EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai Kar

Média- és Oktatásinformatika Tanszék

Interaktív szerkesztő felület adatfolyam gráfhoz

Dr. Illés Zoltán,
oktató, habilitált egyetemi docens

Tatai Áron Péter
Programtervező Informatikus BSc

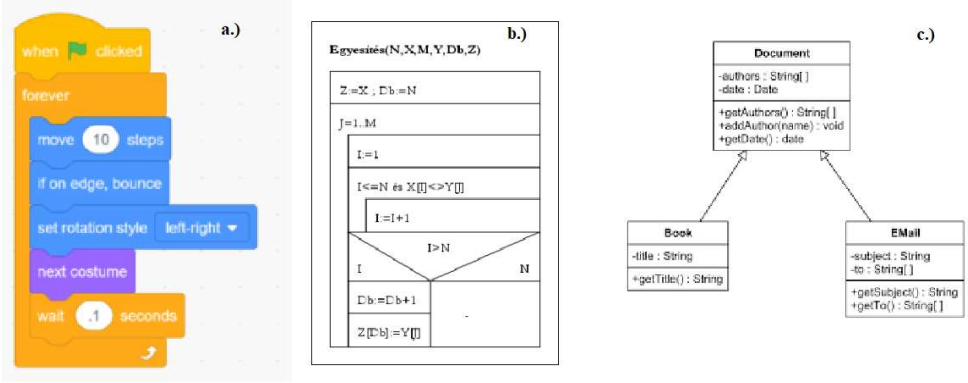
Balogh György,
CTO

Budapest, 2023

Tartalomjegyzék

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] egy pszeudokódot vizuálisan ábrázoló megoldás. Népszerű programozást tanító nyelvek, mint a *Scratch* már tényleges futtatható kódot állítanak össze, amelyben a kódblokkokat vizuálisan lehet szerkeszteni. Ilyen 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*. Ezeket a megoldásokat összefoglalva low-code development platform-nak (LCDP-nek) nevezzük. [1]



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.

Az adatfolyam (dataflow) egy programozási paradigma, ahol a szoftver egyes részeit csúcsként (node), a komponenseit összekötő elemeket pedig élként (line) reprezentálva az alkalmazás egy gráfot alkot.

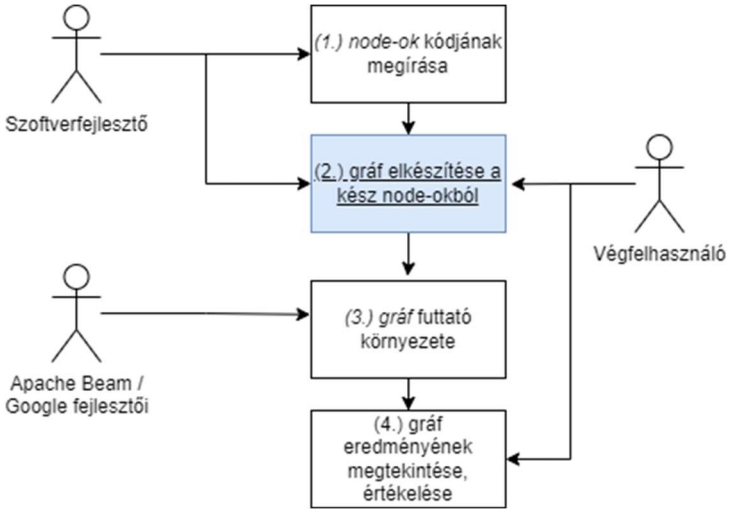
Egyes iparágakban, mint az animáció és videó az ún. node editorok alapvető eszköznek számítanak. Egyes célszoftverekben, mint például a *Blender-ben*, a 3D objektumok textúrálását egy dataflow gráf összeállításával lehet megvalósítani. Az informatikához közvetlenebbül kapcsolódó *GStreamer* videó multimédia Framework is dataflow node-

[1] Szakirodalomban Nassi–Shneiderman diagram.

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 általános környezetet, amely a dataflow modellt alkalmazza: a *Google TensorFlow*[2], az *Apache Beam*[3] és a *Microsoft .NET TPL*[4] is ezt a modellt hasznosítja. Ezek a megoldások általában nagy adatmennyiség gyors feldolgozást célozzák meg.

1.1. 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. ábra Dataflow program actor diagramja, kiemelve a szakdolgozatban megvalósított elemmel.

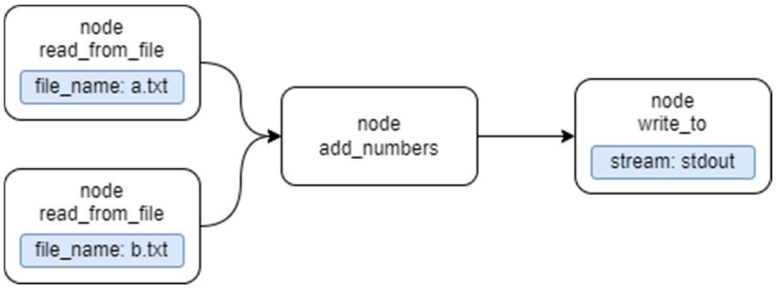
A 2. ábrán egy dataflow program a felhasználási lépéseit mutatja: A (2.) -es elem az hagyományosan egy JavaScript Object Notation (JSON) fájl szerkesztésével történik, amiben könnyű hibát véteni és programozni nem értők számára nehézkes szerkesztést eredményez. A szakdolgozatban erre a hiányzó láncszemre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

[2] Weboldal: https://www.tensorflow.org/ [Elérés: 2023. 03. 20.]
[3] Weboldal: https://beam.apache.org/ [Elérés: 2023. 03. 20.]
[4] Task Parralell Library (TPL) weboldala: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/dataflow-task-parallel-library [Elérés: 2023. 03. 20.]

többi lépésre az ipar jelenleg kielégítő megoldásokat nyújt, így a szakdolgozatban ezeket a lépéseket nem implementálom.

1.2. A dataflow és a gráfszerkesztő elemei

Egy dataflow gráf egy irányított körmentes gráf (DAG), amiben az élek (line-ok) adatot szállítanak, a csomópontok (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 3. ábra mutat.



3. ábra Példa: egy egyszerű gráf, ahol két fájl 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 programot lehetséges használni egy általam integrált ChatGPT-vel párosítva: A felhasználó az elkészítendő gráfot szabadszavasan is megfogalmazhatja, és a GPT-4 segítségével ebből egy kész gráf rajzolódik ki. 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.

2. Felhasználói Dokumentáció

A program segítségével a felhasználó egy gráfot – pontosabban dataflow gráfot – tud létrehozni, szerkeszteni és elmenteni. Ezt a gráfot egy hagyományos node editorban megszokott funkciókkal tudja szerkeszteni: lerak a vászonra különböző node-okat, éleket hoz létre a node-ok között, a node-ok tulajdonságait szerkeszti. Az alkalmazás egy weblap, a megnyitott oldal legnagyobb részét a vászon teszi ki, ahol látható az épp szerkesztett gráf. Ez a vászon mozgatható, nagyítható, és az állapota elmenthető SVG-be.

A kész gráfot a felhasználó letöltheti, és egy dataflow futtató környezetbe bemásolhatja. Egy ilyen gráfot később vissza lehet tölteni, és további módosításokat végezni rajta. A létrehozható node-ok csoportokba vannak szedve, ezeket a csoportokat node group-nak nevezem. Ezek között a node group-ok között lehet váltani, több felhasználói esethez, vagy gráf típushoz használható egy publikált felület.

A web appnak a Graphene (grafén) nevet adtam; e mögött az az intuícióm áll, hogy az alkalmazásnak a kitűzött céljai hasonlóak a grafén kémiai tulajdonságaihoz: könnyűsúlyú, flexibilis, strapabíró és nagy ellenálló képességgel rendelkezik.

Az alkalmazásnak két fő célfelhasználói csoportja van:

- **A fejlesztő:** Maga a dataflow-ban használt node-ok fejlesztője; ő definiálja a létrehozható node típusok-okat. Rendelkezik domain tudással, de nem kell a gráfszerkesztőmnek a belső működését ismernie. Egyedi node típusok létrehozását a Dinamikus node definíciók fejezetben tárgyalom. A fejlesztő megfogalmazhat egy leírást a ChatGPT számára, aminek a segítségével a felhasználó actor később gráfok elkészítését kérhető az AI asszisztenstől.

- **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 átrónák a lehetséges lépéseire, a gráf létrehozása és szerkesztésére fókuszál.

A szakdolgozatban elkészült felület nem egy konkrét dataflow implementációt, hanem általános elveket követ, így rugalmasan személyre szabható[5].

[5] Azonban a szakdolgozat keretein kívül *Proof of concept* szinten sikeresen használatban van egy Apache Beam-ben készített Dataflow szerkesztésére.

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[6], 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 környezet szűkséges.
- A gráf futtatásának kimenetét vizualizálni.

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

A felület elérhető online vagy letölthető GitHubról. A felület forráskódját manuálisan is fordítható, ezt a Telepítési útmutató fejezetben tárgyalom. Mivel a szakdolgozat programja egy weboldal, ezért asztali gépen Chrome böngészővel azonnal megnyitható.

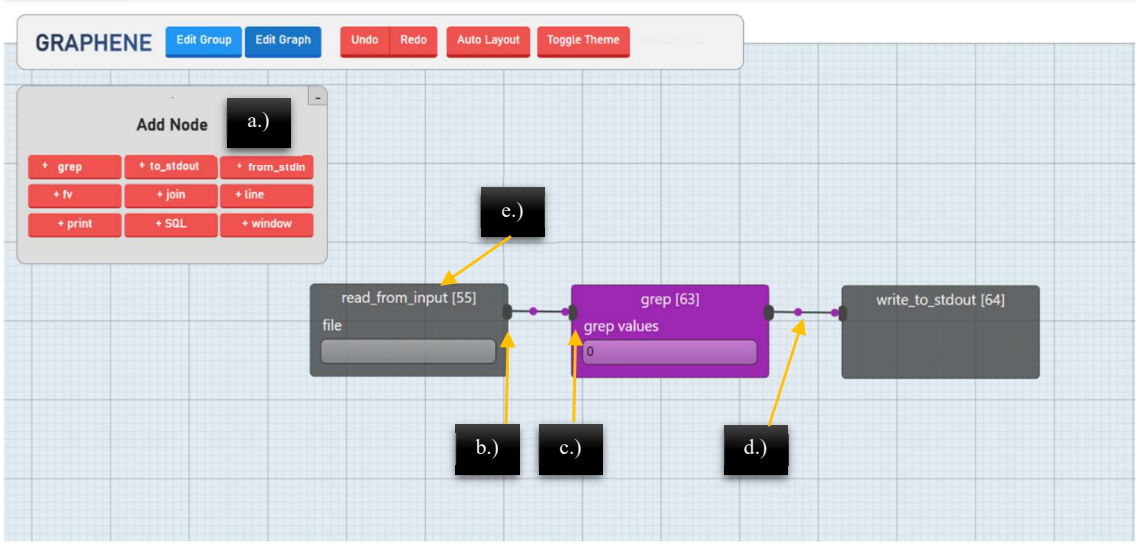
A felhasználói felület három fő elemből áll (gráfszerkesztés, node group-ok, GPT-4), ezért a felhasználói dokumentációt is ezek az elemek szerint tagoltam.

2.1. 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.

[6] 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.

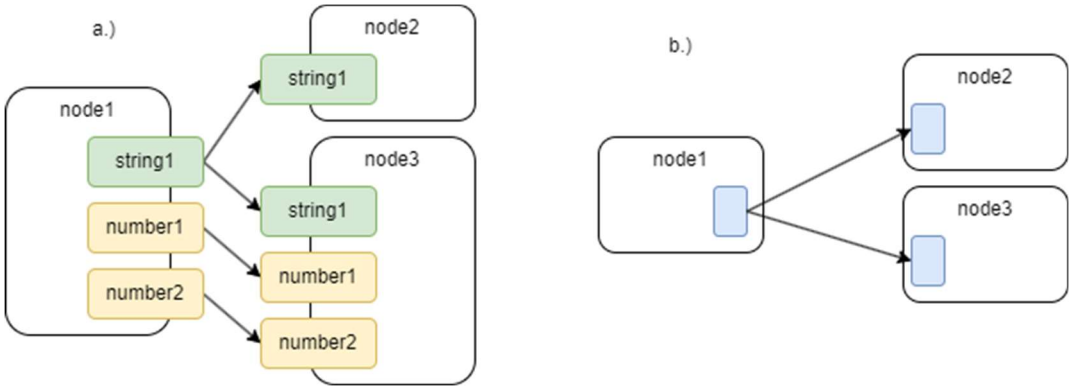


4. ábra A általam elkészített gráfszerkesztő felület, és egyes elemei. Példának egy olyan gráf van beállítva, ami fájl-ok-ból olvas, és azon egy grep-et végez el, majd

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 vagy lila 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 connection-jeik vannak, ilyenre példa a 4. ábra read_from_input és write_to_stdout node-ja.

A hagyományos node szerkesztőkkel ellentétben (például 5. ábra a.), az én szerkesztőmben (5. ábra b.) az élekben lévő adatok nem külön input és output connection-ként jelenik meg, hanem kötegelve. Azaz egy él nem egy specifikus connection-t reprezentál, hanem egy connection listát. Ennek az előnye az, hogy az olyan gráfokban, ahol a node-oknak sok connection-je van, sokkal olvashatóbbá teszi a gráf áttekintését.

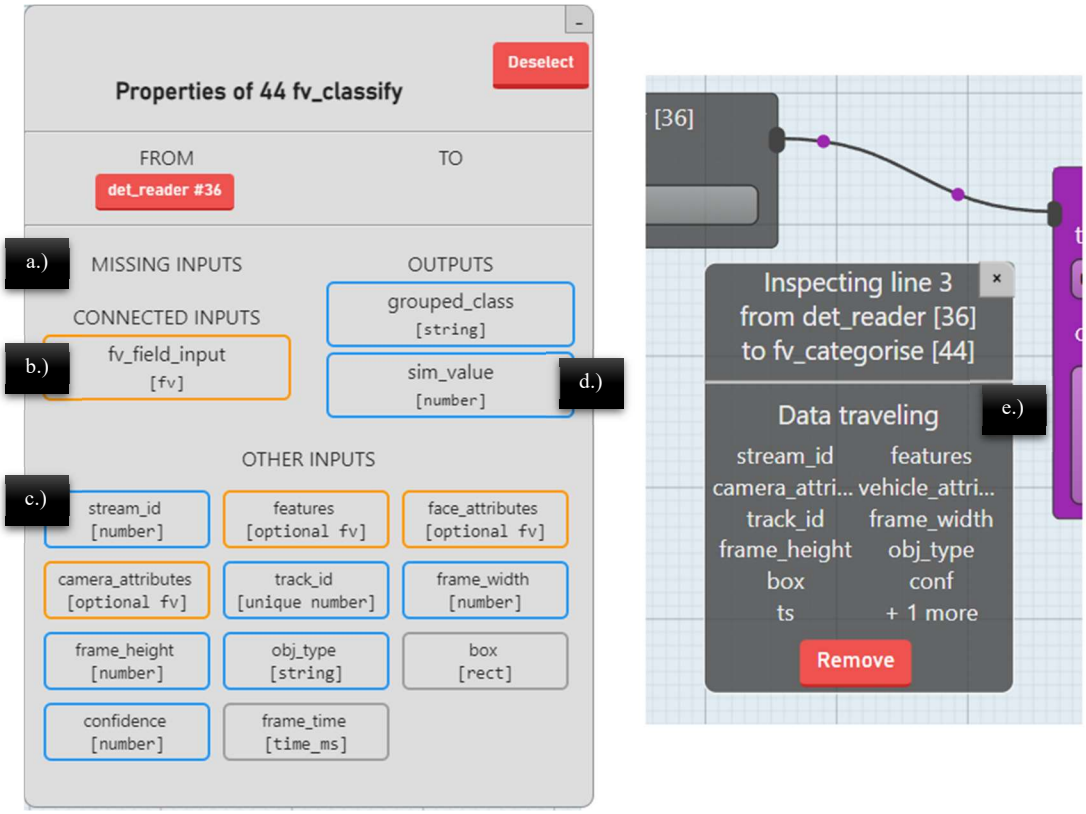


5. ábra A bal oldalon egy általános node editor felépítése, a különböző connection típusokat külön reprezentálva. Jobb oldalt az én gráfszerkesztőm, ahol alaphelyzetben ezek el vannak rejtve.

Az adott node-nak a szűkséges bemeneti paraméterek, azaz connection-ök a properties ablakban jelennek meg az adott node-ra kattintva. Itt csoportosítva vannak az alábbiak szerint az alapból rejtett csatlakozási pontok.:

- *Missing Inputs*: azon input connection-ök, amik hiányoznak, vagyis nincsenek csatlakoztatva a node-hoz. Ebben az esetben a gráf nem teljes, és nem tekinthető futtathatónak. (6. ábra a.)
- *Connected Inputs*: azon input connection-ök, amik csatlakoztatva vannak és az aktuálisan kijelölt *node* ezeket az értékeket felhasználja. (6. ábra b.)
- *Other Inputs*: azon inputok, 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. (6. ábra c.)
- *Outputs:* azon outputok, amiket a node számít ki. Ezek az értékek tovább vezethetőek ebből a node-ból[7]. Fontos részlet, hogy a következő node az összes, jelenlegi node által kapott bemenetet megkapja. (6. ábra d.)

[7] 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.



6. ábra: A properties ablak (balra), és a line inspection ablak (jobbra)

Két node-ot összekötő vonalra kattintva meg lehet tekinteni (6. ábra e.) a benne „haladó" connection-ök nevét. Természetesen az értékét nem, hiszen az alkalmazás a valós értékeket nem kezeli.

Egy-egy in out értéknek látható a típusa (azaz, hogy a dataflow-ban milyen típusú a be és kimeneti érték). Ez a típus szabadszavasan definiált, a részletes struktúrát a szerkesztő nem ismeri. Az értéket validálásra lehet felhasználni az alkalmazás továbbfejlesztése esetén.

Egy connection lehet opcionális, ebben az esetben az *optional* szöveg jelenik meg. Lehet egyedi kulcs (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.

2.1.1. 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:

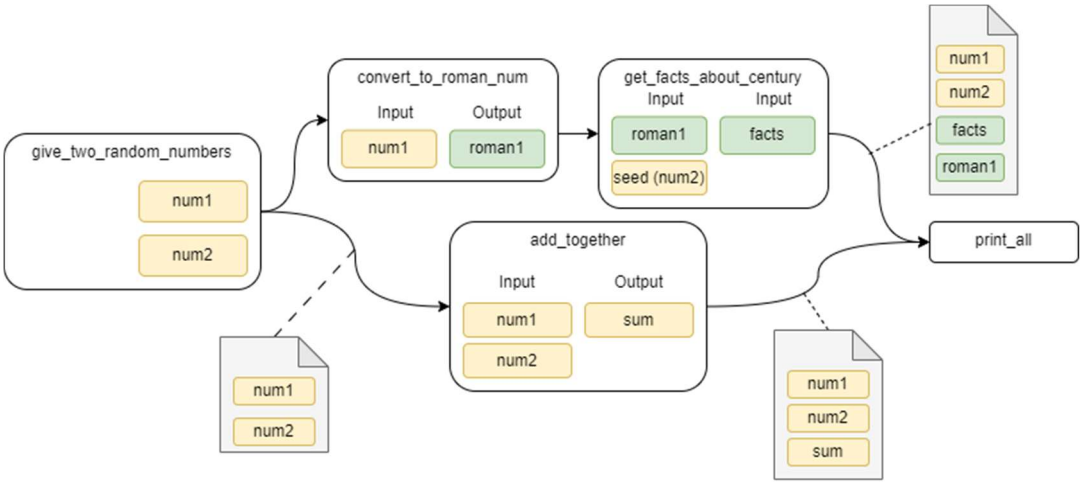
- Egyszerű szöveges beviteli mező
- Szám beviteli mező
- JSON input beviteli mező
- Textarea (többsoros) beviteli mező
- Checkbox beviteli mező
- Rejtett mező, ami a felületen nem jelenik meg, de a konfigurációban megmarad

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

2.1.2. Node-ok között folyó adat ábrázolása

A gráfban a legtöbb node-nak vannak ki- és bemeneti csatlakozásai. Ezek node-onként vannak definiálva, azonban egy-egy érték akkor is áthalad egy node-on, ha az éppen nem használja fel az értéket.

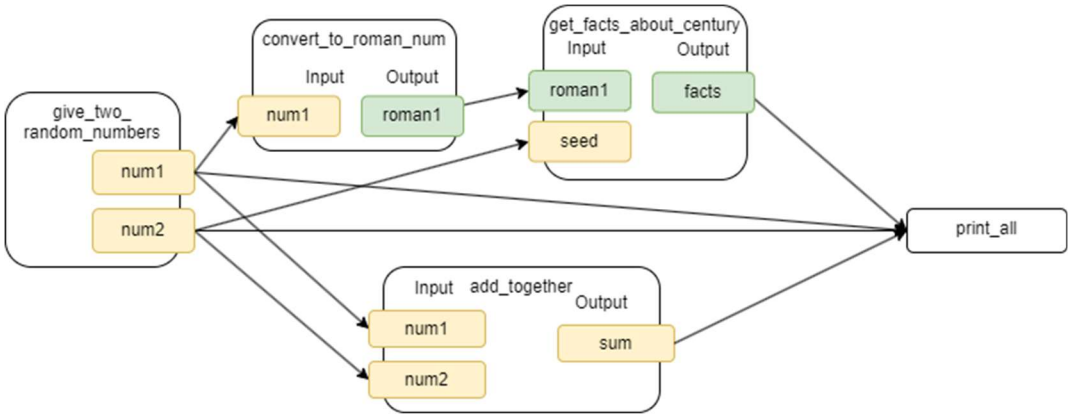
Szemléltetésként, az alábbi ábrán a node-okat fehér dobozban jelölöm, a csatalakozásokat pedig sárga vagy zöld színnel (a sárga a szám, a zöld a szöveg típusú csatlakozás). Az egyes node-okhoz eljutó értékeket ábrázolom szürke dobozban. Jól látszik, hogy a *num2*, értéket a *get_facts_about_century* node fel tudja használni, annak ellenére, hogy az előtte lévő node-nak nincs szám kimenete. Ezt azért tudja megcsinálni, mert létezik út a *give_two_number*-ből. Tehát az output-ok mindig elérhetőek maradnak a későbbi node-ok számára.



7. ábra Példa: Gráf, amiben két számot összeadok, és egy random évszázadról érdekes információkat kérünk le, majd mindezt kiírjuk. A line-okban haladó tartalmat a szürke dobozok jelzik.

Ez azért hasznos, mert későbbiekben nem kell manuálisan is összekötni a *convert_to_roman_num*-ot a *print_all*-al. Ez a plusz információ haladás nem okoz az futtató környezet számára teljesítménycsökkenést.

Összehasonlításképp, a következő vizualizálom azt, ahogy egy általános, nem az általam elkészített gráfszerkesztő alkalmazásban a fenti gráf hogyan nézne ki. Egy hagyományos gráfszerkesztőben egy él egyszerre csak egy típust szállíthat, nem típusok listáját. Azonnal látszik, hogy a felhasználó által létrehozandó line-ok száma szignifikánsan magasabb, és még csak nem is hordoz hasznos információt:



8. ábra Egy általános, nem általam írt gráfszerkesztőben így nézne ki az előző ábra.

2.1.3. 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 fájl-t böngészők meg tudják nyitni, és a gráf elmentett állapotát megjeleníteni. Tehát a megtekintéshez nem szükséges internet, vagy az alkalmazás használata[8]. A gráf újbóli szerkesztéséhez ezt az SVG-t az alkalmazásba be lehet tölteni a *Paste SVG code* gomb megnyomásával.

2.1.4. Segítő funkciók

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

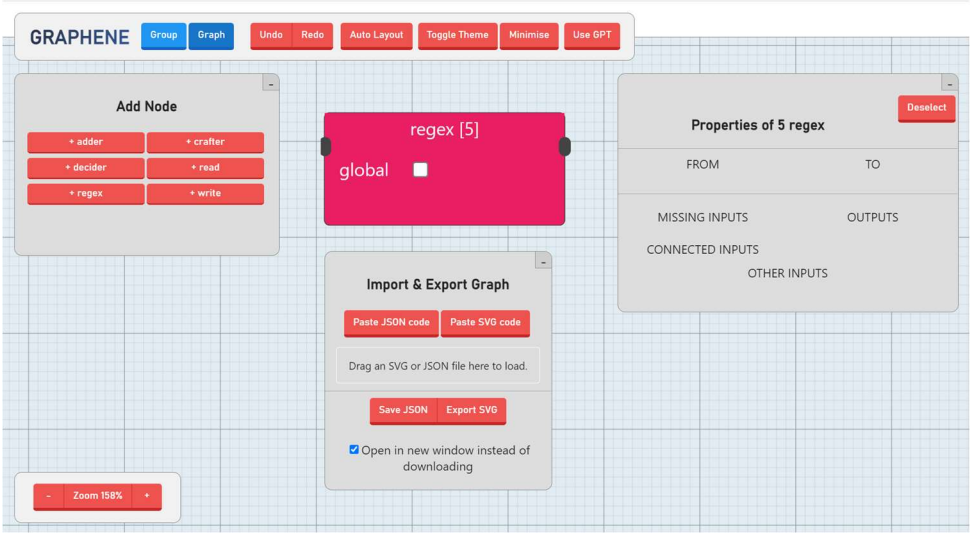
Zoom & Mozgatás: A vászon, amin a node-ok ki vannak rajzolva, mozgatható a bal egérgomb lenyomva tartásával és egér mozgatásával. Az egérgörgővel lehetséges ki- és bezoomolni a vászonra. A zoom eredeti méretét helyreállítani a bal alsó sarokban lévő aktuális zoom méretet kijelző gombbal való kattintással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

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

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.

[8] Hagyományos képszerkesztő alkalmazások nem tudják megnyitni, ennek okait a Rajzoló Technológia Kiválasztása fejezetben tárgyalom.



9. ábra Az alkalmazás egyes segítői funkciói

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.

Drag'n'Drop fájlfeltöltés: A felületre gyorsan fel lehet tölteni a korábban lementett SVG vagy JSON fájlokat úgy, hogy fájlt behúzzuk a *Drag an SVG or JSON file here to load* feliratú területre.

Ablakok elrejtése & mozgatása: Az alkalmazás különböző funkciói, mint az Importálás & Exportálás 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 „_" (minimalizálás) gomb segítségével. Minimalizálás után az az alkalmazás jobb felső sarkában megjelenik egy gomb, amivel vissza lehet állítani az ablakot. Az ablakok a felső sáv megragadásával mozgathatóak is

2.2. 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 kék *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 a *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[9], és a betöltéskor próbálja kiválasztani azt. Ha ez az információ nincs elmentve, akkor az éppen aktuálisan kijelölt node group-ba próbálja beölteni a gráfot.

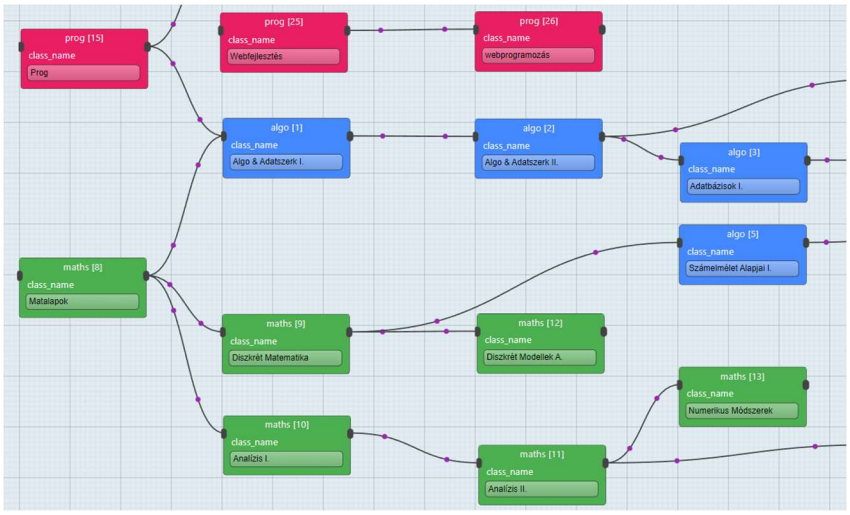
A szakdolgozatban több példa node group-ot készítettem, prezentálva pár lehetséges felhasználási területét az alkalmazásnak:

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:

[9] 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.



10. ábra Részlet az ELTE PTI C szakirányos tantervi hálóból, a különböző színek a különböző tárgycsoportokat jelöli: matematika zöld, algoritmusok kék, programozás piros.

Alkalmazás képességeit lefedő példa node group

Mivel a ELTE IK-s példa gráf az alkalmazás képességeit nem használja ki teljes mértékben, készítettem egy példa node group-ot is, ami prezentálja a weboldal összes főbb képességét.

[TODO]

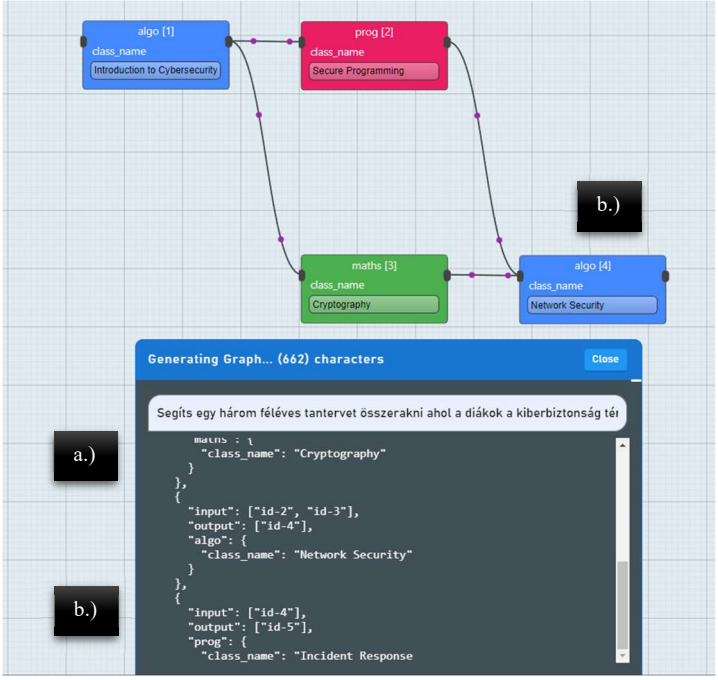
2.3. Gráfgenerálás GPT-4 segítségével

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 és egy adott séma szerinti beszélgetésre van optimalizálva [2, pp. 1-4]. 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 GPT-3.5-ös modell [3, p. 4]. Egyetemi környezetben AI-által írt szövegek kiszűrésére a professzorok is gyakran AI megoldásokat használnak, mint a zeroGPT. 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 [3, p. 91].

Gráf generálás GPT-4-gyel DSL segítségével

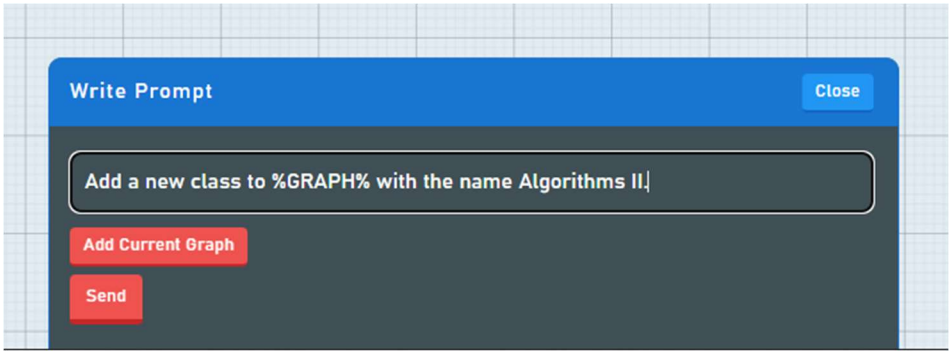
A node group-okhoz a fejlesztő actor létre tud hozni egy `chatgpt.prompt` fájlt. Ennek a fájlnak a tartalmát és szerkesztését A GPT-4 csatlakozás működése fejezetben tárgyalom. Ha létezik ilyen fájl, akkor a felhasználó a *Use GPT* gombra kattintva beírhat egy üzenetet. Ebben az üzenetben lehet egy kívánt gráfot megfogalmazni angolul, amiből a ChatGPT próbál majd egy tényleges gráfot legenerálni. Ilyen üzenetet lehet például megfogalmazni (11. ábra a.): „*Draw me an example graph*” vagy *„Draw a graph where each node type is used once”.*



11. ábra ChatGPT interakció, gráfgenerálás közben.

Természetesen a tényleges üzenet – prompt - az aktuális node group-tól függ, de amennyiben a fejlesztő helyesen írta meg a .prompt fájlt, és a middleware fut, akkor a ChatGPT elkezdi generálni a gráfot; megjelenik alul egy ablak ahol a gráf DSL-jét lehet látni (11. ábra b.). A gráfrajzoló felület pedig, ahogy készül a gráf, próbálja kirajzolni a félkész gráfot, majd amint kész, prezentálja az AI által leírt dataflow-t (11. ábra c.).

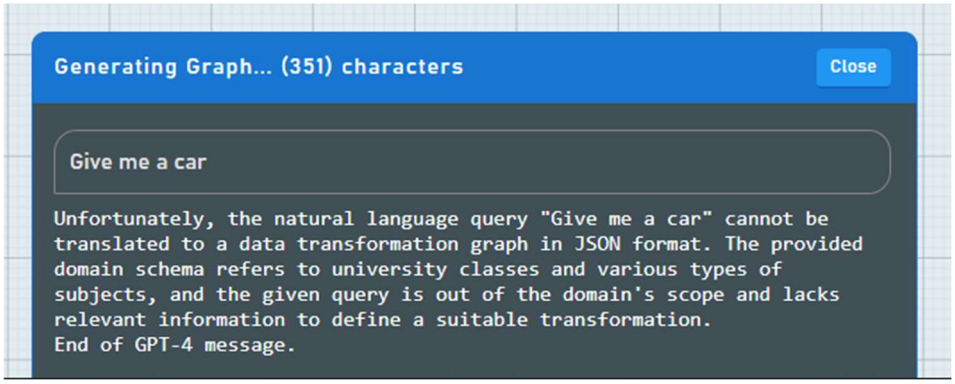
Ezen túl lehetséges az aktuális gráffal kapcsolatban megfogalmazni kérdéseket. Az aktuális gráfot a *Use GPT* ablakon belüli *Add Current Graph* gomb megnyomásával lehet hozzáadni a prompt-hoz (12. ábra). Példa általános üzenetekre: *„Add a new node to %GRAPH%”*, vagy *„There is an issue in this graph, please fix it %GRAPH%”*[10].



12. ábra Már meglévő gráfra megfogalmazott prompt.

[10] Ez a funkció a GPT-4 limitációi miatt nagyon nagy gráfokra nem működik, de 15 node-nál kevesebbekre megfelelően működik.

Egy hibás üzenet beküldése esetén a ChatGPT nem DSL-lel válaszol, hanem egy hibaüzenetet fogalmaz meg, erre példát mutat a 13. ábra. 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.



13. ábra Hibás prompt, vagy értelmetlen prompt esetén a ChatGPT szövegesen válaszol, és leírja a hiba okát.

Összefoglalva tehát, a szakdolgozatban elkészült gráfszerkesztővel nem csak a dataflow JSON fájl 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.

2.4. Hibakezelés

Egy egyszerű JSON fájl 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 pár 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.

Hibás node definíció: Ha a fejlesztő actor hibát vét, és egy node-nak rosszul adja meg a definícióját, akkor az a node nem lesz példányosítható. Ekkor a node lehelyezésekor vagy betöltésekor egy hibaüzenet figyelmeztet erre. Azaz egy hibás node definíció miatt nem lesz az alkalmazás használhatatlan.

2.5. Különbségek más gráfszerkesztőkhöz képest

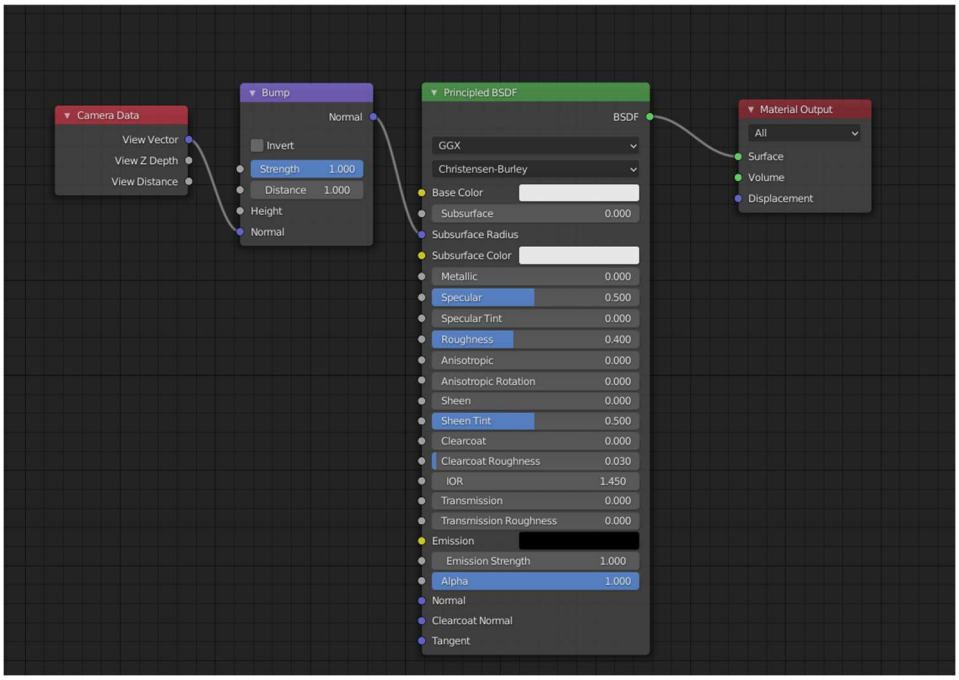
A piacon sok általános vizuális szerkesztő felület érhető el: ingyenesek, mint a *Diagrams.net* és *Lucidchart*, és fizetős zárt forráskódúak mint a *Microsoft Visio*. Ezekből az alkalmazásokból merítettem valamennyi inspirációt, azonban fontos említeni, hogy ezek elsősorban tervezésre és vizualizációkra lettek kifejlesztve: Egy Visio-ban készített osztálydiagramból nem lehet például Java kódot generálni. A Graphene szerkesztő felületemmel az explicit cél, hogy az elkészített gráfból egy kódnak a konfigurációját lehessen exportálni.

Továbbfejlesztés szempontból további előny, hogy a program nyílt forráskódú és jól dokumentált. Míg a Visio-hoz csak plugineket lehet írni, így az én alkalmazásom sokkal jobban formálható felhasználói igényekhez.

Ezért is igazából kissé közelebb van genetikailag az alkalmazás olyan felületekhez, mint a *scratch* vagy a *node red,* csak itt dataflow gráfot szerkeszt a felhasználó, ahol kevesebb egyszerű programozási elem érhető el. További különbség még az, hogy az alkalmazásomban könnyen lehet új node-okat létrehozni, ezeknek a szerkesztéséhez csak egy szövegszerkesztő kell. A legtöbb általános szerkesztő sokkal kevésbé flexibilis ebből a szempontból.

A végfelhasználó számára talán a Blender-ben lévő gráfszerkesztő lehet legközelebb a szerkeszthöz: Ott is egy dataflow gráfot rak össze a felhasználó, de a konkrét implementációja a node-oknak el van rejtve előle. A különbség a Blender-hez képest, hogy ott nem lehetséges új node típust definiálni, és az egész felület az árnyékolás és texúrálásra van optimalizálva. A Blender szerkesztőjében egyes node-oknak 10+ értéke,

és csúcsa van, a szoftvert használva döntöttem úgy, hogy a Graphene grászerkesztőmben ezt az általános problémát meg akarom oldani.



14. ábra A Blender gráfszerkesztője. Jól látszik az, hogy az egyesével feltüntetett property-k a BSDF node-on mennyire zavaróak, és jelen esetben nem is relevánsak.

2.6. Az alkalmazás elérése, kompatibilitás

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

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

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

A software Windows 8.1 fölött, Linux és MacOS X alatt is elindítható, platformfüggetlen. Az alkalmazáshoz a minimum rendszerigény ekvivalens Windows-on a minimum rendszerigénnyel, Linuxon az Ubuntu 20.04 rendszerigényével ekvivalens[11].

Az alkalmazás forráskódból való futtatása az átlag felhasználó számára nem szükséges. A telepítés részletes leírását a Telepítési útmutató fejezetben tárgyalom.

[11] Egységesen legalább 4GB RAM és 2Ghz processzor, és egy X környezet linux esetén.

Böngészőkompatibilitás

Egy weboldal fejlesztése közben nagy hangsúlyt kap a böngészőkompatibilitás témaköre. Ez az eleinte a transpilereknek[12] segítségével, majd a régi böngészőverziók deprekálásának 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[13], 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[14]-ös verzió
Mozilla Firefox	Igen, részben[15], 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 azok minimum verziói

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

[12] 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.

[13] Hivatalos közlemény: https://techcommunity.microsoft.com/t5/windows-it-pro-blog/internet-explorer-11-desktop-app-retirement-faq/ba-p/2366549 [Elérés: 2023. 05. 20.]

[14] A Chromium 101-104-es verziója is használható, amennyiben a felhasználó a *experimental-web-platform-features* flag-et bekapcsolja.

[15] 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 Mozilla, a Firefox fejlesztői által, tehát későbbiekben hibátlanul fog a felület megjelenni.

3. Fejlesztői Dokumentáció

A szakdolgozatomban egy olyan felhasználói felületet készítek, amivel egyszerűen lehet gráfokat – pontosabban dataflow gráfot - szerkeszteni. A felhasználó korábban definiált node-okat tud lerakni egy vászonra, összekötni más node-okkal, node-ok konfigurációját szerkeszteni. Ezt a gráfot utána el tudja menteni JSON-ként, és egy kompatibilis dataflow futtatóval futtathatja is. Az alkalmazás használatát egy opcionális GPT-4 asszisztens teszi egyszerűbbé.

Egy flexibilis gráfszerkesztő felület elkészítésekor fontos volt mérlegelnem azt, hogy mely elemei lesznek az alkalmazásnak modulárisak, dinamikusak, és melyek fixek. Ha túl sok dinamikus elem van, akkor csökken az ergonómia, és az alkalmazás nem lesz versenyképes egy domain specifikusabb megoldással. Ha túl sok fix elem van a programban, akkor viszont limitálva lesz az alkalmazás a későbbiekben, és fejlesztők által is nehezebben bővíthető az alkalmazás. Ez alapján az alkalmazás funkcionalitásának módosítását az alábbi kategóriákba csoportosítottam.

1. Felhasználó actor által dinamikusan szerkesztett tartalom.
2. Fejlesztő actor által szerkesztett tartalom.
3. Egyszerű szoftvermódosítást igénylő változtatás.
4. Fix elem a szoftverben, a módosítás architektúrális újragondolást igényelne.

Módosítás / **Nehézsége**	1.	2.	3.	4.
gráf, node-ok tartalma, vizuális elrendezés, GPT üzenet küldése.	×			
node group-ok, node-ok típusai, GPT prompt megírása, node connection-ök.	részben[16]	×		
új űrlapelem, widgetek, új ellenőrzés, importálás külső forrásból, konvertálás, általános továbbfejlesztés.			×	
SVG technológia, központi state, adattárolás formája, node connection-ök kezelése.				×

2. táblázat: Az alkalmazás különböző elemeinek flexibilitása.

[16] A szakdolgozat publikált változatban ezeket nem tudja a felhasználó szerkeszteni, azonban ennek a lehetősége adva van az alkalmazásban. Tehát a felsorolt elemek közül igény szerint bármelyik a felhasználó actor által is közvetlenül szerkeszthetővé tehető kevés programozói munkával.

Programozási Technológia kiválasztása

A szakkifejezések jelentését a függelék Rövidítések feloldása fejezetében fejtem ki.

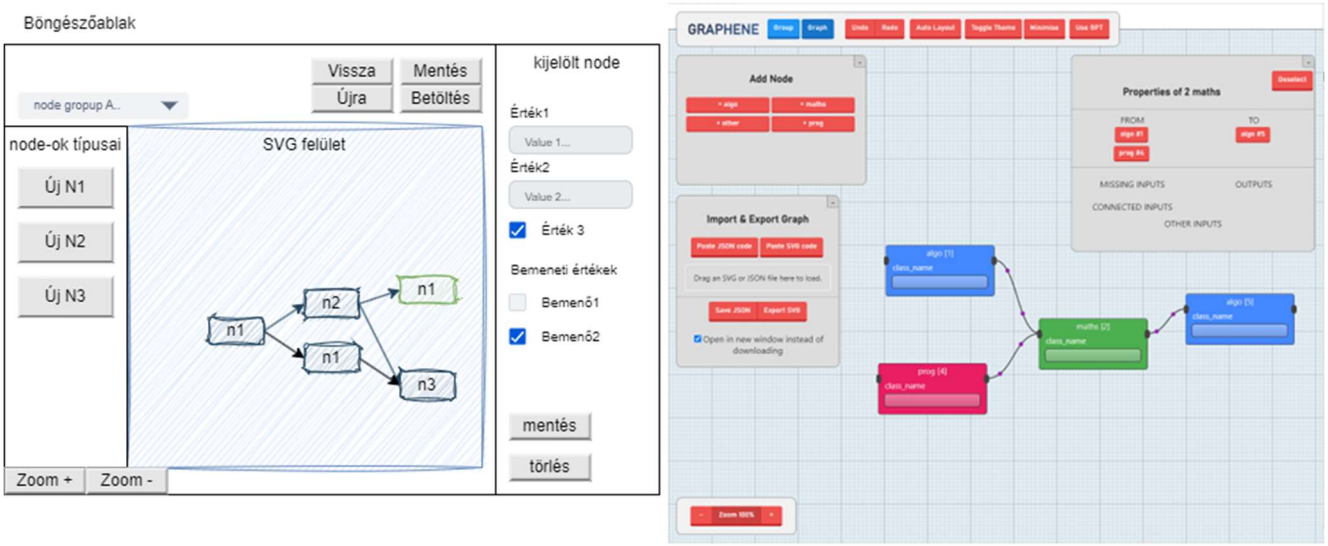
Egy hordozható és technológiailag könnyen integrálható alkalmazás fontos szempont volt a használt technológia kiválasztásánál. Szempont volt számomra az felhasználói ergonómia és a gyors prototipizálás lehetősége is. Ezért egyértelmű volt számomra az, hogy az alkalmazást valamilyen webes technológiával valósítom meg. Ezek a szempontok alapján JavaScript-ben (JS), pontosabban TypeScript-ben (TS) készül el a szakdolgozat. A TypeScript leíró ereje az tartalmazó halmaza a JavaScriptnek, azaz minden JS kód valid TS kód is, de nem minden TS kód valid JS[17]. A TypeScript explicit típusannotációkat vezet be, amik fordítási időben vannak ellenőrizve. A TS bevezetésével olvashatóbbá, megérthetővé és bővíthetőbbé válik a programkód, különösen library-k esetén, mint ez az alkalmazás. [4]

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.

Felület design terve és evolúciója

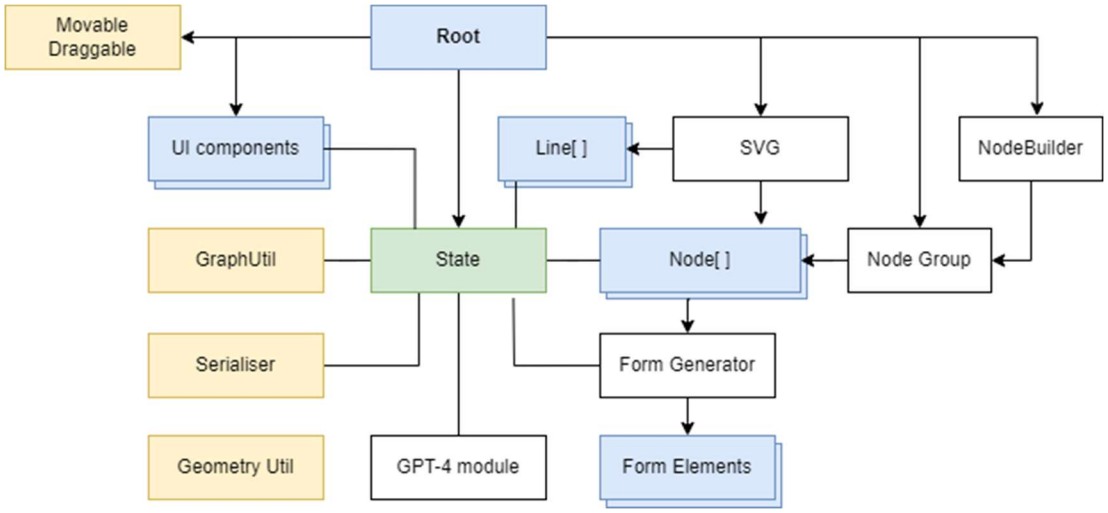
Az alkalmazást a felhasználók számára ergonomikusnak és intuitívnak szánom, ezért sok más hasonló gráfszerkesztőhöz és rajzolóhoz hasonló irányítással láttam el. Az alábbi képen látszik az alkalmazásnak a kezdeti *wireframe* terve, még azelőtt, hogy ezt a témát szakdolgozatommá választottam volna. Az alábbi ábra jobb oldalán meg egy példa képernyőkép van az elkészült alkalmazásról. Pár elemet kiemelnék, ami a megoldandó probléma jobb feltárása miatt módosításra került az eredeti tervhez képest:

[17] TS kódból egy fordító, általában a type script compiler [Elérés: 2023. 05. 20.] (`tsc`) segítségével készül JS kód, a típusannotációk a végfelhasználó számára már nem láthatóak. A TS egy superset-e a JS-nek.



15. ábra. A projekt kezdeti wireframe-je a főbb funkciókkal balra, jobbra a megvalósított szoftver a szakdolgozat keretében.

Az alkalmazásban nincsen sok külön nézet, ezek a nézetek mozgatható és lekicsinyíthető ablakokba vannak becsomagolva. Ez lehetővé teszi a felhasználónak azt, hogy igény szerint rendezze át az elemeket. A node-ok config-ja a node-on belül van vizuálisan. Így első ránézésre is lehet látni, hogy milyen beállításai vannak a node-oknak.



16. ábra

Az alkalmazás a következő főbb programozási egységei, és hozzátartozó fontosabb fájlok az alábbi táblázatban találhatóak:

Komponens	Komponens feladat	Mappa	Fontosabb fájlok

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`	`NodeGroupLoader.ts` `EdgeLoader.ts`

Segítő függvények	Vektorműveletek, alkalmazás szintű konstansok és fájl művelet segítő függvények.	`util`	`const.ts`
			`util.ts`
			`Geom.ts`

UI Űrlap Generálása	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`

3. táblázat Alkalmazás program komponensei, és releváns fájlok.

A fejlesztői dokumentáció főbb szekcióit a felhasználóihoz hasonlóan az alkalmazás fő komponensei szerint tárgyalom (gráfeszerkesztő – rajzoló technológia, node group, GPT-4 csatlakozás), általános fejezeteket szentelve az adattárolásra, a programozási szempontokra és a tesztelésre.

3.1. Rajzoló Technológia

Az szakdolgozatban elkészült alkalmazásnak fő értékajánlata az, hogy egy hagyományosan nehezen érthető dataflow szöveges fájlt egy interaktív felületen tudjon szerkeszteni a felhasználó. A legfontosabb döntés technológiailag tehát a rajzolási technológia a kiválasztása volt. Weben erre számos megoldás létezik.

3.1.1. 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 [5]. 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[18]. 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

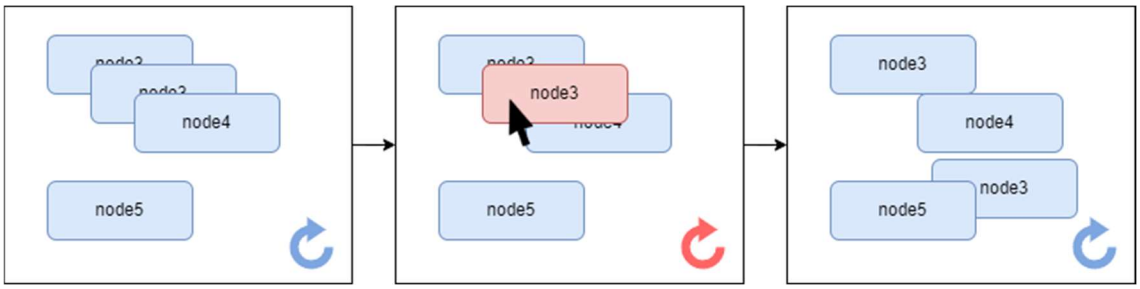
A web API-ban a canvasra többféle módon lehet rajzolni, ebből 2 fő kategóriájú megoldás létezik [6]. Az egyik értéke a fent leírt WebGL megoldás. A másik értéke, a hagyományos

[18] Weboldalak: https://threejs.org/ , https://github.com/supereggbert/GLGE és https://p5js.org/ [Elérés: 2023. 05. 20.]

canvas interfész az ún. „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 változásra az 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 újrarajzolom őket.



17. 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.

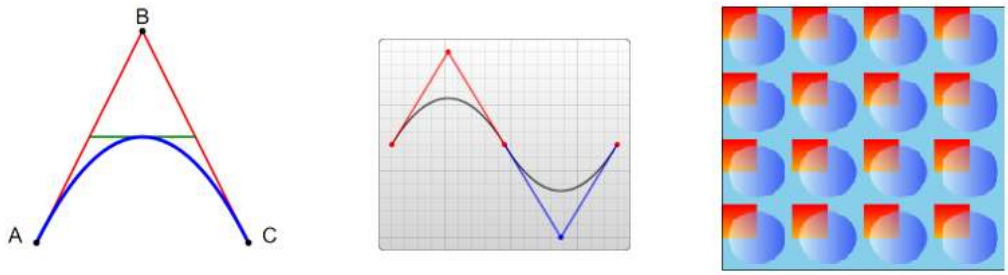
HTML DOM

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

Az SVG (Scalable Vector Graphics) egy, a World Wide Web Consortium (W3C) által specifikált vektorgrafikus képformátum. Szintaxisa XML alapú, egy element általában egy objektumot jelöl. [7] 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[19] segítségével), kiterjesztett CSS kompatibilitás és egy harmadik, ami miatt a szakdolgozatban ezt a megoldást választottam: a külső névterek használata.



18. ábra Példák SVG képességeire, balról jobbra: Bézier Görbe SVG-ben 1 kontrol ponttal, 2 kontrol ponttal és egy ismétlődő mintázat körrel és négyszöggel.

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

Az SVG 1.2 óta [8], 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[20]. Tehát egy SVG-ben el lehet helyezni HTML-t és fordítva. Így, a jóízlés határáig lehetséges egy rekurzív láncot készíteni:

[19] A Synchronized Multimedia Integration Language (SMIL) [Elérés: 2023. 05. 20.] 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 bézier görbén való mozgását SMIL segítségével valósítom meg.

[20] 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.

```
<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 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.
- Kiváló performance tulajdonságokkal bír, és nagy gráfokat is könnyen lehet szerkeszteni vele. Továbbfejlesztés esetén több optimalizációs lehetőség lehetséges.

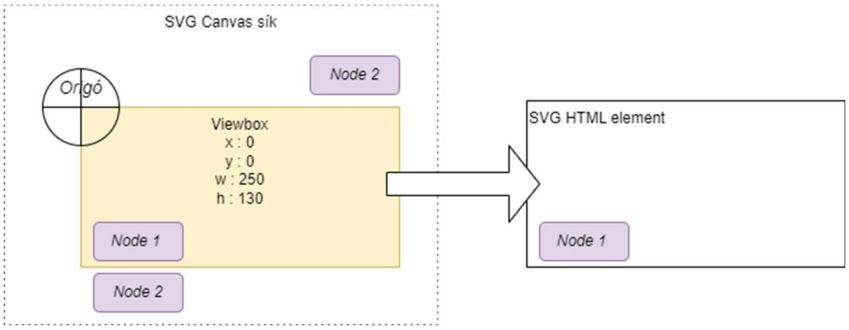
Ezek a fő indokok, amik miatt a szerkesztő felületet ezzel a megoldással valósítom meg.

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.

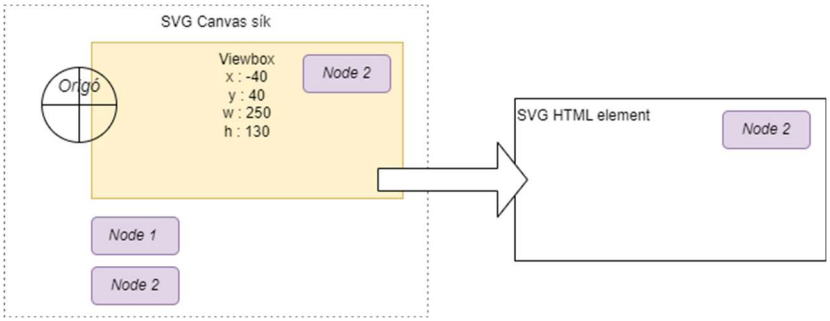
3.1.2. Mozgatás, Nagyítás és Pásztázás SVG-vel

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* tulajdonságok beállításával 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 **19**. ábrán egy alaphelyzetben lévő viewbox van: A változót 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 a képernyőn.



19. 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 20. ábrán látható.

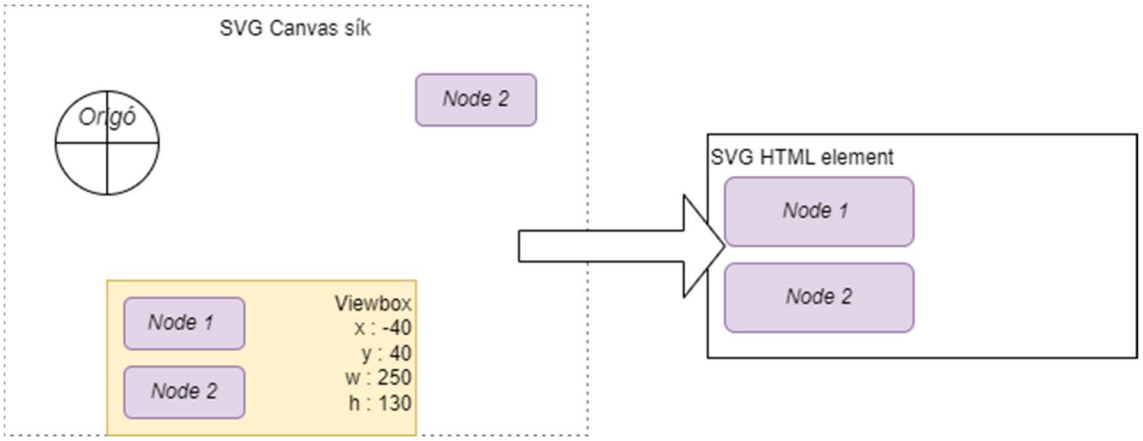


20. ábra Mozgatás után a felhasználó már a Node2-t látja.

A felhasználó az egérgörgővel, vagy a bal alsó sarokban lévő + - gombokkal tud nagyítani és kicsinyíteni is. Az alapértelmezett nagyítási méret helyreállítható a nagyítás aktuális méretére való kattintással. 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:



21. 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.

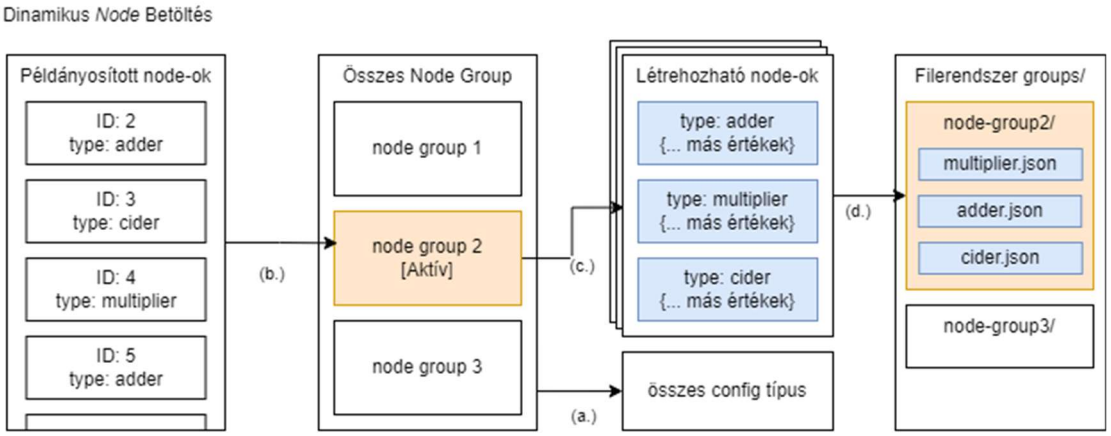
3.2. Dinamikus node definíciók

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. Ellenőrzés céljából, és a node group-ok váltására a *Groups* menüpont áll a felhasználó rendelkezésére.

A szerkesztő képességeinek prezentálásához a node-okat csoportosítom, ún. *node group*-okba szervezem[21]. 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 alábbi 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. 4. táblázatban ábrázolom a node-ok reprezentációinak különböző szintjeit.



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

A előbbi ábrán lévő rövidítések 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::Ctor`[22]

(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 definíciója látható, a 4. táblázatban lévő *más értékek* kifejtése:

[21] A felület ezen túl a input és outputok típusát is dinamikusan kezeli, ezek a dinamikus elemek azonban egy külön modult képeznek.

[22] ctor a konstruktor rövidítése.

```
{
name: "validate",      // kötelezően a JSON fájl 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 `config` mezőben lévő értékeket egy általam készített egyszerű, dinamikus Űrlap Generál kezeli. A node-nak a típusdefiníciója a `/src/app/NodeGroupLoader.tsx` fájlban található.

Ki- és bemenetek

A node ki- és bemeneteit a fenti objektum `inputs` és `outputs` mezőiben lehet definiálni. Az alábbi példán két node connection típus definíciót mutatok be:

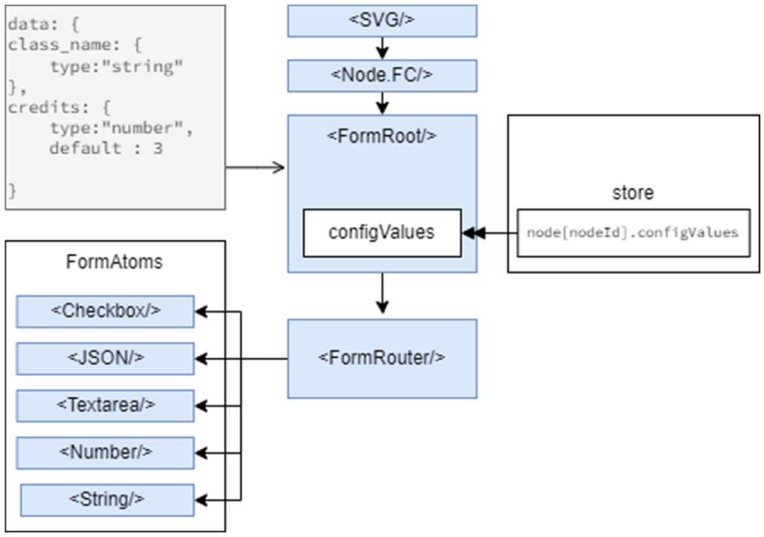
```
inputs : [{
    "type": "schedule",             // connection típusa
    "name": "class_schedule",       // connection neve
    "description": "Special Schedule ",  // opcionális leírás
    "optional": true
},
{
    "type": "number",
    "name": "class_id",
    "unique": true
}],

…
```

A type és name paraméterek kötelezőek, míg a unique, optional és description paraméterek opcionálisak. Az összes nem megadott érték alapértelmezett értéke false vagy üres string. A type paraméterek leképezése globális, a `/app/src/groups/types.json` -ben van definiálva. Itt lehet azt is megadni, hogy a vizuális reprezentáció esetén milyen színű legyen az él kerete. Az értékeknek a típusdefiníciója az `/app/src/EdgeLoader.ts` fájlban található meg.

3.2.1. 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, ez is segített a JavaScript-nek a web domináns programozási nyelvévé válnia[23]. 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 [9]: összehasonlításképpen, a HTML általános szintaxis leírása 10 oldal. Ez a komplexitás sok fejlesztőt eltántorított az űrlapok közvetlen kezelésétől, és a kezelést külső *library*-kra bízza.

Mivel az alkalmazásban 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. Az űrlapkezelő általános működéséről a Node konfigurációja fejezetben írok.



22. ábra Az az alkalmazás űrlapkezelő része. Kékkel a React komponensek vannak jelölve.

Az űrlap „vázlatát” a Dinamikus node definíciók fejezetben bemutatott `config` mező jelenti. Ezután példányosításra kerül a FormRouter által az ábrán is látható megfelelő fajtájú FormAtom: egy számnál a Number, egy boolean-nél Checkbox, és hasonlóan a

[23] A 2020-as években egyértelműnek tűnhet az, hogy a webre a JavaScript az alapértelmezett nyelv. Azonban 2010-es évekig a JavaScript nem egységes implementációja és hiányosságai miatt sok weboldal Java Appletekkel és Adobe Flash alkalmazásokkal oldott meg bizonyos multimédia interakciókat.

többi típusra. Az űrlap egyes értékeit azonban nem ezek a komponensek tárolják, hanem közvetlenül a FormRoot – ami egy-egy node-nál az űrlap gyökere. Sőt, jelenlegi implementációban ez az érték egy pointer közvetlenül a state-ben lévő értékekre. Ez azt jelenti, hogy amikor a szövegmezőt átírja a felhasználó, azonnal változik a központi tárban is az új érték. Ennek előnye az, hogy nincs duplikált adat: nincs több helyen eltárolva egy űrlap értéke. Hátránya az, hogy jelenleg a közvetlen szerkesztés miatt nem lehet visszavonni az űrlap szerkesztéseket.

További űrlap funkciók

A mezőknek lehet alapértelmezett értéket adni, ezt a `default` paraméter megadásával lehet beállítani a konfiguráció során.

Az űrlapkezelő alkalmaz hibák validálására is, ezt jelenleg a JSON komponens teszi meg: Ha hibás JSON-t írt be a felhasználó, erről egy hibaüzenet tájékoztatja.

Az elsősorban dekoratív funkciót kitöltő beállításokra az opcionális `additionalProps` objektum áll rendelkezésre. Jelenleg csak egy mezője lehetséges, a `height`, ami felülírja az alapértelmezett magasságát az aktuális űrlap elemnek. Az űrlapkezelő továbbfejlesztésekor ez a mező hasznos lehet a fejlesztő számára. Erre konkrét példákat a Továbbfejlesztési Lehetőségek fejezetben adok.

3.3. Adattárolás

Az alkalmazás központi tárolóegységét (state) egyfajta könnyűsúlyú adatbázisként használom. Az adatbázis a Create, Read Update és Delete (CRUD[24]) modell szerint módosítja a node és line objektumokat:

	Create	Read	Update	Delete
Node	`addNode(node)`	`getNodeById (ID)`		`removeNode(ID)`
Line	`addLine(line)`	`getLineBetween (ID,ID)`		`removeLine(ID)`

A state-ből való lekérdezések eredményét utána lambda kifejezések (JavaScript-ben *arrow functions*[25]) segítségével szűröm vagy módosítom tovább. Ez a megoldás

[24] CRUD részletes leírása: https://www.sumologic.com/glossary/crud/ [Elérés: 2023. 05. 01.]
[25] Lambda kifejezések JS-ben: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions [Elérés: 2023. 05. 01.]

expresszív és könnyen bővíthető. Egy jó példa ennek a hozzáállásnak a szemléltetésére 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);
}
```

3.3.1. 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[26] az akció alapú előzmény tárolás, így végül minden módosításkor elmentem az állapotteret, és az undo, redo ezek az elmentett állapotterek 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.

[26] 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 igazából más implementációs indok nem lenne rá. Így, egy új mező hozzáadásakor az egész alkalmazáson végig kell vezetni azt a mezőt, függetlenül attól, hogy használva van-e.

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 libraryt 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 komponensek 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 renderelődik a 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 keresztüli 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.

3.3.2. 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[27] 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[28], ezért az olyan bonyolultabb adatstruktúrákra mint a `class`, `Symbol`, `Proxy`, specifikus 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 először 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` függvény valósítja meg. Mivel a gráf szerkesztőben szabadon lehet node group-okat használni és definiálni, egyáltalán nem biztos, hogy a felület aktuális állapota támogatja az elmentett node-okat, ezért itt a beolvasásnál részletes hibakezelést valósítottam meg.

[27] 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.

[28] 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.

SVG Szerializáció

Az alkalmazás egyik legkiemelkedőbb képessége az SVG-be való mentés lehetősége. Az elmentett SVG egy böngészők által megnyitható fájl, ahol a gráf betöltődik, sőt a bemeneti mezők értéke is látszik, a körök animálva vannak. Egy hagyományos képszerkesztő nem tudja ezt az SVG fájlt 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-lel ellentétben az SVG az XML, azaz a tag-ek lezárása nem opcionális[29].
3. Aktuális stíluslap beolvasása: Mivel a SVG támogatja a CSS-t, egyszerűen konkatenálom a rajzoló által is használt CSS-t a készülő SVG fájlhoz.
4. JSON tartalom feltöltése: Az állapottér JSON szerializált változata is eltárolódik az SVG-ben.
5. Fájl letöltésének kezdeményezése, vagy `blob`[30]-ként való megjelenítése új oldalon.

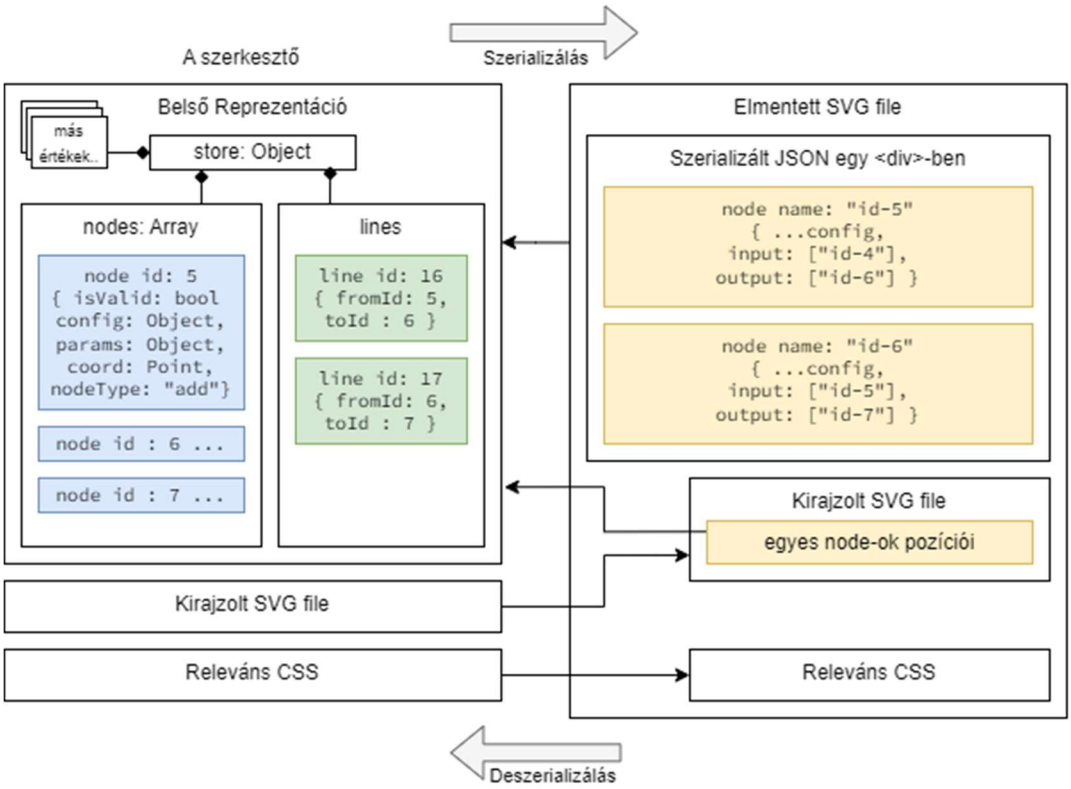
A kiexportált SVG-t is később, csakúgy, mint a JSON-t meg lehet nyitni az alkalmazásban. A beolvasás során elsősorban a SVG-ben eltárolt JSON szerializációt olvassa ki a program, azonban olyan információk, mint a node-ok helyzete a vásznon, az SVG tartalmából vannak kiolvasva.

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

[29] Forrás: https://www.w3.org/TR/xhtml11/ [Elérés: 2023. 05. 20.]
[30] binary large object (blob): bináris adatfolyam.

```
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-be és SVG-ből:



23. ábra: Az alkalmazásban használt adatszerkezetek szerializálás és deszerializálás között.

3.4. 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, ennek az intuíciójáról a Fejlesztői Dokumentáció bevezetésében írok.

Sok gráfszerkesztőben olyan, talán alapvető információ, mint a csatlakozó vonalak, vagy a következő node a node-on belül lenne eltárolva statikusan, esetleg objektum pointerek segítségével. 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 újraszámítás 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. A `volatile_` előtagú változó jelzi a programozó számára, hogy a változóban lévő értékek bármikor változhatnak, és a változó tartalma csak rövid ideig tekinthető naprakésznek, a változásról az alkalmazás többi komponense nem értesül azonnal, a frissítéshez egy külső függvényt kell meghívni. Ezek a változók tartalma általában a felhasználói felületen jelenik meg, nem belső állapot információkat tárolnak, az újraszámításukat egy state módosításnál a `handleSideEffect` függvény meghívásával lehet. Fontos, hogy a `handleSideEffect` nem csinál(hat) újabb explicit state változtatást, hiszen akkor könnyen végtelen ciklusba kerülhetne az alkalmazás[31].

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
```

[31] Ezért is `volatile_` -al ellátott változók módosításai nem state update keretében módosítják a state-et, hanem egy *escape hatch* segítségével közvetlenül a JS objektumon keresztül.

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.: [10].

A `GraphUtil::forEachInOrder` függvénye a legnagyobb komplexitással rendelkező az alkalmazásban. A függvény az összes node-on végigmegy, azonban minden node-ot annyiszor érint, ahány útvonal létezik felé. A függvény paraméternek egy *callback* függvényt kap, amit minden node-ra le fog futtatni a rekurzió. Ez a függvény az alkalmazás több helyén játszik fontos szerepet:

- Az automata elrendezésnél ennek a függvénynek a segítségével rendezem megfelelő pozícióba a node-okat.
- A kördetektálásnál is ezt használom: Ha a rekurzió egy szála olyan node-hoz érkezett, ahol már volt, akkor kört detektál, és terminál a függvény.
- Az élek tartalmának továbbadása ezzel történik.
- Ezzel lehetne ellenőrizni azt, hogy a gráf egybefüggő-e.

A függvény implementációja a kódban részletesen dokumentálva van, itt a callback függvény paramétereit fejtem ki, esetleges továbbfejlesztés esetén ugyanis nem a forEachInOrder függvényt kell átírni, hanem egy callback függvényt hozzá. Ilyenre példa a `GraphUtil::rippleNodeEdgeRefs`.

Ez a megoldás hasonló ahhoz, mind egyes JS-ben beépített függvények, mind például az `Array::Reduce`, ami egy olyan arrow function-t vár, aminek a szignatúrája a következő: `(accumulator,currentValue,?i) => newValue`

A `forEachInOrder` callback függvénynek alábbi szignatúrájúnak kell lennie:

```
(
current: Node,          // jelenleg feldolgozott node
initial: NodeId,        // gyökér csúcs[32]
visited: Line[],        // a rekurzió mely éleken haladt át
[prevNode, item]: ?[Node | null, number] // opcionális paraméter
  // prevNode -> a közvetlenül előző node (null, ha ez a gyökér)
  // item     -> a rekurzió előző lépésében ez hányadik node volt[33]
) => void
```

3.5. 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 készítéséhez. Az alkalmazáshoz a ChatGPT támogatás teljesen opcionális; a funkció nélkül is az összes része használható az alkalmazásnak.

A GPT-4[34] 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 fájlt, 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 fájlt létrehozni 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[35] is, ami prezentálja egy lehetséges megközelítését mutat a gráf elmagyarázására.

[32] Ha a gráfnak több gyökere is van, akkor hasznos ez az érték.
[33] Ez felel meg a Array::map-nél az opcionális „i” értéknek, csak itt az előző rekurzióban elfoglalt sorrendjét jelzi. Ez az érték azért szükséges, hogy a vizuális elrendezésnél, ha egy node-nak több gyereke van, ne egymásra legyenek rakva: A vertikális elkülönítést lehetővé teszi ez az érték.
[34] 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.
[35] A példa a következő fájlban található: `/app/src/dynamic/groups/example/chatgpt.prompt`

Amikor a felhasználó beküld egy kérést angolul, például *„give an example graph”* a prompt fájl betöltődik, és az utasítás a fájlban 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 az online ChatGPT-hez) 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 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[36] 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, mivel ezt érthetően fogalmazza meg, ezt megjelenítem a felhasználó számára.

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.

[36] Két egyszerű módszert használok: Először 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.

A middleware stdout-jában 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.

3.6. Technológiai Kitekintés

A szakdolgozat eredeti intuíciója az ELTE Informatika kar számára egy ún. tantervi háló készítése volt: A szakdolgozat írásakor a hallgatóknak egy táblázatban van publikálva a tanterv[37], ami egy jó megközelítés, azonban a gólyák számára nehéz egy mentális modellt készíteni arról, hogy a tárgyak hogyan épülnek egymásra, és hogy melyik félévben milyen tárgyak lesznek. Ez a szakirányválasztásnál különösen nehéz, három táblázat összevetésével kell megnézni a három szakirányon lévő különböző tárgyakat. Vannak tárgyak amik A és B, és B és C szakirányon közösek, de ezeket nehéz kivenni a táblázatból. Vannak tárgyak, amik össze vannak vonva egyes szakokon, vagy más kreditértékkel rendelkeznek.

Az eredeti tervem tehát a szakdolgozathoz egy tantervi vizualizáció írása volt, ennek el is készítettem egy vázlatát, ami az alábbi oldalon van publikálva: https://rontap.github.io/elteik-tantervihalo/. A dokumentumban táblázat szerűén vannak a tárgyak ábrázolva, és egy tárgyra való kattintással az előfeltételek megjelennek. A megjelenítő hasznos, de két problémám volt a projekttel: Először is, nehezen lehet bővíteni egy ilyen tantervi hálót más szakok irányába. Az ELTE egyes karjai között máshogy működnek az előfeltételek, esetleg szigorlatok vannak stb., és egy ilyen statikus alkalmazásban ezeket sok effort lett volna karbantartani. Másodszor, a hallgatók az ELTE-n sok külső eszközt használnak az órarendjük összeállítására, azonban ezek az ezközök közül sok, az alkalmazást készítő hallgató távozásával idővel elromlik, elavult

[37] Link: https://www.inf.elte.hu/tantervihalok [Elérés: 2023. 05. 20.]

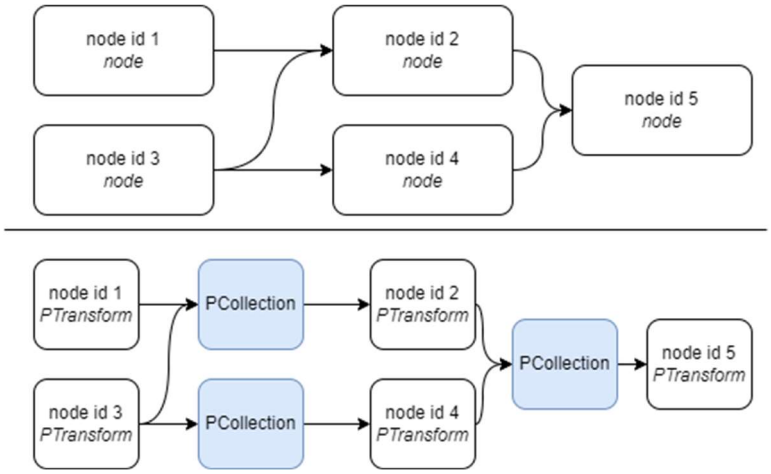
lesz. Az én tesztalkalmazásomnak is ez lett volna a sorsa, hiszen a forráskód – és a tárgyak – nem szerkeszthetőek, nem menthetőek le.

Ezért elhagytam az eredeti szakdolgozat ötletemet, és döntöttem a jóval ambiciózusabb jelenlegi szakdolgozatnak a terve mellett. A tantervi háló összerakás, mint ahogy korábbi fejezetekben bemutattam, továbbra is lehetséges. Most már a hallgató a saját tantervi hálóját személyre tudja szabni, szerkeszteni és lementeni. Csak mellé, dataflow gráfokat is tud az alkalmazás kezelni.

Különbségek egy hagyományos dataflow és a szerkesztő között

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 dataflow-ban és az Apache Beam-ben is két fő kategóriája létezik a node-oknak: *Collection* és *Transform*. [11] A Transform végez műveleteket az adatokkal, míg a Collection csak egy tárolási absztrakció. Főszabály szerint Transform az Collection-be ír, és abból olvas; tehát két Transform közvetlenül nem tud kommunikálni, Collection-nek 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. [12] Ugyanez igaz a Transformok-ra, egy node-ból több példány is létrejöhet. Az elkészült felület csak Transformok-at jeleníti meg, viszont az elkészült gráfban az egyes node-ok output és input mezőiben már a Collection-ök generált azonosítói jelennek meg, a különbséget az alábbi prezentálja.

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 Collection-ök megjelenítését a felhasználó igény szerint állíthassa.



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

3.7. 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*[38] nevű tesztkörnyezetben lett összeállítva. A Jest a Meta OpenSource[38] 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 gráf típusok; illetve az ezeken végrehajtott alapvető (törlés, módosítás, összekötés) és komplexebb (kör ellenőrzés a gráfban, élek értékeinek végig vezetése) műveletek mind tesztelve vannak.

Snapshot Testing

[38] [Elérés: 2023. 03. 10.]

A *snapshot testing* egy elsősorban UI-okra alkalmazott tesztelési módszer, mely az oldal, vagy az oldalnak egyes részeiről[39] készít egy lenyomatot. Ez a lenyomat egy HTML-szerű *.snap* fájl, ami úgy készül el, hogy a tesztelő software elindít egy headless[40] 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 [13]:

- Hamis biztonságérzetet adhat a fejlesztőknek. Minden fejlesztő álma az, hogy egy eltört tesztet kijavítson egy gombnyomással, viszont ezzel nagyon könnyű bugokat is beleégetni a *.snap* fájlba.
- Csak a DOM alakját és helyességét ellenőrzi: a designról, 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 végig futtatja a program sorait, így sok csak felhasználás közben

[39] React használata esetén ez jobban definiált, egyes az egyes függvény komponensekről készít lenyomatot.
[40] 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.

megjelenő hiba már tesztelés fázisban megjelenik. A fenti teszteléssi 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 terminál kimenetén olvashatjuk, azon túl a legenerált `/coverage/Icov-report/index.html` fájl 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.

Manuális integrációs tesztek

A korábban nem lefedett területeit az alkalmazásnak manuális teszteléssel oldottam meg, az alábbiakban ismertetek pár ilyen tesztesetet.

A GPT-4-es middleware-hez nem készültek automata tesztek. Ennek egyik oka az, hogy egy külső API-hoz csatlakozik, és a GPT-4 által generált szöveg nem determinisztikus. Ráadásul, a Python middleware egyszerű, és külön tesztelni nem lenne sok haszna.

A teljes elméleti lefedettség eléréséhez azonban emiatt pár integrációs teszteset is írtam, az alábbi táblázatban látható:

Teszteset neve	Elő-feltétel	Teszt lépés	Elvárt eredmény
Felület vizuális ellenőrzése	A	Az alkalmazás gombjainak és elemeinek vizuális áttekintése.	Nincs hibásan megjelenő elem, vizuális glitch.
Node-ok lerakása	A	Az összes node típusból legalább egyet lehelyez a tesztelő, ezek tartalmát szerkesztheti.	Az alkalmazás működik, a node-ok nem adnak hibaüzenetet.
Gráf szerkesztés	A	Node-ok lerakása után ezeket össze lehet tetszőlegesen kötni, vizuálisan áthelyezni.	Az összekötés működik, a node connectiönökben megjelenik az információ.
Gráf ki-be töltése	A, B	Gráf exportálása majd importálása, SVG-ként lementés és megtekintés.	A kiexportált gráfot később meg lehet nyitni, az SVG-ként lementett fájl azonos vizuálisan a szerkesztőben találhatóval.
Node group váltás	A, B, C	A „group” gombra kattintás után a tesztelő átvált egy másik node group-ra.	Az alkalmazás betölt, az új group-hoz tartozó node-ok jelennek meg.
Kördetektálás	A, B	A gráfban kört készít a tesztelő.	Piros vonal jelzi a kör elemeit.
Téma váltás	A	A tesztelő a témát világosra állítja	Az összes korábban olvasható felhasználói elem olvasható marad.
Undo és Redo	A, B, C	Egy már szerkesztett gráfban az undo és a redo gombok használata.	Az undo az elvártak szerint visszavon, a redo pedig a visszavont lépést újracsinálja.
GPT-4 Happy path	A, D, E	Megfelelő csatlakozás esetén a tesztelő egy példa gráfot kér a GPT-4-től.	A GPT-4 legenerál egy valid gráfot, amit a felhasználói felület megjelenít.
GPT-4 API Hiba	A, D	A GPT-4 csatlakozást hibásan állítja össze a tesztelő	A felület és a middleware figyelmeztet a hibára.

Az előfeltételek kódját pedig ebben a táblázatban fejtem ki:

Előfeltétel Kódja	Előfeltétel leírása
A	A program kezdőképernyője betöltött
B	A programban be van töltve egy gráf, vagy legalábbis node-ok.
C	A programba több node group van aktiválva
D	A Python middleware el van indítva
E	A Python middleware hozzáfér a GPT-4-hez, és sikeresen elindult.

4. Összefoglalás

Az alkalmazás a kezdeti tervekhez képest egy logikus evolúció során jutott el a szakdolgozatban publikált változatig. Az alkalmazás robosztus, gyors, a felület modern. A szerkesztés egyszerű, bár a dataflow koncepciójának ismeretének szükségessége megnehezítheti laikus felhasználóknak a szoftver használatát. Ennek, egy tantervi háló segítségével tudom prezentálni, hogy általános gráfszerkesztőként is megállja a helyét az alkalmazás.

A vászonra rajzolási technológiák közül az SVG választása kifizetődött: az SVG funkciók jól működnek, új funkciók hozzáadása egyszerű és véleményem szerint az alkalmazás egyik kiemelkedő funkciója a közvetlenül SVG file-ba való mentésnek a lehetősége. A különböző modulok, mint node-ok hozzáadása és exportálás külön ablakokba való szervezése nyitva hagyja a helyet további funkciók egyszerű hozzáadására, és a felhasználó általi személyre szabásra. Ezek a mozgatható és minimalizálható ablakok sok professzionális szoftverből ismerős módon működnek.

A szakdolgozat tervezésének kezdete után jelent meg a ChatGPT (2022 november 30.), és érdeklődési körömből adódóan azonnal elkezdtem kísérletezni vele, majd a GPT-4 white paperjét [14] olvasva eldöntöttem, hogy a szakdolgozatomba is integrálom az API segítségével. A külső konzulensem iránymutatásával megfogalmaztam egy példa prompt-ot, amit a felhasználó nem is lát, csak a GPT számára ad iránymutatást arról, hogy milyen szöveget generáljon. Az integráció nem várt stabilitással működik: a GPT-4 ritkán hibázik szemantikailag, és szintaktikai hibát szinte sosem követ el. Ez az AI integráció az alkalmazás kiemelkedő értékajánlata.

4.1. 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 gombra kattintva megjelenik a kép, amin ez a vonal készült, és ott vizuálisan lehet szerkeszteni.

Éles kapcsolat Apache Beammel, vagy más dataflow környezettel

A bevezetőben a 2. ábraán felvázolt lépések izoláltak: a szerkesztővel elkészített gráfot nem lehet közvetlenül beküldeni egy gráf futtató környezetbe, mint például a Google Cloud Dataflow. A nyíl az ábrán a következő lépéseket foglalja magában: egy fájl 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 fájlokat 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

A gráfszerkesztő modul továbbfejlesztésének a célja elsősorban a felhasználói élmény javítása lehet. Itt több funkció implementálásra kerülhet: Több node kijelölése és mozgatása (esetleg törlése), node-ok minimalizálásának lehetősége, jobb z-index szerinti rendezés, továbbfejlesztett bézier görbe rajzolás és jobbklikk menüben több opció elhelyezése. Ezek a funkciók az alkalmazás evolúcióját követve igény szerint implementálhatóak későbbiekben.

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 *user-friendly*-bbé 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` fájl 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 igénybe vehet. Ez azért van, mert a DSL az JSON-ben van megfogalmazva, ami kiváló szerializációs nyelv, azonban sok karaktert használ[41], és a GPT-4 számára a tokenek, azaz szórészletek kigenerálása soká tart[42].

[41] 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.

[42] 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.

5. Függelék

5.1. 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 programozási paradigma
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	Webre kifejlesztett programozási 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 fájl 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	kimenő él bemenő él	Egy node-ban kimenő és bemenő él
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

5.2. Telepítési útmutató

Az alkalmazás forráskódja az alábbi paranccsal letölthető:

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

Software és Package függőségek telepítése

Az alkalmazás helyi elindításához a *node.js* és *npm* programok szükségesek. 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[38] telepítése, majd a terminál újraindítása, majd az alábbi parancsok kiadása szükséges:

```
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[38]

Elindítás fejlesztői módban

```
cd app
npm install
npm start
```

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

Az npm start paranccsal a program lefordul és elindít egy webszervert a *localhost:3000*-es webcímen. A fejlesztés közben használható az ún. HMR[43], kényelmesebbé téve az fejlesztést.

[43] Hot Module Reload (HMR): Amikor egy-egy fájlt szerkeszt a fejlesztő, nem fordul újra az egész alkalmazás, csak a szerkesztett fájl és függőségei. Ezen túl, az alkalmazás állapota is megmarad, tehát az épp aktuálisan szerkesztett gráf sem veszik el.

Fordítás

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

```
npm install
npm build #buildelés
mv build thesis # vagy move build thesis windowson
serve thesis #weboldal helyi webszerverrel való hosztolása
```

Ezek után a lefordult weboldal megnyitható a *localhost:3000/thesis*[44] 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/

GPT-4 Csatlakozás

A GPT-4 csatlakozás használatát a Gráfgenerálás GPT-4 segítségével fejezetben, míg a technikai implementációs részleteket A GPT-4 csatlakozás működése 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.

[44] A */thesis* URL átírható a package.json-ben található *homepage* értékének átírásával.

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` fájlt ahova a megszerzett API kulcsokat kell beilleszteni. Erre példát a `.env_EXAMPLE` fájl nyújt. Ha a környezeti változók beállítását elmulasztjuk, erre hibaüzenet emlékeztet minket.

A venv környezet függőségeit a `pip` csomagkezelővel lehet telepíteni, a környezet bemeneti fájlja 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.

Irodalomjegyzék

Egyes források 5 évnél régebbiek, azonban ezekben az esetekben ezek szabványok jelenleg is aktív változatáról értekeznek, vagy időtálló matematikai bizonyítások.

[1] A. I. D. D. R. A. P. Apurvanand Sahay, „Supporting the understanding and comparison of low-code development platforms," in *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020.

[2] X. Z. S. S. R. G. P. K. Abhinav Rastogi, „Towards Scalable Multi-Domain Conversational Agents: The Schema-Guided," Google Research, Mountain View, California, USA, 29 January 2020. [Online]. Available: https://arxiv.org/pdf/1909.05855.pdf. [Hozzáférés dátuma: 13 05 2023].

[3] OpenAI, „GPT-4 Technical Report," 14 03 2023. [Online]. Available: https://arxiv.org/pdf/2303.08774.pdf. [Hozzáférés dátuma: 13 05 2023].

[4] M. M. Justus Bogner, „To type or not to type?: a systematic comparison of the software quality of JavaScript and typescript applications on GitHub," in *The 2022 Mining Software Repositories Conference*, Pittsburgh, PA, USA, 2022.

[5] L. Caballero, „An Introduction to WebGL — Part 1," 2011. [Online]. Available: https://dev.opera.com/articles/introduction-to-webgl-part-1/. [Hozzáférés dátuma: 13 05 2023].

[6] whatwg, „HTML Standard, Canvas Element," 2 May 2023. [Online]. Available: https://html.spec.whatwg.org/multipage/canvas.html#the-canvas-element. [Hozzáférés dátuma: 13 05 2023].

[7] W3C, 04 10 2018. [Online]. Available: https://www.w3.org/TR/2018/CR-SVG2-20181004/. [Hozzáférés dátuma: 05 01 2023].

[8] A. G. D. S. Erik Dahlström, „W3C SVG WG," W3C, 15 07 2008. [Online]. Available: https://dev.w3.org/SVG/proposals/svg-html/svg-html-proposal.html. [Hozzáférés dátuma: 13 05 2023].

[9] whatwg, „HTML Standard, input element," 2023. [Online]. Available: https://html.spec.whatwg.org/multipage/input.html. [Hozzáférés dátuma: 20 05 2023].

[10] A. E. H. Mark Needham, Graph Algorithms: Practical Examples in Apache Spark and Neo4j, O'Reilly, 2019, p. 241.

[11] Apache Software Foundation, „Basics of the Beam model,” 13 05 2023. [Online]. Available: https://beam.apache.org/documentation/basics/. [Hozzáférés dátuma: 13 05 2023].

[12] R. B. E. S. e. a. Tyler Akidau, „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,* %1. kötet8, %1. szám12, 2015.

[13] L. Vieira, „Snapshot testing React components with Jest,” 2017. [Online]. Available: https://medium.com/@luisvieira_gmr/snapshot-testing-react-components-with-jest-3455d73932a4. [Hozzáférés dátuma: 13 05 2023].

[14] OpenAI, „OpenAI Research,” 2023. [Online]. Available: https://openai.com/research/gpt-4. [Hozzáférés dátuma: 20 05 2023].