

TDK-dolgozat

Csányi Árpád

Szegedi Tudományegyetem
Természettudományi és Informatikai Kar
Informatikai Tanszékcsoport

Hierarchikus rendszerek vizualizációja,
animációja és alkalmazásai

Készítette:
Csányi Árpád
gazdaságinformatika MSc szakos
hallgató, 2. évf.

Témavezető:
Dombi József
egyetemi tanár

Tartalomjegyzék

1. Bevezetés	4
2. Színek és formák	6
2.1. Színmodellek, színkeverés	6
2.1.1. RGB	6
2.1.2. CMYK	7
2.1.3. HSL, HSV/B, HSI	7
2.1.4. Színek csoportosítása	8
2.2. Színhangzatok, harmonikus színek	8
2.3. Alakzatok	10
3. Animáció és interakció	12
3.1. Animáció	12
3.2. Interakció	13
3.2.1. Digitális interakció	13
4. Hierarchián belüli kapcsolatok	16
5. A program	18
5.1. Technológiai háttér	19
5.2. Az adatok	20
5.3. Felhasználói felület	22
5.4. Működés	23
5.5. Hogyan mozog a vonal?	26
5.6. Továbbfejlesztési lehetőségek	29
6. Alkalmazási lehetőségek	30
6.1. Informatikai alkalmazás	30
6.2. Gazdasági alkalmazás	31
6.3. Mi mindenre jó még?	32
Köszönetnyilvánítás	34
Irodalomjegyzék	35

Első fejezet

Bevezetés

A hierarchikus szerkezetek evolúciós találmánynak tekinthetők. Gondoljunk csak *Charles Darwin - A fajok eredete*¹ című művére, melyben lefekteti az evolúcióelmélet alapjait. Lényege, hogy a fajok folyamatosan átalakulnak: új fajok keletkeznek, míg a nem életképesek kihalnak [3]. Az átalakulások oka rendkívül sokféle lehet: környezeti változások (pl.: a jégkorszak vagy egyéb természeti katasztrófák), más fajokkal való küzdelem a területért vagy az élelemért. Ennek az átalakulásnak a számunkra legkézzelfoghatóbb része az ember evolúciója, amely során kialakult a mai ember, a *Homo sapiens*. Alaposabban körüljárva a témát azonnal láthatunk is egy példát a hierarchikus kapcsolatra, ugyanis a *Homo sapiens* fajnak több alfaja is van, amelyek közül a ma élő ember a *Homo sapiens sapiens*.

Darwin munkájának egyik eredménye az élőlényeket hierarchiába soroló *rendszertan* amelynek legfőbb szintjei (avagy alapkategóriái): ország, törzs, osztály, rend, család, nemzetség, faj. Balról jobbra haladva minden szint között szülő-gyerek kapcsolat áll fenn azzal a megszorítással, hogy egy szülőnek több gyereke is lehet, de egy gyereknek pontosan egy szülője van. Az ilyen típusú kapcsolatokat 1-N kapcsolatnak hívjuk.

Az evolúció mint idegen szó jelentése is alátámasztja a hierarchiával való kapcsolatot: fejlődés, fokozatos változás [4]. Ez alapján azt mondhatjuk, hogy a hierarchia különböző szinteken lévő elemei közötti kapcsolatot az evolúció teremti meg.

De ez az alá- és fölérendeltségi viszony nemcsak a természetben van jelen, hanem nagyon sok, az ember által alkotott dologban is. Bizonyos esetekben ezek valóban alá- és fölérendeltségi viszonyt jelentenek, mint például a hűbéri rendszer egyes szintjei vagy a földesurak és a jobbágyok közötti kapcsolat. Más szituációkban pedig a hierarchia rész-egész kapcsolatot jelöl, például egy autónak van kereke, a kerék felniből

¹Elérhető elektronikus formában a Magyar Elektronikus Könyvtáron keresztül: <http://mek.niif.hu/05000/05011/html/>

és abroncsból áll. Vagy tekintve egy informatikai példát: a számítógép alkatrészekből épül fel, úgy mint processzor, memória, amelyek további, apróbb alkatrészekből épülnek fel.

Az előzőekben említett példákban a hierarchiákat mint adottságokat tekintettük. Ezzel szemben vannak olyan esetek, amikor a hierarchiát tudatosan alakítja ki az ember. A szoftverek ennek egy tipikus példája, ugyanis egy nagy alkalmazás sok kis részből áll: az egyik a hálózati kapcsolatokért felel, a másik a megjelenítésért míg a harmadik az adatbevitelért, amelyen belül további két lehetőség van: a billentyűzet és az egér. A felsorolásban jól látszik a hierarchikus szerkezet. Felmerülhet a kérdés, hogy miért hoznak létre ilyen hierarchikus szerkezeteket a szoftverekben? Azért, mert a program futása során általában nincs szükség minden komponensre. Éppen ezért az egyes részeket csomagokba (*package*) rendezik a fejlesztők. Például ha egy Androidos telefonra írt program szeretne bluetooth-on keresztül kommunikálni, akkor biztosan szüksége lesz az `android.bluetooth.BluetoothAdapter`¹ használatára. Ennek a hierarchikus rendszerezésnek van egy olyan előnye is, hogyha a bluetooth csomagból mindenre szükségünk van, akkor egyszerűen írhatjuk azt, hogy `android.bluetooth.*`. Ezzel óvatosan kell eljárni mert lehet, hogy sok olyan csomagot is behúzunk az alkalmazásunkba, amit nem is fogunk használni később. Van olyan programozási nyelv (Go²), amely a fordítás során hibával megáll, ha olyan importált csomagot talál, amit a programban egy helyen sem használunk.

A dolgozat további részében a hierarchikus szerkezetek vizuális megjelenítésével foglalkozunk. A hierarchia vizualizációjának az az előnye a szöveges leírással szemben, hogy ez egy sokkal tömörebb ábrázolásmód. A tömörséget nem feltétlenül a papíron elfoglalt hely jelenti, hanem a hordozott információ mennyisége. Megfelelő kódolással – és hozzá tartozó jelmagyarázattal – nagyban hozzájárulhatunk az adatok könnyebb feldolgozásához és értelmezéséhez.

¹Illetve további komponensekre is, amelyek tárgyalás túlmutat jelen dolgozat céljain.

²<http://golang.org/>

Második fejezet

Színek és formák

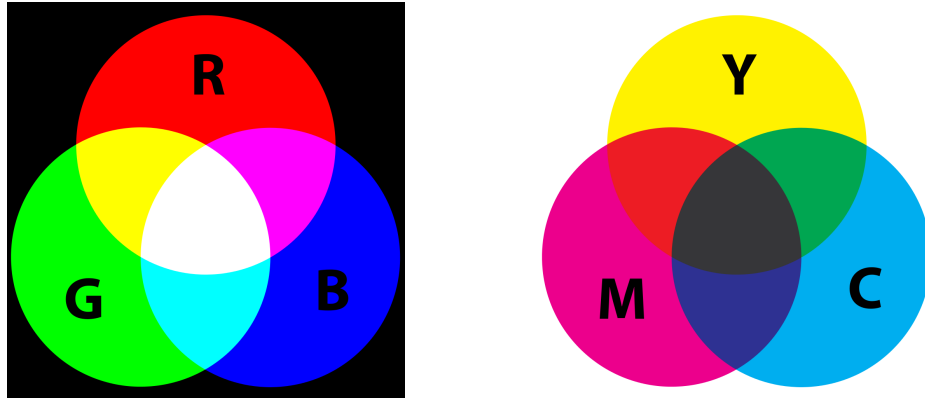
Kis mennyiségű információ megjelenítésére elegendő egy szín. De ha sok a megjelenítendő adat, vagy azokat valamilyen módszer szerint rendszerezve, csoportosítva szeretnénk ábrázolni, akkor mindenképpen ajánlott színek és formák segítségével kódolni az információt. Viszont fontos szem előtt tartani, hogy nem használhatjuk ezeket a vizuális elemeket akármilyen kombinációban, hanem be kell tartani bizonyos szerkesztési szabályokat. Ezek figyelmen kívül hagyásával pont a célunkkal ellentétes eredményt érünk el, ugyanis az ábránk zavaros, nehezen értelmezhető lesz legyen szó akár statikus vagy animált megjelenítésről.

2.1. Színmodellek, színkeverés

A szín nem más mint a fény hullámhosszától függő észlelet. A fénynek van egy, az emberi szem számára látható tartománya, amelyre a retinánk érzékeny. A színek észlelésért az itt található csapok a felelősek, amelyekből három fajta van aszerint, hogy milyen fényre érzékenyebbek: rövid- (400–500 nm), közepes- (450–630 nm) és hosszú (500–700 nm) hullámhosszúságú. Ha vesszük a három fő színt, és a hozzájuk tartozó hullámhosszakat: kék (420–490 nm), zöld (490–575 nm) és vörös (650–750 nm), akkor ezekből nagyjából az látszik, hogy egy-egy csap egy-egy színre specializálódott [10].

2.1.1. RGB

Az RGB színmodell a piros, zöld és kék színekből áll. Nevét az angol Red, Green, Blue szavak kezdőbetűi után kapta. Ez a színmodell úgynevezett additív komponensekkel dolgozik, ami azt jelenti, hogy a három alapszín különböző súlyokkal vett összege határozza meg az ábrázolni kívánt színt. Az informatikában ennek leképezé-



2.1. ábra. Additív (RGB) és szubtraktív (CMYK) színkeverés

sére minden egyes komponenst 1 byte-on ábrázolnak, ami 8 bit, így egy komponens $2^8 = 256$ különböző súllyal rendelkezhet. Ez azt jelenti, hogy ilyen módon összesen $256^3 = 16\,777\,216$ szín ábrázolható. A digitális megjelenítők ezt a színmodellt használják. A színészlelet a kibocsátott fény alapján jön létre.

2.1.2. CMYK

A második színmodell a CMYK, amely bizonyos szempontból pontosan az ellentéte az RGB-nek. Ez ugyanis nem a kibocsátott, hanem a visszavert fény segítségével hozza létre a színérzetet. A modell szubtraktív komponensekből áll, ami azt jelenti, hogy a három alapszín: Cyan, Magenta, Yellow (cián, bíbor, citromsárga) szűrőként használja, hogy a fehérből előállítsa a megfelelő színt. De önmagában ezzel a három színnel nem tudjuk úgy megszűrni a fehér fényt, hogy abból feketét kapjunk, ezért a modellben külön szerepel a fekete, mint kulcsszín (Key).

2.1.3. HSL, HSV/B, HSI

A harmadik színmodell a HSL (Hue-Saturation-Lightness), HSV/B (Hue-Saturation-Value/Brightness) illetve HSI (Hue-Saturation-Intensity). Mindhárom modell hengerkoordináta-rendszerben ábrázolja a színeket. A koordináta-rendszer referenciapontja általában a henger alapját alkotó kör lap középpontja. Az ettől a referenciaponttól vett φ radiális távolság, azaz az ívhossz határozza meg a színárnyalatot (Hue), a ρ sugár irányú távolság a telítettséget (Saturation), míg a z függőleges távolság a harmadik paramétert (világosság/fényesség/intenzitás) írja le. Ezeknek a rendszereknek az előnyük, hogy sokkal közelebb állnak a hagyományos színerék koncepcióhoz mint az RGB, ami egy 3D derékszögű koordináta-rendszert használ. Ezeket a henger formájú színtereket gyakran használják képfeldolgozási alkalmazás-

sokban képek szegmentálására és tulajdonság kinyerésre (feature-detection). Emellett sok színválasztó alkalmazás is ezen a modellen alapul, mivel intuitívabb mint az RGB. A HSV színmodell adatvizualizációs szempontból is jól használható színátmenetek generálásához.

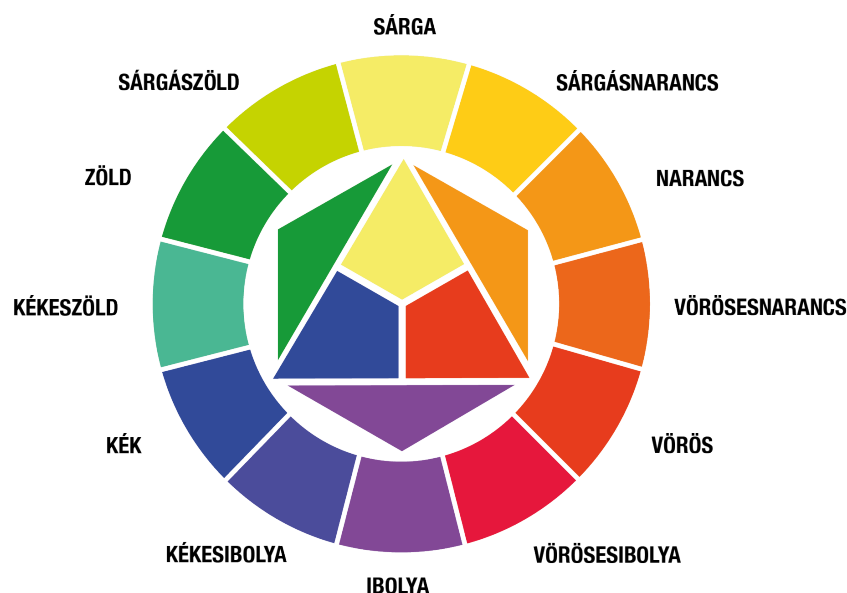
A gyakorlatban sokszor hasznos az RGB és HSL színsémáknak az alpha csatornával kombinált RGBA és HSLA változatát használni. Az alpha csatorna a színek áttetszőségéért felel. Értéke egy $[0-1]$ közötti skálán ábrázolható, ahol 0 a teljesen áttetsző szint jelöli, míg 1 az egyáltalán nem áttetsző változatot.

2.1.4. Színek csoportosítása

A színeket többféleképpen lehet csoportosítani. Ezek a csoportosítások általában szubjektív alapon történnek.

- Meleg színek: piros, narancssárga, citromsárga
- Hideg színek: zöld, kék, ibolya (lila)
- Semleges színek: fekete, szürke, fehér, barna, beige (bézs)

2.2. Színhangzatok, harmonikus színek

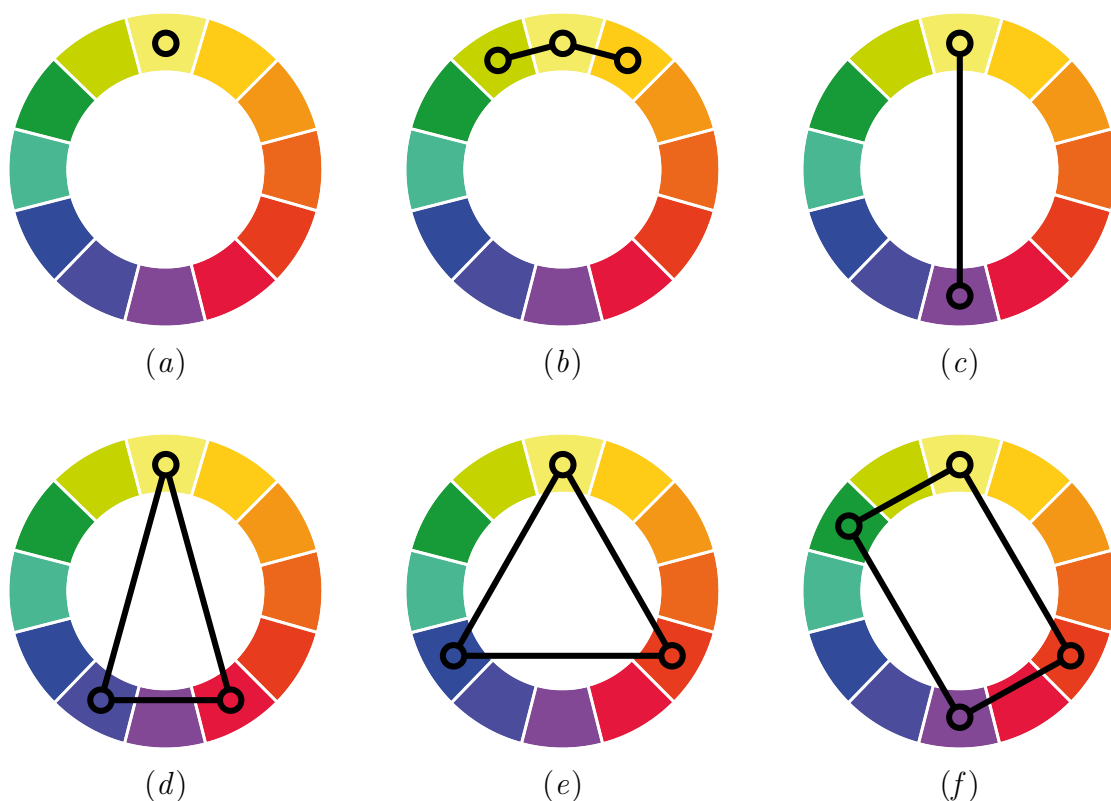


2.2. ábra. A 12 osztatú színekör

A színeket nem használhatjuk tetszőleges kombinációban. Ennek oka, hogyha egy ábrán több szín is van akkor a színek egymásra gyakorolt hatása befolyásolja az észlelt színt. A színsémák olyan színeket csoportosítanak, amelyek jól használhatók együtt. A színsémákhoz jó kiindulási alapot jelent a 2.2. ábrán látható 12 osztatú színekör [9].

A legegyszerűbb színhangzat a *monokromatikus*, amely egyetlen színből áll. Valójában nemcsak 1 színnel dolgozhatunk ilyenkor, ugyanis ez a módszer a HSL modellből csak a színárnyalat (Hue) komponenst zárolja. A telítettség (Saturation) és a világosság (Value/Brightness) szabadon változtatható. Előnye, hogy a színek mindig kiegyensúlyozottak lesznek, hátránya viszont az alacsony kontraszt.

A következő színhangzat az *analóg* színeket tartalmazza. Azt a színhármast hívjuk analógnak, amelyek a 12 osztatú színekörön közvetlenül egymás mellett helyezkednek el. Ez azt jelenti, hogy az egyes színek egymással 30° -os szöget zárnak be. Előnye a monokromatikus színekhez képest, hogy több színárnyalattal dolgozhatunk, viszont mivel a színek egymáshoz közel helyezkednek el a színekörön, továbbra is alacsony kontraszttal rendelkeznek.



2.3. ábra. Színhangzatok: (a) monokromatikus színhangzat, (b) analóg színhangzat, (c) komplementer vagy kiegészítő színhangzat, (d) osztott komplementer színhangzat, (e) tirad színhangzat, (f) tetrad színhangzat

A *kiegészítő* vagy *komplementer színséma* két színből áll, amelyek egymással szemben helyezkednek el a színekörön. A két szín által bezárt 180° -os szög miatt ezeknek a színeknek a legnagyobb a kontrasztja. Akkor néznek ki legjobban ezek a színpárok, amikor egy meleg szín egy hideggel áll szemben.

Az *osztott komplementer színhangzat* a kiegészítő és az analóg hangzatok kombinációja. Egy fő színnel rendelkezik, és további két szín található a vele szemben lévő színtől két irányban 30° -os távolságra. Ez a séma a két korábbi séma előnyeit igyekszik kombinálni, azaz hogy több árnyalattal dolgozhassunk relatív nagy kontraszt mellett.

A *tírad színhangzat* – amint nevében is benne van – három színből áll, amelyek egymástól egyenlő távolságra találhatók, azaz 120° -os szöget zárnak be egymással, azaz a színek a színekör fölé rajzolt egyenlő oldalú háromszög csúcsainál helyezkednek el. Ez a színhangzat a magas kontraszt és a színharmónia optimális arányát célozza.

Az utolsó színhangzat a *tetrad*, ami 4 színből áll úgy, hogy azokat 2 darab komplementer hangzat adja, amelyek tengelyei egymással 60° -os szöget zárnak be. Ez a színhangzat rendelkezik a legtöbb színnel, amelynek következménye, hogy ennél a legnehezebb úgy összeállítani a sémát, hogy a színei harmonizáljanak. Ha minden egyes színt egyenlő mértékben használunk, akkor az ábra kiegyensúlyozatlan lesz. Ennek elkerülése végett ki kell választani egy fő színt, amely dominálni fog a képen.

Mit jelent a színek harmóniája?

A harmónia a különböző hullámhosszú színek közötti matematikai kapcsolatra utal. Gyakorlatiasabb megfogalmazásban két vagy több színt akkor nevezünk harmonikusnak, ha keverékükből a semleges szürke szín jön létre.

A világhálón sok eszköz található színsémák generálására. A paletton-ban (<http://paletton.com>¹) például a fent tárgyalt színhangzatok mellett a felhasználó tetzőlegesen állíthatja az összes paramétert.

2.3. Alakzatok

Bertin [1] az alábbi három alapvető alakzatot határozza meg a síkon:

A legegyszerűbb ábrázolható alakzat a *pont*. A pont egy helyet reprezentál a síkon, amelynek nem értelmezett sem a hossza, sem a területe. Ilyen értelemben a pont független az őt megjelenítő karaktertől vagy jeltől.

¹Példa a paletton alkalmazással generált analóg színhangzatra: 
<http://paletton.com/#uid=50u0u0kvbwajdD0oux-xnoMCEjw>

A *vonal* egy olyan jelenség, amely a síkon mérhető hosszal rendelkezik, de nincs területi kiterjedése. Alap jelentését tekintve nem függ a vonal vastagságától vagy formájától, görbületétől.

A *terület* mérhető dimenziókkal rendelkezik a síkon. A ponttal és a vonallal szemben az őt megjelenítő alakzat által lefedett teljes területre vonatkozik a mérete.

Ezen formák kombinálásával jeleket kapunk, amelyek egyes tulajdonságaihoz hozzá tudjuk kötni az információt. Ezt *információ ráültetésnek* hívjuk. Például egy térképen a városok pontokkal vannak jelölve, amelyeket körök reprezentálnak és általában a kör átmérőjére a város népességének száma van ráültetve.

Az alakzatoknak számos tulajdonsága van, amelyekre rá lehet ültetni egy-egy információt. Például: pozíció, méret, textúra, szín, árnyalat, irányultság, forma. Mivel a ráültetés nem egyértelmű kapcsolat az alakzat tulajdonsága és az információ között (például egy fertőzési térképen a kör mérete nem a népességszámot, hanem a fertőzöttek számát jelöli), ezért az ábrák mellett jelmagyarázat (*legend*) található, amely leírja, hogy az egyes alakzatok és tulajdonságaik milyen információt kódolnak. Egy vizualizáció kiválóságát jól tükrözi, hogy jelmagyarázat nélkül tudja-e a felhasználó dekódolni és értelmezni az ábrázolt adatokat. Ahhoz hogy egy ábra jelmagyarázat nélkül is beszédes legyen, a ráültetés során az ember kognitív megismerő képességét kell figyelembe venni.

Az előző részben megállapítottuk a színekről, hogy az észlelésben fontos szerepük van (például befolyásolják az ábra kontrasztosságát). De a színek mellett az észlelést a megjelenítés környezete is befolyásolja. Ezért a vizuális tervezés során a színeket és formákat együttesen kell tekinteni.

Harmadik fejezet

Animáció és interakció

A színek és formák kombinálása már általában elegendő ahhoz, hogy felkeltsük a felhasználó figyelmét, ellenben az ábra értelmezéséhez nem minden esetben. Legalábbis van olyan technika, amelyek segítségével jelentősen megkönnyíthető illetve felgyorsítható az ábrák értelmezése.

Ez az animáció. Hátránya, hogy csak digitális környezetben használható. Viszont világunk egyre inkább eltolódik a digitális tartalmak felé: kezdve az internetes folyóiratokkal (amelyeknek akár nincs is nyomtatott formája), az e-bookokon keresztül egészen az elektronikus tananyagokig.

3.1. Animáció

A Lumière fivérek találmánya óta az embereket lenyűgözi a mozgókép. Ha egy álló képen valami megmozdul, akkor önkéntelenül is odapillantunk. Vizualizációs szempontból az animáció a mozgástól tágabb fogalom, hiszen a célunk vele a figyelemfelkeltés. Egy képen bármilyen változás felkelti az emberek figyelmét, legyen az akár hirtelen, gyorsan lezajló színváltozás vagy hosszabb ideig tartó, mozgással is együtt járó animáció.

Felmerül a kérdés, hogy valóban ekkora különbség van a animáció és a statikus képek között? A rövid válasz, hogy *nem*. Gyakorlatilag az animáció is statikus képek sorozata azzal a különbséggel, hogy egységnyi idő alatt (1s) annyi statikus képet látunk¹, hogy azokat az agyunk már folytonosnak észleli, tehát a mozgás érzet a feldolgozás során keletkezik. Erre egy egyszerű példa a füzet sarkára rajzolt „táncoló palcikaember”. Egyértelmű, hogy statikus képek sorozatáról van szó, ám amikor „végigpörgetjük” az érzékelés folytonos lesz, és animáció képződik belőle.

¹A klasszikus film esetén ez 24fps (frames per second = képkocka/másodperc)

Az animációnak és mozgásnak van egy további fontos szerepe az ábra esztétikája és harmóniája szempontjából. A statikus képekkel ellentétben a mozgóképek aktiválják az agy előrejelző képességét. Ez azt jelenti, hogy még mielőtt a következő időpillanatot feldolgozná az agyunk, már előre megpróbálja kiszámolni, hogy mire számítszon majd.

Az adatok megjelenítés és prezentálása során sok tudományterület alkalmaz animációt. A [16][14] forrásokban ezekre láthatunk példákat.

3.2. Interakció

Az animáció láttán a felhasználóban természetes igényként jelentkezik, hogy szeretne befolyással lenni erre a folyamatra. Erre ad lehetőséget az animáció továbbfejlesztett változata: az interaktív animáció.

Valós környezetünkben a különböző dolgokkal való interakció általában természetes módon adódik és már egészen kis korban elsajátítható. Gondoljunk csak egy gyurmával játszó kisgyermekre: csupán a kezei használatával azt alakíthat ki a gyurmából, amit csak szeretne. A végeredmény felett teljes kontrollja és döntési szabadsága van.

3.2.1. Digitális interakció

Esetünkben nem egyszerű a helyzet, ugyanis egy digitális termékkel kell a felhasználónak interakcióba lépnie, méghozzá olyan, amellyel nem is tud közvetlenül kapcsolatba lépni, hanem a kettőjük között van egy interfész. Ez lehet a billentyűzet, az egér vagy a manapság egyre elterjedtebb érintőkijelző¹. Az interakciós lehetőségeknek a választott interfész képességei szabnak végső korlátot (akár hardveres vagy szoftveres értelemben).

A digitális termékekkel való interakcióval a UX (User Experience = felhasználói élmény) foglalkozik. Az ezt tervező szakembernek tisztában kell lennie az interakciót biztosító eszköz fizikai tulajdonságaival és képességeivel, és úgy kell megterveznie a folyamatot, hogy a felhasználó számára minél intuitívabb és kényelmesebb legyen a program használata. A UX szorosan összefügg a UI (User Interface = felhasználói felület) fogalmával. Ezt igazolja az is, hogy egy érintőkijelzőn nagyobb felületű gombokat kell elhelyeznünk, mivel ujjunkkal egy kis gombot nehezebben tudunk

¹A felsorolás sorrendjében csökken az interfész által létrehozott „szakadék” nagysága

eltalálni.

A program *billentyűzetről* való vezérlése rendkívül hatékony. Erre a legjobb példa a *vim* (Vi IMproved) szövegszerkesztő. Mint nevéből látszik ez az eredeti *vi* szövegszerkesztőnek a továbbfejlesztett változata¹. A program által kínált nagyfokú hatékonyságnak „ára van”: ismerni kell bizonyos billentyűkombinációkat. De ha valaki egyszer megérti a programmal való interakció alapvető kompozíciós szabályát és megtanulja a legfontosabb vezérlési karaktereket, akkor egy-egy újabb funkció megtanulásához általában csak egy új karaktert kell megjegyezni (például kijelölés: *v*), majd kombinálni a már eddig is ismert parancsokkal (például kurzor mozgatás: *h*, *j*, *k*, *l*).

Joggal mondhatjuk, hogy a billentyűzeten bizonyos betűk megnyomásával parancsokat kiadni nem a legkényelmesebb megoldás. Az emberek elvárásainak általában jobban megfelel az *egérrel* való interakció, még ha az bizonyos esetekben lassabb is. Az egér a képernyőn mindig pontosan 1 pixelre mutat és klasszikus esetben a gombjaival három féle eseményt tudunk kezdeményezni: bal gombbal egyszeres kattintás, bal gombbal dupla kattintás, jobb gombbal egyszeres kattintás. Egy adott programban attól függően hajtódhatnak végre bizonyos parancsok, hogy az egérrel hova és milyen módon kattintottunk. Míg a billentyűzetes vezérlés során a program teljes felülete rendelkezésre áll a megjelenítendő adat számára, addig az egérrel való vezérléshez szükség van a felhasználói felületen elhelyezett gombokra. Bizonyos esetekben ezek az elemek az adat elől veszik el a hasznos területet. De mivel az egér segítségével nagy pontossággal lehet a kurzort pozicionálni, a gombok által elfoglalt hely minimalizálható.

A hordozható számítógépekben úgynevezett *touchpad*-ot találunk, ami egy érintésérzékeny felület. Legelső verziói még csak annyit tudtak mint a hagyományos egerek: az érintésérzékeny felület csak a pozicionálásért volt felelős és a kattintás a parányi téglalap alatt elhelyezett fizikai gombokkal történt. Az idők során ezek a felületek egyre nagyobbak lettek, és a fizikai gombok szerepe is csökkent: a felület megérintésével (koppintással) vagy egymás után kétszeri érintéssel lehetett kattintást vagy dupla kattintást végezni. A fizikai gombokat viszont nem lehetett teljesen elhagyni: legalább egyre mindenképpen szükség volt a jobb gombbal való kattintás miatt. A mai touchpadok még tovább fejlődtek, és egészen kicsire zárták a rést a modern (kapacitív) érintőkijelzők és maguk között: rendelkeznek multi-touch képességgel, azaz meg tudják állapítani, hogy hány ujjunk van egyszerre a felületen, azok milyen irányban mozognak és ennek függvényében tudnak különböző funkciókat

¹Jelenleg is aktívan folyik a *vim* továbbfejlesztése *neovim* néven: <http://neovim.org/>

végrehajtani. Így az egérrel kezdeményezhető akciók száma 3-ról hirtelen megsokszorozódott. Ugyan továbbra is a billentyűparancsok tekinthetők a leghatékonyabbnak, bizonyos funkciók végrehajtásában a touchpad átveszi az uralmat a billentyűzet felett, mert a rajta végzett kézmozdulatok (gesture) sokkal intuitívabbak és kényelmesebbek. Például a három ujjal való „söprés” az asztalok közötti váltásra szemben a **Ctrl** és a kurzorbillentyűk egyidejű megnyomásával.

Mai ismereteinket tekintve a széles tömegek számára is elérhető legkényelmesebb és legintuitívabb interakciós lehetőség az *érintőkijelző*. Gyakorlatilag ugyanazt tudja mint a modern touchpad-ok, azzal a különbséggel, hogy míg ott ujjunk mozgatásával relatív módon¹ pozicionálunk egy kurzort a képernyőn, addig az érintőkijelzők esetében nincs kurzor, hanem közvetlenül a tartalommal lépünk interakcióba. A tény, hogy nincs kurzor szükségessé teszi, hogy a felhasználói felület elemei nagyobbak legyenek, hiszen ujjunk mérete befolyásolja az érintés pontosságát. Viszont mivel ezek az eszközök is támogatják a gesztusokat, ezért sok esetben bizonyos funkciók eléréséhez nem kell gombot elhelyezni a felületen, elég azt valamilyen kézmozdulathoz kötni.

Az érintőkijelzők utódja lehet a LEAP Motion², amelynél a navigációhoz és a funkciók végrehajtásához egyáltalán nincs szükség érintésre, csak egy meghatározott felület felett kell gesztusokat végrehajtani a kezünkkel. Néhány modern okostelefon már rendelkezik hasonló funkcionálitással: például a telefon képernyője feletti „intégetéssel” tudunk hívást fogadni.

¹Az érintőkijelzőkre abszolút pozicionálás jellemző.

²<https://www.leapmotion.com/>

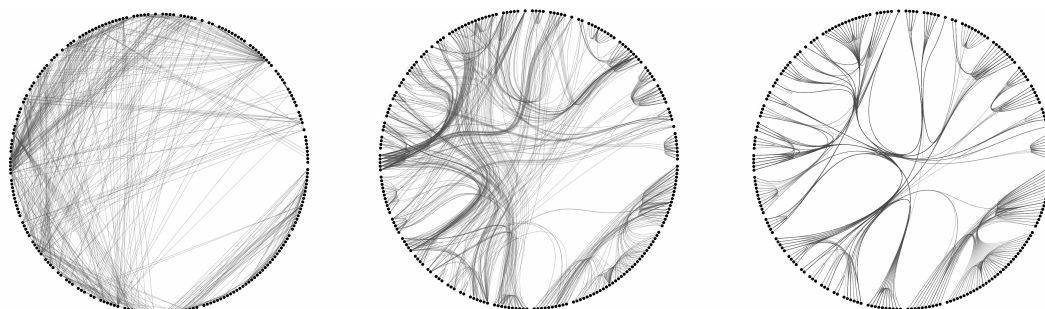
Negyedik fejezet

Hierarchián belüli kapcsolatok

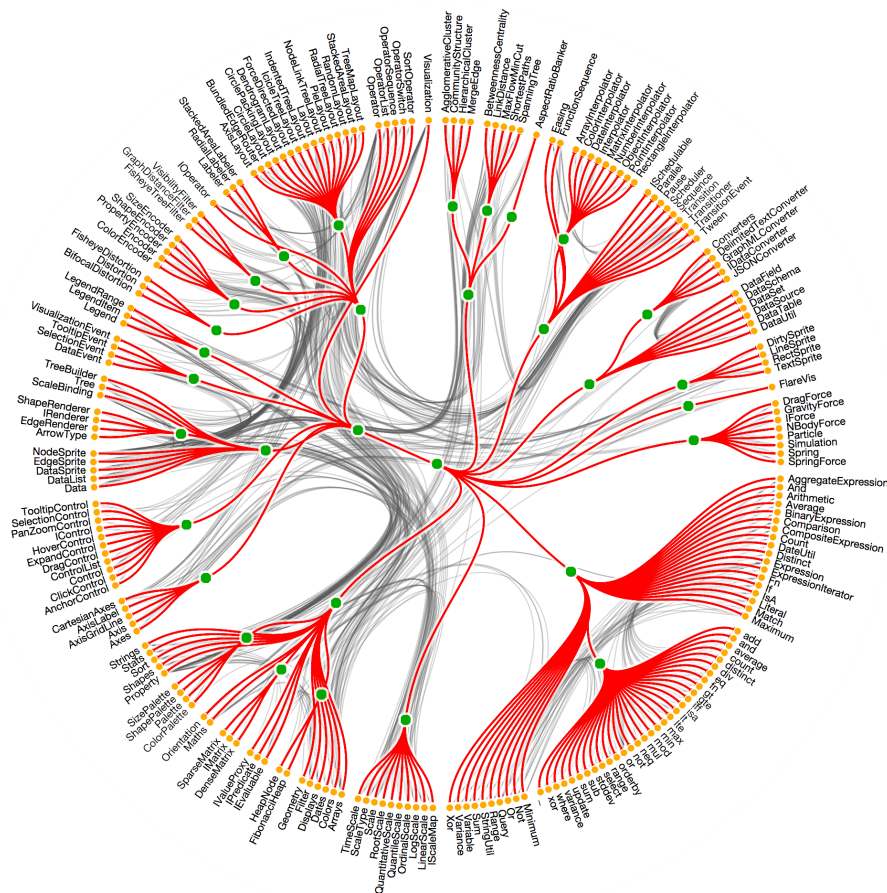
A hierarchikus rendszerek már önmagunkban is rendelkeznek kapcsolatokkal. Ezek a szülő-gyermek kapcsolatok, amelyek egyben a szerkezetet is leírják. De az egyes elemeknek ezen felül is lehet közük egymáshoz. Jelen dolgozat ezen másodlagos kapcsolatok vizualizációjával és animációjával foglalkozik.

A vizualizációs alapot Holten Hierarchical Edge Bundling metódusa adja [7], amely a 4.1. ábrán is látható. Az első körben az elemek csak egyenes vonalakkal vannak összekötve, de ez nem átlátható. A második körben a tension értéke 63, ami azt eredményezi hogy a vonalak görbülnek és egy sokkal tisztább képet kapunk. A harmadik körben a tension értéke maximális, ami azt jelenti hogy az egymás mellett futó vonalak össze vannak vonva, és egy vonalként látszanak.

A tension értéke azt határozza meg, hogy a vonalak görbülete mennyire kövesse a hierarchia szerkezetét: 0 esetén egyáltalán nem követi, 100 esetén pedig teljes mértékben. A vonal hosszát tehát az határozza meg, hogy a két elemnek (amelyeket összeköt) ki az első közös szülője és az mely köztes csomópontokon keresztül érhető el. Ez az ábrázolásmód nemcsak a másodlagos kapcsolatokat mutatja meg, hanem a hierarchia szerkezetére vonatkozó információt is tartalmazza. Alkalmazásunkban ki is használjuk ezt a plusz információt.



4.1. ábra. Példa a tension változtatására: 0, 63, 100.



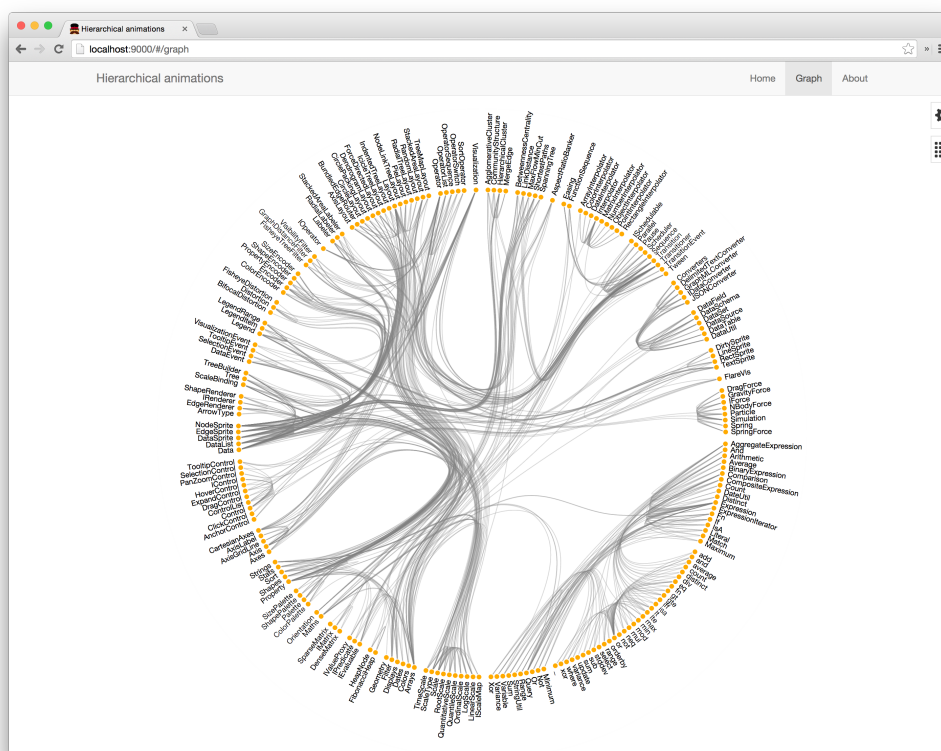
4.2. ábra. A hierarchia és a gráf kapcsolatainak viszonya

A 4.2. ábrán azt láthatjuk, hogy a gráf elemei között lévő kapcsolatok (fekete vonalak) görbülete hogyan viszonyul a hierarchikus szerkezethez, melyet az ábrán piros vonalakkal jelöltünk. A zölddel jelölt pontok az egyes szülőket jelölik. Jól látszik, hogy a levelek a szülőik alapján csoportosítva vannak. Továbbá az is megfigyelhető, hogy a hierarchia egyes szintjei a kör középpontja felé haladó koncentrikus köröket alkotnak, és a kör középpontjában található a gyökér. Ezen az ábrán még jobban látszik az, hogy egy görbe minél közelebb halad a kör középpontjához, annál távolabbi kapcsolatban állnak az adott elemek egymással. Hogy ez a távolság mit jelent (jót/rosszat), az alkalmazási területenként változó lehet.

A görbékkel való ábrázolás átláthatóbbá teszi a képet, de nagy számú kapcsolat esetén ujjunkkal nem tudjuk végigkövetni a vonal útját, így csak sejtethetjük, hogy melyik másik elemhez kapcsolódhat. Ezekre találhatunk megoldást az interneten[5][6], de ezek interaktív vizualizációs szempontból nem tökéletesek. Olyan példát nem találtunk, amely animációt használ ezen szerkezeteknek a megjelenítése során.

A program

Hierarchikus szerkezetek ábrázolására eddig is sok módszer létezett. Ezeknek mindnek más és más volt a célja és rendeltetése. A ma használatos gráfok az esetek nagy részében statikusak, vagy az interakció csak a navigációs lehetőségekre vonatkozik. A megjelenített adat minél rövidebb idő alatt történő megértéséhez nem elegendő ha egy hierarchiát kellően nagy felbontásban tudunk ábrázolni. A felhasználót végig kell vezetni a hierarchián, és minden egyes időpillanatban meg kell neki mutatni, hogy mi az a fontos részlet, amire éppen figyelnie kell.



5.1. ábra. A program az összes kapcsolattal. (tension: 0,85)

Az általunk használt algoritmus nem magát a hierarchiát, hanem egy hierarchián alapuló rendszer kapcsolatait ábrázolja (5.1. ábra). A módszer neve: *Hierarchical Edge Bundling* [7].

A módszert, miszerint a kapcsolódó elemeket nem egyenesekkel, hanem valamilyen görbékkel kötjük össze, több tudományterület is alkalmazza mint vizualizációs eljárást. Az egyik ilyen például az ontológiák vizualizációja[8][12].

5.1. Technológiai háttér

Egy program fejlesztése előtt, meg kell vizsgálni az alkalmazás célcsoportját és annak környezetét. Ez elengedhetetlen a technológiát érintő döntések meghozatalához. Olyan platformot kell választani, amelyet a felhasználók kompromisszumok nélkül tudnak használni. Mivel jelen esetben nem konkrét ügyfél kérésére készült a program, így teljes egészében rajtunk múlt ez a döntés.

Felmerült a *Java*, mivel az alkalmazás így több platformon is futtatható lenne. De nem ezt választottuk mert a java programok grafikus felülete nem a legjobb, így rontaná a felhasználói élményt. Olyan platformot kerestünk, amely megfelel az alábbi kritériumoknak:

- modern, gyorsan lehet fejleszteni benne
- letisztult kezelőfelülettel rendelkezik (vagy könnyen írható benne ilyen)
- platformfüggetlen, esetleg eszközfüggetlen is

Rövid kutatómunka után megtaláltuk a megoldást a webes alkalmazások formájában. Az angol terminológia *webapp*-nak illetve *single-page app*-nak hívja őket. Az elnevezés a működésükre utal: használatukhoz szükség van egy „segédprogramra” ami nem más mint egy „modern böngésző”¹. Manapság ez nem jelent gondot, hiszen általában minden operációs rendszer tartalmaz valamilyen böngészőt: az újabb rendszerek pedig már modernnek mondható böngészővel rendelkeznek.

A „klasszikus” weboldalak úgy működnek, hogy amikor a felhasználó kattint egy linkre (például el szeretne érni egy funkciót), akkor a böngésző küld egy kérést a szervernek, amely válaszban elküldi a teljes megjelenítendő weblapot. Ez több olyan részt is tartalmaz, ami már az előző oldalon is jelen volt, mint például a fejléc vagy a főmenü. Azzal, hogy ezeket ismét megkapja a böngésző, nő a továbbítandó

¹A modern jelző a *de facto* HTML és XHTML szabványok követésére utal. Egyes fejlesztők akkor tekintenek egy böngészőt modernnek, ha az helyesen meg tud jeleníteni egy weboldalt, amely a legfrissebb szabványokat követi. A *W3C* nemrég jelentette be, hogy a HTML5 most már hivatalosan is ajánlott: <http://www.w3.org/2014/10/html5-rec.html.en>.

adat mennyisége, ami egy lassabb kapcsolat esetén (mint manapság a legtöbb mobilinternet) több másodperces várakozást is jelenthet. Amikor böngésző megkapta az adatot, akkor azt még ki is kell renderelnie, ami az esetek többségében nem jelent gondot és manapság már alig észrevehető, de a okos eszközök elterjedésével ismét számolni kell a kisebb erőforrásokkal rendelkező kliensekkel is. A felhasználó pedig nem szeret várni se a hálózatra se a programra.

A *single-page app*ok minden esetben csak azt a részét renderelik újra az oldalnak, amelyet szükséges. Ezt úgy érik el, hogy van egy HTML váz amelybe dinamikusan töltődik be a tartalom. Hogy pontosan mi is történik a háttérben, annak leírása túlmutat jelen dolgozat célkitűzésein.

Két elterjedt keretrendszer van, amelyek segítségével ilyen modern webapp-okat írhatunk: *backbone.js*¹ és *angular.js*². Mindkettő egy JavaScript könyvtár. Mi az *angular*-t választottuk, mert egyszerűbbnek és kényelmesebbnek találtuk a használatát.

A program szempontjából még egy lényeges kérdés van: maga a vizualizációs framework. Mivel a webes alkalmazást választottuk, mindenképpen valamilyen JavaScript könyvtárat kell keresni. Grafikon rajzoló library-ből nagyon sok található. A választásunk a *d3.js*³ nevű könyvtárra esett. Ennek előnye, hogy a webes standardokra építi működését. Használatához tehát elengedhetetlen egy modern böngésző, amely (a mi estünkben legalábbis) támogatja az SVG elemeket és lehetőség szerint a CSS3-at is. A webes standardokra való építésnek további előnye, hogy ha frissülnek, akkor ezt a könyvtárat egyáltalán nem, vagy csak kis mértékben kell továbbfejleszteni ahhoz, hogy az újdonságokat használni tudjuk vele. A programom szempontjából a legnagyobb előnye a *d3js*-nek, hogy az általunk használt *edge bundling* metódus már meg van valósítva benne [2], így nem kellett annak implementációjával foglalkoznunk, hanem a dolgozat célkitűzésére, az animációra tudtunk koncentrálni.

5.2. Az adatok

Az alkalmazás egy statikus fájlból olvassa az adatokat JSON formátumban [11], de már most is aszinkron módon (AJAX) kéri le őket. Tehát technikailag semmi akadályja annak, hogy az adatokat egy szerver állítsa elő, például egy adatbázis lekéréssel, és azokon valamilyen előfeldolgozást végezzen. Preprocessing lehet például az, hogyha az adatbázisból lekért adatok nem hierarchikusak, akkor valamilyen klaszterező algoritmussal több lépésben megpróbálunk alá-fölé rendeltségi viszonyokat kialakítani, és ha ez nem sikerül akkor azt mondjuk az adathalmazra, hogy ezzel a

¹<http://backbonejs.org/>

²<https://angularjs.org/>

³<http://d3js.org/>

```

1 [
2   { "name": "flare.animate.interpolate.ArrayInterpolator",
3     "imports": [
4       {
5         "name": "flare.util.Arrays"
6       },
7       {
8         "name": "flare.animate.interpolate.Interpolator"
9       }
10    ]
11  },
12  { "name": "flare.animate.interpolate.ColorInterpolator",
13    "imports": []
14  },
15  ...
16 ]

```

5.1. kódrészlet. A megjelenítendő hierarchia adatformátuma

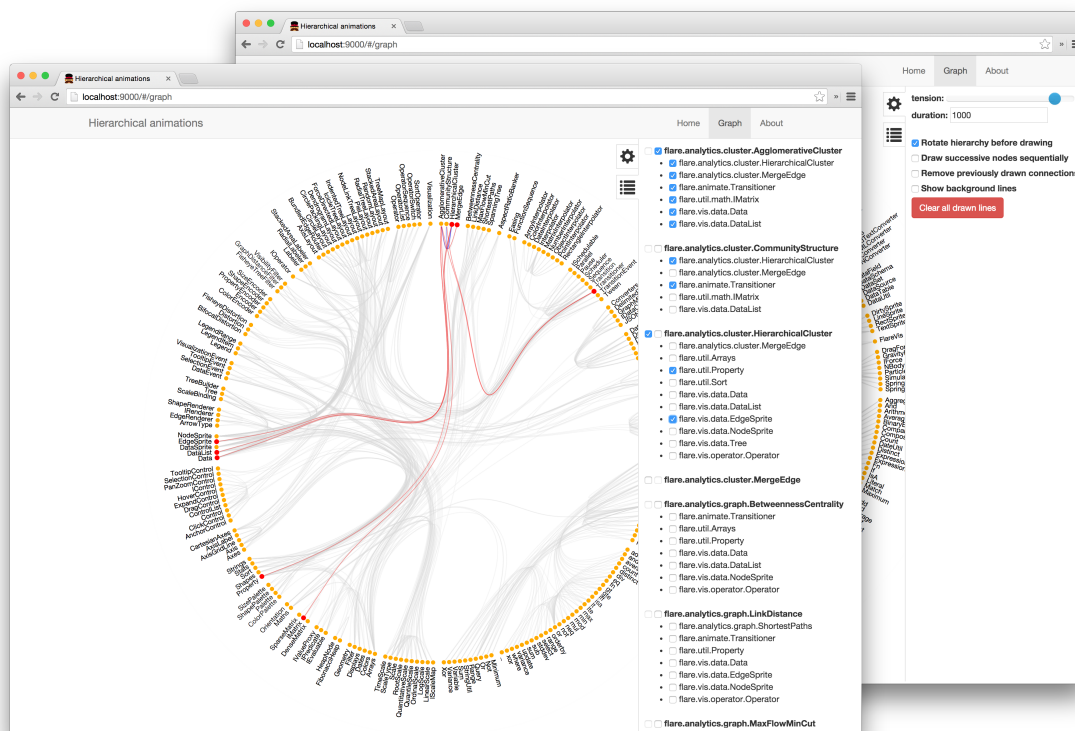
metódussal nem megjeleníthető.

A program jelenleg az 5.1. kódrészletben látható adatformátumot olvassa be. Ez egy tömb, amely objektumokat tartalmaz. Minden egyes objektumnak kötelezően rendelkeznie kell a **name** tulajdonsággal ami nem csak az azonosításra szolgál, hanem ez tartalmazza a hierarchia szerkezetére vonatkozó információt. A hierarchia egyes szintjeit „.” választja el (ez tetszőleges más karakterre cserélhető). Tekintsük a 2. sorban lévő nevet: az **ArrayInterpolator** egy levél, neki már nincs több gyereke. Az ő apja az **interpolate** elem, míg a **flare** elem a dédnagyapja. Ez utóbbi elem egyben a hierarchia gyökere is. A **name** tulajdonságon kívül minden egyes objektumban található egy **imports** tulajdonság is. Ennek értéke egy tömb, ami lehet üres is: ekkor az adott elemből nem indul ki egy kapcsolat sem. Ez természetesen nem zárja ki, hogy rendelkezzen kapcsolattal, ugyanis szerepelhet egy másik objektum **imports** tömbjében, s ezáltal lehetnek bemenő élei. Az **imports** tömb szintén objektumokat tartalmaz, amelyek ugyanúgy rendelkeznek **name** tulajdonsággal – hiszen ez azonosítja azt az objektumot amibe az él érkezni fog – viszont *nem rendelkezhet* **imports** tulajdonsággal.

A fent ismertetett adatszerkezet a minimális követelmény. Ezen felül bármelyik objektum tartalmazhat alkalmazási területre specifikus extra tulajdonságokat, mint például amik a 6.2. részben vannak leírva. Egy másik szituáció lehet az amikor például egy múltbeli hálózati szituációt akarunk szemléltetni. Ekkor a hierarchiánk elemei között a kapcsolatokat időzítési adatokkal kell ellátni: mikor jön létre a kapcsolat, mennyi idő alatt, mennyi ideig él, mennyi idő a felbontása?

5.3. Felhasználói felület

Az 5.2. ábrán látható felhasználói felület két nagy részre bontható: a jobb oldalról becsúszó beállítások panelre, és az adatmegjelenítésre. Előbbi a beállításokon kívül még egy elemlista panelt is tartalmaz amely az ábrán szereplő kapcsolatokat listázza: vastagon az élek kiindulópontja és alatta listába szedve az élek érkezési pontja.



5.2. ábra. A felhasználói felület a jobb oldalról becsúszó panelekkel. Elemlista (előtérben) és beállítások (háttérben).

A beállítások menüben fentről lefelé haladva az alábbi dolgokat tudjuk állítani:

- *tension*: értéke egy csúszkával állítható 0 és 100 között. A programban ez az érték $[0,1]$ közötti skálára van normalizálva. Ha az értéke 0, akkor a kapcsolódó elemek egyenes vonalakkal vannak összekötve. 100-as érték esetén pedig az egy irányba menő élek (közös ősökkel rendelkező elemek élei) maximálisan össze vannak vonva.
- *duration*: az animáció időtartama. Ezredmásodpercben kell megadni, hogy egy él kirajzolása mennyi ideig tartson. Az alapértelmezett értéke $1000\text{ms} = 1\text{s}$.
- *Draw successive nodes sequentially*: ha egy elemnek az összes kimenő élét ki szeretnénk rajzolni, akkor ezen opció ki- vagy bekapcsolásával érhetjük el,

hogy az egyes éleket egymás után vagy egyszerre, párhuzamosan rajzolja ki a program.

- *Remove previously drawn connections*: az opció bekapcsolásával ha kirajzoltattuk egy elem éleit, és szeretnénk egy másik elem éleit megvizsgálni, akkor az új élek kirajzolása előtt a régiek eltávolításra kerülnek így biztosítva az ábra jobb átláthatóságát.
- *Show background lines*: az 5.1. ábrán is látható összes kapcsolat megjelenítése vagy elrejtése. Ha néhány élet szeretnénk megvizsgálni, hogy merre mennek, akkor célszerű kikapcsolni a háttérrel, így jobban tudunk csak az adott élekre koncentrálni (még akkor is ha eleve ki vannak emelve piros színnel). Az 5.2. ábra előterében látszik annak a hatása, ha nem kapcsoljuk ki. A „háttér-élek” színe egészen világos szürke, és részben áttetszőek is. Ez kettős célt szolgál: ha mégis megjelenítjük őket az animáció alatt, akkor ne legyenek annyira hangsúlyosak (és ezáltal zavaróak), illetve a gócpontok – ahol sok él megy – ne összefüggő szürke területet alkossanak. Valamint egy további előnye, hogy ha távolról tekintünk az ábrára, akkor ahol intenzívebb szürkének látjuk a vonal(ak)at, ott több él van, mint ahol egészen halványak.
- *Clear all drawn lines*: ez a gomb főleg a fejlesztés és tesztelés során jön jól, amikor már sok élet kirajzoltunk, és egyszerre el akarjuk tüntetni őket.

A jól megválasztott framework a beállítások implementálása során igazolta magát. Angularban ugyanis nem kell azzal foglalkoznunk, hogyha a felhasználó megváltoztatja a lejátszás sebességét akkor elmentsük az új értéket egy változóba. Egyszerűen csak megadjuk a beviteli mezőnek, hogy melyik változó az ő modellje, és innen kezdve akár a felhasználó akár a program változtatja ezt az értéket, az mindig szinkronban lesz [18]. Mindehhez HTML-ben csak ennyit kellett megadni:

```
<input id="durationSecInput" type="text" ng-model="durationSec" />
```

utána pedig a megfelelő controllerben `$scope.durationSec` változóval tudunk az aktuális értékre hivatkozni.

A elemlista panelt részletesebben a Működés részben tárgyaljuk.

5.4. Működés

A program elindulásakor betöltődnek az alapbeállítások, és a változtatásukhoz tartozó felhasználói felületek. Ha a felhasználó szeretné jobban megérteni az adott

hierarchia kapcsolatait, akkor nem célszerű ha az összes kapcsolatot egyszerre nézi. Helyette ajánlott azokat elrejtetni, és az alkalmazás által nyújtott animációs lehetőségeket használni.



5.3. ábra. A hierarchia egy elemének kimenő (*piros*) és bemenő (*zöld*) élei az animáció után

Az animáció elindítására a körvonalon található sárga karikákra kattintással van lehetőség. Ha a felhasználó egyszer kattint a sárga karikára, akkor kirajzolódik az első kimenő él. Ha nincs kimenő éle az adott pontnak, akkor a felhasználó egy figyelmeztetést kap erről. Egy kapcsolat kirajzolása után egy újbóli kattintás a soron következő kimenő élt rajzolja ki. A kapcsolatok óramutató járásával megegyező irányban rendezettek a körvonalon található feliratok szerint, így egymás utáni kirajzolás esetén a felhasználó szemének nem kell ugrálnia, hanem *előre tudja jelezni*, hogy a következő él hova fog majd érkezni. Ha nem lennének sorba rendezve az élek, a felhasználó a hibás előrejelzése miatt kevésbé *esztétikusnak* és *élvezhetőnek* fogja találni az animációkat.

Ha a felhasználó duplán kattint egy karikára, akkor az adott csúcsból induló összes (ki- és bemenő) kapcsolatot megjelenítjük neki a következő módon:

1. A beállításoktól függően párhuzamosan kirajzolódik az összes kimenő él vagy

egymást követve rajzolódnak ki (mintha a felhasználó mindig kattintott volna még egyet a karikára)

2. Amikor az 1. lépés befejeződött, akkor az összes bemenő él egyszerre rajzolódik ki.

Amint azt már a beállításoknál láttuk, az Angular segítségével relatív könnyen lehet az adatokat szinkronban tartani. A jobb szélén található elemlista egyes elemei előtt lévő checkboxok is ilyen módon frissülnek. Azaz amikor a felhasználó kattint az ábrán, akkor a megfelelő négyzetek bepipálódnak. A vastagon szedett név a kiinduló elem ami mellett két checkbox található: az első az összes bemenő élt rajzolja ki, a másik pedig az összes kimenőt¹. A kirajzolás bármelyik felületről vezérelhető akár azokat változtatva is.

Az 5.2. kódrészletben láthatjuk hogy csupán 23 sor elegendő ahhoz, hogy a változóban tárolt elemlistát a jobb oldali panelben megjelenítsük. Ha server oldalon raknánk össze a megjelenítendő HTML kódot az sem lenne több sor, ellenben a hálózaton keresztül továbbítandó adat nagyobb lenne.

```

1 <div id="nodelist" class="pane overflow"
2   ng-class="{active: sidebarMenu.nodelist}">
3   <dl ng-repeat="source in data">
4     <dt>
5       <input type="checkbox" ng-model="source.showIncomings"
6         ng-disabled="hasImportedBy(source)"
7         ng-change="showHideIncomingLine(source)">
8       <input type="checkbox" ng-model="source.shown"
9         ng-disabled="hasImports(source)"
10        ng-change="showHideLine(source, true, source.imports)">
11       {{source.name}}
12     </dt>
13     <dd>
14       <ul>
15         <li ng-repeat="import in source.imports">
16           <input type="checkbox" ng-model="import.shown"
17             ng-change="showHideLine(source, false, [import])">
18             {{import.name}}
19         </li>
20       </ul>
21     </dd>
22   </dl>
23 </div>

```

5.2. kódrészlet. A jobb oldali elemlista generálása *ngRepeat* segítségével.

¹Ugyan egyik checkboxhoz sincs odaírva, hogy mire való, de az alkalmazás lényegi részének demonstrálása céljából ettől eltekinthetünk. Egy végleges verzióban ezt a részt át kell tervezni.

5.5. Hogyan mozog a vonal?

A kód legérdekesebb és egyben leglényegesebb része: az élek animációja. A vonalak mozgását figyelve azt hinné az ember, hogy az alkalmazás valamilyen fejlett algoritmus alapján tudja hogy a következő időpillanatban mennyit és milyen irányba kell előrehaladnia. De valójában a háttérben egy trükk húzódik meg, ugyanis valós időben számítani az éleket erőforrásigényes feladat lenne és akadozna az animáció. Ez nagyban rontaná a felhasználói élményt.

A kapcsolható háttérként szereplő vonalaknak valójában kettős szerepe van: az egyik amit már eddig is tudtunk, hogy jól mutat mint ábra, a másik pedig hogy mankóként szolgál az animációhoz.

Az 5.3. kódrészlet 3. sorában kiválasztjuk azt az élt amit animálni szeretnénk, majd a 4. sorban hozzáadjuk az ábrához az általunk írt *appendHtml()* metódussal, amelynek kódja az 5.4. kódrészleten látható. Mivel az *SVG* ábra végéhez adjuk hozzá az új elemet így az legfelül fog megjelenni és csak azok a részek takarhatják el, amelyeket később adunk hozzá az ábrához. Az 5. sor nagyon fontos: egy *SVG path*-nek a *getTotalLength()* metódusával megkapjuk a görbe teljes hosszát [15]. A 6–9. sorokban beállítjuk a vonal színét és adunk neki még néhány osztályt, hogy később majd be tudjuk azonosítani.

A fontos részt a 10–11. sorok tartalmazzák. A *path stroke-dasharray* tulajdonságának beállításával a görbe nem folytonos vonallal fog megjelenni, hanem szaggatottal. Ez a tulajdonság nem csak azt mondja meg, hogy szaggatott vonalat szeretnénk, hanem mi magunk definiáljuk benne a szaggatottság „mintáját”. Ennek

```
1 var pathSelector = 'path.from-flare_analytics_cluster_AgglomerativeCluster'
2                   + '.to-flare_animate_Transitioner.background-link';
3 var path = d3.select(pathSelector);
4 var animPath = d3.select('svg > g').appendHtml(paht[0][0].outerHTML);
5 var pathLength = animPath.getTotalLength();
6 animPath.attr('style', 'stroke: #f00')
7             .attr('class', function () {
8                 return d3.select(this).attr('class') + ' shown incoming';
9             })
10            .attr('stroke-dasharray', pathLength + ' ' + pathLength)
11            .attr('stroke-dashoffset', pathLength)
12            .transition()
13            .delay(delay)
14            .duration($scope.durationSec)
15            .attr('stroke-dashoffset', 0);
```

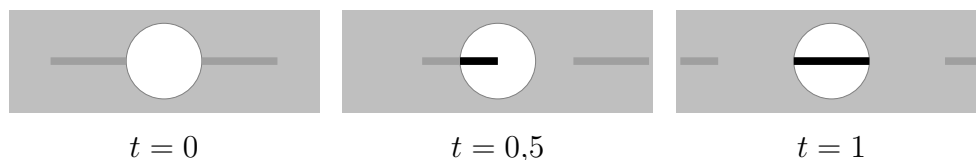
5.3. kódrészlet. Egy él animációja

```
1 d3.selection.prototype.appendHtml = function( tag ) {  
2   var $tag = $(tag);  
3   var node = this.append($tag[0].nodeName);  
4   var attrs = $tag[0].attributes;  
5   for ( var i = 0; i < attrs.length; i++ ) {  
6     node.attr( attrs[i].name, attrs[i].value );  
7   }  
8   return node;  
9 }
```

5.4. kódrészlet. A *d3.selection* objektum bővítése az *appendHtml()* metódussal

formátuma: egymástól vesszővel vagy szóközzel elválasztott listát kell megadnunk, amelynek páratlan elemei a megjelenő vonalakat, páros tagjai pedig a vonalak közötti üres helyet jelölik¹. Az általunk megadott „mintázat”: legyen egy teli rész a görbe teljes hosszában, ezt kövesse egy ugyanilyen hosszúságú üres hely, és ez ismétlődjön. A 11. sorban pedig beállítjuk a *stroke-dashoffset* tulajdonságot, amely a definiált mintázat eltolását mondja meg. Ha értéke 0, akkor a mintázat eleje a görbe elejénél kezdődik. Mi ezt eltoltuk a görbe hosszával. Képzeljük el hogy a görbénk végtelen hosszú, de csak a megadott hosszúságban látszik belőle egy bizonyos szakasz. Az eltolás után tehát a látható szakasz előtt és után is a szaggatott vonal egy teli része van, míg a látható szakasz felet pont a két teli rész közötti üres hely. Az animáció pedig úgy jön létre, hogy ezt az eltolást *\$scope.durationSec* idő alatt visszaállítjuk 0-ra. Hogy melyik időpillanatban mennyi az új eltolási érték, azt a mögöttes library számolja ki, ami jelen esetben a *D3.js*. Ezt a fent leírt folyamatot szemlélteti az 5.4. ábra, ahol az animált vonal helyzetét mutatjuk meg egy-egy időpillanatban. A szürke területre eső részek a valóságban nem látszanak, csak az ábra szemléletesebbé tétele miatt kerültek ábrázolásra. A felhasználó számára megjelenő részek azok, amik a fehér körön belül vannak.

Az 5.3. kódrészletben bemutatott animációval kiemné éleket tudunk kirajzolni. Ha bemenő élt szeretnénk animálni, akkor ugyanezt a kódrészletet használhatjuk, csupán az utolsó sorban kell átírni, hogy a másik irányba csináljuk az eltolást. Tehát a 0 helyett *2*pathLength* fog szerepelni.



5.4. ábra. Az animáció lépései

¹A „5,2,1,2” számsor például ilyen vonalat hoz létre: **— . — . — .**

```
6 animPath.attr('style', 'stroke: #f00').  
7   .attr('class', function() {  
8     return _.without(  
9       d3.select(this).attr('class').split(' '),  
10      'link'  
11    ).join(' ') + ' bullet';  
12  })  
13  .attr('stroke-width', 8)  
14  .attr('stroke-linecap', 'round')  
15  .attr('stroke-dasharray', '0.01 ' + pathLength)  
16  .attr('stroke-dashoffset', pathLength)  
17  .transition()  
18  .delay(delay)  
19  .duration($scope.durationSec)  
20  .attr('stroke-dashoffset', '0.4');
```

5.5. kódrészlet. Az él elején „utazó” kör animációja

A kimenő élek végén van egy kis piros karika, amit úgymond maga előtt tol a vonal. Első megvalósításban ezt úgy oldottuk meg, hogy rajzoltunk egy kör objektumot, és az animáció során mindig kiszámoltuk, hogy adott időpillanatban hol kellene hogy tartson a vonal, és oda mozgattuk a kört. Ez a megoldás az esetek egy részében működött, de nem egyszer megtörtént, hogy a kör lemaradt a vonal elejétől vagy akadozva, ugrálva ment. Egyértelmű volt hogy túl sok számításba telik megkapni a megfelelő koordinátát: valahogy rá kellene bízni ugyanarra a rutinra, ami a görbe animációját végzi.

Az ötletünk, hogy a vonal szaggatottságával tovább lehet játszani. Egy 8 pixel átmérőjű kört szeretnénk mozgatni. Ennek megoldása az 5.5. kódrészleten látható, amelynek eleje megegyezik az 5.3. kódrészletével. A 13. sorban beállítjuk a vonal vastagságát 8 pixelre, rögtön utána pedig a vonal csúcsat (*linecap*) kerekre. Most már könnyű látni, hogy hogy lesz ebből kör¹. A vonal hosszúságát minimálisra kell venni: 0 nem lehet, mert akkor már nem lenne megjeleníthető. A 0,01 pixeles hosszúság pont megfelelő lesz számunkra. Az animáció már mondhatni triviális dolog: a görbe hosszúságával megegyező hosszúságú üres helyet kell kihagyni a teli vonalak (jelen esetben kis golyók) között. Az animáció iránya a 20. sor változtatásával az 5.3. kódrészletnél ismertetett módon kiválasztható.

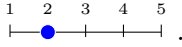
¹A helyes válasz az hogy „sehogy”, de mivel szabad szemmel körnek fog látszani, ezért mi is annak tekintjük.

5.6. Továbbfejlesztési lehetőségek

Mint az a 2. fejezetben is látszik, az információ ráültetéshez egy nagyon gazdag eszköztár áll a rendelkezésünkre a különböző színek és a formák alkalmazásával és kombinálásával. Ezt az eszköztárat gazdagítja a harmadik fejezetben megismert animáció és az interakció, amelynek az ábra feldolgozása, könnyebb megértése során jut nagyobb szerep. Ezen információráültetés finomhangolása is egy jó továbbfejlesztési irány, viszont ezt nehéz általánosan megfogalmazni: ami az egyik iparágnak előnyös, az nem biztos, hogy a másiknak is az, így ezt minden esetben az adott alkalmazási területre specifikusan kell kidolgozni.

A továbbfejlesztésre van egy sokkal érdekesebb lehetőség: nagy adathalmazok vizualizációja. A program jelen formájában csak korlátos méretű adathalmaz megjelenítésére alkalmas (nagyjából 250 levéllel rendelkező hierarchia az, ami még kényelmesen elhelyezhető egy ekkora méretű körvonalon).

Mint az a 4.2. ábrán is látszik, a Holten által kidolgozott megjelenítésnél a hierarchia egyes szintjei egyre kisebb átmérőjű koncentrikus körökön helyezkednek el. Továbbá egyértelmű, hogy a hierarchia egy magasabb szintje ugyanannyi, vagy kevesebb elemet tartalmaz, mint egy alatta lévő, így nagy az esély arra, hogy minél közelebb vagyunk a hierarchia gyökeréhez, annál kevesebb elemet kell elhelyezni a körvonalon, tehát annál valószínűbb, hogy átfedés nélkül el fognak férni egymás mellett. Az egyes koncentrikus körök közötti váltást az ábra *zoomolásával* oldhatjuk meg. Ezen technika alkalmazása nem csak a nagy adathalmazok megjelenítése során lehet hasznos, hanem akkor is, ha a felhasználó kevesebb kapcsolatot szeretne látni, vagy globálisabb összefüggések érdeklik.

A hierarchia egyes szintjei közötti navigációra (zoomolás) az alábbi módszert dolgoztuk ki: a jobb oldali beállítások panelen elhelyezünk még egy csúszkát, amely nem folytonosan, hanem egy skála alapján diszkrét értékek között állítható: . A skála alapjául a hierarchia egyes szintjei szolgálnak. A közbeékelt ábrán egy (a gyökérrel együtt) 5 szintű hierarchia csúszkája látható. Amikor a felhasználó mozgatja, akkor az eddig megjelenített ábra eltűnik, és megjelenik helyett egy újabb ábra, amely megfelelően több vagy kevesebb pontot tartalmaz.

Abban az esetben, ha a felhasználó olyan sok pontot szeretne megjeleníteni ami már nem lesz átlátható a körvonalon, akkor megkérjük arra, hogy a jobb oldalon található listából válassza ki azokat az elemeket, amelyek kapcsolatai érdeklik. Ekkor a program felrajzol egy új ábrát, csak azokkal a kapcsolatokkal, amelyeket a felhasználó kiválasztott: tehát a többi ponttal nem is kalkulálunk¹.

¹Ha például a felhasználó választásának hatására összesen 30 pont között kell csak a kapcsolatokat ábrázolni, akkor az a 30 pont egyenletesen fog elhelyezkedni a körvonalon.

Hatodik fejezet

Alkalmazási lehetőségek

Felmerül a kérdés, hogy ez a vizualizációs technika a világot behálózó minden hierarchiára alkalmazható-e? A válasz nem egyértelmű. Igen, mert minden hierarchiában találhatunk kapcsolatokat az egyes elemek között – azonkívül, hogy egy pont egy másinak gyermeke vagy éppen unokája. Illetve nem, mert vannak olyan hierarchiák, ahol magának a hierarchiának a szerkezete hordozza a fő információt, és az egyes elemek között nem lehetséges fel egyéb kapcsolat vagy annak fontossága elenyésző.

Ilyen például egy családfa, ahol a fő információt a rokoni kapcsolatok hordozzák. Természetesen tudunk egy családfán belül olyan kapcsolatokat mondani, amelyek nem a hierarchia felépítéséért felelősek (tehát nem rokonsági kapcsolatok), így ezek megjeleníthetők az eszközünkkel. De egy családfa további kérdéseket is felvet, ugyanis ha egy kutatáshoz szeretnénk eredményeket kapni mondjuk névegyezőségről, akkor a névegyezőségi kapcsolat nemcsak a legfiatalabb generáció tagjai között állhat fenn, hanem egy gyerek (a fában levél) és mondjuk dédapja között. A programunk jelenleg csak a leveleket helyezi el a körön, így ez a kapcsolat nem is lenne ábrázolható.

6.1. Informatikai alkalmazás

Mint az alkalmazás ismertetése során használt legtöbb ábrán is látszik, jól használható programcsomagok közötti függőségek és kapcsolatok ábrázolására. Vegyünk például egy java nyelven megírt programot, amelynek egy csomagja az alábbi módon néz ki:

```
package flare.analytics.cluster.AgglomerativeCluster
```

A hierarchia egyes szintjeit „.” választja el egymástól.

Az ebből kapott vizualizáció azt mutatja meg, hogy egy-egy package mennyire függ mások által írt funkcionalitástól. Ha például a szoftver egy részének változtatását tervezzük, akkor az ábra segítségével megmondhatjuk, hogy egy esetleges

hiba alkalmával ez a változtatás csupán szűk környezetére lesz kihatással, vagy akár a szoftver nagyobb részét is érintheti. Például egy adatbevitelért felelős csomagban keletkezett hiba a program nagy részére kihatással lehet, ha sok funkcionalitás megvalósításához importálni kellett az adott package-et.

De ha egy olyan „átlagosnak” mondható programot tekintünk, mint mondjuk a *Mozilla Firefox*¹ vagy a *Libre Office*² akkor láthatjuk, hogy ezek olyan sok csomagból állnak, hogy egyszerre nem is lehet kirajzolni őket, mert nem létezik akkora méretű monitor. Ebben az esetben magánál a megjelenítésnél egy sokkal nagyobb kihívást jelent az információ további, lépcsőzetes tömörítése, szem előtt tartva a különböző tömörítési szintek közötti kényelmes átjárást.

6.2. Gazdasági alkalmazás

Nemcsak informatikusok számára készíthetünk hasznos vizualizációkat ezzel a módszerrel, hanem például marketing szakemberek részére is. Vegyünk például egy web-áruházat. Az áruházban adott egy kategóriafa, amely a hierarchiát szolgáltatja számunkra. Az egyszerűség kedvéért tekintsünk egy olyan áruházat, ahol összesen 250 termék van: ez a mennyiség még kényelmesen ábrázolható egy kör mentén átlagos méretű monitort tekintve.

A webshop szeretne nagyobb forgalmat elérni. Úgy érzik, hogy meglévő vásárlóik nem ismerik a teljes termékkínálatukat, mert az sok kis kategóriába van besorolva és az alkategóriák rendszere nem minden esetben logikus. Abban viszont biztosak, hogyha a felhasználók valahogy rátalálnának ezekre a termékekre, akkor vásárolnának is belőlük. A marketinges előáll a megoldással, miszerint az oldalon alkalmazzanak *cross-sell* ajánlatokat. Ez azt jelenti, hogy minden terméknel megjelenítenek további három terméket, amelyek véleményük szerint érdekelhetik a vásárlót. A mi rendszerünkben a hierarchia elemei (termékek) között ezek lesznek az ábrázolandó kapcsolatok.

Amikor már sok adat összegyűlt, akkor a vállalatnál szerettek volna következtetéseket levonni, hogy jó döntés volt-e ez a fejlesztés illetve van-e olyan eset, ahol nem megfelelő terméket ajánlanak. Látva az adatok mennyiségét, túlságosan soknak és átláthatatlannak találták azt. Ezen a ponton van hasznunkra a vizualizáció.

Rendszerünk ezekből az adatokból egy olyan kimutatást tud készíteni, ahol az élek az ajánlatokat jelölik, az élek színei és vastagsága pedig az ajánlat sikerességét. Egy hőtésképszerű színskála alapján a vonal színe az alapján változik, hogy hányan

¹https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Source_Code/Downloading_Source_Archives

²<http://www.libreoffice.org/about-us/source-code/>

kattintottak az adott ajánlatra. Például a piros azt jelenti hogy nagyon sokan, a kék pedig hogy kevesen. Továbbá minél vastagabban van kirajzolva a vonal, annál több vásárlás történt arra az ajánlatra kattintva. Ha pedig egyáltalán nem történt vásárlás csak kattintás, akkor azt jelölhetjük szaggatott vonallal. Ezekenkívül a vonal görbülete és vonalvezetése azt is elárulja nekünk, hogy a kapcsolt termék mennyire esik közeli vagy távoli kategóriába. Ha például egy termékre sokan kattintanak, és sokan is vásárolnak belőle (vastag piros vonal), de mondjuk egy másik gyökér kategóriába tartozik, akkor érdemes lehet átgondolni, hogy nem szükséges-e a terméket új kategóriába helyezni: közelebb ahhoz a termékhez, amelyhez kapcsolódik.

6.3. Mi mindenre jó még?

Mi alapján döntsük el, hogy mire alkalmazható ez a vizualizációs eljárás? Minden esetben adott egy nagy adathalmaz. Most szerencsére nem azt kell eldöntenünk, hogy hogyan jelenítsük meg az adatot, ugyanis az adat információvá alakítása kötöttebb folyamat. Adott, hogy módszerünk milyen adatszerkezeteket tud megjeleníteni: tehát a feladat az adat ilyen formára alakítása [13]. Ez a folyamat több lépésből is állhat. Célszerű minden egyes lépés után egy gyors ellenőrzést végezni, hogy a kapott adatszerkezet továbbra is azt írja le, amit az eredeti adatbázis. Amennyiben ez nem így van (például információvesztés történt), akkor vissza kell lépni egy lépést, és más módon transzformálni az adatot. Nem minden adat alakítható át a megfelelő formára.

Ha rendelkezésünkre áll az információ, akkor már csak arra kell figyelniünk, hogy a vizualizációnk megfeleljen az *Edward Tufte The Visual Display of Quantitative Information* című könyvében található alapszabályoknak [17]. Ezek következők:

- *Grafikai kiválóság (Graphical Excellence)*: a néző számára a gondolatok lehető legnagyobb részét kell átadni az ábrán keresztül a legrövidebb idő alatt minimális tinta felhasználásával minél kisebb területet használva.
- *Grafikai sértetlenség (Graphical Integrity)*: az ábra semmiképpen sem szabad hogy torzítsa az adatot, vagy hamis benyomást keltsen arról.
- *Az adat tinta arány maximalizálása (Maximize the data-ink ratio)*:

$$\text{adat-tinta arány} = \frac{\text{adatrajzolásra használt tinta mennyisége}}{\text{összesen felhasznált tinta mennyisége}}$$

Ezt úgy érhetjük el, hogy kerüljük az ismétlődő és felesleges információk megjelenítését a képen.

- *Esztétika (Aesthetics)*: Tufte véleménye szerint egy ábra akkor nevezhető elegánsnak, ha az adat komplexitása igazodik az ábrázolásmód egyszerűségéhez.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani **Dombi József** tanár úrnak, hogy tanácsaival és útmutatásával segítette munkámat.

Irodalomjegyzék

- [1] Jacques Bertin. *Semiology of Graphics – diagrams, networks, maps*. Ersi Press, 2011 (cit. on p. 10).
- [2] D. Holten algoritmus a D3.js könyvtárban. GitHub. URL: <https://github.com/mbostock/d3/blob/master/src/layout/bundle.js> (cit. on p. 20).
- [3] *Evolúcióelmélet*. Wikipedia. URL: <http://hu.wikipedia.org/wiki/Evol%C3%BAci%C3%B3> (cit. on p. 4).
- [4] Bakos Ferenc. *Idegen szavak és kifejezések szótara*. Akadémiai Kiadó, 2013, p. 184 (cit. on p. 4).
- [5] *Hierarchical Edge Bundling in D3js, v1*. bl.ocks.org. URL: <http://bl.ocks.org/mbostock/raw/1044242/> (cit. on p. 17).
- [6] *Hierarchical Edge Bundling in D3js, v2*. bl.ocks.org. URL: <http://mbostock.github.io/d3/talk/20111116/bundle.html> (cit. on p. 17).
- [7] D. Holten. ?Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data? In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006), pp. 741–748 (cit. on pp. 16, 19).
- [8] Walter Hop, Sven de Ridder, Flavius Frasincar, and Frederik Hogenboom. ?Using Hierarchical Edge Bundles to Visualize Complex Ontologies in GLOW? In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. Trento, Italy: ACM, 2012, pp. 304–311 (cit. on p. 19).
- [9] Johannes Itten. *The Art of Color: The Subjective Experience and Objective Rationale of Color*. Wiley, 1974 (cit. on p. 9).
- [10] Schanda János. *Szín és észlelet. Színtervezés számítógépes felhasználás számára*. Typotex, 2011 (cit. on p. 6).
- [11] *JSON formátum*. URL: <http://json.org/> (cit. on p. 20).

-
- [12] Simone Kriglstein and Günter Wallner. ?Knooks - A Visualization Approach for OWL Lite Ontologies? In: *2010 International Conference on Complex, Intelligent and Software Intensive Systems*. Krakow: IEEE, 2010, pp. 950–955 (cit. on p. 19).
 - [13] Riccardo Mazza. *Intorduction to Information Visualizaztion*. Springer, 2009 (cit. on p. 32).
 - [14] R.E. Meisner, M. Bittner, and S.W. Dech. ?Computer animation of remote sensing-based time series data sets? In: *IEEE Transactions on Geoscience and Remote Sensing* 37.2 (1999), pp. 1100–1106 (cit. on p. 13).
 - [15] *SVG Paths Documentation*. W3C. URL: http://www.w3.org/TR/SVG/paths.html#__svg__SVGPathElement__getTotalLength (cit. on p. 26).
 - [16] T. Tekusova and J. Kohlhammer. ?Applying Animation to the Visual Analysis of Financial Time-Dependent Data? In: *Information Visualization, 11th International Conference*. Zurich: IEEE, 2007, pp. 101–108 (cit. on p. 13).
 - [17] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1987 (cit. on p. 32).
 - [18] *Two Way Data Bindig in Angular Templates*. Angularjs. URL: <https://docs.angularjs.org/guide/databinding> (cit. on p. 23).