# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Uhlík** Jméno: **Jan** Osobní číslo: **457921**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra teoretické informatiky**

Studijní program: **Informatika**

Studijní obor: **Teoretická informatika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Grafy a grafové algoritmy pro knihovnu algoritmů**

Název bakalářské práce anglicky:

**Graphs and graph algorithms for algorithms library**

Pokyny pro vypracování:

Nastudujte současné reprezentace grafů v knihovně algoritmů vyvíjené na katedře teoretické informatiky.Navrhněte vylepšení datových struktur se zaměřením na univerzalitu návrhu.Implementujte vylepšení datových struktur do knihovny algoritmů.Nastudujte současné implementace grafových algoritmů hledající cesty v grafech.Navrhněte vhodné algoritmy hledající cesty v grafech pro přidání do knihovny algoritmů.Implementujte nové algoritmy do knihovny algoritmů.Navrhněte kriteria porovnání různých algoritmů hledání cest v grafu a na původních i nových algoritmech proveďte měření.

Seznam doporučené literatury:

Dodá vedoucí práce.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Jan Trávníček, katedra teoretické informatiky FIT**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **22.01.2018** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: _____

| Ing. Jan Trávníček<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | doc. RNDr. Ing. Marcel Jiřina, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

## III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studenta

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

# Grafy a grafové algoritmy pro knihovnu algoritmů

***Jan Uhlík***

Katedra teoretické informatiky

Vedoucí práce: Ing. Jan Trávníček

9. května 2018

# Poděkování

Rád bych poděkoval všem, kteří mi pomáhali při vzniku této bakalářské práce. V první řadě děkuji Janu Trávníčkovi za skvělé vedení a také za nesčetné podněty a cenné rady hlavně v části implementační.

Speciální dík patří Matyášovi Křišťanovi, Josefu Eriku Sedláčkovi a Miroslavu Sochorovi za podporu v průběhu celého bakalářského studia. Bez jejich pomoci bych se k tvorbě této práce jen steží dopracoval.

Děkuji také svým rodičům a Kátě Jiříkové za neustálou podporu a vytvoření ideálních rodinných podmínek pro úspěšné studium.

Za finální korekturu textu vděčím Lucii Doušové, díky které z práce zmizelo mnoho chyb a překlepů.

V neposlední řadě děkuji Miru Hrončokovi a Adamovi Heroutovi za tipy a triky s psaním bakalářské práce v XeLaTeX.

# Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. května 2018 ……

České vysoké učení technické v Praze

Fakulta informačních technologií



## Odkaz na tuto práci

UHLÍK, Jan. *Grafy a grafové algoritmy pro knihovnu algoritmů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018. Dostupný také z WWW: ⟨`https://gitlab.com/ctu-fit/bachelor-thesis`⟩.

# Abstrakt

Tato bakalářská práce vylepšuje implementaci grafů a grafových algoritmů v knihovně algoritmů ALib vyvíjené na Katedře teoretické informatiky FIT ČVUT v Praze. Hlavní důraz je kladen na univerzalitu návrhu, který vede ke snadnému používaní a jednoduchému rozšíření. Práce dále přidává do této knihovny nové doposud nenaimplementované algoritmy pro prohledávání grafů a hledání cest v nich. Práce také obsahuje měření původních i nově přidaných algoritmů a porovnává experimentálně naměřené hodnoty s teoretickými poznatky.

**Klíčová slova** graf, grafové algoritmy, rozšíření knihovny, ALib, hledání cest, datové struktury, C++

# Abstract

The bachelor thesis improves the implementation of the graphs and graph algorithms in the library ALib. This library has been developed at the Department of Computer Science FIT CTU in Prague. A great deal of emphasis has been placed on the universality of the implementation design which leads to intuitive usage and easy future extension of the library. The thesis also extends the library with new graph traverse and graph pathfinding algorithms and makes experiments with the original ones as well as the new ones.

**Keywords** graph, graph algorithms, library extension, ALib, pathfinding, data structures, C++

# Obsah

# Seznam algoritmů

# Seznam ukázek kódu

# Seznam tabulek

# Seznam obrázků

# Úvod

Dnešní dobu často označujeme jako *digitální věk*. Lidé začali brát moderní digitální technologie jako například internet, GPS navigace nebo mobilní telefony za samozřejmost. Málokdo si pak ale uvědomuje rozsáhlost teoretického aparátu stojícího na pozadí, z něhož vystupují tyto technologie jen jako pomyslná špička ledovce. A proto je i dnes stále aktuální studium a rozvoj těchto teoretických základů.

Tato práce je určena především pro studenty a pedagogy na FIT ČVUT v Praze. Studentům usnadní pochopení některých pojmů a algoritmů z teorie grafů, kdežto pedagogové využijí výsledky této práce pro generování testů a k jejich automatické kontrole.

Jak již bylo řečeno, práce se zabývá oblastí úzce spjatou s teoretickou informatikou, a sice grafy a grafovými algoritmy. Konkrétně se práce specializuje na skupinu algoritmů pro prohledávání grafů a hledání cesty v nich a jejich následnou implementací do *Algoritmové knihovny ALib*, která vzniká na FIT ČVUT v Praze. Práce je rozdělena do tří hlavních částí *teoretické*, *implementační* a *měřící*.

Toto téma jsem si zvolil, neboť problém prohledávání grafů a hledání nejkratší cesty v grafech má široké uplatnění napříč vědními disciplínami. Můžeme jej najít například v dnes tak často diskutované *umělé inteligenci*. Důkazem důležitosti tohoto tématu může být i fakt, že i přes poměrně

dlouhou historii vědeckého bádání přicházejí matematici a informatici se stále lepšími a efektivnějšími algoritmy.

Práce navazuje na předchozí bakalářské a diplomové práce z FIT ČVUT, které se podílely na rozšíření výše zmíněné Algoritmové knihovny. Konkrétně na bakalářkou práci [1] Jana Brože z roku 2017, která byla zaměřená na hledání kostry, maximálního toku a minimálního řezu v grafech, a na práci [2] Davida Roscy z roku 2015 zabývající se isomorfismem na planárních grafem.

Kapitola 1

# Cíl práce

Jak již bylo v úvodu řečeno, tato bakalářská práce se skládá ze tří částí. První *teoretická* část zahrnuje kapitoly 2, 3, 4 a má za cíl navrhnout a popsat vhodné grafové algoritmy zaměřené na prohledávání grafů a hledání cest v nich pro rozšíření knihovny ALib. V popisu každého algoritmu bude výčet jeho vlastností a seznam jeho možných optimalizací.

Druhá *implementační* část má dva hlavní cíle. Prvním z nich je návrh vylepšení datových struktur reprezentující grafy v knihovně ALib a jeho následná implementace. Důraz by měl být kladen na univerzalitu návrhu a dodržení knihovních standardů. Detailnějším popisem provedených změn se zabývá kapitola 5. Druhým cílem je pak implementace navržených a popsaných algoritmů.

Poslední *měřící* část má za cíl stanovení porovnávacích kritérií pro popisované algoritmy a následnou realizaci měření. Seznam stanovených kritérií spolu s výsledky měření je pak v kapitole 6.

Kapitola 2

# Notace a terminologie

Pro zachování společné terminologie jsou následující definice částečně přejaty z vybraných předmětů FIT ČVUT v Praze. Na začátku sekce vždy budou vypsány předměty, ze kterých bylo čerpáno.

## 2.1 Základní pojmy z teorie grafů

Následující definice jsou přejaty z předmětů BI-AG1, BI-AG2 a BI-ZUM čerpajících z [3], [4], [5], [6].

Základním stavebním kamenem pro nás bude zavedení pojmu *graf*. Existuje více způsobů jak tento pojem definovat. Například pomocí uspořádané dvojice množiny vrcholů a množiny hran, tak jako v předmětu BI-AG1. My se ovšem přikloníme k obecnější definici.

**Definice 2.1.** *Neorientovaný graf* je uspořádaná trojice $G = (V, E, \epsilon)$ taková, že

- $V, E$ jsou nějaké disjunktní množiny a
- $\epsilon : E \to \binom{V}{2}$ je prosté zobrazení, které každé hraně přiřadí *neuspořádanou* dvojici vrcholů.

Zobrazení $\epsilon$ nazýváme *incidence*.

**Definice 2.2.** *Orientovaný graf* je uspořádaná trojice $G = (V, E, \epsilon)$ taková, že

- $V, E$ jsou nějaké disjunktní množiny a
- $\epsilon : E \to \binom{V}{2}$ je prosté zobrazení, které každé hraně přiřadí *uspořádanou* dvojici vrcholů.

Budeme zde také používat známou terminologii *soused*, *následník* a *předchůdce* vrcholu.

**Definice 2.3.** Nechť $\epsilon(e) = \{u, v\}$ kde $e \in E$ je hrana v neorientovaném grafu $G = (V, E, \epsilon)$. Pak řekneme, že

- vrcholy $v$ a $u$ jsou *koncové vrcholy* hrany $e$,
- $u$ je *sousedem* $v$ v $G$ (a naopak),
- množinu všech sousedů vrcholu $v$ budeme označovat $Neigh(v)$ (popř. $Succ(v)$ nebo $Pred(v)$),
- $u$ a $v$ je *incidentní* s hranou $e$.

**Definice 2.4.** Nechť $\epsilon(e) = (u, v)$ kde $e \in E$ je hrana v orientovaném grafu $G = (V, E, \epsilon)$. Pak řekneme, že

- $u$ *předchůdcem* $v$ a $v$ *následníkem* $u$,
- množinu všech předchůdců (resp. následníků) vrcholu $v$ budeme označovat $Succ(v)$ (resp. $Pred(v)$).

Pro zjednodušení budeme ve zbytku této kapitoly používat obecný pojem *graf*. Níže uvedené definice pak bude možné analogicky zavést jak pro orientované tak neorientované grafy.

Při studiu umělé inteligence se často pro graf používá označení *stavový prostor*.

**Definice 2.5.** *Stavový prostor* je graf, kde

- vrcholy grafu reprezentují *stavy* (tj. popis stavu řešeného problému)
- a hrany grafu reprezentují *akce* (tj. změna stavu).

Pokud nás zajímá pouze vybraná část grafu, mluvíme o takzvaném *podgrafu*.

**Definice 2.6.** Graf $H = (V_H, E_H, \epsilon_H)$ je *podgrafem* grafu $G = (V_G, E_G, \epsilon_G)$ právě tehdy, když

$$(V_H \subseteq V_G) \wedge (E_H \subseteq E_G) \wedge (\forall e \in E_H : \epsilon_H(e) = \epsilon_G(e)).$$

Tuto skutečnost značíme $H \subseteq G$.

Pro zavedení pojmu *cesta* v grafu si nejprve definujeme obecnější pojem *sled*.

**Definice 2.7.** Posloupnost $S = \langle v_0, e_1, v_1, e_2, \ldots, e_n, v_n \rangle$ se nazývá $v_0v_n$-*sled* v grafu $G = (V, E, \epsilon)$, pokud

- $\forall i \in \{0, \ldots, n\} : v_i \in V$
- a $\forall i \in \{1, \ldots, n\} : \epsilon(e_i) = \{v_{i-1}, v_i\} \in E$.

Sled, pro který platí $i \neq 0$ (tedy obsahuje alespoň jednu hranu) a zároveň $v_0 = v_n$, nazýváme *uzavřeným*, ostatní sledy nazýváme *otevřenými*. *Délkou* $\text{len}(S)$ sledu $S$ rozumíme počet jeho hran, tedy $\text{len}(S) = n$.

Přidáním podmínky pak dostáváme pro nás tak důležitý pojem *cesta*.

**Definice 2.8.** *Cesta* je sled, ve kterém se neopakují vrcholy.

**Definice 2.9.** *Kružnicí grafu* $G$ nazýváme uzavřený sled, ve kterém se neopakují vrcholy s výjimkou prvního a posledního vrcholu sledu.

Ačkoliv ještě nemáme zavedený pojem *ohodnocený graf*, díky výše uvedené definici sledu můžeme zavést další klíčový pojem *vzdálenosti* mezi dvěma vrcholy v grafu.

**Definice 2.10.** Pro libovolné dva vrcholy $u, v \in G$, *vzdálenost* $g(u, v)$ je minimum z délek všech $uv$-cest, popřípadě $+\infty$, pokud žádná taková cesta neexistuje.

V některých případech je vhodné zaměřit svojí pozornost pouze na speciální typy grafů. Do této kategorie jistě patří *stromy*.

**Definice 2.11.** Graf $G = (V, E, \epsilon)$ je *souvislý*, pokud pro $\forall u, v \in V$ existuje $uv$-cesta. Jinak je $G$ *nesouvislý*.

**Definice 2.12.** Graf $G = (V, E, \epsilon)$ nazveme *stromem*, pokud je souvislý a neobsahuje žádnou kružnici.

- Uspořádanou dvojici $(G, k)$, kde $G$ je strom a $k \in V$ nazveme *zakořeněným stromem* a $k$ *kořenem* stromu $G$.
- Leží-li $u \in V$ na cestě z $v$ do kořene $k$, pak $u$ je *předek* $v$ a $v$ je *potomek* $u$.
- Pokud je navíc $\{u, v\} \in \epsilon(E)$, říkáme, že $u$ je *otec* $v$ a $v$ je *syn* $u$.
- Vrcholy rozdělíme podle vzdálenosti od kořene do *hladin*: v nulté leží kořen, v první jeho synové, atd.
- *Hloubka vrcholu* je jeho vzdálenost od kořene (tedy číslo hladiny).

**Definice 2.13.** *Prohledávací strom* je zakořeněný strom, který odpovídá nějakému procesu prohledávání grafu. Strom je zakořeněn v počátečním vrcholu prohledávání a každý vrchol grafu se vyskytuje v prohledávacím stromě právě jednou.

Pro určení asymptotické složitosti některých prohledávacích algoritmů je užitečný pojem *větvící faktor*.

**Definice 2.14.** *Větvící faktor* $b$ je průměrný počet synů všech vrcholů v prohledávacím stromě.

Přidáním dalšího zobrazení do definice *grafu* získáváme již jednou zmíněný *ohodnocený graf*.

**Definice 2.15.** *Ohodnocený graf* je čtveřice $(V, E, \epsilon, \omega)$, kde trojice $(V, E, \epsilon)$ je graf a $\omega : E \to \mathbb{R}$ zobrazení, které přiřazuje hranám jejich ohodnocení. *Délkou* $\text{len}(S)$ sledu $S$ v ohodnoceném grafu rozumíme součet ohodnocení hran, které sled $S$ obsahuje, tedy $\text{len}(S) = \sum_{\forall e \in S} \omega(e)$.

Pro nás zajímavou skupinou grafů budou takzvané *mřížkové grafy* (anglicky *lattice graph*). Do této kategorie patří například *čtvercová* nebo *šestihranná mřížka*. Pro obecnou definici celé této skupiny bychom potřebovali

nemálo matematických pojmů, které by byly nad rámec této práce. Spokojíme se tedy alespoň s konkrétní definicí každé mřížky, kterou budeme v textu používat a pro obecnou definici se odkážeme na příslušnou literaturu [7], [8].

**Definice 2.16.** *(Čtyřsměrná) čtvercová mřížka* je neorientovaný graf takový, že

- množina vrcholů $V \subseteq \{(i,j) \mid i,j \in \mathbb{N}_0\}$
- a množina hran $\epsilon(E) = \{\{(i_1,j_1),(i_2,j_2)\} \mid (i_1,j_1),(i_2,j_2) \in V \wedge |i_1 - i_2| + |j_1 - j_2| = 1\}$.

**Definice 2.17.** *Osmisměrná čtvercová mřížka* je neorientovaný graf takový, že

- množina vrcholů $V \subseteq \{(i,j) \mid i,j \in \mathbb{N}_0\}$
- a množina hran $\epsilon(E) = \{\{(i_1,j_1),(i_2,j_2)\} \mid (i_1,j_1),(i_2,j_2) \in V \wedge (|i_1 - i_2| = 1 \vee |j_1 - j_2| = 1)\}$.

Pokud pracuje s ohodnoceným grafem, pak ohodnocení hran je rovná euklidovským vzdálenostem mezi vrcholy (tedy 1 nebo $\sqrt{2}$).

Příklad osmisměrné čtvercové mřížky je potom na obrázku 2.1.

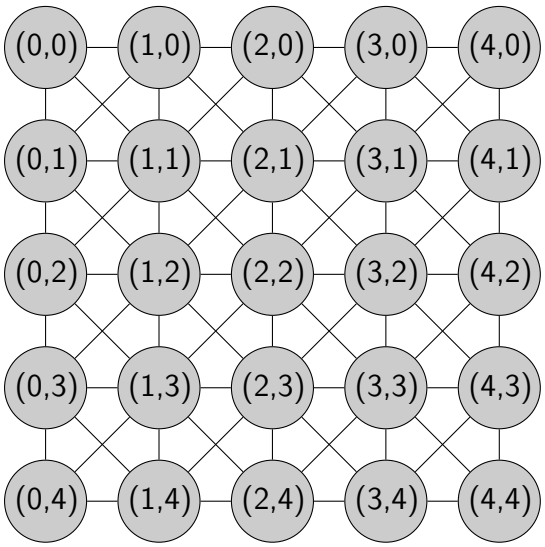
## 2.2 Asymptotická složitost

Následující definice jsou přejaty z předmětu BI-ZDM čerpající z [5].

Pro potřeby porovnání složitostí jednotlivých algoritmů se používá takzvaná asymtotická složitost.

**Definice 2.18.** Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je *nejvýše řádu* $g(n)$, psáno $f(n) = \mathrm{O}(g(n))$, jestliže

$$(\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}^+)(\forall n \geq n_0) : f(n) \leq c \cdot g(n).$$




**Obrázek 2.1:** Příklad osmisměrné čtvercové mřížky

**Definice 2.19.** Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je *téhož řádu jako* $g(n)$, psáno $f(n) = \Theta(g(n))$, jestliže

$$(\exists c_1, c_2 \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}^+)(\forall n \geq n_0) : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

**Definice 2.20.** Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je *nejméně řádu* $g(n)$, psáno $f(n) = \Omega(g(n))$, jestliže

$$(\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}^+)(\forall n \geq n_0) : c \cdot g(n) \leq f(n).$$

Poznamenejme, že tato notace zanedbává multiplikativní konstanty a porovnává pouze řád funkcí. To může být často omezující. Pokud budeme chtít porovnávat algoritmy, jejichž řád je stejný, budeme se muset odkázat na empiricky naměřené hodnoty viz kapitola 6.

## 2.3 Datové struktury

Následující definice jsou částečně přejaty z [9]. Stejné definice jsou využity také v předmětu BI-PA2.

Některé algoritmy vyžadují speciální datové typy pro uchovávání mezivýsledků. Pro sjednocení terminologie zde uvedeme definici a rozhraní jednotlivých námi využívaných *abstraktních datových typů*.

**Definice 2.21.** *Abstraktní datový typ* (dále jen jako ADT) je typ dat, jehož chování nezávisí na vlastní implementaci.

**Definice 2.22.** *Fronta* je ADT, jehož operace výběru vrátí vždy prvek, který byl do vložen jako první (anglicky *first in, first out* ve zkratce *FIFO*). Fronta má následující rozhraní:

- *create* – vytvoř a inicializuj frontu,
- *enqueue* – vlož položku na konec fronty,
- *dequeue* – odeber položku ze začátku fronty, pokud je fronta prázdná, nedělej nic,
- *front* – vrať položku ze začátku fronty, pokud je fronta prázdná, nedělej nic,
- *empty* – vrať *true* pokud je fronta prázdná, jinak *false* (často budeme používat zkrácený zápis $Q = \emptyset$).

**Definice 2.23.** *Zásobník* je ADT, jehož operace výběru vrátí vždy prvek, který byl vložen naposledy (anglicky *last in, first out*, ve zkratce *LIFO*). Zásobník má následující rozhraní:

- *create* – vytvoř a inicializuj zásobník,
- *push* – vlož položku na vrchol zásobníku,
- *pop* – odeber položku z vrcholu zásobníku, pokud je zásobník prázdný, nedělej nic,
- *top* – vrať položku z vrcholu zásobníku, pokud je zásobník prázdný, nedělej nic,
- *empty* – vrať *true* pokud je fronta prázdná, jinak *false* (často budeme používat zkrácený zápis $Q = \emptyset$).

## 2.4 Heuristické funkce

Následující definice jsou přejaty z předmětu BI-ZUM čerpající z [6].

Při hledání nejkratších cest v grafech budeme často využívat nějakou externí informaci o výhodnosti expandování jednotlivých vrcholů.

**Definice 2.24.** Nechť $G = (V, E, \epsilon, \omega)$ je ohodnocený graf a $g$ cílový hledaný vrchol. *Heuristická funkce* je libovolná funkce

$$h : V \to \mathbb{R}_0^+$$

taková, že $\forall v \in V : h(v)$ udává odhad délky cesty mezi vrcholem $v$ a $g$ a $h(g) = 0$.

Návrh správné heuristické funkce silně závisí na podstatě zadaného problému a je většinou velmi složitý. My si zde alespoň ukážeme ty nejzákladnější a nejčastěji používané.

**Definice 2.25.** Nechť $p$ a $q$ jsou dva vektory v $\mathbb{R}^n$. *Manhattanskou vzdáleností* těchto dvou vektorů nazveme

$$d_M(p, q) = \sum_{i=1}^{n} |p_i - q_i|.$$

**Definice 2.26.** Nechť $p$ a $q$ jsou dva vektory z $\mathbb{R}^n$. *Euklidovskou vzdáleností* těchto dvou vektorů nazveme

$$d_E(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

**Definice 2.27.** Nechť $p$ a $q$ jsou dva vektory z $\mathbb{R}^n$. *Chebyshevovou vzdáleností* těchto dvou vektorů nazveme

$$d_{Ch}(p, q) = \max_{i \in \{1,\ldots,n\}} |q_i - p_i|.$$

**Definice 2.28.** Nechť $p$ a $q$ jsou dva vektory z $\mathbb{R}^2$. *Diagonální vzdáleností* těchto dvou vektorů nazveme

$$d_D(p, q) = (|q_1 - p_1| + |q_2 - p_2|) + (\sqrt{2} - 2) \cdot \min(|q_1 - p_1|, |q_2 - p_2|).$$

## 2.5 Ostatní

Na závěr této kapitoly poznamenejme, že bude-li někde v textu potřeba formulovat další pojem, objeví se jeho definice v příslušné kapitole. Zvýší to tak čitelnost textu a odstraní zbytečné listování.

Kapitola 3

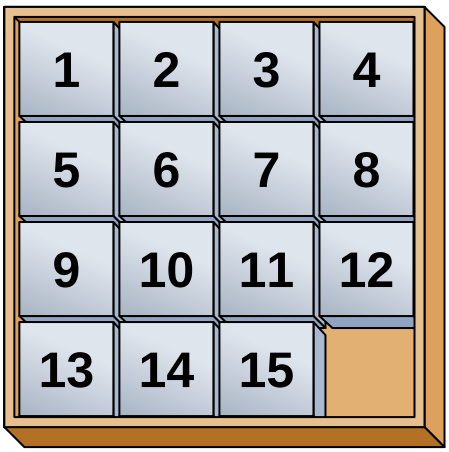
# Algoritmy pro prohledávání grafu

Jednou ze základních kategorií algoritmů v teorii grafů je prohledávání. Cílem těchto algoritmů je systematicky procházet vrcholy a hrany grafu. Ačkoliv se to na první pohled nemusí jevit důležité, mají tyto algoritmy nepřeberné množství praktických i teoretických využití. Uveďme tedy alespoň některé z nich:

- navigování robota v prostoru (například v bludišti),
- hledání souvislých komponent v grafech,
- pohyb postavy v počítačové hře
- a hledání mostů a artikulací v grafech.

Mnoho problémů, u kterých bychom na první pohled nehledali souvislost s grafovými algoritmy, lze redukovat (tzn. převést) právě na prohledávání v grafu.

Krásným příkladem může být klasická hra *15* (též známá jako *Loydova 15*, popřípadě *Lišák*). Hlavolam sestává z krabičky, do které se vejde $4 \times 4$ očíslovaných čtvercových kamenů a kde místo pro jeden kámen je volné. Na začátku jsou kameny náhodně zamíchány. Cílem hráče je pomocí posunů kamenů vždy na volné místo nalézt sérii po sobě jdoucích přesunů tak, aby na konci byly kameny seřazené a volné místo se nacházelo v pravém dolním rohu krabičky.




**Obrázek 3.1:** Loydova 15, zdroj [10]

Tuto hru lze redukovat na grafový problém následovně:

- vrcholy grafu budou tvořeny vždy validním stavem hry
- a hrana v grafu bude mezi dvěma stavy hry, které lze jedním posunem kamene na sebe vzájemně převést.

V takto zavedeném stavovém prostoru se snažíme z námi zadaného počátečního stavu najít libovolnou cestu do stavu koncového (tzn. stav, ve kterém jsou kameny seřazeny).

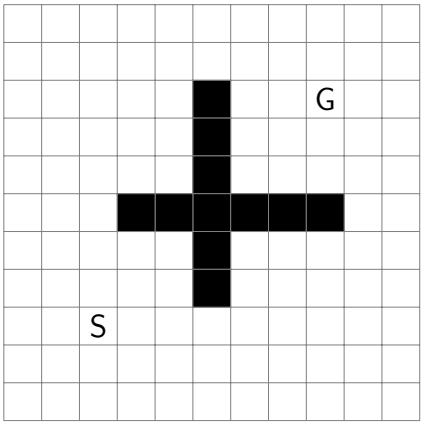
Výše uvedený postup lze s menší či větší modifikací aplikovat i na jiné problémy. Ve velké míře je toho využíváno při studiu a vývoji umělé inteligence. Ta je z velké části postavena právě na prohledávání stavového prostoru. Zmiňme například v dnešní době velmi diskutovaný pokrok umělé inteligence *Alpha Go*, která již poráží velmistry této hry. Ačkoliv se mělo dlouhou dobu za to, že díky složitosti hry *Go* (okolo $10^{5,3 \cdot 10^{170}}$ možných stavů) nebude výkon dnešních počítačů stačit pro poražení lidského hráče.

V následujících sekcích si detailně popíšeme některé tyto algoritmy a u každého se zamyslíme nad jeho vlastnostmi a případnou optimalizací.

Jelikož podstatnou část této práce tvoří implementace těchto algoritmů, u každého algoritmu uvedeme kód, který jej spustí. Abychom vždy nemuseli znovu deklarovat graf, na kterém algoritmus pouštíme, zavedeme si ho

v kódu B.1 a v ukázkách se na něj budeme pouze odvolávat. Jedná se o *osmisměrnou čtvercovou mřížku*, jejíž vizualizaci můžeme vidět na obrázku 3.2.

*Poznámka.* Čtvercová mřížka má dvě základní možnosti vizualizace. První jsme použili v případě vizualizace 2.1, kde vrcholy i hrany jsou explicitně znázorněné. Druhou možností je znázornit pouze mřížku, tak jak je tomu na obrázku 3.2. Vrchol v takové vizualizaci je potom celé políčko v této mřížce a hrany jsou implicitně mezi všemi sousedícími vrcholy. Pokud je políčko černé, znamená to, že vrchol, který by byl v grafu na tomto místě, chybí a tudíž chybí i hrany incidentní s tímto vrcholem.

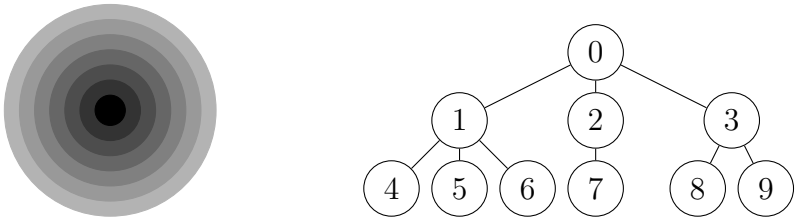



**Obrázek 3.2:** Vizualizace výchozího grafu B.1

## 3.1 Prohledávání do šířky

První algoritmus, který si představíme, se jmenuje *prohledávání do šířky* (anglicky *Breadth-first search*, ve zkratce pak *BFS*). Za objevitele tohoto algoritmu je považován E. F. Moore, který jej jako první popsal ve svém článku [11] z roku 1959. Nezávisle na něm přišel se stejným algoritmem v roce 1961 také C. Y. Lee [12] při zkoumání elektrických obvodů.

BFS pracuje na *neohodnoceném* grafu. Algoritmus dostane na vstupu graf a počáteční vrchol a postupně prohledává vrcholy směrem od vstupního vrcholu. Tento způsob průchodu je označován často jako *záplavové prohledávání*. Toto označení pramení v jisté podobnosti s představou, kdy do




**(a)** Záplavové prohledávání **(b)** Prohledávací BFS strom

**Obrázek 3.3:** Vizualizace BFS algoritmu

vstupního vrcholu přitéká voda a ta se stejně jako BFS šíří skrze hrany do dalších vrcholů. Důležité je, že algoritmus prohledává graf rovnoměrně v každém směru, nebo-li že se šíří ve vlnách viz vizualizace 3.3. Na obrázku také můžeme vidět příklad prohledávacího BFS stromu, kde čísla označují pořadí prohledávání jednotlivých vrcholů.

BFS využívá ve své implementaci ADT *fronta*. Na počátku vloží do fronty počáteční vrchol. Poté v každé iteraci odebere vrchol ze začátku fronty a provede jeho expanzi. Expanzí se zde rozumí proces, kdy do fronty přidáváme všechny doposud nenavštívené následníky vrcholu, které se v ní doposud nevyskytují. Pro detailnější popis se odkážeme na pseudokód 3.1. Ukázka kódu B.2 pak spouští prohledávání na BFS s minimální možnou konfigurací.

### 3.1.1 Asymptotická složitost

Časovou složitost BFS můžeme vyjádřit jako součet mohutností množiny vrcholů a množiny hran tak, jak je uvedeno v tabulce 3.1. Tento vztah vyplývá z faktu, že při prohledávání grafu musíme v nejhorším případě navštívit všechny vrcholy a projít přes všechny hrany. Paměťová složitost je pak rovna pouze mohutnosti množiny vrcholů, a to z důvodu existence fronty $Q$ a množiny $Visited$, které v nejhorším případě obsahují všechny vrcholy grafu.

V umělé inteligenci se často používají grafy s nekonečným množstvím vrcholů, takzvané *nekonečné grafy*. U nich je výše zmíněná formulace časové a prostorové složitosti nepraktická. V těchto případech se odvozuje časová složitost na základě *větvícího faktoru b*. $d$ pak označuje vzdálenost

```
   Input: G = (V, E, ϵ) graf ; s počáteční vrchol
 1 Function BFS(G, s):
 2 |  Visited ← Visited ∪ {s}
 3 |  Q := fronta
 4 |  Q.enqueu(s)
 5 |  while Q ≠ ∅ do
 6 |  |  v ← Q.dequeue()
 7 |  |  forall w ∈ Succ(v) do
 8 |  |  |  if w ∉ Visited then
 9 |  |  |  |  Visited ← Visited ∪ {w}
10 |  |  |  |  Q.enqueue(w)
11 |  |  |  end
12 |  |  end
13 |  end
```

**Algoritmus 3.1:** BFS algoritmus

mezi počátečním vrcholem a vrcholem, kdy je algoritmus ukončen. Jelikož si algoritmus ukládá všechny následníky vrcholů do fronty a množiny navštívených vrcholů, je paměťová náročnost v tomto způsobu zápisu exponenciální.

| | |
|---|---|
| **Časová složitost:** | $\mathrm{O}(\lvert V\rvert + \lvert E\rvert) = \mathrm{O}(b^d)$ |
| **Prostorová složitost:** | $\mathrm{O}(\lvert V\rvert) = \mathrm{O}(b^d)$ |

**Tabulka 3.1:** Asymptotická složitost BFS algoritmu

### 3.1.2 Hledání cesty mezi dvěma vrcholy

BFS lze použít nejenom k systematickému průchodu grafem. Jednoduchou modifikací původního algoritmu 3.1 lze získat cestu mezi zadaným počátečním a cílovým vrcholem, nebo rozhodnout, že taková cesta neexistuje. Jediné, co je nutné si navíc pamatovat, je předek každého vrcholu, který byl expandován. Poté, co algoritmus doběhne, lze postupně rekonstruovat cestu z cílového vrcholu do počátečního. Pokud cílový vrchol nemá žádného předka, znamená to, že v grafu neexistuje taková cesta. Všimněme si, že nalezená cesta bude nutně nejkratší díky pořadí, ve kterém jsou vrcholy procházeny.

Přímá optimalizace vyplývající z podstaty problému hledání cesty mezi dvěma vrcholy je zastavení algoritmu v momentě, kdy nalezneme cílový vrchol. Jelikož pracujeme na *neohodnoceném grafu*, bude doposud nalezená cesta nutně nejkratší. Tato optimalizace je v implementaci obsažena, nicméně nám v nejhorším případě nezmění časovou ani paměťovou složitost.

Důležitou vlastností BFS při využití v umělé inteligenci je *úplnost*.

**Definice.** Algoritmus nazveme *úplným* právě tehdy, když vždy nalezne cestu mezi dvěma zadanými vrcholy, jestliže taková cesta existuje.

Poznamenejme, že právě díky způsobu procházení vrcholů v grafu je zaručené, že algoritmus danou cestu najde i v grafu s nekonečným počtem vrcholů. Na rozdíl od *DFS*, který si popíšeme v další sekci a který tuto vlastnost nemá.

### 3.1.3 Obousměrné prohledávání

Zůstaňme u problému hledání cesty mezi vrcholy. Další možnost optimalizace je začít prohledávat graf z obou dvou zadaných vrcholů, z počátečního i koncového zároveň. Označme prohledávání grafu z počátečního vrcholu jako *dopředné* a z koncového jako *zpětné*. Postupně budeme střídat dopředné a zpětné prohledávání a proces zastavíme v momentě, kdy nalezneme vrchol $v$, jenž byl navštíven jak dopředným tak zpětným prohledáváním. Poté zrekonstruujeme cestu mezi počátečním (resp. cílovým) vrcholem a vrcholem $v$. Výslednou cestu získáme spojením těchto dvou cest. Přesný popis lze najít v pseudokódu 3.2. Tento algoritmus byl poprvé představen v článku [13] z roku 1970.

*Poznámka.* Jestliže se jedná o orientovaný graf, je *zpětné* prohledávání prováděno proti směru orientaci hran. Jinak řečeno, pro každý vrchol se z množiny předchůdců stává množina následníků a naopak.

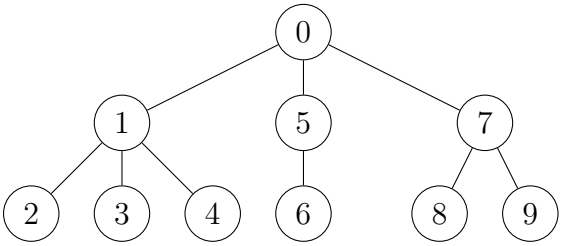
Obousměrné prohledávání je typickou optimalizací, se kterou se setkáme i v dalších algoritmech. Výhodou zde ovšem je fakt, že pracujeme s *neohodnoceným* grafem. Je tak zaručené, že v momentě, kdy objevíme první společný vrchol, je tento vrchol součástí některé nejkratší cesty

mezi počátečním a cílovým vrcholem. Tato optimalizace již zlepší časovou a prostorovou složitost na $O(b^{d/2})$.

## 3.2 Prohledávání do hloubky

*Prohledávání do hloubky* (anglicky *Depth-first search*, ve zkratce *DFS*) je opět algoritmus pro prohledávání *neohodnoceného* grafu. Historie tohoto algoritmu se datuje do 19. století a je připisována francouzskému matematikovi Pierre Trémaux, který studoval strategie pro řešení hlavolamů.

Hlavní rozdíl oproti BFS je ve způsobu procházení jednotlivých vrcholů v grafu. Jak již bylo řečeno, BFS prochází graf po vlnách, oproti tomu DFS se snaží co nejrychleji vzdálit od počátečního vrcholu. Vhodná vizualizace, tak jako tomu bylo u BFS, bohužel neexistuje, avšak dobrou představu si můžeme udělat z prohledávacího DFS stromu 3.5, kde opět čísla označují pořadí prohledávání vrcholů.




**Obrázek 3.5:** Prohledávací DFS strom

Hlavním využívaným ADT je v tomto případě *zásobník*. Ten je možný realizovat dvěma základními způsoby, a to buď explicitně využitím nějaké implementované struktury, nebo systémovým zásobníkem pomocí rekurzivního volání. Častěji se volí druhý zmíněný způsob z důvodů větší přehlednosti kódu a snazší implementace. Deitalní popis opět nalezneme v pseudokódu 3.3.

### 3.2.1 Asymtotická složitost

Časová složitost je u konečných grafů stejná jako u BFS a vychází opět z faktu, že v nejhorším případě procházíme celý graf viz tabulka 3.2. Jelikož

```
   Input: G = (V, E, ϵ) graf ; s počáteční vrchol ; g cílový vrchol
   Output: P cesta mezi s a g
 1 Function BFSBidirectional(G, s, g):
 2     Visited_F ← {s}
 3     Visited_B ← {g}
 4     Q_F := Q_B := fronta
 5     Q_F.enqueue(s)
 6     Q_B.enqueue(g)
 7     predecessor_F(s) = s
 8     predecessor_B(g) = g
 9     while Q_F ≠ ∅ ∧ Q_B ≠ ∅ do
           // Dopředné prohledávání
10         v ← Q_F.dequeue()
11         forall w ∈ Succ(v) do
12             if w ∉ Visited_F then
13                 Visited ← Visited_F ∪ {w}
14                 predecessor_F(w) = v
15                 Q_F.enqueue(w)
16             end
17         end
           // Analogicky pro zpětné prohledávání
18         ...
19         if Visited_F ∩ Visited_B ≠ ∅ then
20             return reconstructPath()
21         end
22     end
23     return noPathExists()
```

**Algoritmus 3.2:** BFS algoritmus s obousměrným prohledáváním

```
   Input: G = (V, E, ϵ) graf ; s počáteční vrchol
 1 Function DFS(G, s):
 2 |   Visited ← ∅
 3 |   DFSRun(s)
 4 Function DFSRun(v):
 5 |   if v ∈ Visited then
 6 |   |   return
 7 |   end
 8 |   Visited ← Visited ∪ {v}
 9 |   forall w ∈ Succ(v) do
10 |   |   DFSRun(w)
11 |   end
```

**Algoritmus 3.3:** DFS algoritmus

si i zde udržujeme množinu všech navštívených vrcholů je i prostorová složitost totožná.

Pokud budeme chtít vyjádřit prostorovou složitost v závislosti na *větvícím faktoru* $b$, musíme si uvědomit, co všechno je ukládáno. Uvažme implementaci s rekurzivním voláním. V hloubce $d$ se na zásobníku nachází přesně $d$ vrcholů, které čekají, až se vynoříme z rekurze. Pro každý tento vrchol si musíme pamatovat $b$ následníků, které jsme již navštívili. To nám dává prostorovou složitost $\mathrm{O}(b \cdot d)$. Všimněme si, že v tomto případě je již prostorová složitost výrazně lepší než u BFS.

| | |
|---|---|
| **Časová složitost:** | $\mathrm{O}(\lvert V\rvert + \lvert E\rvert) = \mathrm{O}(b^d)$ |
| **Prostorová složitost:** | $\mathrm{O}(\lvert V\rvert) = \mathrm{O}(b \cdot d)$ |

**Tabulka 3.2:** Asymptotická složitost DFS algoritmu

### 3.2.2 Hledání cesty mezi dvěma vrcholy

Podobně jak tomu bylo u BFS, i zde lze algoritmus upravit na hledání cesty mezi dvěma vrcholy. Modifikace je naprosto totožná, stačí si tedy opět pamatovat předchůdce jednotlivých vrcholů. Avšak na rozdíl od BFS nalezená cesta nemusí být optimální (tzn. nejkratší). Tento fakt plyne opět z pořadí, ve kterém jsou vrcholy v grafu procházeny.

Jak jsme již dříve uvedli, algoritmus DFS není *úplný*. V případě nekonečného grafu se může stát, že DFS začne prohledávat špatným směrem a nemusí se ke koncovému vrcholu nikdy ani přiblížit. V následující sekci si představíme možnou modifikaci, která tyto nedostatky odstraní.

## 3.3 Iteračně se prohlubující prohledávání do hloubky

Představme si, že bychom chtěli mít algoritmus, který má prostorovou složitost stejnou nebo lepší než DFS, ale splňuje definici úplnosti a navíc je zaručeno, že při případné modifikaci pro hledání cesty mezi dvěma vrcholy nalezne cestu nejkratší. Jako první se touto myšlenkou zaobíral Richard E. Korf ve svém článku [14] z roku 1985. Potřebujeme upravit pořadí procházených vrcholů v grafu tak, aby bylo shodné s procházením algoritmem BFS. Zároveň však nesmíme ukládat příliš mnoho informací, abychom nezhoršili paměťovou složitost algoritmu. Tohoto lze docílit opakovaným spouštěním DFS s postupným zvyšováním maximální hloubky, do které se může zanořit, tak jak je uvedeno v pseudokódu 3.4. Takto modifikovaný algoritmus nazveme *IDDFS* z anglických slov *iterative deepening*.

```
   Input: G = (V, E, ϵ) graf ; s počáteční vrchol ; max_depth
          maximální hloubka zanoření
 1 Function IDDFS(G, s, max_depth):
 2 |   forall n ∈ {1, ..., max_depth} do
 3 |   |   DLS(root, n)
 4 |   end
 5 Function DLS(v, max_depth):
 6 |   if max_depth = 0 then
 7 |   |   return
 8 |   end
 9 |   forall w ∈ Succ(v) do
10 |   |   DFSRun(w, max_depth − 1)
11 |   end
```

**Algoritmus 3.4:** IDDFS algoritmus

### 3.3.1 Asymptotická složitost

Abychom zachovali paměťovou složitost, nesmíme si ukládat již jednou navštívené vrcholy. Tato skutečnost má za následek, že některé vrcholy budeme procházet vícekrát. To se může zdát jako velmi neefektivní, avšak pro *vyvážené stromy* si dokážeme, že se tím nikterak nezmění časová složitost algoritmu.

**Lemma 3.1.** Algoritmus IDDFS spuštěný na vyváženém stromu má časovou složitost $O(b^d)$ a prostorovou složitost $O(d)$.

*Důkaz.* Při běhu algoritmu jsou vrcholy v hloubce $d$ expandovány přesně jednou, $d-1$ pak dvakrát, a tak dál až nakonec je kořen prohledávacího stromu expandován $d+1$. Zajímá nás tedy chování této řady

$$b^d + 2b^{d-1} + 3b^{d-2} + \cdots + (d-1)b^2 + db + (d+1) = \sum_{i=0}^{d}(d+1-i)b^i.$$

Můžeme vytknout $b^d$

$$b^d(1 + 2b^{-1} + 3b^{-2} + \cdots + (d-1)b^{2-d} + db^{1-d} + (d+1)b^{-d}).$$

Provedeme substituci $x := \frac{1}{b} = b^{-1}$

$$b^d(1 + 2x^1 + 3x^2 + \cdots + (d-1)x^{d-2} + dx^{d-1} + (d+1)x^d).$$

Jelikož $b \geq 0$ a $d \geq 0$, jistě platí

$$b^d(1 + 2x^1 + 3x^2 + \cdots + (d-1)x^{d-2} + dx^{d-1} + (d+1)x^d) \leq b^d(\sum_{n=1}^{\infty} nx^{n-1}).$$

Tato řada konverguje pro $|x| < 1$ k

$$b^d(1-x)^{-2} = b^d\frac{1}{(1-x)^2},$$

pro $|x| < 1$ dostáváme tedy vztah

$$b^d(1 + 2x^1 + 3x^2 + \cdots + (d-1)x^{d-2} + dx^{d-1} + (d+1)b^d) \leq b^d(1-x)^{-2}.$$

A jelikož $(1-x)^{-2} = (1-\frac{1}{b})^{-2}$ je pouze multiplikativní konstanta nezávislá na hloubce zanoření $d$, pak pokud $b > 1$ je výsledná asymptotická složitost $O(b^d)$. Jelikož si v průběhu prohledávání pamatujeme pouze cestu do aktuálně prohledávaného vrcholu, bude výsledná prostorová složitost rovna $O(d)$. Důkaz je přejat z článku [14]. □

| | |
|---|---|
| **Časová složitost:** | $O(\|V\| + \|E\|) = O(b^d)$ |
| **Prostorová složitost:** | $O(\|V\|) = O(d)$ |

**Tabulka 3.3:** Asymptotická složitost IDDFS algoritmu

### 3.3.2 Hledání cesty mezi dvěma vrcholy

Pro nalezení cesty mezi dvěma vrcholy si musíme opět pamatovat předchůdce jednotlivých vrcholů. Abychom však nezhoršili prostorovou složitost, budeme si místo ukládání všech předchůdců pamatovat pouze cestu do aktuálně prohledávaného vrcholu. Vždy před expanzí přidáme vrchol na konec cesty a po následném vynoření z rekurzivního volání vrchol odebereme. Na závěr konstatujme, že jelikož je pořadí procházených vrcholů stejné jako u BFS, algoritmus nalezene nejkratší cestu mezi zadanými vrcholy.

### 3.3.3 Obousměrné prohledávání

Budeme-li chtít spustit tento algoritmus v obousměrné variantě, bude nutné si ukládat navštívené vrcholy z poslední hladiny *dopředného prohledávání*. Přesněji řečeno v každé iteraci smyčky s maximální hloubkou zanoření $k$ spustíme dopředné prohledávání a uložíme si všechny vrcholy ležící v hladině $k$. Následně spustíme dvakrát *zpětné prohledávání* s povolenou maximální hloubkou $k$ a $k+1$, kde místo ukládání navštívených vrcholů kontrolujeme, zda nebyly nalezeny předchozím dopředným prohledáváním. Důvodem, proč spouštíme dvakrát zpětné prohledávání, je snaha vyvarovat se zbytečnému dopřednému prohledávání s vyšší maximální hloubkou zanoření (tzn. ošetření sudé a liché délky cesty). Jelikož je nutné si pamatovat vrcholy navštívené dopředným prohledáváním na poslední hladině, je výsledná asymptotická paměťová složitost $O(b \cdot d)$.

Kapitola 4

# Algoritmy pro hledání cest v grafu

Doposud jsme hledali cesty pouze na *neohodnocených* grafech. To může být v některých případech příliš limitující. Jako příklad si uveďme hledání nejkratší cesty na reálné mapě. Máme zadány dvě města $A$ a $B$ a chceme najít nejkratší cestu mezi nimi. Pokud využijeme jeden z algoritmů BFS/IDDFS dostaneme nejkratší cestu ve smyslu, že cesta obsahuje nejmenší počet měst, kterými musíme projet na cestě z města $A$ do města $B$. To ovšem nemusí být nejkratší cesta, co se týče kilometrové vzdálenosti, o kterou nám jde v tomto případě především. Proto si v této kapitole představíme algoritmy pro nejkratších hledání cest na *ohodnocených* grafech.

Zamysleme se ovšem, pro jaký ohodnocených graf má smysl hledat nejkratší cestu. Například na obrázku 4.1 se nachází tzv. *záporná smyčka*. Ta zapříčiňuje, že pokaždé, když přejdeme po jejích hranách, klesne nám délka cesty. Nedává tedy smysl se ptát na nejkratší cestu mezi vrcholy $A$ a $B$. V reálných problémech se zřídka setkáme s grafy, které záporné smyčky obsahují, a tak se zde omezíme pouze na grafy, jenž je neobsahují.

Za účelem zkrácení některých pseudokódů si zde zavedeme speciální značení:

- *Open*: množina vrcholů otevřených vrcholů.
- *Close*: množina vrcholů uzavřených vrcholů.

- $g(v)$: délka doposud nejkratší nalezené cesty z počátečního vrcholu do vrcholu $v$.
- $gmin$: minimální hodnota $g(v)$ pro $\forall v \in Open$.

Pokud to bude nutné, budeme používat dolní index $_F$ (resp. $_B$) pro dopředné (resp. zpětné) prohledávání. Zároveň vynecháme z pseudokódů řádky, kde si ukládáme předchůdce jednotlivých vrcholů. Pozorný čtenář si jistě sám rozmyslí, kde by se tyto operace případně vyskytovaly.

## 4.1 Bellman-Fordův algoritmus

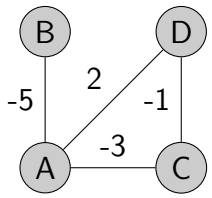
Jeden z prvních algoritmů pro hledání nejkratší cesty v ohodnoceném grafu představil Alfonso Shimble v roce 1955 ve svém článku [15]. Nezávisle na něm publikoval v roce 1956 Richard Bellman ve spolupráci s Lester Ford juniorem článek [16], kde popisuje tentýž algoritmus. Algoritmus hledá nejkratší cesty z počátečního vrcholu do všech ostatních vrcholů a skládá se ze 3 částí

- inicializační,
- relaxační
- a kontrolní.

Případný výskyt záporné smyčky grafu je detekován v kontrolní části. Pro detailnější popis se opět odkážeme na pseudokód 4.1.

Zde je namístě zadefinovat si pojem, který nás bude provázet po zbytek této kapitoly, a sice *relaxace*.

**Definice 4.1.** Proces přepočítávání doposud nejkratších nalezených cest do následníků $w$ vrcholu $v$ nazýváme *relaxací vrcholu* $v$.




**Obrázek 4.1:** Příklad záporné smyčky v grafu

```
   Input: G = (V, E, ϵ, ω) ohodnocený graf ; s počáteční vrchol
 1 Function BellmanFord(G, s):
       // Inicializační část
 2     forall v ∈ V do
 3         g(v) = ∞
 4     end
 5     g(s) = 0
       // Relaxační část
 6     forall i ∈ {1, ..., |V| − 1} do
 7         forall (v, w) ∈ E do
 8             if g(w) > g(v) + ω((v, w)) then
 9                 g(w) = g(v) + ω((v, w))
10             end
11         end
12     end
       // Kontrolní část
13     forall (v, w) ∈ E do
14         if g(w) > g(v) + ω((v, w)) then
15             return negativeCycle()
16         end
17     end
```

**Algoritmus 4.1:** Bellman-Fordův algoritmus

### 4.1.1 Asymptotická složitost

Časovou složitost lze snadno odvodit díky přítomnosti dvou smyček, kdy jedna iteruje přes množinu vrcholů a druhá přes množinu hran. Pro uložení množiny předchůdců a doposud nejkratších cest potřebujeme lineární množství paměti vůči mohutnosti množiny vrcholů viz tabulka 4.1.

| | |
|---|---|
| **Časová složitost:** | $\Theta(\|V\| \cdot \|E\|)$ |
| **Prostorová složitost:** | $\Theta(\|V\|)$ |

**Tabulka 4.1:** Asymptotická složitost Bellman-Fordova algoritmu

### 4.1.2 Optimalizace

Existuje velké množství optimalizací Belman-Fordova algoritmu. My si zde představíme pouze nejznámější z nich.

#### 4.1.2.1 Včasné zastavení

Můžeme vypozorovat, že pokud v jedné iteraci hlavní smyčky (tzn. smyčka přes množinu vrcholů) neproběhne žádná relaxace, neproběhne už v žádné další. Tohoto faktu můžeme využít a případně předčasně ukončit běh algoritmu. Této optimalizaci říkáme *včasné zastavení* a je obsažena v implementaci algoritmu.

#### 4.1.2.2 Yen optimalizace

V roce 1970 publikoval Jin Y. Yen ve svém článku [17] možné optimalizace Bellman-Fordova algoritmu. Jedna z nich je velice podobná *včasnému zastavení*. Jestliže se ohodnocení vrcholu $v$ od poslední iterace nezměnilo, pak se relaxací vrcholu $v$ nezmění ani ohodnocení jeho následníků. Můžeme tedy okamžitě přeskočit tento vrchol. Tato optimalizace nezmění asymtotickou složitost v nejhorším případě. Opět je tato optimalizace obsažena v implementaci algoritmu.

#### 4.1.2.3 SPFA optimalizace

Poslední optimalizace nám poslouží jako pomyslný oslí můstek mezi Bellman-Fordovým algoritmem a skupinou algoritmů nazvaných *Uspořádané prohledávání*. Jihokorejský informatik Fanding Duan v roce 1994 publikoval článek [18], ve kterém představuje vylepšení Bellman-Fordova algoritmu přidáním ADT fronta. Nazval ho *Shortest Path Faster Algorithm* (odtud vznikla zkratka *SPFA*), v překladu *rychlejší algoritmus pro hledání nejkratší cesty*. Navzdory tomuto jménu je asymptotická složitost v nejhorším případě stejná jako u původního algoritmu. Pro stručnost přeskočíme slovní popis algoritmu a pro jeho pochopení se pouze odkážeme na pseudokód 4.2.

Pro detekci záporného cyklu využijeme stejné pozorování jako u původní varianty. V grafu bez záporných smyček nemůže po $|V|$ iteracích být již žádný vrchol relaxován. Proto si pro každý vrchol budeme pamatovat, kolikrát byl vytažen z fronty, a pokud tento počet převýší mohutnost množiny vrcholů, ukončíme běh algoritmu.

```
   Input: G = (V, E, ϵ, ω) ohodnocený graf ; s počáteční vrchol
 1 Function SPFA(G, s):
 2 |  Q := fronta
 3 |  forall v ∈ V do
 4 |  |  g(v) = ∞
 5 |  |  visits(v) = 0
 6 |  end
 7 |  g(s) = 0
 8 |  Q.enqueu(s)
 9 |  while Q ≠ ∅ do
10 |  |  v ← Q.dequeue()
11 |  |  visits(v) = visits(v) + 1
12 |  |  if visits(v) > |V| then
13 |  |  |  return negativeCycle()
14 |  |  end
15 |  |  forall w ∈ Succ(v) do
16 |  |  |  if g(w) < g(v) + ω((v, w)) then
17 |  |  |  |  g(w) = g(v) + ω((v, w))
18 |  |  |  |  if w ∉ Q then
19 |  |  |  |  |  Q.enqueu(w)
20 |  |  |  |  end
21 |  |  |  end
22 |  |  end
23 |  end
```

**Algoritmus 4.2:** SPFA algoritmus

## 4.2 Uspořádané prohledávání

Omezíme-li se pouze na ohodnocené grafy $G = (V, E, \epsilon, \omega)$, kde $\omega : E \to \mathbb{R}_0^+$ (tedy grafy s nezáporným ohodnocením), získáme další důležitou skupinu algoritmů pro hledání cest v grafech. Samotné *uspořádané prohledávání* je pouze koncept algoritmů založený na základě prohledávání vrcholů v pořadí, které určí nějaká prioritní funkce $\pi : V \to \mathbb{R}^+$. Tato funkce slouží jako jakýsi ukazatel, který vrchol je nejlepší (odtud anglický název *Best-first search*) pro další expandování. Vlastnosti výsledného algoritmu jsou úzce spjaty s volbou této prioritní funkce.

Algoritmy založené na tomto konceptu si udržují dvě množiny vrcholů *otevřené* a *uzavřené*. V každé iteraci se z množiny otevřených vrcholů vybere vrchol, jehož priorita je nejvyšší, a ten se následně relaxuje. Jeho doposud neuzavření následníci jsou pak přidáni do množiny otevřených vrcholů. Po relaxaci je vrchol vložen do množiny uzavřených vrcholů. Pro přesný popis algoritmu se opět odkážeme na pseudokód 4.3.

*Poznámka.* Pro dodržení standardního značení budeme uvažovat, že nejvyšší prioritu má vrchol, pro který platí $\pi(v) = 0$, nejnižší pak $\pi(v) = \infty$.

Převážná většina implementací těchto algoritmů je založena na *prioritní frontě*. Jedná se o ADT podobný normální frontě s tím rozdílem, že zde jsou prvky seřazeny podle nějakého klíče (v našem případě podle priority). Jinak tomu není ani v naší implementaci. Toto řešení je výhodné pro náhodné grafy, speciálně pak pro řídké. Na druhou stranu pro grafy úplné bude prioritní fronta zvyšovat asymtotickou složitost v nejhorším případě a je nutné na to pamatovat.

## 4.3 Dijkstrův algoritmus

V roce 1959 představil dánský vědec Edsger W. Dijkstra ve svém článku [19] algoritmus pro hledání nejkratších cest v nezáporně ohodnocených grafech. Algoritmus dostane na vstupu graf a počáteční vrchol $s$, jeho výstupem jsou pak nejkratší cesty z tohoto vrcholu do všech ostatních dosažitelných vrcholů. Pokud požadujeme nalezení konkrétní cesty mezi vrcholy $s$ a $v$, pak můžeme využít, za předpokladu, že zobrazení $\omega$ je prosté (každá hrana má unikátní ohodnocení), stejné modifikace jako u algoritmu BFS. Zastavíme prohledávání v momentě, kdy nalezený vrchol odpovídá cílovému vrcholu $v$.

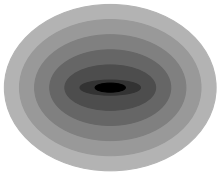
Dijkstrův algoritmus je speciální případ *Uspořádaného prohledávání* pro prioritní funkci:

$$\pi(v) = g(v),$$

kde $g(v)$ je délka cesty mezi vrcholem $v$ a počátečním vrcholem $s$.

```
Input: G = (V, E, ϵ, ω) ohodnocený graf ; s počáteční vrchol ; π
       prioritní funkce
1  Function Best-First Search(G, s, π):
2      Open ← {s}
3      Close ← ∅
4      forall v ∈ V do
5          g(v) = ∞
6          pr(v) = ∞
7      end
8      g(s) = 0
9      pr(s) = π(s)
10     while Open ≠ ∅ do
11         v ← deleteBest(Open)
12         Close ← Close ∪ {v}
13         forall w ∈ Succ(v) do
14             if w ∈ Close then
15                 Continue
16             end
17             new_pr ← π(w)
18             if g(w) > g(v) + ω((v, w)) then
19                 g(w) = g(v) + ω((v, w))
20                 pr(w) = new_pr
21                 if w ∉ Open then
22                     Open ← Open ∪ {w}
23                 end
24             end
25         end
26     end
```

**Algoritmus 4.3:** Uspořádané prohledávání



**Obrázek 4.2:** Vizualizace Dijkstrova algoritmu

Procházení grafu Dijkstrovým algoritmem lze vizualizovat podobně jako u BFS viz obrázek 4.2. Algoritmus se také šíří ve vlnách, avšak některými směry vlna postupuje rychleji.

### 4.3.1 Asymtotická složitost

Časová složitost je přímo závislá na implementaci. Při využití haldy jako prioritní fronty nás zajímá čas potřebný pro provedení operace *sniž klíč* $T_{DK}$ (anglicky *deacrease key*) a *odstraň minimum* $T_{EM}$ (anglicky *extract minimum*). Konkrétní složitosti při různých implementacích jsou následující:

- **naivní řešení bez haldy:** $\mathrm{O}(|V|^2 + |E|)$,
- **binární/binomiální halda:** $\mathrm{O}((|V| + |E|) \cdot \log |V|)$,
- **fibonacciho halda:** $\mathrm{O}(|E| + |V| \cdot \log |V|)$.

Je důležité si uvědomit, že pokud $|E| \sim |V|^2$, potom je řešení s využitím binární/binomiální haldy asymtoticky horší než naivní řešení bez haldy:

$$\mathrm{O}(|V|^2 + |E|) \sim \mathrm{O}(|V|^2) \lesssim \mathrm{O}((|V| + |E|) \cdot \log |V|) \sim \mathrm{O}(|V|^2 \cdot \log |V|).$$

Pokud v implementaci dodržíme pravidlo, že se prvek v haldě smí vyskytovat nejvýše jedenkrát, je prostorová složitost lineární k mohutnosti množiny vrcholů viz tabulka 4.2. Stejně jako u BFS lze odvodit časové a prostorové složitosti na základě *větvícího faktoru*.

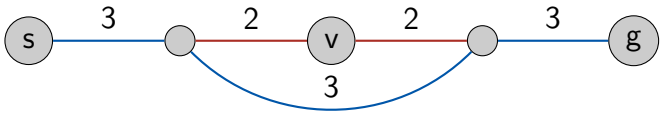
Naše implementace využívá *množinu* z STL knihovny, která je postavena na *červeno-černém stromě*. Asymtotická složitost popsaných operací tedy odpovídá binární/binomiální haldě.

| | |
|---|---|
| **Časová složitost:** | $\mathrm{O}(\lvert V\rvert \cdot T_{DK} + \lvert E\rvert \cdot T_{EM}) = \mathrm{O}(b^d)$ |
| **Prostorová složitost:** | $\mathrm{O}(\lvert V\rvert) = \mathrm{O}(b^d)$ |

**Tabulka 4.2:** Asymptotická složitost Dijkstrova algoritmu

### 4.3.2 Obousměrné prohledávání

Algoritmy Dijkstra a BFS mají jak je vidět mnoho společného. Proto nikoho nepřekvapí, že i zde můžeme pro hledání cesty mezi vrcholy $s$ a $g$ využít obousměrného prohledávání. Na první pohled by se mohlo zdát, že je situace zcela totožná a stačí pouze najít první vrchol $v$, který leží v průniku *dopředného* a *zpětného* prohledávání. Následně pak najít cestu z $s$ do $v$ a z $g$ do $v$ a tyto cesty spojit. Zdání zde ovšem klame, jak ukazuje obrázek 4.3. Výše uvedený postup najde suboptimální cestu s délkou 10, optimální cesta ovšem vrchol $v$ neobsahuje a má délku 9.




**Obrázek 4.3:** Chyba při špatné implementaci Dijkstrova algoritmu s obousměrným prohledáváním

Existuje několik způsobů, jak ošetřit tuto situaci. Nejdříve si však vyslovíme lemma, na kterém bude náš algoritmus postaven (lemma je přejato z [20]).

**Lemma 4.1.** Nechť $G$ je graf, $s$ počáteční vrchol a $g$ cílový vrchol. Nechť $dist[u]$ je vzdálenost získaná dopředným prohledáváním z vrcholu $s$ v $G$ a $dist^R[u]$ je vzdálenost získaná zpětným prohledáváním z vrcholu $g$ v $G$. Označme vrchol $v$ takový, který byl nalezen jak dopředným tak zpětným prohledáváním. Potom existuje nějaká nejkratší cesta mezi $s$ a $t$, která obsahuje nějaký vrchol $u$ takový, že byl nalezen dopředným nebo zpětným prohledáváním (popř. oběma) a pro délku této cesty platí:

$$l(s, g) = dist[u] + dist^R[u].$$

*Důkaz.* Pro rozsáhlost důkaz přeskočíme a odkážeme se pouze na [20]. □

Spustíme tedy prohledávání z obou dvou zadaných vrcholů. Jakmile nalezneme průnik těchto prohledávání, označme vrchol v tomto průniku $v$, máme i cestu mezi zadanými vrcholy. Označme její délku $p$. Tato cesta může být suboptimální, proto pokračujeme v prohledávání a v případě nalezení dalších průniků zkontrolujeme, zda nově nalezená cesta není lepší než původní (pokud ano, upravíme $p$). Algoritmus ukončíme tehdy, když je splněna následující podmínka:

$$g_F(u) + g_B(u) + \alpha \geq p,$$

kde hodnota $\alpha$ je nejmenší ohodnocení hrany v grafu $G$. Tímto je zaručeno, že na konci běhu algoritmus vrátí optimální nejkratší cestu.

Ještě je nutné rozmyslet, jakým způsobem budeme střídat dopředné a zpětné prohledávání. Nabízí se dvě možnosti. Buď budeme střídat tyto dvě prohledávání symetricky (jednou dobředné a jednou zpětné), nebo provedeme vždy relaxaci pouze výhodnějšího prohledávání. V implementaci je zvolena druhá z těchto variant.

Časová složitost se nám touto optimalizací sníží na $O(b^{d/2})$.

## 4.4 Hladové uspořádané prohledávání

Do této chvíle všechny naše algoritmy pracovaly pouze se vstupní grafem a jeho ohodnocením. Představme si nyní, že bychom navíc měli funkci, která říká, jak výhodné je expandovat daný vrchol. Algoritmus, který využívá tuto externí funkci k nalezení nejkratší cesty budeme nazývat *informovaný* a funkci samotnou *heuristickou funkcí*, značíme $h$.

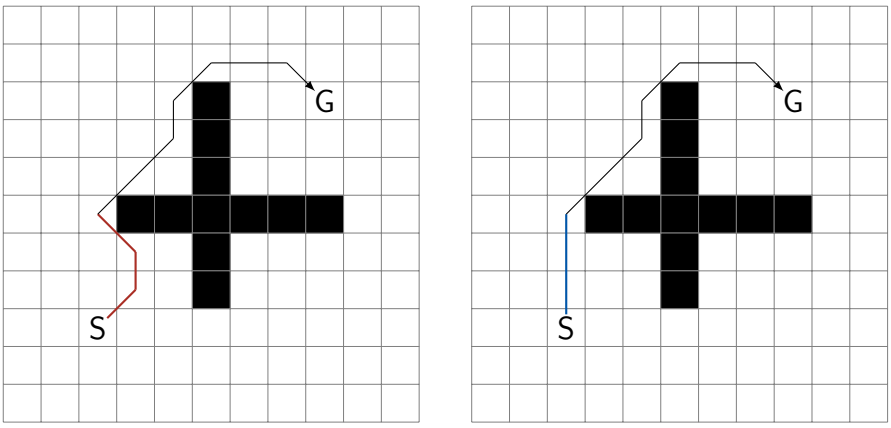
*Hladové uspořádané prohledávání* je dalším speciální případem uspořádaného prohledávání, kde prioritní funkce je rovna

$$\pi(v) = h(v).$$

Takto zadefinované prohledávání postupuje rychle k cílovému vrcholu a zbytečně neexpanduje vrcholy ve špatném směru. Bohužel optimalita nalezené cesty není zaručena, jak ukazuje obrázek 4.4. Pokud totiž

```
Input: G = (V, E, ϵ, ω) ohodnocený graf ; s počáteční vrchol ; goal
       cílový vrchol
1  Function DijkstraBidirectional(G, s, goal):
2      Open_F ← {s}
3      Open_G ← {g}
4      Close_F = Close_B ← ∅
5      p ← ∞
6      α ← min({ω(e)|∀e ∈ E})
7      forall v ∈ V do
8          g_F(v) = g_B(v) = ∞
9      end
10     g_F(s) = g_B(g) = 0
11     while Open_F ≠ ∅ ∧ Open_F ≠ ∅ do
12         if prmin_F + prmin_B + α ≥ p then
13             return reconstructPath(p)
14         end
15         if prmin_F < prmin_B then
16             v ← deleteBest(Open_F)
17             Close_F ← Close_F ∪ {v}
18             forall w ∈ Succ(v) do
19                 if w ∈ Close_F then
20                     Continue
21                 end
22                 if g_F(w) > g_F(v) + ω((v, w)) then
23                     g_F(w) = g_F(v) + ω((v, w))
24                     if w ∉ Open_F then
25                         Open_F ← Open_F ∪ {w}
26                     end
27                 end
28                 if w ∈ Open_B ∪ Close_B ∧ g_F(w) + g_B(w) < p then
29                     p = g_F(w) + g_B(w)
30                 end
31             end
32         else
               // Analogicky pro zpětné prohledávání
33             ...
34         end
35     end
36     return noPathExists()
```

**Algoritmus 4.4:** Dijkstrův algoritmus s obousměrným prohledáváním




**(a)** Suboptimální cesta **(b)** Optimální cesta

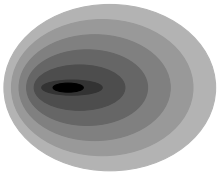
**Obrázek 4.4:** Neoptimalita Hladového uspořádaného prohledávání

algoritmus najde jednou cestu do nějakého vrcholu, už se nesnaží tuto cestu v budoucnu vylepšovat.

Různou volbou prioritní funkce $\pi$ jsme zatím dostali dva různé algoritmy. První byl *Dijkstra*, neinformovaný algoritmus, u kterého je zaručeno nalezení optimální cesty, avšak prohledává zbytečně mnoho vrcholů. Druhým je pak informovaný algoritmus *Hladové uspořádané prohledávání*, který zbytečně neprohledává neperspektivní vrcholy, ale optimalita nalezené cesty není zaručena. Logicky bychom chtěli výhody obou algoritmů spojit. Tím dostáváme algoritmus *A**, který si detailněji popíšeme v následující sekci.

## 4.5 A* algoritmus

V průběhu 60. let minulého století se matematici snažili přijít na způsob, jak efektivně využít heuristickou funkci pro co možná největší zrychlení algoritmů. Vrcholem tohoto snažení byl článek [21], který jako první dokázal, že algoritmy *A1* a *A2* najdou optimální cestu v grafu. Zároveň je v článku popsán algoritmus vzniklý jejich kombinací, který autoři nazvali *A**, a nutné podmínky kladené na heuristickou funkci.



**Obrázek 4.6:** Vizualizace A* algoritmu

Stále se jedná o *uspořádané prohledávání*, kde za prioritní funkci bereme

$$\pi(v) = g(v) + h(v) = f(v),$$

kde $g(v)$ je délka cesty mezi vrcholem $v$ a počátečním vrcholem $s$ a $h(v)$ je heuristická funkce.

Aby byl algoritmus optimální, je nutné od heuristické funkce vyžadovat některé speciální vlastnosti. Pro jejich zavedení bude ovšem potřeba pojem *optimální heuristika*.

*Poznámka.* Následující definice jsou přejaty z předmětu BI-ZUM čerpající z [6].

**Definice 4.2.** Nechť $G = (V, E, \epsilon, \omega)$ je ohodnocený graf a $g$ cílový vrchol. Potom heuristiku $h^*$, která pro každý vrchol $v$ vrací délku nejkratší cesty z vrcholu $v$ do vrcholu $g$, nazýváme *optimální heuristikou*.

Sestrojení takové heuristiky je stejně těžký problém, jako vyřešit problém hledání nejkratší cesty samotný. Proto se spokojíme s heuristickou funkcí, která bude *přípustná* a *monotónní*.

**Definice 4.3.** Nechť $G = (V, E, \epsilon, \omega)$ je ohodnocený graf a $g$ cílový vrchol. Potom heuristiku $h$ nazveme *přípustnou* (anglicky *admissible*) právě tehdy, když

$$\forall v \in V : h(s) \leq h^*(v).$$

**Definice 4.4.** Nechť $G = (V, E, \epsilon, \omega)$ je ohodnocený graf a $g$ cílový vrchol. Potom heuristiku $h$ nazveme *monotónní* (anglicky *monotone*) právě tehdy, když

$$\forall (v, u) \in E : h(u) \leq h(v) + \omega((v, u)).$$

Pro takovou heuristiku je dokázané (důkaz nalezneme v článku [21]), že algoritmus A* nalezne optimální cestu mezi zadanými vrcholy. Poznamenejme, že pokud pro zvolenou heuristickou funkci platí

$$\forall v \in V : h(v) = 0,$$

dostáváme klasický *Dijkstrův* algoritmus.

### 4.5.1 Asymptotická složitost

Časová složitost silně závisí na zvolené heuristické funkci. Jak již bylo uvedeno výše, A* může zdegenerovat na Dijkstrův algoritmus, a proto je v nejhorším případě časová složitost stejná jako právě u Dijkstrova algoritmu. Totéž platí o prostorové složitosti.

### 4.5.2 Obousměrné prohledávání

Krátce po objevu algoritmu *A** přišel Ira Pohl ve své dizertační práci [22] s obousměrnou variantou tohoto algoritmu. Tato modifikace se opět skládá z dvou nezávislých prohledávání *dopředného* a *zpětného*. Každé prohledávání vyžaduje svou vlastní heuristickou funkci, kde $h_F(v)$ (resp. $h_B(v)$) vrací dolní odhad vzdálenosti mezi počátečním (resp. koncovým) vrcholem a vrcholem $v$. Označme $fmin$ minimální hodnotu $f(v)$ pro $\forall v \in Open$. V každé iteraci prohledávání vybereme vrchol $v$ takový, že platí $f(v) = fmin$. Pro tento vrchol následně provedeme expanzi a relaxaci. Budeme si opět pamatovat délku nejkratší doposud nalezené cesty $p$ mezi zadanými vrcholy. Na počátku vyhledávání položíme $p = \infty$. Vždy, když nalezneme vrchol, který je v průniku navštívených vrcholů obou prohledávání, snažíme se vylepšit tuto hodnotu, tedy

$$p = \min\left(p, g_F(v) + g_B(v)\right).$$

Algoritmus zastavíme, jakmile je splněna podmínka

$$\max\left(fmin_F, fmin_B\right) \geq p.$$

Pseudokód je až na výše popsané výjimky totožný s kódem 4.4.

## 4.6 MM algoritmus

V roce 2015 vydali Barker a Korf článek [23], který se zabývá efektivitou obousměrného heuristického prohledávání. Z něj pro nás plyne jedno důležité tvrzení. Nejdříve však označme:

- *Uni-HS* jako jednosměrné heuristické prohledávání a
- *Bi-HS* jako obousměrné heuristické prohledávání.

**Lemma 4.2.** Uni-HS expanduje méně vrcholů než Bi-HS právě tehdy, když pro více jak polovinu vrcholů expandovaných Uni-HS platí:

$$g \leq \frac{C^*}{2},$$

kde $C^*$ je délka optimálního cesty. *Důkaz nalezneme v článku [23].*

Jinými slovy nám toto tvrzení říká, že využití Bi-HS nutně nesníží počet expandovaných vrcholů. Ba dokonce se tento počet může i zvýšit. Proto se budeme snažit přijít na alternativní variantu obousměrného A*, kde je zaručeno, že se obě prohledávání *setkají uprostřed*.

**Definice 4.5.** Říkáme, že se obousměrné prohledávání *setká uprostřed* (anglicky *meets in the middle*) právě tehdy, když hodnota $g_F(v)$ (resp. $g_B(v)$) pro všechny expandované vrcholy $v$ nepřevýší $C^*/2$.

V roce 2016 navázali Holte, Sharon a spol. článkem [24], ve kterém představují obousměrný heuristický algoritmus, kde je garantováno, že obě prohledávání se *setkají uprostřed*. Autoři článku algoritmus nazvali *MM*. Hlavní změnou oproti obousměrné variantě A* je prioritní funkce:

$$\pi_F(v) = \max\left(f_F(v), 2 \cdot g_F(v)\right).$$

Analogicky je potom zavedena $\pi_B(v)$. Označme $C = \min(prmin_F, prmin_B)$, potom v každé iteraci expandujeme vrchol s prioritou $C$. V případě, že takových vrcholů je více, volíme ten, s minimální $g_F$ (resp. $g_B$) hodnotou. Tohoto je v pseudokódu 4.5 docíleno voláním funkce *deleteBest*. Podobně

jako u A* je algoritmus ukončen, jakmile je splněna podmínka:

$$\max(C, fmin_F, fmin_B, gmin_F + gmin_B + \alpha) \geq p,$$

kde $\alpha$ je nejmenší ohodnocení hrany v grafu a $p$ délka doposud nejkratší nalezené cesty (na počátku $p = \infty$).

Asymtotická složitost v nejhorším případě bude opět stejná jako u Dijkstrova algoritmu s obousměrnou optimalizací. Avšak při řešení reálných problémů bude složitost výrazně menší. Její konkrétní hodnota bude závislá na zvolené heuristické funkci a podstatě problému. Pro konkrétní porovnání algoritmů se odkážeme na kapitolu 6.

## 4.7 Iterativně se prohlubující A*

Jak již název napovídá, budeme se nyní zabývat podobnou modifikací algoritmu, jako tomu bylo u *IDDFS*. Výsledkem našeho snažení bude algoritmus, jehož časová složitost zůstane asymptoticky stejná jako u A*, avšak prostorovou složitost výrazně snížíme. Pro připomenutí, hlavní rozdíl mezi DFS a IDDFS byl, že u IDDFS jsme si nepamatovali, které vrcholy jsme již jednou navštívily. U *iterativně se prohlubujícího* A* (anglicky *iterative deeping A**, ve zkratce *IDA**) budeme postupovat obdobně a nebudeme si ukládat žádné informace o navštívených vrcholech. To nutně povede k opětovnému navštívení již jednou prohledávaných vrcholů. Avšak Korf ve svém článku [14] uvádí důkaz, že pokud budeme pracovat s vyváženými stromy a uvažovat pouze heuristickou funkci *s konstantní relativní chybou*, tak se asymptotická časová složitost nezmění.

V obecném případě je časová složitost algoritmu A* závislá na heuristické funkci. Chceme, aby kvalitní heuristická funkce příliš nepodhodnocovala své odhady. Poměr podhodnocených odhadů vůči všem odhadům budeme nazývat *chybovostí* heuristické funkce.

**Definice 4.6.** O heuristické funkci řekneme, že má *konstantní absolutní chybu*, jestliže počet podhodnocechých odhadů je vždy shora omezen nějakou konstantou $k \in \mathbb{N}$, nehledě na množství odhadů.

**Input:** $G = (V, E, \epsilon, \omega)$ ohodnocený graf ; $s$ počáteční vrchol ; *goal* cílový vrchol ; $h_F$ dopředná heuristická funkce ; $h_B$ zpětná heuristická funkce

```
1  Function MM(G, s, goal, h_F, h_B):
2      Open_F ← {s} ; Open_G ← {g} ; Close_F = Close_B ← ∅
3      p ← ∞
4      α ← min({ω(e)|∀e ∈ E})
5      forall v ∈ V do
6          g_F(v) = g_B(v) = ∞
7      end
8      g_F(s) = g_B(g) = 0
9      while Open_F ≠ ∅ ∧ Open_F ≠ ∅ do
10         C = min(prmin_F, prmin_B)
11         if max(C, fmin_F, fmin_B, gmin_F + gmin_B + α) ≥ p then
12             return reconstructPath(p)
13         end
14         if prmin_F < prmin_B then
15             v ← deleteBest(Open_F)
16             Close_F ← Close_F ∪ {v}
17             forall w ∈ Succ(v) do
18                 if w ∈ Close_F then
19                     Continue
20                 end
21                 if g_F(w) > g_F(v) + ω((v, w)) then
22                     g_F(w) = g_F(v) + ω((v, w))
23                     if w ∉ Open_F then
24                         Open_F ← Open_F ∪ {w}
25                     end
26                 end
27                 if w ∈ Open_B ∪ Close_B ∧ g_F(succ) + g_B succ < p then
28                     p ← g_F(succ) + g_B(succ)
29                 end
30             end
31         else
               // Analogicky pro zpětné prohledávání
32             ...
33         end
34     end
35     return noPathExists()
```

**Algoritmus 4.5:** MM algoritmus

V reálných případech je sestrojení takové heuristiky obtížné, a proto se spokojíme s heuristikou s *konstantní relativní chybou*.

**Definice 4.7.** O heuristické funkci řekneme, že má *konstantní relativní chybu*, jestliže chybovost je shora omezena konstantou $k \in \mathbb{N}$.

Jelikož pracujeme s *ohodnoceným* grafem, namísto maximální hloubky zanoření si budeme pamatovat *práh* (anglicky *treshold*) maximálního zanoření. Jestliže pak stoupne hodnota $f(v)$ nad hodnotu prahu, nepokračujeme v prohledávání dalších úrovní. V každé iteraci algoritmu se práh zvýší o minimální hodnotu, která je potřebná k navštívení další úrovně v prohledávání. Pro lepší představu poslouží pseudokód 4.6.

Z popisu algoritmu tedy jasně vyplývá, že jediné ukládané informace jsou vrcholy, které se nachází na cestě mezi aktuálně prohledávaným a počátečním vrcholem, tedy prostorová složitost odpovídá $O(d)$.

## 4.8 Skokové prohledávání

Tato sekce přejímá obrázky a algoritmy z článku [25].

Jako poslední algoritmus pro hledání cesty mezi dvěma vrcholy si představíme *skokové prohledávání* (anglicky *jump point search*, ve zkratce *JPS*). Jedná se o speciální případ algoritmu A*, který pracuje pouze na *ohodnocené osmicestné čtvercové mřížce*. V té lze typicky mezi dvěma libovolnými vrcholy najít více stejně dlouhých cest. Této vlastnosti říkáme *symetrie cest*. V roce 2011 vyšel článek [25] australských informatiků, kteří se jako první zabývali odstraněním zbytečného prohledávání symetrických cest. Výsledkem jejich snažení je pak právě algoritmus JPS.

Základním stavebním kamenem algoritmu JPS je sada *prořezávacích pravidel*. Jejich cílem je pro daný vrchol $x$ a jeho předchůdce $p(x)$ identifikovat vrcholy, jejichž prohledání nepovede k nalezení optimální cesty mezi předchůdcem $p(x)$ a cílovým vrcholem. Tyto vrcholy pak vyloučíme z množiny následníků. Množinu následníků bez prořezaných vrcholů budeme označovat $PrunedSucc(x)$. Jinými slovy porovnáváme dvě cesty $P_1$ a $P_2$. Obě dvě začínají ve vrcholu $p(x)$ a končí v následníkovi $n$ vrcholu $x$, avšak cesta $P_1$ obsahuje jako druhý vrchol právě vrchol $x$, na rozdíl od $P_2$,

```
Input: G = (V, E, ϵ, ω) ohodnocený graf ; s počáteční vrchol ; goal
       cílový vrchol ; h heuristická funkce
1  Function IDA*(G, s, goal, h):
2      bound = h(s)
3      path = [s]
4      loop
5          bound = search(path,0,bound)
6          if bound = ∞ then
7              return noPathExists()
8          end
9      end
10 Function search(path, g, bound):
11     v ← path.last()
12     if f(v) > bound then
13         return f(v)
14     end
15     else if v = goal then
16         exitAndReturn(path);
17     end
18     new_bound = ∞
19     forall w ∈ Succ(v) do
20         if w ∈ path then
21             Continue
22         end
23         path.push_back(w)
24         t = search(path, g + ω(v, w), bound)
25         new_bound = min(new_bound, t)
26         path.pop_back(w)
27     end
```

**Algoritmus 4.6:** IDA* algoritmus

která tento vrchol vůbec neobsahuje. Problém si rozdělíme na dva menší podproblémy, podle směru cesty z předchůdce $p(x)$ do $x$:

**Přímý směr:** Prořežeme všechny vrcholy $n \in Succ(x)$, pro které platí:

$$\text{len}(\langle p(x), \ldots, n\rangle \setminus \{x\}) \leq \text{len}(\langle p(x), x, n\rangle),$$

kde $\langle p(x), x, n\rangle$ je cesta mezi vrcholy $p(x)$ a $x$. Příklad tohoto prořezání je na obrázku 4.7(a), kde jsou šedou barvou označeny prořezané vrcholy.

**Diagonální směr:** Prořežeme všechny vrcholy $n \in Succ(x)$, pro které platí:

$$\text{len}(\langle p(x), \ldots, n\rangle \setminus \{x\}) < \text{len}(\langle p(x), x, n\rangle).$$

Jediním rozdílem je změna nerovnosti na ostře menší. Příklad tohoto prořezání je na obrázku 4.7(c).

Všechny neprořezané vrcholy, tedy vrcholy z množiny $PrunedSucc(v)$, nazýváme *přirozenými* následníky vrcholu $x$. Ty jsou v ukázkách označeny bílou barvou. Poznamenejme, že pokud je vrchol $x$ počáteční vrchol, pak nejsou žádné vrcholy prořezány.

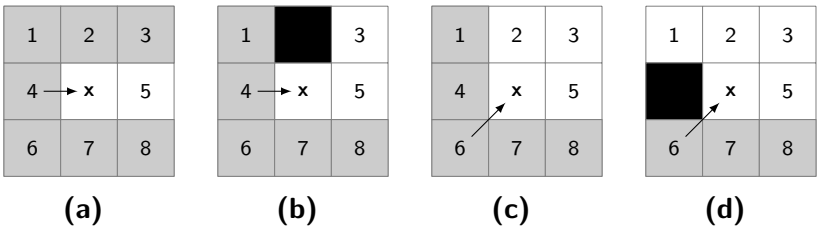
Pokud množina následníků $Succ(x)$ obsahuje *překážku* (anglicky *obstacle*), může to vést k tomu, že nebudeme moci nějaký vrchol prořezat, neboť nebude existovat alternativní kratší cesta do tohoto vrcholu. Takový vrchol nazveme *vynucený*.

**Definice 4.8.** Vrchol $n \in Succ(x)$ je *vynucený* právě tehdy, když

1. $n$ není *přirozeným* následníkem vrcholu $x$
2. a $\text{len}(\langle p(x), x, n\rangle) < \text{len}(\langle p(x), \ldots, n\rangle \setminus \{x\})$.

Příklady vynucených následníků jsou pak na obrázcích 4.7(b) a 4.7(d).

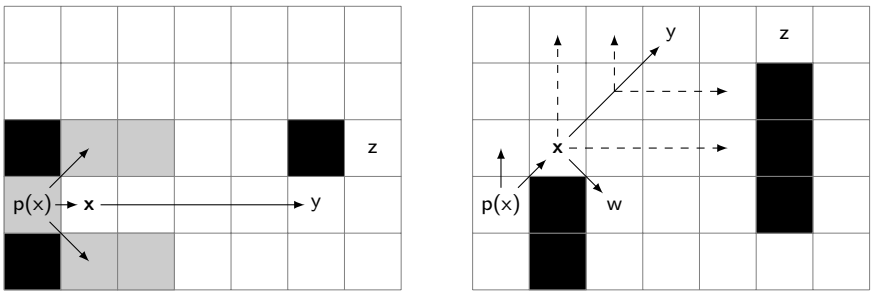
Pro snadnější orientaci v následující definici a následně i v pseudokódu budeme směr mezi dvěma sousedními vrcholy označovat jako $\vec{d}$. Je-li tento směr diagonální, pak je jistě možné ho rozložit na dva přímé navzájem kolmé směry $\vec{d}_1$ a $\vec{d}_2$. Zároveň zapisujeme $y = x + k \cdot \vec{d}$, jestliže vrchol $y$ leží $k$ jednotkových kroků ve směru $\vec{d}$.




**Obrázek 4.7:** Prořezávací pravidla algoritmu JPS

**Definice 4.9.** Vrchol $y$ nazveme *skokem* (anglicky *jump point*) *z vrcholu* $x$ *ve směru* $\vec{d}$, jestliže $y$ minimalizuje hodnotu $k$ ve výrazu $y = x + k \cdot \vec{d}$ a platí jedna z následujících podmínek:

1. Vrchol $y$ je koncový vrchol.
2. Vrchol $y$ má alespoň jednoho vynuceného souseda.
3. $\vec{d}$ je diagonální a existuje vrchol $z = y + k_i \cdot \vec{d_i}$, kde $k_i \in \mathbb{N}$ a $\vec{d_i} \in \{\vec{d_1}, \vec{d_2}\}$, takový, že $z$ je skokem z vrcholu $y$ splňující 1. nebo 2. bod této definice.




**(a)** Vertikální a horizontální prohledávání

**(b)** Diagonální prohledávání

**Obrázek 4.9:** Vizualizace JPS algoritmu

Po zavedení předchozích pojmů se konečně dostáváme k popisu samotného algoritmu. Jedná se stále o algoritmus A* s tím rozdílem, že speciálně volíme množinu následníků. V pseudokódu 4.7 tedy uvádíme pouze části kódu, které se od A* liší. Asymptotická časová a prostorová složitost je stejná, jako v případě algoritmu A*.

```
Input: x prohledávaný vrchol ; s počáteční vrchol ; g cílový vrchol ;
       d⃗ směr
Function PruneSuccessors(x, s, g, d⃗):
    PrunedSucc ← ∅
    forall n ∈ applyPruneRules(x, Succ(x), d⃗) do
        n ← jump(x, direction(x, n), g)
        PrunedSucc ← PrunedSucc ∪ {n}
    end
    return PrunedSucc
Function jump(x, d⃗, g):
    n ← step(x, d⃗)
    if n je překážka nebo mimo mřížku then
        return null
    end
    else if n = g then
        return n
    end
    else if ∃n′ ∈ Succ(n) kde n′ je vynucený then
        return n
    end
    else if d⃗ je diagonální then
        forall i ∈ {1, 2} do
            if jump(n, d⃗_i, g) ≠ null then
                return n
            end
        end
    end
    return jump(n, d⃗, g)
```

**Algoritmus 4.7:** JPS algoritmus

## 4.9 Floyd-Warhallův algoritmus

Na závěr této kapitoly si představíme algoritmus, jehož cílem je najít matici vzdáleností mezi každými dvěma vrcholy v grafu. Tento algoritmus poprvé představil v roce 1962 Rober Floyd ve svém článku [26], nezávisle na něm pak Stephan Warshall. Algoritmus využívá *dynamické programování* a jedno z jeho využití je například hledání tranzitivního uzávěru relace $R$.

### 4.9.1 Asymtotická složitost

Při pohledu na pseudokód 4.8 okamžitě konstatujeme časovou složitost $O(|V|^3)$. Lze pak snadno dokázat, že pokud si budeme v paměti udržovat pouze jednu kopii matice vzdáleností a tu přepisovat, algoritmus bude stále fungovat korektně.

Existují i rychlejší algoritmy pro hledání matice vzdáleností. Jedním z nich je například *Johnsonův algoritmus*, který funguje v čase $O(|V|^2 \cdot \log(|V|) + |V| \cdot |E|)$ a využívá *Fibonacciho haldu*.

| | |
|---|---|
| **Časová složitost:** | $O(\|V\|^3)$ |
| **Prostorová složitost:** | $O(\|V\|^2)$ |

**Tabulka 4.3:** Asymptotická složitost Floyd-Warshallova algoritmu

```
   Input: x prohledávaný vrchol
 1 Function FloydWarshall(G):
 2 |   dist := matice vzdáleností
   |   // Inicializace matice
 3 |   forall v ∈ V do
 4 |   |   forall w ∈ V do
 5 |   |   |   if v = w then
 6 |   |   |   |   dist[v][w] = 0
 7 |   |   |   else
 8 |   |   |   |   dist[v][w] = ∞
 9 |   forall (v, w) ∈ E do
10 |   |   dist[v][w] = ω((v, w))
   |   // Run FloydWarshall
11 |   forall k ∈ V do
12 |   |   forall i ∈ V do
13 |   |   |   if dist[i][k] ≠ ∞ then
14 |   |   |   |   forall j ∈ V do
15 |   |   |   |   |   if dist[k][j] ≠ ∞ then
16 |   |   |   |   |   |   dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
   |   // Kontrola záporné smyčky
17 |   forall n ∈ V do
18 |   |   if dist[n][v] < 0 then
19 |   |   |   return negativeCycle()
20 |   return dist
```

**Algoritmus 4.8:** Floyd-Warshallův algoritmus

Kapitola 5

# Implementace

*„C++ nebyla úplně ideální volba.“*

— Jan Trávníček

*Algoritmová knihovna* ALib[1] (dříve *Automatová knihovna*) vzniká na Fakultě informačních technologií již od roku 2013. Původně byla zaměřena pouze na problematiku automatů a gramatik, avšak v průběhu vývoje se přidávala další a další odvětví teoretické informatiky. Reprezentace grafů a první grafové algoritmy byly přidány v roce 2015 Davidem Roscou v rámci bakalářské práce [2] zaměřené na téma isomorfismu planárních grafů. V roce 2017 přispěl Jan Brož svou bakalářskou prací [1] algoritmy pro hledání kostry, minimálního řezu a maximálního toku v grafu. Zároveň upravil dosavadní implementaci datových struktur reprezentující vrchol, hranu a graf v knihovně.

Celá knihovna je momentálně psaná v jazyce C++ verze 14. Do budoucna se plánuje přejít na novější verzi 17, která přinese mnoho výrazných zjednodušení, počínaje *Using-declaration* a končeje přidáním nových členských funkcí třídám v STL knihovně. V celé knihovně je hojně využívána abstrakce nad datovými typy pomocí *šablon*, které jazyk C++ nabízí. Hlavním přínosem využívání šablon je univerzalita. Koncový uživatel knihovny je pak schopný použít připravené algoritmy s relativně malým úsilím a bez znalosti vnitřní implementace. Standard jazyka C++ má však v tomto ohledu z implementačních důvodů svá omezení, kterými je vývojář limitován. Právě na tuto

[1] Zdrojové kódy knihovny jsou veřejně přístupné na fakultním Gitlabu `https://gitlab.fit.cvut.cz/algorithms-library-toolkit/automata-library`

problematiku naráží citát z úvodu kapitoly, ačkoliv se sluší podotknout, že byl pronesen s jistou dávkou nadsázky.

## 5.1 Analýza existujících řešení

Při návrhu nové implementace byly analyzovány již existující grafové knihovny. Hlavním cílem bylo vytipovat výhody a nevýhody jednotlivých řešení. V sekci 5.4 se nachází detailní shrnutí této analýzy. Při výběru analyzovaných knihoven byl kladen velký důraz na rozmanitost návrhových přístupů. Analyzovány byly následující knihovny:

### 5.1.1 The Boost Graph Library

*The Boost Graph Library* [27] (ve zkratce *BGL*) je pravděpodobně největší a nejznámější grafová knihovna. Nachází se v ní velké množství grafových algoritmů, které jsou navíc velmi dobře optimalizovány. Knihovna je psaná v jazyce C++ a využívá všechny moderní konstrukty, které jazyk nabízí, včetně šablon. Existuje rovněž verze knihovny přizpůsobená pro maximální efektivitu paralelních výpočtů na víceprocesorových systémech.

### 5.1.2 Petgraph

Knihovna *Petgraph* [28] je v porovnání s BGL o poznání menší. Je psána v jazyce *Rust* a využívá prvky objektově orientovaného programování zvané *trait*. Knihovna, stejně jako jazyk samotný, je aktuálně ve vývoji, a tak se rozhraní knihovny velmi často mění.

### 5.1.3 JGraphT

Poslední analyzovaná knihovna *JGraphT* [29] je psaná v jazyce Java. K problému reprezentace datových struktur přistupuje velmi odlišně v porovnání s BGL a Petgraph. Řešení využívá třídní polymorfismus spolu s *rozhraním*, které jazyk Java nabízí. Detailnější analýza je opět v sekci 5.4.

## 5.2 Nedostatky původní implementace

Původní implementace grafových algoritmů a datových typů šablony vůbec nevyužívá. Tato skutečnost činí knihovnu obtížněji použitelnou v reálném nasazení. Navíc je zde velký rozkol ve stylu implementace této části knihovny a částmi ostatními.

Dalším problémem je volba rozložení zodpovědnosti za uchování určitých informací. Pro příklad uveďme informaci o ohodnocení hrany v grafu. V původní implementaci je tento údaj uložen ve třídě `DirectedGraph` (popřípadě `UndirectedGraph`). To zabraňuje vytvoření neohodnoceného grafu, který by neobsahoval tento zbytečný třídní atribut. Zároveň tento přístup neumožňuje využití kontroly datových typů v čase kompilace, což je jedna z nejdůležitějších výhod typovaných jazyků. Namísto toho jsou za běhu programu vyhazovány výjimky, což je v praxi velmi nepraktické a vyžaduje to psaní důkladnějších a delších testů.

S volbou rozložení zodpovědnosti ohledně uchování informací úzce souvisí problém s rozšiřitelností datových struktur a adaptaci problému na již existující struktury. V teorii grafů jsou různé problémy, které vyžadují ukládání různých informací jak na vrcholy tak na hrany v grafu. Například při hledání obarvení grafu je nutné na každém vrcholu držet informaci o aktuální barvě. Původní implementace s touto skutečností vůbec nepočítá, a tak bez hlubšího zásahu do již jednou zmíněných tříd není možné tyto problémy řešit.

Z výše uvedených důvodů byla celá grafová část knihovny vyčleněna do speciálního experimentálního modulu.

## 5.3 Požadavky na novou implementaci

Nová implementace má za cíl odstranit všechny dříve uvedené nedostatky. Navíc by měla připravit solidní základy pro budoucí rozšíření této části knihovny, a to jak o nové grafové algoritmy, tak o nové doposud nenaimplementované datové struktury.

Knihovna ALib obsahuje konzolové rozhraní (ve zkratce *CLI* z anglického názvu *Command-line interface*) pro snadné a rychlé spuštění algoritmů. Aby bylo možné toto rozhraní plně využívat, je nutné provést tzv. *registraci*. Pro registraci datového typu je navíc nutné provést *normalizaci* datových typů. Této problematice se hlouběji věnuje jedna z následujících sekcí, avšak nová implementace by měla být plně kompatibilní s knihovním CLI.

Hlavní požadavky na novou implementaci jsou:

- univerzalita dosažená využitím *šablon* jazyka C++,
- snadná rozšiřitelnost,
- kontrola datových typů v času kompilace,
- dodržení jednotného stylu implementace napříč knihovními moduly,
- efektivita,
- kompatibilita s knihovním CLI
- a zamezení duplicity kódu.

## 5.4 Možné přístupy k nové implementaci

Tato sekce shrne všechny námi uvažované (a následně zamítnuté) možnosti implementace a krátce okomentuje jejich výhody a nevýhody. Vřele doporučuji důkladné pročtení této sekce všem budoucím přispěvatelům, kteří budou rozšiřovat grafový modul knihovny.

### 5.4.1 Naivní přístup

První možný přístup je naivní implementace všech datových typů pomocí vlastní třídy. V knihovně by tak vzniklo mnoho tříd reprezentujících všechny možné typy grafu například:

- orientovaný graf,
- orientovaný multigraf,
- ohodnocený orientovaný graf,

- ohodnocený orientovaný multigraf,
- ohodnocený vrcholově obarvený orientovaný graf,
- ohodnocený vrcholově obarvený orientovaný multigraf, atd.

Ačkoliv je toto řešení v některých grafových knihovnách využito (například v *JGraphT*), vede na exponenciální množství tříd závisejícím na počtu vlastností grafů. Zároveň se zde ve velké míře opakuje kód, a tedy následná modifikace či oprava je velmi obtížná. Opakování kódu lze částečně odstranit využitím dědění, v konečném důsledku ale k úplnému odstranění duplicity nedojde.

### 5.4.2 Vícenásobné dědění

Jazyk C++, na rozdíl od většiny ostatních programovacích jazyků, nabízí možnost vícenásobného dědění. Na první pohled by se mohlo jevit využití této vlastnosti jako ideální volbou. Pro příklad uvažme situaci, kdy chceme pracovat s *ohodnoceným vrcholově obarveným orientovaným multigrafem*. V této implementaci by knihovna nabízela třídy zastupující *orientovaný graf*, *ohodnocený graf*, *vrcholově obarvený graf* a *multigraf*. Výsledný datový typ by bylo možné složit využitím právě vícenásobného dědění od všech těchto tříd.

Hlavní nevýhodou tohoto konceptu je problém zvaný *The Diamond of Dread*. Nechť máme nadtřídu `A` a její dva potomky `B` a `C`. Pokud bychom vytvořili potomka `D`, který dědí od `B` a `C`, dochází k překrývání členských funkcí a proměnných. Tato skutečnost vede k výraznému snížení čitelnosti kódu. Nejenom z tohoto důvodu je komunitou C++ vývojářů výrazně doporučeno se vícenásobnému dědění vyhnout a využít jiné prostředky, jak dosáhnout stejného výsledku.

### 5.4.3 Trait implementace

Moderní jazyky jako například *Scala* nebo *Rust* nabízejí možnost využití traitů (jeden z možných překladů tohoto slova do češtiny je *rys*, avšak v programátorské komunitě není tento překlad příliš využíván). Jedná se o kolekci metod (v terminologii C++ kolekci členských funkcí), které je

možné využít při práci s danou třídou. C++ tento objektově orientovaný návrh přímo nenabízí, nicméně ho lze realizovat využitím *rozhraní* (anglicky *interface*).

Takto zvolená implementace by fungovala správně a splňovala by všechna námi zvolená kritéria. Hlavní nevýhodou tohoto řešení je ale výrazný rozdíl v implementaci této části a zbytku knihovny. Také z hlediska filozofie jazyka by se jednalo o obcházení zavedených standardů a postupů. Z těchto důvodů nebyl tento přístup vybrán, ačkoli se jedná o plně validní řešení.

## 5.5 Popis výsledné implementace

Pro splnění všech vznesených požadavků bylo nutné změnit umístění uložení některých informací o grafu. Zároveň nová implementace využívá šablony jazyka C++, a tak není vynucené používat konkrétní datový typ pro reprezentaci hrany nebo vrcholu. Jediné, co musí být splněno, je předepsané základní rozhraní. Pro usnadnění používání jsou již některé fundamentální třídy připraveny, avšak nic nebrání jejich úpravě či rozšíření. Při dodržení stanoveného rozhraní navíc není nutné po jejich modifikaci nikterak upravovat implementaci algoritmů.

Ve výsledné implementaci může být vrchol grafu reprezentován libovolnou třídou splňující základní rozhraní porovnání a výpisu (přetížení příslušných operátorů). V základních případech bude vrchol grafu reprezentován primitivním datovým typem například `int`. V těch složitějších je možné primitivní datový typ nahradit třídou, která jako členské atributy bude obsahovat všechny potřebné informace související s daným vrcholem. Knihovna aktuálně obsahuje třídu `node::Node`, která je používána u některých starších algoritmů, které ještě nevyužívají šablony.

Podobně pak hranou může být libovolná třída obsahující dvojici vrcholů, která splňuje rozhraní knihovní třídy `std::pair`. Výběr tohoto rozhraní není náhodný a vychází z matematické definice, kde hrana představuje dvojici vrcholů. Stejně jako u vrcholu je možné třídu reprezentující hranu rozšířit o další členské atributy a vytvořit tak libovolnou kombinaci informací, které budou v této třídě uloženy. V knihovně se aktuálně nachází předepsané rozhraní pro

- ohodnocené hrany `edge::WeightedEdge`
- a hrany s kapacitou `edge::CapacityEdge`.

Algoritmy pro hledání cest v ohodnoceném grafu pak spoléhají na dodržení tohoto rozhraní. V případě nedodržení je chyba odhalena v času kompilace, což je jeden z požadavků na novou implementaci.

Díky tomuto modulárnímu přístupu je budoucí rozšíření o další typy grafů velmi jednoduché. Například budou-li se v budoucnu do knihovny přidávat algoritmy pracující s obarvenými hranami, jedinou nutnou modifikací bude předepsání rozhraní pro obarvené hrany.

Na rozdíl od předchozí implementace není informace o orientaci hrany uložena v samotné hraně. To se může z počátku jevit jako velmi nelogické rozhodnutí, které je v rozporu s matematickou definicí. Výsledkem tohoto kompromisu je však výrazné zvýšení efektivity a rychlosti všech grafových algoritmů. Zodpovědnost za uložení této informace byla totiž přenesena na třídu grafu, která podle svého typu využívá optimální vnitřní datové struktury pro uložení množiny vrcholů a hran. Finální implementace obsahuje 6 následujících tříd reprezentující základní typy grafů:

- `graph::UndirectedGraph`,
- `graph::UndirectedMultiGraph`,
- `graph::DirectedGraph`,
- `graph::DirectedMultiGraph`,
- `graph::MixedGraph`
- a `graph::MixedMultiGraph`.

Tyto třídy lze specializovat výběrem příslušných datových typů reprezentující vrchol a hranu. S takto obecnou implementací je možné pokrýt většinu algoritmů z teorie grafů, počínaje hledáním nejkratší cesty a končeje například řešením problému barvení grafu.

V některých případech je výhodné ustoupit od obecnosti a vytvořit speciální implementaci jistého typu grafu. Například osmisměrnou čtvercovou

mřížku lze reprezentovat s využitím obecné třídy neorientovaného grafu. Daleko výhodnější je však vytvořit novou třídu, která bude splňovat stejné rozhraní, ale bude efektivněji ukládat informace o překážkách v mřížce a o ohodnocení jednotlivých hran. Z tohoto důvodu byla v knihovně navíc vytvořena speciální reprezentace čtvercových mřížek `grid::SqaureGrid`. Zároveň implementace počítá s budoucím rozšířením například o šestihranné mřížky.

Při registraci do knihovního CLI naráží implementace datových struktur na limity šablon jazyka C++. Pro úspěšnou registraci je totiž nutné vytvoření obalovacích tříd (tvz. *wrappers*) okolo každé třídy reprezentující základní typ grafu. To má za následek vznik exponenciálního množství pomocných tříd vzhledem k počtu vlastností grafu. Tomuto problému se při stávajícím stavu knihovny nelze vyhnout, avšak díky univerzalitě finální implementace je množství duplicitního kódu sníženo na minimum. V budoucnu je plánováno přidání JIT (*Just-in-time*) kompilace, která výrazně zjednoduší registrace a obejde limity jazyka C++.

## 5.6 Testování

Nad rámec zadání byly pro všechny nově implementované části přidány testy kontrolující jejich funkčnost. U datových typů je kontrolováno především správné chování při přidávání nových vrcholů a hran.

Pro grafové algoritmy byly vytvořeny dvě sady testů. První z nich jsou ručně připravené „zákeřné“ vstupy. Jejich příprava probíhala se znalostí jednotlivých algoritmů a cílila na testovaní vytipovaných kritických míst. V druhé sadě se nachází testy náhodné, které pro rozumně velké vygenerované vstupy porovnávají výstupy jednotlivých algoritmů.

## 5.7 Kompilace

Knihovna je aktuálně kompilovaná pomocí nástroje *GNU Make*. Jedná se o klasickou a prověřenou volbu pro automatizaci kompilace u větších projektů na platformě OS Linux. Avšak mnoho dnešních vývojových prostředí (například *CLion*) vyžaduje pro správné zvýrazňování syntaxe,

inteligetní napovídání a mnoho dalších funkcí použití novějšího nástroje *CMake*. Aby bylo možné tato vývojová prostředí využívat, vznikl nad rámec této práce skript[1] psaný v jazyce Python 3, který automaticky vygeneruje všechny potřebné soubory potřebné pro kompilaci s využitím nástroje *CMake*.

## 5.8 Budoucí vývoj

Finální implementace poskytuje svou univerzálností solidní základy pro budoucí rozšiřování. Vše je připravené pro přidání dalších grafových algoritmů společně se speciálními datovými strukturami. Bylo také myšleno na rozšíření funkcionalit, které knihovna nabízí. Další práce by se tak mohla zaměřit na vzájemný převod mezi vnitřní reprezentací grafů a nějakou formou strukturovaného výstupního dokumentu, ať už ve formátu XML, JSON, YML nebo TikZ.

Pravděpodobně nejdůležitější změnou bude však přechod od datových typů `std::map` a `std::set` k `std::unordered_map` a `std::unordered_set`. Tím by se výrazně zvýšila efektivita všech grafových algoritmů. Tomuto přechodu ovšem brání chybějící knihovní podpora pro hašování na základní třídě `object::Object`.

[1] Skript je volně dostupný na adrese `https://gitlab.com/ctu-fit/alib-cmake`

Kapitola 6

# Měření algoritmů

V předchozích kapitolách jsme si představili několik grafových algoritmů na hledání cest v grafech. Popsali jsme jejich vlastnosti a rozebrali možné optimalizace. Následně jsme shrnuli způsob, jakým byly jednotlivé datové struktury a algoritmy naimplementovány. V této kapitole se budeme zabývat reálným měřením a výsledky tohoto měření porovnáme s očekávanými teoretickými vlastnostmi.

Při měření algoritmů nás budou zajímat tato následující kritéria:

- počet expandovaných vrcholů,
- doba běhu
- a velikost alokované paměti.

Počet expandovaných vrcholů budeme měřit pomocí lambda funkce, která je předávána jako volitelný parametr do každého algoritmu. Pro měření času využijeme standardních funkcí, které od verze C++11 nabízí STL knihovna. Poslední kritérium, velikost alokované paměti, budeme měřit externím programem *valgrind*, konkrétně jedním z jeho nástrojů *massif*. Jedná se o klasický způsob kontroly a měření hlavní paměti na operačním systému Linux. Na ukázce kódu B.13 můžeme vidět minimální konfiguraci pro jednotlivá měření.

Jelikož různé algoritmy vyžadují různé typy grafů, provedeme vícero měření. V každém z těchto měření se zaměříme na speciální typ grafu a test provedeme pouze pro algoritmy, které mají pro tento vstup smysl.

Pro účely měření a testování vnikl samostatný knihovní modul nazvaný *alib2graph_measure*. V něm můžeme vidět jednak ukázkové příklady deklarace a inicializace datových struktur grafů tak příklady volání jednotlivých algoritmů. Modul zároveň může sloužit k rekonstrukci měření a kontrole níže uvedených výsledků.

Všechna níže uvedená měření byla provedena na počítači s konfigurací:

- **CPU:** 1.6GHz Intel Core-4200U.
- **RAM:** 4.096 MB DDR3.
- **OS:** Fedora 27.

*Poznámka.* Naměřené hodnoty budeme uvádět ve formě tabulky a následně vykreslovat do histogramu. Jelikož měříme tři různá kritéria, kde každé kritérium má jinou jednotku, měli bychom pro každé měření uvádět tři grafy. Abychom však zbytečně nezvyšovali počet stránek práce, budeme vynášet všechna naměřená data do jednoho grafu s dvěma $y$ osami. Levá osa je společná pro počet expandovaných vrcholů a dobu běhu v milisekundách. Pravá osa pak představuje velikost alokované paměti v kilobytech. Na závěr poznamenejme, že pro obě osy používáme logaritmické měřítko.

*Poznámka.* Algoritmy IDDFS a IDA* by bylo ideální testovat a měřit na konkrétních reálných problémech jako například při řešení Loydovi 15 nebo Rubicovi kostky. Tím bychom se však již příliš vzdálili od zadání práce a implementace těchto testů by coby do funkčnosti knihovně ALib příliš nepřidala. Z těchto důvodů byla upřednostněna varianta omezeného testování a měření těchto algoritmů, jejíž součástí je i implementace náhodného generování grafů, mřížek a stromů o zadaných parametrech.

## 6.1 Měření na osmisměrné čtvercové mřížce

Abychom se při měření co nejvíce přiblížili reálným problémů, použijeme mapy [30] z her *Dragon Age: Origins* a *Warcraft 3*. Jedná se o klasické

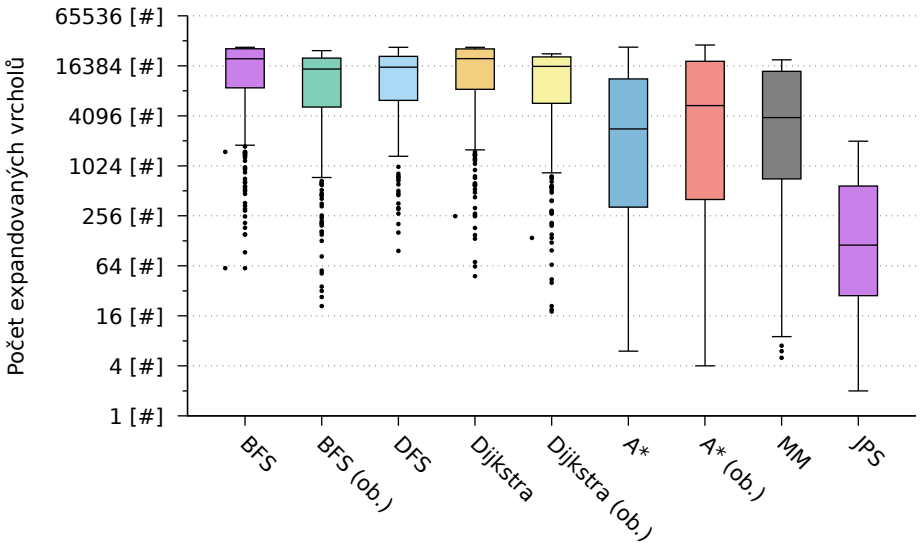
| Scénář: | *brc000d.map.scen* | | *lak506d.map.scen* | | *arena2.map.scen* | |
|---|---|---|---|---|---|---|
| Detaily: | 261 × 257 | 850 | 205 × 194 | 1171 | 206 × 281 | 930 |
| | [#] | [ms] | [#] | [ms] | [#] | [ms] |
| **BFS** | 12028 | 282 | 7964 | 180 | 14365 | 339 |
| **BFS (ob.)** | 3655 | 92 | 8343 | 180 | 12110 | 286 |
| **DFS** | 11818 | 294 | 7378 | 157 | 13268 | 421 |
| **Bellman-Ford** | 140685 | 6731 | 140089 | 5470 | 108029 | 6535 |
| **SPFA** | 26235 | 836 | 19014 | 541 | 35408 | 1007 |
| **Dijkstra** | 11934 | 400 | 7953 | 208 | 14350 | 409 |
| **Dijkstra (ob.)** | 3937 | 115 | 8598 | 231 | 11944 | 336 |
| **A*** | 7599 | 309 | 5354 | 202 | 5783 | 195 |
| **A* (ob.)** | 1422 | 52 | 10250 | 427 | 10351 | 379 |
| **MM** | 1117 | 50 | 7785 | 369 | 6824 | 300 |
| **JPS** | 507 | 174 | 284 | 43 | 98 | 50 |

**Tabulka 6.1:** Výsledky měření pro vybrané scénáře

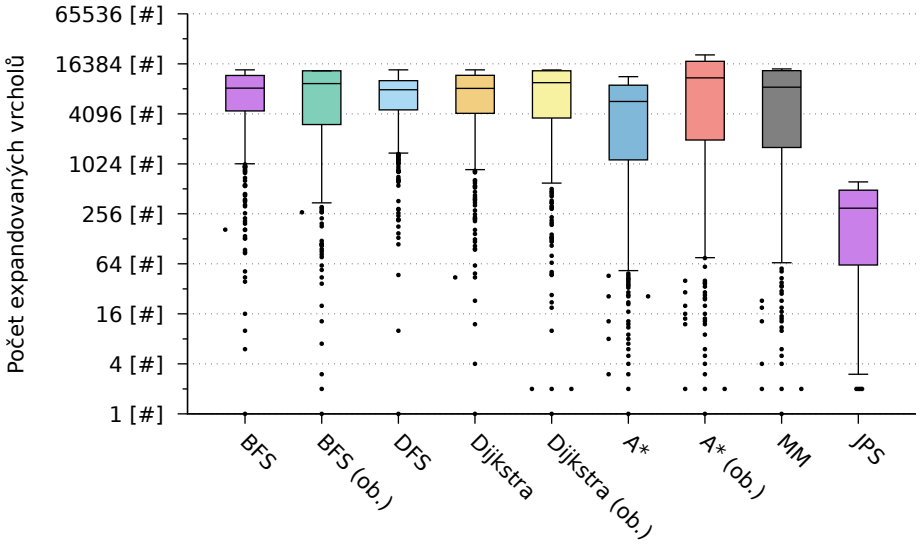
osmisměrné čtvercové mřížky o různých velikostech. Ke každé mapě je navíc sada scénářů s počátečními a koncovými vrcholy, mezi kterými budeme hledat cestu.

Pro základní vizualizaci běhu jednotlivých algoritmů využijeme upravenou mapu *den203d.map*. Na obrázcích 6.3 můžeme vidět výsledky některých vybraných algoritmů, kde počáteční (resp. koncový) vrchol je znázorněn žlutě, nejkratší cesta mezi nimi zeleně a červeně jsou pak zvýrazněny všechny vrcholy, které byly při běhu algoritmu expandovány.

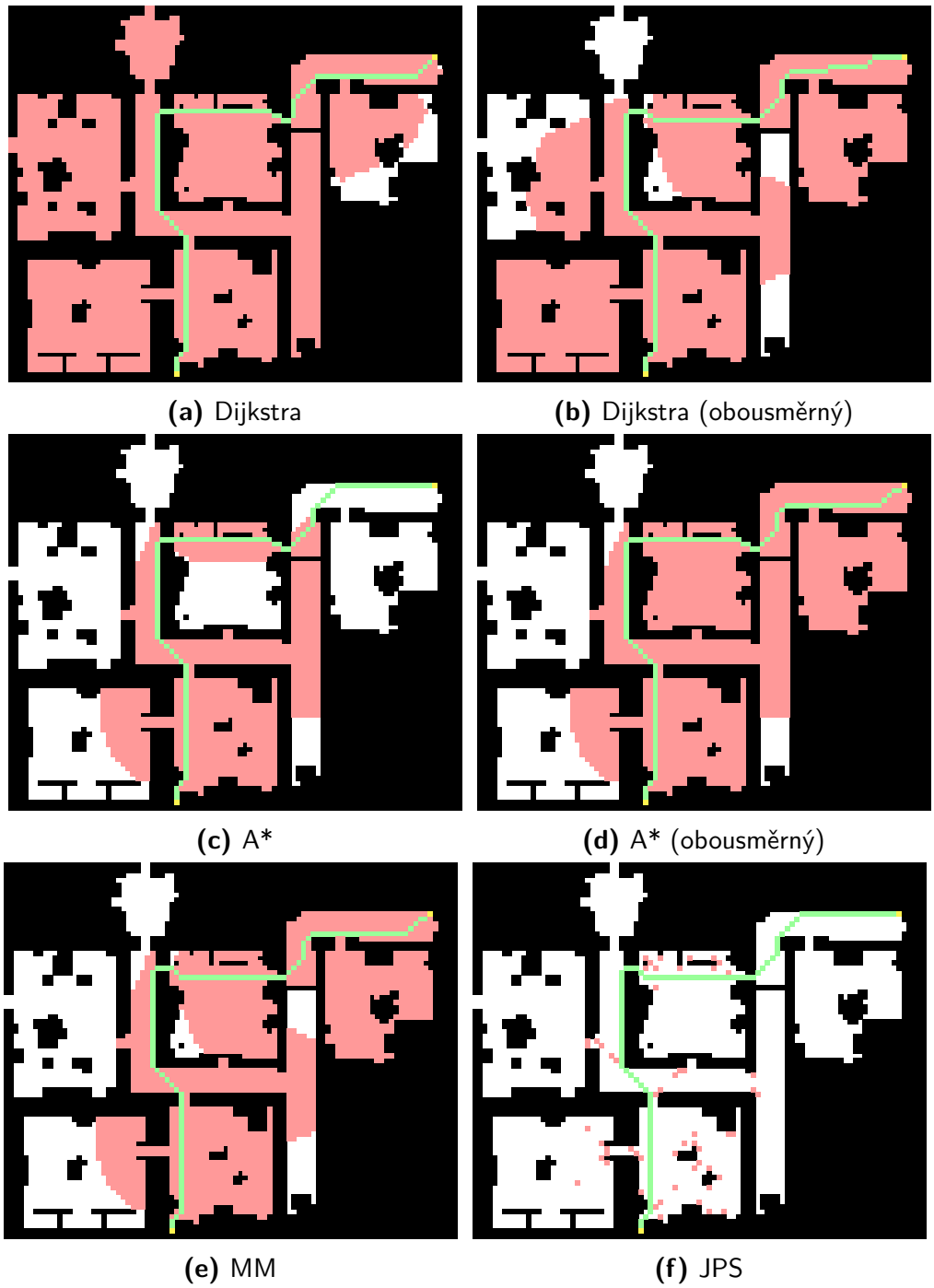
Měření velikosti alokované paměti je bohužel časově velmi náročné a jeho plná automatizace by vyžadovala nemalé množství pomocných skriptů. Navíc by naměřené hodnoty byly pouze orientační, jelikož reálná hodnota je silně ovlivněna povahou problému a prostředím v jakém je algoritmus měřen. Z těchto důvodů vynecháme měření této veličiny z komplexního zátěžového testování a její měření provedeme na speciálním příkladě zvlášť. V tabulce 6.1 můžeme vidět výsledky tohoto automatizovaného měření, kde v každém sloupci jsou průměrné hodnoty pro scénář z prvního řádku. Druhý řádek tabulky ukazuje velikost mapy a počet testů spuštěných nad touto mapou. Naměřené hodnoty jsou pak vyneseny do krabicového grafu na obrázcích 6.1 a 6.2.




**Obrázek 6.1:** Krabicový graf výsledků měření pro scénář *den500d.map.scen*




**Obrázek 6.2:** Krabicový graf výsledků měření pro scénář *lak506d.map.scen*



**(a)** Dijkstra **(b)** Dijkstra (obousměrný)

**(c)** A* **(d)** A* (obousměrný)

**(e)** MM **(f)** JPS

**Obrázek 6.3:** Vizualizace počtu expandovaných vrcholů pro mapu *den203d.map*

Pro detailnější rozbor včetně měření velikosti alokované paměti využijeme mapu *den500d.map*, která má velikost $417 \times 288$ a obsahuje 30070 vrcholů. Výsledky měření najdeme v tabulce 6.2, kde první sloupec obsahuje jméno algoritmu, druhý počet expandovaných vrcholů, třetí dobu běhu (jedná se o průměrnou hodnotu z 10 nezávislých měření) a čtvrtý velikost alokované paměti po odečtení velikosti potřebné k uložení grafu, která činí 8,2 MiB. Naměřené hodnoty jsou pak vynesené do grafu 6.5.

Povšimněme si zvýšené paměťové náročnosti u algoritmu DFS, která je způsobena rekurzivním voláním. Tento nárůst je možné odstranit volbou nerekurzivní implementace algoritmu. U Bellman-Fordova algoritmu i jeho optimalizace SPFA *není* zaručeno, že jednou expandovaný vrchol nebude již vícekrát expandován. Z tohoto důvodu je počet expandovaných vrcholů větší než počet vrcholů v grafu. Zároveň je vidět asymptotický rozdíl oproti ostatním algoritmům.

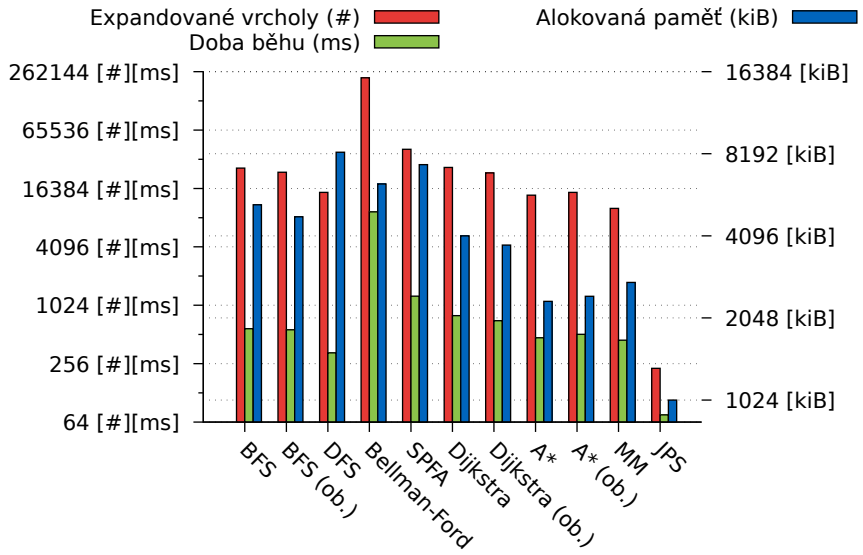
Z grafu můžeme také vypozorovat vysokou korelaci mezi výsledky měření pro algoritmus BFS (popř. obousměrné BFS) a Dijkstrovým algoritmem (popř. obousměrným Dijkstrovým algoritmem). To není překvapivé, jelikož tyto dva algoritmy si jsou svou myšlenkou i pořadím prohledávaných vrcholů na pravidelných grafech (tedy také na osmisměrné čtvercové mřížce) velmi podobné.

Dále vidíme, že obousměrná varianta A* expanduje více vrcholů než jednosměrná varianta. Toto pozorování tedy potvrzuje lemma 4.2. Jak je dále z dat vidět, v tomto konkrétním případě algoritmus MM tento nedostatek odstraňuje. Připomeňme však, že obecně neplatí, že by obousměrné prohledávání expandovalo méně vrcholů než prohledávání jednosměrné.

Poslední měřený algoritmus JPS se zdá být velmi efektivní, avšak je nutné si uvědomit, že algoritmus rekurzivně prochází i vrcholy, které nejsou označené jako expandované.

| Algoritmus | Exp. vrcholy (#) | Čas (ms) | Paměť (kiB) |
|---|---|---|---|
| **BFS** | 26538 | 587 | 5325 |
| **BFS (ob.)** | 24110 | 572 | 4813 |
| **DFS** | 14946 | 331 | 8294 |
| **Bellman-Ford** | 226930 | 9405 | 6349 |
| **SPFA** | 41571 | 1272 | 7475 |
| **Dijkstra** | 27009 | 800 | 4096 |
| **Dijkstra (ob.)** | 23706 | 710 | 3789 |
| **A*** | 13985 | 473 | 2355 |
| **A* (ob.)** | 14923 | 514 | 2458 |
| **MM** | 10219 | 447 | 2765 |
| **JPS** | 229 | 76 | 1024 |

**Tabulka 6.2:** Výsledky měření pro mapu *den500d.map*




**Obrázek 6.5:** Sloupcový graf výsledků měření pro mapu *den500d.map*

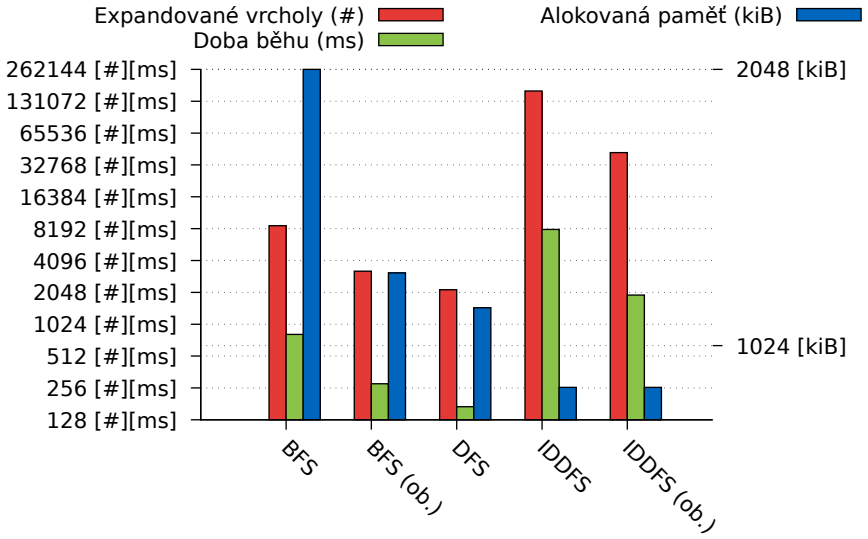
## 6.2 Měření na neohodnocených stromech

Algoritmy pro procházení grafu budeme testovat na náhodně vygenerovaných grafech, ze kterých následně pomocí *Jarník-Primova algoritmu* vytvoříme strom. Níže uvedená měření (tabulka 6.3 a graf 6.6) proběhla pro strom obsahující 10670 vrcholů a 21338 hran.

Z výsledků jasně vyplývá, že iterativně se prohlubující algoritmy expandují řádově více vrcholů. To úzce souvisí s dobou běhu algoritmu. Na druhou stranu je paměťová složitost nepatrně nižší. V testovacích podmínkách je totiž vrchol reprezentován pouze jako jednoduchý datový typ *long*. V reálném nasazení, kdy vrchol v grafu obsahuje daleko více informací, bude paměťový rozdíl výraznější.

| Algoritmus | Exp. vrcholy (#) | Čas (ms) | Paměť (kiB) |
|---|---|---|---|
| **BFS** | 8746 | 820 | 2048 |
| **BFS (ob.)** | 3244 | 280 | 1229 |
| **DFS** | 2167 | 170 | 1126 |
| **IDDFS** | 163648 | 8050 | 922 |
| **IDDFS (ob.)** | 42818 | 1930 | 922 |

**Tabulka 6.3:** Výsledky měření na neohodnocených stromech



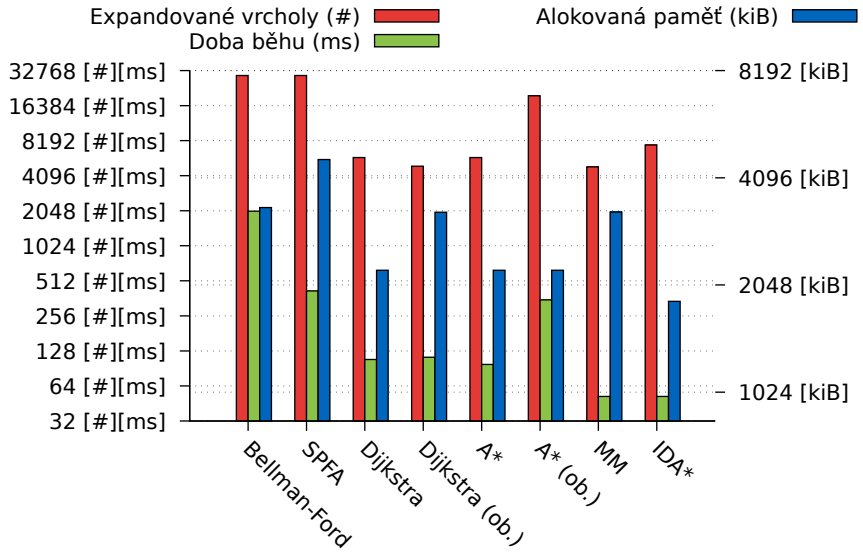

**Obrázek 6.6:** Sloupcový graf výsledků měření na neohodnocených stromech

## 6.3 Měření na ohodnocených stromech

Stejně jako v předchozí sekci vygenerujeme náhodný strom. Jelikož některé algoritmy potřebují pro svůj běh dopřednou (resp. zpětnou) heuristickou funkci, pustíme před samotným měřením 2× dijkstrův algoritmus z počátečního (resp. koncového) vrcholu. Při běhu algoritmu si pak budeme ukládat délku nejkratší nalezené cesty a tuto informaci následně využijeme v heuristické funkci. Níže uvedená měření (tabulka 6.4 a graf 6.7) proběhla pro strom obsahující 29700 vrcholů, 59398 hran a kde se ohodnocení hran pohybovalo v rozmezí 1–100.

| Algoritmus | Exp. vrcholy (#) | Čas (ms) | Paměť (kiB) |
|---|---|---|---|
| **Bellman-Ford** | 29700 | 2033 | 3379 |
| **SPFA** | 29700 | 419 | 4608 |
| **Dijkstra** | 5866 | 108 | 2253 |
| **Dijkstra (ob.)** | 4940 | 113 | 3277 |
| **A*** | 5866 | 98 | 2253 |
| **A* (ob.)** | 19891 | 351 | 2253 |
| **MM** | 4880 | 52 | 3280 |
| **IDA*** | 7522 | 52 | 1843 |

**Tabulka 6.4:** Výsledky měření na ohodnocených stromech




**Obrázek 6.7:** Sloupcový graf výsledků měření na ohodnocených stromech

# Závěr

Cílem práce bylo navrhnout, popsat a naimplementovat vhodné algoritmy pro prohledávání grafů a hledání nejkratších cest v grafech do *Algoritmové knihovny ALib*. Spolu s tím bylo nutné upravit reprezentaci grafových datových struktur v knihovně, a to se zaměřením na univerzalitu návrhu, snadné využití a možné budoucí rozšíření. Posledním cílem bylo stanovení kritérií pro porovnání navržených algoritmů a provedení experimentálního měření.

Všechny tyto cíle byly úspěšně splněny. V první části této práce lze nalézt detailní popis a rozbor vlastností triviálních i složitějších algoritmů pro prohledávání grafů a hledání nejkratších cest v nich. Následuje kapitola, která vysvětluje způsob finální implementace. V samotné implementaci se povedlo přepracováním reprezentace grafových datových struktur docílit zvýšení univerzálnosti, což byl jeden z hlavních cílů této bakalářské práce. Zároveň je tato část knihovny připravena na budoucí rozšíření v podobě nových grafových algoritmů. V poslední části práce jsou stanovena kritéria pro porovnání těchto algoritmů a jsou zde detailně okomentovány výsledky měření.

Za velký úspěch považuji zachování funkčnosti dříve naimplementovaných algoritmů pro hledání maximálního toku a minimálního řezu v síti. Zároveň se knihovna dočkala výrazného rozšíření o nové algoritmy pro hledání cest v grafech. Mnohé z těchto algoritmů byly objeveny teprve nedávno

(JPS 2011 a MM 2016), a tak je knihovna ALib jednou z prvních, která obsahuje jejich implementaci.

V minulosti již byly zpracovány některé práce zaměřující se na implementaci grafických rozhraní pro jednotlivé části knihovny. V budoucnu by bylo jistě zajímavé vytvoření společné celoknihovní webové aplikace, která by umožňovala zobrazovat, vytvářet a modifikovat datové struktury knihovny a následně na nich spouštět knihovní algoritmy. Toto rozhraní by jistě vřele přivítali jak studenti tak pedagogové.

# Zdroje


1. BROŽ, Jan. *Automatová knihovna - Grafy a grafové algoritmy*. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016. Dostupné také z: https://dspace.cvut.cz/handle/10467/66211. Bakalářská práce. Vedoucí práce Radomír POLÁCH.
2. ROSCA, David. *Automatová knihovna - isomorfismus planárních grafů*. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015. Dostupné také z: https://dspace.cvut.cz/handle/10467/63107. Bakalářská práce. Vedoucí práce Radomír POLÁCH.
3. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů*. 1. vyd. Praha: CZ.NIC, z. s. p. o., 2017. ISBN 978-80-88168-22-5.
4. KOLÁŘ, Josef. *Teoretická informatika*. 2. vyd. Praha: České vysoké učení technické, 2009. ISBN 978-80-01-04331-8.
5. NEŠETŘIL, J.; MATOUŠEK, J. *Kapitoly z diskrétní matematiky*. 1. vyd. Praha: Karolinum, 2007. ISBN 978-80-246-1411-3.
6. RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3. vyd. Praha: Prentice Hall, 2009. ISBN 978-0136042594.
7. GRÜNBAUM, Branko; SHEPHARD, G. C. Tilings with congruent tiles. *Bull. Amer. Math. Soc. (N.S.)* [online]. 1980, roč. 3, č. 3, s. 951–973 [cit. 2018-03-02]. Dostupné z: https://projecteuclid.org:443/euclid.bams/1183547682.

8. CHEN, Li. *Digital and Discrete Geometry: Theory and Algorithms*. Springer Publishing Company, Incorporated, 2014. ISBN 978-3-319-12099-7.

9. ANUJ, Chauhan. *Abstract Data Types* [online] [cit. 2018-03-26]. Dostupné z: https://www.geeksforgeeks.org/abstract-data-types/.

10. MARTIN, Chris. *15 puzzle* [online]. 2016 [cit. 2018-03-02]. Dostupné z: https://commons.wikimedia.org/wiki/File:15-puzzle.svg.

11. MOORE, E.F. The shortest path through a maze. *Proceedings of the International Symposium on the Theory of Switching*. 1959, s. 285–292.

12. LEE, C. Y. An Algorithm for Path Connections and Its Applications. *IRE Transactions on Electronic Computers* [online]. 1961, roč. EC-10, č. 3, s. 346–365 [cit. 2018-03-10]. ISSN 0367-9950. Dostupné z DOI: 10.1109/TEC.1961.5219222.

13. POHL, I. *Bi-directional Search* [online]. IBM T.J. Watson Research Center, 1970 [cit. 2018-03-26]. Research report. Dostupné z: https://books.google.cz/books?id%20=%20wTinGwAACAAJ.

14. KORF, Richard E. Depth-first Iterative-deepening: An Optimal Admissible Tree Search. *Artif. Intell.* [online]. 1985, roč. 27, č. 1, s. 97–109 [cit. 2018-03-10]. ISSN 0004-3702. Dostupné z DOI: 10.1016/0004-3702(85)90084-0.

15. SHIMBEL, Alfonso. Structure in communication nets. *Proceedings of the Symposium on Information Networks* [online]. 1955, s. 199–203 [cit. 2018-03-26]. Dostupné z: https://books.google.cz/books?id=0UozngAACAAJ.

16. BELLMAN, R. On a Routing Problem. *Quarterly of Applied Mathematics*. 1958, roč. 16, s. 87–90.

17. YEN, Jin Y. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics* [online]. 1970, roč. 27, č. 4, s. 526–530 [cit. 2018-03-26]. ISSN 0033569X, 15524485. ISSN 0033569X, 15524485. Dostupné z: http://www.jstor.org/stable/43636060.

18. DUAN, Fanding. A faster algorithm for Shortest Path-SPFA. *Journal of Southwest Jiao Tong University* [online]. 1994, s. 207–212 [cit. 2018-03-26]. Dostupné z: https://wenku.baidu.com/view/3b8c5d778e9951e79a892705.html.

19. DIJKSTRA, E. W. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* [online]. 1959, roč. 1, č. 1, s. 269–271 [cit. 2018-03-10]. ISSN 0029-599X. Dostupné z DOI: 10.1007/BF01386390.

20. KULIKOV, Alexander S.; LEVIN, Michael; KANE, Daniel M; RHODES, Neil. *Finding Shortest Path after Meeting in the Middle* [online]. 2018 [cit. 2018-03-10]. Dostupné z: https://www.coursera.org/learn/algorithms-on-graphs/lecture/l8qtA/finding-shortest-path-after-meeting-in-the-middle.

21. HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* [online]. 1968, roč. 4, č. 2, s. 100–107 [cit. 2018-03-10]. ISSN 0536-1567. Dostupné z DOI: 10.1109/TSSC.1968.300136.

22. POHL, Ira Sheldon. *Bi-directional and Heuristic Search in Path Problems*. Stanford, CA, USA: Stanford University, 1969. Dizertační práce.

23. BARKER, Joseph K; KORF, Richard E. Limitations of Front-to-end Bidirectional Heuristic Search. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* [online]. Austin, Texas: AAAI Press, 2015, s. 1086–1092 [cit. 2018-03-26]. AAAI'15. ISBN 0-262-51129-0. Dostupné z: http://dl.acm.org/citation.cfm?id=2887007.2887158.

24. HOLTE, Robert C.; FELNER, Ariel; SHARON, Guni; STURTEVANT, Nathan R. Bidirectional Search That is Guaranteed to Meet in the Middle. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* [online]. Phoenix, Arizona: AAAI Press, 2016, s. 3411–3417 [cit. 2018-03-26]. AAAI'16. Dostupné z: http://dl.acm.org/citation.cfm?id=3016100.3016382.

25. HARABOR, Daniel; GRASTIEN, Alban. Online Graph Pruning for Pathfinding on Grid Maps. In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence* [online]. San Francisco, California: AAAI Press, 2011, s. 1114–1119 [cit. 2018-03-26]. AAAI'11. Dostupné z: http://dl.acm.org/citation.cfm?id=2900423.2900600.

26. FLOYD, Robert W. Algorithm 97: Shortest Path. *Commun. ACM* [online]. 1962, roč. 5, č. 6, s. 345– [cit. 2018-03-26]. ISSN 0001-0782. Dostupné z DOI: 10.1145/367766.368168.

27. JEREMY, Siek; LIE-QUAN, Lee; ANDREW, Lumsdaine. The Boost Graph Library [online]. 2018 [cit. 2018-03-26]. Dostupné z: http://www.boost.org/doc/libs/1_66_0/libs/graph/doc/index.html.

28. BLUSS. The Boost Graph Library [online]. 2018 [cit. 2018-03-26]. Dostupné z: https://github.com/bluss/petgraph.

29. BARAK, Naveh. JGraphT [online]. 2018 [cit. 2018-03-26]. Dostupné z: http://http://jgrapht.org/.

30. STURTEVANT, N. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games* [online]. 2012, roč. 4, č. 2, s. 144–148 [cit. 2018-03-26]. Dostupné z: http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf.

Dodatek A

# Seznam použitých zkratek

| | |
|---|---|
| ADT | Abstraktní datový typ |
| AG1 | Algoritmy a grafy 1 |
| AG2 | Algoritmy a grafy 2 |
| ALib | Algorithm library |
| BFS | Breadth-first search |
| BGL | The Boost Graph Library |
| CTU | Czech Technical University in Prague |
| ČVUT | České vysoké učení technické v Praze |
| DFS | Depth-first search |
| FIFO | First in, First out |
| FIT | Fakulta informačních technologií |
| GPS | Global Positioning System |
| HS | Heuristic search |
| IDDFS | Iterative deeping depth-first search |
| IDA* | Iterative deeping A* |
| JPS | Jump point search |
| JIT | Just-in-time |
| LIFO | Last in, First out |
| MM | Meet in the middle |
| SPFA | Shortest Path Faster Algorithm |
| STL | Standard Template Library |
| ZDM | Základy diskrétní matematiky |

| | |
|---|---|
| ZUM | Základy umělé inteligence |

Dodatek B

# Ukázky kódů

```
#include <grid/GridClasses.hpp>

grid::WeightedSquareGrid8<> graph(11, 11);

graph.addObstacle(2, 5);
graph.addObstacle(3, 5);
graph.addObstacle(4, 5);
graph.addObstacle(5, 5);
graph.addObstacle(6, 5);
graph.addObstacle(7, 5);
graph.addObstacle(5, 3);
graph.addObstacle(5, 4);
graph.addObstacle(5, 6);
graph.addObstacle(5, 7);
graph.addObstacle(5, 8);

// Get default node_type
using node_type = decltype(graph)::node_type;
using weight_type = decltype(graph)::edge_type::weight_type;

// Choose start and goal node
node_type start = ext::make_pair(8l, 2l);
node_type goal = ext::make_pair(1l, 9l);
```

**Ukázka kódu B.1:** Deklarace grafu v knihovně ALib

```
#include <graph/traverse/BFS.hpp>

// Simply run BFS algorithm
graph::traverse::BFS::run(graph, start);

// Find path with BFS algorithm
ext::vector<node_type> path;
path = graph::traverse::BFS::findPath(graph, start, goal);

// Use bidirectional BFS
path = graph::traverse::BFS::
       findPathBidirectional(graph, start, goal);
```

**Ukázka kódu B.2:** Spuštění BFS algoritmu

```
#include <graph/traverse/DFS.hpp>

// Simply run DFS algorithm
graph::traverse::DFS::run(graph, start);

// Find path with DFS algorithm
ext::vector<node_type> path;
path = graph::traverse::DFS::findPath(graph, start, goal);
```

**Ukázka kódu B.3:** Spuštění DFS algoritmu

```cpp
#include <graph/traverse/IDDFS.hpp>

// Simply run IDDFS algorithm
graph::traverse::IDDFS::run(graph, start, max_depth);

// Find path with IDDFS algorithm
ext::vector<node_type> path;
path = graph::traverse::IDDFS::findPath(graph, start, goal);

// Use bidirectional IDDFS
path = graph::traverse::IDDFS::
        findPathBidirectional(graph, start, goal);
```

**Ukázka kódu B.4:** Spuštění IDDFS algoritmu

```cpp
#include <graph/shortest_path/BellmanFord.hpp>
#include <graph/shortest_path/SPFA.hpp>

// Find shortest path with BellmanFord algorithm
ext::pair<ext::vector<node_type>, weight_type> path;
path = graph::shortest_path::BellmanFord::
        findPath(graph, start, goal);

// Use SPFA optimization
path = graph::shortest_path::SPFA::findPath(graph, start, goal);
```

**Ukázka kódu B.5:** Spuštění Bellman-Fordova algoritmu

```
#include <graph/shortest_path/Dijkstra.hpp>

// Simply run Dijkstra algorithm
graph::shortest_path::Dijkstra::run(graph, start);

// Find shortest path with Dijkstra algorithm
ext::pair<ext::vector<node_type>, weight_type> path;
path = graph::shortest_path::Dijkstra::
        findPath(graph, start, goal);

// Use bidirectional Dijkstra
path = graph::shortest_path::Dijkstra::
        findPathBidirectional(graph, start, goal);
```

**Ukázka kódu B.6:** Spuštění Dijkstrova algoritmu

```
#include <graph/shortest_path/GreedyBestFS.hpp>

// Find path with Dijkstra algorithm
ext::pair<ext::vector<node_type>, weight_type> path;
path = graph::shortest_path::GreedyBestFS::
        findPath(graph, start, goal, f_heuristic);
```

**Ukázka kódu B.7:** Spuštění Hladového uspořádaného prohledávání

```cpp
#include <graph/shortest_path/AStar.hpp>

auto f_heuristic_forward = [&](const TNode &n) -> weight_type {
    return graph::common::GridHeuristicFunctions::
        diagonalDistance(goal, n);
};

auto f_heuristic_backward = [&](const TNode &n) -> weight_type {
    return graph::common::GridHeuristicFunctions::
        diagonalDistance(start, n);
};

// Find shortest path with AStar algorithm
ext::pair<ext::vector<node_type>, weight_type> path;
path = graph::shortest_path::AStar::
        findPath(graph, start, goal, f_heuristic_forward);

// Use bidirectional AStar with Pohl condition
path = graph::shortest_path::AStar::
        findPathBidirectional(graph, start, goal,
                              f_heuristic_forward,
                              f_heuristic_backward);
```

**Ukázka kódu B.8:** Spuštění A* algoritmu

```
#include <graph/shortest_path/MM.hpp>

auto f_heuristic_forward = [&](const TNode &n) -> weight_type {
    return graph::common::GridHeuristicFunctions::
        diagonalDistance(goal, n);
};

auto f_heuristic_backward = [&](const TNode &n) -> weight_type {
    return graph::common::GridHeuristicFunctions::
        diagonalDistance(start, n);
};

// Find shortest path with MM algorithm
ext::pair<ext::vector<node_type>, weight_type> path;
path = graph::shortest_path::MM::
        findPathBidirectional(graph, start, goal,
                              f_heuristic_forward,
                              f_heuristic_backward);
```

**Ukázka kódu B.9:** Spuštění MM algoritmu

```
#include <graph/shortest_path/IDAStar.hpp>

auto f_heuristic_forward = [&](const TNode &n) -> weight_type {
    return graph::common::GridHeuristicFunctions::
        diagonalDistance(goal, n);
};

// Find shortest path with IDAStar algorithm
ext::pair<ext::vector<node_type>, weight_type> path;
path = graph::shortest_path::IDAStar::
        findPath(graph, start, goal, f_heuristic_forward);
```

**Ukázka kódu B.10:** Spuštění IDA* algoritmu

```
#include <graph/shortest_path/JPS.hpp>

auto f_heuristic_forward = [&](const TNode &n) -> weight_type {
    return graph::common::GridHeuristicFunctions::
        diagonalDistance(goal, n);
};

// Find shortest path with JPS algorithm
ext::pair<ext::vector<node_type>, weight_type> path;
path = graph::shortest_path::JPS::
        findPath(graph, start, goal, f_heuristic_forward);
```

**Ukázka kódu B.11:** Spuštění JPS algoritmu

```
#include <graph/shortest_path/FloydWarshall.hpp>

// Find distance matrix
ext::map<node_type, ext::map<node_type, weight_type> matrix;
matrix = graph::shortest_path::FloydWarshall::run(graph);
```

**Ukázka kódu B.12:** Spuštění Floyd-Warshallova algoritmu

```
// Measure expanded nodes
unsigned long expanded_nodes = 0;

f_user = [&](const auto &, const auto &) {
    ++expanded_nodes;
};

/* Run algorithm */
```

```
#include<chrono>
#include<vector>

// Measure time
vector<long> times;
for (int i = 0; i < 10; ++i) {
    std::chrono::steady_clock::
        time_point start = std::chrono::steady_clock::now();

    /* Run algorithm */

    std::chrono::steady_clock::
        time_point end = std::chrono::steady_clock::now();
    times.push_back(
        std::chrono::duration_cast
            <std::chrono::milliseconds>(end - start).count()
                );
}

auto average = std::
    accumulate(times.begin(), times.end(), 0l) / times.size();
```

```
#!/bin/bash
# Measure space
valgrind --tool=massif --massif-out-file="log.massif" a.out
```

**Ukázka kódu B.13:** Minimální konfigurace pro měření algoritmů

Dodatek C

# Obsah přiloženého média

- `README.md` ..... stručný popis obsahu média ve formátu Markdown
- `src`
  - `bachelor-thesis` ..... zdrojová forma práce ve formátu XeLaTeX
- `BP_Uhlik_Jan_2018.pdf` ..... text práce ve formátu PDF

**Adresářová struktura C.1:** Obsah přiloženého média