FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Generátor neorientovaných grafů
Student:	Miroslav Jaroš
Vedoucí:	Ing. Michal Šoch, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Seznamte se s termínem neorientovaný graf a s postupy generování neorientovaných grafů dle zadaných parametrů. Dále se seznamte s aktuálními trendy a technologiemi pro tvorbu webových aplikací.

Na základě získaných znalostí navrhněte a naimplementujte webovou aplikaci, která bude generovat neorientované grafy s konkrétními vlastnostmi dle zadaných parametrů. Konkrétně pokryjte minimálně následující typy grafů:

Regulární graf se zadaným počtem uzlů n a stupněm uzlu k.
Graf se zadaným počtem uzlů n a průměrným stupněm uzlu k.
Graf se zadaným počtem uzlů n a minimálním stupněm uzlu k.
Graf se zadaným počtem uzlů n a minimálním stupněm uzlu k a maximálním stupněm uzlu k'.
Úplný graf se zadaným počtem uzlů n.

U všech typů grafů umožněte generování buď neohodnocených grafů, nebo hranově ohodnocených grafů hodnotami ze zadaného intervalu. U prvních 4 typů umožněte volbu souvislosti. Vygenerované grafy uložte do souboru.

Aplikaci řádně otestujte.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 27. září 2022 v Praze.

Bakalářská práce

GENERÁTOR NÁHODNÝCH GRAFŮ

Miroslav Jaroš

Fakulta informačních technologií
Katedra počítačových systémů
Vedoucí: Ing. Michal Šoch, Ph.D.
10. května 2023

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Miroslav Jaroš. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Jaroš Miroslav. *Generátor náhodných grafů.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Poděkování **vii**

Prohlášení **viii**

Abstrakt **ix**

Úvod **1**

1 Cíle práce **3**

2 Náhodné grafy **5**
- 2.1 Teorie grafů . . . 5
  - 2.1.1 Základní definice . . . 5
  - 2.1.2 Souvislost grafu, stromy a cykly . . . 6
  - 2.1.3 Regulární grafy . . . 8
- 2.2 Generování grafů . . . 9
  - 2.2.1 Náhodné regulární grafy . . . 10
  - 2.2.2 Náhodné grafy se stupni v rozmezí . . . 11
- 2.3 Souvislost náhodných grafů . . . 12
  - 2.3.1 Zajištění souvislosti vytvořením kostry . . . 13
  - 2.3.2 Souvislost přepínáním hran . . . 14

3 Analýza problému **17**
- 3.1 Analýza případů užití . . . 17
- 3.2 Očekávaná náročnost řešení . . . 18
- 3.3 Volba technologií . . . 18
  - 3.3.1 Programovací jazyk Go . . . 18
  - 3.3.2 Framework Angular . . . 19
  - 3.3.3 Architektonický styl REST . . . 19
  - 3.3.4 Framework Gin . . . 20
  - 3.3.5 Vestavěná databáze Badger . . . 20

4 Návrh aplikace **21**
- 4.1 Komponenty řešení . . . 21
- 4.2 Backend . . . 21
  - 4.2.1 Zpracování požadavku . . . 22
- 4.3 Návrh REST rozhraní . . . 23
  - 4.3.1 Základní vlastnosti . . . 23
  - 4.3.2 Zdroje využívané v aplikaci . . . 24
  - 4.3.3 Koncové body . . . 25

5 Implementace **29**
5.1 Server . . . 29
5.1.1 Základní datové typy . . . 30
5.1.2 Generování grafů . . . 31
5.1.3 HTTP rozhraní . . . 36
5.1.4 Perzistence dat . . . 39
5.1.5 Konfigurace programu . . . 42
5.1.6 Definice programu . . . 42
5.2 Uživatelské rozhraní . . . 42
5.2.1 Služby a typy pro komunikaci s REST rozhraním . . . 42
5.2.2 Základní obrazovky . . . 43
5.2.3 Formuláře . . . 45

6 Testování aplikace **47**
6.1 Jednotkové testování . . . 47
6.1.1 Testování výkonnosti algoritmů . . . 47
6.2 Integrační testy . . . 48
6.3 Manuální testy . . . 48
6.3.1 Test běhu za reverzní proxy . . . 48
6.3.2 Test akutalizace systému . . . 48
6.3.3 Zahlcení uživatelského rozhraní . . . 48
6.3.4 Zahlcení serveru . . . 49
6.3.5 Test dlouhodobého běhu . . . 49

7 Nasazení řešení **51**
7.1 Možnosti spuštění aplikace . . . 51
7.2 Vývojové sestavení . . . 52
7.2.1 Oddělený běh . . . 53
7.2.2 Vše v jednom . . . 53
7.3 Produkční sestavení . . . 53
7.4 Doporučení pro produkční nasazení . . . 54
7.4.1 Otevření aplikace světu . . . 54
7.4.2 Zabezpečení aplikace . . . 55
7.4.3 Limity aplikace . . . 56

8 Závěr **57**

Obsah přiloženého média **63**

Seznam obrázků

2.1 Souvislý regulární graf se znázorněným mostem 8
2.2 Regulární graf a jeho doplněk 9
2.3 Přepínání u regulárních grafů s lichým stupněm 15

3.1 Diagram případů užití 17

4.1 Sekvenční diagram zpracování požadavku 22

5.1 Intervalový strom pro deset vrcholů se třemi body 34
5.2 Základní obrazovka grafu 43
5.3 Zobrazení detailů požadavku na graf 44
5.4 Základní obrazovka dávek 44
5.5 Zobrazení detailů pro dávku 45
5.6 Formulář požadavku na graf 45
5.7 Formulář požadavku na vytvoření dávky 46
5.8 Neplatně vyplněný formulář 46

Seznam výpisů kódu

1 Vytvoření pole, řezu, mapy a kanálu v jazyce Go 19
2 Požadavek na generování grafu ve formátu JSON 25
3 Struktura požadavku na graf 30
4 Rozhraní překladače vnitřní reprezentace grafu do výstupního formátu 31
5 Rozhraní typu graf. 31
6 Vytvoření nového zdroje náhodných dat 32
7 Vyhledávání bodu v kumulativním počítadle 33
8 Vyhledání bodu v intervalovém stromě 35
9 Rozhraní služby generující grafy 36
10 Rozhraní HTTP služby a její implementace 37
11 Vytvoření Gin router pro REST rozhraní 38
12 Obslužná funkce pro předávání uživatelského rozhraní 39
13 Rozrhaní služby perzistence 40
14 Rozhraní typu klíče pro databázi Badger 40
15 Pomocné funkce pro překlad objektů do/z binární podoby 41
16 Spuštění integračních testů 48
17 Spuštění REST rozhraní pro vývoj 53
18 Spuštění webového serveru, který poskytuje uživatelské rozhraní 53
19 Překlad zdrojových kódů 53

20 Sestavení obrazu kontejnerizačním nástrojem Docker 54
21 Spuštění kontejneru v kontejnerizačním nástroji Docker 54
22 Konfigurace reverzní proxy pro server NGINX 55

Chtěl bych velice poděkovat vedoucímu této práce Ing. Michalu Šochovi, Ph.D., za vynikající spolupráci, otevřený přístup a skvělé vedení, jak v době vývoje a prototypování, tak především v temné době psaní tohoto textu. Taktéž za projevenou trpělivost a nezbytnou přísnost při jejím dokončování. Dále děkuji své manželce Bc. Zuzaně Jarošové za poskytování podpory po celé mé studium, ale především v jeho závěru, který byl pro nás oba velice náročný. V neposlední řadě bych chtěl poděkovat Mgr. Jakubu Smolárovi a Bc. Ondřeji Babcovi za důkladnou revizi textu této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Brně dne 10. května 2023 ....

Abstrakt

Tato bakalářská práce navrhuje webovou aplikaci pro generování náhodných grafů. V první části popisuje základní termíny teorie grafů a pokročilé algoritmy pro generování náhodných grafů. Následně analyzuje zadání z pohledu softwarového inženýrství a navrhuje vytvoření webové aplikace se serverovou částí vystavující REST rozhraní vytvořenou v programovacím jazyce Go a samostatnou komponentou uživatelského rozhraní implementovanou ve frameworku Angular. Praktická část práce poté kombinuje poznatky jak z teoretické části, tak z analýzy, a aplikuje je ve výsledném řešení. Popisuje jeho základní části a věnuje se důležitým prvkům implementace. Následně vysvětluje použité metody testování, a to na několika úrovních: jak jednotkové tak integrační, a zároveň definuje potřebné manuální testy, které byly vykonány pro zajištění kvality řešení. V závěru popisuje doporučený způsob nasazení, provozní limity a postup údržby aplikace pro zajištění jejího výkonu.

Klíčová slova grafy, generátor náhodných grafů, Go, Angular, REST, Webová aplikace

Abstract

This bachelor thesis designs a web application suitable for generation of random graphs. In the first part, it describes basic topics and definitions of Graph Theory and advanced algorithms that are used for the generation. Next, it analyses task description from the software engineering standpoint and designs a web application with a server part written in Go programming language, which exposes REST API, and independent component for user interface, written in the Angular framework.

The practical part of the work then combines knowledge obtained from both theoretical and analytical parts and applies them in the resulting solution. It also describes methods used for application testing on several levels: Unit and integration; it also defines necessary manual test cases executed to assure quality.

Ultimately, it suggests a deployment method, operational limits, and maintenance procedures that ensure the solution's performance.

Keywords graphs, random graph generator, Go, Angular, REST, Web application

Úvod

Při návrhu algoritmů nebo vývoji programů je pro ověření jejich správné funkčnosti dobré mít připravenou vhodnou sadu testovacích dat, na kterých můžeme algoritmus vyzkoušet a ověřit jeho vlastnosti. Nicméně taková vstupní data není většinou jednoduché vytvořit. Pokud tato data vytváří autor algoritmu, může být takový vstup nezáměrně sestavený tak, aby fungoval dobře.

Dalším příkladem může být testování algoritmů pro paralelní výpočty, kde se pro testování používají problémy z třídy složitosti NP-úplných, jako jsou například problémy obchodního cestujícího, nebo čínského listonoše, které oba spadají do matematické disciplíny teorie grafů.

Při ověřování takových algoritmů je tedy vhodné mít dostupný nějaký zdroj vstupů, ideálně náhodných, které ale budou splňovat určitá kritéria. Návrhem takového nástroje se budeme v této práci zabývat.

Struktura textu

Tato práce je členěna do šesti kapitol, které popíší postup od získání teoretického základu, přes analýzu problému, návrh řešení, až po jeho implementaci, testování a nakonec nasazení. V první kapitole provedeme rozbor zadání a stanovíme cíle práce. Zároveň rozebereme současný stav řešení.

V druhé kapitole se podíváme detailně na Teorii grafů, definujeme základní termíny a provedeme výzkum možných postupů, které využijeme při implementaci. Zaměříme se primárně na matematické vlastnosti náhodných grafů, analyzujeme existující algoritmy a pro otevřené otázky navrhneme vlastní.

Ve třetí kapitole se přesuneme k analýze problému z pohledu softwarového inženýrství, provedeme analýzu požadavků, identifikujeme potenciální uživatele a provedeme analýzu případů užití. Zároveň stanovíme očekávanou náročnost řešení z pohledu požadavků na výpočetní zdroje. V neposlední řadě provedeme výběr technologií vhodných pro implementaci řešení a krátce je popíšeme, vysvětlíme základní principy, které tyto technologie používají.

Čtvrtá kapitola obsahuje návrh řešení, zde provedeme konceptuální rozdělení řešení do modulů a přiřadíme jim doménu zodpovědnosti. Popíšeme základní typy objektů vystupujících v systému, jejich interakci a stavy, kterými ve svém životním cyklu procházejí. Dále navrhneme REST rozhraní aplikace, které bude hlavní výkonnou jednotkou zodpovědnou za přijímání a zpracování požadavků. Nakonec se zaměříme i na konceptuální návrh uživatelského rozhraní.

V páté kapitole již rozebereme samotnou implementaci řešení, popíšeme strukturu zdrojového kódu, hlavní balíčky a jejich interakce a zaměříme se především na technologicky zajímavé části. Popíšeme i postup implementace teoretických algoritmů z druhé kapitoly a vysvětlíme.

V šesté kapitole definujeme základní postupy testování aplikace, vysvětlíme průběh testování na různých úrovních a popíšeme manuální testy, které byly vůči řešení vykonány. Poslední ka-

pitolou popíšeme samotné spuštění výsledné aplikace a možnosti nastavení. Zároveň definujeme její limity a uvedeme základní doporučení pro její provoz.

V závěru práce nakonec zhodnotíme finální výstup. Posoudíme nakolik se práci podařilo naplnit zadání a s jakým úspěchem, kde jsou její slabiny a co by bylo vhodné do ní doimplementovat. Také položíme základ k dalšímu rozvoji a vymezíme další vlastnosti, které by se v budoucnu mohli implementovat.

Kapitola 1
Cíle práce

Hlavním cílem této práce je návrh a implementace řešení vhodného pro genrování náhodných grafů, které budou využity při testování paralelních algoritmů v předmětu NI-PDP. Pro jednoduchý přístup studentů k tomuto řešení je navíc zvolena implementace formou webové služby, čímž neklade na uživatele žádné specifické nároky. Uživatel potřebuje pouze webový prohlížeč, který umožní okamžitou interakci s řešením a získání požadovaných grafů.

Zadání jasně specifikuje typy grafů, které má řešení poskytovat. Pro každý graf smí uživatel zadat počet uzlů, které má graf obsahovat, vynutit souvislost grafu a zvolit zda bude výsledný graf hranově ohodnocený, či nikoliv. Dále pro každý požadovaný graf vybírá jednu z následujících kategorií:

- Regulární grafy s vrcholy stupně k.
- Grafy se stupni vrcholů v rozmezí k_m a k_n, kde $k_m < k_n$.
- Grafy v nichž je stupeň vrcholu průměrně a.
- Grafy s minimálním stupněm vrcholu m.
- Úplné grafy.

Zadání také klade důraz na otestování aplikace, tedy součástí řešení bude i sada testovacích scénářů, které verifikují a validují navržený sotfware, případně pomohou s odhalením defektů již v době vývoje. Samozřejmou součástí tohoto testování musí být i ověření nefunkčních požadavků aplikace, tedy například její stability v případě chybových stavů, nebo chování v případě vysokého zatížení uživateli.

Dalším z požadavků je seznámení s moderními trendy a technologiemi vývoje webových aplikací. Na tento požadavek práce zareaguje vhodnou volbou technologií a návrhem řešení, které dodržuje moderní přístupy v tomto odvětví. I proto práce volí architekturu která odděluje aplikační a prezentační logiku do samostatných projeků, jejichž interakce je zajištěna vystavěním rozhraní dle architektonického stylu REST[1].

[1] Representational State Transfer

Kapitola 2
Náhodné grafy

Hlavní náplní této práce je návrh a vytvoření aplikace generující náhodné grafy s vlastnostmi zadanými uživatelem, pro jejíž funkci bude nezbytné využít vhodné algoritmy, které zvládnou řešit tyto problémy s rozumnou efektivitou. Návrh a analýza těchto algoritmů tedy bude vycházet z teorie grafů.

V této kapitole se nejdříve ktátce podíváme na samotnou teorii grafů, definujeme pojmy jako je graf, ohodnocený graf, nebo kostra grafu a odvodíme si některé základní vlastnosti. Následně se podíváme na specifické skupiny grafů (například k-regulární), jejichž generování je cílem práce. V závěru popíšeme algoritmy, které budou využity v implementaci a určíme jejich vlastnosti.

2.1 Teorie grafů

Základy teorie grafů, jako matematické disciplíny, jsou datovány až k Leonardu Eulerovi, který roku 1736 publikoval řešení hádanky „Mostů města Královce“ [1]. Od této doby našla teorie grafů své místo v mnoha odvětvích lidské činnosti. Je velmi vhodným nástrojem pro modelování a není tedy využívána pouze matematiky, ale běžně se objevuje taktéž v chemii, dopravě, modelování procesů a náhodných dějů, a velmi výrazně i v informatice.

2.1.1 Základní definice

Nyní si definujeme několik základních termínů z teorie grafů, na které budeme následně odkazovat při návrhu algoritmů. Následující definice jsou překlady definicí z knihy *Graph theory and its applications* [2] a z kapitoly 1.1 *Fundamentals of Graph Theory* knihy *Handbook of Graph Theory* [3].

▶ **Definice 2.1** (Graf)**.** ***Graf*** *je dvojice $G = (V, E)$, která se skládá ze dvou konečných množin V a E. Prvky množiny V označujeme za vrcholy a prvky množiny E označujeme za hrany. Každá hrana obsahuje dvouprvkovou množinu vrcholů k sobě přiřazených, které nazýváme koncové body hrany, značíme $\mathrm{endpts}(e) = \{u, v\}$, kde $u, v \in V$ jsou koncovými body hrany e.*

▶ **Definice 2.2.** ***Vlastní hrana*** *je hrana, která spojuje dva různé vrcholy.*

▶ **Definice 2.3.** ***Smyčka*** *je hrana, která spojuje jeden koncový bod sama na sebe.*

▶ **Definice 2.4.** ***Vícenásobná hrana*** *je kolekce dvou a více hran, které mají stejné koncové body.*

▶ **Definice 2.5** (Jednoduchý graf)**.** *Graf, který neobsahuje smyčky, ani vícenásobné hrany, označujeme jako* ***jednoduchý graf****.*

Graf je tedy velice jednoduchá algebraická struktura, která zachycuje vazby mezi prvky množiny vrcholů. V odborné liteatuře se běžně navíc rozlišují orientované grafy[1], multigrafy[2] a obecné grafy[3], které se v této práci nicméně nevyskytují, a proto je nemusíme definovat. Pro naše potřeby je tedy termín **graf** označením jednoduchého grafu a takto s ním budeme nadále v textu pracovat.

▶ **Definice 2.6** (Ohodnocený graf)**.** ***Ohodnocený graf*** *je graf, v němž všechny hrany mají přiřazenou číselnou hodnotu zvanou* ***váha****.*

▶ **Definice 2.7** (Sousední vrcholy)**.** *Vrcholy grafu, které jsou spojené hranou, označujeme za* ***sousední vrcholy****.*

▶ **Definice 2.8** (Sousedství)**.** ***Sousedství*** *vrcholu v v grafu G, značíme $N_G(v)$, je množina všech sousedních vrcholů v.* ***Uzavřené sousedství*** *vrcholu v je $N_G[v] = N_G(v) \cup \{v\}$.*

▶ **Definice 2.9** (Incidence)**.** *Pokud je vrchol u koncovým bodem hrany e, potom u označujeme za* ***incidentní*** *hraně e a hranu e za incidentní vrcholu u.*

▶ **Definice 2.10** (Stupeň vrcholu)**.** *Stupněm vrcholu v grafu G, označujeme počet incidentních hran na vrchol v a značíme jej $deg(v)$.*

▶ **Věta 2.11** (Eulerova)**.** *Součet stupňů vrcholů grafu je roven dvojnásobku počtu hran.*

▶ **Důsledek 2.12.** *V libovolném grafu je sudý počet vrcholů s lichým stupněm.*

V tomto bodě již víme, co je graf 2.1, známe termíny jako incidence nebo stupeň vrcholu, s jejichž pomocí nyní můžeme definovat základní typy grafů, se kterými budeme pracovat při náhodném generování.

▶ **Definice 2.13** (Úplný graf)**.** *Jako* ***úplný graf*** *označujeme graf, ve kterém je každá dvojice vrcholů spojena hranou. Úplný graf o n vrcholech značíme K_n.*

▶ **Definice 2.14** (Doplněk grafu)**.** ***Hranový doplňek*** *grafu G je graf $\overline{G}$ nad stejnou množinou vrcholů takový, že vrcholy $\overline{G}$ jsou sousední právě tehdy, když nejsou sousední v G.*

V tomto textu budeme nadále používat pouze termín **doplněk** grafu, nicméně je vždy myšlen **hranový doplněk** grafu definovaný v definici 2.14. Pro úplnost můžeme uvést, že odborná literatura používá také termín *vrcholový doplněk grafu*, se kterým tato práce nijak neoperuje.

▶ **Definice 2.15** (Podgraf)**.** ***Podgraf*** *grafu G je graf H, jehož množiny vrcholů a hran jsou obsaženy v grafu G. Za* ***vlastní podgraf*** *označíme podgraf H grafu G, pokud platí, že množina vrcholů V_H je vlastní podmnožinou V_G a množina hran E_H je taktéž vlastní podmnožinou E_G.*

▶ **Definice 2.16.** *Pro daný graf G,* ***podgraf indukovaný na podmnožině*** *U z V_G, značíme $G(U)$, je podgraf G, jehož množina vrcholů je U a jehož množina hran obsahuje všechny hrany v G takové, že jejich oba koncové body leží na U. Tedy:*

$$V_{G(U)} = U \ \wedge E_{G(U)} = \{e \in E_G | \, \text{endpts}(e) \subseteq U \ \}$$

2.1.2 Souvislost grafu, stromy a cykly

Jedním z atributů, které si může uživatel zvolit pro generované grafy, je jejich souvislost. Intuitivní představa souvislého grafu není složitá. Pokud máme před sebou nakreslený graf, dokážeme většinou poměrně rychle identifikovat, zda mezi všemi dvojicemi vrcholů existuje cesta. Ovšem

[1] Grafy, jejichž hrany mají určený směr
[2] Grafy s vícenásobnými hranami.
[3] Grafy, které připouštějí jak smyčky, tak vícenásobné hrany.

pro odvození pravidel, na jejichž základě budeme takové grafy generovat, musíme nejdříve tyto termíny formalizovat a definovat jejich vlastnosti.

Jako další definujeme komponenty souvisloti, stromy a cykly v grafu, které následně použijeme pro definici algoritmů ověřujících souvislost grafů a případně jaká omezení pro ně existují.

▶ **Definice 2.17.** *V grafu G je **sledem** W z vrcholu v_0 do vrcholu v_n posloupnost*

$$W = \langle v_0, e_1, v_1, e_2, \dots, v_{n-1}, e_n, v_n \rangle$$

střídajících se vrcholů a hran taková, že koncové vrcholy $\text{endpts}(e_i) = \{v_{i-1}, v_i\}$, $\forall i \in \{1, \dots, n\}$.

▶ **Definice 2.18.** ***Délka** sledu W je rovna počtu hran v W.*

▶ **Definice 2.19.** *Sled, který obsahuje jeden vrchol a žádné hrany, označujeme za **triviální sled**.*

▶ **Definice 2.20.** ***Uzavřený sled** je sled, který začíná a končí ve stejném vrcholu. **Otevřený sled** začína a končí v různých vrcholech.*

▶ **Definice 2.21.** *Vrchol v je dosažitelný z vrcholu u, pokud existuje sled z vrcholu u do vrcholu v.*

▶ **Definice 2.22** (Souvislý graf)**.** *Graf G je **souvislý**, pokud pro všechny dvojice vrcholů u a v z G platí, že u je dosažitelný vrchol z vrcholu v.*

Se souvislostí grafů také úzce souvisí termíny cyklus (kružnice), strom a kostra, které si nyní definujeme. Tyto druhy grafu budou následně důležité pro pochopení způsobu jak zajistit, že graf je souvislý, kdy jím být nezbytně musí.

▶ **Definice 2.23.** ***Tah** je sled, ve kterém se neopakují hrany.*

▶ **Definice 2.24** (Cesta)**.** *Tah, ve kterém se neopakují vrcholy, označujeme jako **cestu** (s možnou výjimkou počátečního a koncového vrcholu, které mohou být totožné).*

▶ **Definice 2.25.** *Sled, tah nebo cestu označíme za **triviální** pouze pokud obsahují právě jeden vrchol a žádné hrany.*

▶ **Definice 2.26** (Cyklus)**.** *Netriviální uzavřená cesta se nazývá **cyklus** (taktéž označovaný jako **kružnice**).*

▶ **Definice 2.27.** *Graf, který neobsahuje cykly, označujeme jako **acyklický graf**.*

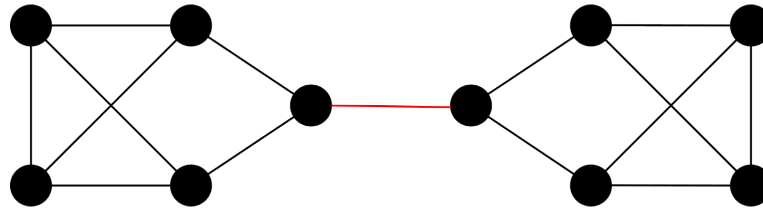
▶ **Definice 2.28** (Strom)**.** ***Strom** je souvislý acyklický graf.*

▶ **Definice 2.29** (Komponenta souvislosti)**.** ***Komponenta souvislosti** grafu G je libovolný maximální souvislý podgraf G. Jinými slovy souvislý podgraf H je komponentou souvislosti grafu G, pokud H není vlastním podgrafem jakéhokoliv jiného souvislého podgrafu G.*

▶ **Definice 2.30** (Most)**.** ***Hranový řez** v grafu G je množina hran D taková, že $G - D$ má více komponent než G. Hrana e je **mostem** v grafu G, pokud $\{e\}$ je hranovým řezem.*

Jinými slovy můžeme říct, že most je hrana, jejíž odstranění zvýší počet komponent souvislosti v grafu o jedna. Příkladem může být graf na obrázku 2.1, kde vidíme červeně znázorněný most, tedy hranu po jejímž odstranění by se graf rozpadl na dvě komponenty souvislosti.

Dalším pozorováním je, že pro všechny stromy platí, že každá jejich hrana je zároveň mostem. Pro každou hranu ve stromě platí, že neleží na žádné kružnici, tedy mezi koncovými body u a v libovolné hrany e stromu, existuje právě jedna cesta, která obsahuje pouze hranu e. Jejím odstraněním by se vrchol u stal nedosažitelným z v, respektive vrchol v by se stal nedosažitelným z u, a tedy podmínka souvislosti by byla porušena.



Obrázek 2.1 Souvislý regulární graf se znázorněným mostem

▶ **Definice 2.31.** ***Kostrou grafu*** *G označujeme takový podgraf G, že jeho množina vrcholů je rovná množině vrcholů z G a zároveň je stromem.*

▶ **Definice 2.32.** ***Hranová souvislost*** *grafu G, značíme $\kappa_e(G)$, je nejmenší počet hran, jejichž odstranění by rozpojilo graf G. Graf G je* ***hranově k-souvislý****, pokud G je souvislý a každý hranový řez obsahuje nejméně k hran (tedy $\kappa_e(G) \geq k$).*

Pro úplnost uvádíme, že podobně jako v předchozích případech, definuje teorie grafů i termín *vrcholové souvislosti*. Stejně jako v předchozích, není tento termín pro tuto práci relevantní, zároveň platí, že pokud v textu mluvíme o souvislém nebo k-souvislém grafu, myslíme vždy graf hranově k-souvislý.

2.1.3 Regulární grafy

Regulární grafy jsou těžištěm této práce, proto jim věnujeme samostatnou sekci, jejich vlastnosti totiž budou podstatné pro následující části této práce.

▶ **Definice 2.33** (Regulární graf)**.** *Libovolný graf, jehož vrcholy mají stejný stupeň, nazýváme* ***regularním grafem****. Jako k-****regulární*** *graf označujeme graf, jehož všechny vrcholy mají stupeň k.*

▶ **Důsledek 2.34.** *Každý úplný graf je zároveň regulární. Protože každý vrchol grafu K_n je sousední všem ostatním vrcholům, platí, že jeho stupeň je roven $n-1$, tedy lze říci, že K_{n+1} je n-regulární graf.*

Dalším pozorováním je, že K_{k+1} je nejmenším možným k-regulárním grafem, tedy nelze sestavit k-regulární graf na méně než $k+1$ vrcholech. V tomto bodě je na čase uvést důležité tvrzení této práce, protože následující důkaz regularity doplňku regulárního grafu bude stěžejním tvrzením pro generování regulárních grafů.

▶ **Tvrzení 2.35.** *Hranový doplňek regulárního G grafu je regulární graf.*

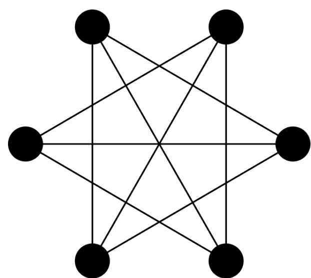
Důkaz. Nechť G je k-regulární graf s počtem vrcholů rovným n, množinou vrcholů V, a graf $\overline{G}$ je jeho doplňkem sestavený nad stejnou množinou vrcholů V. Pro každý vrchol $v \in G$ platí $\deg(v) = k$. Z definice hranového doplňku 2.14 můžeme odvodit:

$$N_{\overline{G}}(v) = V \setminus N_G[v], \; \forall v \in V$$

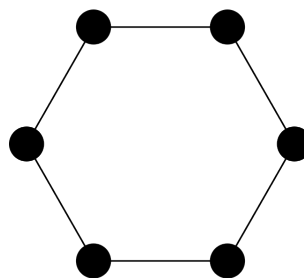
Tedy pro všechny vrcholy v platí, že jejich sousedství v $\overline{G}$ je doplňkem uzavřeného sousedství vrcholu v vůči V. Z definice regulárního grafu 2.33 plyne, že $|N_G(v)| = k$ pro všechna $v \in V$. Potom nezbytně platí, že:

$$|N_{\overline{G}}(v)| = |V| - |N_G[v]| = n - (k + 1), \; \forall v \in V$$

Což znamená, že velikost sousedství všech vrcholů v doplňku grafu je rovna stejné hodnotě $\overline{k} = n - (k+1)$ a tedy graf $\overline{G}$ je $\overline{k}$-regulární. ◀



(a) 3-regulární graf na 6 vrcholech



(b) Doplňěk regulárního – 2-regulární graf

Obrázek 2.2 Regulární graf a jeho doplněk

Tvrzení 2.35 využijeme u generování regulárních grafů, protože nám umožňuje redukovat problém generování k-regulárních grafů s $k > \frac{n}{2}$ na problém generování regulárního grafu s $k' = n - (k + 1)$, k němuž následně vytvoříme jeho doplněk, čímž získáme hledaný k-regulární graf.

▶ **Tvrzení 2.36.** *Nechť k je libovolné sudé číslo $k \geq 2$. Souvislý k-regulární graf je zároveň nejméně hranově 2-souvislý.*

Důkaz. Předpokládejme, že tvrzení neplatí. Tedy existuje k-regulární graf G se sudým k, který je 1-souvislý. Z toho plyne, že takový graf obsahuje nejméně jednu mostovou hranu e, po jejímž odstranění by došlo k rozpadnutí G na dvě komponenty souvislosti. Označme vrcholy

$$u, v \in \text{endpts}(e),\ \deg(u) = \deg(v) = k$$

koncové body hrany e. Pro ně platí, že mají stejný stupeň rovný k. Po odstranění mostové hrany e se jejich stupeň sníží o jedna, tedy na liché číslo a v grafu G vzniknou dvě komponenty souvislosti G_u a G_v. Z důsledku 2.12 víme, že jak v G_u, tak v G_v musí existovat druhý vrchol s lichým stupněm. To je v rozporu s předpokladem k-regularity grafu, a tedy G musí být nejméně 2-souvislý. ◀

2.2 Generování grafů

Se znalostí základních pojmů teorie grafů se nyní zaměříme na základní otázku potřebnou pro generování náhodných grafů. Pokud bychom měli za cíl generovat pouze jednoduché grafy bez dalších podmínek, byl by generující algoritmus poměrně přímočarý.

Jedním z možných přístupů je projít graf po dvojicích vrcholů a „hodit mincí“, pokud padne hlava, vrcholy spojíme hranou, jinak je necháme rozpojené. Tento přístup by nicméně generoval v jistém smyslu podobné grafy, protože počet hran by ve střední hodnotě byl roven $\frac{n(n-1)}{4}$. Další možností by bylo nejdříve si náhodně vybrat počet hran grafu, u kterého víme že je nejvýše roven $\frac{n(n-1)}{2}$ a náhodně vybírat dvojice vrcholů, dokud graf neobsahuje zvolený počet.

Nicméně pro grafy s určenými vlastnostmi se situace komplikuje. Náhodné vybírání hran může vést ke grafům, které nesplňují podmínky, což by v takovém případě vedlo k opakování pokusu, dokud náhodný graf tyto podmínky nebude splňovat. Tento postup by nicméně mohl vést k nekonečné smyčce. Proto si druhy grafů definované v kapitole 1 nyní kategorizujeme z pohledu přístupu a náročnosti.

- **Regulární grafy** jsou na vygenerování nejtěžší, čistě náhodný přístup při vybírání hran způsobuje vytváření grafů, které porušují podmínku regularity, proto je potřeba využít specializovaný algoritmus, který je vygeneruje. Tomu se věnuje sekce 2.2.1.
- **Grafy se stupni v rozmezí** lze redukovat na problém generování regulárních grafů. Z grafu se stupni v daném rozmezí lze[4] odebíráním hran získat k_m-regulární graf, se stupněm rovným

[4] Až na výjimky, které definujeme dále.

minimálnímu požadovanému stupni. Tedy, při generování můžeme začít pokusem o vytvoření regulárního grafu a následně přidat hrany, které splní podmínku grafu.

- **S minimálním stupněm** lze redukovat na problém generování grafů se stupni v rozmezí k a $n-1$ pro grafy s minimálním stupněm k a počtem vrcholů n.
- **S průměrným stupněm** jsou pro vygenerování jednoduché, stačí do grafu náhodně vložit $\frac{a \cdot n}{2}$ hran, kde a je průměrný stupeň a n počet vrcholů. Tedy algoritmus s náhodným výběrem hran, při náhodném průchodu vrcholů bude dostačující a rozumně efektivní.
- **Úplné grafy** jsou na vygenerování triviální – pro každý graf G na n vrcholech existuje přesně jeden graf splňující tuto vlastnost a to K_n. Vzhledem k Důsledku 2.34 zároveň víme, že se tento problém dá redukovat na problém generování regulárních grafů.

Tedy odvozujeme, že hlavním problémem, který potřebujeme pro implementaci této práce je efektivní generování regulárních grafů. S takovým algoritmem následně dokážeme redukcí problému najít řešení i pro ostatní požadované druhy grafů.

2.2.1 Náhodné regulární grafy

Generování náhodných regulárních grafů je komplikovaným problémem, který je řešen již dlouho a stále nemá jednoznačnou odpověď [4]. V průběhu let bylo nalezeno několik algoritmů, které dokážou takové grafy generovat, ale většinou s poměrně nízkou efektivitou, nebo pro omezenou podmnožinu řešení.

Pro ukázku složitosti tohoto problému můžeme uvést výzkum Markuse Meringera, který ve své práci „Fast Generation of Regular Graphs and Construction of Cages“ [5] popisuje algoritmus generující všechny neizomorfní regulární grafy. Na svých stránkách[5] potom sbírá výsledky generování, které prováděl jak on sám, tak i další lidé. Z jeho výsledků můžeme pozorovat, že problém je komplikovaný i pro relativně malá čísla. Například pro 6-regulární grafy s počtem vrcholů 15 existuje 1 470 293 675 souvislých grafů.

2.2.1.1 Steger-Wormaldův algoritmus

Steger-Wormaldův algoritmus byl popsán v práci *Generating random regular graphs quickly* [4]. Nicméně v době publikování algoritmu nebylo ještě jasné, zda tento algoritmus generuje regulární grafy uniformně. Tato vlastnost algoritmu byla následně dokázána v práci *Generating Random Regular Graphs* [6], kde je dokázáno, že algoritmus generuje asymptoticky uniformní náhodné regulární grafy v polynomiálním čase a se složitostí rovnou $\mathcal{O}(nk^2)$ pro $k \in \mathcal{O}(n^{1/3})$.

Vhodnost dvojice bodů není přímo v algoritmu 2.1 definována. Nicméně Steger a Wormald vysvětlují, že dvojice bodů je vhodná, pokud není ze stejné skupiny bodů a v množině C vybraných dvojic již neleží dvojice ze stejných skupin [4].

2.2.1.2 Oprava selhání Steger-Wormaldova algoritmu

Algoritmus 2.1 negarantuje sestavení regulárního grafu především z důvodu realizace náhodného výběru a může selhat, což vede k potřebě algoritmus znovu spustit. Zvláště problematický je potom tento postup u grafů se stupněm $k > n^{1/3}$. V tomto případě může časté selhávání algoritmu způsobovat potenciální zacyklení generátoru.

Proto je potřeba se nějakým způsobem pokusit tyto problémy odstranit, k čemuž využijeme ideu z práce *Uniform generation of random regular graphs of moderate degree* [7].

Jedno ze selhání algoritmu, kdy z množiny bodů již nelze vybrat vhodnou kandidátní hranu, je situace existence volných bodů u již propojených hran, mezi kterými nemůžeme již hranu vytvořit, protože by vedla ke vzniku vícenásobné hrany.

[5] `http://www.mathe2.uni-bayreuth.de/markus/reggraphs.html`

StegerWormald(n, k)

1 Vytvoř U množinu bodů $\{1, 2, \ldots, nk\}$ rozdělenou do n skupin.
2 Vytvoř C množinu dvojic bodů z U.
3 **Opakuj** dokud množina U obsahuje vhodné dvojice bodů:
4 Vyber dva náhodné body i a j z U.
5 **Pokud** jsou i a j vhodné:
6 Smaž i a j z U a vlož $\{i, j\}$ do C.
7 Sestav graf G s hranami z vrcholu r do vrcholu s právě tehdy, když v C existuje dvojice bodů z r-té a s-té skupiny.
8 **Pokud** G není regulární:
9 **Opakuj** algoritmus se stejnými vstupy n a k.
10 **Jinak** vrať G.

Algoritmus 2.1 Steger-Wormaldův algoritmus generující regulární grafy.

V tomto případě můžeme zkusit zasáhnout do existující množiny kandidátních hran a pokusit se změnit některé existující hrany tak, aby byl problém odstraněn. Tedy pokud dojde k situaci, kdy z množiny bodů vybereme pouze již propojené vrcholy, aplikujeme postup popsaný v algoritmu 2.2.

OpravBody(C, u, v)

1 Sestav množinu M vrcholů, které nejsou incidentní jak u, tak v.
2 Náhodně vyber hranu e z množiny hran propojujících vrcholy v M.
3 Označ c_1 a c_2 koncové body e a odstraň e z C.
4 Vlož do C hrany $\{u, c_1\}$ a $\{v, c_2\}$.

Algoritmus 2.2 Opravení

Algoritmus 2.2 simuluje přidání této násobné hrany. Tedy před spuštěním opravy se chováme, jako bychom přidali hranu mezi již sousední u a v. Následně nalezneme všechny vrcholy, které nejsou v sousedství jak u, tak v a identifikujeme všechny jejich hrany. Jednu z těchto hran odstraníme a její koncové body propojíme s u respektive v.

Odebráním hrany jsme snížili stupeň jejích koncových bodů o jedna, nicméně vzhledem k výběru vrcholů, víme že každy z nich můžeme propojit jak s u, tak s v a tedy přidání takových dvou hran nezpůsobí vznik vícenásobné hrany a zároveň vrátí stupně zvolených vrcholů na původní hodnotu.

V případě, že je množina nesousedních vrcholů prázdná, nebo na této množině neexistují hrany, potom signalizujeme zpět selhání a tedy opakujeme celý proces generování od začátku novým spuštěním Steger-Wormaldova algoritmu.

2.2.2 Náhodné grafy se stupni v rozmezí

V úvodu této sekce jsme definovali, že při generování grafů se stupni v rozmezí k_m a k_n můžeme zvolit přístup vygenerování k_m regulárního grafu, ke kterému následně přidáme náhodný počet hran, mezi vrcholy, které neporuší podmínku maximálního stupně. Z definice víme, že počet hran v tomto grafu bude v rozmezí:

$$|V| = \left(\frac{nk_m}{2}; \frac{nk_n}{2} \right)$$

Nicméně v tomto případě nastává problém, že základní regulární graf stupně k_m může být nesestrojitelný v případě, kdy je součin k_m a n počtu vrcholů lichý. Tedy úplně naivní přístup není zcela aplikovatelný. Proto zvolíme spíše přístup úpravy algoritmu pro generování regulárních grafů.

UPRAVENÝSTEGERWORMALD(n, k_m, k_n)

1 Vytvoř U množinu bodů $\{1, 2, \ldots, nk_m\}$ rozdělenou do n skupin.
2 Vytvoř U' množinu bodů $\{nk_m + 1, nk_m + 2, \ldots, nk_m + n(k_n - k_m)\}$ rozdělenou do n skupin.
3 Vytvoř C množinu dvojic bodů z U.
4 **Opakuj** dokud množina U není prázdná:
5 Vyber dva náhodné body i a j z U.
6 **Pokud** jsou i a j vhodné:
7 Smaž i a j z U a vlož $\{i, j\}$ do C.
8 **Jinak**: vyber bod o z U'.
9 **Pokud** jsou body i a o vhodné:
10 Vlož $\{i, o\}$ do C a odstraň i z U.
11 **Pokud** má skupina s odpovídající bodu o volné body v U:
12 Odstraň jeden bod z U odpovídající skupině s.
13 **Jinak** odstraň o z U'.
14 **Jinak** opakuj postup s j místo i.
15 Sestav graf G s hranami z vrcholu r do vrcholu s právě tehdy, když v C existuje dvojice bodů z r-té a s-té skupiny.
16 Náhodně zvol e cílový počet hran $e \in \langle \frac{nk_m}{2}; \frac{nk_n}{2} \rangle$.
17 Přidej hrany do G mezi vrcholy se stupněm menším než k_n dokud v grafu není e hran.

Algoritmus 2.3 upravený Steger-Wormaldův algoritmus generující grafy v rozmezí stupňů.

V algoritmu 2.3 vidíme potřebnou úpravu, která zajistí vygenerování hledaného grafu. Hlavní změnou oproti algoritmu 2.1 je v přidání druhé množiny bodů. Stejně, jako v původním algoritmu drží množina U informaci o volných hranách pro každý vrchol, je množina U' určena, aby držela informaci o volných hranách do maximální úrovně u každého vrcholu.

Základní idea algoritmu je vygenerování grafu, který má stupeň nejméně k_m a v případě, že nedokáže v U najít vhodnou dvojci vrcholů zajišťující k_m regularitu, hledá v množině bodů U', která zajišťuje, že stupně v grafu G nepřesáhnou k_n. Tento postup zároveň výrazně snižuje pravděpodobnost selhání, protože přidává stupeň volnosti při výběru, ale současně vynucuje primární snižování bodů v U.

Takto vygenerovaný graf splňuje podmínky třídy grafu, nicméně počet hran, které je možné v grafu vytvořit není fixní (naorzdíl od regulárních grafů), proto je pro zajištění různosti grafů ještě vybrán náhodný počet hran z přípustného rozmezí a graf je na tento počet doplněn.

2.3 Souvislost náhodných grafů

Pro zajištění souvislosti grafů při jejich generování existují dva základní přístupy:

- Začít generování grafu s existujícím stromem a přidat do něj hrany dle potřeby.
- Po vygenerování grafu identifikovat kompoenenty souvislosti a propojit je hranami.

První přístup garantuje souvislost bez ohledu na stav generátoru náhodných čísel, protože generátor upravuje již souvislý graf. Nicméně pokud je algoritmus, který přidává hrany (jako

například u algoritmu Steger-Wormald), závislý na stavu současného grafu, může tento přístup ohrozit jeho stabilitu, například zvýšením počtu selhání.

Druhý přístup nechává plnou volnost generujícímu algoritmu a dodává souvislost až po provedeném generování, nicméně pro grafy s fixními vlastnostmi může způsobit jejich porušení, například u regulárních grafů nelze již nové hrany po generování přidávat, protože každý vrchol je plně saturovaný.

Z vlastností stromů víme, že každý strom má průměrný stupeň vrcholu nejvýše limitně se blížící 2 v závislosti na n (přesný průměrný stupeň je vždy $2 \cdot \frac{n-1}{n}$).

2.3.1 Zajištění souvislosti vytvořením kostry

V definici 2.31 jsme určili, že kostrou grafu je strom, jehož množina vrcholů je totožná s množinou vrcholů grafu. Obecný problém nalezení minimální kostry grafu – tedy kostry takové, že její hranové ohodnocení je minimální – je velice starým a mnohokrát řešeným problémem.

Vůbec první algoritmus je pro nás obzvláště zajímavý, protože pochází z Československé republiky. Roku 1926 definoval ve své práci „O jistém problému minimálním“ český matematik Otakar Borůvka [8] algoritmus pro vyřešení problému navržení elektrické sítě na jižní moravě.

Pro tuto práci je tento algoritmus zajímavý kvůli přístupu, kterým řeší hledání a propojování hran. Princip fungování algoritmu je v prvním kroku rozdělení po vrcholech na jednotlivé komponenty souvislosti.

V každé iteraci následně algoritmus projde všechny hrany a identifikuje ty, které spojují komponenty mezi sebou. Po této identifikaci pro každou komponentu vybere tu nejlevnější hranu, která byla identifikována a přidá ji do grafu, čímž spojí dvě komponenty. V závěru je v grafu právě jedna komponenta souvislosti, která je stromem s minimálním hranovým ohodnocením.

Na výše uvedeném postupu si můžeme všimnout jedné podstatné vlastnosti algoritmu, která bude pro naše řešení zajímavá, narozdíl od ostatních algoritmů hledání minimální kostry grafu (jako je například Jarníkův[6], či Kruskalův) [9] je totiž primárně zaměřený na komponenty grafu, tedy jeho základní idea je vhodná pro použití při sestavování náhodných stromů, protože nám umožní realizovat náhodný výběr nad vrcholy.

UPRAVENÝBORŮVKA(n, k)

1 Sestav množinu V komponent souvislosti a vlož do ní n samostatných vrcholů.
2 **Dokud** ve V není právě jedna komponenta souvislosti opakuj:
3 Vyber dvě náhodné komponenty A, B z V a odstraň je z V.
4 Vyber náhodné vrcholy a z A a b z B takové,
že jejich stupeň je menší nebo roven k.
5 Vytvoř novou komponentu C z A a B spojením vrcholů a a b hranou.
6 Vlož komponentu C do V.
7 Vrať jedinou komponentu V jako graf.

Algoritmus 2.4 Upravený Borůvkův algoritmus pro generování náhodného stromu o n vrcholech s nejvyšším stupněm k

Algoritmus 2.4 konceptuálně vychází z Borůvkova algoritmu a staví na principu postupného spojování komponent grafu, dokud nejsou propojeny cestou. Implicitním předpokladem je, že každá komponenta grafu obsahuje nejméně jeden vrchol se stupněm, který je menší než k. Lze ovšem triviálně dokázat, že pro každý strom (jimiž tyto komponenty z definice jsou) platí, že obsahuje nejméně jeden vrchol stupně 1.

[6]Což je vůbec druhý algoritmus hledání minimální kostry objevený v roce 1930, následně znovuobjevený Robertem Primem, po němž nese většinou jméno ve světové literatuře.

Podmínkou algoritmu je, že k je ostře menší než n, protože nejvyšší přípustný stupeň vrcholu v libovolném grafu je vždy $n - 1$ a ostře vyšší než 1, s výjimkou $n = 2$ což je jediný souvislý graf na dvou vrcholech.

2.3.2 Souvislost přepínáním hran

U regulárních grafů zvolíme přístup přepínáním hran 2.5. Výhodou tohoto algoritmu je, že zachovává stupeň vrcholů, nevýhodou je, že může vést k rozpojení původních komponent souvislosti.

SpojPřepnutím(G)

1 Do množiny F vlož všechny kompoentny souvislosti z G.
2 **Dokud** množina F neobsahuje právě jeden prvek:
3   Vyber náhodně U a V komponenty souvislosti z F a odstraň je z F.
4   Vyber náhodně hranu u z U a označ její koncové body u_1 a u_2.
5   Vyber náhodně hranu v z V a označ v_1, v_2 její koncové body.
6   Odstraň u a v z G.
7   Vytvoř hrany $\{u_1, v_1\}$ a $\{u_2, v_2\}$ v G.
8   Vlož novou komponentu $U \cup V$ do F.
9 **pokud** je počet komponent souvislosti G větší než jedna.
10   **opakuj** algoritmus se vstupem G.
11 **jinak** Vrať G.

Algoritmus 2.5 Algoritmus spojující komponenty souvislosti přepínáním hran.

Selhání algoritmu nastává v okamžiku, kdy vybrané náhodné hrany pro přepnutí jsou zároveň mosty (viz obrázek 2.1). Ačkoliv vytvořením hran z u_1 do v_1 dojde ke spojení komponent U a V, odebrání hrany u, která je zároveň mostem v U způsobí rozpad komponenty U.

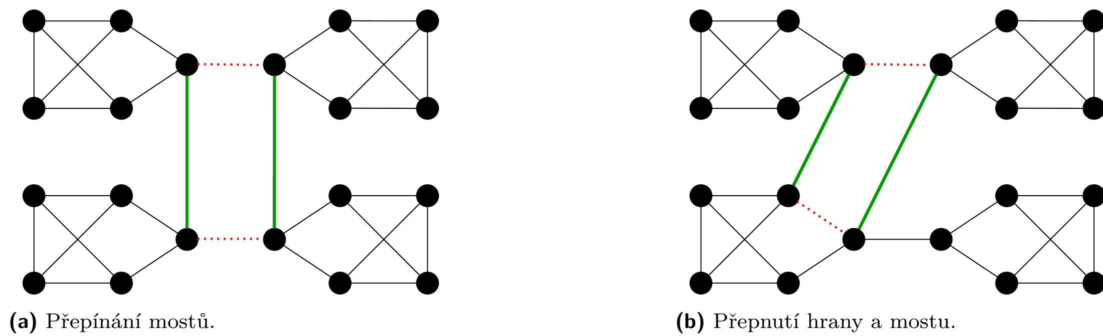
▶ **Tvrzení 2.37.** *Pro k-regulární grafy se sudým stupněm k, algoritmus spojující komponenty přepínáním neselže.*

Důkaz. Selhání algoritmu nastává pouze v případě, kdy jsou vybranými hranami mosty v komponentách souvislosti U a V. Z tvrzení 2.36 víme, že souvislý k-regulární graf se sudým k je také 2-souvislý. Z toho plyne, že neobsahuje hrany, které by byly mosty. Tedy algoritmus nemůže vybrat mostovou hranu a tím pádem selhat. ◀

Z tvrzení 2.37 usuzujeme, že algoritmus 2.5 neselže, pokud má na vstupu regulární graf se sudým stupněm. Nicméně na ostatních regulárních grafech může selhat. Důležitým pozorováním je, že i v případě selhání nedojde ke změně počtu komponent souvislosti. Tedy v případě, že algoritmus rozpojí komponentu odebráním mostu, bude tato komponenta připojena k jiné. Na obrázku 2.3a vidíme výsledek přepínání při výběru dvou mostů. Ačkoliv obě komponenty grafu byly rozděleny odebráním mostu, přepnutí hrany je propojilo s jinou, tedy výsledkem jsou stále dvě komponenty grafu.

V případě výběru pouze jedoho mostu, dojde k připojení nově vzniklých komponent k jedné komponentě a tedy výsledkem se stane právě jedna komponenta souvislosti, jak je vidět na obrázku 2.3b.

V případě grafů s lichým stupněm je tedy jeho použitelnost sporná, vzhledem k tomu, že negarantuje v každém cyklu spojení komponent, teoreticky může dojít k jeho zacyklení, v případě že bude volit pouze mosty. Tedy jeho efektivita závisí na počtu mostů, které se vyskytují v komponentě grafu, protože každý most zvyšuje pravděpodobnost selhání při spojování.



(a) Přepínání mostů.

(b) Přepnutí hrany a mostu.

Obrázek 2.3 Přepínání u regulárních grafů s lichým stupněm

Jednou z možností jak tento problém řešit je rozšíření algoritmu o kontrolu vybraných hran, zda nejsou mosty. Potom by algoritmus garantovaně snižoval v každé iteraci počet komponent a nezbytně by skončil. Problémem této možnosti je její složitost, hledání mostů v grafu je problém s asymptotickou složitostí $\mathcal{O}(ne)$, kde n je počet vrcholů a e je počet hran, tedy v případě k-regulárních grafů bude jeho složitost rovna $\mathcal{O}(n\frac{kn}{2}) = \mathcal{O}(n^2k)$, což je složitost odpovídající problému nalezení komponent grafu, tedy výsledná složitost algoritmu by v tomto případě byla rovna:

$$\mathcal{O}(n^2k) + \mathcal{O}(n) \cdot \mathcal{O}(n^2k) = \mathcal{O}(n^2k) + \mathcal{O}(n^3k) = \mathcal{O}(n^3k)$$

Druhou možností je akceptovat možná selhání a pouze algoritmus zopakovat, efektivita algoritmu bude v ten okamžik ovlivněna pouze pravděpodobností vybrání mostu, a náhodností výběru hrany.

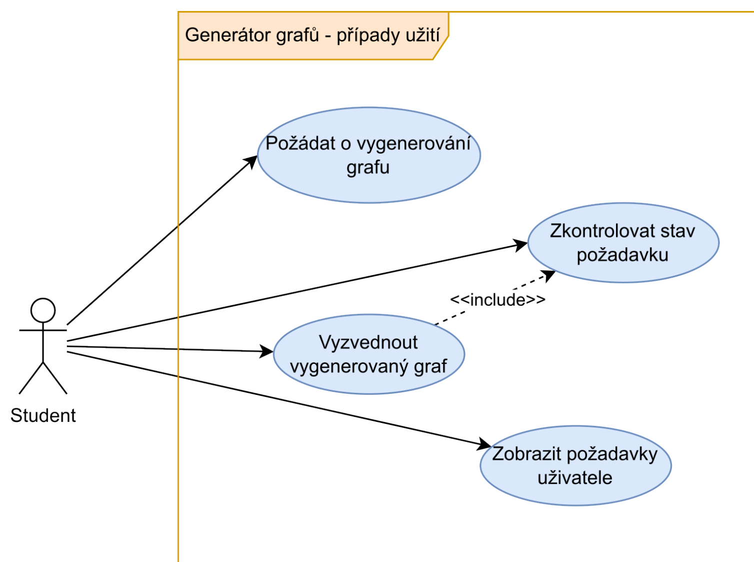
Kapitola 3

Analýza problému

V této kapitole se zaměříme na softwarovou analýzu cílů práce, která nám určí základní rámec pro návrh aplikace. Zároveň provedeme volbu technologií, které nám umožní vystavět vhodné řešení a popíšeme architektonický styl REST, který je v řešení využit jako základní rozhraní celé aplikace. Dále také popíšeme nástroje a frameworky, které budou pro jeho vytvoření použity.

3.1 Analýza případů užití

Jedním z prvních kroků při analýze zadání problému je zjištění zamýšleného používání systému jeho uživateli. Tomuto procesu říkáme analýza případů užití. V jejím rámci identifikujeme aktéry a akce, které aktéři požadují po systému. Jediným aktérem v tomto systému bude student, který požaduje vygenerování grafu.



Obrázek 3.1 Diagram případů užití

Požádání o generování grafu Systém musí poskytovat rozhraní, které umožní studentovi specifikovat všechny požadované druhy grafů a zároveň i všechny atributy, které je nastavují.

Zkontrolovat stav požadavku V případě úspěšného vytvoření požadavku musí systém umožnit studentovi ověřit jeho současný stav.

Zobrazit požadavky uživatele Systém musí být schopen poskytnout studentovi informace o všech platných požadavcích evidovaných v okamžiku vykonání akce.

Vyzvednout vygenerovaný graf V případě kdy je stav požadavku označený za splněný, musí systém umožnit studentovi vyzvednout graf v požadovaném formátu.

3.2 Očekávaná náročnost řešení

V kapitole 2 jsme popsali základní algoritmy, které budou vhodné pro řešení zadaných problémů. Steger-Wormaldův algoritmus má asymptotickou složitost rovnou $\mathcal{O}(nk^2)$, tedy pro $k \in \mathcal{O}(n)$ bude složitost samotného generování $\mathcal{O}(n^3)$ bez selhání. V případě $k > n^{1/3}$ nicméně roste pravděpodobnost selhání a dochází ke spuštění opravujícího algoritmu, který má složitost $\mathcal{O}(n^2)$. Tedy výsledná složitost by měla být v nejhorším případě $\mathcal{O}(n^5)$.

Upravený Steger-Wormaldův algoritmus nespouští opravný algoritmus, tedy jeho složitost je limitně $\mathcal{O}(n^3)$ při vysokých hodnotách k_m a k_n. Pro grafy s průměrným stupněm je složitost rovna $\mathcal{O}(k_a n) \approx \mathcal{O}(n^2)$.

Paměťová náročnost se bude odvíjet od zvolené reprezentace grafu. Pro rámcové určení můžeme vyjít z limitního případu úplného grafu. V takovém případě bude objem paměti potřebné pro uložení grafu závislý na počtu hran, který je roven $\binom{n}{2} = \frac{n(n-1)}{2}$. Tedy asymptoticky budeme potřebovat $\mathcal{O}(n^2)$ paměti pro uložení grafu na n vrcholech.

3.3 Volba technologií

Nyní popíšeme technologie, které budou využity pro vytvoření aplikace. Zároveň se zaměříme na architektonický styl REST a technologie, které použijeme pro implementaci rozhraní v tomto stylu.

3.3.1 Programovací jazyk Go

Vznik programovacího jazyka Go se datuje do roku 2008, kdy jej ve společnosti Google začali vyvíjet Robert Griesemer, Rob Pike a Ken Thompson [10]. Mezi největší projekty napsané v tomto jazyce patří například kontejnerizační engine Docker [11] nebo orchestrační nástroj Kubernetes [12].

Programovací jazyk Go je striktně typovaný programovací jazyk, kompilovaný do strojového kódu cílové platformy s automatickou správou paměti pomocí garbagge collectoru.

3.3.1.1 Typový systém

Typový systém jazyka Go definuje základní primitivní datové typy, jako například `int`, `float`, `byte`, datové typy fixní šířky `int64`, `uint64`, `float64` a speciální typy `string` a `rune`. Dále poskytuje nástroje pro vytváření uživatelem definovaných typů, struktury `struct`) a rozhraní `interface`. Nový typ definuje klíčové slovo `type`. Uživatel může vytvořit nový typ také přejmenováním již existujícího.

3.3.1.2 Řezy, mapy a kanály

Jazyk Go na úrovni syntaxe jazyka poskytuje čtyři základní typy kolekcí: pole, řezy, mapy a kanály. Pole je kolekce objektů stejného typu s fixním počtem prvků, které jsou v jeho rámci

přístupné pomocí indexu. Velikost pole je součástí jeho typu a tím pádem jeho velikost nelze za běhu měnit.

```
var myArray [5]int
var mySlice []int = make([]int, 10)
var myMap map[int]int =  make(map[int]int)
var channel chan bool = make(chan bool)
```

Výpis kódu 1 Vytvoření pole, řezu, mapy a kanálu v jazyce Go

Pole variabilní délky se nazývají řezy (v angličtině slices). Jejich velikost lze v době běhu měnit. Řez má dva základní atributy: počet prvků a kapacitu. Nový řez se vytváří funkcí `make`, která pro řez akceptuje dva nebo tři argumenty. Prvním argumentem je typ řezu, druhým je délka řezu a třetím (volitelným) je kapacita. Délka řezu specifikuje, kolik je v řezu uložených hodnot. Kapacita určuje, kolik prostoru v operační paměti má řez alokováno.

Mapa je typ, který asociuje dva různé typy, jeden v roli klíče, druhý v roli hodnoty. Stejně jako řezy se vytváří funkcí `make`, která v tomto případě akceptuje pouze jeden argument, typ mapy.

Kanál je zvláštní druh fronty, který je primárně určen pro výměnu dat mezi paralelně běžícími částmi aplikace. Čtení a zápis se provádí operátorem `<-` a je blokující, tedy gorutina zapisující do kanálu čeká, dokud data nejsou z kanálu přečtena, respektive přebírající gorutina čeká, dokud se data v kanálu neobjeví.

3.3.1.3 Gorutiny

Mezi základní konstrukty jazyka patří podpora koprogramů přímo na úrovni syntaxe jazyka. Těmto koprogramům se říka gorutiny [13]. Vytvoření nové gorutiny je provedeno klíčovým slovem jazyka `go`.

3.3.2 Framework Angular

Framework Angular je vývojovým frameworkem postaveným nad jazykem TypeScript [14]. Základními konstrukty frameworku jsou komponenty a služby. Komponenta je kolekce definicí HTML šablony, kaskádového stylu (CSS) a obslužného kódu v TypeScript, která definuje zobrazovaný prvek v rámci uživatelského rozhraní. Služba je objekt, který je v rámci aplikace instanciován právě jednou a poskytuje komunikační rozhraní mezi komponentami nebo s REST rozhraním [15].

3.3.3 Architektonický styl REST

Representational State Transfer (zkráceně REST) je architektonický styl návrhu aplikace, který byl definován v disertační práci Roye Fieldinga *Architectural Styles and the Design of Network-based Software Architectures* z roku 2000 [16]. Primárně je tento styl navržen pro distribuované hypermediální systémy[1] a klade na ně následující omezení:

1. **Klient-server** Systém musí mít oddělenou logiku uživateského rozhraní (klient) od logiky zpracování dat (server). Toto omezení zvyšuje přenositelnost a zároveň umožňuje nezávislý vývoj těchto komponent.
2. **Bezstavovost** Komunikace mezi klientem a serverem musí být bezstavová, tedy klient musí serveru doručit veškeré informace potřebné pro úspěšné zpracování požadavku. V případě

[1] Čímž je myšlen primárně World Wide Web.

potřeby stavu (například pro zpracování požadavku, který musí pro splnění projít několika stavy) je za jeho správu zodpovědný klient.

3. **Kešování** Veškeré odpovědi serveru odesílané klientovi musí být implicitně, nebo explicitně označeny jako kešovatelné, nebo nekešovatelné. Pokud je odpověď označena za kešovatelnou, potom je klientovi umožněno tuto odpověď lokálně uložit a znovu využít v situaci stejného požadavku.
4. **Uniformní rozhraní** Komunikace klienta a serveru musí využívat jednotné rozhraní při komunikaci, toto rozhraní musí být schopno: jednotné identifikace zdrojů, manipulace zdrojů pomocí jejich reprezentace, poskytovat sebepopisující zprávy a využívat hypermédia jako jádro aplikačního stavu.
5. **Vrstvený systém** Architektura aplikace musí umožňovat vrstvení komponent, které spolu následně komunikují zasíláním zpráv.
6. **Kód na vyžádání** Aplikace dovoluje klientovi rozšířit svoji funkcionalitu stažením a vykonáním kódu ve formě apletů nebo skriptů, čímž snižuje počet vlastností, které musí být v klientovi předem implementovány.

Architektonický styl REST aplikujeme vystavěním REST rozhraní, které bude založené na protokolu HTTP [17]. Pro implementaci omezení architektonického stylu REST se využívá sémantiky protokolu HTTP [18]:

- Každý zdroj v rámci rozhraní má asociovanou cestu URI přes kterou jsou vykonávány požadované akce.
- Požadovaná akce je realizována odpovídající HTTP metodou.
- Každý HTTP požadavek odpovídá stavovým kódem, který vyjadřuje jaký je výsledek zpracování požadavku.

3.3.4 Framework Gin

Framework Gin rozšiřuje služby balíčku `net.http` standardní knihovny jazyka Go a poskytuje konzistentní způsob popisu REST rozhraní. Zároveň přidává koncept middleware funkcí, které obalují funkce obsluhující požadavky na zdroje a tím umožňují definici jednotného chování aplikace ve specifických situacích (například zpracování chybových stavů, nebo reakce na stav aplikace) [19].

3.3.5 Vestavěná databáze Badger

Vestavěná databáze Badger, je implementace takzvané key-value databáze. Jejím účelem je ukládání dvojic: klíč, hodnota [20]. Vestavěná v tomto kontextu značí, že se nevyužívá jako samostatná služba, ale poskytuje knihovní funkce a běhové prostředí pro její správu je součástí přímo procesu, který ji využívá.

Klíče a hodnoty v databázi Badger jsou řezy bajtů. Klíče jsou uloženy v LSM stromě, čímž poskytuje rychlé vyhledávání klíčů a indexování pomocí prefixu. Zároveň databáze poskytuje plnohodnotné transakční zpracování s ACID vlastnostmi [21].

Kapitola 4

Návrh aplikace

V kapitole 3.1 jsme definovali případy užití a provedli jsme analýzu požadavků na výsledné řešení. V této kapitole se zaměříme na návrh řešení, které nám umožní docílit zadaných funkčních i nefunkčních požadavků a zároveň budeme implementovat všechny případy užití. Nejdříve definujeme rozdělení aplikace do komponent a následně jednotlivé komponenty do modulů, které budou zapouzdřovat hlavní části funkcionality. Popíšeme služby, které v systému vystupují a jakým způsobem spolu interagují.

4.1 Komponenty řešení

Z důvodů složitosti algoritmů použitých pro generování grafů byl zvolen asynchronní model zpracování požadavků na vygenerování grafů. Tento model umožňuje generování grafů, které není vázané na časovou omezenost zpracování HTTP požadavku. Zároveň tento model zajišťuje možnost vytvoření responsivní webové aplikace.

Výsledné řešení je proto implementováno ve dvou nezávislých modulech, kterými jsou:

- **Backend** Služba pro zpracování požadavků na generování grafů, vystavující rozhraní vybudované dle architektonického stylu REST, využívající HTTP protokol.
- **Frontend** Uživatelské rozhraní – klientská část aplikace, vytvářející uživatelské rozhraní pomocí technologií HTML a JavaScript.

Tyto moduly budou následně zodpovědné za zpracování požadavku uživatele. Hlavní část práce nicméně bude přenesena na samotný backend, protože ten bude zodpovědný za vykonání všech kroků nezbytných pro úspěšné vytvoření požadovaného grafu. Komponenta frontend je zodpovědná pouze za vizualizaci dat a jejich prezentaci uživateli.

4.2 Backend

Hlavní server zpracující požadavky uživatelů musí být schopen přijímat požadavky, vyhodnotit je a vrátit uživateli výsledky. Při návrhu webové aplikace, která vykonává tento druh úkolů máme možnost zvolit ze dvou způsobů přístupu ke zpracování:

- **Synchronní model** znamená, že požadavek se zpracovává okamžitě po zaslání požadavku a blokuje volajícího do okamžiku doručení výsledku. Příkladem může být zavolání funkce v programovacím jazyce, či požadavek na zobrazení webové stránky od serveru.

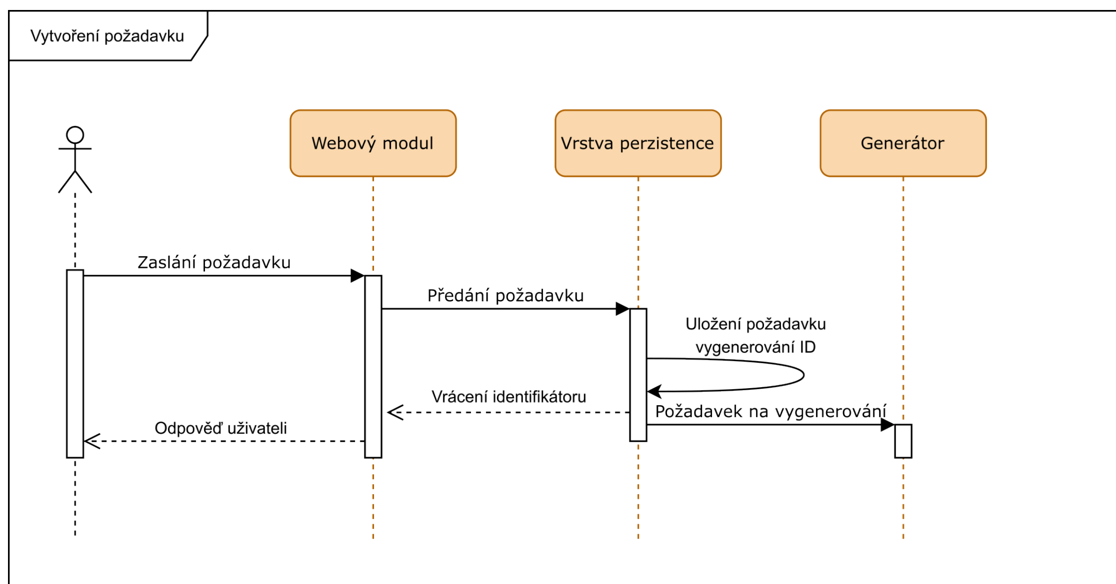
- **Asynchronní model** přijímá požadavek, nicméně neodpovídá výsledkem, místo něj zasílá identifikátor, který umožní volajícímu získat výsledky v okamžiku splnění požadavku.

Z kapitoly 2 víme, že algoritmy použité pro řešení těchto problémů budou polynomiální složitosti, navíc pracují s generátory náhodných čísel a tedy potřebná doba procesorového času pro vykonání požadavku může být různorodá a v některých situacích delší, než doba přípustná pro okamžitou odpověď. Proto při návrhu serverové části aplikace zvolíme asynchronní model.

4.2.1 Zpracování požadavku

Pro úspěšné zpracování požadavků musí požadavek projít několika moduly. Tyto moduly budou zajišťovat správu životního cyklu od vytvoření po úspěšné vyřešení. Hlavními konceptuálními moduly v naší aplikaci budou:

- **Webový server** zajišťuje komunikaci s vnějším světem, je zodpovědný za reprezentaci dat a předávání požadavků dalším komponentám.
- **Vrstva perzistence** ukládá požadavky a výsledky generování, zároveň je zodpovědná za životní cyklus objektů.
- **Generátor** vykonává samotné generování grafů v asynchronním modelu. Přebírá úkoly od vrstvy perzistence a po vyřešení je zasílá zpět.



Obrázek 4.1 Sekvenční diagram zpracování požadavku

Postup vytvoření požadavku je zachycen na obrázku 4.1, který zobrazuje sekvenční diagram popisující způsob komunikace mezi moduly. Po přijetí požadavku a zaslání identifikátoru zpět uživateli, prochází požadavek, nyní již jako serverový objekt, mezi moduly, než je označen za vyřízený.

4.2.1.1 Webový server

Webový server vytváří rozhraní mezi vnějším světem (komponentou frontend, případně přímo uživatelem) a zpracující logikou aplikace. Na jedné straně má otevřená HTTP spojení, která jsou

vstupní branou pro komunikaci, na druhé straně interaguje s vrstvou perzistence a dodává jí požadavky ke zpracování. Hlavními zodpovědnostmi této vrstvy jsou:

- **Validace vstupů**, tedy kontrola správnosti požadavků, ať již z pohledu gramatiky vstupu, tak i z pohledu integritních omezení.
- **Transformace vstupů**, tedy překlad vstupu do ze vstupního formátu do objektové reprezentace.
- **Transformace výstupů**, tedy překlad objektů v systému
- **Prezentace** tedy zajištění dodání výsledků dotazu zpět volajícímu.

4.2.1.2 Vrstva perzistence

Vrstva perzistence spojuje generátor a webový server a zajišťuje životní cyklus objektů. Zároveň je zodpovědná i za zajištění konzistence, integrity a trvanlivosti dat, pro ostatní moduly. Objekty vystupující v jejím kontextu jsou:

1. **Požadavky na generování** které reprezentují příkaz generátoru k vytvoření grafu, drží informaci o stavu, referenci na výsledek a informaci o životnosti. Dále tyto požadavky dělíme na:
   a. **Požadavek na jednotlivý graf** vytváří právě jeden graf zadaných vlastností.
   b. **Požadavek na dávku grafů** vytváří množinu požadavků na jednotlivý graf, které mají pevnou asociaci na něj.
2. **Vygenerovaný graf** drží strukturu popisující graf, která je při požadavku na stažení serializována webovým serverem.
3. **Uživatel** jako entita je vázán na konkrétní požadavek, pro umožnění vyhledávání a třídění požadavků v komponentě frontend.

4.2.1.3 Generátor

Modul generátor je výkonnou jednotkou, která zajišťuje aplikační logiku práce. Modul přebírá požadavky od vrstvy perzistence a následně zajišťuje výběr a spuštění vhodných procedur, které řeší zadaný problém generování. Po úspěšném zpracování předává výsledek zpět vrstvě perzistence pro uložení a případné předání zpět webovému serveru při obsluze požadavku.

4.3 Návrh REST rozhraní

Pro zajištění komunikace backendu a frontendu využijeme architektonický styl REST s jehož pomocí vystavíme veřejné rozhraní, jehož logiku bude implementovat backend z pozice serveru. Frontend bude v tomto kontextu plnit roli klienta.

Toto rozhraní bude realizováno pomocí protokolu HTTP, v němž bude mít každý spravovaný zdroj přiřazenou svoji URI přes kterou bude probíhat interakce se serverem.

4.3.1 Základní vlastnosti

REST rozhraní navrhujeme jako takzvané verzované API, což znamená, že součástí identifikace zdroje obsahuje informaci o verzi rozhraní [22]. V našem návrhu je tento přísutp realizován nastavením předpony cesty URI ke zdroji na `/api/v1/`, za níž je identifikátor prostředku. Tento přístup volíme proto, že umožňuje rozšiřování rozhraní při zachování zpětné kompatibility.

4.3.2 Zdroje využívané v aplikaci

Pro výměnu dat použijeme datový formát JSON [23]. Základními objekty

4.3.2.1 Graf

Graf definujeme jako JSON objekt s následujícími atributy:

- `id` je celé nezáporné číslo, které jednoznačně identifikuje požadavek v rámci celé aplikace. Jedná se o výstupní atribut, který je při vytváření ignorován a nahrazen vygenerovaným identifikátorem.
- `type` je řetězec popisující druh generovaného grafu, jeho hodnoty mohou být: `exact-degree` pro regulární grafy, `between-degree` pro grafy se stupni v rozmezí, `at-least-degree` pro grafy s minimálním stupněm, `average-degree` pro grafy s průměrným stupněm a `complete` pro úplné grafy. Jedná se o povinný atribut požadavku.
- `nodes` je celé číslo, které určuje počet vrcholů generovaného grafu, jde o povinný atribut požadavku.
- `node_degree` je celé číslo, které reprezentuje v případě regulárního grafu stupeň k. V případě požadavku na generování grafu se stupni v rozmezí a minimálním stupněm, reprezentuje minimální stupeň.
- `node_degree_max` je celé číslo využité pouze u požadavků na stupeň v rozmezí a reprezentuje maximální stupeň.
- `node_degree_average` je desetinné číslo, které reprezentuje průměrný stupeň vrcholu a je vyžadováno pouze pro typ grafu s průměrným stupněm.
- `connected` je pravdivostní hodnota vynucující souvislost, pokud je nastaveno na pravdu.
- `weighted` je pravdivostní hodnota vyžadující generování hranově ohodnoceného grafu.
- `weight_min` je celé číslo definující minimální hodnotu váhy hrany. Je použito pouze, pokud je `weighted` nastaveno na pravdu.
- `weight_max` je celé číslo určující maximální váhu, musí být větší nebo rovno `weighted_min`.
- `status` řetězec popisující současný stav požadavku. Má jednu z hodnot: `finished` značí, že požadavek byl splněn a může být stažen, `not-finished` značí, že požadavek je stále ve zpracování, `undefined` je základní hodnota, která uvádí nedefinovaný stav požadavku, který se využívá především pro základní popis grafu u požadavku.
- `seed` semínko pro generátor náhodných čísel,
- `deleted` řetězec reprezentující čas, kdy dojde k odstranění požadavku ze serveru.

4.3.2.2 Dávka

Dávka je reprezentována jako JSON objekt s atributy:

- `id` je celé číslo jednoznačně identifikující dávku v rámci systému.
- `number` je celé číslo, které určuje kolik grafů má být v rámci dávky vygenerováno.
- `base` obsahuje objekt definice grafu, který bude použit jako definice pro generování, pokud má nastavený atribut seed.

```json
{
    "connected": false,
    "deleted": "2023-05-06T18:01:03.41991583+02:00",
    "id": 1232185758,
    "node_degree": 3,
    "nodes": 12,
    "seed": 916889966689073954,
    "status": "not-finished",
    "type": "exact-degree",
    "weight_max": 20,
    "weight_min": 10,
    "weighted": true
}
```

Výpis kódu 2 Požadavek na generování grafu ve formátu JSON

- `deleted` je řetězec popisující čas smazání požadavku a všech závislých objektů.
- `graph_ids` seznam identifikátorů vytvořených grafů, každý z nich je následně dostupný přes koncový bod grafu.
- `status` je řetězec definující stav dávky, nabývá stejných hodnot jako status u objektu graf.

4.3.2.3 Pomocné objekty

V rámci rozhraní ještě definujeme čtyři další druhy objektů, které jsou využity pro speciální užití a jsou poměrně jednoduché:

- Seznam grafů v systému realizujeme JSON objektem, který obsahuje jeden atribut `graphs` asociující seznam celých čísel, která odpovídají identifikátorům grafů. Minimální objekt v tomto případě vypadá takto `{"graphs":[]}`.
- Seznam dávek je strutkurně stejný jako seznam grafů, tedy jedná se o JSON objekt s jediným atributem `batches`, který má asociovaný seznam identifikátorů, tedy minimální objekt vypadá `{"batches":[]}`.
- Objekt popisující chybu obsahuje dva atributy, `error` který odpovídá chybové zprávě a je povinným atributem a `cause`, který může obsahovat doplňkovou informaci k chybě a je nepovinný. Minimální chybový objekt vypadá takto `{"error":"message"}`.
- Limity aplikace definují maximální hodnoty přípustné pro dodané požadavky, obsahuje dva atributy a to: `max_nodes` odpovídající maximálnímu počtu vrcholů v požadavku na graf a `max_batch_size` odpovídající maximálnímu počtu grafů v požadavku na dávku. Minimální objekt potom vypadá následovně `{"max_nodes": 100, "max_batch_size": 50}`.

4.3.3 Koncové body

Pro každý definovaný zdroj nyní zavedeme odpovídající koncové body REST rozhraní, které je zpřístupní a umožní vykonávat definované akce. Pro všechny koncové body platí několik společných pravidel:

- Všechny chybové stavy (tedy HTTP status kódy větší než 400 a menší než 600) vrací v rámci opodvědi chybový objekt.
- Každý koncový bod může skončit s HTTP stavem `503 SERVICE_UNAVAILABLE`, který informuje volajícího, že systém je v současné době ve stavu údržby a neodpovídá na API požadavky.

4.3.3.1 Graf

Všechny požadavky, které pracují s objektem grafu, mají předponu cesty URI nastavenou na `/api/v1/graph`.

POST Při zaslání zprávy na tento koncový bod metodou POST, je očekávaným vstupem objekt popisující graf ve formátu JSON. Aplikace odpovídá buď:

- `201 Created` je odpověď úspěšného vytvoření nového požadavku na vytvoření grafu, tělo návratové hodnoty je JSON obsahující stejný popis jako zaslaný, doplněný o identifikátor požadavku, současný stav požadavku a přiřazený seed.
- `400 BAD_REQUEST` je odpověď, kterou server vrací v případě, že vstupní formát neodpovídá požadovanému formátu, nebo pokud je požadavek na graf nevalidní. Tělo odpovědi obsahuje chybovou zprávu.
- `500 INTERNAL_SERVER_ERROR` v případě, kdy není možné z důvodu překročení limitu počtu požadavků na server.

GET Vrací seznam grafů , které jsou přístupné volajícímu. V případě, že systém není v režimu údržby, je jedinou možnou odpovědí `200 OK` a tělem je objekt seznamu grafů.

4.3.3.1.1 Jednotlivé grafy Každému platnému požadavku na graf, který je evidován v systému, přiřazujeme cestu `/api/v1/graph/:graphId`, kde `:graphId` je identifikátorem grafu.

GET vrací objekt grafu v případě úspěchu. V případě neúspěchu vrací HTTP status `404 NOT_FOUND` reprezentující, že daný objekt již neexistuje.

DELETE požaduje vymazání grafu ze systému. Návratové stavy jsou následující:

- `204 NO_CONTENT` je odpověď serveru v případě úspěšného vykonání požadavku, tělo dopovědi je prázdné.
- `404 NOT_FOUND` odpovídá server v případě požadavku na smazání neexistujícího grafu, v těle dopovědi objekt reprezentující chybu.
- `405 METHOD_NOT_ALLOWED` je navráceno v případě, kdy požadavek nelze splnit, například z důvodu probíhajícího zpracování grafu.

4.3.3.1.2 Stažení grafu Vyzvednutí vygenerovaného grafu je zpřístupněno na koncovém bodu `/api/v1/graph/:graphId/download`. Tento koncový bod je specifický, protože neodpovídá JSON objektem, ale v těle odpovědi serveru je v případě úspěchu textový soubor popisující graf v požadovaném formátu.

Formát výstupu je určen query parametrem [18] `graphKind`, tedy cesta URI je nastavena na `/api/v1/graph/:graphId/download?graphKind=:kind`, kde `:graphId` odpovídá identifikátoru grafu a `:kind` je požadovaný druh reprezentace grafu. V současném stavu jsou přípustnými hodnotami: `matrix` pro reprezentaci ve formátu matice sousednosti a `dot` pro grafy ve formátu GraphViz DOT[1]. Koncový bod odpovídá následně:

- `200 OK` vrací, pokud je graf možné stáhnout, tělo odpovědi je textový soubor v očekávané reprezentaci.
- `400 INVALID_REQUEST` v případě, kdy není možné požadavek splnit, nebo nebyl rozpoznán požadovaný formát.
- `404 NOT_FOUND` v případě, kdy graf nebyl nalezen.

[1] `https://graphviz.org/doc/info/lang.html`

4.3.3.2 Dávka

Chování rozhraní dávky je ve většině situací totožné chování grafu. Předpona koncového bodu pro manipulaci s dávkami je nastavena na `/api/v1/batch`. Přímo tento koncový bod zpracovává vytváření dávek a získávání seznamu dávek stejným způsobem pomocí metod `POST` respektive `GET`. Pro získání konkrétní dávky dle identifikátoru přistupujeme ke koncovému bodu `/api/v1/batch/:batchId`, i zde je chování rozhraní stejné jako u požadavku na graf.

Na koncovém bodě `/api/v1/batch/:batchId/download` realizujeme požadavek na stažení dávky grafů, v tomto případě je místo textového souboru uživateli zaslán archiv ve formátu zip, ve kterém jsou uloženy textové soubory s vygenerovanými grafy. Stejně jako u požadavku na stažení grafu, můžeme specifikovat pomocí query parametru [18], v jakém formátu mají být grafy dodány, na základě tohoto parametru budou do archivu vloženy, buď grafy ve formátu matice sousednosti, nebo ve formátu DOT. Chování v případě chyby je totožné s chováním koncového bodu pro stažení grafu.

4.3.3.3 Limity

Koncovým bodem pro získání limitů aplikace je `/api/v1/limits`, který odpovídá pouze na HTTP metodu `GET` a jako odpověď vrací pouze stav `200 OK` a objekt popisující lmity aplikace. Tento koncový bod nepřijímá žádné argumenty a nemůže selhat, s výjimkou situace údržby systému.

Kapitola 5
Implementace

V této části se zaměříme na popis řešení, použité knihovny a uvedeme konkrétní příklady kódu, který řeší jednotlivé problémy implementace. Celou kapitolu rozdělíme na dvě části:

- popis implementace serveru, který vystavuje REST rozhraní a je implementovaný v programovacím jazyce Go.
- popis implementace klientské části, která je implementovaná v frameworku Angular.

Celý projekt je implementovaný v jednom repozitáři a je dostupný na serveru GitHub[1], verzovaný nástrojem git [24]. Hlavní částí implementace je serverová část naprogramovaná v jazyce Go. Té odpovídají složky `cmd` a `pkg`, dále pak soubory `go.sum` a `go.mod`.

Uživatelské rozhraní je ve složce `ui`, která obsahuje plně definovaný projekt frameworku Angular včetně všech servisních souborů.

Poslední částí je složka `test`, kde je vytvořený samostatný projekt v jazyce Python, určený pro integrační a end-to-end testování produktu. Tento projekt využívá nástroj Poetry, který zajišťuje správu závislostí a spuštění testovacích scénářů. Detailně se této části práce bude věnovat následující kapitola 6.

5.1 Server

V kapitole 4 jsme konceptuálně popsali doménu zodpovědnosti serverové části řešení, identifikovali jsme hlavní moduly a přiřadili jim základní úkoly. Nyní tento koncept asociujeme s reálným kódem.

Nejdříve popíšeme základní datové typy definující rozhraní pro výměnu dat ať již mezi balíčky v rámci serveru, tak i pro komunikaci s vnějším světem. Poté se budeme věnovat postupu generování grafů, tedy popíšeme si balíček, který je za něj zodpovědný. Definujeme jeho rozhraní a utility, které používá pro výkonné zpracování požadavků a také se podíváme na problémy, které vedly k úpravám návrhu.

Dále popíšeme implementaci webového rozhraní aplikace v architektonickém návrhu REST, její balíček, a webový framework Gin, s jehož pomocí je toto rozhraní vystaveno. Také zmíníme další vlastnosti HTTP rozhraní a podpůrné funkce pro zajištění správné funkčnosti.

Jako poslední rozebereme službu perzistence dat, která je synchronizačním bodem mezi uživatelem a generátorem a zároveň zajišťuje uložení výsledků a jejich získání. Současně popíšeme knihovny zajišťující právě uložení a vyhledávání dat a nároky, které kladou jak na cílový systém, tak na komunikaci s dalšími částmi. Zdrojový kód generátoru je implementován ve dvou balíčcích vyšší úrovně, a to:

[1] `https://github.com/soch-fit/GraphGenerator.git`

- Balíček `cmd`, který obsahuje popis produkovaného spustitelného souboru. Jeho obsahem je složka generátor, která obsahuje pouze jeden zdrojový soubor `main.go`. Ten implementuje logiku spuštění aplikace, inicializuje a propojuje jednotlivé služby a řídí životní cyklus aplikace.
- Balíček `pkg` obsahuje veškerou vnitřní logiku aplikace a je vnitřně členěn do balíčků, které implementují potřebné části aplikační logiky a interagují mezi sebou. Zároveň je také vybaven jednotkovými testy, které ověřují funkčnost řešení.

5.1.1 Základní datové typy

Datové typy definované v balíčku `pkg/api` jsou využívány v celé aplikaci a definují, jaká data jsou v aplikaci vyměňována. Zároveň jsou tyto typy využívány i pro serializaci výsledných dat ve formátu JSON uživateli v rámci rozhraní REST a ukládány ve vrstvě perzistence. Základní typy pro výměnu dat jsou:

- `GraphRequest` odpovídá entitě požadavku na graf definovaném v rámci REST rozhraní.
- `BatchRequest` odpovídá požadavku na dávku grafů.
- `LimitResponse` reprezentuje nastavené limity běžící aplikace.
- `ErrorResponse` definuje objekt obsahující chybovou odpověď serveru.

Všechny tyto typy mají definováno, jakým způsobem mají být serializovány do formátu JSON, který je následně zasílán jako odpověď na HTTP požadavek a zároveň umožňují i deserializaci při přijetí požadavku. V ukázce 3 vidíme realizaci vazby atributu struktury na atribut JSON objektu pomocí tagu `json`.

```
type GraphRequest struct {
    Type              GraphType     `json:"type"`
    Weighted          bool          `json:"weighted"`
    Nodes             int           `json:"nodes"`
    NodeDegree        int           `json:"node_degree"`
    Status            RequestStatus `json:"status"`
    Seed              *int64        `json:"seed,omitempty"`
    Timeout           time.Time     `json:"deleted,omitempty"`
    NodeDegreeMax     int           `json:"node_degree_max,omitempty"`
    NodeDegreeAverage float32       `json:"node_degree_average,omitempty"`
    WeightMin         int           `json:"weight_min"`
    WeightMax         int           `json:"weight_max"`
    Connected         bool          `json:"connected"`
    ID                uint32        `json:"id"`
    Owner             *string       `json:"-"`
    BatchId           *uint32       `json:"-"`
}
```

Výpis kódu 3 Struktura požadavku na graf

5.1.1.1 Serializace výsledků

V rámci řešení poskytujeme dva formáty pro vyzvednutí vygenerovaného grafu, matici sousednosti a formát v jazyce DOT. Pro zajištění překladu vnitřní reprezentace grafu (kterou popisujeme v 5.1.2) využíváme rozhraní `GraphTranslator`.

V ukázce 4 je toto rozhraní definováno. Jeho hlavní metodou je `Convert`, která přebírá vygenerovaný graf a provede transformaci grafu do formátu, který umožní rychlý překlad do výstupní podoby. Například v případě maticového formátu je vnitřní stav překladače matice čísel.

```
type GraphTranslator interface {
    Convert(g generator.Graph) bool
    Serialize(writer io.Writer) (io.Writer, error)
    Bytes() []byte
    ContentType() string
    Extension() string
    Kind() string
}
```

Výpis kódu 4 Rozhraní překladače vnitřní reprezentace grafu do výstupního formátu

Metoda `Serialize` provede samotnou transformaci do výstupního proudu, tedy převede svoji vlastní reprezentaci konvertovaného grafu do binárního formátu a zapíše jej do předaného proudu. Metoda `Bytes` provede serializaci a vrátí binární reprezentaci výsledku. Metody `ContentType`, `Extension` a `Kind` jsou následně použity pro identifikaci výstupního formátu a vytvoření souborů.

5.1.2 Generování grafů

Generátor grafů existuje v balíčku `pkg/generator` a je rozdělen do několika balíčků, dle domény zodpovědnosti. Základním datovým typem, se kterým balíček operuje, je rozhraní `Graph`. Toto rozhraní (zobrazené v ukázce 5) popisuje metody, kterými se s výsledkem dá interagovat.

```
// WeightedEdge associates nodes into one edge
type WeightedEdge struct {
        Left, Right int
}

// Graph Basic interface for representation of a graph
type Graph interface {
        Nodes() []string
        Edges() []map[int]bool
        Weights() map[WeightedEdge]int
        Properties() GraphProperties
}
```

Výpis kódu 5 Rozhraní typu graf.

Rozhraní volíme záměrně, namísto implementace přímo základního datového typu, protože umožňuje minimalizovat paměťovou náročnost dle reálné potřeby. Například pro požadavek na vygenerování jednoduchého grafu není potřeba držet informaci o váhách hran.

V rámci generátoru reprezentuje vrchol celé číslo ležící na intervalu $\langle 0; n)$. Vzhledem k této reprezentaci jsou následně vztaženy ostatní vlastnosti základních typů.

V ukázce 5 si můžeme všimnout, že pro reprezentaci hran, je zvolen typ `[]map[int]bool`, tedy pole map vázajících celé číslo na Booleovskou hodnotu. Tato reprezentace (tedy mapa vázající klíč na Booleovskou hodnotu) je idiomatická implementace množiny v jazyce go [25].

Velikost vnějšího pole odpovídá počtu vrcholů vygenerovaného grafu. Obsažená mapa následně drží informaci o všech vrcholech spojených hranou. Tato reprezentace znamená, že každá hrana je uložena na dvou místech, totiž asociace je v tomto případě uložena pro každou hranu, tedy hrana mezi vrcholy 1 a 5 znamená, že v mapě uložené na indexu 1 je uložená dvojice `{1: true}`, respektive v mapě na indexu 5 je uložena dvojice `{5: true}`.

Tento přístup je implementací zvolen záměrně, protože tato reprezentace grafu umožňuje rychlé zjištění celého sousedství vrcholu (které je díky vlastnostem mapy konstantní), za cenu zvýšení paměťové náročnosti. Vzhledem k postupu zpracování grafu, jak při generování, tak i při serializaci, je tento kompromis vhodný.

Opačný přístup je zvolený pro reprezentaci hranových vah, které jsou uloženy v jedné mapě vázající dvojici `WeightedEdge` na celé číslo. `WeightedEdge` je typem realizující dvojici vrcholů, přičemž pro zajištění jednoznačnosti je podmínkou typu, že atributem `Left` je vždy vrchol s nižším indexem. V tomto případě je každá hrana realizována v paměti pouze jednou. Tohoto přístupu můžeme využít především proto, že máme nezávisle uloženou reprezentaci existence hrany a váhy při zpracování zjišťujeme pouze v okamžiku, kdy víme, že mezi vrcholy hrana existuje.

5.1.2.1 Zdroj náhody

Pro zajištění náhodných výběrů, které jsou podmínkou všech generujících algoritmů, využíváme pseudonáhodný generátor implementovaný v `math.rand` standardní knihovny Go. Důvodem této volby je především rychlost řešení, všechny generátory potřebují provést rámcově $\mathcal{O}(nk^2)$ náhodných výběrů, tedy využití například kryptograficky bezpečných, by vedlo k výraznému zpomalení generujících algoritmů.

Základní pseudonáhodný zdroj dat je implementován jako vláknově bezpečný z důvodu potřeby držet stav generátoru [26], což je jeho největší slabinou pro použití v případě generátorů, které běží paralelně, protože zajištění vláknové bezpečnosti vede k vytvoření úzkých hrdel, při kterých dochází ke zpožďování a blokování generujících vláken režií využitou pro synchronizaci.

```go
import "math/rand"
src := rand.NewSource(2358)
rng := rand.New(src)
```

Výpis kódu 6 Vytvoření nového zdroje náhodných dat

Z toho důvodu každý generátor používá vlastní zdroj náhodných dat, který je přiřazený výhradně jednomu běhu generujícího algoritmu, a tedy generátory nesdílí žádný společný zdroj který by vytvářel potřebu synchronizace. Další výhodou je možnost nastavit každému generátoru vlastní `seed`, čímž můžeme zajistit předvídatelné a znovu opakovatelné generování grafů.

5.1.2.2 Množina bodů

V kapitole Teorie grafů 2.2.1 jsme identifikovali Steger-Wormaldův algoritmus jako vhodný pro řešení problému generování regulárních grafů a jeho konceptuální implementace je poměrně přímočará. Jedinou výjimkou je navržení množiny bodů, jejíž vlastnosti jsou klíčové pro správný průběh algoritmu.

5.1.2.2.1 Naivní implementace Zcela naivní implementací množiny bodů by mohlo být pole počítadel, tedy pro každý vrchol vytvoříme jedno počítadlo, které v první iteraci nastavíme na k stupeň. V každé iteraci potom vybíráme náhodně dvojici vrcholů a pokud tvoří vhodnou hranu, snížíme jejich počítadla.

Tento naivní přístup způsobuje selhání algoritmu ve velmi vysoké míře, důvodem je podmínka zmíněná v práci Stegera a Wormalda [4], která vyžaduje, aby pravděpodobnost výběru vrcholu byla proporční počtu volných bodů, což tento přístup nemá.

5.1.2.2.2 Pole bodů Druhou možností je implementace množinou bodů, tedy do vhodného úložiště vložit pro každý vrchol k-krát jeho identifikátor a vybírat náhodně body v rámci této množiny. Tento přístup splňuje podmínky algoritmu na proporčnost výběru bodu, ale výrazně zvyšuje paměťovou náročnost řešení. Potřebná paměť se oproti naivnímu řešení zvýší z $\mathcal{O}(n)$ na $\mathcal{O}(kn)$, kde k je stupeň regulárního grafu a n je počet jeho vrcholů. Pro vysoké stupně $k \in \mathcal{O}(n)$ to fakticky mění paměťovou náročnost na $\mathcal{O}(n \cdot k) = \mathcal{O}(n^2)$. Navíc, v závislosti na zvoleném

uložení množiny, může tento přístup výrazně zvýšit výpočetní složitost řešení potřebou výrazné správy paměti.

5.1.2.2.3 Kumulativní počítadlo Je implementačně podobná naivnímu přístupu, ale místo uložení počítadla pro každý vrchol ukládáme součet volných stupňů všech předchozích vrcholů. Náhodný výběr bodu potom funguje na principu hledání prvku pozici $l \in \langle 0; p)$, kde p je počet zbývajících bodů, přičemž l je zvoleno náhodně.

```
func (p *pointSet) GetPoint(elem int) int {
	left, right := 0, len(p.sums)
	mid := (right - left) / 2

	for mid > 0 && !(elem >= p.sums[mid-1][0] && elem <= p.sums[mid][0]) {
		if elem >= p.sums[mid][0] {
			left = mid
		} else {
			right = mid
		}
		mid = (right-left)/2 + left
	}
	return p.sums[mid][1]
}
```

Výpis kódu 7 Vyhledávání bodu v kumulativním počítadle

Na ukázce 7 je vidět tento přístup. Při hledání využijeme faktu, že kumulativní pole má hodnoty vzestupně uspořádané, tedy při hledání bodu dle jeho pořadí využíváme algoritmus vyhledávání půlením intervalu.

Tedy nalezení bodu má výpočetní složitost $\mathcal{O}(\log(n))$ pro n vrcholů, nicméně smazání prvku má složitost lineární vzhledem k počtu vrcholů generovaného grafu, protože pro smazání bodu musíme snížit všechna počítadla od 0 až po l. Což je stále výrazně lepší než implementace množinou bodů, protože paměťová náročnost je rovna $\mathcal{O}(n)$.

5.1.2.2.4 Intervalový strom Z výše uvedených příkladů vidíme, že pro zajištění všech požadovaných vlastností potřebujeme rozumnější strukturu než pole. Pro tuto implementaci určíme, že pro každý vrchol určíme interval jeho bodů. Tedy pro každý vrchol budou jeho body reprezentovány jako $\langle vk; (v+1)k)$ pro vrchol v a k počet bodů. Tímto sestavíme celkový interval všech bodů $\langle 0; nk)$, kde n je celkový počet vrcholů.

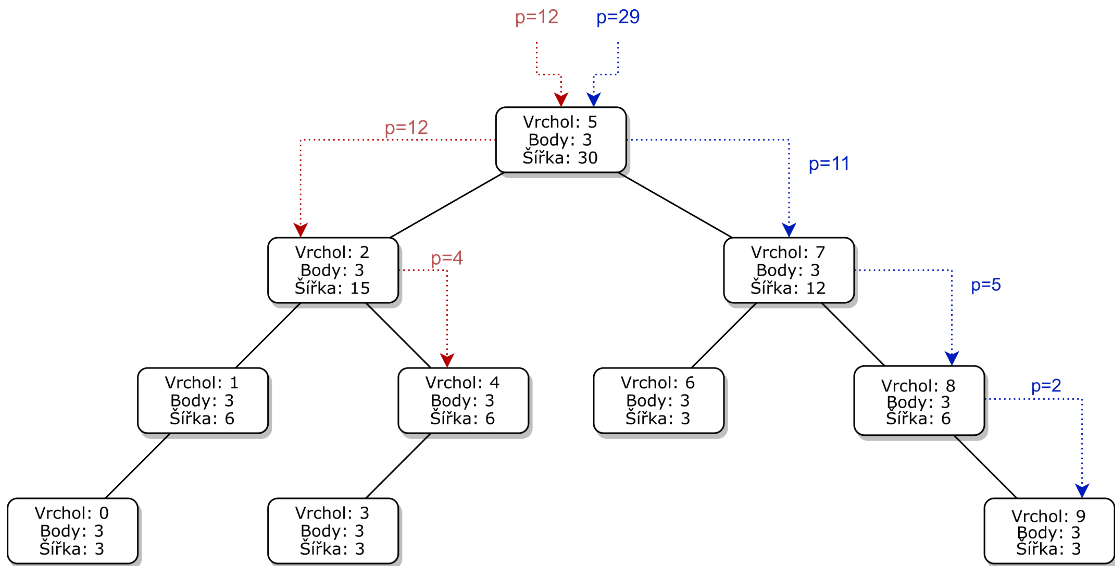
Výběr bodu potom bude realizován náhodným výběrem čísla na tomto intervalu a a pro tuto reprezentaci zvolíme intervalový strom [27] implementovaný jako AVL strom.

Vnitřní struktura intervalového stromu je zachycena na obrázku 5.1, který zobrazuje vnitřní stav stromu pro 10 vrcholů se třemi body pro vrchol. Pro každý vnitřní uzel tohoto stromu držíme informaci o vrcholu, jehož body reprezentuje, počtu volných bodů a relativním rozsahu, který se svými potomky pokrývá.

Asociace pořadí s vrcholem následně probíhá následujícím způsobem: Předpokládejme, že p je pořadí bodu pro nějž chceme zjistit odpovídající vrchol, a předpokládáme, že spadá do intervalu, který intervalový strom popisuje. Začneme s kořenovým uzlem intervalového stromu jako uzlem u a dokud nenajdeme hledaný vrchol, provedeme následující akce.

1. Pokud je p menší, než šíře intervalu uzlu nalevo, opakujeme postup s levým potomkem jako u a nezměněnou hodnotou p.

2. Pokud je p větší než rozsah nalevo, ale menší, než rozsah nalevo zvýšený o počet bodů u, hledaný vrchol je asociovaný s u a vrátíme jej jako výsledek.



Obrázek 5.1 Intervalový strom pro deset vrcholů se třemi body

3. Pokud je p větší než interval nalevo zvýšený o počet bodů současného uzlu, snížíme p o tuto hodnotu a spustíme hledání znovu s u nastaveným na pravý podstrom.

Postup algoritmu můžeme vidět na obrázku 5.1 červeně pro bod na 12. pořadí a modře pro 29. Vidíme, že pro každý posun vpravo provedeme snížení hledaného pořadí, protože pro každý uzel stromu provádíme hledání relativně. Při cestě vlevo p neměníme, protože p leží na daném intervalu. Implementaci hledání v jazyce Go je zobrazeno v ukázce 8.

Odstranění bodu následně provedeme snížením počtu bodů v asociovaném uzlu a propagací změny intervalů ve všech rodičovských uzlech. V případě odstranění vrcholu (počet bodů se sníží na 0) odstraníme uzel z intervalového stromu a provedeme standardní vyvážení AVL stromu pomocí rotací.

Takto implementovaná množina bodů má stejnou složitost vyhledání a mazání $\mathcal{O}(\log n)$, zároveň má lineární paměťovou náročnost vzhledem k počtu vrcholů, jedná se tedy o výrazné zlepšení výkonnosti oproti předchozím implementacím.

5.1.2.3 Regulární grafy

Generátor regulárních grafů implementuje Steger-Wormaldův algoritmus definovaný v kapitole 2.2.1.1 a využívá množinu bodů implementovanou pomocí intervalového pole.

Zastavení algoritmu a detekce nesplnitelnosti je implementována kontrolou zacyklení. Pro každou iteraci, ve které se algoritmu nepodaří vytvořit novou hranu, zvýšíme počítadlo neúspěchů o jedna. Pokud je počet neúspěchů vyšší než dvojnásobek počtu zbývajících hran pro doplnění, označíme množinu bodů za nesplnitelnou a pokusíme se o opravu přepnutím hran. Pokud oprava selže, spustíme algoritmus znovu se stejně nastaveným generátorem náhodných čísel, tedy další průchod algoritmem bude vybírat jiná náhodná čísla.

V případě požadavku na souvislost upravíme graf algoritmem 2.5, který pro jednotlivé komponenty souvislosti provede přepnutí pro jejich propojení.

Jako optimalizaci proti častému selhávání algoritmu u požadavků s vyššími stupni $k > \frac{n}{2}$ využijeme regularity doplňku regulárního grafu 2.35, tedy v případě požadavku na regulární graf stupně vyššího než polovina počtu vrcholů, spustíme algoritmus s $\overline{k} = n - k + 1$ a výsledný graf následně hranově invertujeme.

```
func (t *Tree) GetPoint(n int) (int, error) {
	iter := t.root
	if iter.subTree <= n {
		return -1, ErrPointNTooBig
	}
	for iter != nil {
		leftBound := 0
		if iter.left != nil {
			leftBound = iter.left.subTree
		}
		midBound := leftBound + iter.width
		if n < leftBound {
			iter = iter.left
			continue
		}
		if n < midBound {
			return iter.val, nil
		}
		n -= midBound
		iter = iter.right
	}
	return -1, ErrNoPoints
}
```

Výpis kódu 8 Vyhledání bodu v intervalovém stromě

5.1.2.4 Grafy se stupni v rozmezí

Grafy se stupni v rozmezí generujeme upraveným Steger-Wormaldovým algoritmem, který jsme definovali v 2.2.2, a který využívá dvě množiny bodů. Algoritmus pracuje ve dvou fázích. Nejdříve zajistí vygenerování grafu se stupněm nejméne k_m vyprázdněním první množiny bodů.

Následně volíme náhodný počet hran které doplníme do grafu a pokračujeme vybíráním bodů z druhé množiny, která zajišťuje nepřekročení maximálního stupně. Pokud detekujeme zacyklení algoritmu nemožností doplnit hrany, vracíme vygenerovaný výsledek jako úspěch, protože graf splňuje zadané podmínky.

U požadavku na souvislost volíme v tomto algoritmu vygenerování kostry grafu upraveným Borůvkovým algoritmem 2.4. Před spuštěním generování vygenerujeme náhodný strom s maximálním stupněm k_m a množinu bodů sestavíme upravenou o stav tohoto stromu (tedy od každého vrcholu odečteme jeho současný stupeň, který bude nejméně jedna). Algoritmus následně pokračuje nezměněn.

5.1.2.5 Ostatní druhy grafů

Ostatní druhy grafů jsou na vygenerování poměrně jednoduché, u grafů s minimálním stupněm spustíme algoritmus generování grafu se stupni v rozmezí s předaným $k_m = k$ a $k_n = n - 1$.

Grafy s průměrným stupněm generujeme náhodnou volbou počtu hran, které přidáme do grafu, a která je rovna $\frac{an}{2}$. Pokud je požadavek na souvislost, vygenerujeme nejdříve náhodný strom, který pouze doplníme náhodnými hranami.

Pro úplné grafy pouze vrátíme naplněnou reprezentaci jednoduchého grafu, kde existuje hrana pro každou dvojici vrcholů. U tohoto algoritmu nejsou žádné další podmínky nebo okolnosti potřebné ke zvážení.

5.1.2.6 Souvislé grafy

Pro zajištění souvislosti generovaných grafů jsme v kapitole 2.3 identifikovali dva přístupy:

- Přepínáním hran v již vygenerovaném grafu.

- Vygenerováním náhodného stromu pomocí Borůvkova algoritmu, který následně použijeme jako základ pro další generování.

Spojení přepnutím hran používáme pouze u regulárních grafů a jedná se o poslední fázi algoritmu, tedy jeho použití nevytváří potřebu upravovat samotný generátor.

Upravený Borůvkův algoritmus spouštíme před začátkem samotného generování, je tedy nezbytné správně přenastavit generátory aby pracovaly nad již připraveným stromem. Pro grafy s průměrným stupňěm není potřeba žádná úprava, v případě požadavku na souvislý graf pouze nastavíme počet již existujících hran, kterých je $n-1$ a algoritmus pokračuje běžným způsobem.

V případě generování grafů s minimálním stupněm a se stupni v rozmezí potřebujeme upravit způsob sestavení množiny bodů. Proto proces sestavení množiny bodů upravíme o možnost nastavení počtu bodů pro každý vrchol zvlášť. Následně algoritmus probíhá stejným způsobem.

5.1.2.7 Služba generující grafy

Poslední částí nezbytnou pro generování grafů je běhové rozhraní, které zvládne efektivně zpracovávat předané požadavky. V naší implementaci je reprezentováno službou, která přijímá požadavky a zasílá zpět výsledné náhodné grafy.

Běhové prostředí této služby se skládá z několika (uživatelem zadaných) paralelně běžících výkonných gorutin. Tyto gorutiny přebírají požadavky z kanálu vstupních požadavků, provedou výběr algoritmu a spustí jej s přednastaveným generátorem náhodných čísel dle předaného semínka[2].

Po skončení algoritmu předají výsledek zpět do výstupního kanálu s asociací na identifikátor požadavku a pokračují výběrem dalšího požadavku. Kvůli potřebě údržby databáze, mají generátory přístupný další kanál, který se používá pro zasílání řídících informací.

```
type Service interface {
    StartService() error
    Pause() error
    Resume() error
    PushRequest(req api.GraphRequest) error
    GetRetriever() chan *api.GraphResult
    Stop() error
    FreeBand() int
}
```

Výpis kódu 9 Rozhraní služby generující grafy

Rozhraní této služby si můžeme prohlédnout v ukázce 9.

5.1.3 HTTP rozhraní

Další částí serveru je jeho HTTP rozhraní, které poskytuje dvě základní služby, REST rozhraní pro generování grafů a zároveň poskytování uživatelského rozhraní. Aplikace po spuštění otevírá HTTP port, který je nastaven na hodnotu 8080 v základní konfiguraci.

V celé aplikaci využíváme REST framework Gin, který řídí zpracování vstupů a předání odpovídajícím kontrolním funkcím zodpovědným za vyřešení požadavku. Tato část aplikace je implementována v balíčku `pkg/routers/`.

[2]Semínko, neboli *seed* je iniciální hodnota generátoru, která nastavuje postup generátoru, cíl

5.1.3.1 HTTP služba

Správa HTTP spojení v aplikaci spravuje služba `HTTPService`. Tato služba je zodpovědná za otevření portu, nastavení všech koncových bodů v rámci protokolu HTTP a správu prostředků spojených s obsluhou požadavků.

```go
type HTTPService interface {
    Start() error
    Stop(ctx context.Context) error
}

type GinHttpService struct {
    srv *http.Server
    err chan error
}
```

Výpis kódu 10 Rozhraní HTTP služby a její implementace

V ukázce 10 vidíme definici rozhraní `HTTPService`, které definuje dvě metody pro interakci:

- `Start` inicializuje server pro a spouští jej, pokud při startu dojde k chybě, vrací chybu v návratové hodnotě.
- `Stop` instruuje server k zastavení a čeká dokončení požadavku, předaný parametr `context` je zaslán přímo implementaci HTTP serveru a nastavuje dobu za kterou se musí zastavení serveru provést. Pokud se server v kontextem definovaném časovém intervalu nezastaví, je čekání přerušeno a vrací se chybový stav.

Dále v ukázce vidíme službu `GinHttpService`, která implementuje rozhraní `HTTPService` a pro svoji funkci drží instanci `http.Server`, což je implementace HTTP serveru ze standardní knihovny jazyka Go [28]. Dále drží kanál, který je určen pro získání návratové hodnoty metody `ListenAndServe`, jejíž běh zajišťuje obsluhu HTTP komunikace. `GinHttpService` po zavolání metody start provádí následující akce:

- Vytvoří instanci `http.Server` a nakonfiguruje základní vlastnosti (port, rozhraní).
- Provede konfiguraci REST rozhraní pomocí frameworku Gin.
- Nastaví obslužné funkce pro zpracování chyb a logování.
- Spustí gorutinu, ve které běží hlavní obsluha HTTP serveru.
- V případě selhání spuštění serveru vrací zpět chybový stav.

Spuštění služby je blokující operace, která trvá 2 sekundy. Důvodem je zajištění chybového stavu v případě selhání spuštění HTTP serveru (například kvůli obsazenému portu). Metoda Start po spuštění gorutiny čeká po dobu dvou sekund a pokud je během této doby doručena zpráva v kanálu `err`, vyhodnocuje spuštění jako neúspěšné a vrací doručenou chybu.

5.1.3.2 Rozhraní REST

REST rozhraní definujeme v balíčku `pkg/routers/api/v1` a hlavní funkcí tohoto balíčku je `SetupREST` (viz ukázka 11). Po zavolání této funkce je v rámci frameworku Gin zaregistrována nová skupina routerů, se společným prefixem cesty v URI `/api/v1/`.

V ukázce 11 můžeme vidět nastavení asociace obslužných funkcí s HTTP metodami a URI cestou identifikující zdroj. Zároveň nastavujeme Gin middleware voláním metody `r.Use`.

```
func SetupREST(engine *gin.Engine, svc requests.RequestService) *gin.RouterGroup {
    if engine == nil {
        engine = gin.Default()
    }
    r := engine.Group("/api/v1")
    r.Use(middleware.SetUpCookieMiddleware())
    r.Use(middleware.SetUpRequestService(svc))
    r.Use(middleware.Error())
    r.GET("/limits", handleLimitsRequest)
    r.GET("graph", handleGraphList)
    r.GET("graph/:graphId", handleGraphGet)
    r.DELETE("graph/:graphId", handleGraphDelete)
    r.POST("graph", handleGraphCreate)
    r.GET("graph/:graphId/download", handleGraphDownload)
    r.GET("batch", handleBatchList)
    r.POST("batch", handleBatchCreate)
    r.GET("batch/:batchId", handleBatchGet)
    r.DELETE("batch/:batchId", handleBatchDelete)
    r.GET("batch/:batchId/download", handleBatchDownload)

    return r
}
```

Výpis kódu 11 Vytvoření Gin router pro REST rozhraní

Obslužné funkce pro zpracování požadavku následně provedou konrolu a validaci vstupních parametrů a těla požadavku (v případě metody POST), pokud je požadavek validní, předají ho vrstvě perzistence k vyřízení – tedy získání dat v případě požadavku GET, uložení a zpracování nového požadavku na generování v případě metody POST, nebo požadavek na smazání v případě metody DELETE.

Pokud při zpracování nastane libovolná chyba, nastavují obslužné funkce příznak chyby Gin kontextu a okamžitě ukončují zpracování požadavku. Signalizace chybového stavu v rámci rozhraní REST je následně zpracováno `Error` middlewarem.

Propojení s vrstvou perzistence je realizováno middleware funkcí `SetupRequestService`, která nastavuje Gin kontextu zpracovávaného požadavku paramter `requestsService`, který obsahuje ukazatel na službu `RequestService` (popsána dále v 5.1.4). Tento parametr je následně využíván obslužnými funkcemi pro komunikaci právě s vrstvou perzistence dat. Zároveň kontroluje, zda není `RequestService` v režimu údržby databáze. Pokud v tomto režimu je, potom middleware funkce `SetupRequestService` ukončuje zpracování požadavku rovnou s chybovým hlášením o probíhající údržbě.

5.1.3.3 Dodávání uživatelského rozhraní

V případě, kdy je konfigurací nastaveno serveru dodávat uživatelské rozhraní, potom `GinHttpService` nastavuje zároveň obslužné funkce i pro URI `/`, se kterou budeme asociovat jediný HTML soubor uživatelského rozhraní `index.html`. Uživatelské rozhraní během navigace uživatele nicméně mění URI dle aktuální obrazovky, na které se uživatel nachází (detailně popíšeme v 5.2.2).

Pro správnou funkci uživatelského rozhraní při dodávání serverem tedy potřebujeme zajistit, že v případě požadavku na neexistující cestu, vrátíme jako výsledek právě `index.html` a necháme vyhodnocení platnosti URI na uživatelském rozhraní.

V ukázce 12 vidíme obslužnou funkci vyhodnocující požadavky na neznámé URI. Pokud je požadavek na soubor v kořenové URI, potom zkontrolujeme, zda tento požadavek neobsahuje koncovku typu souboru (např. `.html`). Pokud tomu tak je, předpokládáme, že se jedná o identifikátor obrazovky uživatelského rozhraní a odpovídáme navrácením `index.hml` a vyhodnocení

```
func handleNoRoute(ctx *gin.Context) {
    origUri := ctx.Request.RequestURI
    dir, file := filepath.Split(origUri)
    ext := filepath.Ext(file)
    log.Debugf("Got request to %s %s", file, ext)
    if dir == "/" && (file == "" || ext == "") {
        ctx.File(fmt.Sprintf("%s/index.html", *configuration.Default().UiLocation))
        return
    }
}
```

Výpis kódu 12 Obslužná funkce pro předávání uživatelského rozhraní

validnosti předané cesty necháme vyřešit uživatelské rozhraní.

Pokud URI obsahuje koncovku, nebo se nejedná o požadavek na kořen, potom neděláme nic, pokračujeme standardním zpracováním požadavku na neexistující zdroj.

Ostatní soubory nezbytné pro fungování uživatelského rozhraní jsou potom zpřístupněny na URI odpovídající jejich jménu. Tedy například soubor `index.html` je dodávaný jako odpověď požadavku na URI `/index.html`.

5.1.3.4 Podpůrné funkce

Aplikace v současném stavu nepodporuje autentizaci, a tedy asociace požadavku s uživatelem musí být řešena jinak. Pro zajištění funkcionality koncových bodů poskytujících seznamy objektů (`/api/v1/graph` a `/api/v1/batch`), využíváme techologii HTTP Cookies [29].

Dalšími podpůrnými funkcemi jsou `Logger` a `Recover`. Funkce `Logger` je využívána pro zaznamenávání příchozích požadavků do aplikačního logu. Na základě nastavení „upovídanosti“ zapisuje detaily o příchozích spojeních a výsledku zpracování požadavku. Funkce `Recover` je zodpovědná za zpracování situací, kdy dojde ke kritickému selhání zpracování požadavku propagací nezachycené paniky.

5.1.4 Perzistence dat

Správa požadavků a ukládání výsledků je řízeno službou perzistence dat. Tato služba v řešení vystupuje jako synchronizační bod zodpovědný za celý životní cyklus požadavku od vytvoření, přes vygenerování, dodání výsledku na vyžádání a nakonec jeho smazání. Rozhraní této služby je popsáno v ukázce 13 a existují dvě její implementace:

- **Perzistence v paměti** využívá pouze operační paměť pro ukládání a správu požadavků. Má velmi vysokou propustnost, protože využívá pouze standardní nástroje vyloučení přístupu a nepracuje s diskovým úložištěm.
- **Perzistence na disku** využívá plnohodnotnou key:value databázi pro realizaci ukládání dat, díky tomu poskytuje všechny vlastnosti kladené na databázové úložiště, tedy atomicitu operací, integritu dat, konzistenci a trvanlivost.

5.1.4.1 Paměťová perzistence

Typ `InMemoryService` implementuje službu perzistence dat v operační paměti. Tato služba je primárně určena pouze pro vývoj a testování a nelze ji zapnout konfigurací systému, ale pouze manuálně úpravou kódu. Ačkoliv je velice rychlá, přináší pro běh problémy, které neumožňují její nasazení v produkčním prostředí.

```
type RequestService interface {
    StoreNewRequest(request api.GraphRequest) (api.GraphRequest, error)
    StoreNewBatch(request api.BatchRequest) (api.BatchRequest, error)
    StoreGraph(graph *api.GraphResult) error
    ListRequests(sessionId string) ([]uint32, error)
    ListBatches(sessionId string) ([]uint32, error)
    GetGraphRequest(graphId uint32) (api.GraphRequest, error)
    GetBatch(batchId uint32) (api.BatchRequest, error)
    GetGraph(graphId uint32) (api.GraphResult, error)
    GetBatchResult(batchId uint32) ([]api.GraphResult, error)
    DeleteGraph(graphId uint32) error
    DeleteBatch(batchId uint32) error
    Start() error
    Stop() error
    CheckMaintenance() bool
```

Výpis kódu 13 Rozrhaní služby perzistence

Základem její implementace je několik vláknově bezpečných map [30], které asociují identifikátory s ukládanými daty.

5.1.4.2 Databáze Badger

Pro produkční nasazení využíváme vestavěnou databázi Badger, kterou jsme popsali v 3.3.5. Tato databáze definuje klíče i hodnoty jako řezy bajtů `[]byte`. Nad klíči následně staví stromovou strukturu, ve které je schopna efektivně vyhledávat na základě prefixu.

5.1.4.2.1 Indexování a vyhledávání Pro efektivní vyhledávání požadavků definujeme klíče jako řetězce, a pro každý druh objektu vytvoříme nezávislý prefix. Důvodem použití různých prefixů pro různé objekty je především zjednodušení načítání dat a vyhledávání závislých objektů (například u vazby požadavku na graf s vygenerovaným grafem, či vazby požadavku na dávku se seznamem z ní vytvořených grafů).

```
type KeyProvider interface {
    GetKey() []byte
    GetPrefix() []byte
}
```

Výpis kódu 14 Rozhraní typu klíče pro databázi Badger

V ukázce 14 vidíme rozhraní poskytující klíče. Metoda `GetKey` vrací bajtovou reprezentaci klíče přímo pro jeho realizaci v databázi. Metoda `GetPrefix` následně vrací jednoznačný prefix, který nekoliduje s ostatními klíči a jednoznačně asociuje klíče stejného typu. Klíči v rámci databáze jsou:

- `DbGraphRequest` obsahuje identifikátor grafu a v rámci databáze asociuje požadavky na graf. Prefixem je řetězec `request-`.
- `DbBatch` obsahuje identifikátor dávky a podobně jako u požadavku na graf, asociuje objekty požadavků na dávky. Jako prefix je zvolen řetězec `batch-`.
- `DbGraphResult` asociuje v databázi vygenerovaný graf s identifikátorem požadavku. Prefix je nastaven na `result-`.

- `DbBatchGraph` je indexovací klíč, který vytváří vazbu mezi dávkou a grafem a umožňuje rychlé vyhledání všech grafů asociovaných s dávkou. Klíč obsahuje dvě hodnoty, identifikátor dávky a identifikátor grafu. Jeho prefix je odvozený z identifikátoru dávky a vypadá následovně: `batch-graph-%d-`, kde místo `%d` je identifikátor dávky jako dekadický řetězec.

5.1.4.2.2 Serializace hodnot Stejně jako u klíčů, potřebujeme pro uložení hodnot v databázi Badger provést jejich překlad do binární podoby řezu bajtů. Zároveň potřebujeme zajistit, že tato binární podoba bude obsahovat všechna data a její načtení zpět . K tomu využíváme balíček standardní knihovny jazyka Go `encoding/gob` [31].

Knihovna `gob` je určena pro výměnu datových typů jazyka Go mezi procesy, například přes síťová rozhraní v případě protokolu RPC. Její výhodou je zajištění plné serializace a deserializace všech informací, které jsou vázané na objekt do hloubky a to včetně případných metadat.

```go
func marshall[K any](val K) []byte {
    var buff bytes.Buffer
    translator := gob.NewEncoder(&buff)
    translator.Encode(val)
    return buff.Bytes()
}

func unMarshall[K any](val []byte) K {
    var result K
    reader := bytes.NewReader(val)
    translator := gob.NewDecoder(reader)
    translator.Decode(&result)

    return result
}
```

Výpis kódu 15 Pomocné funkce pro překlad objektů do/z binární podoby

V ukázce 15 vidíme realizaci funkcí, které zajišťují překlad ukládaných objektů do binární podoby

5.1.4.2.3 Údržba databáze Databáze Badger dobře zvládá za běhu zmenšování a udržování klíčů. Hodnoty nicméně ukládá do `log` souborů, do kterých za běhu pouze zapisuje nová data, ale nemění ani neodstraňuje obsah. Tedy při dlouhodobém běhu může dojít k zahlcení disku těmito již neplatnými soubory. Abychom této situaci předešli, spouští `PersistentService` podpůrnou gorutinu, která periodicky spouští údržbu databáze zavřením a novým otevřením databáze. Postup spuštění údržby je následující:

1. Nastaví příznak údržby databáze, čímž zabrání REST rozhraní ve vytváření nových požadavků.
2. Zavolá metodu `Pause` generátoru grafů a počká na dokončení zpracovávaných grafů.
3. Uzavře databázi a znovu ji otevře, čímž spustí proces údržby log souborů.
4. Zavolá metodu `Resume` generátoru grafů, který obnoví svoji činost.
5. Zruší příznak údržby databáze a obnoví přístup REST rozhraní k databázi.
6. V případě selhání otevírání databáze vynutí ukončení procesu, kvůli výskytu neodstranitelné chyby.

5.1.5 Konfigurace programu

Konfiguraci programu řeší balíček `pkg/configuration`, který ji zpřístupňuje ostatním balíčkům pomocí funkce `Default()`. Tato funkce vrací strukturu `Provider`, která obsahuje všechny konfigurační možnosti.

Balíček využívá návrhový vzor jedináček[3] a funkce `Default` po prvním zavolání provede zajištění konfigurace z definovaných proměnných prostředí a možností příkazové řádky. Následující volání funkce `Default` vrací neměnný objekt který ve svých atributech drží základní limity a možnosti aplikace. Detailně jsou tyto možnosti popsány v kapitole 7.1.

5.1.6 Definice programu

Spustitelný program `generator` je definovaný v balíčku `cmd/generator`, který obsahuje pouze jeden zdrojový soubor `main.go`. Ten je zodpovědný za spuštění programu, jeho nastavení a propojení jednotlivých komponent a spuštění služeb. Zároveň definuje základní signály, na které reaguje pro korektní ukončení, kterými jsou `SIGINT` a `SIGTERM`. V případě doručení těchto signálů procesu, začne funkce `main` ukončovat spuštěné služby. Nejdříve ukončí službu `HTTPServer`, následně generátor grafů a jako poslední ukončuje službu perzistence dat.

5.2 Uživatelské rozhraní

Uživatelské rozhraní řešení je naprogramováno v programovacím jazyce TypeScript, který je výchozím pro framework Angular. Pro definici vzhledu byla využita knihovna Bootstrap [32], která poskytuje základní CSS styly a podpůrné knihovny v JavaScriptu, poskytující základní rámec pro tvoření stránky.

Uživatelské rozhraní aplikace je implementováno ve složce `ui/` repozitáře a obsahuje kompletní definici projektu pro NodeJS, který je využit pro správu závislostí, spuštění vývojového prostředí a sestavení výsledné komponenty.

Ve složce `ui/src` se nachází definice aplikace ve frameworku Angular a samotná implementace uživatelského rozhraní je následně ve složce `ui/src/app`, kde se nacházejí všechny zdrojové soubory.

5.2.1 Služby a typy pro komunikaci s REST rozhraním

Abychom mohli popsat fungování komponent zobrazujících objekty v systému, potřebujeme nejdříve definovat, jakým způsobem probíhá komunikace se serverem. Pro zajištění této komunikace má Angular aplikace definovány rozhraní a služby, které zabezpečují funkcionalitu komunikace a překladu HTTP požadavků do vnitřní reprezentace v TypeScript.

5.2.1.1 Rozhraní

Pro popis objektů, které poskytuje rozhraní REST, definujeme TypeScript rozhraní [33], specificky se jedná o `GraphRequest`, popisující požadavek na generování grafu, `BatchRequest` popisující požadavek na generování dávky grafů a `Limits` popisující objekt limitů serveru.

5.2.1.2 Služby

Získávání a zpracování dat v rámci uživatelského rozhraní provádí služby `GraphRequestService` a `BatchRequestService`, které jsou injektovány jednotlivým komponentám běhovým prostředím frameworku Angular.

[3] Anglicky nazývaný singleton.

Tyto služby jsou zodpovědné za korektní komunikaci s REST rozhraním, překlad výsledků do vnitřní reprezentace a zpracování chyb. Komponenty v případě potřeby komunikují s těmito službami pomocí jejich veřejných metod.

Zvláštním druhem služby je potom `MessageService`, která zajišťuje zobrazování dynamických zpráv uživateli. Funguje jako synchronizační bod pro zasílání zpráv jak z komponent.

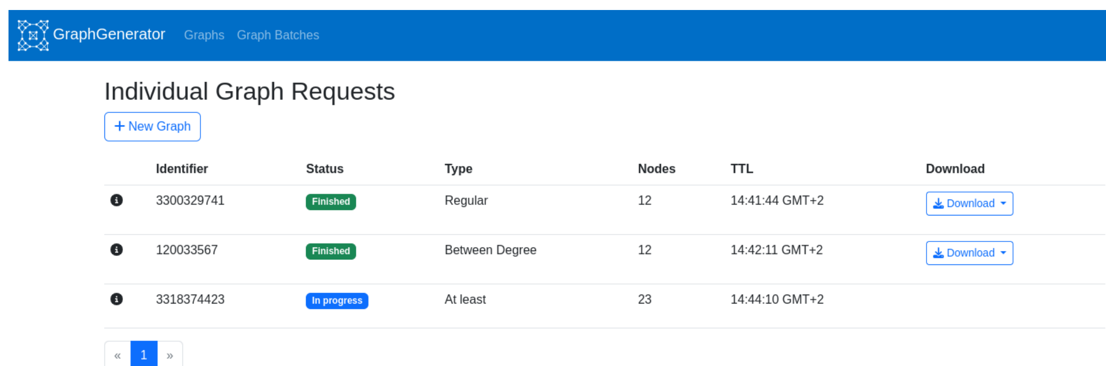
Poslední službou, kterou definujeme je `LimitsService`, která je zodpovědná, za dodání aktuálních limitů REST rozhraní, tedy maximálního počtu vrcholů grafu a maximálního počtu grafů v požadavku na dávku. Tato služba je injektována do komponent formulářů a je využívána validátory při jejich vyplňování uživatelem.

5.2.2 Základní obrazovky

Aplikace definuje dvě základní obrazovky, pro požadavky na graf a pro požadavky na dávky grafů. Navigace k těmto obrazovkám je možná pomocí tlačítek v záhlaví stránky nebo statické cesty v požadavku na webové rozhraní z prohlížeče.

5.2.2.1 Graf

Obrazovka grafu je zodpovědná za prezentaci uživatelem vytvořených požadavků na graf. Její základní rozhraní je zobrazeno na obrázku 5.2.



Obrázek 5.2 Základní obrazovka grafu

Jejím hlavním prvkem je tabulka, která obsahuje všechny platné požadavky, jejich základní vlastnosti a u již vygenerovaných grafů také tlačítko umožňující stažení výsledku v požadovaném formátu.

Pro zjištění detailních informací o grafu je v každém řádku tlačítko reprezentované ikonou informace, které při kliknutí rozbalí kartu popisující detaily, což můžeme vidět na obrázku 5.3.

Obrazovka je implementována v Angular komponentě `RandomGraph`, která při zobrazení nejdříve požádá službu `GraphService` o získání seznamu grafů od API. Následně pro každý graf získá od stejné služby jeho současný stav a uloží jej do interního pole. Toto pole grafů je následně zobrazeno v tabulce.

Tabulka zobrazující požadavky na graf je stránkovaná, aby nedocházelo k zahlcení uživatelského rozhraní vykreslováním vysokého počtu objektů. Zobrazení karty detailů grafu je zpřístupněno po kliknutí na tlačítko informace u příslušného grafu.

Rozhraní zobrazuje, kvůli kompaktnosti, vždy pouze jednu kartu s detaily, tedy pokud uživatel klikne na zobrazení informací v situaci, kdy je zobrazena jiná, dojde ke skrytí původní karty před

	Identifier	Status	Type	Nodes	TTL	Download
🛈	3300329741	Finished	Regular	12	14:41:44 GMT+2	Download
🛈	120033567	Finished	Between Degree	12	14:42:11 GMT+2	Download
🛈	3318374423	Finished	At least	23	14:44:10 GMT+2	Download

ID: 120033567

```
Status:     Finished                     Type:        Between Degree
# Nodes:    12                           Min Deg:     1
Deleted:    2023-05-08 14:42:11 GMT+2    Max Deg:     4
Seed:       8575866565751637000          Weighted:    ⊗
Connected:  ✓
```

Obrázek 5.3 Zobrazení detailů požadavku na graf

zobrazením nové.

Obrazovka také obsahuje tlačítko pro zobrazení formuláře pro vytvoření nového požadavku. Po kliknutí na toto tlačítko dojde k rozbalení karty, která obsahuje komponentu formuláře (tu popíšeme dále).

5.2.2.2 Dávka

Zobrazení dávek grafů je funkčností i navigací velice podobné obrazovce grafů, jednotlivé požadavky jsou taktéž zobrazeny ve stránkované tabulce, která zobrazuje základní vlastnosti a zpřístupňuje možnosti stažení u již vyřízených požadavků a zobrazení detailů. Základní obrazovku si můžeme prohlédnout na obrázku 5.4.

GraphGenerator Graphs Graph Batches

Graph Batches

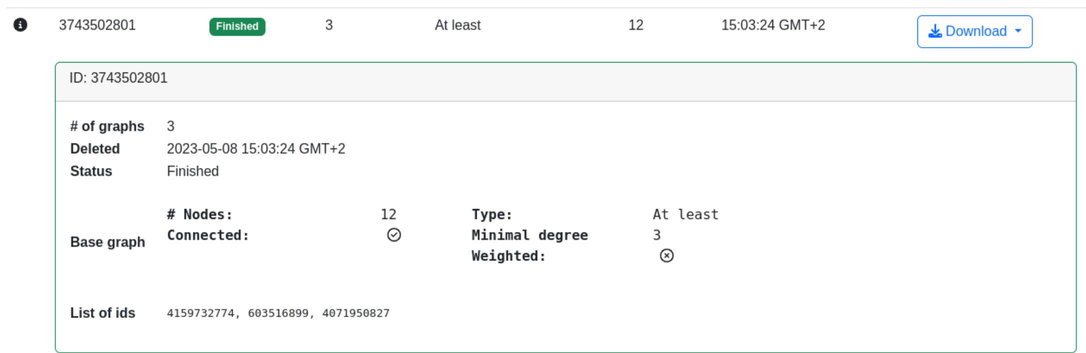
+ Batch

	Identifier	Status	Number	Type	Nodes	TTL	Download
🛈	3977914216	Finished	21	At least	10	15:00:42 GMT+2	Download
🛈	1118721799	Finished	10	Complete	14	15:01:21 GMT+2	Download
🛈	655691409	Finished	4	Between Degree	8	15:01:43 GMT+2	Download
🛈	3743502801	In progress	3	At least	12	15:03:24 GMT+2	

« 1 »

Obrázek 5.4 Základní obrazovka dávek

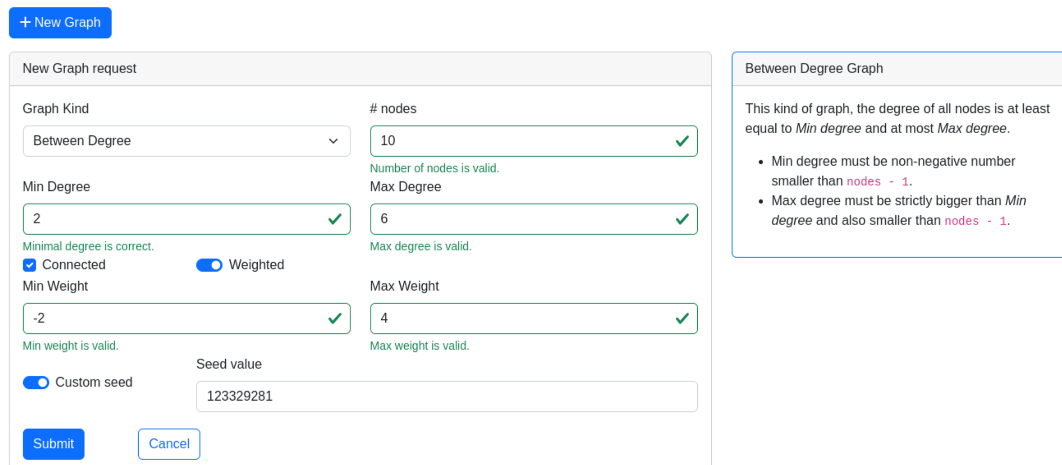
Celá obrazovka je implementována v Angular komponentě `RandomBatch`, a její vnitřní fungování je totožné s komponentou `RandomGraph`, tedy udržuje seznam všech dávek, které jsou uživateli dostupné. V případě obnovování nebo získávání dat z REST rozhraní požaduje data od služby `BatchService`.



Obrázek 5.5 Zobrazení detailů pro dávku

5.2.3 Formuláře

Pro vytvoření nových požadavků definujeme dvě Angular komponenty, které realizují dynamické formuláře frameworku Angular, a to `GraphRequest` a `BatchRequest`. Protože oba druhy požadavků, jak na generování grafu, tak na generování dávky, obsahují stejné atributy se stejnými integritními omezeními (vazba stupňů a počtu vrcholů, limity váhy apod.), vyčlenili jsme tuto stejnou funkcionalitu do sdílené komponenty `GraphRequestFormTemplate`, která ale pouze poskytuje sdílenou funkcionalitu, která je pro obě komponenty stejná.



Obrázek 5.6 Formulář požadavku na graf

Vykreslení formulářů si můžeme prohlédnout na obrázcích 5.6 a 5.7, kde vidíme jednotlivé vstupní elementy v případě vyplnění platnými hodnotami, kdy je tlačítko `Submit` označené za povolené a po kliknutí na něj, je formulář odeslán.

Odeslání formuláře v případě obou komponent se skládá ze dvou kroků, které nastávají v sekvenci. Nejdříve je z dynamického formuláře vyzvednut objekt odpovídajícího rozhraní (tedy `BatchRequest`, resp. `GraphRequest`), ten je následně předán službě pro vykonání POST požadavku na REST rozhraní, a v případě úspěchu, tedy požadavek skončí se stavovým kódem `201`, je vrácený objekt komponentou formuláře zaslán nadřazené komponentě a formulář se resetuje do výchozího stavu. Po doručení nového objektu nadřazené komponentě, je formulář skryt

Graph Batches

+ Batch

New Batch request

graphs

2

Number of graphs is valid.

Graph Kind

Complete

nodes

5

Number of nodes is valid.

Connected Weighted

Submit Cancel

Complete Graph

In **Complete Graph** every node is connected (e.g. there is an edge) with every other node in the graph.

Obrázek 5.7 Formulář požadavku na vytvoření dávky

a objekt je zařazen do pole pro vykreslení.

Individual Graph Requests

+ New Graph

New Graph request

Graph Kind

Between Degree

nodes

12.3

Number of nodes must be whole number.

Min Degree

4

Max degree must be strictly higher than min

Max Degree

4

Max degree must be strictly higher than min

Connected Weighted

Min Weight

12

Min weight is valid.

Max Weight

11

Maximal weight must be at least equal to the weight min

Custom seed

Submit Cancel

Between Degree Graph

This kind of graph, the degree of all nodes is at least equal to *Min degree* and at most *Max degree*.

- Min degree must be non-negative number smaller than `nodes - 1`.
- Max degree must be strictly bigger than *Min degree* and also smaller than `nodes - 1`.

Obrázek 5.8 Neplatně vyplněný formulář

V případě, že stav formuláře není validní, je tlačítko `Submit` označeno za zakázané a formulář není možné odeslat. Zároveň je u prvku formuláře, který obsahuje neplatnou hodnotu zobrazena chybová zpráva a vstupní element je označen červenou barvou, to můžeme vidět na obrázku 5.8.

Uživatel může kdykoliv během vyplňování formuláře stisknout tlačítko `Cancel`, to navrátí formulář do výchozího stavu a instruuje nadřazenou komponentu, aby byl skryt. V případě, že uživatel stiskne tlačítko zobrazující formulář (`+ Graph` nebo `+ Batch`), dochází pouze ke skrytí formuláře, ale jeho stav zůstává nezměněn, pokud nedojde k resetování komponenty, například změnou obrazovky, nebo obnovením stránky.

Kapitola 6

Testování aplikace

6.1 Jednotkové testování

Jednotkové testování je nejnižší úrovní testů které provádíme, typicky jsou jednotkové testy součástí přímo zdrojového kódu. Standardní knihovna jazyka Go obsahuje balíček pro psaní jednotkových testů `testing` [34], pro jejichž spouštění má podporu přímo překladač jazyka Go.

V rámci balíčku `pkg` jsou všechny balíčky vybaveny základní sadou jednotkových testů, které pokrývají základní chování funkcí, především pak chybové stavy.

Při testování byl největší důraz kladen na otestování generujících algoritmů a jejich korektnost. Pro každý generující algoritmus jsou definovány testy pokrývající následující:

- Neplatné vstupy algoritmu, tedy kombinace vstupů, které porušují pravidla zvoleného grafu (například záporné hodnoty).
- Základní funkčnost algoritmu, tedy splnění požadovaných vlastností vygenerovaného grafu.
- Testování splnění atributu souvislosti.
- Testování stability algoritmu pro stejná a různá semínka generátoru náhodných čísel.

Ostatní balíčky aplikace obsahují základní jednotkové testy ověřující funkcionalitu a korektní ošetření chybových stavů. Důvodem omezeného testování ostatních balíčků je především vysoká provázanost služeb, které testování komplikují, a proto se této části funkcionality více věnují testy integrační.

6.1.1 Testování výkonnosti algoritmů

Pro zajištění výkonu algoritmů obsahuje sada testů i připravené testy pro takzvaný benchmarking, tedy testy měřící výpočetní, či paměťovou náročnost řešení. Jejich primárním účelem je ověření předpokládaných vlastností použitých algoritmů, zjištění úzkých hrdel při výkonu (tedy míst, kde dochází ke ztrátám výkonnosti) a v neposlední řadě ověření změn, zda nesnižují výkon řešení.

Pro každý generující algoritmus jsme definovali základní výkonostní testy, které měří jeho chování pro očekávané případy použití, a testy pro porovnání rychlosti algoritmů mezi sebou.

```
$ poetry install
$ poetry run pytest
```

Výpis kódu 16 Spuštění integračních testů

6.2 Integrační testy

Integrační testy v tomto scénáři testují celou aplikaci po jejím sestavení, a očekávají, že aplikace plně běží a dodává uživatelské rozhraní. Tyto testy jsou napsány v jazyce Python a jsou řízeny testovacím frameworkem Pytest [35].

Základním modulem testovací komponenty je složka `suite/api`, která obsahuje definice integračních testů a podpůrných funkcí. Pro spuštění testů je potřeba provést následující příkazy.

První příkaz v ukázce 16 nejdříve vytvoří virtuální prostředí pro jazyk Python a instaluje závislosti testovacího projektu. Druhý příkaz spustí testy ve frameworku Pytest. Předpokladem úspěšného běhu testů je spuštění aplikace, která poslouchá na lokálním portu `8080`.

Pro celé rozhraní jsme definovali sadu testů, které pokrývají chybové stavy, korektní reakce na neplatné požadavky dle specifikace API a reakci API na neplatné požadavky na graf. Dále testy ověřují úspěšné generování grafů, tedy proces vytvoření požadavku, ověření dokončení, získání výsledku a ověření vlastností výstupu.

6.3 Manuální testy

Posledním druhem testování, který jsme vykonali pro ověření aplikace bylo manuální otestování výsledku.

6.3.1 Test běhu za reverzní proxy

V tomto testovacím případě jsme ověřovali, jak bude aplikace fungovat při nasazení za reverzní proxy. Zároveň jsme provedli tento test ve dvou variantách, reverzní proxy s HTTPS a reverzní proxy bez HTTPS. Testovaný program byl nasazen pomocí kontejnerizačního nástroje Docker a jako reverzní proxy byl zvolen server NGINX[1].

Cílem testu bylo ověřit, že aplikace dokáže správně fungovat v této konfiguraci a poskytovat veškeré služby. Současně tento test ověřoval schopnost uživatelského rozhraní správně komunikovat s REST rozhraním. Výsledkem testování bylo určení vhodné konfigurace reverzní proxy, které zajistilo korektní chování prohlížečů při zobrazování uživatelského rozhraní.

6.3.2 Test akutalizace systému

Dalším testem byla schopnost systému obnovit svojí funkci po aktualizaci programu. Test byl proveden v rámci produkční konfigurace odstraněním kontejneru běžící aplikace a vytvořením nového s využitím stejného úložiště databáze. Ověřovali jsme zvláště, zda platná data zůstala po restartování dostupná a nedošlo v jejich případě ke změně chování aplikace.

6.3.3 Zahlcení uživatelského rozhraní

Tímto testem jsme ověřovali, jak uživatelské rozhraní reaguje na případ vysokého počtu požadavků. Pro tento test jsme vytvořili 1 000 požadavků v rámci uživatelského rozhraní a ověřovali

[1]Použití těchto nástrojů bude popsáno v kapitole 7.

jeho výkonnost. Aplikace dokázala zobrazit všechny požadavky bez ztráty výkonu při navigaci. Jedinou pozorovanou anomálií bylo zvýšení paměťových prostředků využívaných uživatelským rozhraním, což není překvapivé, vzhledem k tomu, že každý požadavek je po dobu existence držen zobrazující komponentou.

6.3.4 Zahlcení serveru

Tímto testem jsme ověřovali stabilitu řešení při vysokém počtu požadavků, který překračuje možnosti zpracování. Test probíhal sekvencí paralelně provedených požadavků na generování dávek grafů maximální velikosti. Aplikace během tohoto testu korektně reagovala a začala odmítat nové požadavky. Během testu aplikace se zvýšil objem prostředků využívaných aplikací, ovšem nikdy nepřekročil stanovené limity konfigurací.

6.3.5 Test dlouhodobého běhu

Posledním manuálním testem, který jsme provedli, byl dlouhodobý běh aplikace v rámci platformy CloudFIT[2]. Zde byl nasazen virtuální stroj, na kterém aplikace běžela nepřetržitě po dobu dvou týdnů. Zároveň byly v aplikaci náhodně vytvářeny požadavky, hlavním cílem byla kontrola, zda aplikace při dlouhodobém běhu nezačne vykazovat ztrátu výkonnosti. Současně jsme ověřovali schopnost aplikace odstraňovat neplatné objekty z databáze údržbou. Po dvoutýdenním běhu aplikace využívala 186 MB operační paměti a velikost úložiště databáze byla 28 KB po provedené údržbě.

[2]`https://help.fit.cvut.cz/cloud-fit/index.html`

Kapitola 7

Nasazení řešení

V této kapitole si popíšeme jakým způsobem sestavíme výsledné řešení do spustitelného balíčku, jak provedeme vydání a nasazení do produkčního prostředí a doporučíme některá omezení. Aplikace byla vyvíjena a testována na OS Linux, pro který jsou připraveny i aplikační obrazy obsahující řešení.

7.1 Možnosti spuštění aplikace

Hlavním produktem práce je jeden spustitelný program `generator` a složka s vygenerovanými artefakty pro webové rozhraní. Spustitelný program `generator` má následně možnosti konfigurace pro běh, kterými může uživatel změnit chování programu.

Pro všechny konfigurační možnosti aplikace platí, že je lze nastavit jak přepínačem z příkazové řádky, tak i proměnnou prostředí. Pro většinu hodnot existují základní hodnoty, které řadíme do tří kategorií: produkční, vývojové a testovací.

Produkční hodnoty jsou nastaveny tak, aby poskytovaly vhodné výchozí nastavení pro běh v produkčním prostředí a jsou použity při základním spuštění bez jakékoliv modifikace.

Vývojové hodnoty se použijí, pokud je při spuštění předán přepínač `-devel` a nastavují aplikaci s hodnotami vhodnými pro vývoj a ruční testování, tedy s výrazně vyššími limity na požadavky a výrazně nižší dobou existence požadavku.

Testovací hodnoty se zapínají přepínačem `-test` a jsou určeny pouze pro využití integračními testy a tato konfigurace by neměla být používána mimo ně. Hlavními možnostmi pro běh aplikace jsou:

- **Maximální počet vrcholů**
  - Jedná se o celé kladné číslo, které nastavuje maximální povolený počet vrcholů v požadavku na vygenerování grafu.
  - Proměnná prostředí se jmenuje `GENERATOR_MAX_NODES`.
  - Z příkazové řádky se nastavuje možností `maxNodes`.
  - Výchozí hodnoty jsou: 100 v produkci, 1 000 ve vývojovém prostředí.
- **Maximální počet grafů v dávce**
  - Je celé číslo nastavující maximální přípustný počet grafů v jedné dávce.
  - Definující proměnná prostředí `GENERATOR_MAX_BATCH_SIZE`.
  - Možnost příkazové řádky `maxBatchSize`.

  - Výchozí hodnoty jsou: 50 v produkci, 1 000 ve vývojovém prostředí.

- **Počet pracovních gorutin**
  - Je celé kladné číslo, které nastavuje počet gorutin použitých pro generování grafů.
  - Proměnná `GENERATOR_WORKERS`.
  - Option `workers`.
  - Základní hodnota 4, vývojová hodnota 20.

- **Interval údržby databáze**
  - Nastavuje v jakém časovém rozmezí se má spouštět údržba databáze. Formátem je standardní definice časového intervalu v jazyce Go, tedy, číslo následované jednotkou. Povolené jednotky jsou v tomto přídě: `h` pro hodiny, `m` pro minuty a `s` pro sekundy.
  - Proměnná `GENERATOR_MAINTENANCE_INTERVAL`.
  - Přepínač `maintenanceInterval`.
  - Základní hodnota `24h`, vývojová hodnota `5m`.

- **Platnost požadavku**
  - Časový interval (viz výše) definující dobu platnosti požadavku, po jejím uplynutí je objekt vymazán z databáze.
  - Proměnná `GENERATOR_TTL`.
  - Přepínač `ttl`.
  - Základní hodnota `1h`, vývojová hodnota `10m`.

- **Hodina spuštění údržby**
  - Je celé číslo z intervalu $\langle 0; 23 \rangle$, které určuje základní hodinu, ve které se bude spouštět údržba databáze. Pokud je interval údržby kratší než jeden den, je následující okamžik údržby vypočítán vzhledem k této hodině.
  - Proměnná `GENERATOR_BASE_HOUR`.
  - Přepínač `baseHour`.
  - Základní hodnota 2, vývojová je určena dle času spuštění aplikace.

- **Kořenová složka databáze**
  - Nastavuje z jaké složky má být načtena databáze, pokud neexistuje, je vytvořena a naplněna servisními soubory databáze.
  - Proměnná `GENERATOR_DB_ROOT`.
  - Přepínač `dbRoot`.
  - Základní hodnota: `/var/badger/db`, vývojová hodnota: `./tmp/database`.

7.2 Vývojové sestavení

Pro lokální spuštění aplikace máme dvě možnosti:

- **Oddělený server a klient** je metoda spuštění vhodná pro vývoj uživatelského rozhraní, protože umožňuje nezávisle modifikovat komponenty.

- **Vše v jednom** je spuštění aplikace v jednom sezení, tedy server v tomto případě plní roli jak REST rozhraní, tak i webového, který dodává uživatelské rozhraní.

7.2.1 Oddělený běh

Pro nezávislé spuštění obou komponent potřebujeme spustit pouze dva příkazy, které nezávisle na sobě spustí jak server s REST rozhraním, tak i server dodávající uživatelské rozhraní.

```
$ go run ./cmd/generator/main.go -devel
```

Výpis kódu 17 Spuštění REST rozhraní pro vývoj

Ukázka 17 popisuje spuštění serveru bez překladu do spustitelného souboru.

```
$ cd ui/
$ npm start
```

Výpis kódu 18 Spuštění webového serveru, který poskytuje uživatelské rozhraní

V ukázce 18 vidíme spuštění vývojového serveru, který poskytuje uživatelské rozhraní na adrese `http://localhost:4200`. Zároveň monitoruje změny zdrojových souborů uživatelského rozhraní a v případě změny provede na místě překlad.

7.2.2 Vše v jednom

```
$ go build ./cmd/...
$ npm run build
```

Výpis kódu 19 Překlad zdrojových kódů

Pro lokální spuštění aplikace musíme nejdříve provést překlad zdrojových kódů. V ukázce 19 popisujeme dva příkazy, které provedou překlad všech zdrojových soborů do očekávaných artefaktů. První příkaz spouští překlad zdrojových souborů v jazyce Go. Výsledkem tohoto příkazu je jediný spustitelný soubor `generator`, který bude uložen v kořenové složce se zdrojovými soubory.

Druhý příkaz ukázky 19 provede překlad uživatelského rozhraní, jehož výstup následně umístí do složky `./ui/dist/graphGenerator`. Tato složka bude naplněna jedním souborem `index.html`, několika soubory se skripty v jazyce JavaScript a kaskádovými styly ve formátu CSS.

Zadáním příkazu `./generator -devel` spustíme aplikaci nastavenou ve vývojovém módu (viz výše), která poskytuje uživatelské rozhraní přeložené v předchozím kroku.

7.3 Produkční sestavení

Produkční nasazení je zamýšleno pomocí kontejnerizačního nástroje, který splňuje požadavky OCI[1] [36], jako jsou například Docker či Podman, případně v orchestračním nástroji Kubernetes. Repozitář se zdrojovými soubory obsahuje v kořenové složce soubor `Dockerfile`, který popisuje způsob sestavení do aplikačního obrazu.

Výhodou využití kontejnerizačního nástroje je jednoduchá distribuce výsledného řešení a kontrola nad závislostmi, jako jsou například knihovny. Nevýhodou může být poměrně vyšší náročnost na diskový prostor, ovšem tento problém řešíme pomocí použití minimálních základních obrazů alpine a využitím více fázových sestavení [37].

[1] Open container initiative

```
$ docker build . --tag generator:latest
```

Výpis kódu 20 Sestavení obrazu kontejnerizačním nástrojem Docker

Sestavení kontejnerizačním nástrojem docker je vidět v ukázce 20. Proces sestavení nejdříve vytvoří kontejner ve kterém sestaví artefakty uživatelského rozhraní pomocí NodeJS. Dále vytvoří kontejner a sestaví aplikaci server. V poslední fázi sestavení zkopíruje pouze artefakty ze sestavujících kontejnerů a vytvoří aplikační obraz. Tento obraz následně asociuje s předaným argumentem `tag`.

```
$ docker volume create
988d894692d6405aeec1a7208a663cffae6e03cd42a16cfc67791e5426b75475
$ docker run --detach --restart=always --name generator \
    --env-file ./env.txt -p 8080:8080 \
    -v 988d894692d6405aeec1a7208a663cffae6e03cd42a16cfc67791e5426b75475:/var/badger/db \
    generator:latest
```

Výpis kódu 21 Spuštění kontejneru v kontejnerizačním nástroji Docker

V ukázce 21 je zobrazeno jakým způsobem je spouštěna aplikace v kontejnerizačním nástroji Docker. Pro vytvoření úložiště databáze nejdříve vytvoříme `volume` [38], který bude připojen ke kontejneru. Následné volání příkazu `docker run` vykoná následující akce:

- Vytvoří kontejner se jménem `generator` vycházející z aplikačního obrazu `generator:latest`.
- Připojí předaný `volume` do souborového systému kontejneru na cestě `/var/badger/db`.
- Přesměruje port 8080 kontejneru na port 8080 lokálního rozhraní.
- Nastaví proměnné prostředí pro procesy běžící v kontejneru dle hodnot z předaného souboru proměnných prostředí `./env.txt`.
- Spustí kontejner na pozadí a instruuje kontejnerizační službu, aby se jej pokusila znovu spustit v případě pádu aplikace nebo restartu stroje.

7.4 Doporučení pro produkční nasazení

Jako poslední zmíníme některá doporučení pro nasazení v produkčním prostředí, která se týkají konfigurace, ale i limitů. Prvním doporučením je spouštění aplikace v kontejnerizačním nástroji, tento přístup zvyšuje bezpečnost nasazení, kvůli izolaci zdrojů využívaných aplikací od ostatních procesů na cílové platformě.

Taktéž využití kontejnerizačního nástroje eliminuje problémy se závislostmi na cílovém systému, aplikační obraz obsahuje všechny nezbytné závislosti aplikace na knihovny, a tedy jediným vyžadovaným nástrojem je právě kontejnerizační služba.

7.4.1 Otevření aplikace světu

Aplikace v základní konfiguraci otevírá port 8080 a nepoužívá TLS/SSL, což je záměrné nastavení. Nasazení aplikace přímo na některý ze systémových portů není vhodné a nemělo by být používáno. Místo otevření aplikace na těchto portech doporučujeme použití reverzní HTTP proxy, která bude obsluhovat základní HTTP/S porty včetně zabezpečení pomocí TLS/SSL a odpovídající požadavky bude předávat koncové aplikaci.

```
server {
    listen 80;
    listen [::]:80;
    listen 443 ssl http2;
    listen [::]:443 ssl http2;

    server_name example.com;

    if ($scheme = http) {
        return 301 https://$host$request_uri;
    }
    ssl_certificate /etc/secret/example.com/fullchain.pem;
    ssl_certificate_key /etc/secret/example.com/privkey.pem;

    location / {
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_pass http://localhost:8080;
    }
}
```

Výpis kódu 22 Konfigurace reverzní proxy pro server NGINX

V ukázce 22 můžeme vidět minimální konfiguraci pro server NGINX [39], který vytvoří reverzní proxy server zpřístupňující naše řešení. Konfigurace definuje následující atributy, které jsou nezbytné pro správné fungování řešení:

- Řádky 2-5 definují, že server bude poslouchat na portech 80 (HTTP) a 443 (HTTPS) s otevřeným spojením na všech dostupných rozhraních jak přes IPv4, tak i IPv6.
- Řádek 7 specifikuje serveru, že tato konfigurace je platná pouze pro požadavky směrované na doménové jméno `example.com`.
- Řádky 9-11 vynucují změnu spojení z HTTP na HTTPs pomocí odpovědi o přesměrování dotazu.
- Řádky 12-13 nastavují klíč a certifikát použitý pro zabezpečení spojení pomocí protokolu TLS/SSL.
- Řádky 16-17 nastavují hlavičky přeposlaného požadavku, což je důležité pro správné fungování spojení na REST api.
- Řádek 19 instruuje NGINX k provedení předání požadavku na zadanou lokální adresu, v našem případě na otevřený port 8080, který obsluhuje generátor.

7.4.2 Zabezpečení aplikace

V kombinaci s požadavkem na nasazení za HTTP proxy, velmi doporučujeme uzavření přístupu k aplikačnímu portu pomocí firewallu. Tím omezujeme možný prostor pro útoky na aplikaci a předáváme kontrolu samotné HTTP proxy, která při správné konfiguraci dokáže takové situace detekovat a případně se bránit.

Zároveň velmi doporučujeme nastavení proxy s vynucením TLS/SSL pro komunikaci (viz ukázka 22) s pomocí certifikátů vydaných certifikačními autoritami. Tím se stává aplikace důvěryhodnou pro prohlížeče, a lze tedy garantovat funkčnost jak webového klienta, tak REST rozhraní a jejich vzájemné komunikace.

7.4.3 Limity aplikace

Pro nastavení limitů aplikace musíme vycházet z cílové platformy na které aplikaci nasadíme. Aplikace během svého běhu využívá prostředky různým způsobem dle aktuální potřeby, a může tedy docházet k výkyvům v jejich využívání.

Jedním z limitů aplikace je počet pracovních gorutin použitých při generování grafů. Tyto gorutiny jsou většinu doby ve stavu čekání na událost a nevyužívají žádné zdroje. V okamžiku vzniku požadavku nicméně dochází k jejich probuzení a spuštění výpočetně náročného algoritmu, který operuje pouze v operační paměti, a tedy maximálně využívá procesor. Tedy počet pracovních gorutin musí být takový, aby na cílovém stroji zůstalo dostatek prostředků pro obsluhu nových spojení na server a správu perzistence. Z toho důvodu prvním doporučením je, že počet pracovních vláken by neměl překročit 75 % procesorové kapacity na počet vláken.

Dalším reálným limitem nasazení je objem operační paměti systému. Vyšší paměťovou náročnost mají operace generování grafu a předání výsledků uživateli, kdy dochází k načtení celého výsledku do operační paměti, a následnému překladu do požadovaného formátu. Zajištění nepřekročení těchto limitů realizujeme nastavením maximálního počtu vrcholů generovaného grafu a maximální velikostí požadavku na dávku.

Pokud budeme uvažovat nejnákladnější případ, tedy ohodnocený úplný graf, můžeme určit, že pro graf na 100 vrcholech je potřeba 56 KB prostoru, graf na 1 000 vrcholech potřebuje 8,6 MB a úplný ohodnocený graf na 10 000 vrcholech 896 MB.[2]

V případě předání výsledků je potřeba tuto velikost zdvojnásobit, protože dochází k překladu celého grafu do výstupní reprezentace. Tedy pokud bychom jako limit nastavili 10 000 vrcholů, využila by aplikace $\approx$ 1,7 GB. operační paměti pro zpracování takového požadavku.

Ještě náročnější je předávání dávek grafů. Vytváření souboru ve formátu zip se provádí přímo při zpracování požadavku, tedy pokud bychom uvažovali případ s maximálním počtem vrcholů 1 000 a maximální velikostí dávky 100, potom bude aplikace pro předání dávky uživateli 860 MB pouze pro načtení všech vygenerovaných grafů z dávky. Grafy se následně zpracují sekvenčně, tedy přeložený výsledek se v paměti drží pouze do okamžiku serializace.

Posledním limitem, který je potřeba uvažovat, je náročnost pro uložení dat na disku, ta odpovídá velikosti reprezentace v paměti, zvýšená o režijní náklady databáze – které jsou konstantní pro každý požadavek. Tento limit řešíme vhodným nastavením doby platnosti požadavku na graf.

Základní limity aplikace pro produkční prostředí jsme popsali výše a jsou zvoleny pro potřeby předmětu NI-PDP, tedy maximální počet uzlů grafu je 100 a maximální velikost dávky je 50, z toho plyne, že pro nejhorší možný případ požadavku bude využito $\approx$ 2,8 MB při serializaci dávky. Doba existence požadavku v databázi je následně nastavena na 1 hodinu, což je dostačující čas pro vyzvednutí požadavku.

[2] Což odpovídá našemu odhadu kvadratické paměťové náročnosti uložení grafu.

Kapitola 8
Závěr

V této práci jsme prošli celým procesem návrhu softwarového řešení, od počáteční teoretické rešerše, až po popis výsledného produktu a jeho nasazení. Vysvětlili jsme všechna návrhová rozhodnutí a pokryli požadavky kladené zadáním. Aplikace prošla řádným testováním a byla úspěšně zveřejněna ve veřejném repozitáři[1] pod otevřenou licencí APACHE v2.0[2].

Ačkoliv práce zadání naplnila, její potenciál rozhodně není naplněn. Příležitostí pro její další vývoj je tedy široká paleta a nyní si vysvětlíme některé z nich.

V současném stavu řešení používá vestavěnou databázi pro ukládání a předávání dat, což limituje jeho škálovatelnost, protože samotné generující rutiny musí běžet ve stejném procesu jako webový server obsluhující komunikaci s klienty. Prvním možným rozšířením práce je tedy vytvoření nové implementace služby perzistentní vrstvy, která by využívala nějakou samostatně stojící databázi typu Redis.

Toto oddělení by následně umožnilo rozdělení současného monolitického serveru do samostatných komponent, které by se daly dle potřeby škálovat ve vhodném cloudovém řešení na základě velikosti a počtu příchozích požadavků.

Dalším možným rozšířením aplikace je implementace uživatelských účtů a autentizace. Nyní pro svoji funkčnost aplikace autentizaci nevyužívá, protože je navržena pro okamžité použití, tedy očekává se, že uživatel nebude již jednou vygenerovaná data potřebovat znovu.

V tomto ohledu je jedním z potenciálně zajímavých rozšíření práce možnost ukládání generovaných dat dlouhodobě a umožnění jejich sdílení mezi uživateli. Například by mohla implementovat nového aktéra učitele, který by mohl připravit základní testovací sady pro studenty.

Jinou funkcí vhodnou pro implementaci, kterou by aplikace mohla s dlouhodobým úložištěm přidat, je analýza vyřešených požadavků a uložení takových, které běžně negeneruje. To by mohlo rozšířit různorodost generovaných náhodných grafů o další izomorfismy, jejichž pravděpodobnost vygenerování je velmi nízká, nicméně pro jisté druhy problémů jsou vhodné.

[1] `https://github.com/soch-fit/GraphGenerator`
[2] `https://www.apache.org/licenses/LICENSE-2.0`

Bibliografie

1. WILSON, Robin J. History of Graph Theory. In: GROSS, Jonathan L; YELLEN, Jay; ZHANG, Ping (ed.). *Handbook of graph theory, second edition*. 2. vyd. Philadelphia, PA: Chapman & Hall/CRC, 2013, kap. 1.3, s. 31–46. Discrete Mathematics and Its Applications. ISBN 978-1-4398-8019-7.
2. GROSS, Jonathan L; YELLEN, Jay. *Graph theory and its applications*. 3. vyd. Oakville, MO: Apple Academic Press, 2018. Textbooks in Mathematics. ISBN 978-1-4822-4948-4.
3. GROSS, Jonathan L; YELLEN, Jay. Fundamentals of Graph Theory. In: GROSS, Jonathan L; YELLEN, Jay; ZHANG, Ping (ed.). *Handbook of graph theory, second edition*. 2. vyd. Philadelphia, PA: Chapman & Hall/CRC, 2013, kap. 1.3, s. 2–20. Discrete Mathematics and Its Applications. ISBN 978-1-4398-8019-7.
4. STEGER, A.; WORMALD, N. C. Generating Random Regular Graphs Quickly. *Combinatorics, Probability and Computing* [online]. 1999, roč. 8, č. 4, s. 377–396 [cit. 2023-04-21]. Dostupné z DOI: `10.1017/S0963548399003867`.
5. MERINGER, Markus. Fast generation of regular graphs and construction of cages. *Journal of Graph Theory* [online]. 1999, roč. 30, č. 2, s. 137–146 [cit. 2023-04-21]. Dostupné z DOI: `https://doi.org/10.1002/(SICI)1097-0118(199902)30:2<137::AID-JGT7>3.0.CO;2-G`.
6. KIM, J.; VU, Van Hanh. Generating Random Regular Graphs. *Combinatorica* [online]. 2006, roč. 26, s. 683–708 [cit. 2023-04-21]. Dostupné z DOI: `10.1007/s00493-006-0037-7`.
7. MCKAY, Brendan D; WORMALD, Nicholas C. Uniform generation of random regular graphs of moderate degree. *Journal of Algorithms* [online]. 1990, roč. 11, č. 1, s. 52–67 [cit. 2023-04-29]. ISSN 0196-6774. Dostupné z DOI: `https://doi.org/10.1016/0196-6774(90)90029-E`.
8. BORŮVKA, Otakar. O jistém problému minimálním. *Práce Moravské přírodovědecké společnosti* [online]. 1926, s. 37–58 [cit. 2023-04-21]. Dostupné z: `http://dml.cz/dmlcz/500114`.
9. WIKIPEDIA CONTRIBUTORS. *Spanning tree — Wikipedia, The Free Encyclopedia* [online]. 2022 [cit. 2023-04-29]. Dostupné z: `https://en.wikipedia.org/w/index.php?title=Spanning_tree&oldid=1121900925`.
10. *Frequently Asked Questions (FAQ)* [online]. Google [cit. 2023-04-30]. Dostupné z: `https://go.dev/doc/faq`.
11. *Docker docs*. Docker overview [online]. Docker inc. [cit. 2023-05-10]. Dostupné z: `https://docs.docker.com/get-started/overview/`.
12. *Kubernetes documentation*. Overview [online]. kubernetes.io [cit. 2023-05-10]. Dostupné z: `https://kubernetes.io/docs/concepts/overview/`.

13. TIŠNOVSKÝ, Pavel. Rozhraní, metody, gorutiny a kanály v programovacím jazyku Go. *ROOT.CZ* [online]. 2018 [cit. 2023-05-02]. Dostupné z: https://www.root.cz/clanky/rozhrani-metody-gorutiny-a-kanaly-v-programovacim-jazyku-go/.
14. *What is Angular?* [Online]. Angular.io [cit. 2023-04-30]. Dostupné z: https://angular.io/guide/what-is-angular.
15. *Angular documentation* [online]. Angular.io [cit. 2023-05-10]. Dostupné z: https://angular.io/docs.
16. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures* [online]. Irvine, 2000 [cit. 2023-04-21]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Dis. pr. University of California.
17. FIELDING, Roy T.; NOTTINGHAM, Mark; RESCHKE, Julian. *HTTP/1.1* [RFC 9112]. RFC Editor, 2022. Request for Comments, č. 9112. Dostupné z DOI: 10.17487/RFC9112.
18. FIELDING, Roy T.; NOTTINGHAM, Mark; RESCHKE, Julian. *HTTP Semantics* [RFC 9110]. RFC Editor, 2022 [cit. 2023-05-05]. Request for Comments, č. 9110. Dostupné z DOI: 10.17487/RFC9110.
19. *Gin Documentation* [online]. gin-gonic.com [cit. 2023-05-09]. Dostupné z: https://gin-gonic.com/docs/.
20. *Badger documentation* [online]. Dgraph Labs, Inc. [cit. 2023-05-10]. Dostupné z: https://dgraph.io/docs/badger/get-started/.
21. JAIN, Manish Rai. *Concurrent ACID Transactions in Badger* [online]. Dgraph Labs, Inc., 2017-10-05 [cit. 2023-05-06]. Dostupné z: https://dgraph.io/blog/post/badger-txn/.
22. KLEIER, Tim. *How to Version a REST API* [online]. freecodecamp.org, 2020-03-03 [cit. 2023-05-07]. Dostupné z: https://www.freecodecamp.org/news/how-to-version-a-rest-api/.
23. CROCKFORD, Douglas; MORNINGSTAR, Chip. *Standard ECMA-404 The JSON Data Interchange Syntax* [online]. 2017 [cit. 2023-04-30]. Dostupné z DOI: 10.13140/RG.2.2.28181.14560.
24. CHACON, Scott; STRAUB, Ben. *Pro git: Everything you need to know about Git* [online]. Second. Apress, 2023. 2.1.391-2-g2fbc35b8 [cit. 2023-05-02]. Dostupné z: https://git-scm.com/book/en/v2.
25. GERRAND, Andrew. Go maps in action. *Go blog* [online]. 2013 [cit. 2023-05-02]. Dostupné z: https://go.dev/blog/maps.
26. *Go documentation.* Package math.rand [online]. Google [cit. 2023-05-02]. Dostupné z: https://pkg.go.dev/math/rand.
27. WIKIPEDIA CONTRIBUTORS. *Interval tree — Wikipedia, The Free Encyclopedia* [online]. 2022 [cit. 2023-05-06]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Interval_tree&oldid=1122891003.
28. *Go documentation.* Package http [online]. Google [cit. 2023-05-02]. Dostupné z: https://pkg.go.dev/net/http.
29. BARTH, Adam. *HTTP State Management Mechanism* [RFC 6265]. RFC Editor, 2011 [cit. 2023-05-06]. Request for Comments, č. 6265. Dostupné z DOI: 10.17487/RFC6265.
30. *Go documentation.* Package sync [online]. Google [cit. 2023-05-02]. Dostupné z: https://pkg.go.dev/sync.
31. *Go documentation.* Package encoding/gob [online]. Google [cit. 2023-05-07]. Dostupné z: https://pkg.go.dev/encoding/gob.

32. *Bootstrap documentation* [online]. getbootstrap.com [cit. 2023-05-10]. Dostupné z: https://getbootstrap.com/docs/5.3/getting-started/introduction/.

33. *Typescript documentation*. Interfaces [online]. Microsoft [cit. 2023-05-10]. Dostupné z: https://www.typescriptlang.org/docs/handbook/interfaces.html.

34. *Go documentation*. Package testing [online]. Google [cit. 2023-05-02]. Dostupné z: https://pkg.go.dev/testing.

35. *Pytest documentation* [online]. pytest.org [cit. 2023-05-09]. Dostupné z: https://docs.pytest.org/en/7.3.x/.

36. MCCARTY, Scott. *A Practical Introduction to Container Terminology* [online]. Red Hat, Inc., 2018-02-22 [cit. 2023-05-03]. Dostupné z: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction#.

37. *Docker docs*. Multi-stage builds [online]. Docker inc. [cit. 2023-05-03]. Dostupné z: https://docs.docker.com/build/building/multi-stage/.

38. *Docker docs*. Volumes [online]. Docker inc. [cit. 2023-05-05]. Dostupné z: https://docs.docker.com/storage/volumes/.

39. *NGINX Reverse Proxy* [online] [cit. 2023-05-03]. Dostupné z: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/.

Obsah přiloženého média

```
readme.txt ........................ stručný popis obsahu média
build
 ├─ generator.tar ................. vyexportovaný image sestavení pro docker
 └─ readme.txt .................... stručný popis použití vyexporovaného obrazu
src
 ├─ GraphGenerator ................ zdrojové kódy implementace
 │   ├─ cmd ....................... zdrojové soubory pro spustitelné programy v Go
 │   ├─ pkg ....................... knihovní funkce generátoru grafů
 │   ├─ ui ........................ složka s projektem uživatelského rozhraní
 │   └─ test ...................... testovací projekt pro integrační testování aplikace
 └─ thesis ........................ zdrojová forma práce ve formátu LaTeX
text .............................. text práce
 └─ thesis.pdf .................... text práce ve formátu PDF
```