A picture containing text, emblem, symbol, badge

Description automatically generated

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai Kar

Média- és Oktatásinfromatika Tanszék

Interaktív szerkesztő felület dataflow gráfhoz

Tatai Áron Péter

*Programtervező Informatikus BSc*

Dr. Illés Zoltán,  
*oktató, habilitált egyetemi docens*

Balogh György,   
*CTO*

Budapest, 2023

**Tartalomjegyzék**

[1. Bevezetés 1](#_Toc135499444)

[1.1. Egy dataflow gráfszerkesztő 2](#_Toc135499445)

[1.2. A dataflow és a gráfszerkesztő elemei 2](#_Toc135499446)

[2. Felhasználói Dokumentáció 4](#_Toc135499447)

[2.1. Gráf szerkesztése 5](#_Toc135499448)

[2.1.1. Node konfigurációja 7](#_Toc135499449)

[2.1.2. Mentés és Betöltés 7](#_Toc135499450)

[2.1.3. Segítő funkciók 8](#_Toc135499451)

[2.2. Node group-ok váltása 9](#_Toc135499452)

[2.3. Hibakezelés 10](#_Toc135499453)

[2.4. Gráfgenerálás GPT-4 segítségével 11](#_Toc135499454)

[2.5. Az alkalmazás elérése 12](#_Toc135499455)

[2.5.1. A GPT-4 middleware telepítése 14](#_Toc135499456)

[3. Fejlesztői Dokumentáció 16](#_Toc135499457)

[3.1. Rajzoló Technológia 18](#_Toc135499458)

[3.1.1. Rajzoló Technológia Kiválasztása 18](#_Toc135499459)

[3.1.2. Mozgatás, Nagyítás és Pásztázás SVG-vel 22](#_Toc135499460)

[3.2. Dinamikus node definíciók 23](#_Toc135499461)

[3.2.1. Űrlap Generálás 25](#_Toc135499462)

[3.2.2. Node Connectionök definíciója 25](#_Toc135499463)

[3.3. Adattárolás 26](#_Toc135499464)

[3.3.1. Állapottér, Visszavonás és Újra 26](#_Toc135499465)

[3.3.2. Szerializálás és Perzisztálás 27](#_Toc135499466)

[3.4. A GPT-4 csatlakozás működése 31](#_Toc135499467)

[3.5. Programozási szempontok, algoritmusok 32](#_Toc135499468)

[3.6. Technológiai Kitekintés 34](#_Toc135499469)

[3.7. Tesztelés 35](#_Toc135499470)

[4. Összefoglalás 38](#_Toc135499471)

[4.1. Továbbfejlesztési Lehetőségek 38](#_Toc135499472)

[5. Függelék 41](#_Toc135499473)

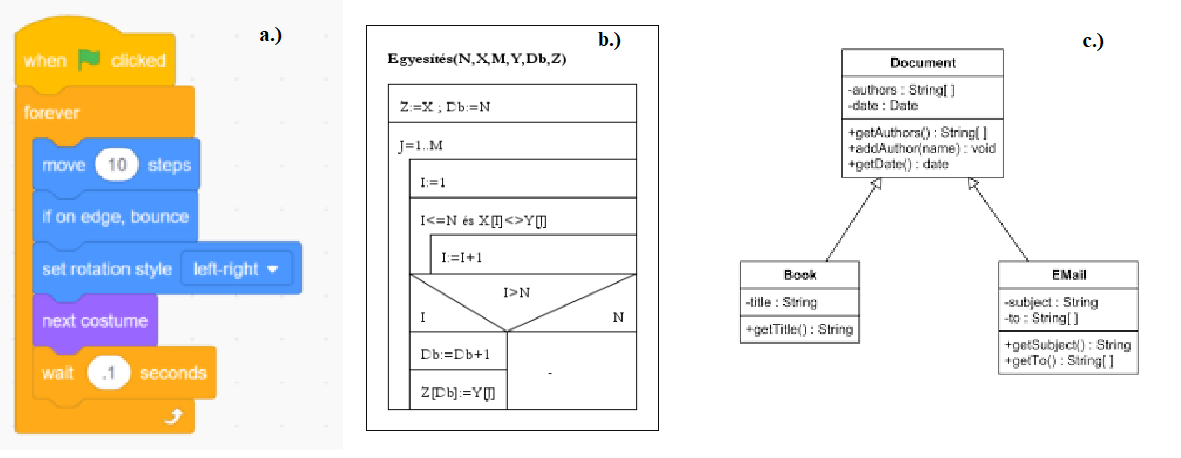
[5.1. Rövidítések feloldása 41](#_Toc135499474)

[5.2. Telepítési útmutató 42](#_Toc135499475)

[**Irodalomjegyzék** 44](#_Toc135499476)

1. Bevezetés

A programok működésének vizualizációja a ’90-es évektől kezdve fontos szerepet játszik a kód tanításában, dokumentációjában és tesztelésében. A Unified Modeling Language (UML) például széleskörűen használatos nagyobb architektúrák felépítésének vizualizációjára. Az ELTE Informatikai Karon tanított *struktogram[[1]](#footnote-2)* egy pszeudokódot vizuálisan ábrázoló megoldás. Népszerű programozást tanító nyelvek, mint a *Scratch* már tényleges futtatható kódot állítanak össze, amelyben a kódblokkokat vizuálisan lehet szerkeszteni. Ezekre a megoldásokra példát az 1. ábra mutat. Az informatika térnyerésével olyan szakembereknek is kell kódot írniuk vagy módosítaniuk, akik alapvetően nem informatikai háttérrel rendelkeznek, erre elterjedt megoldás a mérnökök körében a *LabView*.

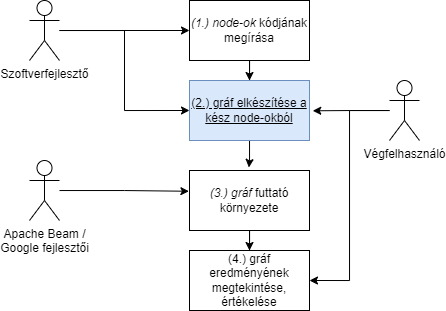


1. ábra, egyes kód vizualizációs környezetek. Balról jobbra: Scratch, Struktogram és UML. A Scratch kód közvetlenül futtatható, a Struktogramból generálható program, azonban az UML osztálydiagram csak magas szintű definíciót nyújt, kód nem generálható belőle.

A kód egységek között erős izoláció található, az egyes egységek egymásra általában hatásmentesek. Az animáció és videó iparában az ún. *node editorok* alapvető eszköznek számítanak, egyes célszoftverek, mint például a *Blender* nagyban támaszkodnak a gráfszerkesztő funkciójára. A *GStreamer* videó multimédia framework is node-okkal valósítja meg a videó kódolás elemeit. Ennek ellenére, a dataflow mint programozási megoldás kevésbé elterjedt a hagyományos deklaratív, OOP és funkcionális megoldások mellett a programozási iparban. 2016-tól kezdődően azonban több nagy piaci szereplő is publikált olyan környezetet, amely a dataflow modellt alkalmazza: a *Google TensorFlow,* az *Apache Beam* és a *Microsoft .NET TPL[[2]](#footnote-3)* is ezt a modellt hasznosítja. Ezek a megoldások általában nagy adatmennyiség gyors feldolgozást célozzák meg.

* 1. Egy dataflow gráfszerkesztő

A dataflow definíciójából adódóan jól vizualizálható gráfként. Léteznek node editorok a piacon, azonban ezek többnyire általános szerkesztők, nem kifejezetten dataflow-ra specializáltak. A Google a saját termékéhez készített szerkesztőt, azonban ez korlátozott funkciókkal bír és zárt forráskódú. **Szakdolgozatomban egy általános dataflow környezetekre igazított node editort készítek el.**

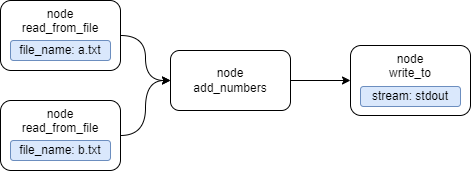


2. Dataflow program actor diagramja, kiemelve a szakdolgozatban megvalósított elemmel.

Az 1. -es ábra egy dataflow program a felhasználási lépéseit mutatja: A (2.)-es pont az hagyományosan egy JavaScript Object Notation (JSON) file szerkesztésével történik, amiben könnyű hibát véteni és programozni nem értők számára nehézkes szerkesztést eredményez. A szakdolgozatban erre a hiányzó láncszemre készítek egy megoldást.

* 1. A dataflow és a gráfszerkesztő elemei

Egy dataflow gráf egy irányított körmentes gráf (DAG)*,* amiben az élek (*line*-ok) adatot vagy adatokat szállítanak, a csúcsok – ún. node­-ok - pedig ezeket az adatokat feldolgozzák. Egy node általában egy funkciót képvisel egy programban, ami lehet akár egy összeadás művelet is, de általánosságban bonyolultabb: egy rendezés, egy SQL join, vagy az adatfolyamon egy ablak létrehozása. Ezek a node-ok egymás után fűzéséből alakul ki a végleges program, erre egyszerű példát a 2. ábra mutat.



3. ábra Példa: egy egyszerű gráf, ahol két file tartalmát olvassuk be soronként, és a bennük lévő számokat soronként összeadjuk, majd kiírjuk a standard outputra. A kék keretes elemek a felhasználó által módosítható paraméterek.

A node-oknak a fejlesztő definiálhat tulajdonságokat, amik a gráf szerkesztésekor állítható értékek. A előbbi ábrán egy *read\_from\_file* node-nál érdemes az olvasott file nevét node tulajdonságaként definiálni.

A szakdolgozatban elkészült felület nem egy konkrét dataflow implementációt, hanem általános elveket követ, így rugalmasan személyre szabható[[3]](#footnote-4).

A programot opcionálisan lehetséges használni az ChatGPT-vel párosítva: A felhasználó az elkészítendő gráfot szabadszavasan is megfogalmazhatja, és a GPT-4 segítségével ebből egy kész gráfot rajzolok. Ezzel tehát a szakdolgozatban elkészített weboldalon azon túl, hogy a felhasználónak nem kell egy JSON dataflow-t megírnia, elég csak angolul megfogalmaznia az elkészítendő gráfot, és az AI asszisztens elkészít ebből egy kész gráfot, amit a felhasználó utána tovább tud szerkeszteni.

1. Felhasználói Dokumentáció

TODO TODO Tehát az elkészült alkalmazást egy felhasználó a következőkre **nem tudja használni:**

* Programkódot írni. A gráfszerkesztő nem egy programozási nyelv, nincsenek benne beépített *if, throw, switch* utasítások[[4]](#footnote-5), a control flow kizárólag adatfolyamként működik.
* A gráf kódját futtatni. A szerkesztő csak vizualizációt és validációt nyújt, a futtatáshoz az Apache Beam vagy hasonló dataflow futtató szűkséges.
* A gráf futtatásának kimenetét vizualizálni.

Ezek tudatosan meghozott korlátozások részemről. Sok fejlesztő számára ugyanis kényelmetlen egy teljesen vizuális programozási környezet – mint a LabView - használni, a folyamat többi lépésére pedig kiforrott eszközök állnak rendelkezésre, ezekre fókuszálva kevesebb egyedi értéket tudna nyújtani a szakdolgozatom.

TODO an intro

Az alkalmazásnak két fő célfelhasználói csoportja van.

* **A fejlesztő**: Maga a dataflow-ban használt *node-*ok fejlesztője; ő definiálja a létrehozható node típusok-okat. A definícióra jelenleg erre nincs automatizált mód, ennek menetét a fejlesztői dokumentációban tárgyalom.
* **A felhasználó:** A már definiált node típusokból létrehozza a gráfot, amit utána egy dataflow-ba tud exportálni. A felhasználói dokumentáció (és maga a szerkesztő is) ennek az *actornak* a lehetséges lépéseire, a gráf létrehozása és szerkesztésére fókuszál.

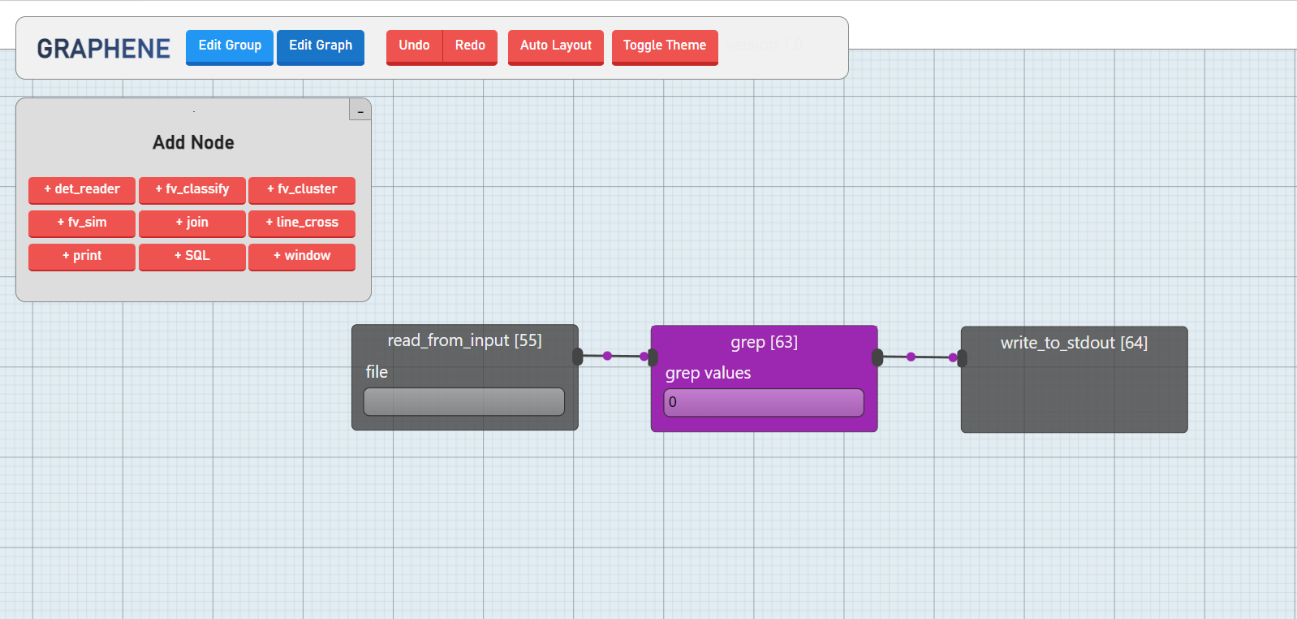
A felület [elérhető online](https://rontap.github.io/thesis/) vagy letölthető GitHubról [TODO]. A felület forráskódját manuálisan is fordítható, ezt Az alkalmazás telepítése, futtatása fejezetben tárgyalom. Mivel a szakdolgozat programja egy weboldal, ezért asztali gépen Chrome böngészővel azonnal megnyitható.

A felhasználói felület három fő elemből áll, ezért a felhasználói dokumentációt is ezek az elemek szerint tagoltam.

* 1. Gráf szerkesztése

Az alkalmazás elsődleges célja az, hogy a felhasználók könnyen és gyorsan tudjanak speciális gráfokat szerkeszteni, és a gráfok node-jain módosításokat végezni. Az alábbiakban lépésről lépésre szemléletem egy ilyen gráf létrehozását, és módosítását.

A felületen az *Add Node* ablakon belül lehet hozzáadni kívánt típusú node-okat (4. ábra. a.). Ezekre a gombokra kattintva lehelyeződik egy új node (4. ábra e.). Ez a node mozgatható a vásznon a bal egérgom lenyomva tartásával.



d.)

e.)

a.)

c.)

b.)

4. ábra A általam elkészített gráfszerkesztő felület, és egyes elemei. Példának egy olyan gráf van beállítva, ami file-ok-ból olvas, és azon egy grep-et végez el, majd

A legtöbb node-nak van két oldalán egy-egy ovális csatlakozó pont, ezek a csatlakozó pontok az élekhez. Két node között élt létrehozni a következőképp lehet: A *source* node kimeneti (4. ábra b.), azaz jobb oldalán lévő oválisra (továbbiakban *output connection*) kattintva megjelenik egy virtuális, sárga színű él. Ezt lehet összekötni egy *target* node bal oldalán lévő (4. ábra c.) oválisra való kattintással (továbbiakban *input* *connection*). Egy félkész élt jobb egérgombbal való kattintással lehet visszavonni. Amennyiben sikeres az összekötés, egy fehér színű él jelenik meg, amiben vizualizációt szolgáló kék körök haladnak az adat haladásának irányának megfelelően (4. ábra d.). Így létrejött a csatlakozás a source és a target node között.

Léteznek olyan node-ok, amiknek csak input, vagy csak output connectionjeik vannak, ilyenre példa a 4. ábra read\_from\_input és write\_to\_stdout node-ja.

A hagyományos node szerkesztőkkel ellentétben, itt az élekben lévő adatok nem külön *input és output connection*-ként jelenik meg, hanem egy külön ablakban tekinthetőek meg. Ennek az előnye az, hogy kevés node, de sok connection számnál leegyszerűsíti a gráf vizuális bonyolultságát, és ezzel a felhasználó általi elkészítési időt. Ez a *properties* ablak, amit a node­-ra való kattintással lehet fókuszálni.

[ todo itt egy kép arról, hogy milyen lenne ez.]

Az adott node-nak a szűkséges bemeneti paraméterek a properties ablakban jelennek meg. Itt csoportosítva vannak az alábbiak szerint:

* *Missing Inputs*: azon input connectionök, amik nincsenek hiányzonak. Ebben az esetben a gráf nem teljes, és nem tekinthető futtathatónak.
* *Connected Inputs*: azon input connectionök, amik csatlakoztatva vannak és az aktuálisan kijelölt *node* ezeket az értékeket felhasználja.
* *Other Inputs*: azon inputok – és egyben outputok, amik nincsenek felhasználva, csak áthaladnak a node-on. Alapvető viselkedés az, hogy egy node az összes értéket, amit megkap, de nem szűkséges neki, változatlanul továbbadja.
* *Outputs:* azon outputok, amiket a node számít ki. Ezek az értékek továbbvezethetőek ebből a node-ból[[5]](#footnote-6).

Egy-egy in out value-nak látható a típusa (azaz, hogy a dataflow-ban milyen típusú a be és kimeneti érték). Ez a típus szabadszavasan definiált, a részletes struktúrát a szerkeszőt nem ismeri. Az értéke validálásra lehet felhasználni. [todo példa képpel hibás értékre]

Egy connection lehet opcionális, ebben az esetben az *optional* szöveg jelenik meg. Lehet egyedi (key) típusú, ekkor a *unique* szöveg jelenik meg.

Node és line is eltávolítható az arra való kattintással, majd a *delete* gomb megnyomásával. A Node esetén ilyenkor az összes csatlakozott line is törlésre kerül. Node ezen kívül eltávolítható még az adott node fejlécére való jobb egérgomb kattintással, és az ott megjelenő menüben a *Delete node* gomb megnyomásával.

* + 1. Node konfigurációja

A legtöbb node-nak egyes paraméterei állíthatóak a szerkesztő felületen is. Amennyiben egy node-nak van szerkeszthető config-ja, a config mezői a node-ban megjelennek, a felhasználó által szerkeszthetőek. A felületen jelenleg elérhető különböző bemeneti típusok:

* Egyszerű szöveges beviteli mező
* Szám beviteli mező
* JSON input beviteli mező
* Textarea (többsoros) beviteli mező
* Checkbox beviteli mező
* Rejtett mező, ami a felületen nem jelenik meg, de a konfigurációban megmarad

A Továbbfejlesztési Lehetőségek fejezetben kifejtem, hogy ezekhez a típusokhoz tetszőleges plusz típust hozzá lehet adni.

* + 1. Mentés és Betöltés

A szerkesztett gráf elmenthető két formátumban. Az első formátum egy *szerializált* változata az gráfnak JSON formátumban menthető a *Save JSON* gomb megnyomásával, amit utána az alkalmazás vissza is tud tölteni a *Paste JSON code* gomb megnyomásával.

A másik elmenthető formátum az SVG. A gráfszerkesztő alkalmazásoknak nincsen standardizált formátuma emiatt, általában a gráf megjelenítéséhez kell az alkalmazás használata. Sőt, a legtöbb általános alkalmazásnak is saját, egyedi formátuma van. Ezzel szemben, a szakdolgozatom egyik célja a felhasználóbarát megjelenítés és szerkesztés. Ezért a kiexportált SVG file-t böngészők meg tudják nyitni, és a gráf elmentett állapotát megjeleníteni. Tehát a megtekintéshez nem szűkséges internet, vagy az alkalmazás használata[[6]](#footnote-7). A gráf újbóli szerkesztéséhez ezt az SVG-t az alkalmazásba be lehet tölteni a *Paste SVG code* gomb megnyomásával.

* + 1. Segítő funkciók

A felhasználó élmény elősegítésre számos segítő funkció érhető el, ezeket röviden ismertetem.

**Zoom & Mozgatás:** A vászon, amin a node-ok ki vannak rajzolva, mozgatható a bal egérgomb lenyomva tartásával és egér mozgatásával. Az egérgörgővel lehetséges ki- és bezoomolni a vászonra. A zoom eredeti méretét helyreállítani a bal alsó sarokban lévő aktuális zoom méretet kijelző gombbal való kattintással lehet.

**Undo & Redo:** Az alkalmazásban teljesértékű visszavonás és újra funkciók érhetőek el. Strukturális változtatások a node-ok és line-okkal kapcsolatban, illetve vizuális változtatások (mint egy node áthelyezése) is el vannak mentve.

**Auto Layout:** A gráf JSON-ban elmentett változata nem tartalmaz információt a gráf vizuális elrendezéséről, hiszen a dataflow rendszerek számára ez nem egy hasznos információ. A könnyebb kezelhetőség érdekében azonban az importálás esetén a felület automatikusan elrendezi a felület a node-okat. Szerkesztés közben is a *Auto Layout* gomb megnyomásával újra rendeződnek node-ok vizuálisan.

**Jobbklikk Menü:** Az egér jobb klikkje kontextustól függően más-más menüt jelenít meg. Üres területen jobb kattintással gyorsan hozzá lehet adni node-okat. Egy node-ra jobb kattintással törölni lehet azt.

**Téma:** A felület alapvetően egy sötét témát használ. Nyomtatáshoz és felhasználó preferencia estén elérhető egy világos téma is. Ez a *Toggle Theme* gombbal változtatható.

**Drag’n’Drop filefeltöltés:** A felületre gyorsan fel lehet tölteni a korábban lementett SVG vagy JSON file-okat úgy, hogy a file-t behúzzuk a *Drag an SVG or JSON file here to load.* feliratú területre.

**Ablakok elrejtése:** Az alkalmazás különböző funkciói, mint az Importálás & Exporálás és a node hozzáadás egy mozgatható ablakban található. Ezeket az ablakokat a felső pöttyös sort megfogva lehet áthelyezni. Igény szerint minimalizálni is lehet ezeket az ablakokat: a minimalizálandó ablak jobb felső sarokban található *„\_”* gomb segítségével. Minimalizálás után az az alkalmazás jobb felső sarkában megjelenik egy gomb, amivel vissza lehet állítani az ablakot.

* 1. Node group-ok

A felületen nem csak egy fajta, hanem tetszőleges számú különböző gráf típust el lehet készíteni. Ezek között a gráf típusok (node group-ok) között a szerkesztő alkalmazásban különböző között könnyen lehet váltani. A *Edit Group* gombra kattintva a felhasználó megtekintheti az elérhető node group-okat. A node group-okat a fejlesztő adja hozzá, így több típusú gráfot is el lehet készíteni egy frontend segítségével. Az Edit Group felületen lehet váltani a node group-ok között a bal oldalon található *Node Groups*-ban található gombok közül az egyikre kattintással. Ha ezután a felhasználó rákattint az *Edit Graph*-ra, akkor a felületen megváltoznak a hozzáadható node-ok, már az új felhasználói esethez tud gráfokat készíteni.

A node group nézetben a következő ellenőrző funkciók érhetőek még el: Egy node blueprint-re kattintva megnézhető annak a forráskódja: ez hasznos a fejlesztő actornak amikor összeállítja a node-okat. Az egyik blueprintre kattintva tudja ellenőrizni, hogy a node típus rendesen betölt-e, és az igénye szerint rajzolódik-e ki. Ha a node kódja hibásan van megírva, erre hibaüzenet figyelmeztet.

A felület egy gráf JSON-be mentéskor elmenti, hogy melyik node group volt szerkesztve[[7]](#footnote-8), és a betöltéskor próbálja kiválasztani azt. Ha ez az információ nincs elmentve, akkor az éppen aktuálisan kijelölt node group-ba próbálja beölteni a gráfot.

A szakdolgozatban több példa node group-ot készítettem, prezentálva pár lehetséges felhasználási területét az alkalmazásnak.  
 **ELTE IK Tantervi háló**

Az egyetemeken a tárgyak közötti kapcsolatot sokszor egy tantervi háló elkészítésével vizualizálják. Két tárgy között lehet él például, ha az egyik előfeltétele a másiknak. Egy tantervi háló lehet táblázat szerű, vagy gráf elrendezésű. Bár szigorúan véve egy tantervi háló nem dataflow, csak egy DAG, az alkalmazásom DAG-ok megjelenítésére és szerkesztésére is használható. A szakdolgozatnak az eredeti motivációját pont egy tantervi háló készítése hozta. Ebből elkészítettem egy kezdetleges változatot, azonban ez kevésbé újrafelhasználható, és vizuálisan is kevésbé flexibilis.

Készítettem egy általános tantervi hálókra szabott node group-ot, és egy konkrét, az ELTE PTI BSc 2018-as C szakirányos mintát tervéről is összeállítottam egy gráfot a szerkesztőben:

A screenshot of a computer

Description automatically generated with low confidence

5. ábra Részlet az ELTE PTI C szakirányos tantervi hálóból.

**Some Other Example**

[TODO]

* 1. Hibakezelés

Egy egyszerű JSON file szerkesztéséhez képest egy szakdolgozatban elkészülő felület egyik nagy előnye a validálás, ellenőrzés lehetősége. A felületen sok hibát el sem lehet követni, a node-ok nevét, típusát nem lehet elgépelni, a line-ok azonosítóját sem kell frissíteni manuálisan, hiszen ezeket a gráfszerkesztő kezeli.

Ezeken túl azonban a felületnek két plusz, magas szintű validálási funkciója létezik:

**Kör detektálás:** A dataflow nyelvek keretében kizárólag DAG-ok készíthetőek, így amennyiben a felhasználó kört készít, ez vizuálisan jelezve van: a kört képező vonalak piros színűek lesznek és az adat haladását reprezentáló körök sem láthatóak.

**Hiányos gráf csatlakozások:** Egyes node-ok bemenetei értékei, ha nincsenek bekötve, a gráf nem tekinthető teljesnek. Egy node-ra kattintva a *missing inputs* mezőben látszanak a még be nem kötött mezők.

**Hibás gráf:** Importáláskor a gráf ellenőrizve van, ha hibát talál a program a gráfban, akkor ezt jelzi a felhasználónak. Ha esetleg egy node-ban programozási hiba található, nem áll le az egész alkalmazás, hanem csak az a node lesz elérhetetlen. Ez lehetővé teszi az állapot elmentését esetleges szoftver hiba esetén is.

TODO

* 1. Gráfgenerálás GPT-4 segítségével

A mesterséges intelligencia mindig is megragadta az emberek fantáziáját. 2022-23-ban, olyan nagy nyelvi modellek (Large Language Model - LLM) lettek publikálva vagy frissítve, mint az OpenAI által készített GPT-3 és ChatGPT, ami alapjában megváltoztatta sok ember számára azt, hogy mire lehet képes egy ma egy AI. A ChatGPT ingyenes elérhetősége és egyszerű használata miatt azonnal beépült a köztudatba.

Annak ellenére, hogy előző években is nagymértékben használtak informatikai cégek AI-t, ezek elsősorban háttérben voltak, útvonaltervezésben vagy az árak megállapításában volt szerepük. Legközelebb a ChatGPT-hez a 2016-ben bemutatott Google Asszisztens áll, ami azonban elsősorban keresésre van optimalizálva. A ChatGPT-nek azonban a Google Asszisztenssel szemben látszólagos kreativitása van, tud verset, programkódot írni, és javítani, nagyrészt emlékszik a beszélgetésre és ismeri a kontextust. Sokak szerint ez egy inflexiós pont a mesterséges intelligenciában: sok feladatot el tudnak végezni ezek a modellek. A magyar érettségi amerikai megfelelőjén, az SAT teszten az átlagos diáknál jobban teljesít a GPT3.5-ös modell (OpenAI, 2023, old.: 4). Egyetemi környezetben AI-által írt szövegek kiszűrésére a professzorok is gyakran AI megoldásokat használnak. A GPT következő generációja, a GPT-4 jelenleg béta stádiumban van, azonban bizonyos feladatokat nagyságrendekkel jobban tud elvégezni. Ezek közé a feladatok közé tartozik egy komplex leírás alapján valid programkód vagy domain specifikus nyelv (DSL) generálása (OpenAI, 2023, old.: 91).

**Gráf generálás GPT-4-gyel DSL segítségével**

A node group-okhoz a fejlesztő actor létre tud hozni egy chatgpt.prompt file-t. Ennek a file-nak a tartalmát és szerkesztését a 5.4 GPT-4 API csatlakozás fejezetben tárgyalom. Ha létezik ilyen file, akkor a felhasználó a *Use GPT* gombra kattintva beírhat egy üzenetet. Ebben az üzenetben lehet egy kívánt gráfot megfogalmazni angolul, amiből a ChatGPT próbál majd egy tényleges gráfot legenerálni. Ilyen üzenet lehet például: „*Draw me an example graph*” vagy *„Draw a graph where each node type is used once”*. Természetesen a tényleges üzenet – prompt - az aktuális node group-tól függ, de amennyiben a fejlesztő helyesen írta meg a .prompt file-t, és a middleware fut, akkor a ChatGPT elkezdi generálni a gráfot; megjelenik alul egy ablak ahol a gráf DSL-jét lehet látni. A gráfrajzoló felület pedig, ahogy készül a gráf, próbálja kirajzolni a félkész gráfot, majd amint kész, prezentálja az AI által leírt dataflow-t.

Egy hibás üzenet beküldése esetén a ChatGPT nem DSL-el válaszol, hanem egy hibaüzenetet fogalmaz meg. A generálás nem determinisztikus, azaz nem feltétlenül generálódik ugyanaz a dataflow ugyanarra a prompt-ra. Egy jó prompt megfogalmazása több próbálkozásba telik, és a gráf node-jainak részletes ismerete hasznos.

Összefoglalva tehát, a szakdolgozatban elkészült gráfszerkesztővel nem csak a dataflow JSON file írását egyszerűsítem meg, hanem magát a gráf elkészítését is. Elég a felhasználónak a kívánt eredményt szabadszavasan megfogalmaznia, és az AI asszisztens elkészít egy kezdeti változatot belőle. Lehetséges, hogy ez elsőre nem tökéletes, azonban így is nagy segítséget nyújt a folyamatban.

* 1. Az alkalmazás elérése

Az alkalmazás legfrissebb változata elérhető a *github pages* oldalon publikálva: <https://rontap.github.io/thesis/>

A forráskód letölthető a <https://github.com/rontap/thesis> vagy git segítségével közvetlenül, a következő paranccsal.

git clone git@github.com:rontap/thesis.git

A software Windows, Linux és MacOS alatt is elindítható, platformfüggetlen. Az alkalmazás forráskódból való futtatása az átlag felhasználó számára nem szűkséges. A telepítés részletes leírását a Telepítési útmutató fejezetben tárgyalom.

**Böngészőkompatibilitás**  
Egy weboldal fejlesztése közben nagy hangsúlyt kap a böngészőkompatibilitás témaköre. Ez az eleinte a transpilereknek[[8]](#footnote-9) hála, majd a régi böngészőverziók deprekálása miatt hála egyre kevésbé hangsúlyos. A JavaScript motorok közül a *SpiderMonkey*, ami a Firefox böngészőben van használva, és a *V8* motor, ami többek között Chrome, Safari, Edge és Opera böngészőkben van használva; mindkettő motor és ráépülő böngészők hibátlanul kezeli az alkalmazás megjelenítését. Az Internet Explorer támogatása 2022-ben végetért[[9]](#footnote-10), így a szakdolgozatot nem lehet benne megtekinteni. A támogatott böngészőket az alábbi táblázatban foglalom össze.

|  |  |
| --- | --- |
| **Böngésző** | **Támogatott**? |
| Chrome, Edge, egyéb Chromium alapú böngészők | Igen, minimum 105[[10]](#footnote-11)-ös verzió |
| Mozilla Firefox | Igen, részben[[11]](#footnote-12), minimum 90-es verzió |
| Opera | Igen, minimum 90-es verzió |
| Apple Safari | Igen, minimum 16.0-ás verzió |
| Internet Explorer | Nem |

1. táblázat A szerkesztő által támogatott böngészők és verziók

A felület használatához jelenleg szűkséges egér, az alkalmazás összes képességét érintőképernyőn és csak billentyűzetes használattal nem lehet elérni.

* + 1. A GPT-4 middleware telepítése

A GPT-4 csatlakozás használatát a Gráfgenerálás GPT-4 segítségével fejezetben, míg a technikai implementációs részleteket a GPT-4 API csatlakozás fejezetben tárgyalom. Ebben a fejezetben kizárólag az API csatlakozás telepítési és elindítási útmutatója található.

Az AI segítség használatához el kell indítani egy segédprogramot. Ez teljesen opcionális, az alkalmazás összes funkciója működik ennek a modulnak az elindítása nélkül is. Fontos megjegyezni, hogy a GPT-4 a szakdolgozat írásakor még béta stádiumban van, API hozzáférést elsősorban kutatási célokkal lehet igényelni az OpenAI-tól.

A Python kód és környezet a /api mappában található. A program használatához egy python3 virtuális környezetet (venv) kell készíteni. Ez az általános python3-as környezet kialakítási lépésit követi, tehát nagyban függ a felhasználó rendszerétől, friss általános leírást online lehet találni.

A használat előtt létre kell hozni egy .env file-t ahova a megszerzett API kulcsokat kell beilleszteni. Erre példát a .env\_EXAMPLE file nyújt. Ha a környezeti változók beállítását elmulasztjuk, erre hibaüzenet emlékeztet minket.

A venv környezet függőségeit a pip csomagkezelővel lehet telepíteni, a környezet bemeneti file-ja pedig a /api/gpt4.py. Elindítani az alkalmazást a *venv\_python* ./gpt4.py paranccsal lehet, ahol a *venv\_python* értéke a virtuális környezet python futtathatója, ami platformfüggő. Windows 10-en a .\api\venv\Scripts\python.exe elérési úton található.

Amennyiben az alkalmazás sikeresen elindul, a következő üzeneteket lehet látni:

Starting GPT WS Service...  
GPT Connection works.  
WS Service is active at: ws://localhost:8765

Ezután, a felületet frissítve már lehet használni a ChatGPT-t a gráfok generálására.

**Hibakezelés**

Mivel ez a modul egy külső szolgáltatáshoz csatlakozik, több hiba is előfordulhat. Ha az első sor nem jelenik meg, akkor a python3 környezet hibásan van összeállítva, esetleg függőségek nincsenek feltelepítve.

A második sort a program a GPT segítségével generáltatja, ezzel letesztelve a hozzáférést. Ha ez hibásan jelenik meg, esetleg egyáltalán nem, akkor az alábbi gyakori hibák lehetségesek: Nem elérhető az OpenAI szervere, hibás a API key, nincs ennek az API keynek hozzáférése a GPT-4-hez, módosítást végeztek az OpenAI API-ján. Ezeknek a hibáknak mivel az eredete a szakdolgozaton kívül van, megoldást is az OpenAI ChatGPT dokumentációban lehet találni.

A harmadik hibás megjelenése a WebSocket kapcsolat hibáját jelzi. Az hibaüzenet ilyenkor részletes, általában egyértelműen rámutat a hiba okára. Amennyiben egyéb hiba miatt leállna a WebSocket, a middleware automatikusan újraindítja.

1. Fejlesztői Dokumentáció

**[TODO small intro]**

**Programozási Technológia kiválasztása**

Egy hordozható és technológiailag könnyen integrálható alkalmazás fontos szempont volt a használt technológia kiválasztásánál. Szempont volt számomra az felhasználói ergonómia és a gyors prototipizálás lehetősége is. Ezek a szempontok alapján JavaScript-ben, pontosabban TypeScript-ben készül el a szakdolgozat. A TypeScript egy *superset-e* a JavaScriptnek, azaz minden JS kód valid TS kód is[[12]](#footnote-13). A TypeScript explicit típusannotációkat vezet be, amik fordítási időben vannak ellenőrizve. A TS bevezetésével olvashatóbbá, megérthetővé és bővíthetőbbé válik a programkód, különösen libraryk esetén. (Justus Bogner, 2022)

Mivel a felhasználói felület komplex és dinamikus, az elemek kirajzolásánál a React nevű Open Source JS frontend libraryt választottam. Kiemelendő, hogy a React tényleges felhasználó felületeket nem szolgáltat: az alkalmazás design-ja és a komponensei teljes mértékben saját munkám.

Az alkalmazás a következő főbb programozási egységekből áll:

|  |  |  |  |
| --- | --- | --- | --- |
| **Komponens** | **Komponens feladat** | **Mappa** | **Fontosabb fileok** |
| Node és Line osztály | A node és line-okkal kapcsolatos funkciók osztályokba vannak szervezve. A példányosítást és törlést is ezek az osztályok végzik. | node | Node.tsx  Line.tsx  NodeFC.tsx |
| Létrehozható Node-ok betöltése | A példányosítható node-ok betöltése, és a *node group*-ok közti váltás. | node | Types.ts  Builder.ts |
| Gráfműveletek | A gráf gyökér és levél nodejainak lekérdezése, gráfbejárás és kör detektálás. | graph | GraphUtil.ts |
| Szerializáció | SVG és JSON importálás és exportálás | graph | Serialiser.ts |
| Állapottér | A teljes állapottér egy speciális adatstruktúrában való tárolása. Az összes komponens innen kérheti le az elemeket. | graph | State.ts |
| SVG és műveletek | Az SVG komponens kirajzolása, a vászon és az elemek mozgathatósága. Automatikus gráf elrendezés. Line információ. | svg | Svg.tsx  Movable.ts  Draggable.ts  Positioning.ts  InspectLine.tsx |
| UI Elemek | Újrafelhasználható UI elemek. | ui/components/ \*.tsx | |
| UI Felület | A felhasználó felület fő elemei, CSS. | ui  ui/ styles | \*.tsx |
| Node Group betöltése | A létrehozható node típusok betöltése és ezek közti váltás; típusdefiníciók. | app | DynamicReader.ts  EdgeLoader.ts |
| Segítő függvények | Vektorműveletek, alkalmazás szintű konstansok és file művelet segítő függvények. | util | const.ts  util.ts  Geom.ts |
| UI Űrlap (Form) | A node-ok belsejében megjelenő űrlap elemeit dinamikusan megjelenítő komponensek. | ui/form | FromRoot.tsx  FormRouter.tsx  atoms/\*.tsx |
| Node, Node group definíció | A betöltött Node-ok vázlata JSON formátumban. Saját adattípusok. | dynamic/groups | /\*/\*.json |
|  |  |  |  |
|  |  |  |  |

A fejlesztői dokumentációt a felhasználóihoz hasonlóan az alkalmazás fő komponensei szerint tárgyalom.

* 1. Rajzoló Technológia

Az szakdolgozatban elkészült alkalmazásnak a *value proposition*-je az, hogy egy hagyományosan nehezen érthető dataflow szöveges file-t egy interaktív felületen tudjon szerkeszteni a felhasználó. A legfontosabb döntés technológiailag tehát a rajzolási technológia a kiválasztása volt. Weben erre számos megoldás létezik.

* + 1. Rajzoló Technológia Kiválasztása

Az alábbiakban részletezem a felmerült lehetőségeket, és az egyes megoldások előnyei, hátrányait, és végül a kiválasztott technológiát.

**WebGL**

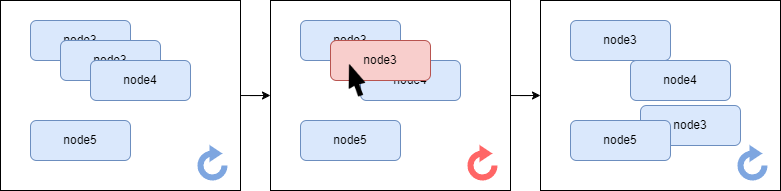
A WebGL az OpenGL weboldalakra szabott változata, amivel alacsony szintű grafikai kódot lehet írni (Caballero, 2011). Egy WebGL *renderelőt* az alapokról írni hosszas munka lett volna, akár majdnem egy egész szakdolgozatnyi feladat, így azt elvetettem.

Sok WebGL-re épülő könyvtár (library)létezik, a legnépszerűbbek közé tartozik a Three.js, GLGE és a P5.js. Ezek viszont sok olyan funkciót tartalmaznak, mint például az elemek mozgatása, amiket én szerettem volna implementálni. Célja volt még a szakdolgozatnak, hogy minél kevesebb függősége legyen, és ezeknek a libraryknak a használata azt jelentette volna, hogy a teljes rajzolás komponens egy már létező megoldásokon alapszik. Ez leszűkítette volna a más programozók általi továbbfejlesztési lehetőségeket is.

**HTML Canvas**

A web API-ban a canvasra többféle módon lehet rajzolni, ebből 2 fő kategóriájú megoldás létezik (whatwg, 2023). Az egyik értéke a fent leírt WebGL megoldás. A másik értéke, a hagyományos canvas interface a 2d, amivel egy CanvasRenderingContext2D objektumot példányosítunk.

A *2DContext* segítségével magasabb szintű kontrollt kapunk, mint a WebGL context. Shaderek helyett vonalakat, kockákat, szöveget lehet az API segítségével kirajzolni, ami nagyban megkönnyíti a fejlesztést.

Implementációs szinten a WebGL minden változásraaz egész *buffert* (az az a vászon tartalmát) újra rajzolja, viszont nem elég gyors a folyamatos újra rajzoláshoz. Kezdetben a 2DContextel kísérleteztem a szakdolgozat keretében, a különböző főbb elemeket (node-ok, vonalak, háttér) külön, egymásra rakott canvas layerekre raktam, és így szelektíven újrarajzoltam őket. **

**6. ábra** Példa: Amikor egy node-ot mozgatunk, a mozgatás kezdetekor az összes node-ot tartalmazó layer újrarajzolódik a mozgatott node nélkül. Egy másik layeren (piros) megjelenik viszont a mozgatott node. Így amikor mozgatjuk, csak azt az egyet kell újra rajzolni. Amikor a mozgatást befejezzük, visszakerül a mozgatott objektum az eredeti layer-re.

A megoldás plusz előnye, hogy a *z-index*, vagyis az elemek egymás fölött, alatti elhelyezkedése viszonylag könnyen manipulálható. Ami végül miatt elvetettem ezt a megoldást, az a bemeneti mezők, gombok hiánya. Ahhoz egyes *node*-ok tartalmát szerkeszteni lehessen helyben, egy teljes implementációt kellett volna létrehozni a bemeneti mezőkre, gombokra és checkboxokra.

**HTML DOM**

A 2DContext megoldás kontrasztjaként felmerült hagyományos HTML elemekkel felépíteni a rajzolót: Előnye, hogy elérhetőek a hagyományos bemeneti mezők, és az, hogy a kirajzolást a böngésző végzi, illetve, hogy az elemek z-index-ét dinamikusan és könnyen lehet állítani CSS segítségével.

Hátránya, hogy az node-okat összekötő vonalakra és egyéb, primitív formákra nincs szép megoldás, illetve a zoom és a mozgatás is nehézkes. Ezért a HTML DOM megoldás mellé kerestem egy másik, ezzel kompatibilis technológiát, ami ezeket jól kezeli.

**SVG + HTML DOM**

Chart

Description automatically generated with medium confidenceAz SVG (Scalable Vector Graphics) egy által specifikált vektorgrafikus képformátum. Szintaxisa XML alapú, egy element általában egy objektumot jelöl. (W3C , 2018) Első változatában egyszerű formák, szövegek elrendezésére volt alkalmas, kifejezetten a web számára kifejlesztve. A 2.0-ás verziójával azonban komplexebb képességeket is kapott, mint például az animáció lehetősége (CSS vagy SMIL[[13]](#footnote-14) segítségével), kiterjesztett CSS kompatibilitás és egy harmadik, ami miatt a szakdolgozatban ezt a megoldást választottam.

7. ábra Bezier Görbe SVG-ben

HTML-en belül lehetséges közvetlenül SVG-t elhelyezni; .css file-okban hivatkozni lehet egy SVG belső struktúrájára, JavaScripttel ezeket az elemeket szerkeszteni.

Az SVG 1.2 óta (W3C WG: Erik Dahlström, Anthony Grasso, Doug Schepers, 2008), egy SVG-n belül lehetséges a<foreignObject>element segítségével olyan objektumokat, névtereket használni, amik alapból nem részei az SVG-nek. Ilyen névtér lehet például az (X)HTML[[14]](#footnote-15). Tehát egy SVG-ben el lehet helyezni HTML-t és fordítva. Így, a jóízlés határáig lehetséges egy rekurzív láncot készíteni:

<html>  
 <body>  
 **<svg>** <foreignObject ...>  
 .. egyéb HTML kód  
 <div xmlns="http://www.w3.org/1999/xhtml">  
 **<svg>** .. egyéb SVG kód  
 <foreignObject ...>  
 .. rekurzív egymásba skatulyázás   
 </foreignObject>  
 **</svg>** </div>  
 </foreignObject>  
 **</svg>** </body>  
</html>

Erre a funkcionalitásra kifejezetten jó böngésző támogatás létezik. Ez a kompozíciója HTML-nek és SVG-nek lehetővé teszi a két megoldás legjobb elemeit ötvözni.

Az egyes előnyöket és azok implementációs részleteit külön fejezetben fejtem ki

* Az SVG-ben a mozgatás és a nagyítás egy natívan támogatott funkció
* A bemeneti mezők hagyományos HTML elemek, vagyis felhasználóbarát módon lehet a szövegbevitelt kezelni
* Egyszerű komponensek, mint vonal, *bézier* görbe SVG által könnyen kirajzolhatóak, ezek az elemek könnyen animálhatóak, és kompatibilisek HTML elemekkel.
* A látható állapot elérhető szövegként, tesztelhető és könnyen elmenthető, sőt, visszaolvasható.

Ezek a fő indokok, amik miatt a szerkesztő felületet ezzel a megoldással valósítom meg.

Hátrány ennél a megvalósításnál, hogy az SVG-n belüli elemeknél a sorrendiség az a kódban való sorrendiséggel egyezik, azaz egy mozgatott elemet nem lehet z-index-ben előre mozgatni. Így ebben a megvalósításban az épp mozgatott node lehet, hogy mozgatás közben egy másik node „mögött” van, ami egy vizuális tökéletlenség.

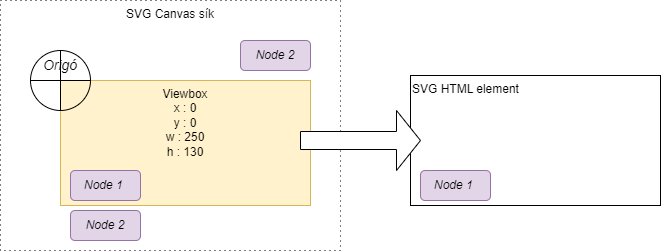
Ettől függetlenül, a fontosabb része a z-index­-nek megvalósítható a sorrendiséggel is: mindig a háttér van leghátul, utána a vonalak és legelöl a node-ok szerepelnek.

**Teljesítmény összehasonlítás**

TODO TODO TODO

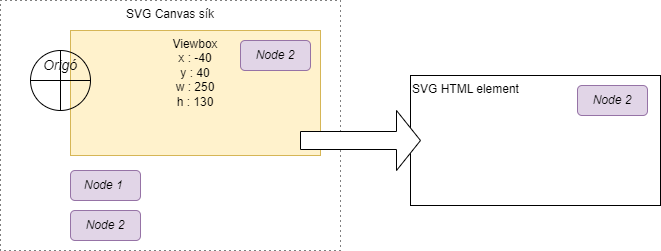
* + 1. Mozgatás, Nagyítás és Pásztázás SVG-vel

Az SVG-ben a vászon mérete végtelen x és y koordináta szerint. Mivel végtelen méretű vásznat nem tudunk megjeleníteni, ezért ezt a weboldalon megjelenítendő SVG méretét a *width, height* propertykkel tudjuk méretezni. Egy külön érték, a *viewbox* felel azért, hogy ebben az SVG ablakban pontosan a vászon melyik területét jelenítjük meg. Ennek a viewboxnak a dinamikus módosításával tudjuk a vásznat mozgatni, és rajta zoomolni. Az implementáció az /svg/Movable.ts osztályban található. Az 5. ábrán egy alaphelyzetben lévő viewbox van: A vátozót beállítom a weboldalon megjelenített elem méretére, így a lehető legtöbb tartalom látható. A Viewboxon belül van a *Node1*, így az látható a felhasználó számára is a leképezés után *(SVG HTML element az ábrán)* A *node2 és 3* azonban nem látható.



8. ábra Leképezés az SVG vászon és viewboxból a megjelenített HTML elementre.

Ha a felhasználó lenyomja a bal egérgombot és mozgatja az egeret, a viewbox x és y koordinátáját frissítem a háttérben. Egy mozgatott változat a 6. ábrán látható.



9. ábra Mozgatás után a felhasználó már a node2-t látja.

A felhasználó az egérgörgővel, vagy a bal alsó sarokban lévő + - gombokkal tud nagyítani és kicsinyíteni is. A zoomolás technikai megvalósítása a következő:

// svg viewbox visszaállítása az alapértelmezetre  
svgImageAct?.setAttribute('viewBox', Geom.viewBox(viewBox));  
// dw és dh zoom méretét szabják meg. A direction az -1 vagy +1  
// a zoom.speed pedig egy előre definiált sebesség, amivel a zoomolás // történik  
let dw = viewBox.w \* direction \* CONST.zoom.speed;  
let dh = viewBox.h \* direction \* CONST.zoom.speed;  
// mx és my értékek a kurzor jelenelegi pozíciója. Így érhető el az, hogy a zoomolás az a kurzor felé történjen  
let dx = dw \* mx / svgSize.w;  
let dy = dh \* my / svgSize.h;  
// az új viewbox kiszámolása   
viewBox = {  
 x: viewBox.x + dx,  
 y: viewBox.y + dy,  
 w: viewBox.w - dw,  
 h: viewBox.h - dh  
};

A zoomolás vizuális reprezentációját az alábbi ábrán szemléltetem:

*Diagram

Description automatically generated*

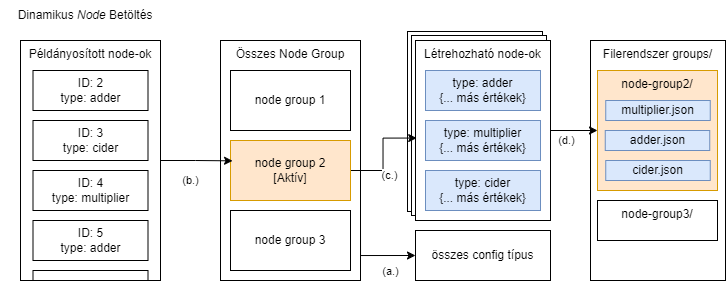
10. ábra Zoomolt és mozhatott állapot. Látszik, hogy a felhasználó számára vizuálisan nagyobbak a node-ok, bemeneti mező, gombokat beleértve. Az aspect ratio természetesen meg van őrizve a zoomolásnál és a mozgatásnál is.

* 1. Dinamikus node definíciók

Mivel nem egy konkrét dataflow gráfhoz készítem a szerkesztőt, fontos feladat volt a node típusok flexibilis létrehozása. Eredetileg erre is terveztem egy külön szerkesztőfelületet, egy „*node szerkesztő* szerkesztő” nézetet, azonban ennek hasznosságáról nem voltam meggyőződve, és ahogy tisztázódott a célközönsége az alkalmazásnak, egyértelművé vált, hogy egy node típus létrehozása ritka feladat a többi akcióhoz képest. Így a node definíciókat a fejlesztő *actornak* kell létrehoznia, JSON formátumban.

A szerkesztő képességeinek prezentálásához a node­-okat csoportosítom, ún. *node* *group*-okba szervezem[[15]](#footnote-16). A csoportok ennek a mappának az almappáiban találhatóak. Itt az almappa neve a csoportnak a neve. Ezek a csoportok között a felületen lehetséges váltani. A node group-ok az alábbá helyen találhatóak:

/app/src/dynamic/groups/nodes

A felületen egyszerre egy node group aktív, az alkalmazás rajzoló felület része nem is támogatja egyszerre több node group kirajzolását. A 2. táblázatban ábrázolom a node-ok reprezentációinak különböző szintjeit. 

2. táblázat: Az összes réteg amin a fejlesztő által létrehozott JSON file áthalad amikor a felhasználó egy új node-ot példányosít.

A előbbi ábrán lévő rövidítésék az alábbi akciókat és az alkalmazás programkódjában lévő helyet jelölik:  
(a.) - src/dynamic/groups/types.json – Az összes node group között megosztott komponens; a létrehozható config típusok konvertálása a beépített típusokra.  
(b.) - Osztály példányosítása, Node::Constructor  
(c.) – A NodeBuilder::getType segítségével. Az elérhető node típusok a NodeBuilder::types-ból vannak lekérdezve   
(d.) – Az aktuális node group tárolása NodeGroup::activeNodeGroup-ban. Az összes node group a NodeGroup::everyNodeGroupDefinition-ban van tárolva.

Az alábbiakban egy egyszerű node JSON file definíciója látható, a 2. táblázatban lévő *más értékek* kifejtése:

{  
name: "validate", // kötelezően a JSON file neve  
type: "validate\_days",// az alkalmazáson belül megjelenített név  
className: "blue", // szín, és egyéb vizuális tulajdonságok  
inputs: [], // bemeneti tulajdonságok. értéke lehet false is  
outputs: [], // kimeneti tulajdonságok. értéke lehet false is  
**config**: { // a node tényleges állapottere, tulajdonságai  
 self: "algo", // a szeralizálásban az objektum key-e, amiben  
 data: { // a data mezőben szereplő értékek lesznek  
 class\_name": { // egy saját adatmező kulcsa  
 type: "string" // az adatmező típusa.  
 }  
 }  
 }   
}

A configmezőben lévő értékeket egy általam készített egyszerű, dinamikus Űrlap Generáló kezeli.

**Ki- és bemenetek**

[TODO STILL]

* + 1. Űrlap Generálás

A korai web egyik mozgatórugója volt az űrlapok által nyújtott interaktivitás. Egy űrlapot HTML+JavaScript-ben sokak szerint könnyebb volt elkészíteni, mint bármilyen más akkori alternatív leíró nyelven. Ahogy nőtt a weben elvégzett feladatok komplexitása, úgy nőttek az űrlapok és a képességeik – validálás, mentés, ellenőrzés. Ez negatív folyamatokat is elindított: A *feature* *creep* mellett a specifikáció is összetett lett: A HTML specifikáción belül, csak az űrlap bemeneti mező - <input/> - leírása 30 oldal (whatwg, 2023): összehasonlításképpen, a HTML általános szintaxis leírása 10 oldal. Ez a komplexitás sok fejlesztőt eltántorított az űrlapok közvetlen kezelésétől, és a kezelést külső *library-k* ra bízza.

Mivel a node-ok belsejében dinamikusan kell űrlapot generálni, én is több ilyen library-val is kísérleteztem, de végül egy saját, korlátozott funkciókkal rendelkező űrlap generáló írása mellett döntöttem.

[TODO STILL]

* + 1. Node Connectionök definíciója

[TODO]

* 1. Adattárolás

Az alkalmazásban vizuálisan, és kód szempontból is a rajzoló funkcionalitás foglalja el a legtöbb helyet.

[TODO]

* + 1. Állapottér, Visszavonás és Újra

Egy robosztus szerkesztő felületen alapvető elvárás az *undo*, *redo* funkcionalitás. Az visszavonás és ismétlés alapvetően két fő irányzat szerint implementálható:

* Módosításkor elmentésre kerül a régi **állapottér**, vagy esetleg a *delta*, a két állapot közti különbség. Hátránya, hogy nagy állapottérnél vagy kis változtatásoknál nagy méretű objektumokat eredményez. Előnye az implementálás egyszerűsége.
* Módosításkor elmentésre kerül az **akció**. Ilyenkor a visszavonáshoz minden akciónak az inverzét kell definiálni, például node hozzáadásnak a törlés, (x,y)-al mozgatásnak (-x,-y)-al mozgatás. Előnye, hogy nem kell elmenteni az állapotokat, csak az akciókat, és a tesztelhetőség javítása. Hátránya, hogy minden lépésnek kell inverzét definiálni.

Egy gráfeszkesztőhöz mindkettő irányzat jól illik, azonban egyes funkciók implementálását megnehezítette volna[[16]](#footnote-17) az akció alapú előzmény tárolás, így végül minden módosításkor elmentem az állapotteret, és az undo, redo ezek között vált.

Az állapotteret, és az akciókat ezen az állapotteren egy központi osztállyal kezelem a *zustand* nevű library segítségével. Az alkalmazás összes komponense ebben a központi state – ben tárolja a közös információt.

**Állapottér Implementációja**

Bár az implementált megoldás nem teljesen egyezik az MVC (Model View Control) patternnal, vannak hasonlóságok: A state objektum változói és getterei a **Model** elemeivel, míg a state objektum függvényei és setterei a **Control** elemeivel megfeleltethetőek. A React libray alapvetően nem nyújt megoldást az alkalmazás DOM fájában egymástól ’távol’ álló elemeik közti kommunikációra. A felhasznált *zustand* library itt a MVC patternből az **Update** akcióért felel, ami a különböző kompensek közti kommunikációt biztosítja.

Az alkalmazás aktuális állapota egy központi state-ben való tárolása leegyszerűsíti a redo és undo funkcionalitás implementálását, hiszen az állapottér teljes cseréjével az egész alkalmazás felülete igény szerint újra renderelődika frissült információkkal. Ez a megoldás önmagában azonban nem elégséges. Elég belegondolni, hogy egy *node* mozgatása közben a node pozíciója több százszor, ezerszer változik. Minden pixelnyi elmozdulást azonban nem érdemes vagy lehetséges eltárolni. Ugyanez érvényes egy node törlésekor, amikor a hozzá tartozó line-ok is törlésre kell, hogy kerüljeket. Ezeket az akciókat csoportosítani kell, és egyszerre commitolni – azaz egy atomikus módosításnak kell venni. Az előzményt kezelő függvény ezért ki lett egészítve egyes akciók csoportosítására. A node-ok és hozzátartozó line-ok mozgatását a következő képpen oldottam meg:

Minden line, node alapból a MVC patternban lévő aktuális Modell értékét használja a pozícióhoz. Azonban, amikor egy node mozgatás alatt van, a mozgatott érték a Modellben nincs frissítve, csak a DOM-ban. Ezt egyszerű HTMLElement::setAttribute(’x’ , newPosition.x)-el (és y-ra ugyanezt megtéve), közvetlenül állítom a megszokott Control-on keresztül frissítés helyett. Amikor a mozgatás véget ért (a bal egérgomb nincs lenyomva, vagy az egér elhagyta a böngésző ablakot), akkor véglegesedik a pozíció, és a Control ténylegesen értesítve van a View által.

[knowledge about storing forms in state]

* + 1. Szerializálás és Perzisztálás

Egyes programozási nyelvek, mint a Java, kitűnő szerializációs lehetőségeket nyújtanak. A JavaScript változókat ezzel szemben három kategóriába lehet sorolni szerializációt tekintve:

* A JSON a *de-facto* megoldás, azonban itt csak alapvető típusok szerializálhatóak: number, boolean, string, stb. Újabb[[17]](#footnote-18) típusok azonban nem, sőt, a Function-t sem lehet szerializálni.
* Egyes újabb típusokat, mind például a Map, egyszerű és általános módon lehet szerializálni, mert közvetlenül lehet belőle Array-t csinálni a Array.from(myMap) függvénnyel és deszerializáció során újra Map-é konvertálni a new Map(myMapArray) függvénnyel.
* A JavaScript dinamikus típusrendszere miatt azonban nem lehetséges univerzális szerializert írni osztályokra[[18]](#footnote-19), ezért az olyan bonyolultabb adatstruktúrákra mint a class, Symbol, Proxy, osztályszinten kell szerializációt implementálni.

A beépített szerializálás hiánya egyben lehetőséget is adott számomra arra, hogy az elmentett állapottér független legyen a belső állapottértől. Így, a JSON-ba szerializáció során csak a dataflow rendszerek számára is releváns információkat tartom meg. SVG-be szerializásás során eltárolok olyan adatokat is, amik a megjelenítéshez hasznosak, mint például a node-ok pozíciója a vásznon.

**JSON Szerializáció**

JSON-ba importálás során előszőr a *node* osztályokból egy JSON lenyomat készül az általam írt Serialiser::toJSON függvény segítségével. A ki-és bemeneti élek, amik az alkalmazásban egy külön lines[] tömbben van tárolva, átkonvertálódnak a hagyományos dataflow reprezentációra.

Importálás során több lépést kell végrehajtani, példányosítani kell a *node-*okat a példányosítást a Node::fromSerialised. függvény végzi. A teljes importálást a Serialiser::fromJSON fügvény valósítja meg. Mivel a gráf szerkesztőben szabadon lehet node group-okat használni és definiálni, egyátalán nem biztos, hogy a felület aktuális állapota támogatja az elmentett node-okat, ezért itt a beolvasásnál részletes hibakezelést valósítottam meg.

**SVG Szerializáció**

Az alkalmazás talán legkiemelkedőbb képessége az SVG-be való mentés lehetősége. Az elmentett SVG egy böngészők által megnyitható file, ahol a gráf betöltődik, sőt a bemeneti mezők értéke is látszik, a körök animálva vannak. Egy hagyományos képszerkesztő nem tudja ezt az SVG file-t korrektül megjeleníteni, mert az SVG-ben XHTML névteret (namespace-t) is használok. XHTML kirenderelésére a képnézegető alkalmazások nincsenek felkészítve. Az SVG-**be** exportálás ezekből a lépésekből áll:

1. a fő SVG tartalom DOM-ból való kiemelése és szöveggé alakítása
2. HTML tartalom valid XHTML-re való konvertálása: Ez azért szűkséges, mert a HTML-el ellentétben az SVG az XML, azaz a tag-ek lezárása nem opcionális [ref].
3. Aktuális stíluslap beolvasása: Mivel a SVG támogatja a CSS-t, egyszerűen konkatenálom a rajzoló által is használt CSS-t a készülő SVG file-hoz.
4. JSON tartalom feltöltése: Az állapottér JSON szerializált változata is eltárolódik az SVG-ben.
5. File letöltésének kezdeményezése, vagy blob-ként való megjelenítése új oldalon.

A kiexportált SVG-t is később, csakúgy, mint a JSON-t meg lehet nyitni az alkalmazásban. A beolvasás során elsősorban a SVG-ben eltárolt JSON szerializációt olvassa ki a program, azonban olyan információk, mint a node*-*ok helyzete a vásznon, az SVG tartalmából vannak kiolvasva.

Az SVG-ben lévő elem pozíciója az alábbi lépések segítségével jut el az állapottérben létrehozott node-ba:

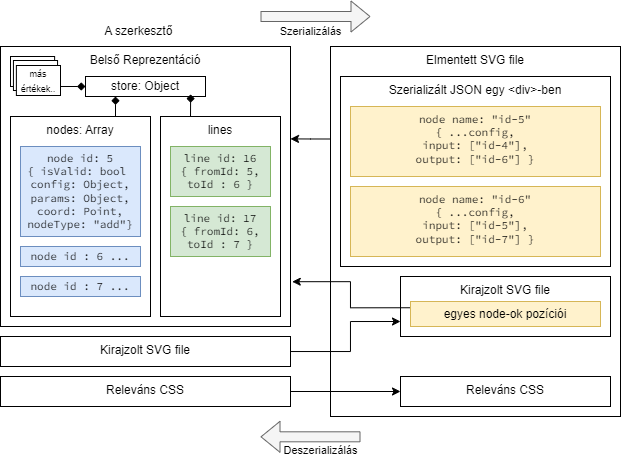
const svgHint = svgDocument.querySelector("div > svg");

…

nodeSd.ref = Node.*fromSerialised*(  
 *// a szerializálandó objektum JSON formátumban* nodeSd,  
 *// foreignObject elem az SVG-ből* svgHint?.querySelectorAll(`foreignObject[data-id]`)[processedNodeNth++]  
);

…

// ha szerializáló kap egy foreignObjectet, akkor beállítja a koordinátákat.  
if (svgFO) {  
 node.coords = Point.*fromString*(  
 svgFO.getAttribute('x'),  
 svgFO.getAttribute('y')  
 );  
}

Az alábbiakban egy összefoglaló ábrán mutatom be a konvertálás teljes menetét SVG:

* 1. A GPT-4 csatlakozás működése

A GPT-4 egy Large Language Model (LLM), ami az OpenAI cég által van fejlesztve. Erre ráépítve implementálták a ChatGPT szolgáltatást, ami egy chatbotként üzemel, a modell korábbi változatával online is ki lehet próbálni regisztráció után chat.openai.com. Ez a chat funkció elérhető API-ként is, én ezt az API-t felhasználva készítettem egy AI segítőt a gráfok készítéséhezAz alkalmazáshoz a ChatGPT támogatás teljesen opcionális; a funkció nélkül is az összes része használható az alkalmazásnak.

A GPT-4[[19]](#footnote-20) API-hoz egy szakdolgozat keretében elkészített python3 program csatlakozik, ami utána egy websocket segítségével csatlakozik a felhasználói felülethez. A felhasználó által kért szöveg magában természetesen nem lenne elég ahhoz, hogy a ChatGPT válaszoljon egy helyes gráffal.

Ahhoz, hogy egy node group-hoz lehessen a AI generálót használni, kell írni egy szöveges file-t, amiben részletesen, szövegesen leírjuk az adott gráf működését a ChatGPT számára. Ez a fejlesztő *actor* felelőssége, még a buildelés vagy indítás előtti fázisban kell egy chatgpt.prompt szöveges file-t létehozni a node típusokat leíró JSON-ök mellé. Egy jó prompt írása hosszadalmas feladat, és többszöri próbálkozást igényel, és nem garantált a tökéletes kimenet minden egyes alkalommal. A szoftver mellé készítettem példa prompt-ot[[20]](#footnote-21) is, ami prezentálja egy lehetséges megközelítését a gráf elmagyarázára.

Amikor a felhasználó beküld egy kérést angolul, például *„give an example graph”* a prompt file betöltődik, és az utasítás a file-ban lévő *%QUERY%* string helyére kerül behelyettesítésre, majd a WebSocket::send függvénnyel az API felé elküldve.

**Feldolgozás és API válasz**

A teljes Python middleware kifejezetten kicsi – 100 sor – és a központi funkcionalitása a OpenAI API-jának a használata. Az érdekesség itt az, hogy az API-t *streaming­ mode-*ban használom, azaz nem egyben várom meg a teljes válaszát az AI-nak, hanem (hasonlóan a chat.openai.com-hoz) minden egyes JSON szövegrészlet megérkezésekor az aktuális szövegrészletet azonnal tovább küldöm a frontend felé. A websocket technológiát is épp emiatt használom, mert így minden szövegdarabot könnyen vissza tud a middleware küldeni a frontend felé. Ez javítja a felhasználói élményt, mert egy hosszú gráf leírása akár másfél percbe is telhet. A streaming mode használatával azonban a generálás közben a felhasználó látja, ahogy a AI éppen „írja” a gráfot, tudja ellenőrizni a helyességét, és leköti a felhasználó figyelmét.

A frontend ki próbálja rajzolni a félkész gráfot, ezt úgy teszi, hogy ahogy érkezik a félkész JSON, megpróbálja valid JSON-né alakítani[[21]](#footnote-22) és kirajzolni a felületre. Így a félkész gráf is már kirajzolódik a felületre, ha a felhasználó észreveszi, hogy az aktuális generálás hibás, azt leállíthatja, és új szöveggel próbálkozhat

**Limitációk és Hibakezelés**

Az API erősen limitált, elsősorban *proof-of-concept* célból lett elkészítve. Nem tekinthető kész terméknek, mert hiányzik belőle az autentikáció, a szigorú hibakezelés és több kliens kiszolgálásának a lehetősége.

Amennyiben olyan prompt-ot kap a GPT-4, amiből nem tud gráfot generálni, egy hibaüzenetet fogalmaz meg.

* 1. Programozási szempontok, algoritmusok

A szakdolgozat programkódjának kialakításakor elsősorban a modularitás és a flexibilitás szempontjait vettem figyelembe. Az űrlap, a node-ok, és a node group-ok is dinamikusan vannak kezelve, [..]

Az alkalmazás központi tárolóegységét (state) egyfajta könnyűsúlyú adatbázisként használom – például az adatbázis kommunikációs modellből Create, Read Update és Delete (CRUD) akciók közül 3 megtalálható a state-ben: State::addNode(Node) , ::getNodeById(id), ::removeNode().

A state-ből való lekérdezések eredményét utána lambda kifejezések (JavaScript-ben arrow functions) segítségével szűröm vagy módosítom tovább. Ez a megoldás expresszív és könnyen bővíthető. Egy jó példa ennek a hozzállásnak a szemlétetésére a node osztály egyik metódusa: a prevNodes visszaadja azokat a node-okat, amik a jelenlegi node-ba vezetnek:

get prevNodes(): NodeId[] {  
 return State.getState().lines  
 .filter(line => line.to === this.ID)  
 .filter(line => line.from !== this.ID)  
 .map(line => line.from);  
}

Sok gráfszerkesztőben ez a talán alapvető információ a node-on belül lenne eltárolva. Azzal, hogy hogy például az előző csúcsok nincsenek explicit tárolva, csak dinamikusan lekérdezve, nem kell arra külön figyelmet fordítani, hogy ha egy csúcsot eltávolít a felhasználó, akkor az összes referenciát is törölje az alkalmazás.

Egyes mezőkre azonban az újraszámolás nagyon költséges lenne. Például egy node-nak az összes ősét megkeresni, vagy az összes bekötött node connection értéket kiszámolni nemcsak időigényes, általában felesleges is. Ezekre a mezőkre a C nyelvből átemelt volatile kulcsszót használom prefixként[[22]](#footnote-23). A volatile\_ prefixű változó jelzi a programozó számára, hogy a változóban lévő értékek bármikor változhatnak, és a változó tartalma csak rövid ideig tekinthető naprakésznek, a frissítéshez egy külső függvényt kell meghívni. Ezek a változók tartalma általában a felhasználói felületen jelenik meg, nem belső állapot információkat tárolnak.

Hasonlóan a pozíciók kezelésére létrehozott Point osztály is a fenti kód példával analóg módon, expresszíven fogalmazza meg egy változó értékének megadásának a lépéseit:

const toPoint = DragHandlerInst  
 .getCursor(toEvt) //event-ből kurzorpozíció kinyerése  
 .add(CONST.box.padLeft, 0); //ad-hoc pont hozzáadása

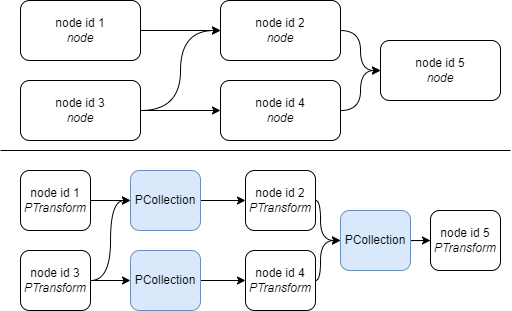
**Egy komplexebb algoritmus**

Az alapszakos szakdolgozat keretében nem kísérlek meg a gráfalgoritmusok témaköréhez egyedi és új algoritmussal hozzájárulni, a gráfelméletben előforduló algoritmikus problémák implementációja és optimalizációja jó lefedettséggel rendelkezik, lásd pl: (Mark Needham, 2019).

[todo forEachInForder]

* 1. Technológiai Kitekintés

**Különbségek egy hagyományos dataflow és a szerkesztő között**



11. ábra Felül a szerkesztőben látható gráf. Alul a JSON kimenetben generált, és a dataflow runnerek által ténylegesen használt gráf.

Az import és export közben történik egy konverzió, ami a felhasználók számára nem releváns, azonban implementációs szempontból érdekes különbség egy dataflow gráf és az alkalmazásban szerkesztett gráf között. A Google dataflowban és az Apache Beam-ben is két fő kategóriája létezik a nodeoknak: *Collection* és *Transform*. (Apache Software Foundation, 2023) A Transform végez műveleteket az adatokkal, míg a collection csak egy tárolási absztrakció. Főszabály szerint Transform az Collectionbe ír, és abból olvas; tehát két Transform közvetlenül nem tud kommunikálni, Collectionnek pedig nincsenek írási, olvasási képességei. Ez flexibilitást ad a rendszernek, hiszen az adat így tárolható másik fizikai szerveren is, illetve igény esetén több szerverre szétosztva. (Tyler Akidau, 2015) Ugyanez igaz a transform-okra, egy node-ból több példány is létrejöhet. Az elkészült felület csak transform-okat jeleníti meg, viszont az elkészült gráfban az egyes node-ok output és input mezőiben már a Collectionök generált azonosítói jelennek meg.

Ez a döntésem nem korlátozza a létrehozható gráfok számát, azonban egyes határestekben nehezítheti a gráf megrajzolását. Egy potenciális továbbfejlesztési iránya a projektnek például az, hogy a Collectionök megjelenítését a felhasználó igény szerint állíthassa.

* 1. Tesztelés

Felhasználói felületeket tesztelni hagyományosan nehéz, de legalábbis fárasztó feladat. Nem elsősorban a komplexitása miatt, hanem mert nehéz olyan teszteket definiálni, amik ténylegesen hibákat szűrnek ki.

**Tesztkörnyezet**

A tesztelés környezet a [*Jest*](https://jestjs.io/) nevű tesztkörnyezetben lett összeállítva. A Jest a [Meta OpenSource](https://opensource.fb.com/) által fejlesztett JavaScript tesztelési Framework, ami számos tesztelési paradigmákat támogat, és szorosan van integrálva az ugyancsak Meta által készített React library-vel. Az alkalmazás tesztelését három különböző módszerrel oldom meg:

**Unit Testing**

A gráf és a node-ok belső modellje kiválóan alkalmaz az *unit testing-*re. A működésük jól definiált, és könnyen lehet *parametrikus* tesztekkel edge-case-eket is lefedni. Emiatt a node, line és gráftípusok; illetve az ezeken végrehajtott alapvető (törlés, módosítás, összekötés) és komplexebb (kör ellenőrzés a gráfban, élek értékeinek végigvezetése) műveletek mind tesztelve vannak.

[TODO cplx]

**Snapshot Testing**

A *snapshot testing* egy elsősorban UI-okra alkalmazott tesztelési módszer, mely az oldal, vagy az oldalnak egyes részeiről[[23]](#footnote-24) készít egy lenyomatot. Ez a lenyomat egy HTML-szerű *.snap* file, ami úgy készül el, hogy a tesztelő software elindít egy headless[[24]](#footnote-25) Chromium böngészőt, vele elvégzi az oldal megnyitását, majd a kapott DOM struktúrát lementi. Így alkalmas a modern, JavaScript alapú Frameworkok tesztelésére is, ahol a végleges DOM struktúrát csak a weboldal betöltése után lehet megtekinteni.

Egy ilyen snapshot teszt minden minimális módosításnál el fog törni: egy új css class beírása, belső HTML szöveg átírásakor, és persze refaktoráláskor is. A célja az, hogy a fejlesztő lássa, hogy a módosítása melyik komponensekben okoz változást. Az eltört tesztnél a Jest programcsomag azonnal felajánl egy *update snapshot with new structure* opciót, így olyan változtatásoknál, ahol számít a fejlesztő arra, hogy frissíteni kell a *.snap*­-et, minimális plusz teendő van. Viszont a fejlesztő így látja, ha a módosítása váratlan helyen is okozott változást, és le tudja ellenőrizni, hogy kívánt-e az a változtatás.

Ugyanarról a komponensről lehet több snapshotot csinálni, és egy snapshot azt is érzékeli, ha egy komponens futás idejű hiba miatt nem tud *renderelődni*. Ezzel a *manuális tesztelés* egy jelentős, repetitív részét elvégzi.

Az alkalmazásban a fő SVG megjelenítési funkciók vannak snapshot tesztelve.

A snapshot testingnek azonban vannak limitációi. (Vieira, 2017):

* Hamis biztonságérzetet adhat a fejlszőknek. Minden fejlesztő álma az, hogy egy eltört tesztet kijavítson egy gombnyomással, viszont ezzel nagyon könnyű bugokat is beleégetni a *.snap* file-ba.
* Csak a DOM alakját és helyességét ellenőrzi: a design-ról, rossz kontrasztú, képernyőről kilógó, egymást eltakaró elemekről nem ad információt.

**Coverage Testing**

A lefedettség tesztelés részben megoldja sokak szemében a JavaScript és egyéb scriptnyelvek vélt hátrányát, ahol a legtöbb kódban jelentkező hiba fordítás után jelenik meg. A tesztelő software végigfuttatja a program sorait, így sok csak felhasználás közben megjelenő hiba már tesztelés fázisban megjelenik. A fenti tesztelési módszerek is hozzájárulnak a coverage testinghez, de készítettem néhány tesztet magasabb code coverage elérése érdekében. A teljes code coverage nem 100%-os, azonban a legtöbb fontos komponensnek 100%-os a lefedettsége.

**Tesztek futtatása**

npm test #tesztek futtatása  
npm test -- --coverage #coverage testing

A coverage testing eredményét a console-on túl a legenerált /coverage/Icov-report/index.html file megnyitásával interaktívan is lehet ellenőrizni.

**Tanulságok a Tesztelésből**

A gráffal kapcsolatos műveleteket az első pillanattól kezdve teszteltem, ez nagyban megkönnyítette a fejlesztést, és sok egyébként nehezen létrehozható bugot megmutatott az alkalmazásban. A line osztályban például rávilágított a tesztelés több hibára is. Eleinte lehetett vonalat nem létező node-okhoz kötni, két node között több vonalat létrehozni. Ezek nem részei az elvárt működésnek, és a tesztesetek megírása rávilágított a hiányosságra. A snapshot tesztelés segített megbizonyosodni abban, hogy amikor átalakítok egy felületrészt, nem lesznek váratlan következményei. A Unit tesztek whitebox testing, míg a snapshot tesztek blackbox testing elvei szerint készültek. A korábban nem lefedett területeit az alkalmazásnak manuális teszteléssel oldottam meg, az alábbiakban ismertetek pár ilyen tesztesetet:

**Manuális tesztek**

TODO some examples

1. Összefoglalás

TODO Summary…

* 1. Továbbfejlesztési Lehetőségek

Az elkészült szakdolgozat stabil és flexibilis, azonban sok komponensét úgy alakítottam ki, hogy könnyű legyen a továbbfejlesztés akár nekem, akár más fejlesztők számára. A node-ok már most is dinamikusan szerkeszthetőek, itt programozói tudás nem is kell a bővítéshez. Néhány *feature-*t nem implementáltam a szakdolgozatként publikált változatban, mert a fő funkcionalitástól elvették volna a figyelmet. Az alábbiakban pár felmerült továbbfejlesztési irányt és lehetőséget nézek át.

**Validálás**

A felületen jelenleg kezdetleges validálás történik, ezek elsősorban vizuálisan vannak megjelenítve. Továbbfejlesztési irányként felmerül az, hogy a hibákat egy külön nézetben lehessen áttekinteni, esetleg megoldani. Egy meglévő gráf importálása esetén, ha hiba lép fel, akkor a gráfot nem tudja a felület beölteni. Itt lehetséges egy jobb validációt írni, ami flexibilisebben kezeli a hibás vagy más verzióban készült gráfokat.

**Továbbfejlesztett űrlap**

Jelenleg 6 alap típusú bemeneti mezőt lehet az űrlapgenerálóban használni: boolean, checkbox, input, textarea, JSON és hidden. Több, új fajta bemeneti mező támogatását hozzá lehet adni, erre a programkód flexibilisen fel van készítve.

Ezen túl az alkalmazás támogatja a bemeneti mezők helyett ún. widget-ek kirajzolását. A widgeteket eredetileg az alkalmazás részének terveztem, azonban nem volt szűkséges a teljes implementációjuk, így erre nincsen a kódban példa. Egy widget egy speciális bemeneti mező, külön React komponenssel. Olyan felhasználói interakciókhoz hasznos egy widget, ahol valamilyen bonyolultabb, use-case specifikus beállítást kell csinálni. Például, egy vonal koordinátáit egy sima bemeneti mezőben megjeleníteni nem annyira felhasználóbarát: lehetne implementálni egy olyan widget-et, hogy egy gomb-ra kattintva megjelenik a kép, amin ez a vonal készült, és ott vizuálisan lehet szerkeszteni.

**Éles kapcsolat Apache Beam-el, vagy más dataflow környezettel**

A bevezetőben a 2. ábrán felvázolt lépések izoláltak: a szerkesztővel elkészített gráfot nem lehet közvetlenül beküldeni egy gráf futtató környezetbe, mint például a Google Cloud Dataflow. A *nyíl* az az ábrán egy file letöltése, majd feltöltése egy másik alkalmazásba. Ebből adódik egy természetes továbbfejlesztési lehetőség: A szakdolgozatban elkészült felület összekötése egy ténylegesen futó dataflow környezettel. A felület szintaktikai ellenőrzést végez ugyan, de a szemantikai ellenőrzést csak a tényleges dataflow környezet tudja biztosítani. Így nem csak a felhasználói kényelem nő, hanem a hibák ellenőrzése és könnyebb lesz.

***Node szerkesztő* szerkesztő**

A kezdeti tervekhez képest a jelenlegi *Edit Group* nézetben kevesebb funkció érhető el. A gráf típusok dinamikus hozzáadását nem implementáltam, hiszen ez azt akciót ritkán, és akkor is csak a fejlesztő actor végzi el, az általános felhasználóknak csak zavaró tényező lett volna. Azonban ez a megoldás jelenleg megköveteli, hogy a fejlesztő el is indítsa lokálisan a programot, és a file-okat explicit szerkessze amennyiben új node típusokat akar hozzáadni. Ezt a node típus – és node group – hozzáadást le lehet programozni továbbfejlesztésként, így egy node szerkesztő szerkesztővé válna az alkalmazás.

**Gráfszerkesztő felület**

TODO

**GPT-4 integráció továbbfejlesztés**

Az alkalmazásban a legnagyobb potenciál szerintem a GPT-4-es integrációjában található. Ez a funkció egyedülálló a gráfszerkesztő alkalmazások között jelenleg. Plusz képessége az, hogy DSL kódot generál ugyan, de nem kell DSL vagy informatikai tudás ahhoz, hogy DSL-ből generált gráfot a felhasználó szerkessze. Ezzel az alkalmazás sokkal felhasználó barátabbá válik. Ezt az integrációt számos funkcióval lehet bővíteni:

* Meglévő gráf beküldése és elemzése, szöveges leírást generálni arról, hogy mit csinál az adott gráf.
* Meglévő gráfon módosítási javaslatokat kérni, vagy továbbfejlesztést kérni.
* Előző üzenetek perzisztálása: Jelenleg minden beküldött üzenet teljesen új session-t indít el, így nehéz pontosítani egy eredeti kérdésen. A middleware továbbfejleszésével megoldható, hogy az AI kontextusként megkapja az összes előző üzenetet egy gráf összeillesztése során.
* A chatgpt.prompt file részben automatikus kigenerálása a node leírások alapján: Jelenleg a fejlesztő actornak egy hosszú leírást kell adnia az adott gráf működéséről. Ez egy hosszadalmas feladat, és nehéz jól megfogalmazni. A node leírásokból ennek a prompt-nak nagy része generálhatóvá tehető.
* Sebesség javítás DSL módosítással: Jelenleg egy nagyobb gráf kirajzolása akár két percet is igénybevehet. Ez azért van, mert a DSL az JSON-ben van megfogalmazva, ami kiváló szerializációs nyelv, azonban sok karaktert használ[[25]](#footnote-26), és a GPT-4 számára a tokenek, azaz szórészletek kigenerálása soká tart[[26]](#footnote-27).

1. Függelék
   1. Rövidítések feloldása

A szakdolgozatban sok, elsősorban webes technológiához és adatfolyam programozáshoz kapcsolódó rövidítést használok, ezeknek feloldását az alábbi táblázatban fejtem ki.

|  |  |  |
| --- | --- | --- |
| **Rövidítés, Kifejezés** | **Feloldása** | **Rövid jelentése** |
| *OOP* | Object Oriented Programming | Objektumorientált progamozást |
| *DOM* | Document Object Model | Egy API-t biztosít a JavaScript részére a HTML fa különböző elemeinek manipulására. |
| *CSS* | Cascading Style Sheets | A HTML weboldalak stílusának leírására specifikált nyelv |
| *node* | gráf csúcsa |  |
| *line* | gráf éle |  |
| *dataflow* | adatfolyam |  |
| *build, test, deploy* |  | A CI folyamatnak egyes lépései. |
| *CI* | Continous Integration | Automatizmusok egy alkalmazás buildelésére, tesztelésére és deployolására használt automatizmus |
| *JS / TS* | JavaScript, TypeScript | Progamozási nyelv |
| *SVG* | Scalable Vector Graphics | Vektografikus képformátum; elsősorban webre tervezver. |
| *XML, (X)HTML* | eXtensible Markup Languge  HyperText Markup Language | Általános leírónyelv,  XML kompatibilis HTML kód. |
| *JSON* | JavaScript Object Notation | JS objektumok szerializációra használt file típus. |
| *UI Library* | User Interface Library | Felhasználó felületek elkészítését megkönnyítő softwarekönyvtár |
| *React* |  | OpenSource UI library |
| *node, npm* | node package manager | JS csomagkezelő és környezet |
| *canvas* | vászon, webes rajzolási API |  |
| *github, VCS* | version control system | verziókezelő softwarecsomag |
| *source, target* | kifok  befok |  |
| *output, input* | kimenet  bemenet |  |
| *z-index* | A Z-Index egyes megjelenő elemeknek a sorrendiségét definiálja. Minél magasabb a Z-Index, annál ’feljebb’, közelebb van sorrendben egy elem a felhasználóhoz. Azonos Z-index esetén a DOM-ban később szereplő elem lesz teljesen látható. | |
| *stdout, stdin* | standard input & output | Ki- és bemeneti értékek a programban |
|  |  |  |
|  |  |  |

* 1. Telepítési útmutató

Az alkalmazás forráskódja az alábbi paranccsal letölthető:

git clone git@github.com:rontap/thesis.git

A software Windows, Linux és MacOS alatt is elindítható, platformfüggetlen.

**Software és Package függőségek telepítése**

Az alkalmazás helyi elindításához a *node.js* és *npm* programok szűkségesek. A támogatott node.js verzió a 16-os. Az npm az része a node.js által biztosított szoftvercsomagnak. Verzió ellenőrzése a következő paranccsal lehetséges:

node -v

Telepítés Ubuntu Linux 18.04 LTS és későbbi verziókban: <https://github.com/nvm-sh/nvm> telepítése, majd a terminál újraindítása, majd az alábbi parancsok kiadása szűkséges:

cd app  
nvm install 16 # node 16-os verziójának telepítése  
nvm use 16 # aktiválás  
npm i -g serve *#opcionális*

Telepítés Windows 10+ és MacOS alatt: <https://nodejs.org/en/blog/release/v16.16.0>

**Elindítás fejlesztői módban**

cd app  
npm install  
npm start

A függőségek npm install paranccsal telepíthetőek, ehhez a lépéshez internetkapcsolat szűkséges. Ez a telepítési fázis első alkalommal tovább is eltarthat, későbbiekben általában nincs rá szűkség.

Az npm start paranccsal a program lefordul és elindít egy webszervert a *localhost:3000*- es webcímen. A fejlesztés közben használható az ún. HMR[[27]](#footnote-28), kényelmesebbé téve az fejlesztést.

**Fordítás**

A projekt végfelhasználók által való kényelmes használatához a projektet le kell buildeleni. A buildelés során a forrásfile-ok jobban lesznek optimalizálva, a projekt mérete – *bundle size* – is jelentősen kisebb.

npm install  
npm build #buildelés  
mv build thesis # vagy *move build thesis* windowsonserve thesis #weboldal helyi webszerverrel való hosztolása

Ezek után a lefordult weboldal megnyitható a *localhost:3000/thesis[[28]](#footnote-29)* URL-en.

**Folyamatos Integráció (CI)**

Az alkalmazáshoz egy *github actions* CI is készült. Ez a folyamatos integráció a következő lépéseket hajtja végre a github repository-ban.

* Ha a git repository *main* branchére történt push-olás, akkor
* A CI a felhőben telepíti a függőségeket és lebuildeli az alkalmazást, majd
* Lefuttatja a teszteket, és amennyiben minden teszt sikeres volt, akkor
* Deployolja a *github pages* szolgáltatásra, ahol publikusan elérhető lesz az új változat.

Összefoglalva tehát, egy működő kód módosítás eredménye folyamatos integráció segítségével pár percen belül látható lesz – igény szerint - publikusan is. A jelenleg publikált URL a következő: <https://rontap.github.io/thesis/>

# Irodalomjegyzék

Az irodalomjegyzékben felsorolt, hálózaton elérhető források esetén az utolsó elérési  
dátum egységesen 2023. 05. 13.

Apache Software Foundation. (2023. 05 13). *Basics of the Beam model*. Forrás: https://beam.apache.org/documentation/basics/

Justus Bogner, M. M. (2022). To type or not to type?: a systematic comparison of the software quality of JavaScript and typescript applications on GitHub. *The 2022 Mining Software Repositories Conference*, (old.: 667 (12)). Pittsburgh, PA, USA. Forrás: https://dl.acm.org/doi/pdf/10.1145/3524842.3528454

Mark Needham, A. E. (2019). *Graph Algorithms: Practical Examples in Apache Spark and Neo4j.* O'Reilly. Forrás: https://books.google.hu/books?id=z4WZDwAAQBAJ

OpenAI. (2023. 03 14). *GPT-4 Technical Report*. Forrás: https://arxiv.org/pdf/2303.08774.pdf

OpenAI. (2023). *OpenAI Research*. Forrás: https://openai.com/research/gpt-4

Tyler Akidau, R. B. (2015). The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment, 8*(12). Forrás: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43864.pdf

Vieira, L. (2017). *Snapshot testing React components with Jest*. Forrás: https://medium.com/@luisvieira\_gmr/snapshot-testing-react-components-with-jest-3455d73932a4

W3C . (2018. 10 04). Forrás: Scalable Vector Graphics (SVG) 2: https://www.w3.org/TR/2018/CR-SVG2-20181004/

W3C WG: Erik Dahlström, Anthony Grasso, Doug Schepers. (2008. 07 15). *W3C SVG WG*. Forrás: https://dev.w3.org/SVG/proposals/svg-html/svg-html-proposal.html

whatwg. (2023. May 2). *HTML Standard, Canvas Element*. Forrás: https://html.spec.whatwg.org/multipage/canvas.html#the-canvas-element

whatwg. (2023). *HTML Standard, input element*. Forrás: https://html.spec.whatwg.org/multipage/input.html

1. Szakirodalomban Nassi–Shneiderman diagram. [↑](#footnote-ref-2)
2. Task Parralell Library (TPL) weboldala: <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/dataflow-task-parallel-library> [↑](#footnote-ref-3)
3. Azonban a szakdolgozat keretein kívül *Proof of concept* szinten sikeresen használatban van egy Apache Beam-ben készített Dataflow szerkesztésére. [↑](#footnote-ref-4)
4. A felület nem zárja ki, hogy fejlesztők definiáljanak ilyen node-okat. Egy *if* node létrehozása eléggé könnyű, a két kimenet a *then* és az *else* ágat reprezentálhatnák például. Sok tényleges haszna ennek azonban nincsen. [↑](#footnote-ref-5)
5. Ha egy node-nak van azonos nevű ki és bemeneti értéke is, akkor alapvetően a kimeneti érték érvényesül, de mivel ez dataflow implementáció függő, a szerkesztő nem korlátoz egyik irányban sem. [↑](#footnote-ref-6)
6. Hagyományos képszerkesztő alkalmazások nem tudják megnyitni, ennek okait a Rajzoló Technológia Kiválasztása fejezetben tárgyalom. [↑](#footnote-ref-7)
7. Ezt a node\_group\_used mezőben tárolja el a JSON és az SVG-be mentés esetén is. Abban az esetben, ha a node group neve időközben megváltozik, ezt a mezőt átnevezve helyre lehet állítani a mentést. [↑](#footnote-ref-8)
8. A *transpiler* egyfajta preprocesszor, ami a JavaScript esetében általában a modern nyelvi elemeket, mint pl az osztály, átfordítja olyan JavaScript kódra, amit régebbi böngészők is támogatnak. A 2010-es években, amíg az IE böngésző támogatva volt, elengedhetetlen volt a transpilerek használata. Mivel már az IE nincs támogatva, és az összes népszerű böngésző elég jól támogatja az új nyelvi funkciókat, kevesebb szerepe van a transpilereknek. [↑](#footnote-ref-9)
9. Hivatalos közlemény: <https://techcommunity.microsoft.com/t5/windows-it-pro-blog/internet-explorer-11-desktop-app-retirement-faq/ba-p/2366549> [↑](#footnote-ref-10)
10. A Chromium 101-104-es verziója is használható, amennyiben a felhasználó a *experimental-web-platform-features* flag-et bekapcsolja. [↑](#footnote-ref-11)
11. A Firefox a szakdolgozat írásakor még nem támogatta teljesen a :has() CSS pszeudoszelektort. Ez kisebb vizuális tökéletlenségekhez vezet a felület használatakor, azonban a főbb funkcionalitásokat nem érinti. A :has szelektor támogatása tervbe van véve a Mozzilla, a Firefox fejlesztői által, tehát késöbbiekben hibátlanul fog a felület megjelenni. [↑](#footnote-ref-12)
12. TS kódból egy fordító, általában a [type script compiler](https://www.typescriptlang.org/docs/handbook/compiler-options.html) (tsc) segítségével készül JS kód, a típusannotációk a végfelhasználó számára már nem láthatóak. A TS egy superset-e a JS-nek. [↑](#footnote-ref-13)
13. A Synchronized Multimedia Integration Language ([SMIL](https://developer.mozilla.org/en-US/docs/Web/SVG/SVG_animation_with_SMIL)) az SVG egyes elemeinek egyszerű animálását lehetővé tévő nyelv. Funkciói nagyrészt nem relevánsak, mert az SVG CSS animációk jobb megoldást nyújtanak, azonban komplex mozgásokat könnyen lehet definiálni benne. A szakdolgozatomban a körök bezier görbén való mozgását SMIL segítségével valósítom meg. [↑](#footnote-ref-14)
14. Valójában csak XHTML lehetséges, mert a HTML 5 az nem feltétlen *valid* XML, és az SVG-ben csak XML-t tartalmazó névteret lehet használni. Az XHTML-ben ezzel szemben követelmény, hogy egyszerre valid XML és HTML is legyen, így számomra a megfelelő választás. [↑](#footnote-ref-15)
15. A felület ezen túl a input és outputok típusát is dinamikusan kezeli, ezek a dinamikus elemek azonban egy külön modult képeznek. [↑](#footnote-ref-16)
16. Például, ahhoz hogy a node hozzáadás tökéletes inverze legyen a törlés, a törlésnél nem csak a node azonosítóját, hanem a teljes node objektumot meg kellene adni, akkor is, ha igzából más implementációs indok nem lenne rá. [↑](#footnote-ref-17)
17. Újabb alatt itt a Javascript ES6-ös verziójában és utána megjelent funkciókat értem. Az ES6 2015-ben jelent meg, olyan új struktúrákkal mint a class, map, set, lambda kifejezések. Ezeknek a konvertálását a JSON.stringify nem támogatja. [↑](#footnote-ref-18)
18. Egyes osztályokra lehetséges implementálni szerializációt, azonban általánosat nem. Preprocesszor és az eval függvény használatával részben lehetséges, azonban az eval használata korlátozott, a preprocesszor pedig dinamikus osztályokat nem tud szerializálni. [↑](#footnote-ref-19)
19. Az egyik indok, ami miatt a GPT-3 helyett csak a GPT-4 használható, hogy a GPT-3-nak nem elég nagy a „memóriája” – azaz a prompt size - ahhoz, hogy egy bonyolult gráfdefiníciót megjegyezzen. [↑](#footnote-ref-20)
20. A példa a következő file-ban található: /app/src/dynamic/groups/example/chatgpt.prompt [↑](#footnote-ref-21)
21. Két egyszerű módszert használok: Előszőr is, levágok 0-5 karaktert a JSON végéből, majd lezárom a JSON-t ]} -ökkel. Ha épp olyan szövegrészletnél járunk, ahol az AI „befejezett” egy node-ot, akkor a valid JSON-t kirajzoltatom a felületre. [↑](#footnote-ref-22)
22. [↑](#footnote-ref-23)
23. React használata esetén ez jobban definiált, egyes az egyes függvény komponensekről készít lenyomatot. [↑](#footnote-ref-24)
24. A headless mód lehetővé teszi a Chromium böngészőt mint API-t használni: Nem jelenik meg a tényleges alkalmazás, csak a teszt környezet számára fontos elemek töltenek be a memóriába. [↑](#footnote-ref-25)
25. Ez a sok node-ot tartalmazó dataflow-oknál különösen gond: a prompt-nak van egy maximális token száma, JSON-el könnyen ki lehet ebből a limitből futni, ami azt jelenti, hogy a GPT-4 számára nem lesz a teljes struktúra ismert. [↑](#footnote-ref-26)
26. Sőt, minden tokent ugyanannyi idő alatt generál. Ez azt jelenti, hogy a JSON esetén az előre ismert és kevésbé fontos kulcsokat ugyanannyi idő kigenerálni, mint a ténylegesen hasznos értékeket. [↑](#footnote-ref-27)
27. HMR - Hot Module Reload – amikor egy-egy filet szerkeszt a fejlesztő, nem fordul újra az egész alkalmazás, csak a szerkesztett file és függőségei*.* Ezen túl, az alkalmazás állapota is megmarad, tehát az épp aktuálisan szerkesztett gráf sem veszik el. [↑](#footnote-ref-28)
28. A */thesis* URL átírható a package.json-ben található *homepage* értékének átírásával. [↑](#footnote-ref-29)