Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Rekurzivní algoritmy v teorii grafů

Matěj Pátek

Bakalářská práce
2023

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Matěj Pátek**
Osobní číslo:	**I19133**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**Rekurzivní algoritmy v teorii grafů**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Algoritmy z teorie grafů mají velké použití např. v logistice, plánování projektů a v různých typech sítí. U některých algoritmů se nevyhneme rekurzi – tedy rozkladu problému na podproblémy. Cílem práce bude studium rekurzivních grafových algoritmů a jejich použitelnosti na konkrétních úlohách. Jako příklad uveďme např. určení všech možných maximálních toků v síti či určení všech nejkratších cest. Cílem teoretické části bude popis rekurzivních grafových algoritmů. Cílem praktické části bude jejich implementace ve vlastní aplikaci.

Rozsah pracovní zprávy:
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

BALAKRISHNAN, V. *Schaum's outline of theory and problems of graph theory*. New York: McGraw-Hill, c1997, viii, 293 p. ISBN 00-700-5489-4.
TÖPFER, Pavel. *Algoritmy a programovací techniky*. 1. vyd. Praha: Prometheus, 1995, 299 s. ISBN 80--858-4983-6.
MATOUŠEK, Jiří. *Kapitoly z diskrétní matematiky*. Vyd. 1. Praha: Karolinum, 2002, 381 s. ISBN 80-246--0084-6.

Vedoucí bakalářské práce: **RNDr. Josef Rak, Ph.D.**
Katedra matematiky a fyziky

Datum zadání bakalářské práce: **16. prosince 2022**
Termín odevzdání bakalářské práce: **12. května 2023**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2023

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 8. 5. 2023

Matěj Pátek

PODĚKOVÁNÍ

Chtěl bych poděkovat panu RNDr. Josefu Rakovi, Ph.D. za odborné vedení práce a cenné rady, které mi pomohly tuto práci zkompletovat. Dále bych chtěl poděkovat své rodině, která mě během studia podporovala.

ANOTACE

Algoritmy z teorie grafů mají velké použití např. v logistice, plánování projektů a v různých typech sítí. U některých algoritmů se nevyhneme rekurzi – tedy rozkladu problému na podproblémy. Cílem práce bude studium rekurzivních grafových algoritmů a jejich použitelnosti na konkrétních úlohách. Jako příklad uveďme např. určení všech možných maximálních toků v síti či určení všech nejkratších cest. Cílem teoretické části bude popis rekurzivních grafových algoritmů. Cílem praktické jejich implementace ve vlastní aplikaci.

KLÍČOVÁ SLOVA

Java, algoritmus, rekurze, graf, hloubka, šířka, detekce kružnic, vrchol, hrana

TITLE

Recursive algorithms in graph theory

ANNOTATION

Algorithms from graph theory have many applications in e.g., logistics, project planning and various types of networks. For some algorithms, recursion - i.e., decomposing the problem into subproblems - is unavoidable. The aim of this paper will be to study recursive graph algorithms and their applicability to specific problems. Examples include determining all possible maximum flows in a network or determining all shortest paths. The aim of the theoretical part will be to describe recursive graph algorithms. The goal of the practical part is to implement them in own application.

KEYWORDS

Java, algorithm, recursion, graph, depth, breadth, circle detection, node, edge

Obsah

SEZNAM OBRÁZKŮ

SEZNAM ZKRATEK

SEO	Search Engine Optimization
SERP	Search Engine Results Page
LIFO	Last In First Out
FIFO	First In First Out
FCFS	First Come First Served
DFS	Depth First Search
BFS	Breadth First Search
JRE	Java Runtime Environment
CSS	Cascading Style Sheets
API	Application Programming Interface
JDK	Java Development Kit

ÚVOD

Tato bakalářská práce se věnuje problematice algoritmů v teorii grafů, které jsou klíčové pro mnoho oblastí informatiky a matematiky. Teorie grafů se využívá v různých oblastech, jako jsou například počítačové sítě, sociální sítě, navigační systémy nebo biologie.

Cílem práce je nejenom popsání teoretických základů algoritmů a teorie grafů, ale také vytvoření užitečné aplikace, která umožní uživatelům vizualizaci a porozumění těmto algoritmům. Tato aplikace umožní uživateli zadat graf a spustit na něm jeden z implementovaných algoritmů, který bude následně vizualizován. Uživatel bude moci sledovat postupné řešení a výsledek algoritmu.

Práce je rozdělena do několika kapitol. V první kapitole se budeme zabývat pojmem algoritmus a jeho využitím v různých oblastech. V druhé kapitole budou vysvětleny základní pojmy teorie grafů, jako jsou orientované a neorientované grafy, vrcholy, hrany, cyklicita grafu apod.

V další kapitole se budeme věnovat rekurzi a konkrétně hanojské věži, která je typickým příkladem pro ilustraci práce s rekurzí. Dále rozebereme datové struktury fronta a zásobník, které se nedílně pojí k algoritmům, které budou v práci popisovány.

Poté následuje část, kde budou popsány jednotlivé algoritmy, konkrétně DFS (prohlídka do hloubky), BFS (prohlídka do šířky), detekce kružnic a jeden z algoritmů na tok v síti, konkrétně Ford-Fulkerson. Každý z těchto algoritmů bude navíc vysvětlen na příkladu.

V další kapitole se zaměříme na samotnou aplikaci, kde bude vysvětlena její obsluha a veškeré její funkcionality. Aplikace bude vyvinuta v programovacím jazyce Java a pro grafické uživatelské prostředí bude využita knihovna JavaFX.

V závěru práce jsou shrnuty výsledky a návrhy na další možná rozšíření aplikace.

1 ALGORITMUS

Algoritmus původně pochází z latiny a označuje přesný postup řešení úkolů a problémů. V moderní době se tento termín často používá v souvislosti s programováním, tvorbou webových stránek a aplikací. Dále je spojován s internetovým marketingem a SEO a zahrnuje algoritmy používané vyhledávači k hodnocení webů a jejich zobrazení ve výsledcích vyhledávání. Tyto algoritmy umožňují vyhledávacím robotům posoudit kvalitu webových stránek a určit jejich pořadí v SERP. [1]

1.1 Algoritmus v programování

Algoritmus je pro programátory plán, jak řešit problémy, podobně jako recept v běžném životě. Popisuje postup, který nám pomáhá dosáhnout požadovaného výsledku s určitými ingrediencemi nebo akcemi. Například, návod na přípravu bramborové polévky nebo jak zavázat tkaničky jsou příklady algoritmů. Programování pak znamená převést algoritmus do programovacího jazyka. Výsledkem programování je program, který postupuje krok po kroku podle algoritmu, aby dosáhl požadovaného výsledku. [1]

1.2 Historie

Začátek vzniku algoritmu sahá až ke vzniku lidstva, protože i pravěké činnosti měly pevně daný postup a výsledek. Pojem algoritmus v té době nebyl znám, ale postupy, které splňují jeho podmínky, se objevily již v období starověkého Řecka. Dokládají to například Euklidův algoritmus pro výpočet největšího společného dělitele z 3. století př. n. l. nebo Herónův algoritmus pro výpočet obsahu trojúhelníku a algoritmus pro hledání odmocniny z čísla z 1. století n. l. Tyto algoritmy podstatu a použití algoritmu demonstrovaly, i když pojem „algoritmus“ tehdy neexistoval. [1]

Samotný pojem algoritmus vzniknul zhruba o tisíc let později v 9. století našeho letopočtu, a je spojován se jménem perského matematika Abú Abd Alláh Muhammad ibn Músá al-Chwárizmí, který vytvořil systém arabských číslic. Prvním významem slova algoritmus tak bylo „provádění aritmetiky pomocí arabských číslic“ a používal se k vyjádření zejména matematických postupů. Pojem algoritmus ve smyslu dnešního významu se používá zhruba od 20. století. [1]

1.3 Využití známých algoritmů

Algoritmy jsou v současnosti používány v široké škále vědních oborů. Nejčastěji se s nimi setkáváme v matematice a programování, kde jsou klíčovým nástrojem pro sestavení teoretického řešení daného problému. Vytváření algoritmů je dnes samostatným vědním

oborem, který se zaměřuje na rozvoj existujících algoritmů, z nichž existuje mnoho, jako například Eratosthenovo síto pro nalezení prvočísel, Dijkstrův algoritmus pro nalezení nejkratší cesty v grafu, Ballman-Fordův algoritmus pro výpočet nejkratší cesty v ohodnoceném grafu nebo algoritmus de Casteljau pro výpočet bodu na Beziérově křivce. [2]

1.4 Vlastnosti algoritmu

Aby měl algoritmus význam, je důležité, aby byl snadno pochopitelný pro toho, kdo bude daný bude provádět. Z toho důvodu by měl mít následující vlastnosti:

- správnost – výstup algoritmu musí být přesný, tedy aby produkoval správné výsledky pro různá vstupní data,
- rezultativnost – vždy musí být dosaženo nějakého výsledku, ať pozitivního či negativního tak, aby nedošlo ke zbytečnému zdržení,
- konečnost – měl by být rozdělen na dílčí fáze, které vedou z počátku (bodu A) do konce (bodu B),
- hromadnost – algoritmus musí být aplikovatelný na určitou skupinu úloh, nikdy by neměl řešit jediný specifický problém,
- determinovanost neboli jednoznačnost – musí být přesně určeno, co má nastat po dokončení každého kroku,
- opakovatelnost – výsledek musí být vždy stejný, pokud zadáme totožná vstupní data. [2]

2 ZÁKLADNÍ POJMY

Teorie grafů je matematická oblast, která se objevila v 18. století a vznikla díky práci švýcarského matematika Leonharda Eulera – „Problém sedmi mostů města Královce“. Euler použil „geometrii pozic“, což dnes nazýváme teorií grafů, a proto je považován za zakladatele této disciplíny. V roce 1936 maďarský matematik Dénes Kőnig publikoval první učebnici teorie grafů s názvem „Theorie der endlichen und unendlichen Graphen“. Teorie grafů má dnes praktické uplatnění při řešení mnoha úloh, protože umožňuje snadnou a intuitivní modelaci reálných situací jako množiny objektů a vztahů mezi nimi. [3]

Teorie grafů je důležitou součástí diskrétní matematiky, která se úspěšně využívá při řešení mnoha praktických problémů. Její výhodou je, že dokáže intuitivně modelovat reálné situace jako množinu objektů (vrcholy) a vztahy mezi nimi (hrany). Pokud dokážeme převést konkrétní úlohu do řeči teorie grafů, můžeme využít již existující obecné řešení a známé algoritmy při implementaci řešení. Snadnost implementace objektů a postupů teorie grafů v počítači je bezesporu výhodou zásadní. [3]

2.1 Graf

Graf je klíčovým pojmem v diskrétní matematice. Definujeme ho jako algebraickou strukturu, která slouží k popisu objektů a jejich vztahů. Graf se skládá z vrcholů a hran, které spojují tyto vrcholy. Nespornou výhodou je, že je lze snadno a přehledně zobrazit. Vrcholy se reprezentují jako body nebo symboly a hrany jako křivky, které je spojují. Tento způsob zobrazení je intuitivní a umožňuje snadno porozumět vztahům mezi vrcholy a hranami. [3]

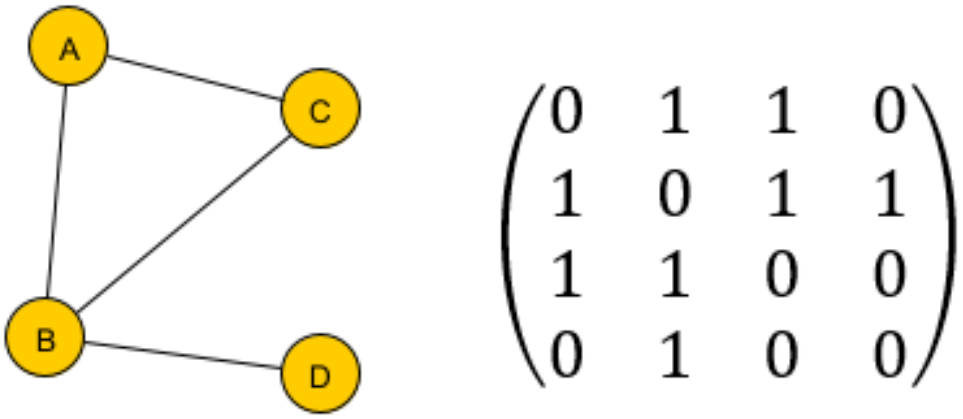
Při složitých analýzách a implementaci algoritmů pro řešení složitých úloh nestačí pouze intuitivní reprezentace grafů jako puntíků a čar v rovině. Podstatou struktury grafu jsou objekty a jejich vazby. Pokud mezi dvěma objekty existuje vazba, reprezentujeme ji jako dvojici (dvouprvkovou množinu) příslušných objektů (vrcholů). Pokud vazba mezi dvěma objekty neexistuje, tuto dvojici objektů do množiny hran nezařadíme. [3]

Definice 1*: **Graf** G je uspořádaná dvojice G = (V, E), kde V je nějaká neprázdná množina vrcholů a E je množina dvouprvkových podmnožin množiny V. Prvky množiny V se jmenují **vrcholy** grafu G a prvky množiny E **hrany** grafu G.* [7]

Matice sousednosti je jedním z nejběžnějších způsobů reprezentace grafů v algebře. Je to čtvercová matice, jejíž prvky jsou definovány takto:

- pokud existuje hrana spojující vrcholy i a j, pak prvek na řádku i a sloupci j matice souslednosti je roven jedné,
- pokud hrana mezi vrcholy i a j neexistuje, pak prvek na řádku i a sloupci j matice souslednosti je roven nule. [7]

Matice sousednosti může být orientovaná nebo neorientovaná v závislosti na tom, zda jsou hrany v grafu orientované nebo ne. V neorientovaném grafu je matice symetrická podle hlavní diagonály, zatímco v orientovaném grafu tomu tak nemusí být. [7]

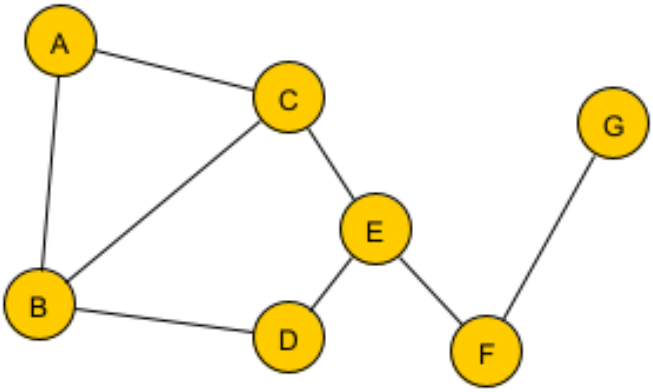


Obrázek 1: Graf a jeho matice sousednosti

2.1.1 Neorientovaný graf

Definice 2: ***Neorientovaný graf*** *je trojice G = (V, E, ε) tvořená konečnou množinou V, jejíž prvky nazýváme uzly, konečnou množinou E, jejíž prvky nazýváme neorientovanými hranami a zobrazením ε (vztah incidence), které přiřazuje každé hraně e jedno nebo dvouprvkovou množinu uzlů.* [4]

Příkladem může být využit například pro reprezentaci sítě vodních toků nebo třeba rodinných příbuzenských vztahů.

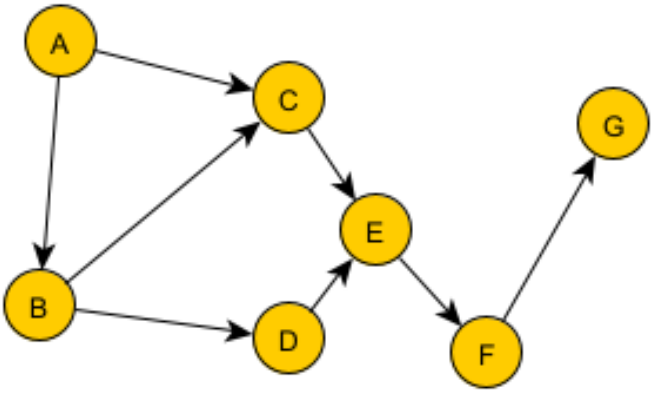


Obrázek 2: Neorientovaný graf

2.1.2 Orientovaný graf

Definice 3*: **Orientovaný graf** je trojice G = (V, E, ε) tvořená konečnou množinou prvků V, jejíž prvky nazýváme uzly, konečnou množinou E, jejíž prvky nazýváme orientovanými hranami a zobrazením ε: E → V², které nazýváme vztahem incidence, a které přiřazuje každé hraně e ∈ E uspořádanou dvojici uzlů.* [4]

Příkladem využití může být reprezentace silniční sítě, kde má každá ulice jednoznačně definovaný směr.

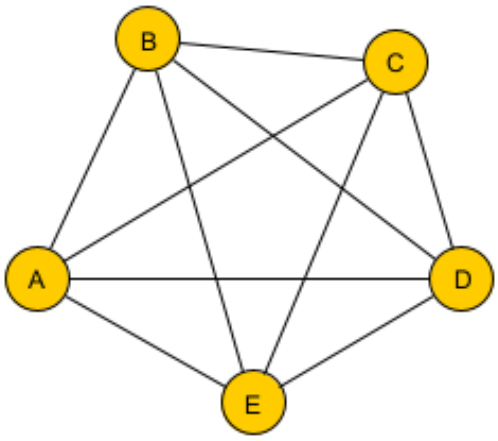


Obrázek 3: Orientovaný graf

2.1.3 Kompletní graf

Definice 4: *Graf na n vrcholech, kde $n \in N$, který obsahuje všech $\binom{n}{2}$ hran se nazývá úplný nebo také **kompletní graf** a značí se K_n.* [3]

Jinými slovy je to takový graf, kde je každý uzel (vrchol) propojen hranou s každým jiným uzlem v grafu. [4]



Obrázek 4: Kompletní graf

2.1.4 Sled, tah, cesta, kružnice a dráha

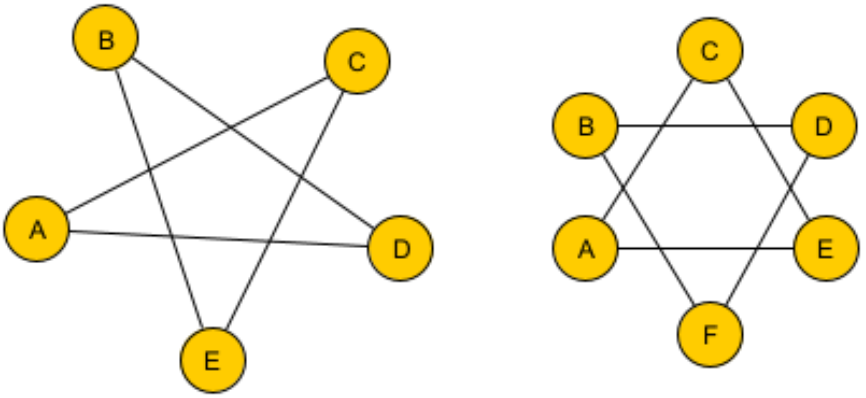
Definice 5: ***Sledem*** *mezi vrcholy u a v, nazveme posloupnost vrcholů a hran:*

$S = \{u, h_1, u_1, h_2, \ldots, u_{n-1}, h_n, v\}$.

Vrcholy u a v budeme nazývat krajními vrcholy sledu. Pokud jsou krajní body sledu stejné, hovoříme o uzavřeném sledu. ***Sled****, ve kterém se neopakuje ani jedna hrana nazveme tahem. Tahem, ve kterém se neopakuje ani jeden vrchol nazveme* ***cestou****. Cestu mezi dvěma vrcholy budeme značit m (u, v).* ***Uzavřený sled****, ve kterém se neopakuje žádná hrana a žádný vrchol (kromě okrajového) nazýváme* ***kružnicí****. V případě orientovaného grafu orientovaného grafu se místo cesty používá termín* ***dráha****.* [18]

2.1.5 Souvislý a nesouvislý graf

Definice 6: *Neorientovaný graf nazveme* ***souvislým****, pokud mezi každými dvěma různými vrcholy existuje alespoň jedna cesta. V opačném případě hovoříme o grafu* ***nesouvislém****. Orientovaný graf nazveme* ***silně souvislý****, pokud mezi libovolnou dvojicí vrcholů u a v existuje jak dráha z u do v a tak dráha z v do u. V případě, že alespoň jedna z drah mezi vrcholy u a v hovoříme o* ***orientovaně souvislém grafu****. V případě, že je graf souvislým ve smyslu neorientovaného grafu (absentujeme orientace u hran) hovoříme o* ***neorientovaně souvislém grafu****.* [18]

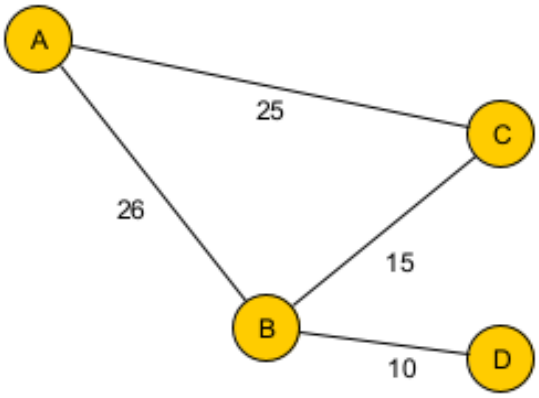


Obrázek 5: Souvislý a nesouvislý graf (inspirováno ze zdroje [5])

2.2 Ohodnocení grafu

Definice 7: *„**Ohodnocení grafu** G je funkce w: $E(G) \rightarrow R$, která každé hraně $e \in E(G)$ přiřadí reálné číslo $w(e)$, kterému říkáme váha hrany. **Ohodnocený graf** je graf G spolu s ohodnocením hran reálnými čísly. **Kladně ohodnocený** (říkáme také vážený) graf G má takové ohodnocení w, že pro každou hranu $e \in E(G)$ je její váha $w(e)$ kladná.“* [3]

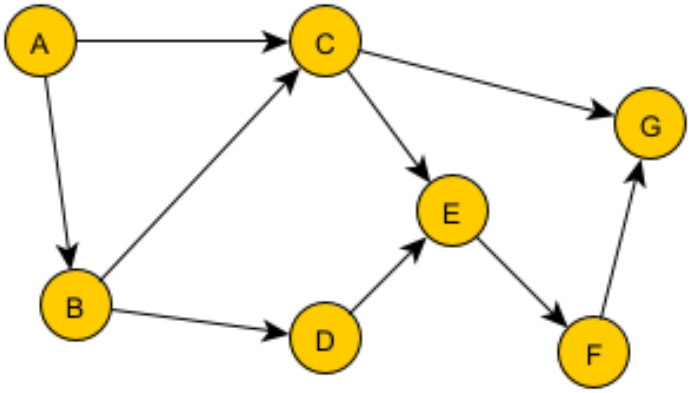
Hodnoty hran v grafu bývají nejčastěji kladná čísla, ale není to podmínkou a hrana může být v podstatě ohodnocena jakýmkoliv končeným reálným číslem. Pokud jsou všechny hodnoty kladné, mluvíme o „kladně váženém“ nebo jen „váženém“ grafu. V praxi se často omezujeme na ohodnocení přirozenými čísly, jelikož tyto hodnoty obvykle odpovídají fyzikálním veličinám a můžeme zvolit dostatečně malou jednotku pro jejich vyjádření tak, aby rozdíly v desetinách neměly význam. Používání přirozených čísel nám také umožňuje vyhnout se chybám při početních operacích nebo zaokrouhlování, které by mohly vést k rozdílným výsledkům algoritmů na různých systémech. [3]



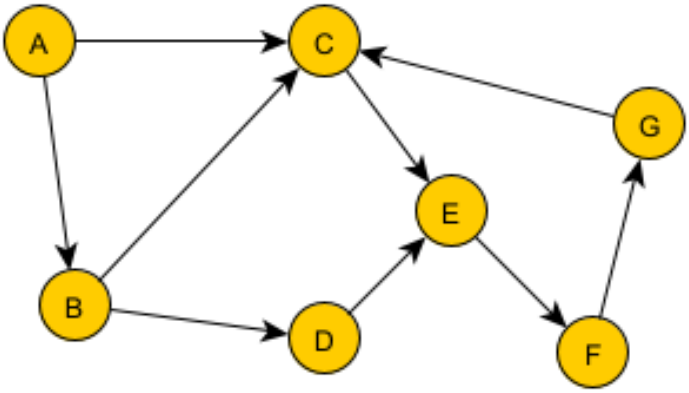
Obrázek 6: Ohodnocený graf

2.3 Cyklicita grafu

Graf, který neobsahuje žádný cyklus nazýváme acyklický. Disponuje-li alespoň jednou kružnicí, nazývá se graf cyklický. [4]



Obrázek 7: Acyklický graf



Obrázek 8: Cyklický graf

Pro dosažení cyklicity grafu, respektive pro vznik kružnice v grafu, stačilo pouze změnit orientaci hrany mezi vrcholem C a vrcholem D. V grafu následně vznikla kružnice (C, E, F, G).

2.4 Síť

Definice 8: ***Síť*** *je čtveřice $S = (G, z, s, w)$, kde G je orientovaný graf. Vrcholy $z \in V(G)$, $s \in V(G)$ budeme nazývat zdroj a stok v síti S. Funkce $w: E(G) \rightarrow R+$ je kladné ohodnocení hran, které každé hraně přiřadí tzv. kapacitu hrany.* [3]

Síť je speciální typ grafu, který splňuje několik podmínek. Konkrétně se jedná o graf, který má konečný počet uzlů a hran, který je souvislý, orientovaný, acyklický a ohodnocený. V síti existuje právě jeden počáteční uzel, ze kterého mohou vést hrany do dalších uzlů s ohledem na směr hran. [4]

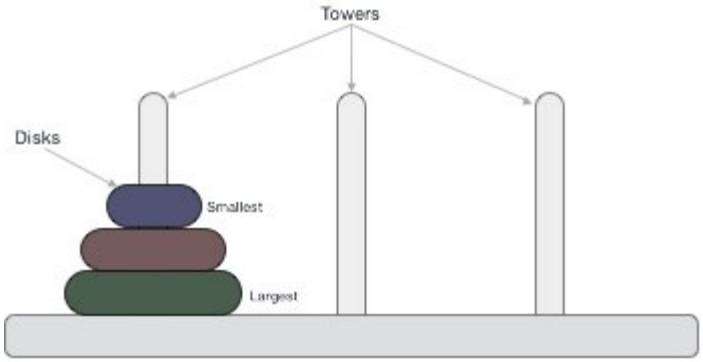
3 REKURZE

Proces, při kterém funkce volá sama sebe přímo nebo nepřímo, se nazývá rekurze a příslušná funkce se nazývá rekurzivní funkce. Pomocí rekurzivního algoritmu lze některé problémy řešit poměrně snadno. Příkladem takových problémů jsou Hanojské věže, průchody stromem, prohlídka grafu do šířky a další. Rekurzivní funkce řeší určitý problém tak, že volá kopii sebe sama a řeší menší podproblémy původních problémů. Podle potřeby lze generovat mnoho dalších rekurzivních volání. Podstatné je vědět, že bychom měli zajistit určitý případ, abychom tento rekurzivní proces ukončili. Můžeme tedy říci, že pokaždé funkce zavolá sama sebe s jednodušší verzí původního problému. [6] [8]

Rekurze je mocným nástrojem pro programátory, protože umožňuje psát krátké a elegantní programy, které by jinak byly pracnější. Avšak, použití rekurze může mít svá rizika, pokud nejsou používána opatrně a vhodně. Pokud se rekurze použije na nevhodném místě, může to vést k nesrozumitelnému a nepřehlednému kódu. Hledání a odstraňování chyb v rekurzivních programech je navíc obtížnější než v programech bez rekurze. Nesprávné použití rekurze může také vést k pomalým programům, které jsou prakticky nepoužitelné. Proto by měla být rekurze používána opatrně a jen tam, kde je to opravdu účelné a nezpůsobí přílišné zpomalení výpočtu. [8]

3.1 Hanojská věž

Hanojská věž je matematický hlavolam, který se skládá ze tří věží (kolíčků) a více než jednoho disku. Tyto disku jsou různě velké a jsou na sebe naskládány ve vzestupném pořadí, to znamená, že menší sedí nad větším. Existují i další varianty hlavolamu, kde se počet disků zvyšuje, ale počet věží zůstává stejný. [9]



Obrázek 9: Hanojská věž se třemi disky [9]

3.1.1 Pravidla

Úkolem je přesunout všechny disky do jiné věže, aniž by byla porušena posloupnost uspořádání. Pravidla, která je potřeba při řešení dodržovat jsou následující:

- Mezi věžemi lze v daném okamžiku přesunout pouze jeden disk.
- Odstranit lze pouze vrchní disk.
- Větší disk se nesmí položit na menší. [9]

Hanojskou věž s n disky lze vyřešit v minimálně 2^n-1 krocích. Pokud bychom tedy měli hlavolam se 3 disky, pak by řešení trvalo 7 kroků (8-1=7). [9]

3.1.2 Algoritmus

Při řešení tohoto hlavolamu se postupuje následovně: vždy se přesune nejmenší kotouč, a poté jiný kotouč než nejmenší. Pokud se přesouvá nejmenší kotouč, pohybuje se vždy o jednu věž dál ve stejném směru, s tím, že při sudém počtu kotoučů se pohybuje doprava a při lichém počtu kotoučů doleva. Pokud se nejmenší kotouč již nachází na poslední věži v daném směru, přesune se na věž na opačném konci. Pokud se má přesunout jiný kotouč než nejmenší, existuje pouze jediný způsob, jak to provést. Tímto způsobem lze tento hlavolam vyřešit pomocí nejmenšího počtu tahů. [10]

3.1.3 Rekurzivní řešení

Rekurzivní řešení tohoto problému se zakládá na tom, že při řešení musí být největší kotouč vždy přesunut minimálně jednou. Pokud máme n kotoučů, můžeme nejprve rekurzivně přesunout n-1 kotoučů z počáteční věže na odkládací věž. Poté můžeme přesunout největší kotouč z počáteční věže na cílovou věž, a nakonec rekurzivně přesunout n-1 kotoučů

z odkládací věže na cílovou věž. Při přesunu n-1 kotoučů můžeme opět použít stejný postup, což nám umožní použít rekurzivní volání této procedury pro každý menší počet kotoučů. [10]

Celkový postup tedy spočívá v následujících krocích:

(1) Pokud je n>1, přesuneme n-1 kotoučů z počáteční věže na odkládací věž pomocí rekurzivního volání této procedury.

(2) Přesuneme největší kotouč z počáteční věže na cílovou věž.

(3) Pokud je n>1, přesuneme n-1 kotoučů z odkládací věže na cílovou věž pomocí rekurzivního volání této procedury. [10]

4 SEZNAMY PRVKŮ

Množiny a seznamy jsou základní datové struktury pro ukládání a manipulaci s daty. V množině nezáleží na pořadí prvků a každý prvek má jedinečný klíč. V seznamu jsou prvky uloženy v pevně stanoveném pořadí a mohou se vyskytovat opakovaně. Existují dvě základní možnosti, jak realizovat seznam – pomocí pole nebo lineárního spojového seznamu. Přidání a odebírání prvků se provádí na konci seznamu a lze je implementovat s konstantní časovou složitostí pro zásobníky a fronty. Vyhledávání hodnot v seznamu má vždy lineární složitost a není typickou operací pro zásobníky a fronty. [11]

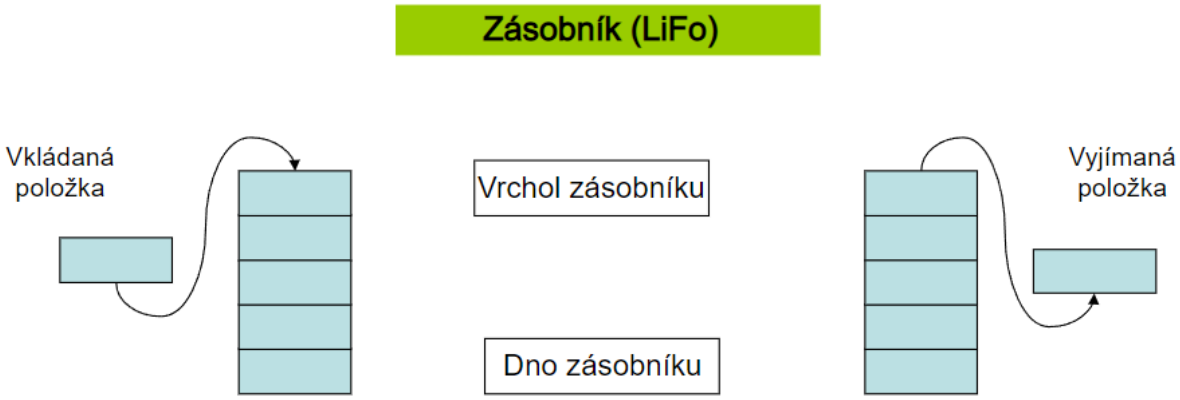
V případě pole musí být deklarována předem jeho maximální délka, což může být nevýhodou. Na druhou stranu, pole jsou rychlejší a mohou být využity pro situace, kdy je potřeba rychle získat určitý prvek na základě jeho indexu. Lineární spojový seznam je flexibilnější a dynamicky se mění v závislosti na počtu prvků, což umožňuje úsporu paměti v případě, kdy není známa maximální délka seznamu. Nicméně, dynamická alokace a použití ukazatelů může být paměťově náročnější a operace s lineárním spojovým seznamem mohou být pomalejší. [11]

Zásobník a fronta jsou speciální druhy seznamů, které mají specifické vlastnosti. Zásobník umožňuje přidávání a odebírání prvků pouze na jednom konci seznamu a lze ho implementovat pomocí pole nebo lineárního spojového seznamu. Přidání a odebírání prvků na konci seznamu má konstantní časovou složitost. Fronta umožňuje přidávání na jeden konec a odebírání z druhého konce seznamu a lze ji také implementovat pomocí pole nebo lineárního spojového seznamu. Stejně jako u zásobníku má přidání a odebírání prvků konstantní časovou složitost. [11]

Vyhledávání hodnot v seznamu je operace, která vyžaduje lineární průchod celým seznamem, což má lineární časovou složitost. Tato operace je však netypická pro zásobníky a fronty, protože ty se zaměřují na práci s prvky na koncích seznamu. [11]

4.1 Zásobník

Zásobník je asi nejpoužívanějším typem seznamů. Konec seznamu (tzv. vrchol zásobníku) slouží jak k přidávání, tak i k odebírání prvků. Je to jediné místo, kde je možné s prvky pracovat, jelikož se nemohou přidávat ani odebírat na druhém konci seznamu (označovaném jako dno zásobníku) ani nikde uprostřed. Pokud tedy chceme odebrat jeden prvek ze zásobníku, bude to ten, který do něj byl vložen jako poslední a je aktuálně na vrcholu zásobníku. V anglicky psané literatuře se zásobník označuje jako struktura typu LIFO. [11]

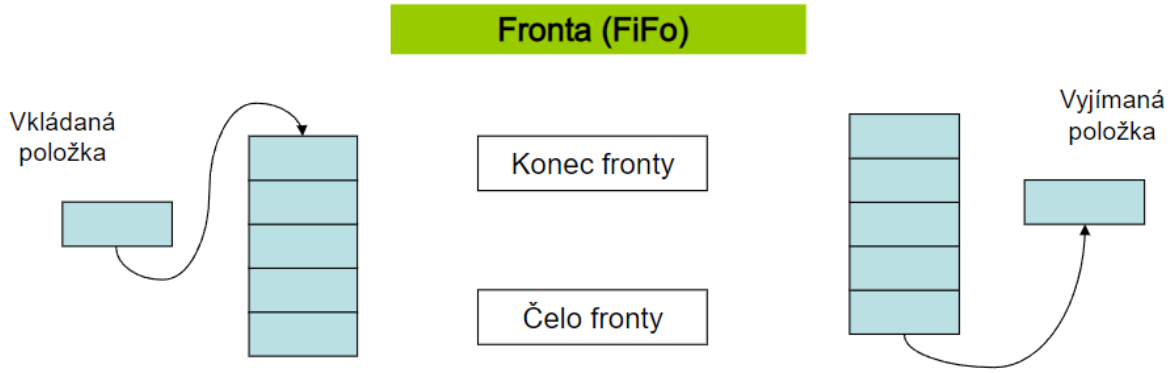


Obrázek 10: Zásobník [12]

Zásobník má nespočet využití. Používá se například při řešení úloh metodou „rozděl a panuj“ k odložení informaci o tom, které dílčí úkoly je ještě třeba vykonat, slouží jako základní datová struktura pro řízení průchodu do hloubky, používá se při zpracovávání aritmetických výrazů. Techniku zásobníku využívá i samotný mechanismus volání procedur a funkcí ve vyšším programovacím jazyce, kdy v paměti typu zásobník je přidělován prostor pro umístění lokálních proměnných volaných pod programů. Zásobník s operacemi vkládání a vypouštění prvků se velice snadno programuje, z hlediska náročnosti obsluhy je to nejjednodušší typ seznamu. Používá se proto i tehdy, potřebujeme-li dočasně uložit nějaké údaje a později je průběžně zpracovávat, aniž by nám záleželo na jejich pořadí. [11]

4.2 Fronta

Fronta je seznam, kde jsou uložena data v pořadí, jako když lidé čekají ve frontě v obchodě. Ti, kteří přišli dříve, jsou obsluhováni dříve a odejdou dříve. Nová data jsou vkládána na konec fronty („příchod“), zatímco data jsou odebírána z druhého konce fronty („odchod“). V anglicky psané literatuře se fronta označuje zkratkou FIFO nebo také FCFS. [11]



Obrázek 11: Fronta [12]

Fronta se v programech používá tehdy, pokud potřebujeme dočasně odložit nějaká data a zachovat pro další zpracování jejich původní pořadí. Slouží jako základní datová struktura pro řízení průchodu do šířky. Její programová realizace v poli i dynamicky je o něco komplikovanější, než tomu bylo v případě zásobníku. [11]

Fronta, zásobník a rekurze jsou tři klíčové koncepty, které mohou být propojeny několika způsoby. [13]

Při rekurzi se vytváří nové instance funkce nebo procedury a ty se ukládají na zásobník. Výpočet pokračuje s novou instancí, dokud se nedosáhne podmínky ukončení rekurze, poté se výpočet vrací zpět na předchozí úroveň a pokračuje se dál. Zásobník tak slouží k ukládání kontextu volání funkce nebo procedury a k jejich následnému obnovení. Frontu lze využít například k řízení úloh v rámci fronty čekajících procesů. [13]

V některých případech se mohou využít kombinace těchto konceptů. Například může být použit zásobník k ukládání stavu rekurzivní funkce a fronta ke zpracování nezpracovaných prvků. Tento přístup může být výhodný v situacích, kdy potřebujeme řešit problém, který lze nejlépe vyřešit rekurzivní funkcí a současně udržovat správné pořadí zpracování prvků. [13]

Celkově lze tedy říci, že rekurze, zásobník a fronta jsou tři klíčové koncepty, které mohou být využity samostatně nebo v kombinaci pro řešení různých problémů. [13]

5 POUŽITÉ ALGORITMY

5.1 DFS

Algoritmus DFS je rekurzivní algoritmus, který využívá myšlenku zpětného vyhledávání. Zahrnuje vyčerpávající prohledání všech uzlů, a to tak, že se postupuje dopředu, pokud je to možné, jinak se postupuje zpětně. [14]

S algoritmem je spojeno slovo „backtrack“, což znamená, že pokud se na aktuální cestě neobjeví žádné další uzly, je potřeba se vrátit zpět po stejné cestě, aby bylo možné najít další potenciální uzly, kterými je možné v průchodu pokračovat. Všechny uzly na aktuální cestě budou navštíveny, dokud nebudou projity všechny nenavštívené uzly, načež bude vybrána další cesta. [14]

5.1.1 Průběh algoritmu

Standardně se implementuje DFS tak, že dělíme vrcholy do dvou kategorií, a to na vrcholy navštívené a nenavštívené. Účel algoritmu je označit všechny vrcholy jako navštívené a zároveň se vyhnout cyklům. [15]

Algoritmus funguje následovně:

(1) Do zásobníku se vloží libovolný vrchol.

(2) Vrchol se odebere ze zásobníku a označí se jako navštívený (jakým způsobem se pracuje se zásobníkem bylo vysvětleno podrobněji v kapitole č. 4).

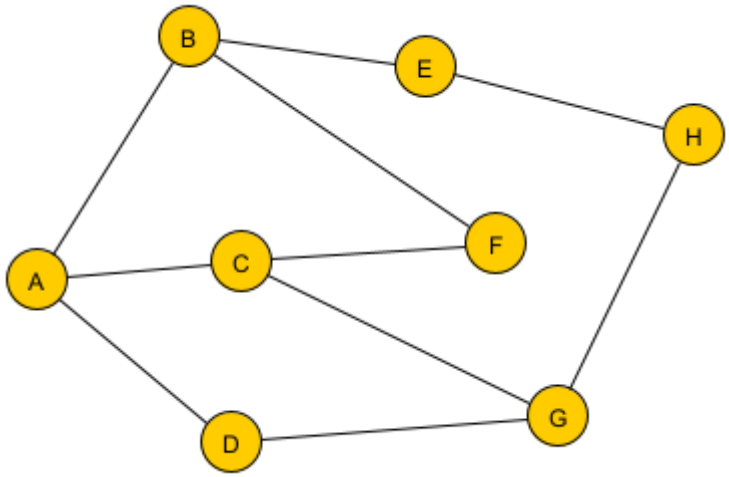
(3) Vytvoří se seznam sousedních vrcholů tohoto vrcholu (pouze ty, které ještě nebyly označeny jako navštívené) a ty se postupně vloží do zásobníku.

(4) Kroky č. 2 a č. 3 se opakují, dokud není zásobník prázdný. [11]

Hlavním rozdílem mezi rekurzivní a iterativní implementací DFS je strategie, jak se vybírají a zpracovávají vrcholy. V iterativní implementaci DFS se obvykle zpracovává poslední potomek aktuálního vrcholu, zatímco v rekurzivní implementaci se zpracovává první potomek aktuálního vrcholu. Tento rozdíl se projevuje v okamžiku, kdy se navštíví nevyzkoumaný vrchol s více potomky. V rekurzivní implementaci se nejdříve zpracuje potomek s nejnižším indexem a následně se volá rekurzivní funkce pro další potomky, zatímco v iterativní implementaci se do zásobníku vkládají všechny potomky a postupně se zpracovávají. [15]

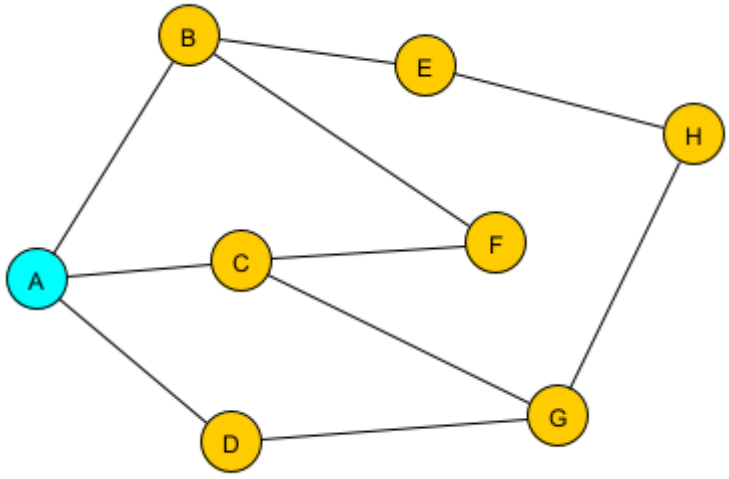
5.1.2 Ukázka DFS na příkladu

Prohlídku grafu do šířky si ukážeme na následujícím grafu:



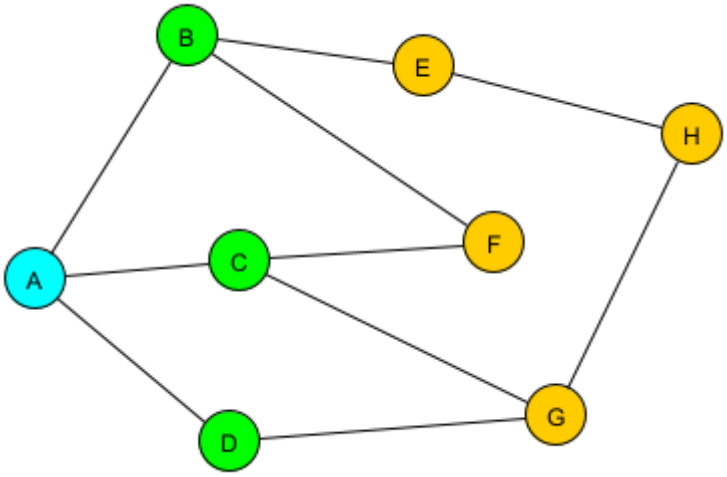
Obrázek 12: Graf pro ukázku DFS

Prvním krokem je výběr vrcholu, ze kterého budeme dále postupovat. Vybereme tedy například vrchol A a označíme ho jako navštívený.



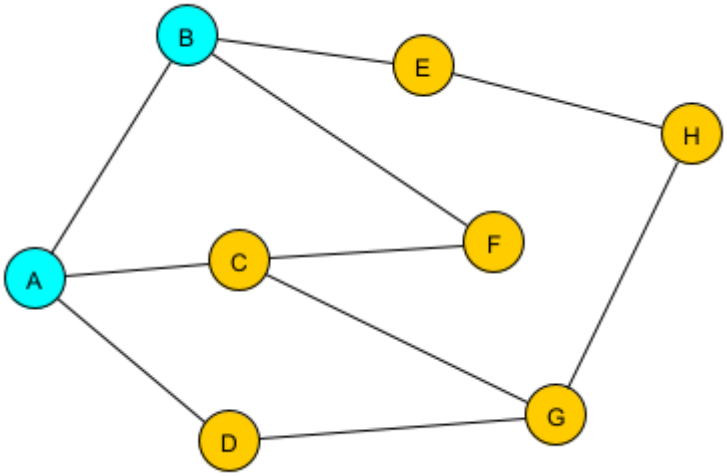
Obrázek 13: První krok DFS

V dalším kroku projdeme všechny sousední vrcholy vrcholu A. Těmi jsou v tomto případě vrcholy B, C a D.



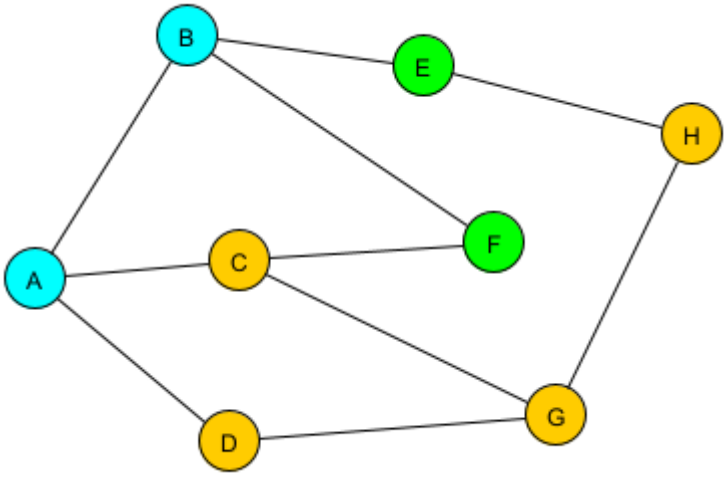
Obrázek 14: Druhý krok DFS

Následně vybereme libovolný sousední vrchol, zvolme třeba vrchol B, a rekurzivně pro něj zavoláme DFS (označíme ho jako navštívený).



Obrázek 15: Třetí krok DFS

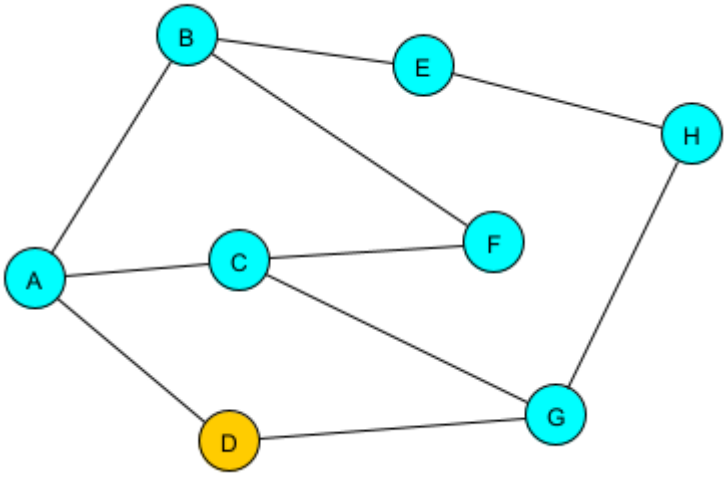
Ve čtvrtém kroku opět projdeme všechny sousední vrcholy naposledy navštíveného vrcholu.



Obrázek 16: Čtvrtý krok DFS

Tyto dva kroky, respektive označení vrcholu jako navštívený a následné hledání jeho sousedních vrcholů opakujeme do chvíle, dokud nenastane situace, že daný vrchol již nemá žádné sousední vrcholy, které by bylo možné navštívit. V tomto případě je potřeba se vrátit na předchozí vrchol a opět opakovat předchozí dva kroky.

Tato situace může nastat například v kroku číslo 13.



Obrázek 17: Třináctý krok DFS

Z obrázku je patrné, že většina vrcholů grafu byla již označena jako navštívené a posledním navštíveným vrcholem je vrchol F, který nemá žádné další nenavštívené sousedy. Proto musíme vrátit DFS o krok zpět na vrchol C a zkontrolovat, zda tento vrchol má nějaké nenavštívené sousedy, kteří by mohli být navštíveni. Pokud ne, vrátíme se opět o krok zpět na vrchol G. Zde

jsme objevili nenavštívený sousední vrchol, který označíme jako navštívený a poté se vrátíme zpět na vrchol G. Tento postup opakujeme, dokud se nedostaneme do situace, kdy se nacházíme na počátečním vrcholu, ze kterého jsme algoritmus zahájili. Výsledkem rekurzivní DFS v tomto příkladu by mohlo být procházení uzlů v následujícím pořadí: A → B → E → H → G → C → F → D.

5.1.3 Využití

Hloubkové prohledávání je preferovaným přístupem k řešení mnoha problémů v oblastech jako je plánování úloh, serializace dat nebo mapování tras. Tento algoritmus se také používá jako podprogram v jiných složitějších algoritmech, jako například v Hopcroft-Karpově algoritmu pro hledání shody v grafu nebo ve Ford-Fulkersonově algoritmu. Dále je užitečný při procházení stromů a grafů pro řešení problémů, jako je obchodní cestující, hledání nejkratší cesty, detekce cyklů v grafech a v topologickém třídění. Tento algoritmus se také používá k řešení hádanek s jediným řešením, jako jsou bludiště a sudoku, a může být využit pro analýzu sítí, jako například testování, zda je graf bipartitní. DFS je také klíčovým prvkem algoritmů pro nalezení různých druhů komponent grafu nebo třeba pro nalezení kostry neorientovaného grafu. [16] [19]

5.2 BFS

BFS je algoritmus, kdy je třeba začít procházet od vybraného uzlu (zdrojového nebo výchozího uzlu) a procházet graf po vrstvách, tedy prozkoumat sousední uzly (uzly, které jsou přímo připojeny ke zdrojovému uzlu). Poté je třeba postupovat směrem k sousedním uzlům další úrovně. Jak je patrné ze samotného názvu algoritmu, tak je třeba procházet graf tzv. „po šířce“. [17]

5.2.1 Průběh algoritmu

Stejně jako u DFS dělíme vrcholy do dvou kategorií na vrcholy navštívené a nenavštívené. Účel algoritmu je také stejný jako u předešlého.

Algoritmus funguje následovně:

(1) Do fronty se vloží libovolný vrchol.

(2) Vrchol se odebere z fronty a označí se jako navštívený (jakým způsobem se pracuje s frontou bylo vysvětleno podrobněji v kapitole č. 4).

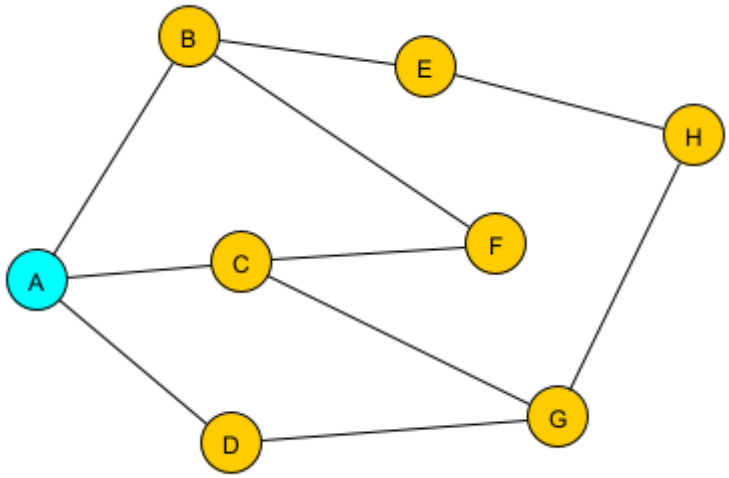
(3) Vytvoří se seznam sousedních vrcholů tohoto vrcholu (pouze ty, které ještě nebyly označeny jako navštívené) a ty se postupně vloží do fronty.

(4) Kroky č. 2 a č. 3 se opakují, dokud není fronta prázdná. [11]

5.2.2 Ukázka BFS na příkladu

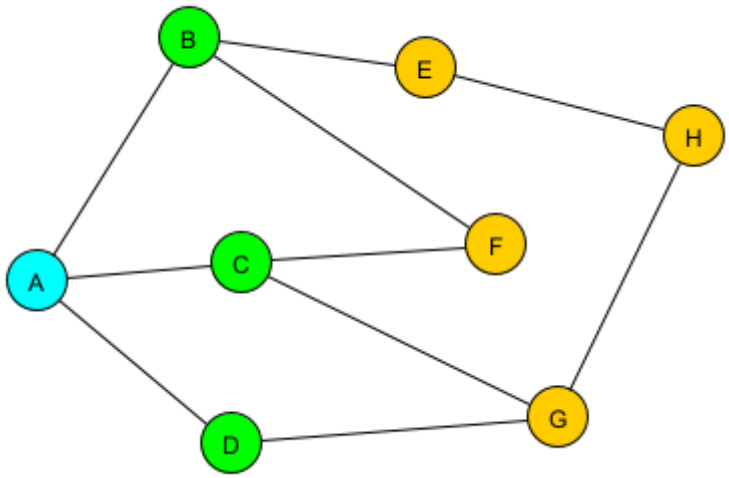
Příklad si ukážeme na stejném grafu, jako byla ukázka algoritmu DFS, tedy obrázku č. 12.

Prvním krokem je výběr libovolného vrcholu, ze kterého začneme graf procházet. Vybereme například vrchol A a vložíme ho do fronty.



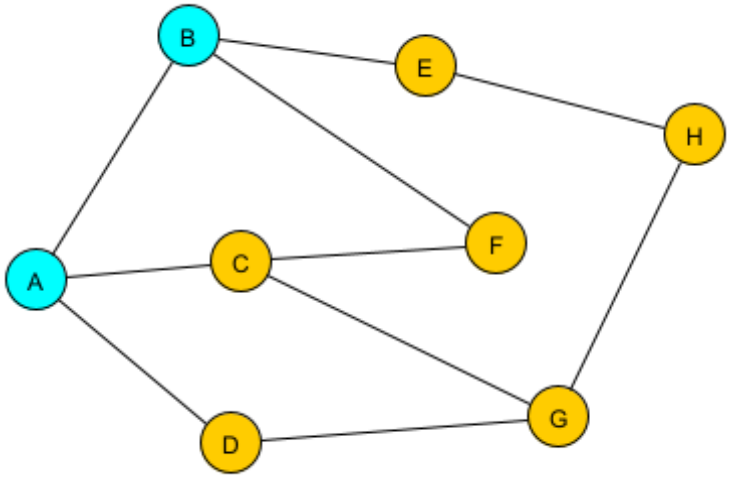
Obrázek 18: První krok BFS

Následujícím krokem je vložení všech sousedních vrcholů, které ještě nebyly označeny jako navštívené, do fronty. Těmito vrcholy jsou vrchol B, C a D.



Obrázek 19: Druhý krok BFS

V dalším kroku vybereme vrchol z fronty a označíme ho jako navštívený.



Obrázek 20: Třetí krok BFS

Následně opět přidáme sousední nenavštívené vrcholy do fronty. Tento proces opakujeme, dokud není fronta prázdná. Výsledkem BFS v tomto příkladu by mohlo být procházení uzlů v následujícím pořadí: A → B → C → D → E→ F → G → H.

5.2.3 Využití

Prohledávání do šířky je účinnou technikou pro hledání komponent souvislosti a kostry grafu, stejně jako pro hledání nejkratší cesty mezi dvěma uzly. Tato technika je také užitečná pro testování, zdali je graf bipartitní (rozložitelný na dva disjunktní podgrafy) a pro výpočet maximálního toku v grafu pomocí Ford-Fulkersonovy metody. [19]

5.3 Detekce kružnic

Jak již vyplývá z názvu samotného algoritmu, jeho účelem je najít v grafu kružnice, tedy uzavřené cesty v grafu, které se vrací do svého počátečního vrcholu. Existuje několik algoritmů, které mohou být použity k detekci kružnic. Nejčastěji se však používá algoritmus DFS, případně BFS, oba tyto algoritmy byly popsány a vysvětleny v předchozích podkapitolách. [20]

5.3.1 Průběh algoritmu

Algoritmus funguje následovně:

(1) Vybereme libovolný neoznačený vrchol a označíme ho jako navštívený.

(2) Pro každého souseda vrcholu:

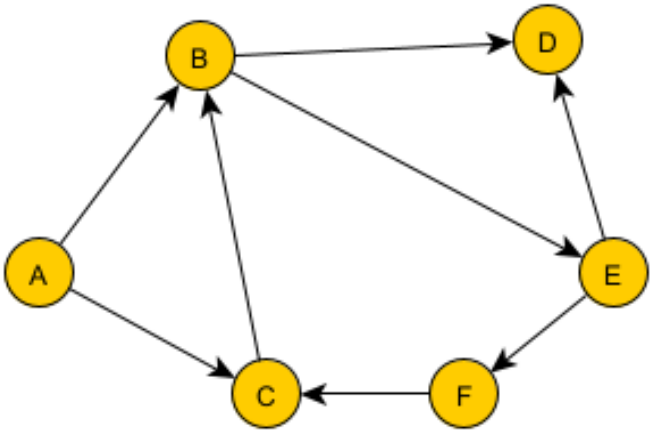
a) Pokud sousední vrchol není označen jako navštívený, přidáme ho na cestu a pokračujeme rekurzivním voláním funkce pro tento sousední vrchol.

b) Pokud je sousední vrchol označen jako navštívený a není předchůdcem aktuálního vrcholu, znamená to, že jsme nalezli kružnici v grafu a můžeme algoritmus ukončit.

(3) Odstraňujeme aktuální vrchol z cesty.

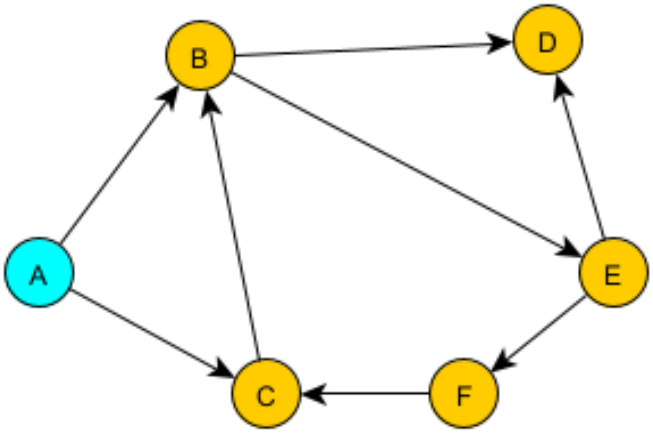
5.3.2 Ukázka detekce kružnic na příkladu

Detekci kružnic si ukážeme na následujícím grafu:



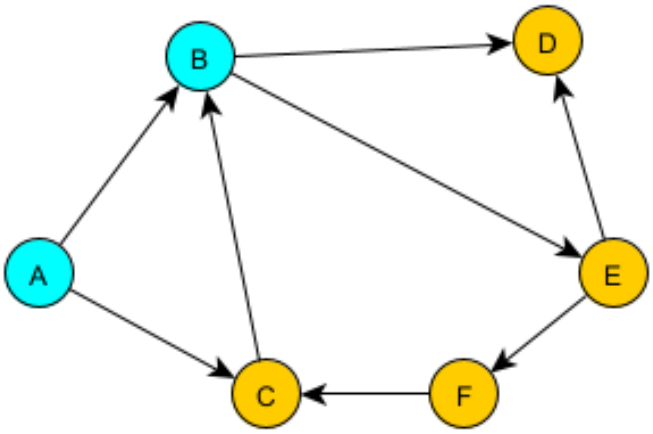
Obrázek 21: Graf pro ukázku detekce kružnic

Prvním krokem je výběr libovolného vrcholu, ze kterého začneme graf procházet. Vybereme například vrchol A a přidáme ho na cestu.



Obrázek 22: První krok detekce kružnic

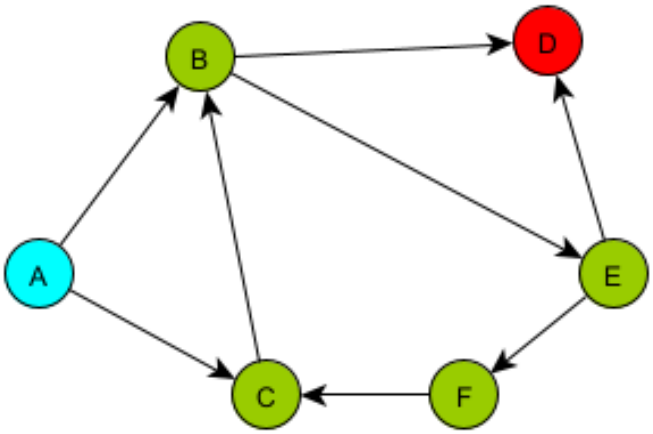
V dalším kroku vybereme sousední vrchol a opět ho přidáme na cestu.



Obrázek 23: Druhý krok detekce kružnic

Algoritmus detekce kružnic, který používáme, je založen na DFS. Postupně projdeme jednotlivé vrcholy grafu, přičemž přidáváme každý vrchol do cesty. Pokud u některého vrcholu není možné pokračovat dále, vrátíme se zpět.

Hlavní rozdíl oproti standardnímu DFS algoritmu spočívá v tom, že u každého vloženého vrcholu zároveň kontrolujeme, zda nějaký z jeho sousedů není vrcholem, který se aktuálně nachází na zkoumané cestě. Pokud ano, znamená to, že jsme objevili kružnici v grafu.



Obrázek 24: Poslední krok detekce kružnic

Jak je patrné z obrázku, vrcholy grafu jsme procházeli v následujícím pořadí: A → B → D → E → F → C, přičemž z vrcholu D se bylo nutno vrátit zpět, jelikož již nebylo možné nikam

pokračovat. Následně při dosažení vrcholu C jsme zjistili, že jeho sousední vrchol se nachází v právě zkoumané cestě, a tudíž se v grafu nachází kružnice B → E → F → C.

5.3.3 Využití

Detekce kružnic je důležitou technikou v teorii grafů, která nachází své uplatnění v mnoha oblastech. Často se používá v distribuovaných algoritmech založených na zprávách, kde může pomoci při detekci a opravě chyb v komunikaci mezi uzly. Dalším využitím je v souběžných systémech, kde se detekce kružnic používá k detekci deadlocků a dalších problémů souvisejících se vzájemným vyloučením. V kryptografii se detekce kružnic používá k určení klíčů zprávy, které mohou mapovat stejnou zprávu na stejnou zašifrovanou hodnotu. Tento mechanismus může být využit k účinnému a bezpečnému šifrování komunikace a ochraně soukromí dat.

5.4 Ford-Fulkerson

Ford-Fulkersonův algoritmus slouží k výpočtu maximálního průtoku v síťovém grafu tím, že postupně upravuje tok a zvyšuje ho tak, aby splňoval omezení kapacity hran. [26]

Důležitým pojmem pro pochopení algoritmu je zbytkový graf, který se vytváří v průběhu výpočtu. Jedná se o graf, který obsahuje informace o tom, jak lze tok podél každé hrany upravit. Každá hrana v síťovém grafu je nahrazena až dvěma novými hranami: dopřednou hranou se stejným směrem, která znamená, o kolik lze tok zvýšit, a zpětnou hranou uchovávající, o kolik lze tok snížit. [26]

Algoritmus začíná s prázdným tokem a poté opakovaně hledá augmentační cesty v zbytkovém grafu od zdroje k cíli. Augmentační cesta je cesta, po které lze ještě posílat tok, a to tak, aby se nasytila jedna hrana s nejnižší kapacitou. Tímto způsobem je zachována platnost kontraktů na tok a zvyšuje se celkový průtok. [26]

Ford-Fulkersonova metoda neuvádí, jakým způsobem nalézt augmentační cesty, a existují různé strategie pro jejich hledání. Volba strategie má však významný vliv na dobu běhu algoritmu. [26]

Definice 9*: **Zlepšující cesta** je cesta neorientovaná cesta $P = \{z = v_0, v_1, \ldots, v_{n-1}, u = v_n\}$ taková, že pro každou dvojici sousedních vrcholů v_i a v_{i+1} spojených hranou h_i platí:*

(1) $p(h_i) = [v_i, v_{i+1}] \Rightarrow y(h_i) < o(h_i)$,

(2) $p(h_i) = [v_{i+1}, v_i] \Rightarrow y(h_i) > 0$.

Množinu hran na zlepšující cestě P splňující podmínku (1) označíme X^{+}_{P}, množinu hran na zlepšující cestě P splňující podmínku (2) označíme X^{-}_{P}. [18]

5.4.1 Průběh algoritmu

Algoritmus funguje následovně:

(1) Nastavíme tok všech hran v síti na 0.

(2) Zvolíme zdrojový uzel S a cílový uzel T.

(3) Nalezneme úplný tok:

a) Najdeme cestu mezi uzly S a T

b) Vypočteme kapacitu cesty, což je nejmenší kapacita hrany v této cestě.

c) Tuto hodnotu přičteme každé hraně na této cestě.

d) Hrana, jejíž tok a kapacita jsou rovny je nasycena a nemůže být použita v žádné další cestě.

e) Tento krok opakujeme, dokud jsou v síti dostupné cesty mezi S a T.

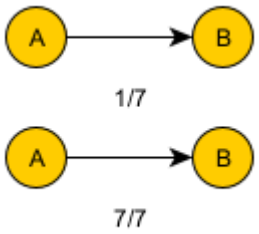
(4) Nalezneme maximální tok:

a) Najdeme zlepšující cestu mezi S a T (hrany lze procházet i zpětně, pokud tok dané hrany není roven 0 viz Obrázek č. 25).

b) Vypočteme kapacitu cesty, což je minimální kapacita hrany v této cestě, případně tok hrany, pokud ji procházíme zpětně.

c) Tuto hodnotu přičteme každé hraně na této cestě, respektive odečteme, pokud je hrana procházena zpětně.

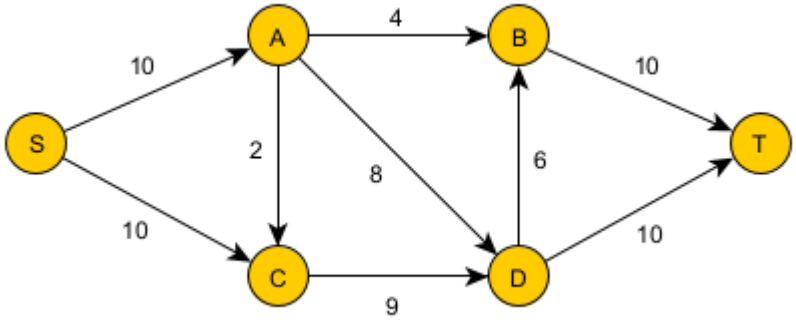
d) Tento krok opakujeme, dokud jsou v síti dostupné zlepšující cesty mezi S a T. Pokud žádné nejsou, tak jsme našli maximální tok, ten je součtem toků hran, které vedou přímo do T. [18]



Obrázek 25: Hrany umožňující zpětný průchod

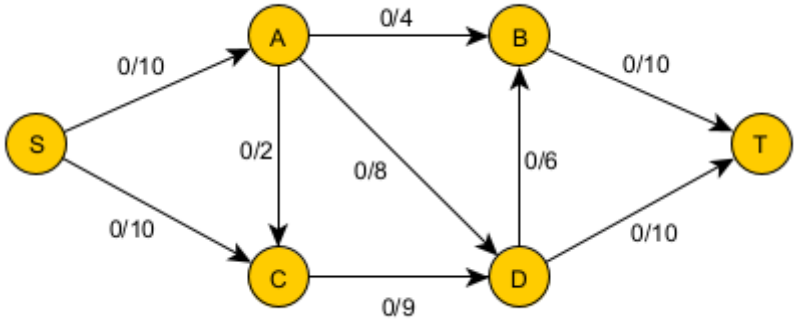
5.4.2 Ukázka Ford-Fulkersonova algoritmu na příkladu

Algoritmus si popíšeme na následující síti:



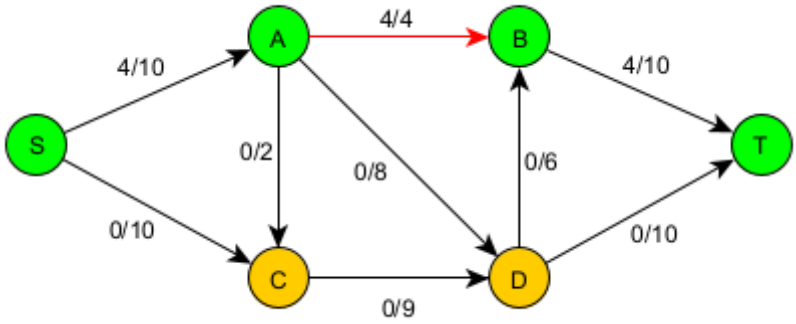
Obrázek 26: Graf pro ukázku Ford-Fulkersonova algoritmu

Prvním krokem je nastavení toku všech hran v síti na hodnotu 0.



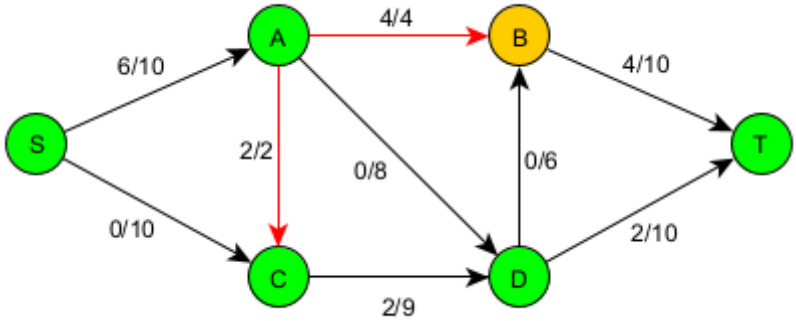
Obrázek 27: První krok Ford-Fulkersonova algoritmu

V následujícím kroku vybereme jednu z možných cest, zvolíme například S → A → B → T a upravíme tok všech hran na této cestě o hodnotu 4. Hrana mezi vrcholy A a B je nyní nasycená, tzn. že již nemůže být součástí další cesty.



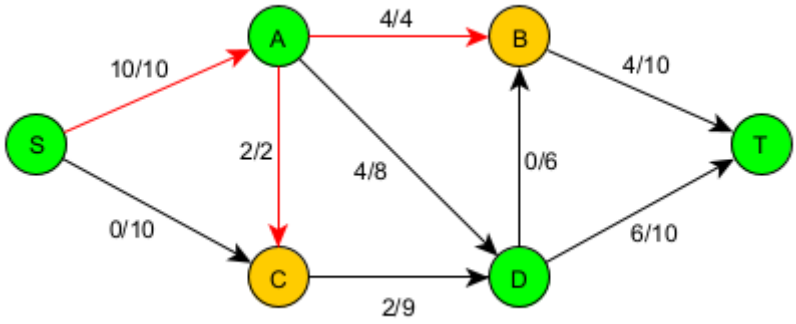
Obrázek 28: Druhý krok Ford-Fulkersonova algoritmu

V třetím kroku vybereme další cestu S → A → C → D → T a opět upravíme tok všech hran na této cestě tak, aby alespoň jedna hrana byla nasycena.

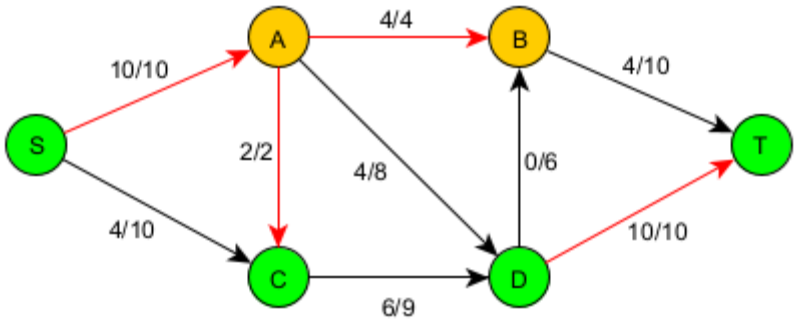


Obrázek 29: Třetí krok Ford-Fulkersonova algoritmu

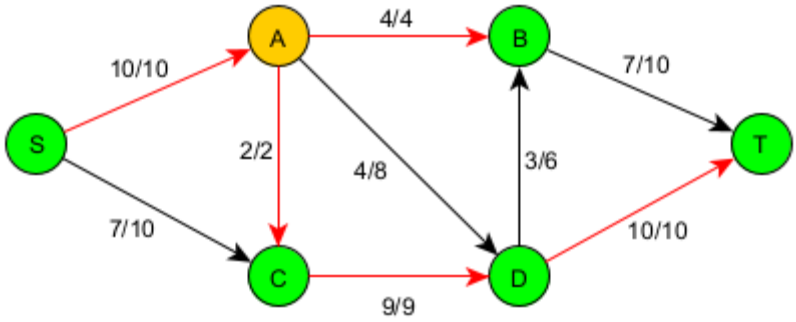
Tímto způsobem pokračujeme, dokud jsou v síti dostupné cesty mezi S a T, u kterých lze aktualizovat tok.



Obrázek 30: Čtvrtý krok Ford-Fulkersonova algoritmu



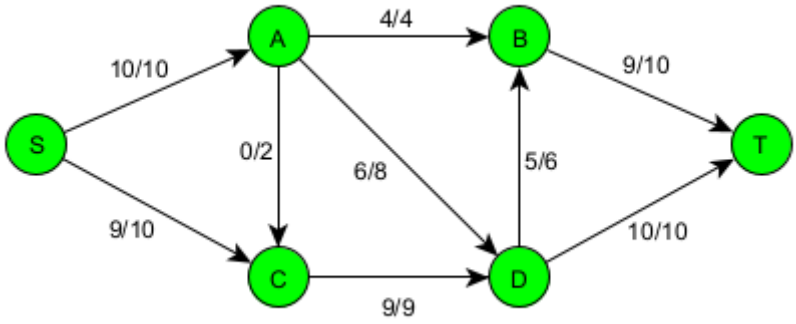
Obrázek 31: Pátý krok Ford-Fulkersonova algoritmu



Obrázek 32: Šestý krok Ford-Fulkersonova algoritmu

Po dokončení kroku č. 6, jsme nalezli úplný tok, který ale v tomto případě stále není tokem maximálním. Z toho důvodu je potřeba hledat zlepšující cesty, abychom tento tok vylepšili. V tomto příkladu se nachází pouze jedna zlepšující cesta, a to konkrétně S → C → A → D → B → T. Na této cestě upravíme tok stejně jako v předchozích krocích s tím rozdílem, že hranu

mezi vrcholy C a A procházíme zpětně, tudíž zde tok odečteme. Další zlepšující cesty v síti již nejsou, tudíž maximální tok v síti je 19 (9+10).



Obrázek 33: Poslední krok Ford-Fulkersonova algoritmu

5.4.3 Využití

Ford-Fulkersonův algoritmus je často využíván v různých oblastech a má široké spektrum využití. [27]

Příkladem může být hledání vrcholově nespojitých cest v grafu, které se dosahuje rozdělením každého vrcholu na příchozí a odchozí vrchol, které jsou spojeny hranou s jednotkovou kapacitou toku, zatímco ostatním hranám je přiřazena nekonečná kapacita. Maximální tok zdroje do propadu se rovná počtu vrcholově nesouvislých cest v grafu. [27]

Dalším příkladem je segmentace obrazu. V tomto případě se obraz převede na graf toku, kde každý pixel je vrcholem a který spojuje sousední pixely-vrcholy v systému 4 sousedů. Existují další 2 vrcholy, které jsou zdrojem a stokem. Poté se na grafu spustí algoritmus s cílem najít minimální řez, který vytváří optimální segmentaci. [27]

Algoritmus může být využit i v oblasti plánování leteckých linek k nalezení proveditelného oběhu v síti, což umožňuje naplánovat n letů s použitím nejvýše k letadel. Graf toku se vytváří tak, že máme hranu z jednoho letiště do druhého, pokud stejné letadlo může obsloužit oba lety. [27]

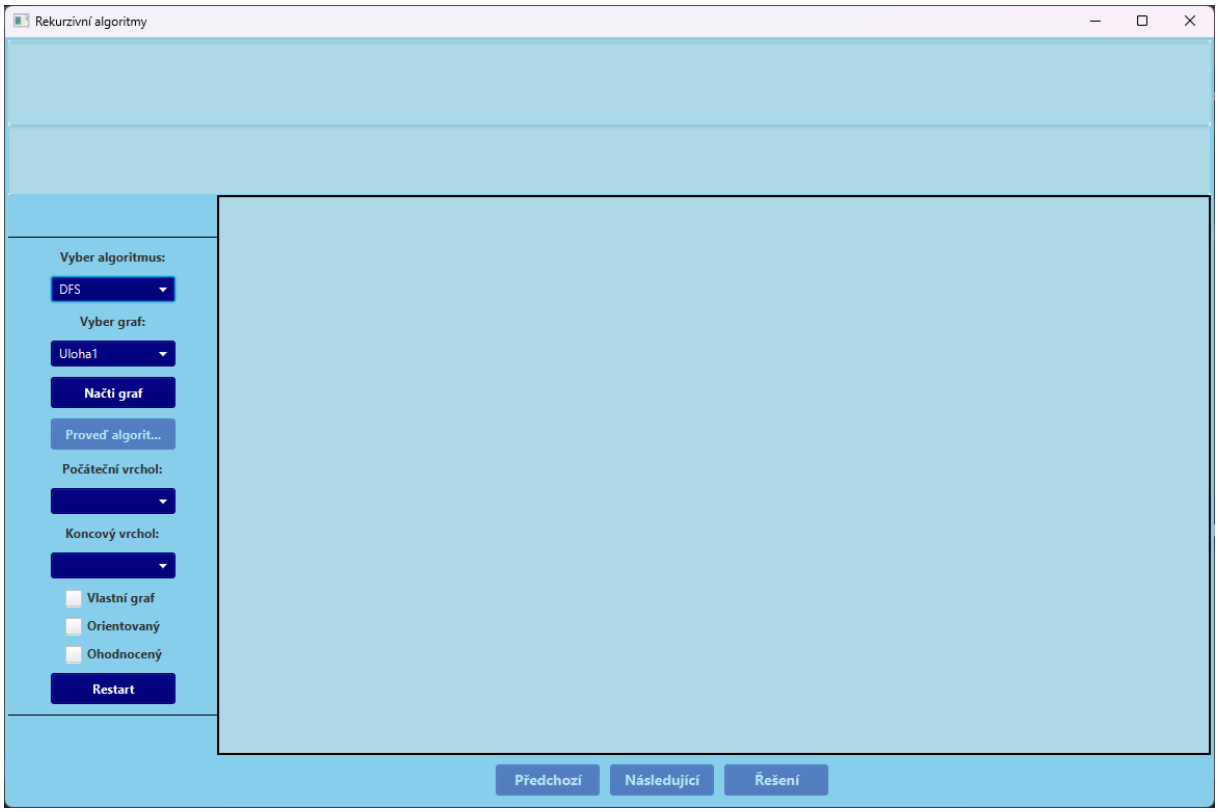
6 NÁVOD K POUŽITÍ APLIKACE

6.1 Potřebné prerekvizity

Spuštění samotné aplikace je poměrně jednoduché, stačí pouze otevřít spustitelný soubor v přiložením zip archivu a zároveň mít nainstalované JRE ve verzi 1.8. Na vyšších verzích by měla aplikace také fungovat bez nutnosti úprav nebo re-kompilace, nicméně, mohou existovat určité změny v chování Java prostředí mezi jednotlivými verzemi, což může ovlivnit výkon aplikace nebo její schopnost spouštět určité funkce. Proto doporučují spouštět aplikaci s výše zmiňovanou verzí JRE.

6.2 Hlavní okno

Samotná aplikace se skládá z jednoho hlavního okna, ve kterém jsou umístěny veškeré funkcionality a na kterém probíhají všechny interakce s uživatelem.



Obrázek 34: Hlavní okno po spuštění aplikace

Hlavní okno se skládá ze 4 hlavních částí:

- horní část,
- levý panel,

- spodní panel,
- hlavní oblast.

6.2.1 Horní část

V horní části okna se nachází dvě textová pole (vrchní a spodní). Pole umístěné na samém vrcholu okna slouží pro zobrazování textového postupu při procházení jednotlivými algoritmy pomocí krokování, kterým aplikace disponuje. Textové okno umístěné níže slouží pro zobrazení finálního řešení právě zkoumaného algoritmu.

Obrázek 35: Ukázka funkce textových polí v průběhu algoritmu Ford-Fulkerson

6.2.2 Levý panel

Levý panel disponuje několika výběrovými tlačítky pro nakonfigurování konkrétní úlohy, která bude prováděna. Jak je již viditelné z předchozího obrázku, tak první výběrové tlačítko slouží pro výběr samotného algoritmu.

Na výběr jsou zde následující algoritmy:

- prohlídka do hloubky,
- prohlídka do šířky,
- detekce kružnic
- Ford-Fulkerson.

Po výběru algoritmu si uživatel z dalšího výběrového pole zvolí úlohu, která v tomto případě představuje graf, který se následně po stisknutí tlačítka „Načti graf“ zobrazí v hlavní oblasti ve středu hlavního okna.

Po načtení grafu se do výběrových boxů ve spodní části levého panelu načtou všechny vrcholy, které daný graf obsahuje. Výběrový box „Počáteční vrchol“ slouží pro všechny dostupné algoritmy v aplikaci kromě detekce kružnic. Box „Koncový vrchol“ slouží pouze pro algoritmus Ford-Fulkerson.

Tlačítko „Proveď algoritmus“ vykoná vybraný algoritmus na zvolené úloze a zároveň „odemkne“ další funkce pro postupný průchod algoritmu nebo zobrazení řešení, tato tlačítka se nacházejí ve spodním panelu aplikace.

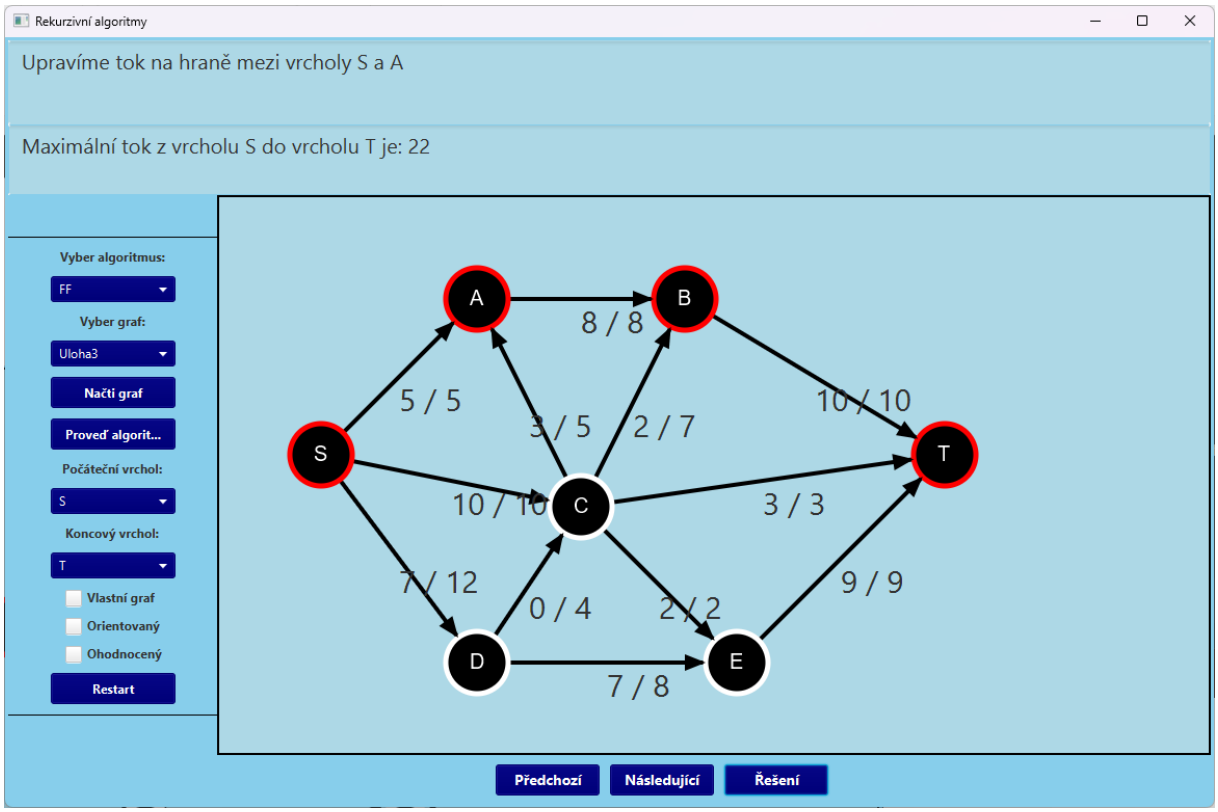
V dolní části panelu se nacházejí tři checkboxy (výběrová pole). První checkbox se nazývá „Vlastní graf“. Pokud uživatel tento checkbox zvolí, může vytvořit vlastní graf klikáním na vyznačené místo v hlavní oblasti. Uživatel může přidávat vrcholy na obrazovku pomocí pravého tlačítka myši a vytvářet hrany mezi vrcholy kliknutím na oba vrcholy. Výběr prvního vrcholu je možné zrušit stisknutím klávesy „Escape“ Druhý checkbox se nazývá "Orientovaný". Pokud uživatel zvolí tento checkbox, graf bude orientovaný, tzn. že hrany budou mít specifický směr. Třetí checkbox se nazývá "Ohodnocený". Pokud uživatel zvolí tento checkbox, hrany vytvořené uživatelem budou mít hodnotu. Pokud uživatel vytvoří ohodnocenou hranu, zobrazí se mu dialogové okno, které umožní zadání hodnoty této hrany.

6.2.3 Spodní panel

Spodní panel obsahuje tři tlačítka, z nich první dvě označená jako „Předchozí“ a „Následující“ slouží pro procházení jednotlivých kroků algoritmu. Pokud však uživatel nechce daný algoritmus procházet krok po kroku a rovnou zjistit výsledek, má zde k dispozici tlačítko „Řešení“. Postup i řešení se zobrazují v již zmiňovaných textových polích v horní části aplikace.

6.2.4 Hlavní oblast

Poslední a zároveň nejdůležitější částí je oblast, jejíž hlavní účel je zobrazení vybraného grafu. Pokud se uživatel rozhodne využít krokování, respektive tlačítek „Předchozí“ a „Následující“, tak se postup nepromítne pouze do textových polí, ale zároveň je v grafu právě zkoumaná část barevně označena. Aplikace také umožňuje s jednotlivými vrcholy manipulovat pomocí tahu myši.



Obrázek 36: Ukázka průběhu algoritmu Ford-Fulkerson

Na obrázku můžeme vidět průběh algoritmu Ford-Fulkerson, ve kterém je aktuálně vybrána první cesta z počátečního vrcholu S do koncového vrcholu T. V horní části aplikace je také vidět, že je aktuálně využíváno krokování (procházení postupu algoritmu) a zároveň si uživatel nechal vypsat i výsledné řešení algoritmu.

7 PROGRAMOVÁ ČÁST

7.1 Balíčky

Zdrojový kód je rozdělen do čtyřech balíčků: mainscreenfx, algoritmy, krokovani, a struktury.

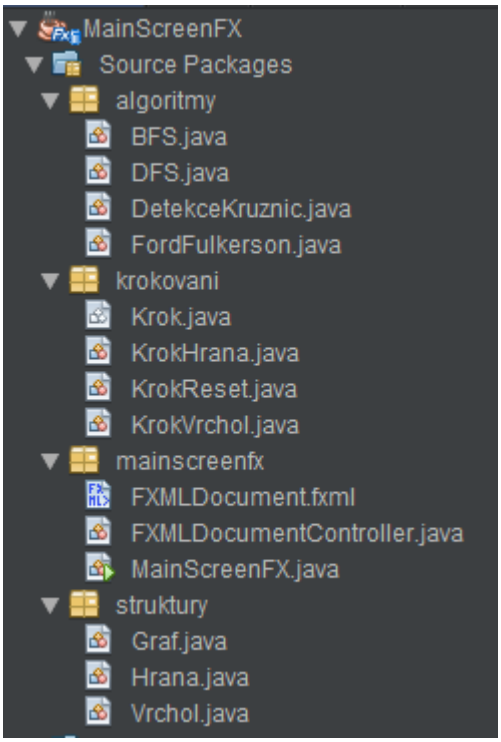
V balíčku mainscreenfx se nachází FXML soubor *FXMLDocument*, který zastává funkci grafické stránky celého hlavního okna aplikace. Společně s tímto souborem je zde také nosná třída celého projektu *FXMLDocumentController*, která se stará o veškerou funkcionalitu prvků hlavního okna. Z toho důvodu je tato třída také nejvíce rozsáhlá.

Balíček algoritmy obsahuje jednotlivé třídy pro každý algoritmus, který je obsažen v této aplikaci, tzn. třídy: *BFS, DFS, DetekceKruznic a FordFulkerson.*

Ve třetím balíčku s názvem krokovani se nachází rozhraní Krok, které je implementováno všemi ostatní třídy v tomto baličku, těmi jsou: *KrokHrana, KrokVrchol* a *KrokReset*. Právě díky těmto třídám je uživatel schopen procházet postup jednotlivých algoritmů, jak směrem vpřed, tak i směrem zpět.

Posledním balíček obsahuje jednotlivé struktury, které byly potřeba k vytvoření grafu. Mezi tyto třídy patří:

- *Graf* – uchovává seznam všech vrcholů, které daný graf obsahuj a dále například zda je orientovaný, ohodnocený apod.,
- *Vrchol* – uchovává základní informace o daném vrcholu v grafu jako název, umístění a seznam všech hran, které z tohoto vrcholu vycházejí,
- *Hrana* – uchovává si informace o počátečním a koncovém vrcholu, zda je hrana orientovaná a jakou má případně kapacitu nebo tok.



Obrázek 37: Rozložení tříd mezi balíčky

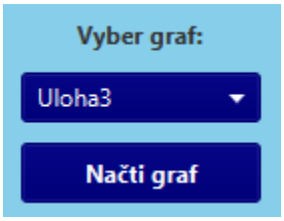
Celkově jsou tyto balíčky důležité pro správnou organizaci kódu a rozdělení zodpovědností mezi jednotlivé části aplikace. Díky tomuto rozdělení může být kód aplikace lépe udržovatelný, přehledný a rozšiřitelný.

7.2 Zobrazení grafu

Existují dva způsoby, jakým je možné v aplikaci graf zobrazit.

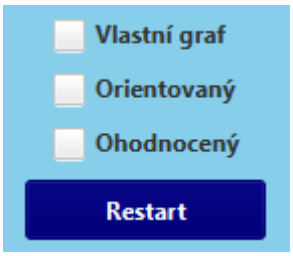
První možností je výběr dané úlohy z výběrového menu a stisknutím tlačítka "Načti graf", čímž se graf načte z textového souboru. Soubor obsahuje všechny potřebné informace grafu pro jeho správné zobrazení. Jsou zde seznamy vrcholů a hran, které graf obsahuje. Každý vrchol je identifikován unikátním označením, které ho odlišuje od ostatních vrcholů. Každá hrana obsahuje informace o tom, které dva vrcholy spojuje a zda je hrana orientovaná nebo neorientovaná. Pokud je graf ohodnocený, v souboru jsou také uloženy hodnoty pro každou hranu. Další důležitou informací v souboru jsou souřadnice vrcholů pro jejich vykreslení. Tyto souřadnice určují, kde se na obrazovce nachází každý vrchol. Díky nim lze graf zobrazit na obrazovce bez nutnosti manuálního pohybu vrcholů. Nicméně v aplikaci je implementována funkce pohybu vrcholů, takže s nimi může po načtení uživatel libovolně manipulovat. Soubor obsahuje všechny potřebné informace pro zobrazení grafu v aplikaci. Díky tomu je možné

s grafem pracovat a interaktivně ho upravovat, což umožňuje uživatelům lépe porozumět datům, které graf reprezentuje.



Obrázek 38: První možnost zobrazení grafu

Druhou možností je nechat uživatele vytvořit si svůj vlastní graf. Stačí zakliknout checkbox „Vlastní graf“ a případně poté zaškrtnout checkbox „Orientovaný“ nebo „Ohodnocený“ pro specifikaci hran. Uživatel graf vytváří klikáním na plochu pravým tlačítkem myši. Pokud chce vytvořit hranu mezi dvěma vrcholy, stačí stisknout pravým tlačítkem na prvním vrcholu a poté pravým tlačítkem na druhém vrcholu. Pokud si uživatel rozmyslí výběr prvního vrcholu, ze kterého má hrana vycházet, stačí zmáčknout klávesu „Escape“.



Obrázek 39: Druhá možnost vytvoření grafu

Následující část kódu má funkci načíst vrcholy grafu, jejich souřadnice a určit, zda se jedná o orientovaný nebo případně i ohodnocený graf:

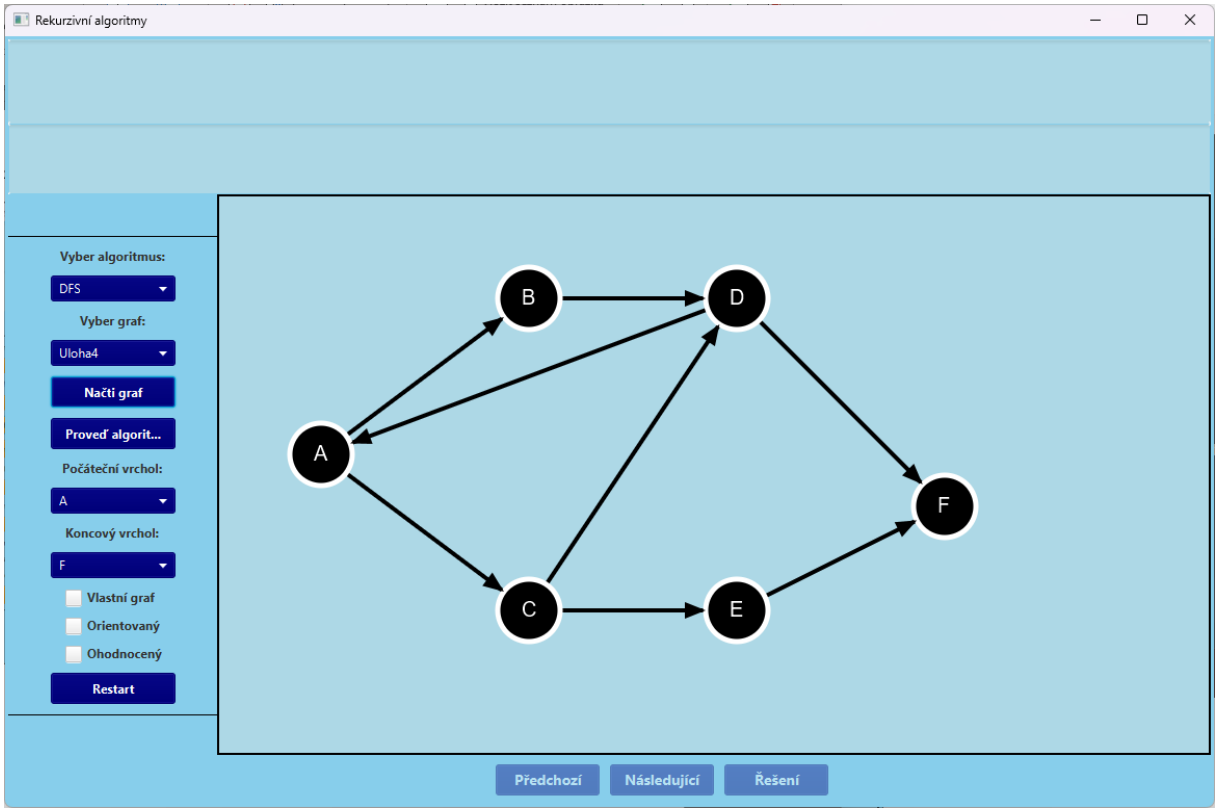
```
private Graf vytvorGraf() throws FileNotFoundException, IOException {
        Graf = new Graf();
        InputStream is = getClass().getClassLoader().
       getResourceAsStream(cbVyberGrafu.getValue() + ".txt");
        BufferedReader br = new BufferedReader(new InputStreamReader(is));

        String orientovany = br.readLine(); //zjisti zda je orientovany
        if (orientovany.equals("Orientovany")) {
            graf.setOrientovany(true);
        }
        String ohodnoceny = br.readLine(); //zjisti zda je ohodnoceny
        if (ohodnoceny.equals("Ohodnoceny")) {
            graf.setOhodnoceny(true);
        }
```

```
        String temp[] = br.readLine().split(" "); // nacte vsechny vrcholy
        for (String string : temp) {
            graf.pridejVrchol(new Vrchol(string));
        }

        String sourX[] = br.readLine().split(" "); // nacte souradniceX
        for (String : sourX) {
            souradniceX.add(Integer.parseInt(string));
        }

        String sourY[] = br.readLine().split(" "); // nacte souradniceY
        for (String string : sourY) {
            souradniceY.add(Integer.parseInt(string));
        }
```

Po načtení těchto základních informací je také potřeba pospojovat vrcholy, respektive přiřadit jednotlivým vrcholům hrany.

```
private void priradHrany(BufferedReader br, Graf graf) throws
NumberFormatException, IOException {
        String line;
        while ((line = br.readLine()) != null) {
            String strArr[] = line.split(" ");
            int delkaPole = strArr.length;
            Vrchol prvniVrchol = graf.getVrchol(strArr[0]);
            for (int i = 1; i < delkaPole; i++) {
                Vrchol druhyVrchol = graf.getVrchol(strArr[i]);
                Hrana novaHrana = new Hrana(prvniVrchol, druhyVrchol,
graf.jeOrientovany());
                if (graf.jeOhodnoceny()) {
                    i++;
                    novaHrana.setKapacita(Integer.parseInt(strArr[i]));
                }
                if (!druhyVrchol.getHrany().contains(novaHrana)
&& !graf.jeOrientovany()) {
                    prvniVrchol.pridejHranu(novaHrana);
                    druhyVrchol.pridejHranu(novaHrana);
                } else {
                    prvniVrchol.pridejHranu(novaHrana);
                }
            }
        }
    }
```

Zde je ukázka, jak vypadá orientovaný neohodnocený graf po načtení do aplikace:



Obrázek 40: Ukázka grafu po načtení

8 POUŽITÉ TECHNOLOGIE

8.1 Java

Java je programovací jazyk a počítačová platforma, kterou poprvé vydala společnost Sun Microsystems v roce 1995. Od skromných začátků se vyvinula tak, že dnes pohání velkou část digitálního světa a poskytuje spolehlivou platformu, na níž je postaveno mnoho služeb a aplikací. [21]

8.2 JavaFX

JavaFX je sada grafických a mediálních balíčků, která vývojářům umožňuje navrhovat, vytvářet, testovat, ladit a nasazovat bohaté klientské aplikace, které fungují konzistentně na různých platformách. [22]

Kód aplikací JavaFX je napsán jako rozhraní Java API a může odkazovat na rozhraní API z libovolné knihovny Java. Aplikace JavaFX mohou například využívat knihovny Java API pro přístup k nativním systémovým funkcím a pro připojení k serverovým middlewarovým aplikacím. [22]

Vzhled a ovládání aplikací JavaFX lze přizpůsobit. Kaskádové styly (CSS) oddělují vzhled a styl od implementace. [22]

8.3 Scene Builder – Gluon

Scene Builder je nástroj pro návrh uživatelského rozhraní pro JavaFX. [23]

8.4 NetBeans

NetBeans IDE je vývojové prostředí pro vývoj webových, podnikových, desktopových a mobilních aplikací v jazyce Java. IDE podporuje nejnovější verze JDK, Java EE a JavaFX. [24]

8.5 yEd Graph Editor

YEd Graph Editor je aplikace pro tvorbu a editaci grafů. Je to nástroj pro vizualizaci a manipulaci s daty v podobě grafů, který umožňuje uživatelům vytvářet různé typy grafů, včetně stromových, síťových, tokových a hierarchických grafů.

Nabízí uživatelsky přátelské rozhraní, které umožňuje uživatelům vytvářet a upravovat grafy intuitivním způsobem. Obsahuje také mnoho nástrojů pro přizpůsobení a rozšíření funkcionality grafů, jako jsou různé typy uzlů, hran, štítků a barev. [25]

ZÁVĚR

Cílem bakalářské práce bylo vysvětlení základních pojmů z teorie grafů a popsání vybraných algoritmů. Tyto algoritmy měly být navíc implementovány do vlastní aplikace, která poskytne uživateli užitečný nástroj pro práci s grafy. Dle mého názoru byl tento cíl splněn.

Uživatel si může buď zvolit jednu z integrovaných úloh anebo si graf může vytvořit podle svých představ. Při vývoji aplikace jsem se zaměřoval především na její funkčnost, přehlednost a jednoduchou a intuitivní obsluhu pro uživatele. Aplikace je také připravena na další rozšíření v podobě zakomponování dalších algoritmů, nebo případně i funkcionalit.

Příkladem mohu uvést umožnění vstupu uživatele do průběhu jednotlivých algoritmů, jelikož algoritmy lze vyřešit více způsoby a uživatel je pouze pozorovatel jednoho z možných řešení. Další možné vylepšení by mohlo být například přidání další možnosti pro vytvoření grafu v podobě vygenerování podle předem daných požadavků.

Přestože by aplikace mohla být obohacena o tyto funkcionality, tak si myslím, že aplikace je už teď dostatečným přínosem a splňuje požadavky, které byly stanoveny pro tuto bakalářskou práci.

POUŽITÁ LITERATURA

[1] HOLČÍK, Jiří, KOMENDA, Martin (eds.) a kol. *Matematická biologie: e-learningová učebnice* [online]. 1. vydání. Brno: Masarykova univerzita, 2014. [cit. 2023-05-01]. ISBN 978-80-210-8095-9. Dostupné z: https://portal.matematickabiologie.cz/index.php?pg=zaklady-informatiky-pro-biology--algoritmizace-a-programovani.

[2] Lessner, D., Lána, M., Podrázká Tomková, M., Haut, J. (2020). Co je to algoritmus. *Popelka – Online učebnice*. [online] Jihočeská univerzita v Českých Budějovicích, Pedagogická fakulta. [cit. 2023-05-01]. Dostupné z: https://popelka.ms.mff.cuni.cz/~lessner/mw/index.php/Učebnice/Algoritmus/Co_je_to_algoritmus.

[3] KOVÁŘ, Petr. *Úvod do Teorie grafů* [online]. Ostrava, 2014 [cit. 2023-05-01]. Dostupné z: https://mi21.vsb.cz/sites/mi21.vsb.cz/files/unit/uvod_do_teorie_grafu.pdf. Příprava přednášek a cvičení. Vysoká škola báňská – Technická univerzita Ostrava a Západočeská univerzita v Plzni.

[4] *Teorie grafů* [online]. Ostrava: Vysoká škola Báňská, 21. 6. 2006 [cit. 2023-05-01]. Dostupné z: http://books.fs.vsb.cz/SystAnal/texty/21.htm

[5] MACEK, Lukáš. *Minimální kostra grafu a její využití* [online]. Praha, 2019 [cit. 2023-05-02]. Dostupné z: https://www2.karlin.mff.cuni.cz/~tuma/Aplikace19/Prace/Lukas_Macek.pdf. Semestrální práce. Univerzita Karlova, Matematicko-fyzikální fakulta.

[6] Introduction to Recursion – Data Structure and Algorithm Tutorials. *GeeksforGeeks* [online]. Noida, 2017 [cit. 2023-05-02]. Dostupné z: https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/

[7] MATOUŠEK, Jiří. *Kapitoly z diskrétní matematiky* [online]. Praha: Karolinum, 2002 [cit. 2023-05-01]. ISBN 80-246-0084-6. Dostupné z: https://lms.umb.sk/pluginfile.php/58452/mod_lesson/page_contents/9220/Nesetril.pdf

[8] TÖPFER, Pavel. *Programování pro střední školy: rekurze* [online]. Praha: Nakladatelství Fortuna, 1998 [cit. 2023-05-01]. ISBN 80-716-8563-1. Dostupné z: https://ksvi.mff.cuni.cz/~topfer/Texty/TextRekurze.pdf

[9] Data Structure & Algorithms – Tower of Hanoi. *Tutorialspoint* [online]. Hyderabad, c2023 [cit. 2023-05-01]. Dostupné z: https://www.tutorialspoint.com/data_structures_algorithms/tower_of_hanoi.htm

[10] HORDINA, Josef. Možná řešení Hanojské věže. *Mozkolam* [online]. 2017 [cit. 2023-05-01]. Dostupné z: https://mozkolam.cz/mechanicke-hlavolamy/solitery/mozna-reseni-hanojske-veze/

[11] TÖPFER, Pavel. *Algoritmy a programovací techniky* [online]. Praha: Prometheus, 1995 [cit. 2023-05-01]. ISBN 80-858-4983-6. Dostupné z: http://mff.cz/data/ADS_PTAPT.pdf

[12] BLÁHOVÁ, Simona. Zásobník (LiFo) Fronta (FiFo). In: *SlidePlayer* [online]. c2023 [cit. 2023-05-01]. Dostupné z: https://slideplayer.cz/slide/2343147/

[13] Difference between Stack and Queue Data Structures. *GeeksforGeeks* [online]. Noida, 2019 [cit. 2023-05-01]. Dostupné z: https://www.geeksforgeeks.org/difference-between-stack-and-queue-data-structures/

[14] GARG, Prateek. Depth First Search. *HackerEarth* [online]. San Francisco, 2016 [cit. 2023-05-01]. Dostupné z: https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

[15] Depth First Search (DFS). *Programiz* [online]. Kathmandu: Parewa Labs, [2020] [cit. 2023-05-01]. Dostupné z: https://www.programiz.com/dsa/graph-dfs

[16] GUPTA, Divyansh. Depth-first Search (DFS) Algorithm. *Interview Kickstart* [online]. Santa Clara, 2023 [cit. 2023-05-01]. Dostupné z: https://www.interviewkickstart.com/learn/depth-first-search-algorithm

[17] GARG, Prateek. Breadth First Search. *HackerEarth* [online]. San Francisco, 2016 [cit. 2023-05-01]. Dostupné z: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

[18] RAK, Josef. *Teorie grafů*. Univerzita Pardubice, c2011-2020.

[19] PAVLÁSEK, Ondřej. *Grafy, grafové algoritmy a jejich využití* [online]. Brno, 2010 [cit. 2023-05-01]. Dostupné z: https://core.ac.uk/download/pdf/30284463.pdf. Bakalářská práce. Vysoké učení technické v Brně, Fakulta podnikatelská.

[20] *Grafové algoritmy* [online]. [2009] [cit. 2023-05-01]. Dostupné z: https://is.muni.cz/el/1433/podzim2009/IB111/um/grafy/grafy.pdf?lang=en

[21] What is Java technology and why do I need it? *Java* [online]. Redwood City: Oracle Corporation, c2023 [cit. 2023-05-01]. Dostupné z: https://www.java.com/en/download/help/whatis_java.html

[22] PAWLAN, Monica. What Is JavaFX? *Oracle* [online] Redwood City: Oracle Corporation. c2008-2013 [cit. 2023-05-01]. Dostupné z: https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm

[23] Scene Builder 18 release. *Gluon* [online]. Eindhoven: Gluon, 6. 4. 2022 [cit. 2023-05-01]. Dostupné z: https://gluonhq.com/scene-builder-18-release/

[24] The Smarter and Faster Way to Code. *Oracle* [online]. Redwood City: Oracle Corporation, 2018 [cit. 2023-05-01]. Dostupné z: https://www.oracle.com/cz/tools/technologies/netbeans-ide.html

[25] YEd – graph editor. *YWorks* [online]. Tübingen: yWorks, c2023 [cit. 2023-05-01]. Dostupné z: https://www.yworks.com/products/yed

[26] RYBIČKOVÁ, Alena. *Network flow* [online]. Praha, 2019 [cit. 2023-05-01]. Dostupné z: https://www.fd.cvut.cz/personal/rybicale/TGA_04_Network_flow.pdf. Učební materiál. České vysoké učení technické v Praze, Fakulta dopravní.

[27] MONGA, Sargam. Applications of the Max Flow Problem. *OpenGenus* [online]. New Delhi, 2020 [cit. 2023-05-01]. Dostupné z: https://iq.opengenus.org/maximum-flow-problem-overview/

[28] WEKESA, Erick. Cycle Detection in a Graph in C#: Application of cycle detection. *Section* [online]. Kiambu, 24. 12. 2021 [cit. 2023-05-01]. Dostupné z: https://www.section.io/engineering-education/graph-cycle-detection-csharp/