, de lehetnek operátorok, végfelhasználók és olyan közvetett felhasználók is, akikre a szoftver bármilyen hatással van, vagy függnek tőle. A használhatóság minőségjellemző például a következő részjellemzőkre bontható:

- Megérthetőség;
- Megtanulhatóság;
- Működtethetőség.

d) Hatékonyság

A szoftver teljesítményszintje és a felhasznált erőforrások mennyisége közötti kapcsolatra vonatkozó attribútumok halmaza.

Az erőforrások közé tartozik minden más szoftvertermék, hardver, egyéb anyagok (papír, lemezek), és a működtető-karbantartó személyzet szolgáltatásai. A hatékonyság minőségjellemző például a következő részjellemzőkre bontható:

- Válasz- és végrehajtási idők;
- Erőforrás-kihasználás.

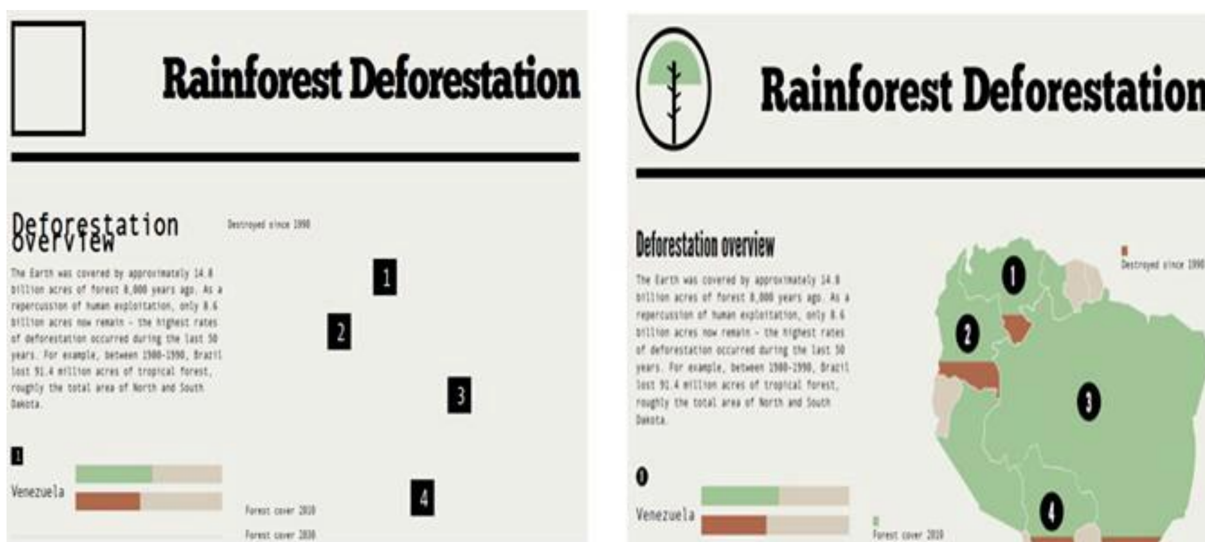
e) Karbantarthatóság

A módosításokhoz szükséges erőfeszítéseket jellemző attribútumok halmaza. A módosítások közé tartozik a hiba-kijavítás, a továbbfejlesztés, a szoftver környezeti változásokhoz és változó követelményekhez történő igazítása. A karbantarthatóság minőségjellemző például a következő részjellemzőkre bontható:

- Hibafeltáráshoz szükséges erőfeszítés;
- Módosításhoz, hibajavításhoz szükséges erőfeszítés;
- Stabilitás, azaz a módosítások mellékhatásainak kockázata;
- Tesztelhetőség.

f) Hordozhatóság
A szoftvernek egyik környezetből a másikba történő átviteli lehetőségével kapcsolatos attribútumok halmaza. A környezet jelenthet szervezeti-, hardver- vagy szoftverkörnyezetet. A hordozhatóság minőségjellemző például a következő részjellemzőkre bontható:
- Adaptálhatóság;
- Telepíthetőség;
- Illeszkedés hordozhatósági szabványokhoz és megállapodásokhoz;
- Más szoftver helyettesítésének képessége.

A minőségbiztosításra (Quality Assurance, QA) hárul az a feladat, hogy a megfelelő megjelenést ellenőrizze, azaz az összes lehetséges kimeneti kombinációt kell vizsgálni a vizuális hibák ellenőrzése érdekében. Az 5. ábra ugyanazt a weboldalt mutatja Internet Explorer 8 és Chrome böngészőben:



5. ábra. *Ugyanaz a weboldal IE8 (balra) és Chrome (jobbra) böngészőben. Forrás: [3]*

2. A vizuális teszt szükségessége

Gondoljunk csak arra, ha az oldalunkon megjelenő hirdetés valamilyen módon nem jól kivehető. Ebben az esetben ezek a hibák komoly anyagi vonzattal járhatnak a hirdető cégek felé. Tehát ezek a megjelenítési hibák elsősorban komoly anyagi hátrányt jelentő hibák, és csak másodsorban kozmetikai, kinézeti hibák.

Ezekkel az érvekkel a vezetőség számára is világos, hogy a tesztelésre szánt pénz igenis kifizetődő lehet.

Vagy tekintsünk egy olyan konvertert, amely az egyik formátumú képet egy másikká alakítja át.

A vizuális hibák a legnagyobb cégeknél is előfordulhatnak. A szoftvereken leggyakrabban funkcionális tesztek százait végzik el, felmerülhet tehát a kérdés, hogy akkor mégis miért fordulnak elő vizuális hibák a szoftverben.

A válasz az, hogy a funkcionális tesztek nem alkalmasak a vizuális hibák kimutatására, tesztelésére.

3. Miért nem tudja a funkcionális teszt ellenőrizni a vizuális hibákat?

Természetesen megoldható, olyan tesztek írásával, amely komponensenként ellenőrzi annak méretét, elhelyezkedését, színét, viselkedését stb., viszont ez a tesztek olyan méretű felfúvódásával járna, amely később az átláthatóságot, használhatóságot veszélyeztetné.

Az alábbi Javascript függvény igaz vagy hamis értékkel tér vissza, annak függvényében, hogy a paraméterként adott HTML elem látszik-e a viewporton (a böngésző azon területe, ahol a weboldal ténylegesen megjelenik) belül (forrás: [4]).

```
var isInViewport = function (elem) {
    var bounding = elem.getBoundingClientRect();
    return (
        bounding.top >= 0 &&
        bounding.left >= 0 &&
        bounding.bottom <= (window.innerHeight ||
document.documentElement.clientHeight) &&
        bounding.right <= (window.innerWidth ||
document.documentElement.clientWidth)
    );
};
```

4. Vizuális tesztelés

Mivel az automatizált funkcionális tesztelő eszközök kevésbé alkalmasak vizuális hibák felkutatására, a vállalatok vizuális hibákat emberi tesztelők segítségével próbálják megtalálni

Az emberi vizuális tesztelés azt jelenti, hogy összehasonlítunk két képernyőképet, egyet az ismert jó alapképből, a másikat pedig az alkalmazás legújabb verziójából. Minden egyes képpár esetén sok idő kell, hogy minden hibát megtaláljunk a két kép között. A befektetett idő nagyban függ a kép bonyolultságától, részletességétől, hiszen ha csak egy olyan képet kell összehasonlítani, ahol három gomb és egy panel található, az jóval rövidebb idő lesz, mintha például két Facebook oldalról készült képet vennénk alapul.

A tesztelés

- rendkívül időigényes lehet, mert néhány különbséget nehéz észrevenni,
- megbízhatatlan, mert sok esetben a szemünk is becsaphat bennünket, és olyan különbségeket is megjelölhetünk, amelyek valójában nem is léteznek;
- monoton.

5. Kihívás a manuális tesztelésben

A weboldalak tesztelése a vizuális elemek és funkcióik ellenőrzésével kezdődik, adott operációs rendszeren, adott böngészőn, annak adott verzióján, adott képernyőméreten, orientáción és felbontáson.

Ha egy rendszeren tökéletesen végrehajtható a teszt, akkor ezeket az elemeket kombináljuk új beállításokkal. ebben rejlik a tesztelés egyik legnagyobb erőfeszítést igénylő feladata.

Az alábbi képlet adja meg a tesztesetek számát:

$$N = (OS * BR) * (\mathrm{M} * 2 + PC)$$

ahol

N: az összes teszteset
OS: operációs rendszerek száma
BR: böngészők száma
M: mobiltelefon képernyő felbontások száma
PC: számítógép képernyő felbontások száma

Például képzeljük el, hogy az applikációt teszteljük:

a) 5 operációs rendszeren:
- Windows,
- MacOS,
- Android,
- iOS,
- Chrome.

b) 5 népszerű böngészőn:
- Chrome,
- Firefox,
- Internet Explorer
- Microsoft Edge,
- Safari.

c) 2 képernyő orientáció esetén:
- portré és
- tájkép.

d) 10 standard mobilos felbontáson és 18 standard PC/Laptop felbontáson:
- XGA-tól a
- 4G-ig.

A számolás után azt kaphatjuk, hogy összesen 21 kombináció állítható elő, ehhez hozzájön a mobilok kétféle orientációjának a körülbelül tízszerese (ennyi különböző felbontású telefont különböztetünk meg, ennél jóval több lehet) azaz 2x10=20, és még a körülbelül 18 féle PC/Laptop felbontás.

Ezeket összegezve kaphatjuk:

$$25 * (20+18) = 21 * 38 = 798$$

Ez 798 különböző képernyő konfiguráció, amit tesztelni kell, ami rengeteg tesztelést jelent, és ez csak egy oldalnyi vagy képernyőnyi adat.

Ráadásul, sok cég hetente, vagy akár naponta is kiad egy új frissítést a szoftverére, amelyet újabb tesztelést igényelnek. Itt merül fel a tesztelés automatizálása.

6. Vizuális regresszió tesztelés

A vizuális regresszió tesztelés, a weboldalak megjelenését vizsgálja, akár különböző böngészők esetén, vagy különböző termék verziók között. Ezen folyamat automatizálása kiemelten fontos, mivel ha manuálisan történik, akkor nagyon sok emberi erőforrást kell befektetni, amit máshol, jobban is fel lehet használni. [7] bináris transzformációval végzi el az összehasonlítást. Yang et. al [8] a 3D alakzatokra fókuszál. Vizuális mintázatokat keres mesterséges intelligencia módszerekkel [9]. Videók feldolgozásával foglalkozik Alamri et al [10]. Számos szabadalom született a tématerülettel kapcsolatban: [11-16]. Vannak már eszközök [17-24] erre a célra, amik fel vannak okosítva, hogy hagyják figyelmen kívül a dinamikus tartalmakat vagy szűrjék ki a hamis pozitív eredményeket, viszont már egy egyszerű script is jócskán csökkentheti az időt, amit a manuális tesztelőknek kellene eltölteni egy-egy weboldal tesztelésével.

A legegyszerűbb program, amit ilyen célra írhatunk, az egy olyan program, ami összehasonlít két képet valamilyen módon és valamilyen eredmény formátumba megjelöli, hogy hol találhatók különbségek a két kép esetén. Ilyenkor a tesztelő dolga a képeket elkészíteni és biztosítani a programnak, viszont az összehasonlítást az elvégzi.

Az alábbi megvalósítás [5] alapján készült, Python nyelven íródott és a Pillow képfeldolgozó könyvtárat használja. A függvény megkap egy képet, valamint x és y koordinátákat, ami a kép egy négyzet alakú szeletének az egyik sarok pontját jelzik és a négyzet szélességét és magasságát onnantól számítva. Majd kiszámolja a négyzeten belüli átlagos fényességet és visszaadja azt. Helyet kapott még egy érzékenységi faktor, amit minél magasabbra állítunk annál kevésbé lesz érzékeny majd az összehasonlítás.

```
def process_region(self, image, x, y, width, height):
    region_total = 0

    # This can be used as the sensitivity factor, the larger it is the less
sensitive the comparison
    factor = 100

    for coordinateY in range(y, y+height):
        for coordinateX in range(x, x+width):
            try:
                pixel = image.getpixel((coordinateX, coordinateY))
                region_total += sum(pixel)/4
            except:
                return

    return region_total/factor
```

Ezt a függvényt felhasználva kapunk mindkét kép szeleteire egy értéket, és a megfelelő szeletek értékeit összehasonlítva megkapjuk, hogy mely szeletek térnek el. Bármilyen is legyen a kép, az alábbi függvény 160x120 szeletre osztja fel a képeket, majd kiszámolja, hogy egy szeletnek mekkora a szélessége és a magassága. Most már csak végig kell iterálni a képen és meghívni az előző függvényt, majd összehasonlítani a kapott értékeket és a könyvtár segítségével berajzolni azokat a négyzeteket, amelyek eltérnek. Ebben az esetben az eredményt az egyik képre rajzolt piros négyzetek jelentik, amiket egy új eredmény képként mentünk le, de lényegében bármilyen formában, akár szöveg fájlba írva koordinátákkal együtt is vissza lehet adni.

```
def analyze(self):
    print("Analyzing screenshots...")
    screenshot_chrome = Image.open("screenshots/screenshot_chrome.png")
    screenshot_firefox = Image.open("screenshots/screenshot_firefox.png")
    columns = 120
    rows = 160
    screen_width, screen_height = screenshot_chrome.size
    block_width = ((screen_width - 1) // columns) + 1 # this is just a
division ceiling
    block_height = ((screen_height - 1) // rows) + 1

    for y in range(0, screen_height, block_height+1):
            for x in range(0, screen_width, block_width+1):
                    region_staging = self.process_region(screenshot_chrome,
x, y, block_width,                         block_height)
                    region_production =
self.process_region(screenshot_firefox, x, y,
block_width, block_height)
                    if region_staging is not None and region_production is
not None and                    region_production != region_staging:
                        draw = ImageDraw.Draw(screenshot_chrome)
                        draw.rectangle((x, y, x+block_width,
y+block_height), outline =                         "red")
    screenshot_chrome.save("screenshots/result.png")
    print("Result screenshot saved!")
```

Ha megelégszünk ennyivel, akkor nincs más dolgunk csak a megfelelő névvel elhelyezni a képeket a megfelelő mappába és a script összehasonlítgatja nekünk. Viszont manuális képeket nem egyszerű készíteni és ha esetleg a kép vágásakor egy két pixelnyi hibát követünk el, akkor sok piros négyzet lesz az eredmény, amin nem az a hiba, amit keresünk. Tehát a következő lépés a képkészítés automatizálása lenne. Erre a célra az eléggé elterjedt Selenium automatizálási keretrendszert használtuk. Sok funkciója van, de nekünk csak annyi kell, hogy készítsen egy képet különböző böngészőkkel és mentse őket le. Ahhoz, hogy ez megtörténhessen a seleniumon kívül szükség lesz két

webdriverre, esetünkben a chrome-ra és a firefox-ra, ezeket csak fel kell installálni és hozzáadni a PATH-hoz. Az alábbi függvénnyel inicializáljuk a drivereket, megadjuk nekik a *headless* argumentumot, ami lényegében azt jelenti, hogy nem nyitja meg a böngészőt, és azt, hogy maximalizálja az ablakot, azért, hogy ne legyen gond abból, hogy az egyik böngésző alapértelmezetten kisebb méretű ablakban nyílik, mint a másik.

```
def set_up(self):
    # Chrome
    options = webdriver.ChromeOptions()
    options.add_argument('headless') # Headless option
    options.add_argument("start-maximized");
    self.ChromeDriver = webdriver.Chrome(options=options)
    # Firefox
    options = webdriver.FirefoxOptions()
    options.headless = True # Headless option
    self.FirefoxDriver = webdriver.Firefox(options=options)
    self.FirefoxDriver.maximize_window()
```

A képkészítő függvény a weboldal url-t, a lementendő kép elérési útját és a driver típusát kéri. Az utóbbi alapján készít valamely driverrel egy képet és menti a megadott elérési útra.

```
def screenshot(self, url, file_name, driver_type):
    path = os.path.join('./', 'screenshots', file_name)
    if (driver_type == 'chrome'):
            print("Capturing", url, "screenshot as", file_name, "with 
chrome...")
            self.ChromeDriver.get(url)
            height = self.ChromeDriver.execute_script("return 
document.body.scrollHeight")
            self.ChromeDriver.set_window_size(1920,height+100)
            self.ChromeDriver.save_screenshot(path)
            print("Done.")
    elif (driver_type == 'firefox'):
            print("Capturing", url, "screenshot as", file_name, "with 
firefox ...")
            self.FirefoxDriver.get(url)
            height = self.FirefoxDriver.execute_script("return 
document.body.scrollHeight")
            self.FirefoxDriver.set_window_size(1920,height+100)
            self.FirefoxDriver.save_screenshot(path)
            print("Done.")
```

Ezek voltak a fontosabb lépései a script működésének. Az előnye ennek a scriptnek az, hogy egyszerű és jól leszűkíti a manuálisan átnézendő képek körét. A hátránya mindenképpen az, hogy nem elég komplex ahhoz, hogy dinamikus tartalmakat kezeljen, amik viszont elég nagy részét teszik ki a weboldalaknak manapság. Lényegében minden pixel csoportot, amely más azt hibásnak jelent, viszont mondjuk egy hír oldalnál nem az a fontos, hogy milyen hír jelenik meg, hanem az elrendezés. Nagy probléma az is, hogy kis eltérésekre is nagyon érzékeny, amik főleg különböző böngészőknél jelentkeznek, így mikor egy pici eltérés elviekben nem is lenne baj, akkor is berajzol egy csomó négyzetet. Persze az valamennyire ellensúlyozható az érzékenység faktor kalibrálásával, de így is elég zavaró lehet.

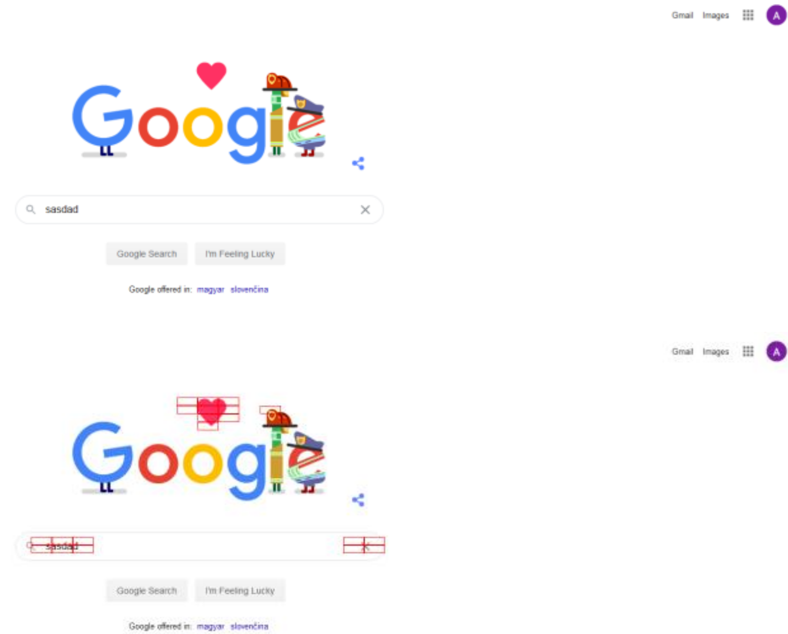
Ami a script teljesítményét illeti, a futási időt a kép mérete befolyásolja a legjobban. Kevésbé görgethető weboldalak képeinél a kép készítés ideje nagyobb, mint az analizálás ideje, azonban, ez a kép méretének növelésével megfordul. A képkészítés továbbá nem ad mindig ugyanolyan eredményt, mivel a külső driver processztől függ. Tehát hogy milyen gyorsan tölt be az oldal, milyen gyors az internet, stb. is befolyásolják képkészítési gyorsaságot. Az analizálás a hardvertől és a kép méretétől függ, az 1. táblázatban megadott időeredmények érvényesek:

1. táblázat

Weboldal	Méret	Kép készítés ideje (s)	Analizálás ideje (s)	Teljes idő (s)
google.hu	kicsi	8	3	20
miskolc neptun	közepes	6	8	25
hvg.hu	nagy	16	40	67

Megjegyzés: a részidők azért nem adják ki a teljes futási időt, mivel abba beleszámít a driverek inicializálása és memóriából való kitörlése is.

A következő ábra a teszteredményeket mutatja. Jól látható, hogy az egészen apró eltéréseket is kiemeli a kód.



6. ábra. *Teszteredmények a Google weboldallal.*

7. Összefoglalás

A kidolgozott script alkalmas arra, hogy szemléltesse az automatikus regressziós tesztelést a használt technológiákkal, valamint alkalmas lehet egyszerűbb statikus weboldalak gyors tesztelésére, azonban dinamikus, kiterjedtebb weboldalakhoz más megoldások használata indokolt. Beláttuk, hogy a vizuális tesztelés viszonylag könnyen automatizálható és egyszerűbb statikus tartalmakkal rendelkező oldalak esetén jól működik. Azonban dinamikus tartalmak esetén a tesztkörnyezet konfigurálása több időt vesz igénybe, illetve a tesztesetek futtatása is hosszabb.

8. Köszönetnyilvánítás

A cikkben ismertetett kutatómunka az EFOP-3.6.1-16-2016-00011 jelű „Fiatalodó és Megújuló Egyetem – Innovatív Tudásváros – a Miskolci Egyetem intelligens szakosodást szolgáló intézményi fejlesztése" projekt részeként – a Széchenyi 2020 keretében – az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg".

Irodalom

[1] W3Counter, "Browser & Platform Market Share," [Online]. Available: https://www.w3counter.com/globalstats.php. [Hozzáférés dátuma: 2020. 04. 28.].

[2] StatCounter: "Global Stats," [Online]. Available: https://gs.statcounter.com/. [Hozzáférés dátuma: 2020. 04. 28.].

[3] Srinivasan, N. Planning your browser compatibility tests [Online]. Available: https://blog.aspiresys.com/testing/planning-your-browser-compatibility-tests/. [Hozzáférés dátuma: 2020. 04. 28.].
[4] Go Make Things, "How to test if an element is in the viewport with vanilla JavaScript," [Online]. Available: https://gomakethings.com/how-to-test-if-an-element-is-in-the-viewport-with-vanilla-javascript/. [Hozzáférés dátuma: 2020. 04. 28.].
[5] Ussenov, R.: "Automating Visual Regression Tests with Python and Selenium," [Online]. Available: https://blog.rinatussenov.com/automating-manual-visual-regression-tests-with-python-and-selenium-be66be950196. [Hozzáférés dátuma: 2020. 04. 28.].
[6] Tóth, T: Minőségmenedzsment és informatika. elektronikában, Műszaki Könyvkiadó, Budapest, 1999., ISBN 080 90 00147650
[7] Zhang, C., Li, C., Cheng, J. Few-shot visual classification using image pairs with binary transformation. IEEE Transactions on Circuits and Systems for Video Technology, 2019. **https://doi.org/10.1109/TCSVT.2019.2920783**
[8] Yang, J., Ren, Z., Xu, M., Chen, X., Crandall, D., Parikh, D., Batra, D. Embodied visual recognition. arXiv preprint arXiv:1904.04404. (2019).
[9] Jeelani, I., Albert, A., Han, K., & Azevedo, R. Are visual search patterns predictive of hazard recognition performance? Empirical investigation using eye-tracking technology. Journal of construction engineering and management 2019, 45(1):04018115. **https://doi.org/10.1061/(ASCE)CO.1943-7862.0001589**
[10] Alamri, H., Cartillier, V., Das, A., Wang, J., Cherian, A., Essa, I., Lee, S. Audio visual scene-aware dialog. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition 2019, pp. 7558-7567. **https://doi.org/10.1109/CVPR.2019.00774**
[11] Miller, E. F. (2019). U.S. Patent No. 10,489,286. Washington, DC: U.S. Patent and Trademark Office.
[12] Van, P. H. O., Phipps, D. A., & Lin, S. (2019). U.S. Patent No. 10,241,901. Washington, DC: U.S. Patent and Trademark Office.
[13] Fryc, L., Tisnovsky, P. (2016). U.S. Patent No. 9,298,598. Washington, DC: U.S. Patent and Trademark Office.
[14] Kogan, O., Levin, A. (2019). U.S. Patent Application No. 16/068,782.
[15] Unified model for visual component testing. U.S. Patent No 9,720,811, 2017.
[16] Arieli, G. (2019). U.S. Patent Application No. 15/937,892.
[17] Stocco, A., Yandrapally, R., Mesbah, A. Visual web test repair. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering 2018, pp. 503-514. **https://doi.org/10.1145/3236024.3236063**
[18] Mahajan, S., Gadde, K. B., Pasala, A., & Halfond, W. G. Detecting and localizing visual inconsistencies in web applications. In 2016 23rd Asia-Pacific Software Engineering Conference (APSEC) 2016, pp. 361-364. IEEE. **https://doi.org/10.1109/APSEC.2016.060**
[19] Bajammal, M., Mesbah, A. Web Canvas Testing through Visual Inference. In 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST) 2018, pp. 193-203. IEEE. **https://doi.org/10.1109/ICST.2018.00028**
[20] Leotta, M., Stocco, A., Ricca, F., Tonella, P. Automated generation of visual web tests from DOM-based web tests. In Proceedings of the 30th Annual ACM Symposium on Applied Computing 2015, pp. 775-782. **https://doi.org/10.1145/2695664.2695847**

[21] Leotta, M., Clerissi, D., Ricca, F., Tonella, P. Visual vs. DOM-based web locators: An empirical study. In International Conference on Web Engineering 2014, pp. 322-340. Springer, Cham. **https://doi.org/10.1007/978-3-319-08245-5_19**
[22] Jia, X., & Liu, H. Rigorous and automatic testing of web applications. In Proceedings of the 6th IASTED International Conference on Software Engineering and Applications 2002, pp. 280-285.
[23] Pautasso, C. JOpera: An agile environment for web service composition with visual unit testing and refactoring. In 2005 IEEE Symposium on Visual Languages and Human-Centric Computing 2005, pp. 311-313.
[24] Saad, M. B., Gançarski, S., & Pehlivan, Z. A novel Web archiving approach based on visual pages analysis. In The 9 th International Web Archiving Workshop (IWAW 2009) Corfu, Greece, September/October, 2009 Workshop Proceedings.