VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VYUŽITÍ DYNAMICKÉHO PROGRAMOVÁNÍ V GRAFOVÝCH ALGORITMECH

DYNAMIC PROGRAMMING IN GRAPH ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTIN BILOŠ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. IVANA BURGETOVÁ, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Biloš Martin**
Program: Informační technologie
Název: **Využití dynamického programování v grafových algoritmech**
Dynamic Programming in Graph Algorithms
Kategorie: Algoritmy a datové struktury

Zadání:

1. Seznamte se s technikou dynamického programování.
2. Prostudujte problémy nad grafy, které lze touto technikou efektivně řešit.
3. Po dohodě s vedoucí navrhněte aplikaci, která bude demonstrovat výhody dynamického programování na vybraných úlohách nad grafy.
4. Navrženou aplikaci implementujte.
5. Aplikaci otestujte a zhodnoťte dosažené výsledky.

Literatura:

- Mareš, M., Valla, T.: *Průvodce labyrintem algoritmů,* CZ.NIC, 2017, ISBN 978-80-88168-19-5.
- Demel, J.: *Grafy a jejich aplikace*, Academia, 2002

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Burgetová Ivana, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 30. října 2018

Abstrakt

Tato práce se zabývá grafovými algoritmy, jejich využitím a přínosem optimalizační metody dynamického programování v tomto sektoru. Grafové algoritmy najdou využití v mnoha odvětvích lidské činnosti i dnes. Používají se ve směrování paketů nebo například v navigaci.

V práci jsou zpracovány tři metody, které patří mezi grafové algoritmy. Tyto problémy řeším klasickým i dynamickým způsobem a následně jsou zjištěná data porovnána. Výsledky jsou následně předvedena uživateli pomocí aplikace.

Abstract

This work is about graph algorithms, their use and the benefit of the optimization method of dynamic programming in this subject. Graph algorithms find use in many sectors of human activity. They are used in packet routing or, for example, navigation.

There are three methods of graph algorithms used in this work. This problems are solved with classic and dynamic way and measured data are compared. Results are then presented to the user via the graphic application.

Klíčová slova

Dynamické programování, C++, gtkmm, optimalizace, graf, nejkratší cesta, obchodní cestující, skrytý markovův model, Viterbiho algoritmus

Keywords

Dynamic programming, C++, gtkmm, optimalization, graph, shortest path, traveling salesman, hidden markov model, Viterbi algorithm

Citace

BILOŠ, Martin. *Využití dynamického programování v grafových algoritmech.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ivana Burgetová, Ph.D.

Využití dynamického programování v grafových algoritmech

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením paní Ing. Ivany Burgetové, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Biloš
14. května 2019

Poděkování

Chtěl bych poděkovat hlavně své vedoucí paní Ing. Ivaně Burgetové Ph.D. za výborné rady a trpělivost, kterou se mnou měla. Také bych rád poděkoval své rodině a kolegům za podporu.

Obsah

1 Úvod **3**

2 Dynamické programování v grafových algoritmech **4**
2.1 Dynamické programování 4
2.2 Grafy 5
2.2.1 Neorientovaný graf 5
2.2.2 Orientovaný graf 5
2.2.3 Markovovy skryté modely 5
2.3 Grafové algoritmy 6
2.3.1 Nejkratší cesta 7
2.3.2 Problém obchodního cestujícího 8
2.3.3 Dekódování sekvence - Markovovy skryté stavy 9

3 Návrh a implementace **11**
3.1 Použité technologie 11
3.1.1 Jazyk C++ 11
3.1.2 Gtkmm knihovna 12
3.1.3 Správa grafů 13
3.2 Návrh aplikace 13
3.3 Implementace aplikace 14
3.3.1 Postup vývoje 14
3.3.2 Implementace z hlediska GUI 18
3.3.3 Součásti aplikace 20
3.4 Ovládání a popis aplikace 21
3.5 Možnost následného využití 25

4 Porovnání algoritmů **27**
4.1 Nejkratší cesta 28
4.2 Obchodní cestující 29
4.3 Dekódování sekvence 32

5 Testování **35**

6 Závěr **36**

Literatura **37**

A Manuál **38**

B Formát grafových souborů **39**

C obsah paměťového média **41**

Kapitola 1

Úvod

Tato práce pojednává o využití optimalizační metody dynamického programování v oblasti grafových algoritmů, o jejím přínosu a porovnání s neoptimalizovanou implementací algoritmu. Smyslem této práce je demonstrovat tento přínos na vybraných algoritmech. Práce se dá rozdělit na dvě základní součásti: teoretickou a praktickou.

V teoretické části jsou popsány základní pojmy a problémy, kterými se tato práce zabývá, a které mají přesah do praktické části práce. Bude celkově nastíněna situace ohledně vybraných algoritmů a dynamického programování, které zde bude blíže popsáno. Blíže budou popsány algoritmy nalezení nejkratší cesty, problém obchodního cestujícího a dekódování sekvence.

Praktickým výstupem práce je aplikace s grafickým uživatelským rozhraním, postavená na jazyku C++ s využitím grafické knihovny GTKmm, která implementuje vyjmenované algoritmy. Aplikace umožňuje řešit zpracocované problémy nad vlastními grafy. Dále byla použita pro testování těchto implementací a porovnání výstupů.

Kapitola 2

Dynamické programování v grafových algoritmech

Tato kapitola pojednává o optimalizační metodě dynamického programování. V první části je rozebrána samotná metoda a v druhé části se pojednává o grafových algoritmech, kde by tato optimalizační metoda mohla přinést výrazné zrychlení. Přítomný je také popis úloh s jejich implementacemi využívajícími dynamické programování. Nejdůležitějším posláním této kapitoly tedy je spíše teoretický popis dynamického programování, použitých algoritmů a vysvětlení pojmů a znalostí, které budou použity v následujícím textu. Lze ji chápat také jako přípravu znalostí a vstup do problematiky pro lepší pochopení pozdějšího textu.

2.1 Dynamické programování

Dynamické programování je optimalizační metoda, která je založena na rozkládání složitějších problémů na menší a hlavně opakující se podproblémy, což ji předurčuje pro využití hlavně v případech, které pro své řešení mohou vyžadovat rekurzi, ale předně je možná jejich dekompozice na jednodušší opakující se celky [8].

Metoda má do dnešních dní velmi širokou škálu algoritmů, které staví na této myšlence a používají ji pro řešení různorodých problémů.

Mezi její hlavní výhody patří ušetření času a systémových prostředků, za pomoci uložení výsledků opakujících se stejných podproblémů. Typickým příkladem využití může být výpočet Fibbonačiho posloupnosti, kde jsou při využití rekurze přítomny často opakující se výpočty v rámci podstromů [8]. Pro jeden problém může existovat více řešení využívajících dynamické programování, která se mohou lišit v pohledu na danou problematiku.

Z ukládání mezivýsledků však plyne i její nevýhoda, kterou může být větší paměťová náročnost a v některých specifických případech i zpomalení. Tento případ je zmíněn podrobněji v kapitole 4.2. Dynamické programování bohužel není dostupné pro libovolný algoritmus. Algoritmus musí splňovat více podmínek. Mezi nimi například Bellmanovou rovnici, z které vyplývá, zda daný problém se dá rozložit na 2 a více samostatných podproblémů, které lze řešit separátně [2].

Princip dynamického programování spočívá v nalezení rekurzivního algoritmu. Nalezní opakovaných výpočty stejných podproblémů, jejichž výsledky budou ukládány v tabulce. Důležitou vlastností je také odstranění rekurze za pomocí vhodného pořadí řešení podproblémů [8].

2.2 Grafy

Pojem Graf bude v rámci této práce symbolizovat strukturu, krerá má ve velmi zjednodušeném pojetí seznam vrcholů a hran, které spojují již zmíněnýné vrcholy. Daná hrana vždy spojuje 2 specifické vrcholy a tím mezi nimi vytváří propojení hlavně na logické úrovni [4]. V rámci této práce budou zastoupeny pouze dva základní typy grafů. Neorientovaný graf 2.2.1 a graf popisující Markovovy skryté modely 2.2.3.

2.2.1 Neorientovaný graf

Neorientovaný graf budeme chápat takovým grafem, kde je hrana mezi dvěma vrcholy oboustranně průchozí a má oboustranně stejnou hodnotu. Řečeno jinak jedná se o graf, kde není důležité, který vrchol v daném přechodu je na prvním a druhém místě [4]. V rámci použitých grafů se bude počítat s tím, že koncové vrcholy hrany jsou rozdílné, tedy že není možné udělat hranu mezi jedním a tím samým vrcholem.

Matematicky lze vyjádřit neorientovaný graf jako trojici $G = (V, E, \epsilon)$, kde V je konečná množina vrcholů, E je konečná množina neorientovaných hran a ϵ nazýváme vztahem incidence. Toto zobrazení přiřazuje každé hraně $e \in E$ dvojici vrcholů a určité ohodnocení[4].

Typickým příkladem takového grafu může být dopravní spojení mezi městy nebo i spojení určitých míst v rámci samotného města. Také se může jednat o pěší vzdálenost mezi určitými zajímavými místy. Možností, jakým způsobem dojít k této reprezentaci grafu je mnoho.

2.2.2 Orientovaný graf

Orientovaný graf je velmi podobný neorientovanému grafu popsanému v minulé podkapitole. Jediný zásadní rozdíl se skrývá ve vlastnosti, která zajišťuje, že spojení mezi vrcholy není symetrické a dokonce může být pouze jednosměrné.

Matematicky lze vyjádřit orientovaný graf vyjádřit velmi podobně jako graf neorientovaný s tím rozdílem, že u dvojice vrcholů rozlišujeme pořadí bodů. První z nich nazýváme počátečním a druhý koncovým bodem [4].

Využití tohoto grafu je reálnější než u neorientovaných grafů. Typickým příkladem je doprava, kdy letenka z jednoho místa do druhého může stát dvojnásobek ceny letenky opačným směrem. Také může zohledňovat uzavírky na cestách. Toto paradigma má mnohem více možností, které může prezentovat. Její použití směřuje do oborů jako je informatika, genetika, bankovnictví a mnoho dalších [4].

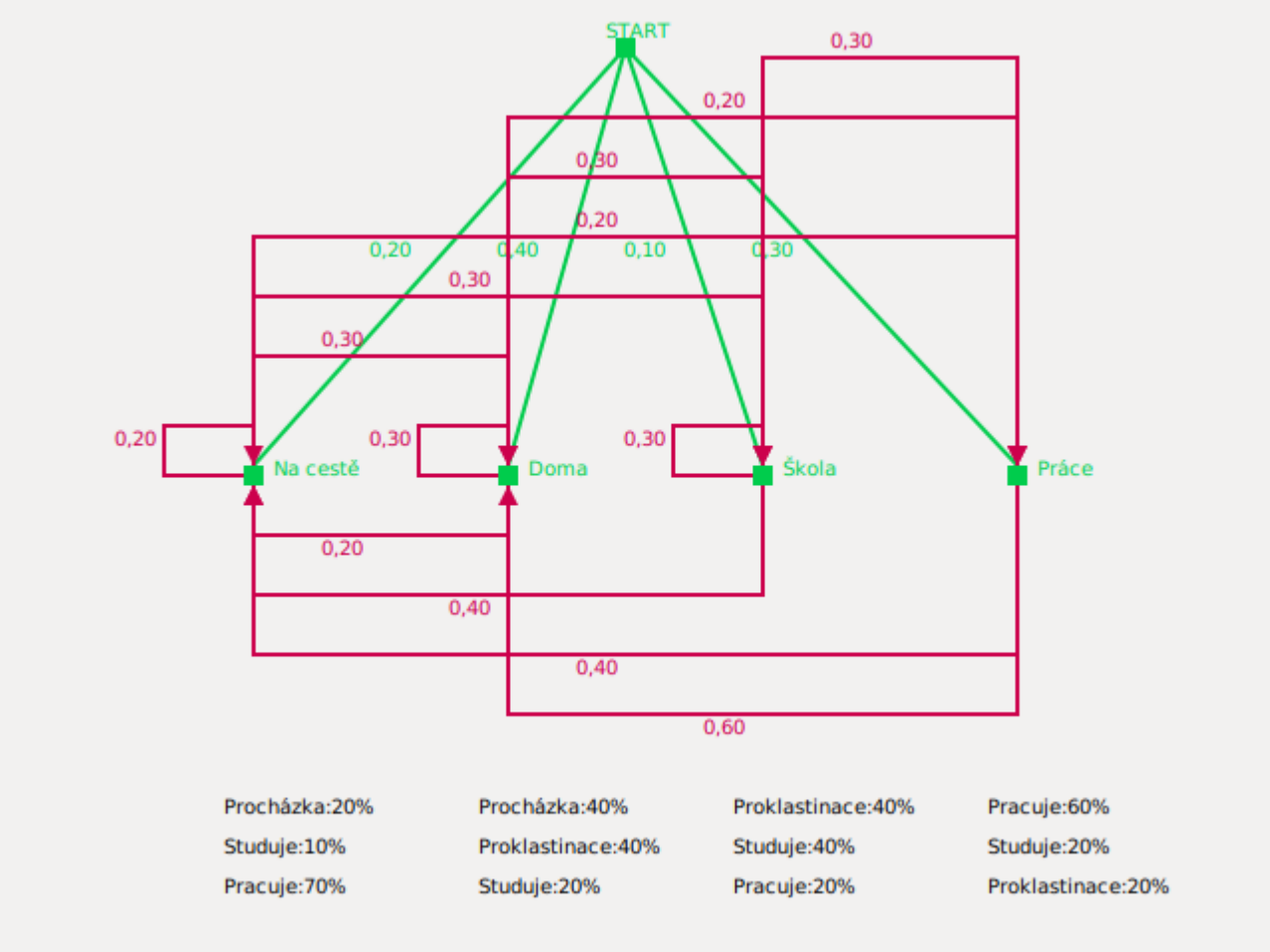
2.2.3 Markovovy skryté modely

Markovůvy skryté modely jsou statistické modely, popisjící Markovovy modely se skrytými stavy. To znamená, že na výstupu nemáme posloupnost stavů, ale posloupnost pozorování, přičemž stav, ze kterého dané pozorování pochází zůstává skryt. Pro jejich popis využijeme stavy (vrcholy), pravděpodobnosti přechodů mezi nimi (hrany), pozorování a pravděpodobnost určitého pozorování v určitém stavu.

Pouze v ohledu samotných stavů a přechodů mezi nimi se jedná o orientovaný graf, popsaný v minulé kapitole 2.2.2. S tím rozdílem, že hrany jsou ohodnoceny pravděpodobností přechodu mezi stavy. Avšak stavy mají další vlastnost, kterou je emitování pozorování, což jinými slovy znamená, že pokud se algoritmus nachází v určitém stavu, tak s určitou pravděpodobností emituje pozorování, které je součástí výstupu daného modelu. Dalším

rozdílem oproti orientovaným grafům je neexistující startovní vrchol, ale pravděpodobnost v jakém stavu se algoritmus nachází v počáteční fázi průchodu a samozřejmě již zmíněná pravděpodobnost přechodu mezi stavy [6].

Výsledkem takového grafu bývá posloupnost pozorování, emitovaného stavy, jejichž posloupnost je nám neznámá. Pochopitelnějším vysvětlením bude následující obrázek 2.1, který ukazuje možnou grafovou podobu entit tohoto modelu. Jedná se již o jednu z obrazovek z výsledné aplikace. Zelené hrany označují startovní možnosti a jejich pravděpodobnosti. Zelené body v prostřední části obrázku jsou stavy a červené čáry jsou hrany i s jejich pravděpodobnostmi. Pod každým stavem následuje seznam možnách pozorování v daném stavu.



Obrázek 2.1: Možná grafová podoba Markovových skrytých stavů

Tyto modely se uplatňují v mnoha sektorech lidské činnosti. Může jít o bioinformatiku, informatiku, bankovnictví, genetiku a mnoho dalších možností [6].

2.3 Grafové algoritmy

Pod tímto velmi obecným pojmem budeme v rámci této práce chápat takové algoritmy, které nějakým způsobem pracují s grafy, nebo je pro ně toto paradigma s úspěchem využitelné. Následující podkapitola se bude zabývat některými z nich.

Díky možnosti přeložení algoritmických otázek do teorie grafů je možnost často nalézt velmi snadné a elegantní řešení daného problému [2].

2.3.1 Nejkratší cesta

Základním algoritmem, který dokumentuje tato práce, je algoritmus nejkratší cesty. Jedná se o algoritmus, jehož cílem je nalezení nejkratší, respektive nejméně ohodnocené, cesty mezi dvěma vrcholy. Orientuje se pomocí ohodnocení hran, které nějakým způsobem reflektuje náročnost hrany z pozorovaného hlediska. Může jít o vzdálenost, čas nebo například energetické ztráty v rámci elektrické sítě. Vždy záleží na aktuálním náhledu na situaci jako na celek.

Na řešení tohoto problému existuje více různých algoritmů. Implementace základního algoritmu je poměrně jednoduchá, ale velmi náročná na čas i systémové prostředky. Spočívá v rekurzivním prohledávání stavů, kdy se v každém stavu vydáváme všemi možnými hranami do dalších stavů, přičemž se zvyšuje naakumulovaná hodnota o ohodnocení hrany [2]. Daný algoritmus se dá popsat následující posloupností akcí:

1. Algoritmus začíná ve startovním bodě s ohodnocením 0.
2. Kontroluje zda se nenachází v koncovém (tedy hledaném) bodě. Pokud ano, jde do 3, jinak následuje bod 4.
3. Kontroluje naakumulované ohodnocení, pokud je nižší než aktuálně uložené, tak ho přepíše za své a uloží svou cestu. Větev v obou případech končí.
4. Pro všechny hrany aktuálního vrcholu: Porovnává již navštívené vrcholy s druhým vrcholem hrany. Pokud se vrchol dané hrany nenachází v projité cestě, zvýší ohodnocení o ohodnocení hrany, změní vrchol na koncový vrchol hrany a volá se znovu (tedy od bodu 2). Pokud vrchol je nalezen v cestě, neprovádí nic.

Poté již stačí pouze po skončení průchodu vzít uložené hodnoty a nalezenou cestu.

Algoritmus nejkratší cesty, který na problém využívá metodu dynamického programování se jmenuje Bellmanův-Fordův algoritmus, který je popsán následovně:

1. Vytvoření paměťové struktury zapouzdřující informaci o nalezené cestě a jejím ohodnocení pro každý vrchol grafu a její vyplnění nesmyslně vysokými hodnotami. Také si vytváří seznam
2. Vytvoření fronty vrcholů, které jsou "otevřené". Přidání startovního vrcholu do fronty.
3. Pro aktuální vrchol fronty: Zkontroluje všechny hrany, ve kterých je vrchol.
4. Přičte svoje aktuální ohodnocení k ohodnocení hrany, pokud je menší než hodnota uložená ve struktuře, hodnotu přepíše a vrchol dá (pokud tam již není) do fronty.
5. Odstraňuje vrchol fronty a opakuje od bodu 2. Pokud je fronta prázdná, končí.

Následně již stačí pouze vrátit hodnotu, která se váže ke koncovému vrcholu. Výhoda tohoto přístupu je poměrně jasná. Neprochází se všechny možné cesty, ale pouze ty, které jsou perspektivní z aktuálního hlediska v hledání, což výrazně urychlí celkový průchod grafem.

Časová složitost Bellman-Fordova algoritmu je $\mathcal{O}(mn)$, kde m je počet hran a n je počtem vrcholů.

2.3.2 Problém obchodního cestujícího

Problém obchodního cetujícího spočívá v navštívení všech vrcholů v grafu za co nejmenší naakumulované ohodnocení. Problém má několik variant a pro potřeby tohoto srovnání byla vybrána varianta, která přidává podmínku, že každý stav může být navštíven pouze jednou, což mírně zlehčuje situaci tím způsobem, že některé typy grafů bohužel nebudou mít řešení, ale základní algoritmus se mírně zrychlí. To znamená, že musí existovat alespoň jeden uzavřený tah, který projde všechny vrcholy grafu. Problém má využití v dopravě, či rozvozových službách, ale díky jeho náročnosti se využívají jeho různé optimalizace.

Základní algoritmus funguje podobně jako základní algoritmus hledání nejkratší cesty 2.3.1. Rozdíly jsou hlavně v podmínkách ukončení, podle kterých se algoritmus řídí, a také algoritmus se může (respektive musí) vrátit do startovního vrcholu. Pro jistotu však přidávám popis:

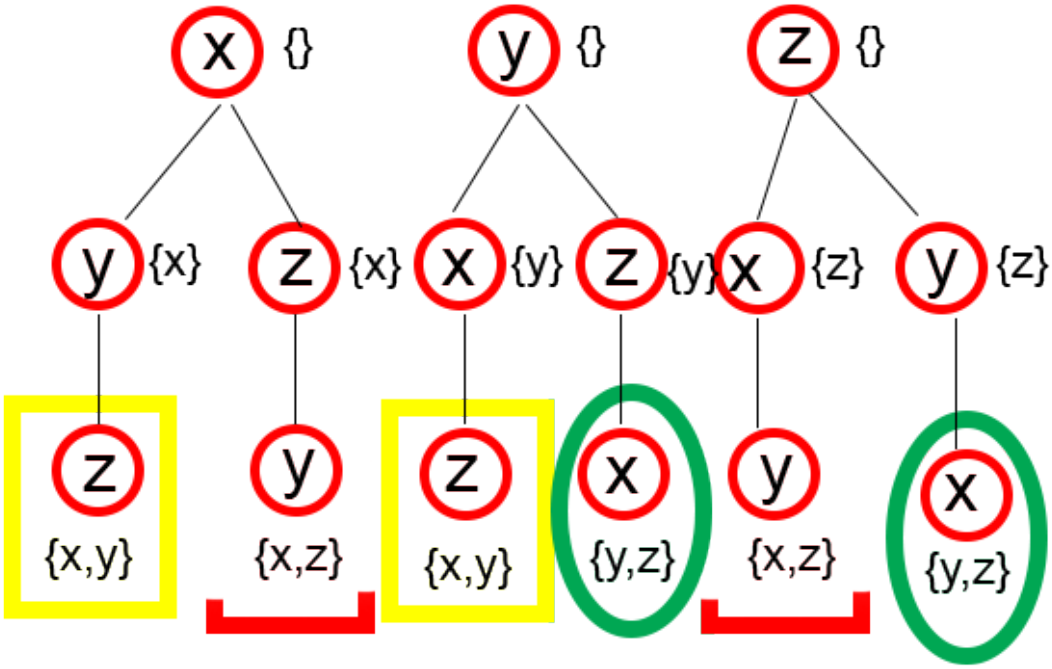
1. Algoritmus začíná ve startovním bodě s ohodnocením 0.
2. Kontroluje zda se nenachází zase v počátečním bodě a zda tah prošel všechny vrcholy. Pokud ano, jde do 3, jinak následuje bod 4.
3. Kontroluje, zda již nebylo nalezené řešení, pokud ano, porovná ohodnocení a pokud je nižší, zapíše svůj tah. Pokud nebylo již nalezeno řešení, zapisuje se automaticky.
4. Pro všechny hrany s aktuálním vrcholem: Porovnává již navštívené vrcholy s druhým vrcholem hrany. Pokud se vrchol dané hrany nenachází v projité cestě (a nejde o startovní vrchol), zvýší ohodnocení o ohodnocení hrany, změní stav na koncový stav hrany a volá se znovu (tedy od bodu 2), pokud vrchol nalezen je, neprovádí nic.

Dynamickou cestou řeší tento problém Held-Karpův algoritmus. Tato metoda využívá matici sousednosti, která se používá pro správu ohodnocení hran sousedních vrcholů. Popsán je následujícím seznamem úkonů:

1. Algoritmus začíná vytvořením matice sousednosti a jejím naplněním.
2. Vytváří si frontu otevřených "cest"a dává na vrchol startovní vrchol (je možné dát libovolný)
3. Pokud je, vezme prvek z fronty, porovná jej s prvky fronty (porovnává na základě navštívených vrcholů), pokud nalezne shodu, porovná váhu, náročnější prvek fronty smaže.
4. Prvek vezme z fronty a "rozvine jej"(bod 5)
5. což znamená, že jde přes hrany, které vedou do ještě nenavštívených vrcholů, popřípadě startu. Cestám přidá druhý vrchol hrany, zvýší ohodnocení o ohodnocení hrany a dává do fronty.
6. Pokud fronta není prázdná (nebo nezbývá jediná položka), opakuje bod 3.

Toto řešení sice algoritmus zrychlí, ale spíše symbolicky [5], pro větší grafy je toto řešení stále daleko od ideálu. Zrychlení Held-Karpova algoritmu spočívá v nalezení takových cest, které se aktuálně nacházejí ve stejném vrcholu a během výpočtu již navštívili shodné vrcholy, avšak jejich aktuální ohodnocení je díky cestě přes různé hrany rozdílné. Principem

je tyto shody během výpočtu hledat a hůře ohodnocené z těchto stejných cest již dále nepočítat, protože jejich výpočet by byl zbytečný. Více vysvětlí obrázek 2.2.



Obrázek 2.2: Zobrazení průchodu Held-Karpova algoritmu, Inspirace obrázku: [5]

Stejně označené listy zobrazené na obrázku jsou ony stejné cesty, které jsou aktuálně ve stejném stavu a na nich probíhá vybrání lepší z nich a odstranění té, která má vyšší ohodnocení.

Časová složitost naivního přístupu je $\mathcal{O}(n!)$ a časové složitost dynamického přístupu je $\mathcal{O}(2^n n^2)$ [7] [5], přišemž n je v tomto případě počet vrcholů.

2.3.3 Dekódování sekvence - Markovovy skryté stavy

Tento algoritmus řeší problém nalezení nejpravděpodobnější posloupnosti skrytých stavů na základě posloupnosti pozorování, která jsou daná algoritmu na vstup. Tato metoda má široké využití i v samotném rámci informačních technologií, kdy se používá například v sítích, práci se zvukovými stopami a signály obecně. využití nalezne také v genetice.

Základní algoritmus je stále relativně jednoduchý, kdy projdeme všechny možnosti, které se nám naskytují. Popis je následující:

1. Pokud není žádná cestu, použije startovní pravděpodobnosti stavů, vynásobené možností prvního pozorování. Cestu nastaví na daný stav i s jejím ohodnocením a volá se znovu.
2. Kontroluje délku cesty a délku sekvence pozorování. Pokud se shodují, následuje bod 3, pokud ne, bod 4
3. Pokud je pravděpodobnost vyšší než uložená, nahradí ji a uloží i cestu. V obou případech větev výpočtu končí.

4. Pokusí se daný stav "rozvinout", tedy použije všechny možné hrany do dalších stavů, jejich pravděpodobností vynásobí aktuální pravděpodobnost, kterou následně vynásobí pravděpodobností pozorování z nového stavu, nový stav přidá do cesty.

5. Opakuje od bodu 1 rekurzivním voláním

Dynamické řešení se nazývá Viterbiho algoritmus, který je široce využíván v bioinformatice, dekódování zvukových stop nebo například i ve výše zmíněných sítích. jeho formálnější popis:

1. Vytvoření tabulky, která udržuje informace o "nejpravděpodobnějším"předchůdci daného stavu s daným pozorováním.

2. Vytvoření tabulky, zatím nejpravděpodobnější cesty pro každý stav.

3. Začíná naplněním prvních prvků pro každý stav, pomocí startovních pravděpodobností.

4. Pro každý stav, po délku sekvence opakuj:

5. Hledej nejpravděpodobnějšího předchůdce ze všech cest, se zohledněním pravděpodobnosti přechodu i pravděpodobnosti pozorování.

6. Rozšiř cestu o minulý stav

7. Konec cyklu;

8. Nakonec algoritmus jde řešením pozpátku po nejpravděpodobnějších stavech, které uhcovává tabulka cest.

Viterbiho algoritmus má významnou výhodu oproti nedynamické metodě v tom, že neprochází přímo všechny možné cesty, ale pouze zkoumá, jak se daný algoritmus do daného stavu dostal z minulého. Časová složitost Viterbiho algoritmu je $\mathcal{O}(TS^2)$, přičemž T je délka hledané sekvence a S je počet stavů [3]. Matematická složitost naivního algoritmu je $\mathcal{O}(TS^T)$ [1].

Kapitola 3

Návrh a implementace

Při návrhu a celkové implementaci bylo mnoho překážek, které bylo nutné překonat. V této kapitole budou popsány některé zajímavější pasáže z celkového vývoje. Kapitola bude rozdělena do dvou hlavních logických celků. V kapitole budou postupně rozebrány rozebrány primárně použité technologie 3.1 a celkový návrh 3.2. V následující části 3.3 se popis zaměří hlavně na implementaci aplikace jako celku a úskalí, která bylo nutné vyřešit a hlavně proč se použilo řešení, které je implementováno. V závěru kapitoly bude popsáno ovládání aplikace 3.4 a případné možnosti budoucího využití 3.5.

Z důvodu lepšího pochopení využití určitých technologií zde bude mírně rozšířeno zadání. Přesněji jde o ujasnění, že samotná aplikace využívá grafické rozhraní, které bylo využito z důvodu lepší ovladatelnosti a celkového kontextu aplikace. Toto rozhodnutí přináší mnoho nesporných výhod, ale i nevýhod. Oba pohledy budou zmíněny, pokud se k tomu bude vázat nějaký aktuálně rozebíraný problém.

3.1 Použité technologie

Tato část je věnována popisu možností a některých rozhodnutí na začátku vývoje, která ovlivnila celkové směrování. Aplikace celý vývoj směřovala k multiplatformnímu použití nebo alespoň k jeho, co největšímu zastoupení. Tím je myšleno, aby případný převod aplikace na jinou platformu stál, co nejméně úsilí, nejlépe žádné. Tento fakt by nesporně napomohl distribuci mezi více potencionálních uživatelů.

3.1.1 Jazyk C++

Prvním rozhodnutím, které zásadně ovlivnilo celou aplikaci je, který programovací jazyk bude využit pro následnou implementaci a zda je v jeho možnostech naplnit potřeby, které jsou nezbytné z hlediska splnění vytyčených cílů. Bylo rozhodnuto využít jazyk C++, přesněji standard C++14.

Jazyk C++ se řadí k nejpoužívanějším jazykům, určeným pro vývoj, na světě. Vzniknul jako pouhé rozšíření základního jazyka C, který začal být v jistých ohledech nedostačující. Toto rozšíření spočívalo hlavně v možnosti vývoje v objektově orientovaném prostředí, což velkou mírou usnadnilo vývoj složitějších konstruktů. I přes toto rozšíření však jazyk C++ patří k výkonostně úsporným jazykům. Tyto výhody vyplývají primárně z jazyka C. Daní za malou náročnost však je mimo jiné i velmi obsáhlá správa paměti, na kterou si vývojář musí dávat pozor.

Mezi další možnosti lze počítat jazyky Java nebo Python, které by mohly daný problém zvládnout, při správném použití, o úroveň lépe, s méně problémy a bez větších potíží s pamětí. Avšak to by samozřejmě neslo daň v podobě pravděpodobně celkově pomalejšího prostředí a některých podrobnějších specifik (náročné ovlivnění uvolňování paměti, pro začítečníka nezvyk syntaxe v případě Pythonu). [1]

3.1.2 Gtkmm knihovna

Vzhledem ke grafickému směřování aplikace bylo nutné vybrat některou z nabízených knihoven nebo frameworků na vytváření grafiky. Pokud bych samozřejmě nechtěl vytvořit vlastní, avšak to by bylo nad rámec této práce, protože i velmi jednoduchá grafická knihovnička by pravděpodobně byla nesmírně složitá.

Existuje několik různých multiplatformních knihoven pro vytváření grafických aplikací. Mezi hlavní zástupce patří GTK+ a Qt. Existuje několik dalších řešení, pro danou problematiku, která jsou obvykle mířena na některé součásti, které nejsou plně pokryty již zmíněnými řešeními. Typickým příkladem může být knihovna SFML, která je výtečná pro vytváření 2D grafiky hlavně herního obsahu, ale již pokulhává ve vytváření vlastního uživatelského rozhraní, které je pro tuto úlohu velmi důležité.

Qt se pravděpodobně již nedá nazývat pouze knihovnou, ale spíše celým frameworkem pro vytváření uživatelských rozhraní. Jedná se o velmi rozšířené a používané řešení. Výhodou je, že už v základu je objektově orientované. Bohužel její náročnost k pořádnému pochopení frameworku je relativně náročné. Jako příklad uvedu komplexnost některých řešení, která neodpovídají podobným řešením v rámci C++ standardu. Například uchovávání textu, kdy Qt používá vlastní implementaci. Tento fakt ještě podtrhuje velmi zanedbaná dokumentace (alespoň do určité verze, může jít už o překonaný problém), kdy je závažný problém najít řešení některých elementárních problémů.

Knihovna GTK+ vznikla původně jako grafické jádro aplikace Gimp. Avšak od té doby prošla velmi prudkým rozvojem a stala se jednou z hlavních knihoven pro vytváření uživatelských rozhraní v rámci systému Unix. Je na ní postavené například i uživatelské prostředí Gnome, které patří mezi velmi hojně používané na systémech linuxového typu [11]. Aktuálně se dokončují práce na verzi 4, která je dostupná v testovacích verzích (aktuálně verze 3.96).

GTK+ je vyvinuta v základním Jazyku C a díky tomu je velmi snadno portovatelná na mnoho různých programovacích jazyků. Hojně rozšířená je již základní verze, ale skutečnou sílu knihovně dodávají až její různé odnože, kdy mezi nejdůležtější patří pygtk pro jazyk Python nebo GTKmm (GTK minus minus) pro jazyk C++. Gtkmm je pouze zapouzdření samotného GTK+ do objektů, tak aby bylo využitelné objektově orientované paradigma. Její výhodou je použití a plná kompatilbilita s nativními C++ kontejnery a vlastnostmi jakou je dědičnost nebo polymorfismus [9].

Pro knihovnu Gtkmm (respektive GTK+) jsem se rozhodl hlavně díky jejímu zaměření na linuxové systémy, které jsou pro tuto aplikaci primární (ale se zachováním multiplatformního řešení). Svou roli jistě také hrálo to, že s touto knihovnou mám bohatší a lepší zkušenosti než v knihovně Qt. Qt mne také odradila svou velkou komplexností a také tím, že vývojová větev pro Linux patří spíše k těm okrajovým a je zde přítomno relativně velké množství bugů. Gtkmm je také mnohem propojenější se standardem C++, což je v rámci této aplikace nesporná výhoda.

[1] Tímto nemá být řečeno, že by se jiné jazyky pro tento účel nehodily, je možné použít širokou škálu různých jazyků, vždy záleží na použití i preferencích autora.

3.1.3 Správa grafů

Stejně jako pro grafiku existuje několik dostupných řešení pro správu grafů, dokonce s již předpřipravenými jistými typy úloh. Prvotní pokusy směřovaly k použití multiplatformní knihovny boost, která má velmi široké spektrum využití a použití. Z této knihovny velmi často čerpá i samotný standard při přidávání nových možností do jazyka C++. Patrně se jedná o velmi univerzální počin, protože je zde možné nastavit grafu a jeho entitám nepřeberné množství vzájemně ovlivnitelných vlastností. Rozhodně jde o dobrou vlastnost, avšak jako celek je poté řešení mnohem náročnější na pochopení a správné použití.

Vývoj tímto směrem (s využitím knihovny boost) však ustal po implementaci prvního jednoduchého grafu. Knihovna byla pro použití a potřeby této práce až moc složitá a v některých ohledech neintuitivní. Při hlubším zkoumání by pravděpodobně převážily kladné vlastnosti, ale i tak by se jednalo o až moc silnou knihovnu pro tento účel. Proto byla vyvinuta vlastní jednoduchá správa grafů, která může být modifikována přímo pro aktuální potřeby.

Použité řešení je v zásadě poměrně jednoduché. Jedná se o datovou strukturu, která v sobě uchovává jen informace o vrcholech a hranách mezi nimi, i s jejich ohodnocením. Třída má k dispozici také sadu metod, které pracují s již zmíněnou datovou částí. Řešení by mělo být poměrně snadno modifikovatelné za dodržení přítomných metod a jejich návratových dat. Bohužel řešení není dostatečně pružné, aby bylo možné jednoduše pojmout i Markovovy skryté modely (2.2.3), takže graf implementující tuto problematiku je oddělen.

K tomuto kroku bylo přistoupeno, po zvážení všech aktuálně uvědomnělých kladů i záporů takového řešení. Při využití stávající implementace základního grafu by bylo nutné nějakým způsobem data, která by dával takto implementovaný graf, dále korigovat a to by znamenalo zbytečnou zátěž, která by se projevila v grafické nebo algoritmické části. Bylo možné samozřejmě využít také metodu dědičnosti a tyto úpravy tedy udělat tímto způsobem, avšak k tomuto kroku taky nebylo přikročeno a raději byla vytvořena nová implementace grafu, trochu rozdílného rázu, pro Markovovy skryté modely.

3.2 Návrh aplikace

V samotném návrhu se nepočítalo s některými pozdějšími úpravami, avšak po celou dobu vývoje byl brán zřetel na prvotní naplánování aplikace. Základní myšlenkou aplikace je samozřejmě porovnávat dynamické a normální řešení dané problematiky, ale toto je poměrně široký pojem, který by měl být objasněn na správnou míru v rámci této podkapitoly. Porovnávání algoritmů bude probíhat na základě časové nebo prostorové složitosti.

Základem aplikace je komunikace s uživatelem a možnost nějakým způsobem měnit, načítat a ukládat zavedené grafy. V plánu také byla možnost nějakým způsobem uživateli ukázat daný graf, aby měl lepší představu o jeho celkové podobě a pokud možno také zobrazení samotného výsledku, ke kterému metody došly, aby bylo možné nějakým způsobem ověřit pravost použitých řešení, což platí, jak pro dynamické, tak i normální algoritmy.

Výhodou tohoto typu řešení je to, že uživatel nemusí být nijak technicky znalý, aby si mohl danou problematiku otestovat. Zjednodušení s sebou samozřejmě nese také to, že pokročilejší uživatel by možná uvítal více detailů řešení, která jsou před uživatelem skryta a také více dostupných výsledných dat. I přes tento směr by však aplikace měla být alespoň dostačující pro obě skupiny potenciálních uživatelů.

Samotným výstupem aplikace by měla být tabulka s výslednými změřenými časy algoritmů, což znamená nutnost implementace návazných funkcionalit. Aplikace by měla být

schopna výsledky také ukládat/načítat pro zlepšení celkové představy o problematice v širším rámci než pouze na pár grafech a také ukázat grafické zobrazení výsledných metod. Nejlepší možností zobrazení takových dat by měl být graf závislosti, který nějakým způsobem ukáže porovnání metod a jejich náročnost vzhledem k vlastnostem grafu.

S tímto souvisí také možnost spuštění aplikace bez grafického rozhraní, jehož cílem nebude ukázání a zobrazení metod, ale samotný sběr dat, ke kterému grafická nádstavba není nezbytně nutná. K efektivnějšímu základnímu sběru dat slouží jednoduchý skript, který je součástí řešení. Data budou postupně ukládána do souboru a následně jsou přístupná přes načtení ze souboru v obrazovce výsledků. Během tohoto přechodu dat budou pochopitelně data ochuzena o některá nepotřebná data, typickým příkladem může být výsledná cesta, která má smysl pouze v rámci grafu, na jehož základě vznikla.

3.3 Implementace aplikace

Implementace aplikace nebyla vždy úplně přímočará a jednoduchá. Během vývoje logicky přicházely menší změny konceptu, nové možnosti, které se zrovna nabízely k implementaci a některá úskalí. Všechna tato témata budou popsána v této podkapitole, která bude rozdělena na dvě mírně odlišné části. První část bude pojednávat o směru vývoje, jakým směrem se vývoj ubíral během času, a druhá část bude popisovat použité třídy a metody v rámci knihovny GTKmm. Použité třídy a metody ze standardní knihovny se nacházejí spíše v první části.

3.3.1 Postup vývoje

Již bylo zmíněno, že pro vypracování této práce byl použit primárně jazyk C++, avšak tato informace by chtěla mírně upřesnit. Z jazyka C++ jsem využil některé části standardní knihovny. Hlavně bych chtěl zmínit existující implementaci kontejnerů, hlavně třídu `std::vector`, která slouží jako naprostý základ mnoha funkčností napříč celou aplikací. Jsou použity velkou měrou v implementaci samotných grafů, použitých výsledků a pro řešení mnoha dalších drobnějších implementačních problémů, pro představu uvedu posloupnost výsledných stavů jako výsledků algoritmů, které jsou určeny k vyznačení v hlavním kreslícím okně. Další použitou třídou C++ je `std::string` včetně jejích metod pro lepší a snadnější práci s textovými řetězci. Okrajově jsou také využity bezpečné ukazatele implementované ve třídě `std::unique_ptr`. Při řešení jsem také využil metody dědičnosti, hlavně v rámci grafické implementace.

Následujícím krokem tedy bylo vytvoření vlastní implementace grafu, která již byla popsána ve výše zmíněné části. Nevýhodou této implementace grafu může být fakt, že pro nalezení potřebné hrany nebo vrcholu se používá jednoduchá základní struktura typu `std::vector`, která se prochází od prvního prvku k poslednímu a porovnává se s hledanými entitami. Je možné, že s využitím jiného náhledu na problematiku by mohlo být dosaženo nižší časové náročnosti. Graf také využívá pomocnou podtřídu implementující hrany, které využívá ve své implementaci. Třída implementující graf obsahuje následující metody:

- AddVertex/RemoveVertex - Přidání/odebrání vrcholu
- AddEdge/RemoveEdge - Přidání/odebrání hrany
- GetEdges - Vrátí hrany pro určitý vrchol
- GetALLEdges - Vrátí seznam všech existujících hran v grafu

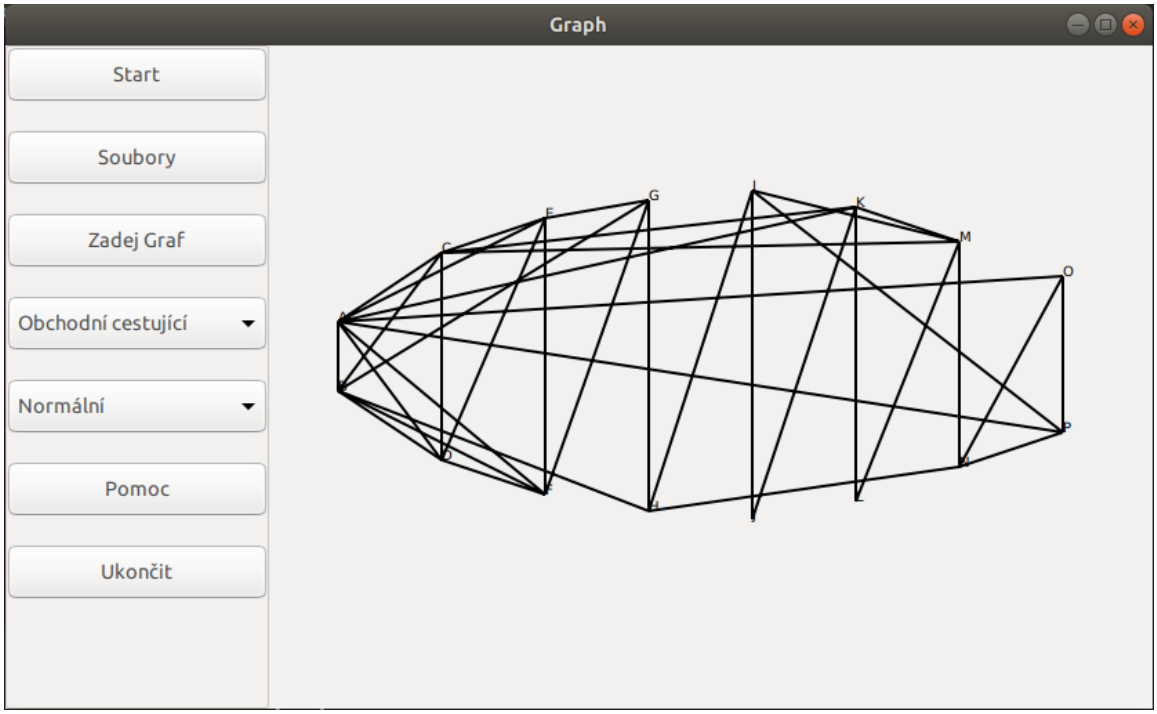
- GetAllVertex - Vrátí seznam všech vrcholů v grafu
- Initialize - Slouží k zadání/změně počátečního a koncového vrcholu
- GetStart/GetEnd - Vrácení počátečního/koncového vrcholu
- Clear - Odstraní veškterou implementaci grafu
- Empty - Kontrola, zda je graf prázdný

Po implementaci a základním otestování správy grafů následovala další část, kterou bylo vytvoření alespoň základních funkčních implementací algoritmů. Jako první zde byl pokus o vývoj Problému Čínského Listonoše, jinak také známého jako "route inspection problem", ale bohužel se jednalo také o slepou vývojovou větev, protože jsem nenalezl možnost, jak by na tento problém mohlo být využito dynamické programování. Následujím algoritmem, který byl implementován byla nejkratší cesta, následována problémem obchodního cestujícího. Oba zmíněné grafy využívají graf popsaný v 2.2.1. Algoritmy jsou situovány v samostatné třídě `Algorithms`, kde mají podobu následujících statických metod [2]:

- ShortestPath - nedynamická verze nejkratší cesty
- DynamicShortestPath - dynamická verze nejkratší cesty
- Salesman - nedynamická verze obchodního cestujícího
- HeldKarp - dynamická verze obchodního cestujícího (Held-Karpův algoritmus)
- Markov - nedynamická verze dekódování sekvence
- Viterbi - dynamická verze dekódování sekvence (Viterbiho algoritmus)

První grafická verze byla velmi podobná již aktuální verzi. Jednou zásadní změnou byl jiný algoritmus pro vykreslování samotného grafu. Vstupní podmínkou je, co možná nějvětší přesnost zobrazení ve smyslu nemožnosti překrytí hrany jinou hranou, což znamená, že se nesmí stát, aby byly 3 (nebo více) samostatné vrcholy postaveny v řadě. Představa původní verze bylo uspořádání vrcholů do dvou obloukových řad, které se nápadně podobaly tvaru elipsy. Od této představy však bylo ustoupeno z důvodu náročnější implementace, která však znamenala nulový kladný efekt oproti vykreslení v kruhu. Původní verze je zobrazena na obrázku 3.1. Další spíše minoritní změnou bylo pozdější zvýraznění ovládacích prvků aplikace.

[2] Dekódování sekvence bylo přidáno později, ale pro úplnost zmiňuji



Obrázek 3.1: Prvotní podoba aplikace

Nakonec bylo zvoleno dynamické vykreslení vrcholů do kruhu, což je i aktuální stav, který je zobrazen na obrázku 3.2. K dynamickému rozdělení používám vydělení 360 stupňů počtem vrcholů. Pomocí této hodnoty zjistím, o jaký úhel má být následující vrchol posunut. Vrchol je poté reprezentován trojicí hodnot. Souřadnicemi x, y a jeho jménem, které ukládám do `std::vector` a používám pro následující vykreslení hran. Souřadnice vypočítám pomocí goniometrických funkcí a poloměru, který se vypočítává z velikosti okna a je snížen o určitou hodnotu.

Dalším vývojovým milníkem bylo přidání funkčnosti ovládacích prvků aplikace a tedy vytvoření sady podoken pro tuto funkčnost. Tato podokna jsou lépe popsána i demonstrována v pozdějším textu 3.4. Během implementace těchto entit nenastaly žádné potíže hlubšího rázu, které by stály za zmínku a ani si neuvědomuji nějaké zajímavé, či kreativní řešení, které bylo využito. Jedinou výjimkou je vyřešení propojení grafické verze s objektem třídy grafu. Vzhledem k oddělení těchto dvou součástí je objekt grafu vytvořen ještě před prvotním spuštěním grafické aplikace a následně je předáván pomocí odkazů. Samotný graf je uchován tedy mimo grafickou část. Veškerá práce s grafem tedy probíhá přes jeho odkaz (respektive alias), který dostane hlavní okno jako parametr při vytvoření. Pro spouštění algoritmů a práci se soubory je implementována třída zajišťující mezivrstvu mezi grafickou implementací a samotným grafem. Tato třída má sadu následujících statických metod:

1. ExecuteGraph - spouští všechny implementované metody
2. Execute[algoritmus] - každý algoritmus má svou funkci pro spuštění (v té se měři i doba běhu algoritmu)
3. AddGraphFromFile/SaveGraphToFile - načítání a ukládání grafu ze/do souboru

4. AddMarkovGraphFromFile/SaveMarkovGraphToFile - načítání a ukládání reprezentace Markovova grafu ze/do souboru
5. SaveResults/LoadResults - načtení a uložení výsledků do souboru
6. GetMarkProb - pomocná funkce pro načtení desetinného čísla ze souboru
7. MYSTOD - pomocná funkce pro načtení desetinného čísla z objektu třídy `std::string`, nebyla použita funkce `std::stod`, z důvodu toho, že je ovlivněna lokalizací.

Primárním problémem aktuálně byly zatím neexistující metody dynamického programování a také nemožnost nějakým způsobem kontrolovat výsledky, natož je porovnávat. Výsledky algoritmu byly zatím vraceny pouze textově, což bylo krajně nedostačující. V tuto chvíli jsem si uvědomil, že by bylo dobré mít vlastní třídu jen s metodami algoritmů pro jejich lepší správu a případné změny, což jsem udělal a začal implementaci dynamických metod. Přes mnohá úskalí, mezi která například patří vlastní programátorské chyby, jsem se dostal k funkčnímu výsledku, který vracel stejné výsledky (ve smyslu cest a váhy) jako nedynamické verze. Pro vrácení a práci s výsledky používám vlastní strukturu, která obsahuje všechny zajímavé údaje z hlediska porovnání nebo vypsání výsledků. Pro implementaci pole těchto entit využívám třídu `std::vector`. Tato Třída obsahuje následující položky:

1. NumEd, NumVer - počet hran/vrcholů
2. Metod - název metody, ke které se data vážou (který problém a zda dynamicky nebo ne)
3. Way - Nalezená cesta
4. Height - klasický graf: výsledná váha výsledku, Markov: délka sekvence
5. MarkP - Nalezená pravděpodobnost výsledku u Markova
6. Time - doba běhu (v mikrosekundách)

Následovalo dotvoření lepší správy pomocných oken, v čele s výsledky. Během tohoto vývoje jsem udělal pár změn. Jedna ze změn se zabývá lepší správou oken a jejich implementací. Předělal jsem celé řešení na jednu hlavní třídu pro všechna vedlejší okna, což s sebou nese všechny výhody takového řešení. Díky tomuto jsem následně byl schopen efektivněji využívat odkaz na hlavní okno pro správu meziokenní komunikace. Bylo také zlepšeno chování aplikace při otevření více oken stejného typu najednou. Nadále to není možné, protože hlavní okno si udržuje odkazy na okna v `std::vector`, které zde jsou uloženy pomocí třídy `std::unique_ptr`. Ta zajišťuje, že daný index, který vždy náleží určitému typu okna může v určitou chvíli nabývat pouze jeden odkaz na jedno okno. Pokud již je index obsazený, tak se automaticky uvolní. V tomto období také začaly získávat přídavná okna doprovodnou funkcionalitu.

Zásadní změnou, kterou jsem musel řešit bylo přidání pozadí pro grafickou reprezentaci Markovových skrytých stavů. Důvody, proč nebyla využita aktuální implementace, jsou popsány zde 3.1.3. Proto byla přidána další třída grafu, která tyto možnosti poskytuje. Jsou vytvořeny také 2 pomocné třídy `TransProp` pro uchování pravděpodobnosti přechodu a koncového stavu/pozorování, které je součástí `MarkVertex`, který slouží pro implementaci stavu. Třída grafu poté tyto třídy zahrnuje ve své datové části. Grafová třída obsahuje relativně širokou paletu vstupně-výstupních funkcí. Funkce, které vybočují z tohoto schématu jsou:

1. Clear - odstraní aktuální data grafu
2. Empty - kontrola prázdnosti grafu
3. IsState - kontrola, zda řetězec reprezentuje stav
4. IsObserve - kontrola, zda řetězec reprezentuje pozorování
5. EndCheck - kontrola, zda data grafu dávají funkční celek

Tato změna však znamenala relativně velké množství změn v implementaci grafické části, ať již šlo o vykreslení samotného grafu či třeba jeho změnu v samostatném okně. Mnoho těchto záležitostí vyžadovalo zdvojení implementace, podle aktuálního grafu. Dalším problémem byla změna okna, o kterou se stará samostatný systém, který je reakcí na změnu a zajistí schování a ukázání předem vybraných prvků.

Ve vykreslení grafu jsem přešel k podobnému přístupu jako ve vykreslování klasického grafu. Též se vykresluje do jisté míry dynamicky, ale přístup kruhu mi zde přišel nedostatečný, takže jsem se rozhodl stavy dát do řady a přechody mezi nimi vykreslovat jako soustavu čar, přičemž vykreslení každé čáry zvýší hladinu, ve které se daná čára nachází, aby byl pokud možno graf co nejsrozumitelnější. Příklad takové grafové podoby je na obrázku 2.1.

Podle prvotního plánu měla být přítomna ve výsledcích pouze tabulka s porovnáním výsledků, avšak toto by z hlediska celé práce bylo poněkud nedostatečné, když se má primárně jednat o porovnání doby běhu daných algoritmů a proto byla vytvořena funkcionalita, která dokáže s dostatkem dat vykreslit grafy nad daty, která jsou přítomna v tabulkách. Zobrazena je buďto tabulka nebo graf, aktuální reprezentaci je možné měnit stiskem tlačítka. Celé to funguje skrýváním a odkrýváním widgetů v rámci okna. Postupně zde přibývala také tláčítka pro lepší customizaci dat a grafu.

V poslední fázi před závěrečným testováním a hledáním chyb, byl přidán taky slider ve spodní části grafu, který by měl také pomoci v orientaci v grafu. Celkový graf mi začal v některých případech připadat málo zajímavý, tak jsem přidal tuto možnost pro změnšení zobrazovaných dat. Čas, podle kterého se dané výsledky vykresluje na osu y, se dynamicky přizpůsobí právě zobrazovaným datům. Je možné se dostat do fáze, kdy už nejsou přítomna žádná data, poté jsem přidal výpis chyby a pokud zobrazovaná data berou pouze jediný segment, tak jsem upravil rozsah hodnot naměřeného času o $\frac{1}{3}$, aby vyšší hodnota nevykreslila pouze čáru jdoucí z vrcholu osy y.

Všechna okna, včetně grafu, by se měla dynamicky překreslovat a reagovat na změnu velikosti okna. V několika případech je tato funkce dynamická pouze částečně. V některých ohledech je pozicování fixní, k tomuto kroku jsem přikročil z důvodu snazší implementace některých problémových částí. Typickým příkladem jsou okraje vykreslovací plochy nebo od ostatních objektů v rámci vykreslovací plochy. Lze to velmi dobře vidět při zvětšení velikosti okna grafové části v okně výsledků.

3.3.2 Implementace z hlediska GUI

Jak již bylo napsáno, tak pro implementaci byla využita knihovna GTKmm, která poskytuje mnoho rozličných prvků pro správu a vytváření grafických prostředí. Avšak některé prvky nejsou dostačující nebo je potřeba jejich úprava. Tímto vzniká samozřejmě obrovské množství možností, jak některé části aplikace vytvořit. Primární okno dědí z třídy `Gtk::Window`, která zajišťuje jeho vykreslování a správu objektů, které zahrnuje. K dalšímu členění v

rámci okna jsem využil objekty třídy `Gtk::Box`, které zajišťují logickou návaznost mezi prvky a jejich řazení. Obvykle je nutné dát více takových prvků do sebe. Celá levá lišta v rámci hlavního okna aplikace patří do jednoho objektu, který je poté přidaný do hlavního Boxu, který ještě zahrnuje kreslící plochu a je přidán jako celek do hlavního okna.

Pro vykreslovací okno jsem využil třídu `Gtk::DrawingArea`, která slouží primárně k vykreslování pomocných prvků, které není možné vykreslit pomocí jiných nebo by jejich implementace pomocí jiných tříd byla velmi obtížná, což je například kreslení základních obrazců. Pro ovlivnění vykreslování je nutné použít virtuální metodu, která se provede kdykoliv uzná objekt za nutné. Například při událostech, které interagují s objektem nebo při jeho překrytí a následném zobrazení. Tuto metodu je samozřejmě možné taky vyvolat programátorsky, pokud je to z nějakého důvodu potřebné. V této práci je tato metoda využita například při změně grafu, nebo při zobrazení výsledku.

Právě tato třída byla využita pro vykreslení reprezentace grafů v rámci hlavního okna. Při této implementaci jsem se potýkal také s řadou překážek. Bylo nutné implementovat dynamické pozicování textu, které se vykreslí podle umístění vrcholu vůči zbytku grafu a také bylo nutné počítat s velikostí (pixelové) daného výpisu, aby byl o tuto hodnotu nápis posunut, aby nezasahoval do samotného grafu.

Tuto třídu jsem využil také v okně výsledků pro vykreslení grafu naměřených hodnot. Tato část byla implementačně poměrně složitá a bylo nutné výsledky nejdříve vyfiltrovat, seřadit a pokud existovaly stejné entity podle kterých se graf orientoval, tak je i zprůměrovat, abych měl jen jednu hodnotu času. Využil jsem algoritmus `std::sort`, který pro porovnání využívá funkce z třídy `Clue`. Samotné zprůměrování jsem pak již implementoval sám. Následně jsou tato data předána objektu, který je mojí implementací dědící z `Gtk::DrawingArea`, kde se dynamicky zhodnotí, na jaký prostor se graf má vykreslit. Toto se dělá porovnáním maximální hodnoty na osách a celkové velikosti prostoru grafu. Podle tohoto poměru se určí body, kterým daná data odpovídají. K tomu bylo nutné ještě počítat s možností použití slideru, který mění velikost zobrazovaných dat. V rámci tohoto se mění maximální hodnoty grafu, tedy i celkový poměr. Avšak je nutné vykreslovat graf pro data, která jsou o 1 větší než je vybraný rámec, aby bylo vidět, jakým směrem dané hodnoty míří.

Celé GTK+, tedy i GTKmm, využívá pro vykreslování knihovnu Cairo, alespoň, co se týče modernějšího pojetí GTK+, starší verze využívaly možnosti dané systémem X Window. S knihovnou Cairo a jejími možnostmi jsem se také dostal do kontaktu při implementaci této práce. Typickým příkladem je vykreslovací metoda v `Gtk::DrawingArea`, která má jako parametr Cairo context, pomocí kterého se dá již reálně kreslit. Je zde velmi široká paleta funkcí, které mohou být použity pro vykreslování. Tato metoda je dostupná i v rámci ostatních Widgetů a byla v rámci aplikace využita například k označení tlačítek, ale možnosti zde jsou pochopitelně omezené, pokud chci ponechat původní styl vykreslování.

Ovládací prvky používají většinou třídu `Gtk::Button`, která je základním stavebním kamenem pro různé objekty, které mají provést nějakou funkci po stisknutí nebo při jiném typu události, který se daného objektu týká, záleží na implementaci. Tato třída je také základem pro `Gtk::RadioButton`, který je zde využit jako přepínač mezi různými možnostmi. Například pro výběr, který typ grafu má být ovlivněn v rámci okna změny grafu nebo v okně výsledků. Tyto vlastnosti pochopitelně mohou mít i další objekty, ale nejsou k tomu přímo určeny.

Posledním velmi důležitým použitým grafickým prvkem je `Gtk::TreeView`, který slouží k výpisu dat v tabulkové struktuře. Již od základu je velmi ohebný a dá se použít i k zobrazení tlačítek, která mohou nějakým způsobem modifikovat nebo používat data, ke kterým se

vážou. Tato možnost však není v této práci implementována. Tuto třídu využívám k výpisu entit hlavně v okně grafů a okně výsledků. K této třídě je navázáno několik dalších, které fungují jako celek. Tato třída však funguje pouze vertikálně, tedy, že další řádky se dávají pod sebe, což v některých případech není ideální a dovedu si představit, že bych využil řazení prvků vedle sebe. K této třídě se váže i využití pomocné třídy `Gtk::ScrolledWindow`. Tato třída zobrazuje pouze část objektů, která zahrnuje. Tento pohled se ovlivňuje pomocí bočních posuvníků.

Nesmírně důležitou a zatím opomíjenou částí aplikace je také samotné měření časové náročnosti algoritmů. V mém provedení se jedná o celkem jednoduchou třídu zahrnující dva časové údaje, které se naplní pomocí metod startu a ukončení. Následně již stačí pouze získat jejich rozdíl a výsledkem je čas, který algoritmus potřeboval k provedení. Je samozřejmě nutné, aby všechny algoritmy měly stejné podmínky a metoda startu se volala přímo před zavoláním algoritmu a metoda konce přímo za koncem. Pokud metoda vyžaduje nějakou předpřípravu dat, tak to zahrnuji do tohoto intervalu, aby byla zajištěna, co největší možná rovnost vstupních podmínek všech použitých algoritmů. Využívám měření času pomocí standardní knihovny chrono. Hledal jsem i jiná řešení, avšak nenalezl jsem přesnější měření, které by bylo multiplatformní. Bohužel toto řešení je relativně nepřesné a v ideálním případě by vyžadovalo spouštění bez grafické nádstavby a za použití, co nejmenšího množství aplikací pracujících na pozadí. Respektive měření je závislé na aktuálním zatížení systému, které z logiky věci není konstantní. Snahou řešení tohoto problému jsou opakované testy, které se průměrují.

Sluší se zmínit, že během grafického vývoje aplikace jsem aktivně čerpal a využíval informace hlavně z oficiálních materiálů a tutoriálů, které jsou volně dostupné zde [10]. Je tedy možné a pravděpodobné, že některé části si mohou být relativně podobné a hlavně u mé implementace třídy `Gtk::TreeView` by mohla být podobnost velmi blízká, což je však následek toho, že se jedná o oficiální a dostupné materiály, které slouží právě k účelu výuky, jak správně a efektivně použít třídy dostupné v rámci této knihovny. Vzhledem k množství využití na mnoha místech jsem se rozhodl to zmínit přímo v práci a také v hlavičce zdrojových souborů.

3.3.3 Součásti aplikace

Tato část je pouze doplňková a slouží jako sumarizace všech funkčností aplikace. Tedy velká její čast bude pouze opakování předešlé sekce. Aplikace se skládá z možnosti spouštění, jak v textovém (pouze okrajově), tak grafickém režimu. V grafickém režimu se skládá z hlavního okna, kde v levo jsou situovány ovládací prvky. Zbytek okna zabírá vykreslovací plocha, která vykresluje grafy a také má možnost vyznačení výsledku v rámci grafu. Aplikace obsahuje také několik podpůrných oken, které jsou obvykle odezvou aplikace na určitou akci a jsou blíže popsána v následující podkapitole.

Některá podpůrná okna aplikace mají spodní lištu, která slouží ke komunikaci s uživatelem, tedy aby věděl, zda se například některá akce nezdařila nebo naopak se provedla bez obtíží. Nejvýznamnějším vedlejším oknem je okno s výsledky. Toto okno porovnává získaná data a je schopno vyprodukovat vlastní graf, který se dynamicky vykreslí podle velikosti prostoru, který mu je dán. Mezi další okna patří okno s nápovědou, okno pro načtení a uložení grafu do souboru a nakonec okno, které umožňuje měnit graf.

Součástí aplikace jsou samozřejmě i řešení skrytá před uživatelem mezi něž patří vlastní implementace grafů a jejich entit nebo způsob jejich volání a získávání jejich výsledků.

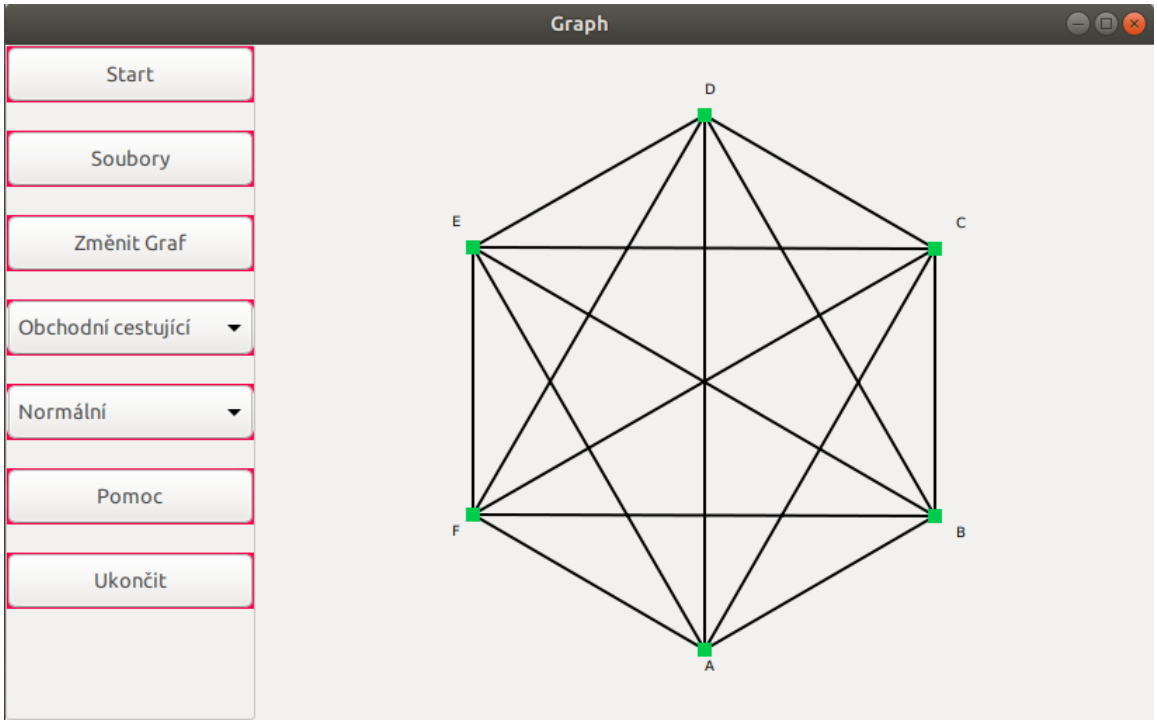
Ve zdrojovém kódu je snaha členit samostatné součásti do logických celků a podle nich je ukládat do souborů.

3.4 Ovládání a popis aplikace

Podrobnější možnosti spuštění aplikace jsou popsány v A. Aplikace je ovládána primárně přes grafické rozhraní. V hlavním okně k ovládání slouží levý panel, jehož ovládací prvky mají následující funkcionalitu:

- Start - Slouží k provedení vybraných metod nad daným grafem, popřípadě grafy. Po provedení otevírá okno s výsledky. Je možné, že otevření okna bude chvíli trvat díky době trvání provádění algoritmů. Po tuto dobu by nad celou aplikací měl být jiný typ cursoru.
- Soubory - Otevře pomocné okno s možnosti načtení, či uložení grafu z/do souboru.
- Změnit Graf - Otevře pomocné okno s možnosti změny načtených grafů.
- VýběrMetody - Slouží k výběru metod, které se mají provést. Varianty jsou "Nejkratší cesta", "Obchodní cestující", "Dekódování sekvence"a "Všechny". Ovlivňuje i vykreslovaný typ grafu.
- VýběrVarianty - Slouží k výběru typu metody, kterou se má daný problém řešit. Možnosti jsou "Normální", "Dynamická"a "Oboje". Speciální případ je, když ve VýběruMetody je vybráno všechno, poté je tento výběr neaktivní.
- Pomoc - Slouží k zobrazení okna s nápovědou.
- Uzavři - Zavře aplikaci.

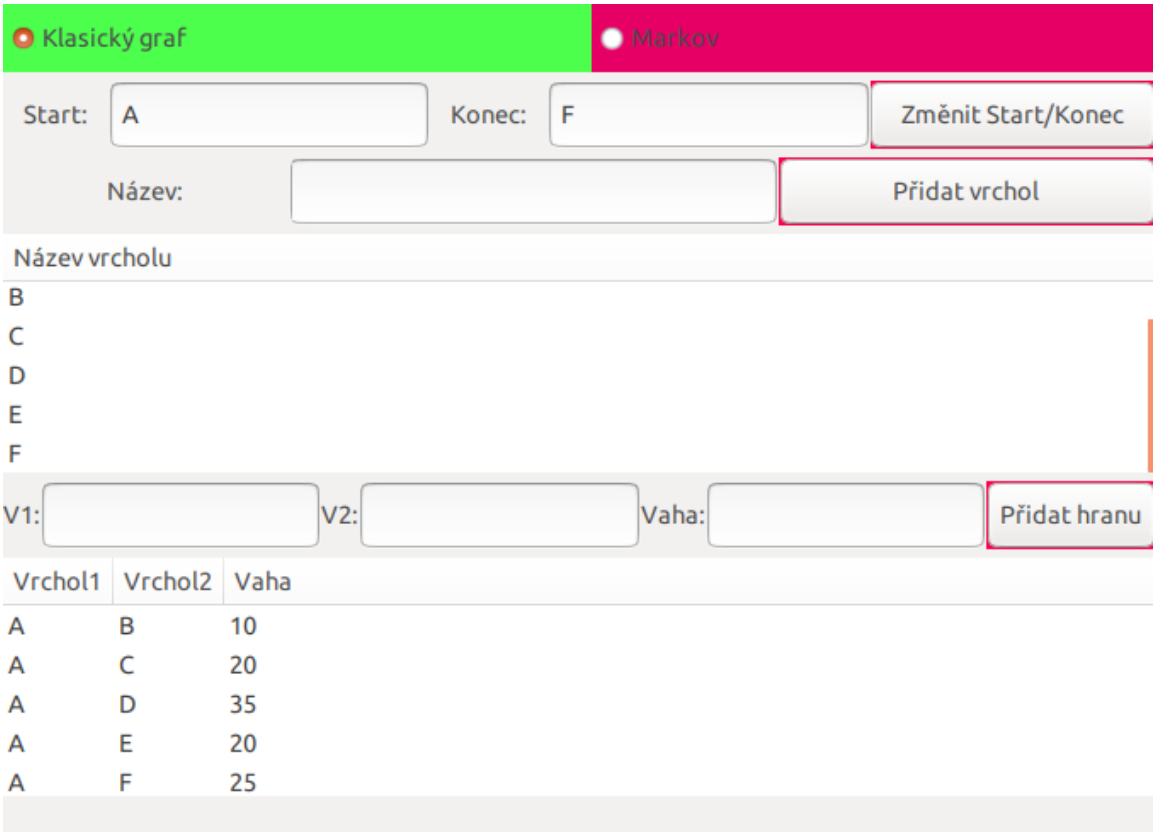
Všechna okna mohou být otevřena maximálně jedenkrát v určitý čas. Po stisku tlačítka na otevření okna se minulé okno zruší a otevře nové. Zbytek plochy hlavního okna není z pohledu ovládání zajímavý a slouží k vykreslení aktuálně načteného grafu. Plocha je schopna vykreslovat oba typy grafů, které jsou implementovány. Vykresluje se zde také případné řešení úloh po aktivování v okně výsledků. Podobu aplikace více přiblíží obrázek 3.2.



Obrázek 3.2: Aktuální podoba aplikace v prostředí GNOME

Okno souborů je poměrně jednoduché. Horní výběr slouží k označení typu grafu, na který se má daná akce vztahovat. Níže je okno, do kterého lze zadat název souboru, ve kterém je uložen popis grafu, či do kterého má být uložen. V nejnižší části okna jsou dvě tlačítka, která slouží pro provedení akce uložení/načtení.

Okno změny grafů patří k již složitějším oknům. V horní části je situován výběr s jakým typem grafu se bude v rámci tohoto okna pracovat. V rámci celého okna jsou vypsány entity grafu, takže okno může sloužit také jako jistá kontrola, z čeho je graf složen. Okno je mírně odlišné, pokud je vybrán klasický graf nebo HMM. První popis se bude vztahovat směrem ke klasickým grafům. Vybranou možnost lze poznat podle zelené barvy pozadí výběru. Okno je na obrázku 3.3.



Obrázek 3.3: Podoba okna pro změnu grafů

U klasického grafu jsou přítomny 2 zadávací okna, která slouží pro zadání i změnu počátečního a koncového vrcholu. Tato akce se provede po kliknutí na tlačítko situované v pravo s popisem "Změnit Start/Konec". O úroveň níže je okno s možností přidáním nového vrcholu do grafu. Stiskem tlačítka "Přidat vrchol"se vrchol přidá do grafu. Výpis níže vypisuje všechny vrcholy obsažené v grafu. Na další úrovni je možnost přidání hrany. V1 a V2 slouží k zadání bodů, které hrana spojuje a váha pro zadání hodnoty hrany.Tlačítko přidat hranu ji přidá do grafu. Posledním výpisem je výpis hran skládajících se ze tří entit: dvou vrcholů a váhy. Dvojklikem na některou položku ve výpisu, tak danou položku odstraníte. U vrcholu se odstraní také všechny hrany, ve kterých se vyskytuje. Pokud je nutná změna hrany, tak zde není přítomna a je nutné danou hranu odstranit a znovu zadat.

Při výběru HMM mají ovládací prvky podobné efekty jako u klasického grafu, avšak jsou zde odlišnosti, které si zaslouží popis. Místo změny startovního a koncového bodu je zde možnost uložení sekvence. Sekvence je posloupnost pozorování oddělena čárkami (tedy znak ’,’). Následována možností přidání vrcholu, které se chová identicky jako u klasického grafu. Je zde přítomna i možnost přidat pozorování, která slouží k přidání nového typu možného pozorování. Výpis stavů je stejný jako výpis stavů u klasického grafu, ale je zde i výpis pozorování, který následuje za výpisem stavů. Tyto dvě položky jsou odděleny speciálním řádkem "———". Přidání hrany je stejné, jen místo váhy je zde pravděpodobnost určitého přechodu/pozorování. Desetinné číslo pravděpodobnost zadávajte pouze s desetinnou tečkou. Následuje výpis přechodů, který se dělí na 3 samostatné části. První jsou pravděpodobnost toho, že se systém na začátku nachází v daném stavu. Druhé jsou pře-

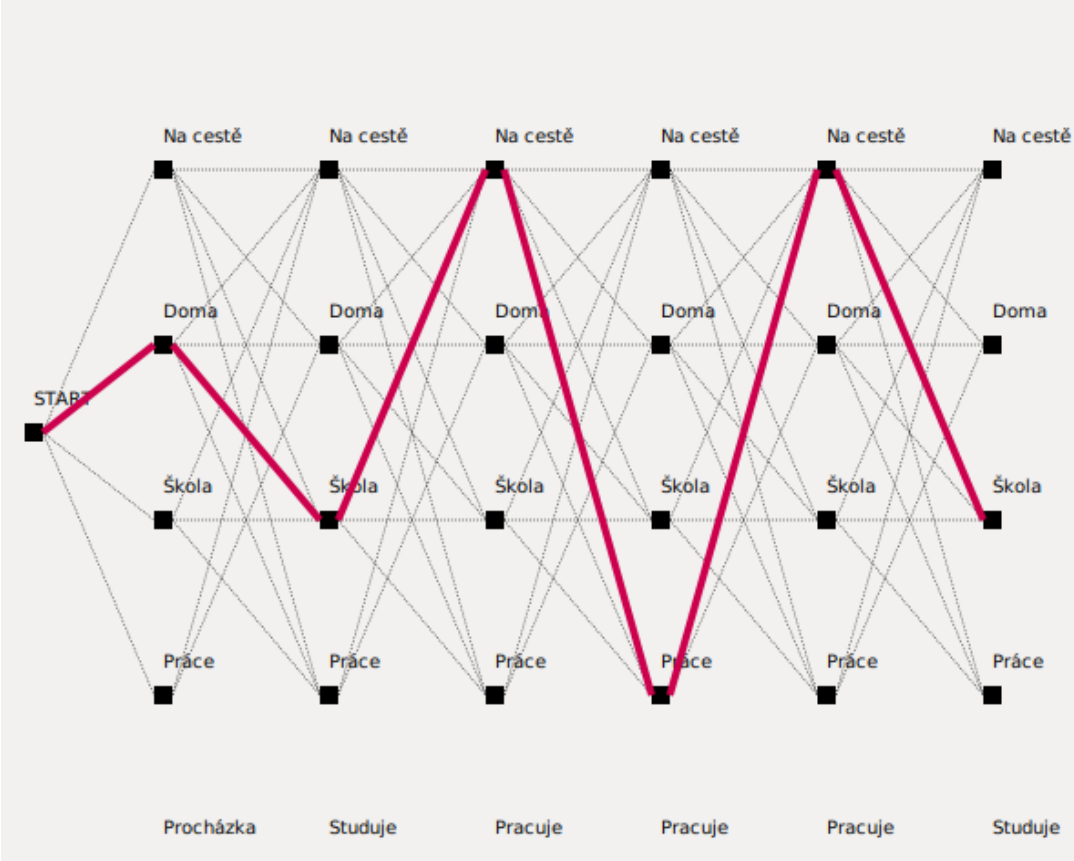
chody typu Stav -> Stav a jako poslední jsou přechody Stav -> Pozorování, což znamená možnost daného pozorování v daném stavu.

Automatická kontrola takového grafu není přítomna. Jediné kontroly jsou na přidání pozorování/stavů, které již existují a nelze zadat pravděpodobnost výšší než 1. Kontrola také zajišťuje nemožnost zadání pozorování do V1. Pro ovlivnění pravděpodobnosti stavu na začátku je nutné do textového pole V1 vyplnit "_START". Existuje také kontrola na nemožnost zadání vrcholu s tímto názvem. Neexistuje kontrola na duplikaci hran. Graf je možno také zkontrolovat pomocí tlačítka Zkontroluj graf. Ve spodní části okna se vypíše, zda je graf v pořádku nebo ne, případně důvod chyby.

Soubory, které zadávají graf do aplikace mají velmi striktní podobu, kterou je potřeba dodržovat pro předejití možného špatného načtení. V načítacích funkcích jsou přítomny kontroly, ale ty nejsou všespásné a oddělení položek probíhá pomocí tabulátoru a nového řádku na předem definovaných místech. Jiné způsoby oddělování (například vícenásobná mezera) jsou neakceptovatelné a může dojít k nedefinovanému chování aplikace. Soubory jsou normálně čitelné. Mělo by se jednat o efektivnější možnost tvorby a změny grafů. Podrobnější informace naleznete v B.

Nejsložitějším oknem je okno pro výpis výsledků. V jeho horní části je výběr podle kterých entit je graf vykreslen. Přítomny jsou počty vrcholů, počty hran a stupeň vrcholů. V prostřední části je oblast, která může měnit vykreslování podle aktuálního stavu, proto její popis nechám na závěr odstavce. Následována je možností "Změň zobrazení", která ovlivňuje aktuální stav. Na stejné úrovni jsou ještě tlačítka pro výběr metody, která má sloužit pro aktuální porovnávání. O úroveň níže jsou 2 tlačítka pro načtení, či uložení výsledků do/z souboru. Název souboru je specifikován podle zadání uživatele v zadávacím okénku na pravé straně.

Prostřední část okna se může zobrazovat buď jako tabulka nebo vykreslení grafu. Tabulka reprezentuje dobu výpočtu různýchalgoritmů pro různé počty vrcholů/hran/stupňů vrcholů. Entita podle, které jsou výsledky řazeny se mění pomocí horního výběru. Je implementována také možnost data v tabulce zprůměrovat, kdy se tabulka seřadí podle dané entitiy(například počtu vrcholů). Což znamená, že všechny přítomné výsledky, které mají společné rysy (například počet vrcholů = 4), ale rozdílné algoritmy(dynamický/nedynamický) se vypíší v rámci jednoho řádku tabulky. Při dvojím poklikání na řádek, který se váže k aktuálnímu grafu je v zobrazovací části hlavní aplikace vykreslen výsledek, ke kterému daná metoda došla. Výsledek může být zobrazen například následovně 3.4 .Při volbě druhé možnosti jsou hodnoty z tabulky dány do okna v podobě grafu. Graf je také ovlivněn, podle kterých entit se mají výsledky skládat. Graf se dynamicky zobrazí v rámci zobrazovací plochy. Pro lepší možnost selekce výsledků slouží v tomto stavu spodní "slider", který slouží k omezení vykreslovaných dat. Maximální hodnota času se dynamicky upraví podle maximální zobrazované hodnoty. Maximální hodnota pro porovnání podle různých entit může přinést různé maximální časy díky získávání průměru z různých dat.



Obrázek 3.4: Zobrazení výsledku pro Markovův graf

3.5 Možnost následného využití

Při samotném návrhu i implementaci byla snaha o možnosti případného budoucího využití, ať akademického, či jiného směru. Tento záměr je podpořen několika návrhovými řešeními a samozřejmě oddělením základních částí aplikace od sebe. Základním oddělením je oddělení samotných grafů od implemetace algoritmů. Samotná grafická část je také oddělena od zbývajících dvou entit a pouze volá metody, které implementují algoritmy, a interpretuje jejich výsledky.

Rozhodně je možné aplikaci dále rozvinout a nějakým způsobem vylepšit. Mezi reálně představitelné směry následného vývoje patří rozšíření algoritmické základny, připravení dalšího typu grafu, ať již odvozeného nebo nově vyvinutého s novou sadou jeho metod. Bylo by možné přidat další parametry, podle kterých by bylo možné dané metody zkoumat (například paměť, respektive paměťovou náročnost), ale bylo by nutné provést ještě jistý průzkum, zda se jedná o jednoduše realizovatelný problém. Největší překážkou, kterou si dovedu představit je možná nesourodost tohoto požadavku s grafickým výstupem aplikace, ale teoreticky by bylo možné vytvořit systém, který metodu spustí třeba na jiném vlákně a poté by data o použité paměti mohla být dosažitelná i v rámci hlavního programu.

Rozhodně by bylo možné také vylepšit textový režim, který by se mohl stát plnohodnotným zastupitelem, pokud by bylo nutné program spouštět bez grafického rozhraní. Poté je však otázkou, zda by výsledný program k něčemu vedl. V tomto směru by rozhodně bylo zajímavé zapojit do programu výstup z programu GNUplot, který dokáže kreslit velmi dobré

grafy. Velmi zajímavé by mohlo být i využití `std::thread` pro zajištění odezvy programu při dlouhých dobách běhů algoritmů.

Za zvážení by také stálo použití konfiguračních souborů na některé základní věci jako je například základní velikost okna, použité barevné spektrum a možná i nějaké záležitosti, které jsou aktuálně pevně dané, ale pro uživatele by mohlo být prospěšné mít možnost tyto věci ovlivnit. Typickým příkladem by mohlo být chování grafu výsledků, zda začínat v bodu [0,0] nebo v jiném.

Kapitola 4

Porovnání algoritmů

Všechny algoritmy jsou porovnávány podle doby běhu, což může být určitě v jistém smyslu limitující, avšak důležité je ukázat výhody nebo i možné nevýhody optimalizační metody dynamického programování a její vlastnosti a pro tento účel je to dostačující. Všechny výsledné časy jsou vypsány v mikrosekundách.

Data, která zde budou použita, byla vždy sbíráná s co možná nejmenším množstvím aplikací běžících na pozadí, aby bylo odstíněn výkonnostní propad zajištěný jinými aplikacemi. V tomto případě je poněkud nešťastné využití GUI, které možnosti omezuje, hlavně, co se týče porovnání například paměťové stránky. Avšak je pravděpodobné, že i ve směru spotřeby paměti je dynamické programování lepší díky odbourání časté rekurze, ale bude záležet případ od případu.

Otázkou zůstává podle jaké grafové entity dané časy porovnávat. Zda podle počtu vrcholů,počtu hran nebo nějaké jiné vlastnosti. V mé implementaci se ukázal jako nejlepší porovnávací prvek počet hran, avšak to může souviset se samotnou implementací základní grafové struktury (a také s vybranými grafy), která velkou mírou výsledky bude ovlivňovat. Průměrný stupeň vrcholů by měl patřit k lepším porovnávacím znakům, avšak jako nejlepší mi přišla kombinace obou pohledů porovnávání. Porovnávání pouze podle počtu vrcholů nezohledňovalo počet hran, což se na výsledných grafech výrazně projevilo. Tímto nemá být řečeno, že by porovnávání například podle počtu vrcholů nedávalo smysl, dává (při nejmenším v algoritmech, kde je přítomno v časové složitosti), ale porovnávání podle jiných vlastností s sebou přináší přesnější a lepší výsledky (z hlediska grafu). Důležitou vlastností je, že porovnávání podle všech těchto kritérií má shodný výsledek, tedy že dynamické programování se kladně podepsalo na době trvání algoritmů.

O tom, zda použít průměrný stupeň vrcholů nebo počet hran také rozhoduje počet samostatných výsledků (tedy počet zobrazených výsledků po zprůměrování podle podobnosti v dané entitě). Při menších počtech byly oba grafy srovnatelné, možná graf zohledňující průměrný stupeň vykazoval o trochu lepší výsledky. Při vyšších počtech se výsledek obrátil spíše v prospěch počtu hran, alespoň, co se testovaných grafů týče.

Bohužel porovnání může ovlivnit také mnoho jiných věcí, které by šly ovlivnit již mnohem složitěji. Jednou z těchto věcí byla technologie přetaktování jádra, nad kterou však nemám žádnou kontrolu. Další může být (v případě mého stroje) technologie "falešných jader"(tuším Hyperthreading), kdy je možné, že toto jádro dostane můj proces.

Ve všech algoritmech se při základním použití setkalo dynamické programování s kladnou odezvou a výsledné časy velmi významně snižovalo. Avšak velmi záleželo na typu grafu, který byl aktuálně měřen. Existují typy grafu, které byly s dynamickou metodou pomalejší než s klasickou, tento případ bude více rozveden v 4.2.

Důležitým faktem také je, že tyto výsledky se vztahují k implementaci algoritmů v rámci této práce, takže porovnávané algoritmy jsou ty, které jsou implementovány zde a mohou se jistým způsobem lišit od výsledků, které mohou být udávané jinými zdroji. Z hlediska času je samozřejmě také velmi důležité, na jak výkonném stroji je daný algoritmus spouštěn, takže poté časové výsledky z jednoho počítače se nemusí nikterak shodovat s výsledky z jiného počítače.

4.1 Nejkratší cesta

Nejkratší cesta je algoritmem, kde se implementace dynamického programování setkala s největším úspěchem. Časy dynamického algoritmu jsou několikanásobně nižší než časy, které potřeboval klasický algoritmus. Za zmíňku také stojí, že implementace využívající dynamické programování byla rychlejší ve všech testovaných možnostech než odpovídající klasická metoda.

Počet vrcholů/hran	4/6	5/10	6/15	7/21	8/28
Klasická (μs)	125	642	3480	22860	180345
Dynamická (μs)	54	84	204	286	433

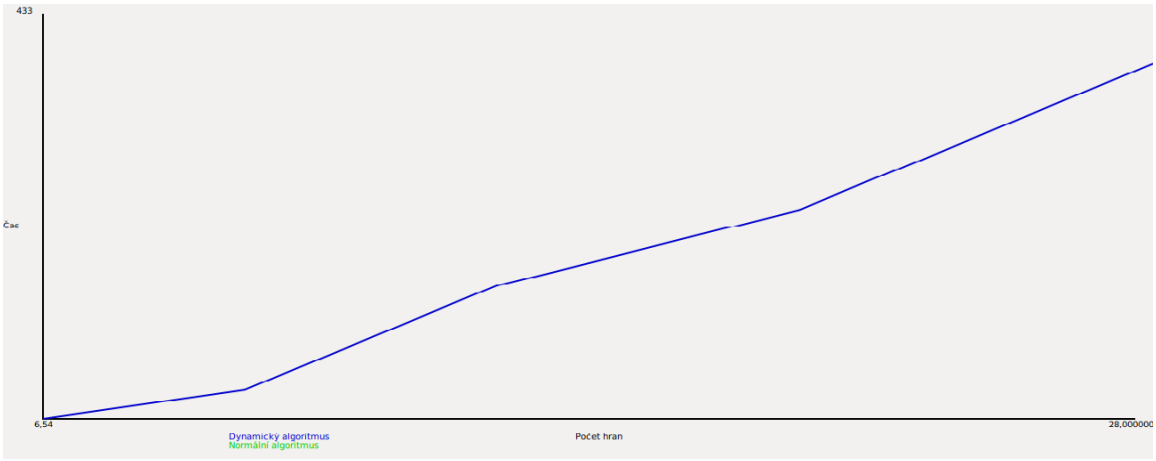
Tabulka 4.1: Porovnání klasické a dynamické metody u nejkratší cesty

Tabulka 4.1 [1] ukazuje porovnání dynamické a klasické cesty implementace algoritmu. Už z prvního pohledu lze říci, že dynamická metoda exceluje a s přibývající náročností se rozdíl zvětšuje. U dynamické metody jsou data velmi blízko funkce $t = 2 * m * n$, což řádově odpovídá časové náročnosti uvedené v 2.3.1. Naivní metoda má v tomto případě trend popsaný $t = k * n!$.

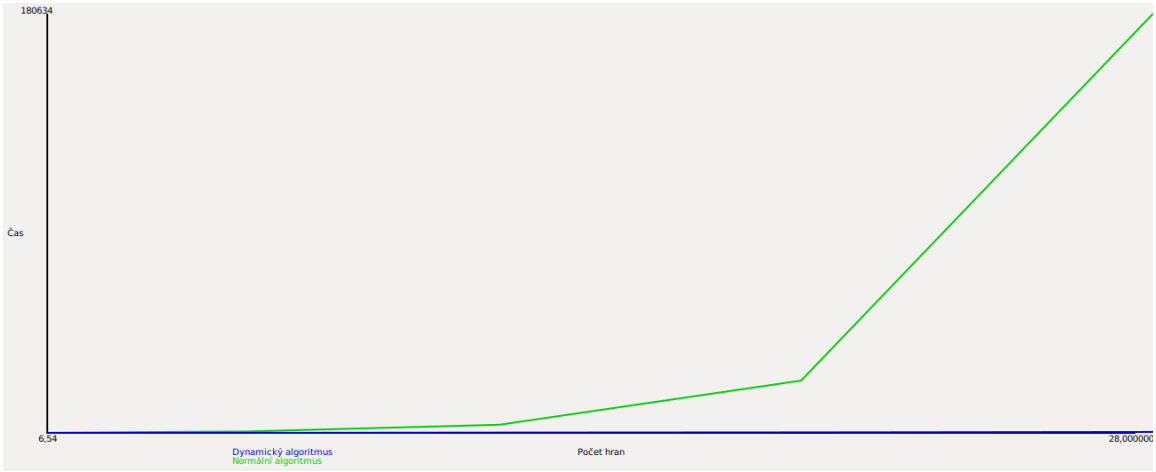
Graf 4.2, který má jiná (mnohem obsáhlejší) vstupní data, ukazuje velmi podobné výsledky. Takže lze říci, že dynamická metoda je podstatně rychlejší než klasická metoda využívající rekurzi a dá se dosáhnout výsledku v rámci řádově nižších časových úseků než za použití klasické metody. Tím lze tuto metodu v přijatelném čase aplikovat i na mnohem větší a složitější grafy. Toto zrychlení je možné díky tomu, že pokud algoritmus ví, že daného vrcholu mohu dosáhnout s lepším ohodnocením, tak není důvod nadále počítat další možnosti.

Grafy 4.1 a 4.2 ukazují průběh porovnání doby běhu nejdříve pouze pro dynamickou metodu a následně pro obě metody. Samostatný průběh pro nedynamickou metodu není přítomen, protože to je již vyjádřeno společným grafem. Oba grafy jsou vytvořeny na základě údajů z tabulky 4.1.

[1] Všechny tabulky jsou porovnány podle počtu vrcholů, protože se podle mne jedná a nejreprezentativnější porovnání vzhledem k počtu zobrazených dat. Vždy se jedná o úplné grafy, takže zbývající 2 hodnoty se dají odvodit



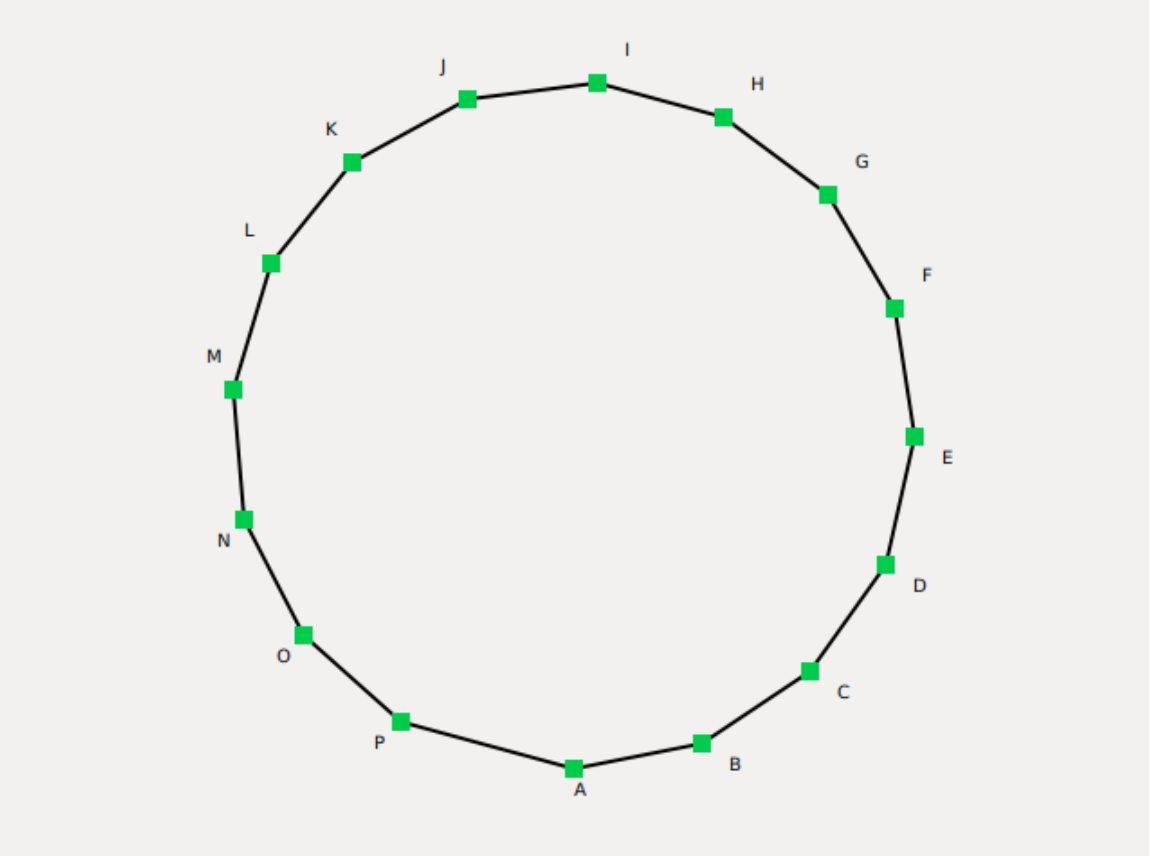
Obrázek 4.1: Závislost výsledků časů běhu u dynamické metody problému nejkratší cesty vzhledem k počtu hran.



Obrázek 4.2: Závislost výsledků časů běhu u obou metod problému nejkratší cesty vzhledem k počtu hran.

4.2 Obchodní cestující

Použití dynamické metody však selhává v jednoduchých grafech, kdy každý vrchol má pouze 2 hrany a tímto způsobem celý graf vytváří kruhovou strukturu, jak je vidět na obrázku 4.3.



Obrázek 4.3: Kruhový graf

Data, které patří typu grafů, který je předestřen na obrázku jsou v tabulce 4.2. Je pravděpodobné, že je to zapřičiněno tím, že metoda pro svůj běh potřebuje relativně velkou počáteční paměťovou strukturu jejíž prvotní inicializace je celkem náročnou součástí algoritmu. Postupným přidáváním hran do grafu se však stává mnohem výhodnější alternativou k neoptimalizované metodě průchodu.

Počet vrcholů/hran	16/16	15/15	14/14	13/13	12/12	11/11	10/10
Klasická (μs)	4953	4353	3824	3270	2871	2494	2029
Dynamická (μs)	23415	17984	13632	10518	7215	5267	3591

Tabulka 4.2: Porovnání klasické a dynamické metody u obchodního cestujícího ve speciálním typu grafu.

Metoda obchodního cestujícího vykazuje také zlepšení po použití dynamického programování ve většině použitých grafů, jak ukazuje tabulka 4.3.

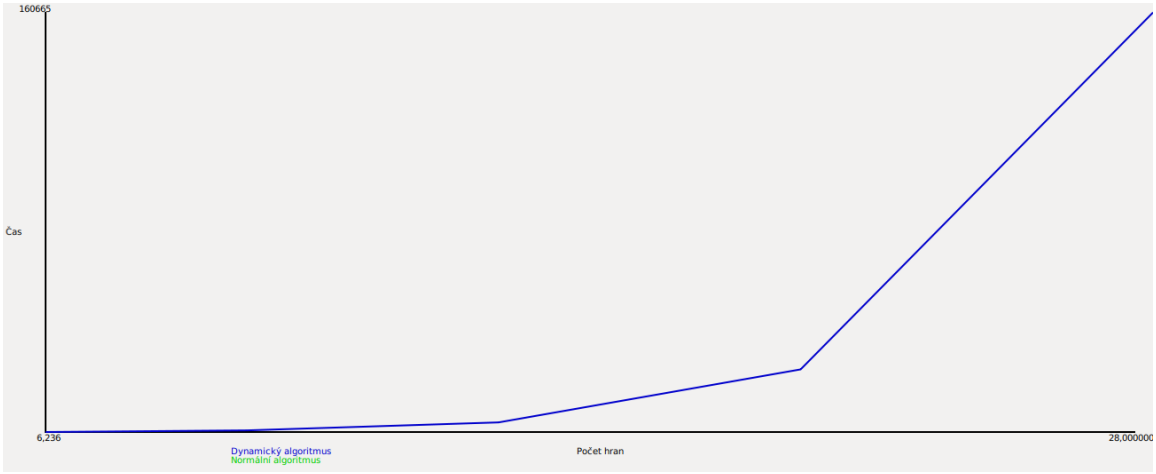
Počet vrcholů/hran	4/6	5/10	6/15	7/21	8/28
Klasická (μs)	686	8152	80855	963526	8700821
Dynamická (μs)	236	863	4033	24189	160665

Tabulka 4.3: Porovnání klasické a dynamické metody u obchodního cestujícího podle počtu vrcholů a hran

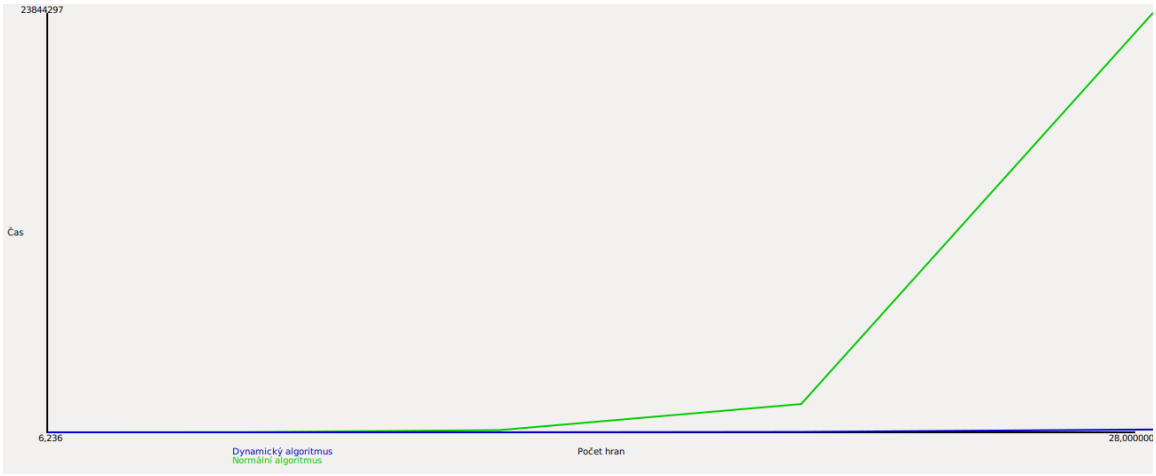
Jak ukazují data z tabulky 4.3, tak reálný efekt dynamického řešení u této metody sice zajišťuje zlepšení, ale její efekt není natolik významný jako u nejkratší cesty. Je to zapříčiněno tím, že algoritmus i tak musí projít většinu možností, které existují a porovnávání cest již během samotného výpočtu sice uspoří některé cesty, avšak zdaleka nedosahuje posunu, který byl pozorován u nejkratší cesty. Dá se říci, že metoda zajistí, že v obstojném čase se dá obsáhnout větší graf, avšak stále jde o velmi náročnou metodu a na velké a složité grafy jí také brzy začne docházet dech.

Pohledem na data lze říci, že nárůst časové náročnosti dynamické metody se velmi podobá nárůstu klasické metody u problému nejkratší cesty. Podle zdroje má mít dynamické řešení obchodního cestujícího exponenciální časovou složitost a neoptimální verze má mít faktoriálovou časovou složitost [5]. Výše zmíněná data však tomuto jevu neodpovídají. Dynamická metoda má bohužel velmi zřetelné jevy faktoriálu. Standardní metoda rozhodně reaguje na vstupní data více než faktoriálově (alespoň, co se testovaného úseku grafu týče). Důvodů může být více, ale předně bych se zaměřil na kontrolu, ve chvílích, kdy algoritmus došel do konce v případě klasické metody a její zefektivnění. V případě dynamické metody je možné, že používám více cyklů než je reálně potřebné. Do jisté míry to lze dávat za vinu i základnímu grafu, který pravděpodobně šel udělat optimálnějším způsobem.

Grafy 4.4 a 4.5 ukazují stejné případy jako tomu bylo u nejkratší cesty. Jedná se o stejná data jako v případě tabulky 4.3.



Obrázek 4.4: Závislost výsledků časů běhu u dynamické metody problému obchodního cestujícího vzhledem k počtu hran.



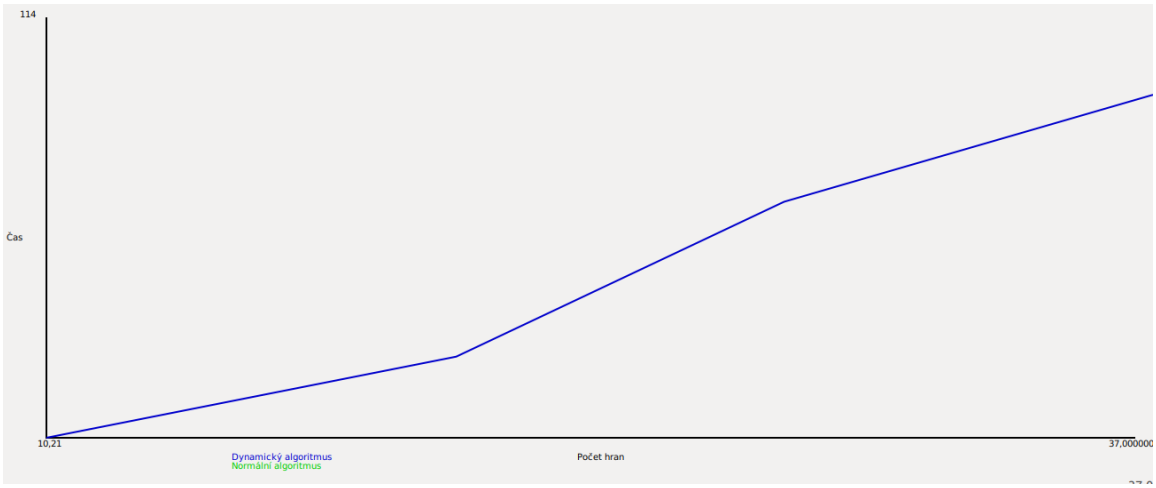
Obrázek 4.5: Závislost výsledků časů běhu u obou metod problému obchodního cestujícího vzhledem k počtu hran.

4.3 Dekódování sekvence

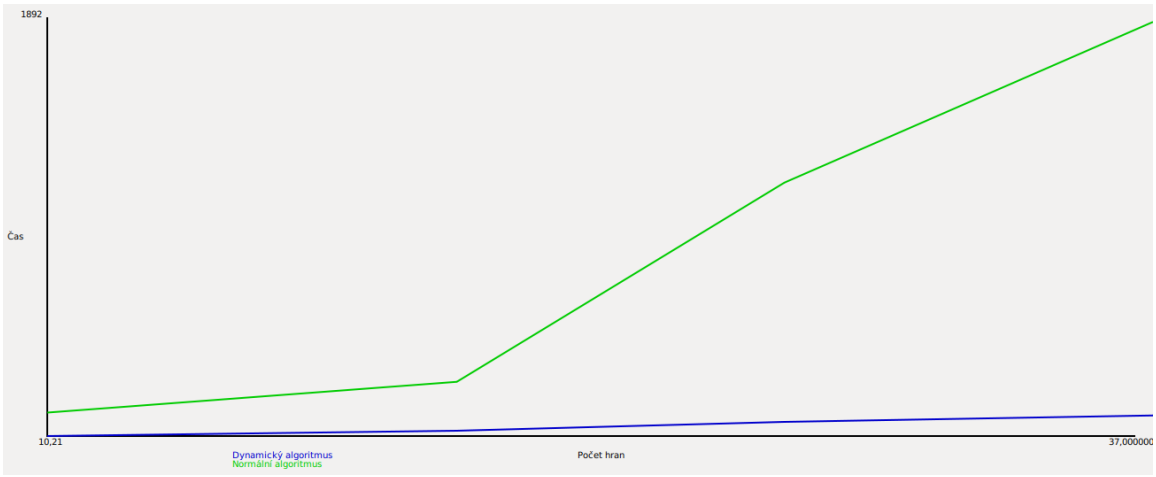
U metody dekódování sekvence došlo také k výraznému posunu ve zlepšení časů průběhu. V nejjednodušších grafech byly délky průběhu algoritmů velmi blízko sobě, ale po přidání, ať již vrcholů, nebo délky pozorované sekvence se pomyslné nůžky mezi srovnávanými časy zvětšovaly ve prospěch dynamického pojetí problému. Vypozorovat zde správný parametr je poměrně složité, protože zde přichází v úvahu více parametrů, které mohou daný výsledek ovlivnit, což nalezení optimální kombinace parametrů poměrně ztěžuje.

Je zřejmé, že porovnání podle počtu stavů nebo počtu přechodů mezi stavy (i počtem možných pozorování ze stavů) je poměrně nepřesné. Jako řešení této situace se nabízí využití již zavedených vlastností se zohledněním délky sekvence. Zohlednění spočívá ve vydělení času velikostí hledané sekvence. Toto zohlednění je velmi příznivé pro Viterbiho algoritmus, protože závislost na délce sekvence je u něj čistě lineární (nebo blízko ní), na rozdíl od nedynamického algoritmu, kterého závislost na délce sekvence je exponenciální.

Pro účely tohoto porovnání však byly vybrány grafy, jejíž sekvence je stejně dlouhá (6 položek). Tyto grafy byly použity, protože nás zajímá ještě jiná závislost. Obrázky 4.6 a 4.7 ukazují stejné případy jako u předcházejících metod a tabulka 4.4 jsou data, ze kterých tabulka vznikla. Zde přítomné počty hran jsou součtem všech možných hran mezi stavy, tak i hran, možných pozorování ze všech stavů.



Obrázek 4.6: Závislost výsledků časů běhu u dynamické metody problému dekódování sekvence vzhledem k počtu hran.



Obrázek 4.7: Závislost výsledků časů běhu u obou metod problému dekódování sekvence vzhledem k počtu hran.

Data z tabulky 4.4 se dají interpretovat několika způsoby. První z nich je, že ukázková data jsou relativně malá, aby bylo možnost ukázat nějaký fenomén, který by provázel tyto hodnoty. Díky tomu, že vím, že poslední zkoumaný údaj je z "neúplného grafu"(nejsou vyplněny všechny možné přechody ať již mezi stavy nebo emitováním pozorování), což logicky snižuje jeho náročnost.

Počet stavů/hran	2/10	3/20	4/28	5/37
Klasická (μs)	127	266	1157	1931
Dynamická (μs)	21	43	85	114

Tabulka 4.4: Vliv počtu stavů a délky sekvence na výsledný čas.

Všechny časy jsou již s vydělenou délkou sekvence. Časová složitost dynamické metody je v rámci očekávaných hodnot, protože se dá proložit (s malými odchylkami) rovnicí $t = k*S^2$.

U klasické metody nelze pozorovat očekávanou náročnost, ale rozhodně je možné nalézt určité spojitosti, které indikují, že by zde mohla existovat rovnice $t = k * S^x$, kde x je číslo nějakým způsobem závislé na vstupu. Takže výsledky by do jisté míry mohly korespondovat s těmi z 2.3.3. Za nepřesnosti potenciálně mohou neúplnosti grafů (nemožnost přechodu z libovolného do libovolného stavu), které mohou ovlivnit hlavně klasickou metodu.

Kapitola 5

Testování

Aplikaci jsem se snažil důkladně otestovat, ať již z hlediska stability, tak i z hlediska správnosti udávaných výsledků. Samotné testování probíhalo v několika fázích, abych předešel, co největšímu množství bugů, nestabilit i špatných výsledků metod již během jejich vývoje, což se ukázalo být správnou cestou. Díky tomu jsem mohl práci řešit po určitých menších celcích a tyto celky spolupracovaly pouze přes předem dané komunikační kanály.

Aplikaci jsem se snažil také předložit svým známým a bližším spolužákům, abych měl nějakou zpětnou vazbu a také lepší detekci bugů, protože každý člověk může vymyslet nějaký jiný, mnou neočekávaný způsob ovládání, který by mohl způsobit nestabilitu, neočekávaný stav nebo rovnou pád celého programu, čemuž se logicky snažím předejít.

Proběhlo také testování správnosti výsledků, které vrací metody a jejich porovnání. U metody obchodního cestujícího 2.3.2 se může stát, že bude nalezena přímo opačná cesta, která je rovnocenná s původní. Také je možné nalezení jiné cesty, pokud mají stejné ohodnocení, které je nutné v tomto případě zkontrolovat. Za zmínku stojí také kontrola, zda graf, který je algoritmu předán, má vůbec v kontextu dané úlohy řešení, ale toto bylo v rámci této práce pouze okrajové. Pokud takový případ nastane, tak výpočet sice proběhne, ale nejsou vráceny žádné výsledky.

Poslední testovací sadou, která otestuje obě zmíněné oblasti je jednoduchý skript, který je zde primárně pro sběr dat, ale svým průchodem také otestuje, zda výsledné časy sedí v rozumných a přijatelných mezích, protože nějaký velký skok by i přes průměrování šel v grafu hned vidět. Mnoha průchody se také testuje stabilita algoritmické a grafové části aplikace.

Aplikace byla také testována v obvyklém provozu, kdy byla spuštěna obvykle v grafickém režimu. Poté byla vytvořena jednoduchá reprezentace grafu pomocí okna pro změnu grafů a následně byl testován. Praktikoval se i přístup načtení daného grafu ze souboru. Díky oběma přístupům bylo nalezeno několik chyb drobnějšího rázu, které byly opraveny.

Aplikaci jsem vždy spouštěl a testoval na svém osobním počítači, který funguje na systému Ubuntu 18.04. Typ operačního systému je 64bitový. Teoreticky zajímavá by mohla být ještě informace o procesoru: i3-4005U doplněné 4 GiB RAM. Z tohoto systému pochází všechna data.

Kapitola 6

Závěr

V rámci bakalářské práce byla vytvořena aplikace s grafickým rozhraním, která umožňuje vytvářet a modifikovat grafy, nad kterými je posléze možné provádět algoritmy a doby trvání těchto algoritmů porovnávat. Pro tuto možnost aplikace obsahuje samostatné okno, kde je možné výsledky porovnávat podle různých kritérií, čímž splňuje požadavky zadání. V některých případech požadavky rozšiřuje o další funkcionalitu. Zmíněné grafy je také možné načítat a ukládat do souborů.

Aplikace implementuje problémy hledání nejkratší cesty, dekódování sekvence a problém obchodního cestujícího. Všechny tyto problémy jsou implementovány v dynamickém i nedynamickém algoritmu z nichž některé bohužel neodpovídají předpokládané složitosti, ale stále platí, že dynamické programování zrychlilo výpočet potřebný k výpočtu. Kromě porovnání časových průběhů program umožňuje i zobrazení výsledků metod v samotném grafu.

Aplikace je připravena na možné budoucí modifikace, ať již by šlo o přidání nové reprezentace grafu nebo novou implementaci další metody, či vylepšení/doplnění stávajících. Mnoho součástí aplikace bylo vytvořeno se snahou tyto budoucí modifikace usnadnit. Zajímavé by mohlo být třeba využití vláken pro spouštění výpočtů.

Literatura

[1] *Hidden Markov Models*. [Online; navštíveno 13.05.2019].
URL http://www.cs.utexas.edu/~dana/HMMs3.pdf

[2] *Problém nejkratší cesty*. [Online; navštíveno 02.05.2019].
URL https://www.algoritmy.net/article/36597/Nejkratsi-cesta

[3] *The Viterbi algorithm*. [Online; navštíveno 13.05.2019].
URL http://www.cim.mcgill.ca/~latorres/Viterbi/va_alg.htm

[4] Demel, J.: *Grafy a jejich aplikace*. Academia, 2002, ISBN 80-200-0990-6.

[5] Joshi, V.: *Speeding Up The Traveling Salesman Using Dynamic Programming*. [Online; navštíveno 03.05.2019].
URL https://medium.com/basecs/speeding-up-the-traveling-salesman-using-dynamic-programming-b76d7552e8dd?fbclid=IwAR27Aosu_f_7tmewHI-DrzD6E6aLOzygiS4IE2ZRvgOi621jWfIoefTmVfw

[6] Kamila Fačevicová, P. K., Karel Hron: *Markovovy řetězce a jejich aplikace*. Univerzita Palackého v Olomouci, 2017, ISBN 978-80-244-5432-0, [Online; navštíveno 02.05.2019].

[7] Li, Y.: *A New Exact Algorithm for Traveling Salesman Problem with Time Complexity*. [Online; navštíveno 13.05.2019].
URL https://arxiv.org/abs/1412.2437

[8] Martin Mareš, T. H.: *Průvodce labyrintem algoritmů*. CZ.NIC, 2017, ISBN 978-80-88168-22-5.

[9] Project, T. G.: *gtkmm Reference Manual*. [Online; navštíveno 03.05.2019].
URL https://developer.gnome.org/gtkmm/3.0/

[10] Project, T. G.: *Programming with gtkmm 3*. [Online; navštíveno 03.05.2019].
URL https://developer.gnome.org/gtkmm-tutorial/stable/

[11] Team, T. G.: *The GTK Project*. [Online; navštíveno 03.05.2019].
URL https://www.gtk.org

Příloha A

Manuál

Tato sekce představuje zprovoznění daného řešení.

Instalace

- Potřebné knihovny by měly být již nativně na Vašem počítači, pokud to tak však není, tak je nutné doinstalovat knihovnu GTKmm. Oficiální návod pro instalaci je dostupný zde https://www.gtkmm.org/en/download.html.
- Je také nutné, aby Váš překladač g++ podporoval standard C++14.

Spuštění

Program přeložíte pomocí příkazu make, který vytvoří spustitelný soubor Graph, který lze spouštět následovně: `./Graph <TGPrep> <NormCesta> <MarkCesta>`

- Parametr `<TGPrep>` je přepínač grafického a textového režimu a nabývá hodnot "-t"pro textový režim, "-g"pro grafický režim nebo "-h"pro výpis nápovědy. "-t"nelze použít samostatně bez `<NormCesta>` a `<MarkCesta>`
- Parametr `<NormCesta>` je cesta k souboru, který obsahuje popis Neorientovaného grafu
- Parametr `<MarkCesta>` je cesta k souboru, který obsahuje popis grafu popisující reprezentaci Markovových skrytých stavů

Možnosti spustění aplikace jsou:

- `./Graph -t Examples/Norm/Full6 Examples/Mark/file2`
- `./Graph -g Examples/Norm/Full6 Examples/Mark/file2`
- `./Graph -g`
- `./Graph -h`

Příloha B

Formát grafových souborů

Soubory, které obsahují data grafů mají následující tvar. Oddělení položek na jednom řádku je provedeno pomocí tabulatoru, oddělení řádků, pouze pomocí prázdného řádku. Bohužel nejsou přípustné žádné jiné způsoby oddělení a také není možné vkládat žádné komentáře.

Tvar souboru pro popis Neorientovaného grafu může být následujícího tvaru:

```
  A   B   D   E   G
A   -  10  35  20 100
B  10   -  10   -  25
D  35  10   -  30   -
E  20   -  30   -  10
G 100  25   -  10   -
A   G
```

- Na prvním řádku je seznam vrcholů
- Na následujících řádcích je matice, která popisuje hrany. "-"znamená, že hrana neexistuje. Teoreticky stačí zadat pouze polovinu grafu podle osy na diagonále mezi levýho horním rohem a pravým spodním. Jsou možné oba způsoby vyplnění
- na posledním řádku jsou přítomny počáteční a koncový vrchol

Tvar souboru popisující graf Markovových skrytých modelů je popsán v následujícím tvaru:

```
Na cestě  Doma  Škola
Procházka  Studuje  Proklastinace  Pracuje
Na cestě:0.2  Doma:0.4  Škola:0.1  Práce:0.3
Na cestě  Na cestě:0.2  Doma:0.3  Škola:0.3  Práce:0.2
Doma  Na cestě:0.2  Doma:0.3  Škola:0.3  Práce:0.2
Škola  Na cestě:0.4  Škola:0.3  Práce:0.3

Na cestě  Procházka:0.2  Studuje:0.1  Pracuje:0.7
Doma  Procházka:0.4  Proklastinace:0.4  Studuje:0.2
Škola  Proklastinace:0.4  Studuje:0.4  Pracuje:0.2
```

Procházka Studuje Pracuje

- Na prvním řádku je seznam stavů
- Na druhém řádku je seznam pozorování
- Na třetím řádku je seznam pravděpodobností, v jakém stavu se bude systém nacházet na začátku
- Na následujících řádcích je matice, která popisuje hrany. Řádek vždy začíná názvem stavu a možnými hranami, včetně jejich pravděpodobností oddělené dvojtečkou
- Na následujících řádcích je matice, která popisuje pravděpodobnosti emitování pozorování určitých stavů
- Po dalším volném řádku, na posledním řádku je přítomná hledaná sekvence

Považuji za slušné zmínit, že soubor /Examples/Mark/file1 je inspirován příkladem z https://cs.wikipedia.org/wiki/Viterbiho_algoritmus, bylo to z důvodu snadného ověření základního průchodu algoritmu.

Příloha C

obsah paměťového média

- Zdrojové soubory v adresáři **src**
- Soubory pro dokumentaci v adresáři **doc**
- **makefile** pro překlad zdrojových souborů
- Dokumentace k práci **xbilos00_BP.pdf**
- Podadresář se soubory s daty normálních grafů **Examples/Norm**
- Podadresář se soubory s daty grafů HMM **Examples/Mark**
- Soubory s výsledky v adresáři **Res**
- Jednoduchý testovací skript **Execute.sh**