

# Využití evolučního algoritmu pro řešení bludišť

Bc. Jiří Večerka

---

Diplomová práce  
2015



Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky

---

## **ZADÁNÍ DIPLOMOVÉ PRÁCE**

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Jiří Večerka**  
Osobní číslo: **A12739**  
Studijní program: **N3902 Inženýrská informatika**  
Studijní obor: **Informační technologie**  
Forma studia: **kombinovaná**

Téma práce: **Využití evolučního algoritmu pro řešení bludišť**

Téma anglicky: **Utilization of Evolutionary Algorithm for Solving Maze Problems**

Zásady pro vypracování:

1. Definujte a klasifikujte bludiště a labyrinty.
2. Uvedte a prezentujte typické příklady labyrintů a bludišť.
3. Vypracujte přehled vhodných algoritmů pro vytváření různých typů bludišť.
4. Zvolte vhodný evoluční algoritmus pro řešení základních úloh v bludištích.
5. Vytvořte počítačovou aplikaci pro generování bludišť a řešení úloh v bludištích s animací.

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. DOOB, Penelope Reed. The Idea of the Labyrinth: From Classical Antiquity through the Middle Ages. New York: Cornell University Press, 1992. ISBN 0-80-148000-0.
2. FISHER, Adrian. The Amazing Book of Mazes. New York: Harry N. Abrams, 2006. ISBN 978-08-109-4311-7.
3. ŠPAŇHELOVÁ, Kateřina. Labyrint. Praha, 2006. Diplomová práce. Univerzita Karlova, Pedagogická fakulta, Katedra výtvarné výchovy.
4. ZELINKA, I., Z. OPLATKOVÁ, M. ŠEDA, P. OŠMERA a F. VČELAŘ. Evoluční výpočetní techniky, principy a aplikace. Praha: BEN, 2008. ISBN 80-7300-218-3.
5. DORIGO, Marco a Thomas STÜTZLE. Ant Colony Optimization. Cambridge: The MIT Press, 2004. ISBN 978-02-620-4219-2.
6. DORIGO, Marco. Ant colony optimization and swarm intelligence: 4th international workshop, ANTS 2004, Brussels, Belgium, September 5 - 8, 2004 : proceedings. Berlin: Springer, 2004, xii, 434 s. ISBN 3540226729.
7. HEROUT, Pavel. Učebnice jazyka Java. České Budějovice: KOPP, 2000. ISBN 80-7232-115-3.
8. HEROUT, Pavel. Java - grafické uživatelské prostředí a čeština. České Budějovice: KOPP, 2001. ISBN 80-7232-150-1.

Vedoucí diplomové práce:

**doc. Ing. Zuzana Komínková Oplatková, Ph.D.**  
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce:

**6. února 2015**

Termín odevzdání diplomové práce:

**15. května 2015**

Ve Zlíně dne 6. února 2015



doc. Mgr. Milan Adámek, Ph.D.  
*děkan*



doc. Mgr. Roman Jašek, Ph.D.  
*ředitel ústavu*

### **Prohlašuji, že**

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

### **Prohlašuji,**

- že jsem na diplomové/bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....  
podpis diplomanta

## **ABSTRAKT**

Tato diplomová práce přináší stručný přehled problematiky labyrintů a bludišť. V teoretické části jsou nadefinovány základní vlastnosti bludišť a jejich rozdělení. Je zde vysvětlena spojitost mezi vytvářením a řešením bludišť a matematickou teorií grafů. Dále práce obsahuje nezbytný teoretický základ k vybranému evolučnímu algoritmu řešící bludiště, kterým je optimalizace mravenčí kolonií. Praktická část práce potom podrobně popisuje uvedené algoritmy pro vytváření i řešení bludišť, na základě nichž pak byla implementována aplikace názorně ilustrující proces vytváření i řešení bludišť.

Klíčová slova: bludiště, teorie grafů, algoritmy, optimalizace mravenčí kolonií, S-ACO, java, applet

## **ABSTRACT**

This thesis presents a brief overview of labyrinths and mazes. The theoretical part defines us a basic categorization and classification of mazes. Then the connection between creating and solving mazes and the graph theory is explained. The thesis also contains the theoretical background needed to understand the ant colony optimization, which is the selected evolutionary algorithm. The practical part of the thesis then describes the algorithms themselves, which was the basis for implementing application for illustrating processes of generating and solving mazes.

Keywords: Maze, Graph Theory, Algorithms, Ant Colony Optimization, S-ACO, Java, Applet

Mé poděkování patří doc. Ing. Zuzaně Komínkové Oplatkové, Ph.D. za odborné vedení, trpělivost a ochotu, kterou mi v průběhu zpracování diplomové práce věnovala.

# OBSAH

<b>ÚVOD.....</b>	<b>8</b>
<b>I TEORETICKÁ ČÁST.....</b>	<b>10</b>
<b>1 DEFINICE BLUDIŠTĚ .....</b>	<b>11</b>
1.1 KLASIFIKACE DLE VZHLEDU BLUDIŠTĚ.....	12
1.1.1 Rozdělení dle topologie.....	12
1.1.2 Rozdělení dle dimenze .....	12
1.1.3 Rozdělení dle typu mozaiky.....	13
1.1.4 Rozdělení dle směřování .....	13
1.2 KLASIFIKACE DLE VNITŘNÍ STRUKTURY BLUDIŠTĚ .....	14
1.2.1 Vlastnosti vyplývající ze základních prvků bludiště.....	14
1.2.2 Vlastnosti vyplývající ze systému bludiště .....	15
1.3 REÁLNÉ LABYRINTY A BLUDIŠTĚ.....	15
<b>2 TEORIE GRAFŮ .....</b>	<b>17</b>
2.1 GRAFY A STROMY .....	17
2.1.1 Základní typy grafů .....	17
2.1.2 Stromy.....	19
2.1.3 Nejkratší cesta v grafu.....	19
2.2 REPREZENTACE BLUDIŠTĚ GRAFEM .....	20
2.2.1 Spojová reprezentace .....	21
2.2.2 Matice sousednosti .....	22
<b>3 ALGORITMY GENERUJÍCÍ BLUDIŠTĚ .....</b>	<b>23</b>
3.1 ALGORITMY VYCHÁZEJÍCÍ Z TEORIE GRAFŮ .....	23
3.1.1 Algoritmus prohledávání do hloubky.....	24
3.1.2 Primův algoritmus .....	24
3.1.3 Kruskalův algoritmus .....	25
3.2 ELLERŮV ALGORITMUS .....	25
3.3 ALGORITMUS PŮLENÍ INTERVALŮ.....	26
<b>4 ALGORITMY ŘEŠÍCÍ BLUDIŠTĚ .....</b>	<b>27</b>
4.1 ALGORITMUS SLEDOVÁNÍ ZDÍ.....	27
4.2 ALGORITMUS VYPLŇOVÁNÍ SLEPÝCH KONCŮ .....	27
4.3 TRÉMAUXŮV ALGORITMUS.....	28
4.4 ALGORITMUS HLEDÁNÍ NEJKRATŠÍCH CEST .....	28
4.5 DIJKSTRŮV ALGORITMUS .....	29
4.6 ALGORITMUS NÁHODNÁ MYŠ .....	30
<b>5 OPTIMALIZACE MRAVENČÍ KOLONIÍ.....</b>	<b>31</b>
5.1 BIOLOGICKÁ INSPIRACE .....	31
5.2 S-ACO ALGORITMUS .....	33
5.3 ACO META-HEURISTIKA .....	35
5.3.1 Fáze vytváření řešení.....	37
5.3.2 Fáze aktualizace feromonu.....	38
5.3.3 Vnější zásahy .....	38

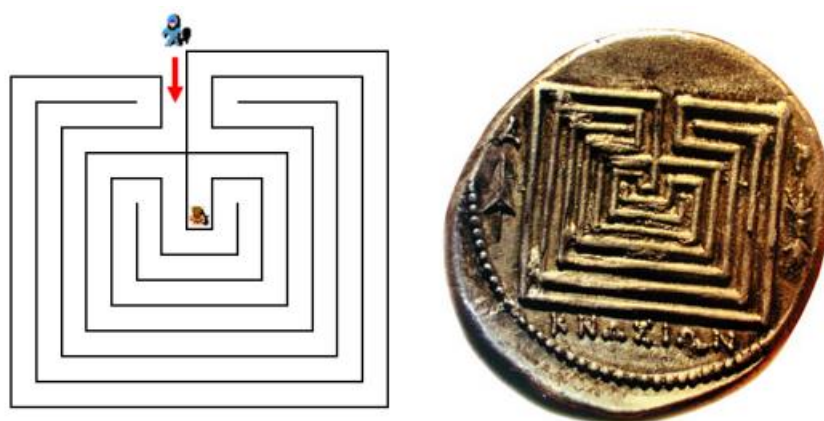
5.4	APLIKACE ACO .....	38
5.4.1	Problém obchodního cestujícího .....	39
5.4.2	Problém směřování komunikačních sítí .....	39
<b>II</b>	<b>PRAKTICKÁ ČÁST .....</b>	<b>40</b>
<b>6</b>	<b>IMPLEMENTACE ALGORITMŮ .....</b>	<b>41</b>
6.1	POUŽITÉ TECHNOLOGIE .....	41
6.1.1	Java.....	41
6.1.2	Java applet.....	42
<b>7</b>	<b>IMPLEMENTACE ALGORITMŮ .....</b>	<b>43</b>
7.1	DATOVÝ MODEL REPREZENTUJÍCÍ BLUDIŠTĚ .....	43
7.2	ALGORITMY GENERUJÍCÍ BLUDIŠTĚ .....	44
7.2.1	Algoritmus prohledávání do hloubky.....	44
7.2.2	Primův algoritmus.....	45
7.2.3	Kruskalův algoritmus .....	46
7.2.4	Ellerův algoritmus .....	48
7.2.5	Algoritmus půlení intervalů .....	49
7.3	ALGORITMY ŘEŠÍCÍ BLUDIŠTĚ .....	51
7.3.1	Algoritmus sledování zdí .....	51
7.3.2	Algoritmus vyplňování slepých konců.....	52
7.3.3	Trémauxův algoritmus .....	53
7.3.4	Algoritmus hledání nejkratších cest .....	54
7.3.5	Dijkstrův algoritmus.....	55
7.4	IMPLEMENTACE ACO.....	56
7.4.1	Problém hledání nejkratší cesty .....	56
7.4.2	Pseudokód S-ACO .....	57
7.4.3	Třídy algoritmu .....	57
7.4.4	Umělý mravenec .....	58
7.4.5	Inicializace algoritmu.....	58
7.4.6	Metoda nalezení cesty .....	60
7.4.7	Výběr následujícího uzlu.....	60
7.4.8	Aktualizace feromonů .....	61
7.4.9	Vypařování feromonů .....	62
7.4.10	Ukončení algoritmu.....	62
<b>8</b>	<b>POROVNÁNÍ ALGORITMŮ .....</b>	<b>63</b>
	<b>ZÁVĚR .....</b>	<b>66</b>
	<b>SEZNAM POUŽITÉ LITERATURY.....</b>	<b>67</b>
	<b>SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....</b>	<b>68</b>
	<b>SEZNAM OBRÁZKŮ .....</b>	<b>69</b>
	<b>SEZNAM TABULEK.....</b>	<b>70</b>
	<b>SEZNAM PŘÍLOH.....</b>	<b>71</b>



## ÚVOD

Pojem bludiště se často zaměňuje s termínem labyrint. Rozdíl mezi těmito pojmy pochopitelně existuje, jednoduše vychází ze struktury problému. Bludištěm je myšlena skládanka v podobě větvených cest s možností volby cesty a směru, oproti tomu labyrint obsahuje do svého středu pouze jedinou cestu a je navržen tak, aby bylo jednoduché se v něm orientovat.

První dochovaná zmínka o labyrintu (řecky λαβύρινθος labyrinthos), se objevuje v řecké mytologii. Dle pověsti největší řecký stavitel Daidalos navrhl a postavil pro krétského krále Mínoa na ostrově Knóssos na zakázku složitý a rozlehlý labyrint. V něm král ukryl nevlastního syna Mínotaura, krvelačnou obludu s tělem muže a hlavou býka. Mínotaura nakonec zabil statečný aténský hrdina Théseus. Daidalos postavil Labyrint tak šikovně a chytře, že on sám mohl jen stěží uniknout poté, co stavbu dokončil. Théseovi však pomohla Ariadné (dcera krále Mínoa) a její osudové klubko nití, které mu pomohlo najít cestu zpět [1].



*Obr. 1 – Labyrint na ostrově Knóssos*

Původní účel bludiště, kterým byla ochrana, byl postupem času překonán. V současnosti se bludiště v různých podobách objevuje v mnoha odvětvích lidské činnosti. Vědecká sféra například bludiště využívá k testování inteligence a orientačních schopností různorodých živočichů. Ve velké míře se bludišť chopil zábavní průmysl. Bludiště se v něm objevují jako reálné stavby v zábavních parcích, jako zábavné hříčky v tiskovinách nebo jako více či méně složité problémy v počítačových hrách. V neposlední řadě je třeba zmínit realizace bludišť v architektuře, zejména zahradní. Zahradní bludiště jsou po světě realizována dodnes [2].

Bludiště lze zkonstruovat několika různými způsoby. Mohou být navržena či nakreslena ručně nebo s využitím algoritmizace principů použitých pro jejich vytváření za pomoci výpočetní techniky. V rámci této práce budou definovány základní vlastnosti bludišť, bude položen matematický základ teorie grafů a bude představeno několik algoritmů, vycházející

z teorie grafů a které se pro generování bludišť používají. Dále budou také uvedeny některé algoritmy, které naopak v již existujícím bludišti řešení hledají.

Mimo klasické výpočetní algoritmy hledající řešení bludiště budou v této práci uvedeny základy metody optimalizace mravenčí kolonií (Ant Colony Optimization, zkráceně ACO) a následně popsány mechanismy implementace základního S-ACO algoritmu.

Hlavním cílem této práce tedy je poskytnout ucelený přehled teoretických informací o bludištích a labyrintech s ohledem na teorii grafů, dále popsat vybrané klasické algoritmy tvořící i řešící bludiště a nastínit řešení základních úloh v bludištích pomocí vybraného evolučního algoritmu.

## **I. TEORETICKÁ ČÁST**

## 1 DEFINICE BLUDIŠTĚ

Jak už bylo zmíněno v úvodu této práce, bludiště je spleť cest, většinou s jednou či více slepými větvemi. Je to rébus, který je řešitelný a má cíl, který je dosažitelný.

Nejmenší element bludiště budeme nazývat buňka. Jak je buňka v bludišti reprezentována, respektive jaký má tvar, záleží na typu mozaiky bludiště. Další základní částí, tvořící bludiště jako celek složený z buněk, je potom zeď. Tu definujeme jako neprůstupné propojení (hrana) dvou sousedících buněk bludiště.

Buňka v rámci bludiště může existovat ve čtyřech podobách a to jako křižovatka, slepý konec, buňka jako součást cesty a izolovaná buňka. Toto rozdělení je provedeno na základě počtu propojení sousedících buněk a jejich existence v bludišti je daná typem směřování, v bludišti se tedy nemusí vyskytovat [3].

*Tab. 1 – Typy buněk vyskytujících se v bludišti*

Typ buňky	Počet propojení se sousedními buňkami
Křižovatka	3 a více dle typu mozaiky
Slepý konec	1
Buňka jako součást cesty	2
Izolovaná buňka	0

Pokud v bludišti existuje souvislá posloupnost mezi sebou propojených buněk mezi dvěma buňkami typu křižovatka nebo slepý konec, budeme této posloupnosti říkat cesta. Pro zjednodušení se v rámci této práce bude cestou délky 1 považovat souvislé propojení buněk mezi dvěma křižovatkami. Je to s ohledem na reprezentaci bludiště teorií grafů, která bude prezentována dále.

Trasou budeme potom nazývat posloupnost propojených buněk mezi dvěma libovolnými buňkami v bludišti. Tras může v bludišti pochopitelně existovat více. Je zřejmé, že toto tvrzení platí jen pro bludiště. Labyrint ze své definice může mít jen jednu trasu.

Bludiště obecně dělíme do několika různých kategorií, přičemž klasifikujeme bludiště do dvou hlavních kategorií: podle vzhledu a podle vnitřní struktury bludiště. V následujících podkapitolách se seznámíme s jednotlivými kategoriemi a typy bludišť.

## 1.1 Klasifikace dle vzhledu bludiště

Klasifikace dle vzhledu bludiště nám určují podobu bludiště jako takového. Říkají nám, jak bude vypadat jednotlivá buňka tvořící bludiště, nad jakým prostorem bude bludiště definováno nebo jak je uspořádaná vnitřní struktura bludiště.

### 1.1.1 Rozdělení dle topologie

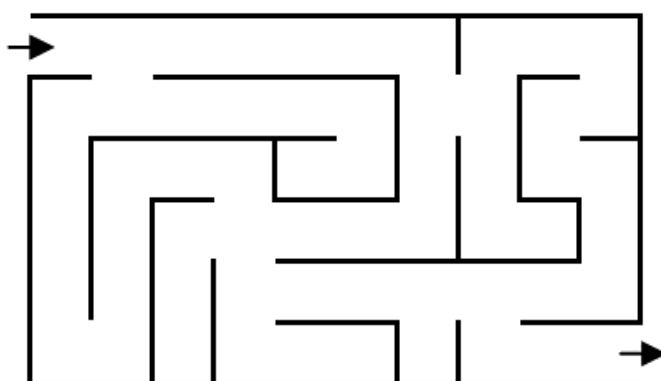
Rozdělení bludišť dle topologie popisuje geometrii prostoru, ve kterém je bludiště definováno [3].

- **Bludiště s normální topologií** je definováno v Euklidovském prostoru, tedy prostor splňující Eukleidovy axiomy.
- **Bludiště s planární topologií** naopak není definováno v Euklidovském prostoru, tedy nemá normální topologii. Příkladem je například bludiště na povrchu koule.

### 1.1.2 Rozdělení dle dimenze

Rozdělení bludišť podle dimenze, je jedno ze základních rozdělení. Vymezuje nám prostor, ve kterém bude bludiště existovat [7].

- **Dvou dimenzionální bludiště (2D)** je základní kategorie, do níž patří většina bludišť, které známe, ať už na papíře nebo v reálném světě. Jejich vizualizace lze provést prostým půdorysným nákresem.



Obr. 2 – Ukázka dvou dimenzionálního bludiště

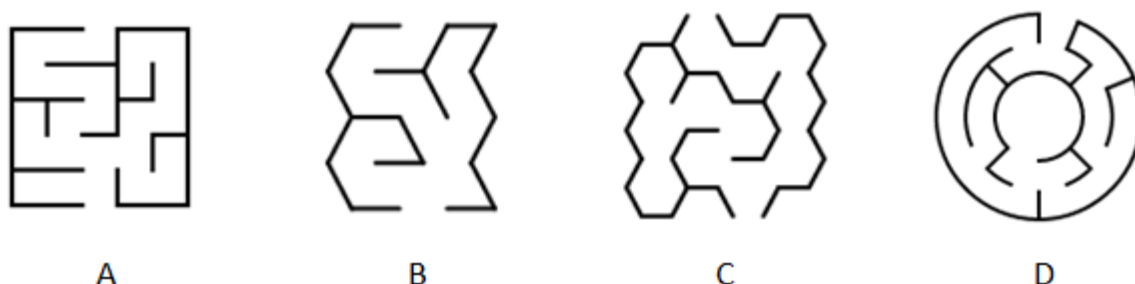
- **Tří dimenzionální bludiště (3D)** lze jednoduše popsat a vizualizovat jako více dvou dimenzionálních bludišť umístěných nad sebou, vzájemně propojených i ve zbývajících dvou směrech směrové růžice.

- **Více dimenzionální bludiště (4D a více)** jsou poměrně těžce představitelná. Často se vykreslují jako 3D bludiště navíc se speciálními "portály" přenášejícími objekt v rámci bludiště i v dalším rozměru, např. v čase.
- **Weave bludiště (2,5D)** je speciální případ dvou rozměrného bludiště, které ale navíc obsahuje přemostění některých pasáží.

### 1.1.3 Rozdělení dle typu mozaiky

Rozdělení bludišť dle typu mozaiky nám určuje jaký je vzhled jednotlivých buněk, které tvoří mřížku bludiště. Bludiště dle typu mozaiky dělíme na **ortogonální bludiště**, dále na skupinu **omega bludišť** mající jinou než ortogonální mozaiku a **speciální typy bludišť** jako je například bludiště fraktální. Výčet typů je uveden následovně [7]:

- **Ortogonální bludiště** jsou tvořena čtvercovými buňkami a kde i cesty bludiště se kříží v pravých úhlech. Jde o typ bludiště s nejčastější reprezentací.
- **Delta a Sigma bludiště** jsou bludiště tvořeny buňkami ve tvaru trojúhelníku, respektive šestiúhelníku.
- **Theta bludiště** je uspořádáno v soustředných kružnicích a počáteční bod cesty je na okraji bludiště a cílový bod je v jeho středu nebo naopak. Tento typ bludiště bývá velmi často realizován.
- **Upsilon bludiště** má buňky tvořeny osmiúhelníky propojenými čtverci a každá buňka může mít až osm propojených cest.
- **Fraktální bludiště** je speciální typ bludiště, kde každá buňka bludiště je opět bludištěm tvořícím rekurzivně sebe sama do nekonečna dle principů fraktální geometrie.

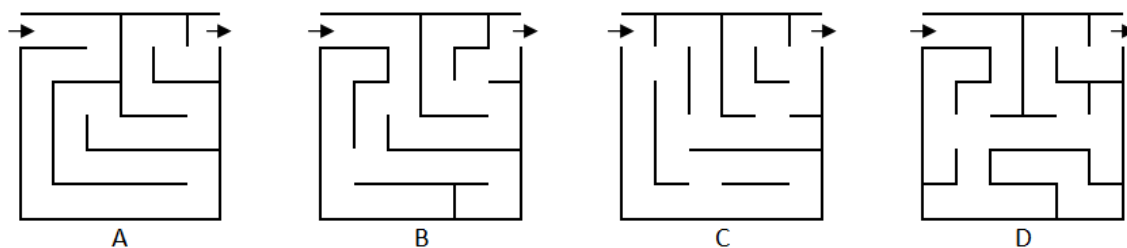


Obr. 3 – Typy mozaiky: Ortogonální (A), Delta (B), Sigma (C), Theta (D)

### 1.1.4 Rozdělení dle směrování

Rozdělení bludišť dle směrování vymezuje vnitřní strukturu bludiště. Řeší, jakým způsobem se bude objekt v rámci bludiště pohybovat a jak jsou jednotlivé cesty v bludišti propojeny.

- **Unicursal bludiště** nám definuje bludiště bez křížení cest nebo rozcestí. V podstatě jde o jednu dlouhou cestu, která prochází celým bludištěm od počátečního bodu do cílového, tedy o labyrint [3].
- **Perfektní bludiště** jsou definována tak, že mezi libovolnými dvěma body bludiště vede vždy jen jedna cesta. Tedy že neobsahuje žádné zacyklení nebo oddělené části. Může ale obsahovat rozcestí. Stejně jako předchozí typ bludiště má vždy jen jedno řešení [7].
- **Řídké bludiště** má povoleny oddělené nepřístupné části a tudíž neplatí podmínka existence propojení dvou libovolných bodů bludiště. Do určité části bludiště se tedy nelze dostat.
- **Spletená bludiště** mají umožněny zacyklení, ale nemají povoleny tzv. slepé konce.



Obr. 4 – Typy směrování: Unicursal (A), Perfektní (B), Spletené (C) a Řídké (D)

## 1.2 Klasifikace dle vnitřní struktury bludiště

Na rozdíl od předchozích rozdělení zabývajících se podobou bludiště, tato klasifikace nám umožní poznat vlastnosti vnitřní struktury bludiště a vztahy mezi jejich základními prvky.

### 1.2.1 Vlastnosti vyplývající ze základních prvků bludiště

Skupina vlastností vyplývajících ze základních prvků bludiště jsou převážně statistické hodnoty, které popisují konkrétní základní prvky bludiště.

První sledovanou veličinou je obvykle celkový počet buněk bludiště. Její hodnota popisuje **rozsáhlost bludiště**.

Dalšími sledovanými statistickými veličinami jsou počty výskytů slepých konců a křižovatek v bludišti. Obvykle je uváděno jejich procentuální vyjádření vzhledem k celkovému počtu buněk a hodnotí se jejich vzájemný vztah. Jejich hodnoty také souvisí s další sledovanou veličinou, kterou je **celkový počet cest**. V případě perfektních bludišť bude počet cest vždy o jednu menší, než je součet počtu křižovatek a počtu slepých konců.

Poslední sledovanou veličinou je **délka trasy**. Dle definice je trasou myšlena posloupnost propojených buněk mezi dvěma libovolnými body bludiště, jak ale bude uvedeno dále v kapitole 4, tak za řešení bludiště se považuje zejména nejkratší trasa mezi vstupním a výstupním bodem bludiště.

### 1.2.2 Vlastnosti vyplývající ze systému bludiště

Tyto vlastnosti popisují vztahy mezi cestami a ostatními prvky bludiště. Nejsou v rámci této práce řešeny a jsou zde uvedeny jen pro úplnost. Jejich obsáhlý popis lze nalézt v [7] a [3].

- **Zkreslení** (v originále bias) je jedna z vlastností, která charakterizuje rovné úseky cest vyskytující se v bludišti, včetně směru, kterým vedou. V případě ortogonálních bludišť se sleduje poměr cest v horizontálním a vertikálním směru.
- Další vlastnost popisující rovné úseky cest v bludišti je **běh** (v originále run). Kvantifikuje délku rovných úseků cest, kdy se vychází z velikosti bludiště.
- **Elitnost** (v originále elite) je vlastnost popisující vztah délky trasy řešení a celkové velikosti bludiště. Tuto vlastnost nelze nijak kvantifikovat, protože je závislá na umístění vstupního a výstupního bodu bludiště. Bludiště je elitní, pokud je délka trasy řešení bludiště malá a přímá.
- Další charakteristika se týká existence **symetrií** (v originále symmetry) v bludišti. Pokud je bludiště vytvářeno algoritmy, které využívají generátory náhodných čísel, symetrie se v něm nevyskytují vůbec.
- Poslední vlastností popisující systém bludiště je **tok** (v originále river). Tok popisuje vztah délky cest vzhledem k počtu slepých konců.

Mimo počtu cest v bludišti, existují další vlastnosti, které popisují komplexnost bludiště. Jsou to délka cest a jejich klikatost.

## 1.3 Reálné labyrinty a bludiště

Labyrinty a bludiště bývají realizovány jak symbolicky tak i fyzicky. Symbolicky jsou zobrazovány na obrazech nebo jako vzory na keramice, mincích, různých mozaikách či stěnách jeskyní [4].

Novodobé fyzické realizace se datují do dvanáctého století, kdy se bludiště začaly objevovat v zahradách v Anglii. Pro svou oblibu se postupně rozšířily po celé tehdejší civilizované Evropě, zejména v Itálii a Francii. Typická bludiště nalezneme například v zahradě Andre



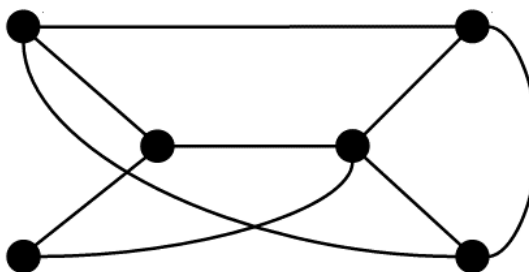
Le Notre ve Versailles a Caboni Villa Pisani v Itálii. V Čechách je nejznámějším bludištěm zrcadlový labyrint na Petříně v Praze.

## 2 TEORIE GRAFŮ

Jak již bylo uvedeno v první kapitole, bludiště se skládá z křižovatek, slepých konců a cest, které vše propojují. Proto je pro jeho reprezentaci výhodné využít teorie grafů a bludiště vyjádřit jako graf. V následujících podkapitolách bude uvedena teorie grafů a podán matematický základ, který bude použit v dalších kapitolách.

### 2.1 Grafy a stromy

V teorii grafů je graf chápán jako něco jiného než je graf funkce či výsečový graf. Graf  $G$  je definován jako uspořádaná dvojice dvou konečných množin  $V$  a  $E$ . Tedy  $G = (V, E)$ . Prvky množiny  $V$  se označují jako uzly a prvky množiny  $E$  jako hrany [5].



Obr. 5 – Ukázka grafu

Každou hranu  $e$  tvoří neuspořádaná dvojice vrcholů  $e = \{u, v\}$  pro  $u, v \in V$ . Vrcholy  $u$  a  $v$  jsou nazývány konce hrany  $e$ . Vrcholy, se kterými je vrchol  $u$  spojen hranou, se nazývají sousedi vrcholu  $u$ .

#### 2.1.1 Základní typy grafů

Pro práci s grafy a jednoduchý popis toho, co s grafy děláme, potřebujeme znát základní pojmy:

- **Izomorfismus:** Dva grafy  $G = (V, E)$  a  $G' = (V', E')$  nazýváme izomorfní, jestliže existuje bijektivní (vzájemně jednoznačné) zobrazení  $f : V \rightarrow V'$  takové, že platí:  $\{x, y\} \in E$ , právě když  $\{f(x), f(y)\} \in E'$ . Zjednodušeně, izomorfismus můžeme popsat jako přejmenování vrcholů [5].
- **Stupeň vrcholu:** Stupeň vrcholu  $u$  v grafu  $G$  určuje počet jeho sousedů.

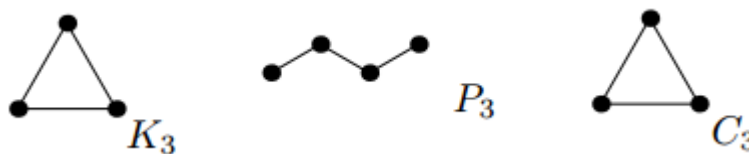
- **Nadgraf / podgraf grafu:**  $H$  je podgraf grafu  $G$  (značí se  $H \subseteq G$ ) právě tehdy když  $H$  je graf a  $V(H) \subseteq V(G)$ ,  $E(H) \subseteq E(G)$ . Podgraf  $H$  tedy vznikne z grafu  $G$  odstraněním některých hran a vrcholů včetně hran z nich vedoucích. Obráceně platí, že graf  $G$  je nadgraf grafu  $H$ , tedy graf  $G$  obsahuje podgraf  $H$  [5].
- **Maximální / minimální graf:** Graf  $G$  je maximální nebo minimální pro nějakou grafovou vlastnost  $A$ , pokud graf  $G$  splňuje vlastnost  $A$  a žádný nadgraf nebo podgraf ji už nesplňuje [5].



Obr. 6 – Izomorfismus:  $A \rightarrow 1$ ,  $H \rightarrow 2$ ,  $P \rightarrow 4$ ,  $V \rightarrow 3$

Pokud graf splňuje určitou vlastnost, má svůj název. K základním třídám grafů patří:

- **Úplný graf:** Úplný graf je graf bez smyček, tedy hran, jejichž oba koncové uzly jsou shodné, a ve kterém jsou každé dva uzly spojeny hranou. Značí se  $K$ .
- **Cesta:** Cestou je označován neprázdný graf  $P = (V, E)$ , kde  $V = \{x_0, x_1, \dots, x_k\}$  a  $E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k\}$ . Vrcholy  $x_0$  a  $x_k$  jsou koncové vrcholy cesty a ostatní vrcholy jsou vrcholy vnitřní. V cestě se žádný uzel nevyskytuje dvakrát. Délkou cesty je označován počet hran na cestě [5].
- **Cesta v grafu:** Cesta v grafu  $G$  je podgraf grafu  $G$ , který je izomorfní cestě. Říkáme, že cesta  $P$  vede mezi vrcholy  $x_0$  a  $x_k$ , proto se někdy značí  $x_0Px_k$ .
- **Kružnice:** Kružnice je uzavřená cesta, tedy neprázdný graf  $C = (V, E)$ , kde  $V = \{x_0, x_1, \dots, x_k\}$  a  $E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k\} \cup \{x_k, x_0\}$  a délkou kružnice je myšlen počet jejích hran [5].
- **Kružnice v grafu:** Kružnice v grafu  $G$  je podgraf grafu  $G$ , který je izomorfní kružnici.
- **Souvislý graf:** Graf  $G$  je souvislý, jestliže mezi každou dvojicí vrcholů  $x, y \in V$  existuje cesta.

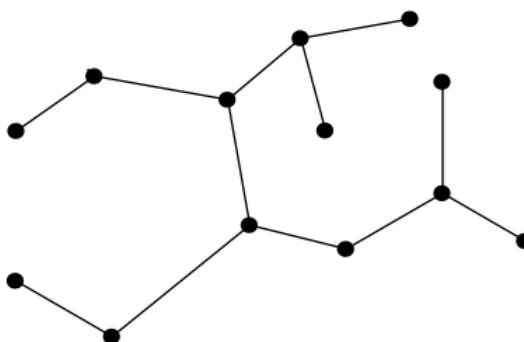


Obr. 7 – Typy grafů: úplný graf  $K_3$ , cesta  $P_3$  a kružnice  $C_3$ .

### 2.1.2 Stromy

Stromy jsou velmi důležitou třídou grafů. Strom  $T$  je souvislý graf bez kružnic [6]. Stromy mají jednoduchou strukturu, a proto se s nimi počítá vcelku triviálně. Díky jejich vlastnostem jsou používány i při návrhu složitějších algoritmů [5].

Pokud se nepožaduje podmínka souvislosti grafu, jde o graf skládající se z několika stromů, který se označuje jako les. Vrcholy stromu stupně 1 se nazývají listy.



Obr. 8 – Strom  $T$

Strom  $T$  je souvislý graf s  $n - 1$  hranami. Přidáním libovolné hrany ke stromu  $T$  vznikne kružnice. Mezi libovolnými dvěma vrcholy stromu  $T$  vede jednoznačně určená cesta. Jednoznačná cesta z vrcholu  $x$  do vrcholu  $y$  ve stromě  $T$  se značí  $xTy$ .

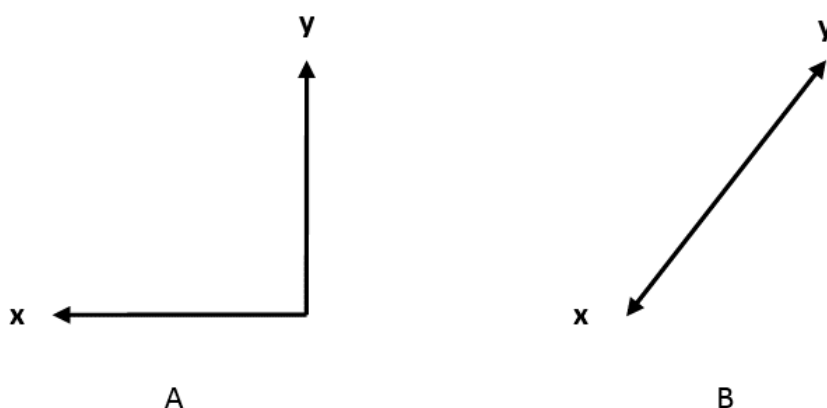
### 2.1.3 Nejkratší cesta v grafu

Velká skupina algoritmů řešících bludiště vychází z problematiky hledání nejkratší cesty v grafu. Jedná se tedy o určení vzdálenosti dvou vrcholů grafu, respektive určení délky hran mezi těmito vrcholy.

Vzdálenost je matematicky popsána pojmem metrika, vzdálenost v grafech je potom nazývána grafová metrika. Nechť  $G$  je graf s ohodnocením hran  $c: E(G) \rightarrow \mathbb{R}$ . Číslo  $c(e)$  se nazývá cena hrany  $e$ , v kontextu hledání nejkratší cesty je ale označováno pojmem délka hrany  $e$ . Délka cesty mezi dvěma vrcholy  $x = v_0 e_1 v_1 e_2 \dots v_{k-1} e_k v_k = y$  se vypočítá jako

$\sum_{i=1}^k c(e)$ . Vzdálenost dvou vrcholů  $x$  a  $y$  v grafové metrice je potom délka nejkratší cesty mezi vrcholy  $x$  a  $y$ . Mezi nejpoužívanější metriky v rovině patří:

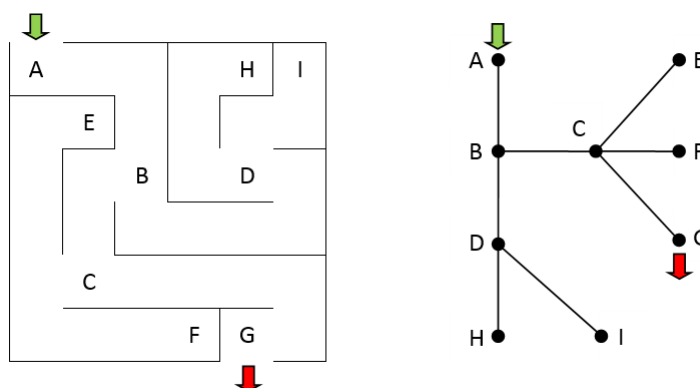
- **eukleidovská metrika** – vzdálenost dvou bodů  $p_1$  a  $p_2$  je délka úsečky  $p_1p_2$ , neboli vzdálenost vzdušnou čarou,
- **maximová metrika** – vzdálenost  $p_1$  a  $p_2$  je maximum z  $|x_1 - x_2|$  a  $|y_1 - y_2|$ ,
- **manhattanská metrika** – je nazvána po Manhattanu v New Yorku, kde se chodí v síti pravoúhlých ulic, které jsou rovnoběžné s osami souřadného systému a vzdálenost  $p_1$  a  $p_2$  je  $|x_1 - x_2| + |y_1 - y_2|$  [6].



Obr. 9 – Příklady metrik: metrika manhattan (A) a eukleidovská metrika (B)

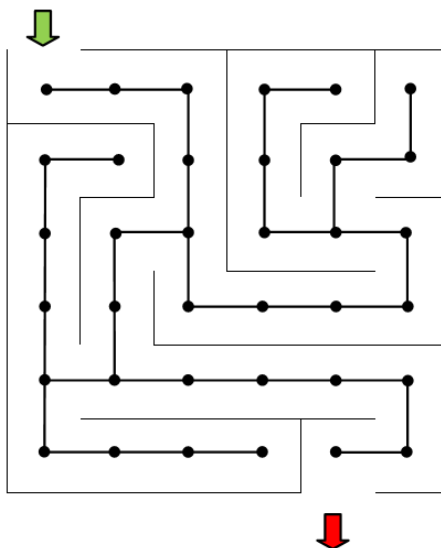
## 2.2 Reprezentace bludiště grafem

Jak tedy vytvořit graf, který bude reprezentovat bludiště? Převodění bludiště je jednoduché. Cesty bludiště budou v grafu odpovídat hranám, křižovatky a slepé konce budou představovat uzly grafu.



Obr. 10 – Klasická reprezentace bludiště grafem

Výše uvedená reprezentace bludiště není složitá, ale je nepraktická z hlediska implementace algoritmů vytvářejících či procházejících bludiště. Mnohem výhodnější je reprezentovat každou buňku bludiště jako uzel grafu a přechody mezi buňkami hranou s cenou 1. Reprezentace bludiště grafem použitá v této práci je uvedena na Obr. 11.

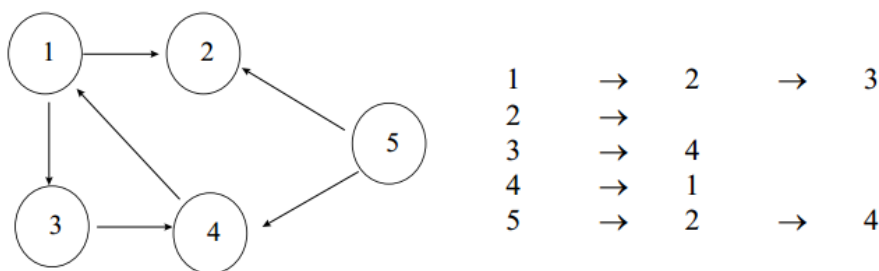


Obr. 11 – Reprezentace bludiště grafem

Výše uvedené reprezentace grafu byly obrazové. Jak ale bude graf popsán matematicky nebo v počítači?

### 2.2.1 Spojová reprezentace

Patrně nejpoužívanějším způsobem matematické reprezentace grafu je spojová reprezentace. Má-li graf  $n$  vrcholů, pak spojová reprezentace obsahuje  $n$  spojových seznamů. Každý z těchto seznamů potom obsahuje ukazatele na všechny vrcholy, do kterých vede hrana z vrcholu  $n$  [6].

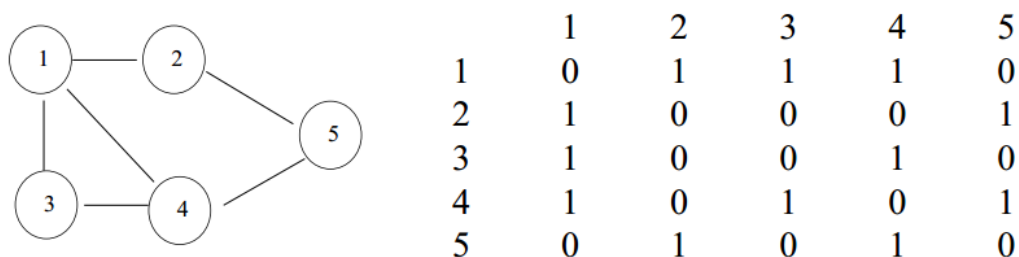


Obr. 12 – Příklad zápisu grafu pomocí spojové reprezentace

### 2.2.2 Matice sousednosti

Další typická matematická reprezentace grafu je zápis pomocí matice sousednosti. Tato matice v podstatě zachycuje, které vrcholy spolu sousedí [5]. Matice sousednosti se pro graf  $G = (V, E)$  s  $n$  vrcholy definuje jako matice  $A = (a_{u,v})$ , která má velikost  $n \times n$ , a kde

$$a_{u,v} = \begin{cases} 1 & \Leftrightarrow uv \in E \\ 0 & \Leftrightarrow uv \notin E \end{cases}$$



Obr. 13 – Příklad zápisu grafu pomocí matice sousednosti. Volně dle [6]

Vhodnost použití jednotlivých reprezentací se odvíjí dle toho, co se bude v grafu dělat. Mezi nejčastější operace v grafu patří:

- Test, zdali dva libovolné vrcholy  $u \in V$  a  $v \in V$  jsou spojeny hranou.
- Průchod všech sousedů vrcholu  $v \in V$ .

V tabulce je pro zajímavost shrnuta časová složitost jednotlivých reprezentací, včetně jejich prostorové složitosti.

Tab. 2 – Složitost reprezentací dle [5]

Reprezentace	Test hrany	Procházení sousedů	Prostorová složitost
matice sousednosti	$O(1)$	$O(n)$	$O(n^2)$
spojová reprezentace	$O(n)$	$O(\text{počet sousedů})$	$O(n + m)$

### 3 ALGORITMY GENERUJÍCÍ BLUDIŠTĚ

Algoritmy generující bludiště můžeme rozdělit do několika skupin, podle toho jakým způsobem je bludiště vytvářeno.

První skupinou jsou algoritmy založené na vytváření cest, tedy na bourání zdí. Pro tyto algoritmy se používá pojem Passage Carvers algoritmy. Příkladem těchto algoritmů v této práci uváděných jsou Primův nebo Kruskalův algoritmus. Druhou skupinou jsou algoritmy, které zdi naopak přidávají. Těm se říká Wall Adders algoritmy. Mezi tyto algoritmy patří například algoritmus půlení intervalů [3].

Existuje i třetí skupina, využívající pro generování bludiště šablon, tedy připravených grafických objektů. Jako příklad můžeme uvést šablonu soustředných kružnic, využitou pro tvorbu theta bludiště.

Obecný princip generování bludiště je obvykle založen na postupné tvorbě bludiště při zachování podmínek vycházejících z klasifikace bludiště, jako například podmínek nevytváření cyklů a oddělených částí u perfektního bludiště.

V dalším textu budou konkrétně představeny nejzajímavější a nejpoužívanější algoritmy vytvářející perfektní bludiště.

#### 3.1 Algoritmy vycházející z teorie grafů

Algoritmy vycházející z teorie grafů. Tyto algoritmy jsou postaveny na hledání minimální kostry grafu.

Je několik možností jak minimální kostry grafu nalézt. Všechny pracují iteračně: postupně konstruují takové podgrafy daného grafu, které se přibližují hledané minimální kostře. První možností je vypouštění hran výchozího grafu, další přidávání hran k podgrafu, který je na počátku prázdný, anebo konečně výměnou hran v nějaké výchozí kostře. Pochopitelně jde o to, aby tyto postupy byly co nejefektivnější, aby iterace nebyly výpočetně složité a postupovaly ke hledané kostře grafu co nejrychleji [7].

Nejefektivnější jsou algoritmy používající postupné přidávání hran. Minimální kostra se začíná vytvářet počínaje prázdnou množinou hran  $T$ , ke které se přidávají vhodné hrany tak dlouho, až se získá úplná minimální kostra. Základní podmínkou tohoto postupu je, že množina hran  $T$  je podmnožinou hran nějaké minimální kostry a tato podmínka se nesmí přidáním nové hrany porušit.



Uvedený postup lze popsat následujícím pseudokódem [3]:

```
1   vytvoř prázdnou množinu T
2   while (T netvoří kostru) do
3       najdi vhodnou hranu [u, v] pro T
4       T := T U {[u, v]}
5   end
6   return T
```

Pro vytváření bludišť je tento postup velmi důležitý a vychází z něho řada dalších algoritmů.

Efektivní průchod grafu musí splňovat následující podmínky:

- Každou hranou se projde jen jednou. Jednou tam a jednou zpět.
- Hranou se jde zpět, až když z vrcholu nevede další cesta.
- Hranou, která vede do už navštíveného vrcholu, se algoritmus vrací ihned zpět.

Pokud algoritmus tyto podmínky splňuje, je konečný a korektní.

### 3.1.1 Algoritmus prohledávání do hloubky

Prvním z představených algoritmů je algoritmus prohledávání do hloubky, v originále Depth First Search (též DFS nebo recursive backtracker algorithm). Je určený k procházení grafu a má široké uplatnění. Jeho principů se využívá při zjišťování topologického uspořádání nebo detekci cyklů daného grafu. Princip algoritmu je jednoduchý a snadno se implementuje s využitím zásobníku.

Algoritmus probírá hrany vycházející z naposled objeveného uzlu, který má ještě neprobrané hrany. Když probere všechny jeho hrany, vrátí se zpět k původnímu uzlu. Z něho zase pokračuje po další dosud neprobrané hraně. Algoritmus pracuje tak dlouho, dokud se neobjeví všechny uzly dosažitelné z prvního výchozího uzlu. Jestliže zbývá nějaký neobjevený uzel, volí se jako další výchozí uzel a algoritmus pokračuje z něho, dokud nejsou objeveny všechny uzly. Algoritmus pro dočasné ukládání nenavštívených uzlů používá zásobník (manipulace s daty metodou LIFO) [5].

Asymptotická složitost algoritmu prohledávání do hloubky je  $O(V + E)$ , kde  $V$  je počet uzlů grafu (celkový počet buněk bludiště) a  $E$  je počet hran grafu.

### 3.1.2 Primův algoritmus

Primův algoritmus (též Jarníkův algoritmus, Jarník-Primův algoritmus, Prim-Jarníkův algoritmus nebo DJP algoritmus) slouží k určení minimální kostry grafu. Minimální kostra

je taková, když součet vah hran kostry je minimální. Algoritmus byl vynalezen v roce 1930 českým matematikem Vojtěchem Jarníkem a v roce 1957 znovu nezávisle objeven americkým matematikem Robertem Primem. Algoritmus byl opět objeven v roce 1959 Edsgerem Dijkstrou.

Algoritmus vychází z libovolného uzlu grafu a udržuje si seznam již objevených uzlů a jejich vzdáleností od propojené části grafu. V každém svém kroku připojí ten z uzlů, mezi nímž a propojenou částí grafu je hrana nejnižší délky a označí sousedy nově připojeného uzlu za objevené, případně zkrátí vzdálenosti od již známých uzlů, pokud byla nalezena výhodnější hrana. Algoritmus končí v okamžiku, kdy jsou propojeny všechny uzly grafu [7].

Asymptotická složitost Primova algoritmu je  $O(|E| + |V| \log|V|)$ , kde  $V$  je počet uzlů grafu (celkový počet buněk bludiště) a  $E$  je počet hran grafu.

### 3.1.3 Kruskalův algoritmus

Kruskalův algoritmus slouží k nalezení minimální kostry souvislého ohodnoceného grafu, jehož hrany mají nezápornou délku. Algoritmus byl poprvé publikován v roce 1956 Josephem Kruskalem a je nejjednodušší implementací meta-algoritmu. Je téměř shodný s algoritmem publikovaným v roce 1926 českým matematikem Otakarem Borůvkou, liší se tím, že Borůvkův algoritmus přidává hrany, zatímco Kruskalův uzly. Oba algoritmy jsou zástupci tzv. hladového hledání (greedy search).

Princip algoritmu je následující. Vytvoříme si z původního grafu nový graf, který obsahuje stejné uzly jako původní graf, avšak žádné hrany. Poté seřadíme hrany do neklesající posloupnosti podle cen hran mezi uzly a postupně přidáváme hrany do nového grafu. Pokud se po přidání hrany vytvoří v grafu kružnice, hranu odejmeme. Přidávání hran opakujeme do doby, než získáme spojitý graf, nebo nám dojdou hrany [7].

Asymptotická složitost Kruskalova algoritmu je  $O(E \log|V|)$ , kde  $V$  je počet uzlů grafu (celkový počet buněk bludiště) a  $E$  je počet hran grafu.

## 3.2 Ellerův algoritmus

Ellerův algoritmus nevychází z teorie grafů a je zajímavý tím, že vytváří bludiště po jednotlivých řádcích. Pro vytvoření nového řádku bludiště algoritmus vychází jen z řádku předchozího. Díky tomu jde o velmi rychlý algoritmus, rychlejší než jiné populární algoritmy používané pro generování bludišť (jako jsou například Primův nebo Kruskalův algoritmus).

Bludiště se generuje v lineárním čase a může být teoreticky nekonečně dlouhé. Algoritmus generování bludiště vychází z toho, že každá buňka v řádku je zahrnuta v unikátním množině (setu). Pokud jsou dvě buňky ve stejném setu, není mezi nimi zeď. Vertikálně musí v každém setu existovat minimálně jedno propojení [7]. Algoritmus je vcelku podobný Kruskalovu algoritmu, ten ale pracuje s celým bludištěm najednou.

Asymptotická složitost Ellerova algoritmu je  $O(r^3)$ , kde  $r$  je počet řádků.

### 3.3 Algoritmus půlení intervalů

Všechny doposud uvedené algoritmy jsou koncipovány jako algoritmy vytvářející cesty. Algoritmus půlení intervalů (v originále recursive division algorithm) je oproti tomu založen na přidávání zdí. Algoritmus je podobný algoritmu prohledávání do hloubky, a oba algoritmy využívají zásobníku.

Algoritmus je založený na neustálém dělení plochy bludiště, jak v horizontálním tak i vertikálním směru a propojením dvou vzniklých „podploch“ bludiště cestou. Zajímavostí tohoto algoritmu je jeho fraktální povaha, teoreticky lze plochy dělit do nekonečna a získat tak víc a víc detailnější bludiště [7].

Asymptotická složitost algoritmu půlení intervalů je  $O(n^2)$ , kde  $n$  je *počet řádku \* počet sloupců*.

## 4 ALGORITMY ŘEŠÍCÍ BLUDIŠTĚ

Vyřešení bludiště je pro člověka výzvou už po několik století. Na otázku jak úspěšně vyřešit bludiště, respektive nalézt jeho řešení, není jednoznačná odpověď. V první řadě je nutné určit, co se chápe pod pojmem „řešení bludiště“. Za řešení totiž lze považovat:

- Nalezení alespoň jedné trasy mezi dvěma libovolnými body bludiště, zejména tedy mezi vstupním a výstupním bodem bludiště.
- Nalezení všech tras mezi dvěma body bludiště.

V rámci této práce se za řešení bludiště bude považovat nalezení nejkratší trasy mezi vstupním a výstupním bodem bludiště.

Řešících algoritmů existuje celá řada, ale obecně se dají rozdělit na dvě skupiny. Jednou jsou algoritmy, které potřebují znát strukturu bludiště. Ty potom ze známé struktury bludiště odstraňují nepotřebné buňky nebo cesty a tím docházejí k řešení. Druhou skupinou jsou algoritmy, které k řešení simulují průchod bludištěm od počátečního do cílového bodu.

V následujících podkapitolách budou ve zkratce představeny některé z algoritmů řešící bludiště.

### 4.1 Algoritmus sledování zdí

Algoritmus je známý pod anglickým názvem Wall follower. Jde o snad nejznámější z algoritmů simulujících průchod bludištěm. Je založený na primitivním principu, kdy při vstupu do bludiště se vybere pravá nebo levá zeď a podél ní se pořád pokračuje (odtud sledování zdí). Směr postupu je daný, používá se i při vstupu do křižovatky a je během celého algoritmu neměnný.

Ne všechny typy bludišť jsou tímto algoritmem řešitelné. V případě perfektních, unicursal a řídkých bludišť algoritmus nalezne řešení vždy. V případě spleteného bludiště může v určitém případě dojít k zacyklení algoritmu, což lze vyřešit označením buněk, které již byly navštíveny a jejich vynechání v případě jejich další návštěvy [3].

### 4.2 Algoritmus vyplňování slepých konců

Algoritmus vyplňování slepých konců (v originále Dead-end filler) je příkladem algoritmu pracujícího s celým bludištěm. Princip algoritmu je opět jednoduchý. Postupně prochází celé bludiště, a pokud narazí na slepý konec, označí si ho a od něho i celou cestu až k nejbližší křižovatce (vyjme buňky z množiny možných řešení). Takto se postupuje do té doby, než

jsou eliminovány všechny slepé konce a na výstupu je jen množina buněk obsahující trasu řešící bludiště. Algoritmus tedy nenachází řešení bludiště jako takové, pouze ho redukuje.

Algoritmus opět nalezne řešení v případě perfektních, unicursal a řídkých bludišť. Ve spletených bludištích algoritmus sice řešení nalezne, ale nedokáže eliminovat cykly, které považuje za součást řešení, tudíž nalezená trasa není nejkratší [3].

### 4.3 Trémauxův algoritmus

Autorem tohoto algoritmu je francouzský matematik Charles Trémaux a je navržen tak, aby mohl být použitý člověkem v reálném bludišti. Postupuje obdobně jako Théseus v krétském labyrintu a jeho princip je také podobný algoritmu prohledávání do hloubky.

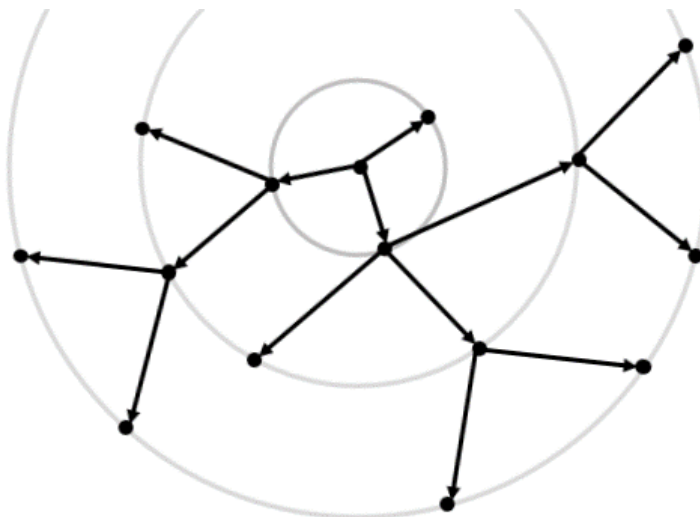
Algoritmus prochází bludiště, přičemž si pamatuje buňky, kterými už prošel. Zde je analogie k Ariadnině niti. Na křižovatce je zvolen směr dalšího postupu náhodně po hraně, která ještě nebyla navštívena. Pokud je cesta slepým koncem, algoritmus se vrací zpět k poslední navštívené křižovatce. Směr ke slepému konci se označí jako nesprávný, analogicky k Théseově křídě, a opět se pokračuje náhodným směrem, dokud nejsou všechny možné směry křižovatky vyčerpány [2].

Algoritmus si opět bez problémů poradí s perfektním, unicursal nebo řídkým bludištěm. Pro řešení spletených bludišť je potřeba zakázat procházení buněk, které jsou již v zásobníku. Díky tomu se cyklus bude jevit jako slepý konec.

### 4.4 Algoritmus hledání nejkratších cest

Algoritmus hledání nejkratších cest vychází z teorie grafů a je implementací metody prohledávání do šířky (v originále Breath First Search, též BFS nebo algoritmus vlny).

Vychází se z podmínky, že hrany grafu jsou stejně dlouhé. Celý graf je potom prozkoumáván postupně ve vlnách. V první vlně se zkoumají vrcholy ve vzdálenosti 1 od vstupního vrcholu, ve druhé vlně vrcholy ve vzdálenosti 2 a tak dále dokud není nalezen cíl. V počítači se vlny řeší pomocí fronty (manipulace s daty metodou FIFO), která je průběžně zpracovávána [5].



Obr. 14 – Znáznornění zpracování grafu ve vlnách

Protože algoritmus prochází celý graf, vždy najde nejkratší cestu mezi dvěma vrcholy. V případě reprezentace bludiště grafem uvedené v kapitole 2.2 je možné nastavit algoritmu ukončovací podmínku při nalezení cílové buňky. Trasu řešení bludiště lze na výstup algoritmu dostat zpětným průchodem nejkratší určené cesty.

#### 4.5 Dijkstrův algoritmus

Dijkstrův algoritmus je nejrychlejší známý algoritmus pro nalezení všech nejkratších cest ze zadaného vrcholu do ostatních vrcholů grafu, který neobsahuje hrany záporné délky. Algoritmus vymyslel v roce 1959 nizozemský informatik Edsger Dijkstra. Aplikace algoritmu jsou populární zejména v oblasti navigačních systémů a počítačových her. Jeho rozšířením o heuristickou část, ve které se odhaduje vzdálenost k předpokládanému cíli, se získá další známý algoritmus nazývaný A\* algoritmus (též A-Star algoritmus) [7].

Vstupem algoritmu jsou ohodnocený graf, počáteční vrchol a cílový vrchol, výstupem pak je nejkratší trasa mezi zadanými vrcholy, nebo zpráva o neexistenci takové trasy.

Na algoritmus lze pohlížet jako na zobecněné prohledávání grafu do šířky, kdy vlna se ale nešíří na základě počtu hran od zdroje, ale v závislosti na vzdálenosti od zdroje, ve smyslu ceny hran. Vlna proto zpracovává jen ty vrcholy, k nimž již byla nalezena nejkratší cesta.

Dijkstrův algoritmus je použitelný pouze tehdy, má-li graf pouze nezáporně ohodnocené hrany. V opačném případě není schopen garantovat, že při zpracování uzlu byla již nalezena nejkratší možná cesta.

## 4.6 Algoritmus náhodná myš

Posledním zde uvedeným algoritmem je algoritmus náhodná myš (v originále Random mouse). Jde o triviální metodu napodobující pohyb myši v bludišti, uvedenou zde pouze jako zajímavost.

Základem algoritmu je náhodný pohyb po bludišti. Na začátku se vybere směr, tím se pokračuje, dokud se nenarazí na cíl, nebo křižovatku. Na křižovatkách se volí směr náhodně, ze slepých konců se vrací zpět. Není vyloučen pohyb po již navštívených cestách nebo v cyklech. Algoritmus nemá žádnou paměť, výstupem je tedy pouze informace o nalezení cíle, a to v čase, který nemusí být konečný [3].

## 5 OPTIMALIZACE MRAVENČÍ KOLONIÍ

Téměř ve všech oborech lidské činnosti můžeme nalézt optimalizační problémy, kdy hledáme nejlepší možné řešení. Tyto problémy lze řešit tradičními optimalizačními metodami, jako jsou numerické metody, variační metody, lineární a nelineární programování nebo dynamické programování, anebo k jejich řešení využijeme optimalizační metody umělé inteligence. Jde například o genetické algoritmy, simulované žíhání, metody Monte-Carlo nebo různé heuristické algoritmy. Tyto metody nalézají využití hlavně při řešení problémů, u kterých neznáme exaktní algoritmus nebo algoritmus známe, ale není pro rozsáhlost problému jeho využití možné. V takových případech se musíme spokojit s postupem, který dokáže v krátkém výpočetním čase najít sice ne optimální, ale velmi kvalitní řešení [8].

Algoritmus optimalizace mravenčí kolonií (v originále *Ant Colony Optimization*, zkráceně ACO) byl poprvé prezentován Marcem Dorigem v roce 1992, který ve své disertační práci popsal algoritmus založený na chování mravenců, který nazval Ant System (AS), a úspěšně ho aplikoval na řešení problému obchodního cestující (TSP). Pravděpodobně nejčastějším a nejúspěšnějším směrem výzkumu, využívajícím algoritmy vycházející z chování mravenčích kolonií, jsou optimalizační problémy. Úspěšné aplikace těchto algoritmů najdeme v oborech optimalizace, robotiky či komunikačních sítí [9].

V následujících kapitolách bude stručně nastíněna biologická inspirace, uveden základní S-ACO algoritmus a obecná podoba ACO meta-heuristiky. V závěru kapitoly budou uvedeny nejčastější aplikace ACO algoritmů.

### 5.1 Biologická inspirace

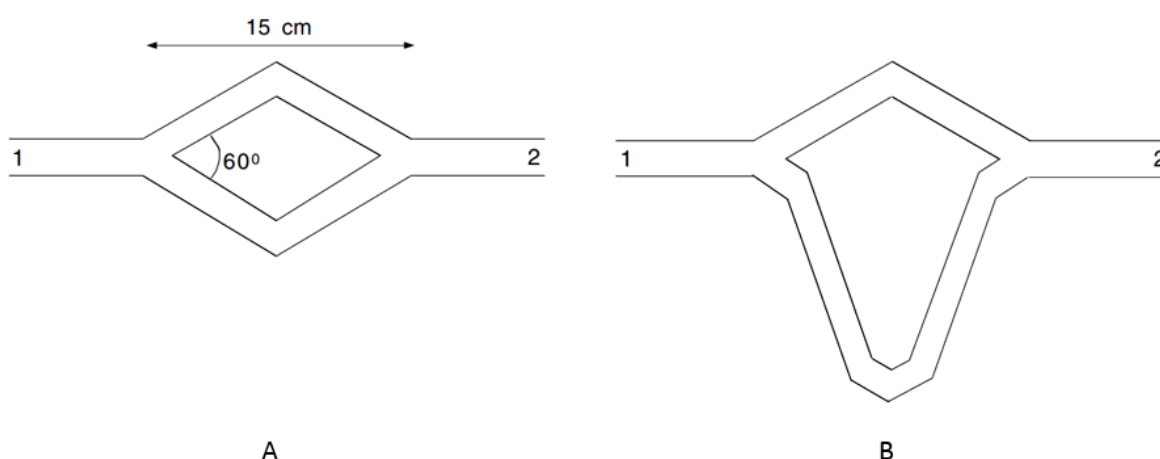
Hmyzí kolonie vykazují vysoký stupeň organizovanosti. Včely, vosy nebo mravenci utvářejí společenstva, kde jednotlivci v různých rolích přispívají jednoduchými úkony ke komplexnímu chování celého roje. V rámci společenstva neexistuje žádný hlavní prvek a jeho chování tedy není řízeno centrálně.

Biologickou inspirací metody optimalizace mravenčí kolonií bylo pozorování kolonií argentinských mravenců (*linepithema humile*). Tito mravenci jsou schopni nalézt, s ohledem na dané podmínky, nejkratší cestu vedoucí mezi mraveništěm a jídlem, které hledají. Fakt, že informace se společenstvím mravenců šíří prostřednictvím interakce mezi jednotlivými členy a také prostřednictvím interakce členů s okolním prostředím, poprvé zaznamenal Pierre-Paul Grassé (1895-1985). Tuto nepřímou komunikaci nazval stigmergie (*stigmergy*)



a popsal, že komplexní chování společenství vychází z jednoduchých komunikačních vzorů. Aplikováním jednoduchých pravidel, kterými se řídí jednotlivý jedinci, pak vzniká komplexní chování celku, které je schopné řešit optimalizační úlohy [10].

Studiem stigmergie založené na feromonech, tedy chemických látkách vylučovaných argentinskými mravenci, se dále s kolegy zabýval Jean-Louis Deneubourg. V experimentu nazvaném "experiment dvou mostů" propojil kolonii argentinských mravenců ke zdroji potravy dvěma cestami stejné délky. Z pozorování vyplynulo, že počáteční náhodný pohyb mravenců při hledání a dopravě potravy do mraveniště byl ovlivňován vylučovanými feromony a putování mravence přestalo být náhodné, jakmile narazil na feromonovou cestu. V okamžiku kdy mravenec našel potravu, začal sám vypouštět feromony a vracel se po feromony značené cestě zpět do mraveniště. Tím byla cesta intenzivněji značkována a postupně přitahovala víc mravenců.



Obr. 15 – Uspořádání „experimentu dvou mostů“, kde obě cesty mají stejnou délku (A) nebo jdou délky cest různé (B). Mravenci putují z mraveniště (1) k potravě (2) a zpět. Volně dle [10].

Experiment dále rozvinul S. Goss a kolegové tak, že jedna cesta k potravě byla podstatně delší. Také v tomto případě, se počáteční náhodný pohyb mravenců ustálil, ale s tím rozdílem, že mravenci si pro potravu volili kratší cestu. Na tomto základě Goss vytvořil matematický model tohoto chování, kdy pokud v jednom okamžiku  $m_1$  mravenců použije první cestu a  $m_2$  mravenců druhou, pak pravděpodobnost  $p_1$  se kterou si nový mravenec vybere první cestu je:

$$p_1 = \frac{(m_1 + k)^h}{(m_1 + k)^h + (m_2 + k)^h}$$

kde parametry  $k$  a  $h$  vychází z experimentálních dat. Popsaný model chování mravenců byl potom hlavní inspirací pro vytvoření algoritmů Optimalizace mravenčí kolonií.

Původní algoritmus Optimalizace mravenčí kolonií nazvaný Ant System (AS), byl publikovaný Marcem Dorigem a Thomasem Stützlem v roce 1992. Od té doby byla uvedena celá řada jeho variant, jejich zkrácený přehled je uveden v tabulce (Tab. 3).

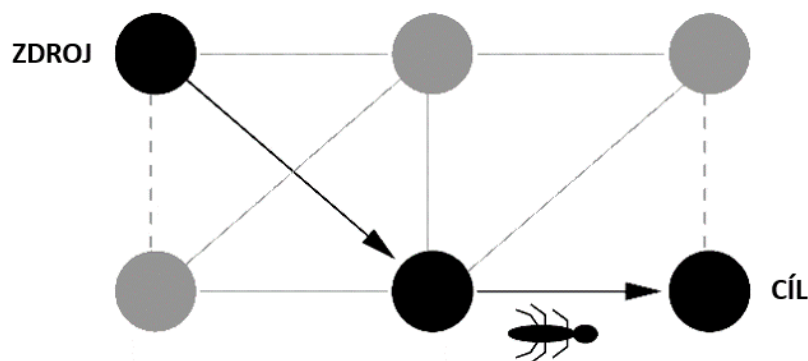
Všechny níže uvedené algoritmy vycházejí ze stejné myšlenky, kterou je napodobování mravenčího chování. Výpočetní metoda řešící komplexní problémy je potom vyjádřena jako hledání optimální cesty v grafu.

Tab. 3 – Neúplný seznam variant ACO algoritmů (řazeno chronologicky) [10]

Algoritmus	Autoři	Rok uvedení
Ant System	M. Dorigo, V. Maniezzo a A. Coloni	1991
Elitist Ant System	M. Dorigo	1992
Ant-Q	L.M. Gambardella a M. Dorigo	1995
Ant Colony System	M. Dorigo a L.M. Gambardella	1996
MIN-MAX Ant System	T. Stützle and H. H. Hoos	1996
Rank-based Ant System	B. Bullnheimer, R. F. Hartl a C. Strauss	1997
Ants	V. Maniezzo	1999
Hyper-Cube Ant System	C. Blum, A. Roli, a M. Dorigo	2001

## 5.2 S-ACO algoritmus

Dále bude uveden princip jednoduchého algoritmu S-ACO (v originále *Simple Ant Colony Optimization algorithm*), na kterém bude ilustrováno základní chování ACO algoritmů a uvedení jejich elementárních prvků.



Obr. 16 – Mravenci hledající optimální cestu v grafu. Volně dle [9].

Úkolem mravenců je sehnat co nejvíce potravy a po co nejkratší cestě (tedy v nejkratším čase) ji donést zpět do mraveniště. Kratší cesta je proto postupem času značkováána feromony silněji než cesta delší, kde feromony se s časem vypařují, a delší cesta tedy přestává být pro mravence atraktivní [8].

Umělí mravenci tedy podobně jako jejich přírodní vzory mají za úkol nalézt nejkratší možné spojení mezi dvěma vrcholy grafu, který vhodně reprezentuje daný optimalizační problém. Necht'  $G = (E, V)$  je graf s  $e = |E|$  vrcholy. Algoritmus optimalizace mravenčí kolonií může v grafu  $G$  nalézt nejkratší cestu mezi zdrojovým vrcholem  $s$  a cílovým vrcholem  $d$ , přičemž délka cesty je dána počtem hran, které tvoří tuto cestu.

Ke každé cestě grafu mezi dvěma libovolnými vrcholy  $(i, j)$  je přiřazena proměnná  $\tau_{ij}$  nazývaná umělý feromon (v originále *artificial pheromone*). Tato proměnná je čtena a zapisována umělými mravenci putujícími po hranách grafu. Množství tohoto umělého feromonu, tedy jeho intenzita, je výchozí parametr pro umělé mravence při určení, zdali daná cesta může být součástí vhodného řešení. Každý umělý mravenec postupně vytváří řešení problému [10].

Na každém vrcholu je další postup umělého mravence určen stochasticky. Rozhodovací pravidlo pro mravence  $k$  nacházejícího se na vrcholu  $i$  využije intenzitu umělého feromonu  $\tau_{ij}$  k určení s jakou pravděpodobností  $p_{ij}^k$  bude k dalšímu postupu vybrán vrchol  $e \in N_i$ , kde  $N_i$  jsou všechny sousední vrcholy k vrcholu  $i$ :

$$p_{ij}^k = \begin{cases} \frac{\tau_{ij}^\alpha}{\sum_{l \in N_i^k} \tau_{il}^\alpha} \dots \text{kdýž } j \in N_i \\ 0 \dots \text{kdýž } j \notin N_i \end{cases}$$

kde parametr  $\alpha$  určuje vliv feromonů na výběr následujícího vrcholu [10]. V průběhu hledání řešení mravenci přidávají feromon k cestám, které použijí. V základní verzi algoritmu je množství tohoto feromonu konstantní, tedy je vždy přidáno  $\Delta\tau$  feromonu. V případě že se v čase  $t$  přemístí umělý mravenec  $k$  mezi vrcholy  $i$  a  $j$ , změní se intenzita feromonu  $\tau_{ij}$  následovně [10]:

$$\tau_{ij}(t) \leftarrow \tau_{ij}(t) + \Delta\tau^k$$

Tímto pravidlem, které simuluje přidávání feromonů reálných mravenců v přírodě, se použitím cesty mezi vrcholy  $i$  a  $j$  zvýší pravděpodobnost jejího užití při dalším výběru.

Zjednodušeně řečeno, čím více je cesta využívána, tím větší je pravděpodobnost jejího výběru v budoucnosti.

Je zde ale nebezpečí, že díky této vlastnosti sice umělí mravenci naleznou řešení, toto ale nebude nejlepší. Z tohoto důvodu je do algoritmu přidán prohledávací mechanismus – kdy podobně jako u reálné feromonové cesty, se umělý feromon vypařuje. Postupným automatickým snižováním intenzity feromonu v každé kroku algoritmu je přirozeně upřednostněno vybírání a prohledání nových cest. Princip vypařování je zajištěn jednoduše, v každé iteraci je množství feromonu sníženo,  $\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}$ , kde parametr určující rychlost vypařování feromonu  $\rho \in (0,1]$  [10].

Samotná optimalizace potom probíhá v několika po sobě jdoucích iteracích, kdy v každé iteraci provede  $m$  mravenců své úkoly a následně se vyhodnotí cesty, které mravenci objevili. Proces končí splněním ukončujících podmínek, jako jsou například nalezení optimálního řešení, předem definovaný počet iterací bez zlepšení nebo jen dosažení určitého počtu iterací.

Testování takto nastaveného algoritmu prokázalo, že algoritmus je efektivně schopen nalézt nejkratší cestu mezi dvěma body, představující mraveniště a zdroj potravy. Experimenty ale také ukázaly, že čím víc roste složitost procházeného grafu, tím je algoritmus víc nestabilní a roste důležitost nastavených parametrů algoritmu [10].

Takto definovaný algoritmus ale má z důvodu jednoduchosti také řadu omezení. V následující kapitole bude představen algoritmus, který má základ v S-ACO algoritmu, ale je rozšířen o několik vlastností, které jeho omezení odstraňují.

### 5.3 ACO meta-heuristika

Heuristika obecně znamená zkusmé nalezení řešení problémů, pro něž neznáme přesný řešící algoritmus nebo metodu. Heuristické řešení daného problému je často jen přibližné, založené na poučeném odhadu, intuici, zkušenosti nebo prostě na zdravém rozumu. První odhad řešení se může postupně zlepšovat, i když heuristika nikdy nezaručuje nalezení nejlepšího řešení. Zato je univerzálně použitelná, jednoduchá a rychlá [8].

Meta-heuristika potom řeší problémy na velmi obecné úrovni pomocí kombinace heuristik a objektivních funkcí bez znalosti jejich vnitřního fungování. Přistupuje k nim jako k tzv. černým skříňkám. Příklady meta-heuristik jsou třeba simulované žíhání nebo tabu search.

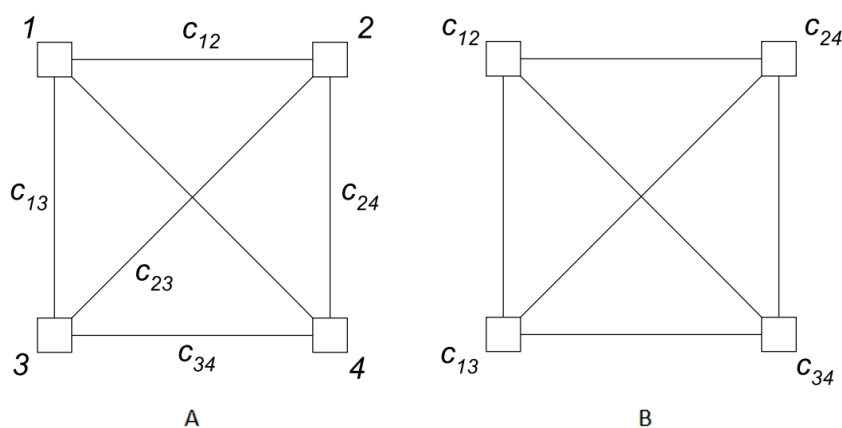
Model ACO meta-heuristiky byl formulován Marcem Dorigem a kolegy v roce 1999 a je založen na rozšíření S-ACO algoritmu, kdy umělým mravencům jsou přidány vlastnosti, které reální mravenci nemají, ale které překonávají omezení S-ACO algoritmu. Například určitá forma paměti.

Dále uvedená meta-heuristika může potom být aplikována na diskretní optimalizační problémy, které lze popsat následujícím modelem  $P = (\mathcal{S}, \Omega, f)$ , kde [10]:

- Množina  $\mathcal{S}$  je prohledávaným prostorem řešení definovaným nad konečnou množinou diskretních rozhodovacích proměnných  $X_i, i = 1, \dots, n$ .
- $\Omega$  je konečná množina omezujících podmínek mezi proměnnými.
- A funkce  $f$  je objektivní funkcí  $f: \mathcal{S} \rightarrow \mathbb{R}_0^+$ , která má být minimalizovaná nebo maximalizovaná.

Obecná proměnná  $X_i$  má hodnoty v množině  $D_i = \{v_1, v_2, \dots, v_i^{|D_i|}\}$ . Úspěšným řešením  $s \in \mathcal{S}$  je nazváno přiřazení všech hodnot k proměnným  $X_i$  beze zbytku, které splňují všechny omezující podmínky v množině  $\Omega$ . Řešení  $s^* \in \mathcal{S}$  je nazváno globálním minimem nebo maximem pouze tehdy když  $f(s^*) \leq f(s) \forall s \in \mathcal{S}$  [10].

Uvedený model optimalizačního problému je použitý k definování modelu využití feromonů u ACO meta-heuristiky. Hodnota feromonu  $\tau_{ij}$  je přiřazena ke každému možnému propojení  $c_{ij}$  tvořící prvky řešení hodnoty  $v_i^j$  k proměnné  $X_i$ . Tato propojení tvoří množinu  $\mathcal{C}$ .



Obr. 17 – Příklad konstrukčního grafu kde prvky řešení jsou tvořeny hranami (A) nebo vrcholy (B) grafu.

Jak již bylo zmíněno, umělý mravenec vytváří řešení optimalizačního problému procházením konstrukčního grafu  $G = (E, V)$ , kde  $V$  je množina vrcholů a  $E$  je množina hran mezi nimi. Graf může být vytvořen z dvěma různými způsoby, kdy prvky řešení (propojení  $c_{ij}$ ) jsou reprezentovány buď vrcholy, nebo hranami. Rozdílný způsob reprezentace grafů je uveden na Obr. 17.

Zjednodušeně lze potom ACO meta-heuristiku rozdělit do tří fází: vytváření řešení, aktualizace feromonu a vnější zásahy. Tyto tři fáze se nemusejí odehrávat postupně, ale naopak asynchronně a paralelně. Záleží pouze na volbě autora implementace, jaký přístup zvolí. Jednotlivé fáze algoritmu jsou popsány dále v této podkapitole.

Algoritmus samotný lze pak popsat následujícím pseudokódem (volně dle [10]):

```
1   nastav parametry
2   inicializuj feromonové stopy
3
4   procedure ACO_metaheuristika()
5       while (nejdou splněny podmínky pro ukončení) do
6           mravenci_cinnost()
7           vyparování_feromonu()
8           (vnější zásahy)
9       end
10  end
11
12  procedure mravenci_cinnost()
13      while (jsou mravenci k dispozici) do
14          vytvoř_mravence()
15          aktivuj_mravence()
16      end
17  end
18
19  procedure aktivuj_mravence()
20      inicializuj_mravence()
21      M = aktualizuj_paměť_mravence()
22      while (aktuální_stav != hledaný_stav) do
23          A = nacti_sousední_uzly()
24          P = spočítej_pravděpodobnosti(A, M,  $\Omega$ )
25          následující_stav = vyber_posun(P,  $\Omega$ )
26          posun_se_do_stavu(následující_stav)
27          M = aktualizuj_paměť_mravence
28      end
29  end
```

### 5.3.1 Fáze vytváření řešení

V této fázi všichni umělí mravenci paralelně a asynchronně procházejí sousedícími vrcholy konstrukčního grafu a postupně staví svá řešení, přičemž využívají feromonových stop a

dodatečných heuristických informací o úloze, jako například v případě problému obchodního cestujícího upřednostňují kratší hrany grafu. Jakmile umělý mravenec dokončí sestavování svého řešení, vyhodnotí ho, tedy zjistí hodnotu účelové funkce pro toto řešení a tato hodnota bude použita v následující fázi aktualizace feromonu.

### 5.3.2 Fáze aktualizace feromonu

Aktualizace feromonu upravuje intenzitu feromonových stop na použitých cestách v konstrukčním grafu. Intenzita feromonu se buď zvyšuje, nebo snižuje. Ke zvýšení typicky dochází, pokud je stopa používána mnoha umělými mravenci. Ke značnému zvýšení ale může dojít i tedy, použije-li spojení jeden umělý mravenec, který však najde velmi úspěšné řešení. Feromonová stopa tak zanechává ostatním umělým mravencům informaci o preferované volbě.

Intenzita feromonu se naopak snižuje v průběhu času, pokud tedy umělí mravenci spojení používají málo nebo vůbec, preference této volby se postupně vytratí. Tento mechanismus představuje užitečnou formu zapomínání, která zabraňuje předčasné konvergenci do lokálních optim.

### 5.3.3 Vnější zásahy

Vnější zásahy slouží k provedení úprav výpočtu, které nelze provádět na úrovni umělého mravence. Tato fáze není nutná, je volitelná, ale používá se k různým vylepšením řešení z globální perspektivy. Někdy se jedná se buď o dodatečné lokální hledání v okolí nalezených řešení. Může však jít i o zvýhodnění nejlepšího nalezeného řešení pomocí posílení jím použitých feromonových stop.

## 5.4 Aplikace ACO

Variety algoritmů optimalizace mravenčí kolonií vycházejí z původního algoritmu, který různým způsobem rozšiřují anebo upravují. Jednou ze základních modifikací je přidání tzv. elitismu, který spočívá v posílení vlivu nejlepších dosud nalezených řešení na průběh výpočtu. Dalšími variantami mravenčích algoritmů je například systém založený na pořadí anebo systém využívající prvky tabu search.

Algoritmy optimalizace mravenčí kolonií a jejich varianty byly úspěšně aplikovány na spoustu optimalizačních problémů. Většina těchto problémů se řadí mezi NP-těžké

problémy, tedy problémy, které není žádný známý algoritmus schopen uspokojivě řešit v reálném čase.

Dále budou uvedeny dvě nejznámější aplikace mravenčích algoritmů, jejichž vyčerpávající popis lze nalézt v [10].

#### 5.4.1 Problém obchodního cestujícího

Typickým příkladem NP-těžkého problému a zároveň i problému, který byl optimalizací mravenčí kolonií řešen jako první, je **problém obchodního cestujícího** (v originále Traveling Salesman Problem, též TSP).

Úkolem obchodního cestujícího je nalézt optimální trasu zadanou množinou měst tak, že každé město obchodník navštíví jen jednou a délka této trasy bude minimální [8]. Problém obchodního cestujícího je tedy teorií grafů definován jako nalezení Hamiltonovské kružnice minimální délky v orientovaném grafu.

Volba trasy, ukládání a vypařování feromonů je potom řízeno konkrétním zvoleným mravenčím algoritmem, který je k řešení problému použit.

#### 5.4.2 Problém směrování komunikačních sítí

Komunikační sítě obecně mohou být rozděleny sítě využívající technologii přepojování okruhů (circuit-switched) a sítě využívající technologii přepojování paketů (packet-switched). Typickým příkladem sítě s přepojováním okruhů je telefonní síť, ve které virtuální či fyzický propoj zůstává stejný po celou dobu komunikace. Oproti tomu v sítích využívající přepojování paketů (tzv. data networks) každý paket putuje různou trasou.

**Problém směrování komunikačních sítí** (anglicky *routing*, též routování) se potom dá zjednodušeně popsat jako problém vytváření a používání směrovacích tabulek směřující datové toky tak, aby měřitelný výkon počítačové sítě byl maximální, aby tedy síť měla co největší prostupnost [10].

Mravenčí algoritmy byly v této oblasti poprvé úspěšně aplikovány variantou ACO algoritmu AntNet v roce 1998.



## **II. PRAKTICKÁ ČÁST**

## 6 IMPLEMENTACE ALGORITMŮ

Součástí práce je také vytvoření aplikace a implementace jednotlivých algoritmů, jak algoritmů bludiště vytvářejících, tak řešících, včetně jednoduché a přehledné vizualizace toho, jak pracují. Díky tomu je možné snáze pochopit a porovnat jednotlivé algoritmy, zobrazit tvorbu a řešení bludiště.

### 6.1 Použité technologie

Pro tvorbu programu pro vizualizaci jednotlivých algoritmů byl použit programovací jazyk Java a vývojové prostředí IntelliJ IDEA (Community edition). Samotný program je potom spouštěn v internetovém prohlížeči jako javový applet.

#### 6.1.1 Java

Java je programovací jazyk, vyvinutý firmou Sun Microsystems v roce 1995. Je používán pro programy, které pracují na různých systémech počínaje čipovými kartami (platforma JavaCard), přes mobilní telefony a různá zabudovaná zařízení (platforma Java ME), aplikace pro desktopové počítače (platforma Java SE) až po rozsáhlé distribuované systémy pracující na řadě spolupracujících počítačů rozprostřené po celém světě (platforma Java EE). Tyto technologie se jako celek nazývají platforma Java [11]. V roce 2007 firma Sun uvolnila zdrojové kódy a Java je dále vyvíjena jako open source. Mezi základní vlastnosti jazyka Java patří:

- jazyk má jednoduchou syntaxi vycházející ze syntaxe jazyka C/C++,
- je objektově orientovaný, kdy vyjma osmi primitivních datových typů jsou všechny ostatní datové typy objektové,
- a je interpretovaný, tedy místo skutečného strojového kódu vytváří tzv. mezikód (též bajtkód), který může pracovat na libovolné platformě, pro kterou existuje interpret jazyka Java (tzv. virtuální stroj Javy neboli JVM).

Hlavní výhody programovacího jazyka Java jsou jeho robustnost a přenositelnost. Jazyk Java je určen pro psaní vysoce spolehlivých programů a z tohoto důvodu neumožňuje některé konstrukce, které bývají častou příčinou chyb. Podporuje také zpracování vícevláknových aplikací [11].

Programovací jazyk Java nepřináší jen samé výhody. Proti jazykům, které provádějí tzv. statickou kompilaci (např. C/C++), je start programů psaných v Javě pomalejší, protože

prostředí musí program nejprve přeložit a potom teprve spustit. Další nevýhodou projevující se hlavně u jednodušších programů je větší paměťová náročnost při běhu programu, způsobená nutností mít v paměti celé prostředí.

Java poskytuje i nástroje pro tvorbu grafického uživatelského rozhraní. Už od první verze je její součástí balík `java.awt` (abstract window toolkit, též AWT) poskytující třídy pro tvorbu rozhraní a od verze 1.2 také komplexnější a propracovanější balík `javax.swing`. Při implementaci programu byl použit právě balík AWT. Ten je sice dnes již považován za zastaralý, protože neobsahuje některé dnes zcela běžné grafické komponenty, ale pro vizualizaci práce algoritmů byla dostačující základní řada komponent, které jsou jeho součástí [12].

### 6.1.2 Java applet

Programy napsané v Javě se podle cílového použití dělí na dvě skupiny, kdy první skupinou jsou aplikace (běžné programy) a druhou skupinou jsou tzv. applety. Tyto pro svůj běh vyžadují kompatibilní prohlížeč internetových stránek a nainstalovaný interpreter Javy (JRE). Nejsou tedy spouštěny přímo, jako aplikace, ale jsou spouštěny při otevření HTML dokumentu v prohlížeči internetových stránek. Provádění programu je ale u obou skupin shodné, program je interpretován.

Aplikace byla naprogramována ve formě Java appletu hlavně proto, že takto může být spuštěna ve většině internetových prohlížečích a na více operačních systémech. Navíc i přes jednoduchý design programu, který je diktovaný následným umístěním appletu do HTML dokumentu, lze využít všechny důležité ovládací prvky, které zajistí dostatečnou míru interaktivity aplikace s uživatelem.

Jako ladící nástroj při tvorbě programu byl použit Applet Viewer, samostatný program pro zobrazování appletů v rámci HTML souboru nebo už v průběhu jejich návrhu. Jeho výhodou je malá paměťová náročnost (v porovnání s dnešními robustními prohlížeči internetových stránek) a tím pádem i téměř nulové omezení rychlosti běhu appletů.

## 7 IMPLEMENTACE ALGORITMŮ

V následující kapitole budou podrobněji popsány v jednotlivé algoritmy jak generující tak řešící bludiště. Řešícímu algoritmu ACO pak bude pro jeho složitost věnována samostatná kapitola.

### 7.1 Datový model reprezentující bludiště

Použitým datovým modelem tvořícím bludiště je třída buňka (**GCell**). Tato třída obsahuje základní informaci o buňce, a to o její pozici v bludišti. Dále obsahuje také informace o existenci zdí kolem ní a také pomocné pole obsahující informace pro vizualizaci buňky v programu a pole, které využívají jednotlivé algoritmy.

```
1    class GCell {
2        int x;           // souřadnice X
3        int y;           // souřadnice Y
4        int bits;        // určení zdí kolem buňky
5        int state;       // stav buňky pro vykreslení
6        int temp;        // pomocná proměnná
7        double phero;    // hodnota feromonu pro ACO
8    }
```

Existenci zdí mezi jednotlivými buňkami zajišťuje výčtový typ **DIR**, pomocí kterého lze určit souřadnice jejího souseda v daném směru a bitovým porovnáním potom určit zdali mezi dvěma sousedními buňkami existuje zeď či nikoliv. Směry jsou definovány následovně:

```
1    NORTH (1, 0, -1)
2    SOUTH (2, 0, 1)
3    EAST (4, 1, 0)
4    WEST (8, -1, 0)
5
6    NORTH.opposite = SOUTH
7    SOUTH.opposite = NORTH
8    EAST.opposite = WEST
9    WEST.opposite = EAST
```

Jednotlivé parametry u definovaných směrů značí:

- První parametr značí bitou změnu, tedy není-li v daném směru postavena zeď. Pokud má buňka ze všech čtyř stran postavené zdi, hodnota jejího parametru bits = 0. Pokud nemá buňka ze všech čtyř stran žádnou zeď, pak analogicky tomu je hodnota

jejího parametru  $\text{bits} = 15$ . Pokud je buňka například slepým koncem s východem směrem SOUTH, je hodnota jejího parametru  $\text{bits} = 2$ .

- Další dva parametry potom určují změnu souřadnic  $(x, y)$  aktuální buňky v určeném směru. Například je-li aktuální buňka určena souřadnicemi  $(2, 2)$ , ve směru NORTH má její soused souřadnice  $(2 + 0, 2 + (-1)) = (2, 1)$ .

Bludiště je poté vytvořeno jako dynamické pole buněk (kolekce `ArrayList <GCell>`). Tuto metodu byla použita právě proto, že v rámci programu jsou vizualizovány a řešeny bludiště o různých velikostech a klasické dvourozměrné pole by muselo být alokováno v paměti v maximální velikosti, kdežto při použití dynamického pole lze alokovat je potřebnou část paměti danou právě vybranou velikostí bludiště.

Prvek v dynamickém poli na základě souřadnic buňky  $(x, y)$  se potom určí přepočtem  $\text{index\_pole} = (x + (y * W))$ , kde  $W$  je aktuální šířka bludiště.

## 7.2 Algoritmy generující bludiště

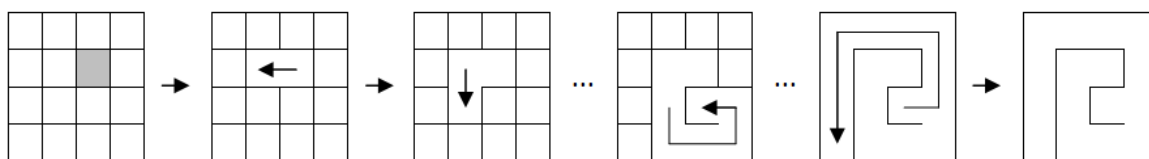
Bludiště je níže uvedenými algoritmy generováno náhodně. To znamená, že vychází-li algoritmus z nějaké počáteční buňky nebo uzlu grafu, je tato volena náhodně.

### 7.2.1 Algoritmus prohledávání do hloubky

Algoritmus na začátku zvolí libovolný uzel a označí ho jako otevřený, zpracuje ho a zavolá sám sebe na všechny dosud neobjevené sousedy daného uzlu. Po návratu z rekurze uzel označí jako uzavřený. Takto dojde k průchodu všech větví grafu do maximální hloubky [5].

Algoritmus používá rekurzi, což může způsobovat problémy při generování velkých bludišť, například při vyčerpání prostoru zásobníku. Tomuto lze předejít využitím vlastního zásobníku, který bude reprezentovat cestu od počáteční k právě prohledávané buňce. Místo rekurzivního volání algoritmu se pak prochází vytvořený zásobník, dokud na něm je něco uloženo a zpracovává se buňka na jeho vrcholu.

Princip algoritmu je znázorněn na obrázku Obr. 18.



Obr. 18 – Vytvoření bludiště algoritmem prohledávání do hloubky

V rámci implementace tohoto algoritmu byla použita varianta s rekurzí, jejíž pseudokód vypadá potom následovně:

```
1  procedure DFS (buňka)
2      označ buňku za navštívenou
3      foreach (sousední buňka)
4          if not (sousední buňka navštívena) then
5              odstraň zeď mezi buňkou a sousední
                buňkou
6              DFS (sousední buňka)
7          end
8      end
9  end
10
11  DFS (náhodná počáteční buňka)
```

### 7.2.2 Primův algoritmus

Jak již bylo uvedeno v kapitole 3.1.2, Primův algoritmus vychází z libovolného uzlu grafu a udržuje si seznam již objevených uzlů a jejich vzdáleností od propojené části grafu. V každém svém kroku připojí ten z uzlů, mezi nímž a propojenou částí grafu je hrana nejmenší délky a označí sousedy nově připojeného uzlu za objevené, případně zkrátí vzdálenosti od již známých uzlů, pokud byla nalezena výhodnější hrana. Algoritmus končí v okamžiku, kdy jsou propojeny všechny uzly grafu.

Bludiště se potom pomocí tohoto algoritmu generuje následovně:

1. Vytvoří se tři množiny: vnitřní V, na pomezí P a mimo bludiště M.
2. Při inicializaci se všechny buňky bludiště umístí do množiny mimo bludiště M.
3. Náhodně se vybere buňka a přesune do množiny vnitřní V a její sousední buňky se přesunou do množiny na pomezí P.
4. Následně se z množiny P vybírají buňky, dokud množina není prázdná a s každou vybranou buňkou se provedou dvě operace: buňka se přesune do množiny V a probourá se zeď do libovolné sousední buňky, která je již v množině V. Všechny sousedící buňky k vybrané buňce z množiny M se přesunou do množiny P.

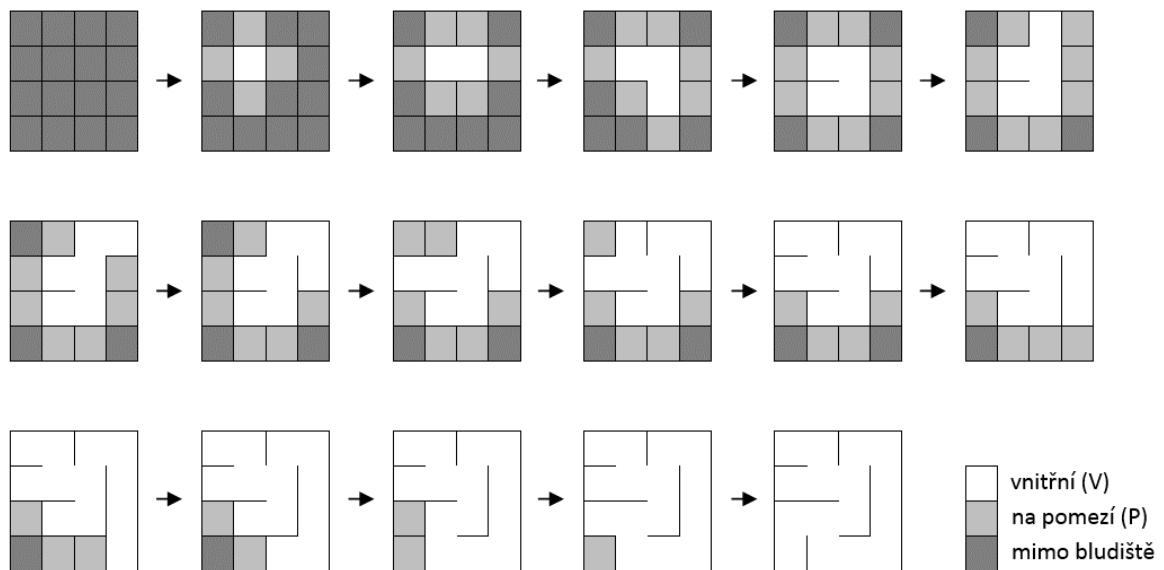
Je zřejmé, že vygenerovaná část bludiště (množina V) tvoří strom tvořící kostru. Buňky v množině na pomezí (množina P) reprezentují uzly ležící mimo kostru grafu, ale obsahují hrany vedoucí k uzlům. Tyto hrany jsou algoritmem postupně vybírány a přidávány do kostry grafu, kdy přidaná hrana ve skutečnosti reprezentuje zbořenou zeď mezi buňkami.

Princip algoritmu je znázorněn na obrázku Obr. 19 a lze ho popsat následujícím pseudokódem [7]:

```

1   vytvoř prázdné množiny V, P, M
2   foreach (buňka v bludišti)
3       přidej buňku do M
4   end
5
6   vyber náhodnou buňku v bludišti
7   přidej ji do P
8
9   while (P je neprázdná) do
10      vyber náhodnou buňku z P
11      přidej ji do V
12      odeber ji z P
13      odstraň zeď mezi vybranou buňkou a libovolnou
        sousední buňkou, která je ve V
14
15      foreach (sousední buňka v M)
16          přidej sousední buňku do P
17      end
18  end

```



Obr. 19 – Vytvoření bludiště Primovým algoritmem

### 7.2.3 Kruskalův algoritmus

Jak již bylo uvedeno v kapitole 3.1.3, vytvoříme si z původního grafu nový graf obsahující stejné uzly jako původní graf, avšak žádné hrany. Poté seřadíme hrany do neklesající posloupnosti podle cen hran mezi uzly a postupně přidáváme hrany do nového grafu. Pokud

se po přidání hrany vytvoří v grafu kružnice, hranu odejmeme. Přidávání hran opakujeme do doby, než získáme spojitý graf, nebo nám dojdou hrany. Tvorba bludiště Kruskalovým algoritmem je znázorněna na obrázku Obr. 20.

Bludiště je potom algoritmem generováno následovně:

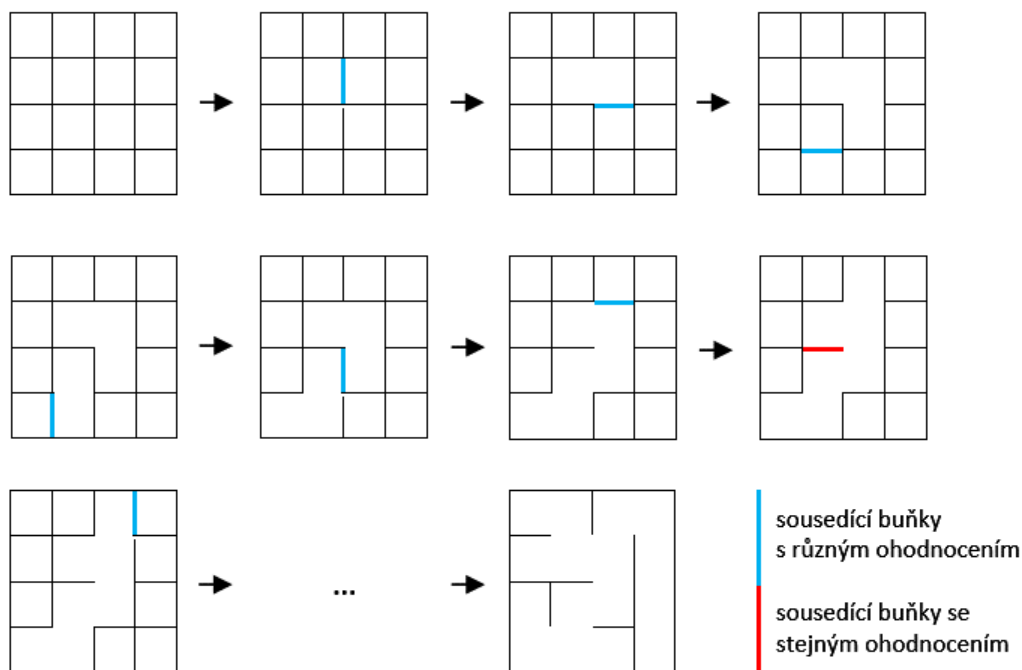
1. Vytvoří se množina les  $L$  obsahující všechny buňky bludiště, přičemž každá buňka je unikátně ohodnocena (například barvou nebo číslem).
2. Vytvoří se množina strom  $S$  obsahující všechny zdi mezi dvěma buňkami.
3. Dokud není množina  $S$  prázdná, náhodně se z ní vybírají zdi a s každou vybranou zdí se provedou následující dvě operace: porovná se ohodnocení (barva) sousedících buněk, které zeď propojuje, a pokud je ohodnocení různé (buňky mají rozdílnou barvu) je zeď zbourána a buňky se ohodnotí stejně (obě nyní mají shodnou barvu). Vybraná zeď je z množiny  $S$  odebrána.

Na počátku bude mít bludiště všechny zdi postaveny. Hodnota všech hran v grafu je stejná a algoritmus z nich vybírá náhodně. Pokud je hrana vybrána a vložena do množiny les  $L$  pak je zeď k hraně náležící zbořena. Výstupem algoritmu je tedy jeden strom, který reprezentuje vygenerované bludiště.

Pseudokód popisující algoritmus vypadá potom následovně [7]:

```
1   vytvoř prázdné množiny L, S
2   foreach (buňka v bludišti)
3       přidej buňku do L a označ buňku unikátním číslem
4       přidej zdi buňky do S
5   end
6
7   while (S je neprázdná) do
8       vyber náhodnou zeď z S
9       if (čísla buněk z L v S jsou různá) then
10          odstraň zeď
11          sluč čísla vybraných buněk z L
12       end
13       odstraň zeď z S
14  end
```





Obr. 20 – Vytvoření bludiště Kruskalovým algoritmem

#### 7.2.4 Ellerův algoritmus

Jak bylo uvedeno v kapitole 3.2, Ellerův algoritmus vytváří bludiště po jednotlivých řádcích. Pro vygenerování jednoho řádku jsou zapotřebí jeho tři průchody. Algoritmus pracuje následovně (viz také obrázek Obr. 21):

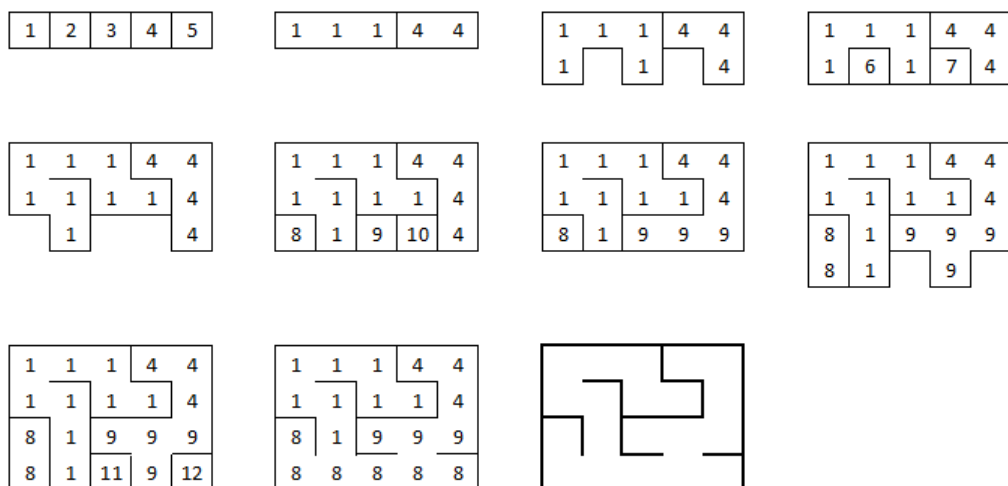
1. Každá buňka prvního řádku bludiště se při inicializaci unikátně ohodnotí, například barvou či číslem, podobně jako u Kruskalova algoritmu.
2. Náhodně se zbourají zdi mezi sousedícími buňkami, které mají různá ohodnocení. Propojeným buňkám se sloučí ohodnocení.
3. Náhodně se vyberou vertikální propojení na následující řádek. Vždy ale musí existovat alespoň jedno vertikální propojení mezi sloučenými buňkami v řádku.
4. Vygeneruje se nový řádek. Buňky, které jsou vertikálně propojené s předchozím řádkem, jsou ohodnoceny stejně jako jejich vertikální sousedé. Buňky, které nemají vertikální propojení, jsou opět ohodnoceny unikátně.
5. Algoritmus zopakuje kroky propojení sousedících buněk v rámci nového řádku (krok 2) a vytvoření vertikálních propojení (krok 3). Je vygenerován nový řádek, s předchozím řádkem se již nepracuje.
6. Při vytváření posledního řádku se již nevytváří jeho vertikální propojení, jen se provede krok propojení sousedících buněk. Řádek se tedy prochází jen dvakrát.

Pseudokód popisující algoritmus vypadá potom následovně [7].:

```

1  foreach (řádek v bludišti)
2      foreach (buňka v řádku)
3          if (horní zeď buňky není) then
4              označ buňku číslem horní buňky
5          end else
6              označ buňku unikátním číslem
7          end
8      end
9
10     foreach (buňka v řádku)
11         if (soused vpravo má jiné číslo)
12         and random.(má se zbourat zeď) then
13             odstraň pravou zeď
14             sluč čísla buňek
15         end
16     end
17
18     foreach (číslo v řádku)
19         náhodně vyber minimálně jednu buňku se
20         stejným číslem a odstraň pro ni spodní zeď
21     end

```



Obr. 21 – Vytvoření bludiště Ellerovým algoritmem

### 7.2.5 Algoritmus půlení intervalů

Princip algoritmu je jednoduchý, plocha bludiště se rozdělí vertikální nebo horizontální hranou na dvě nové plochy. V nově vytvořené hraně se vytvoří jedna cesta (propojení) tak, aby byly plochy propojeny. Toto rozdělení se pak rekurzivně opakuje na nově vzniklé plochy

a na jejich další rozdělení. Dělení ploch končí, jakmile výška nebo šířka dělené plochy je rovna požadované výšce nebo šířce buňky bludiště.

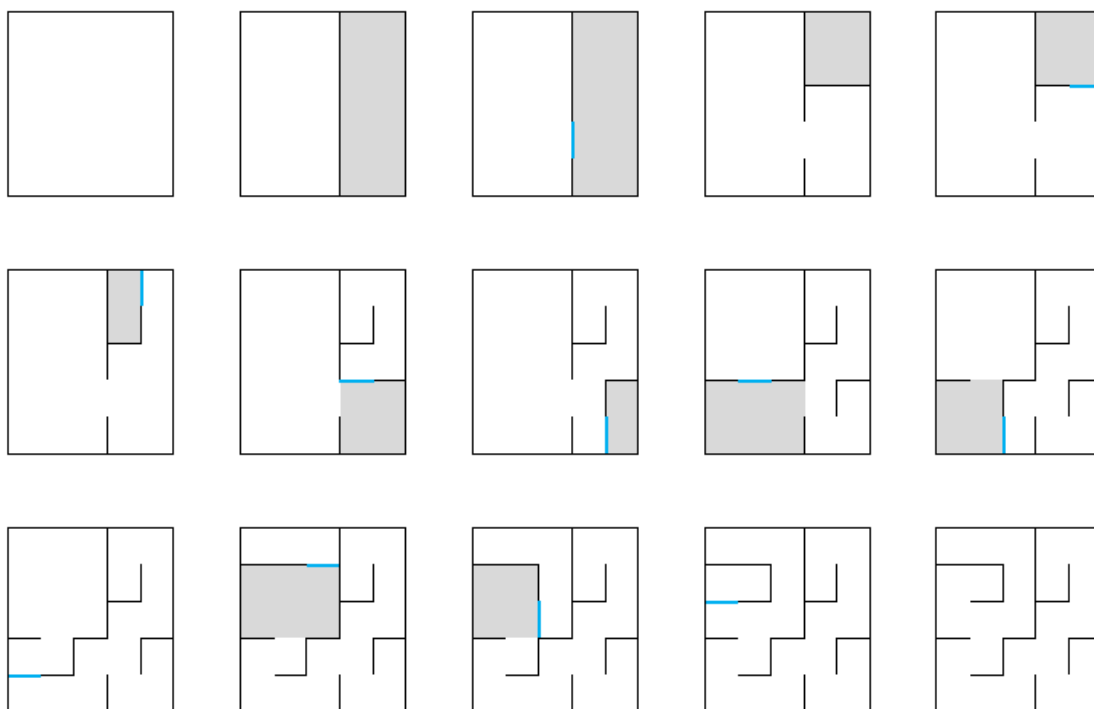
Rekurze může být v algoritmu nahrazena pomocí zásobníku, na který se ukládají objekty reprezentující plochy, které je ještě potřeba rozdělit. Pokud již není možné plochu rozdělit, je tato odebrána ze zásobníku. Algoritmus se ukončí, jakmile je zásobník prázdný.

V rámci implementace tohoto algoritmu byla použita varianta s rekurzí, kdy pseudokód procedury dělení a procedur samotného dělení vypadá následovně [7].:

```
1  procedure divideV(levá, pravá, horní, spodní)
2      šířka = pravá - levá
3      výška = spodní - horní
4
5      if (šířka > 1 and výška > 1) then
6          if (šířka > výška) then
7              divideV(levá, pravá, horní, spodní)
8          else if (výška > šířka) then
9              divideH(levá, pravá, horní, spodní)
10         else if (šířka = výška) then
11             Random(divideV or divideH)
12         end
13     end
14 end
15
16 procedure divideV(levá, pravá, horní, spodní)
17     místo = náhodně vyber kde se bude dělit
18     vytvoř zdi (horní ... spodní)
19     náhodně vytvoř cestu v (horní ... spodní)
20
21     rdivision(levá, místo, horní, spodní)
22     rdivision(místo, pravá, horní, spodní)
23 end
24
25 procedure divideH(levá, pravá, horní, spodní)
26     místo = náhodně vyber kde se bude dělit
27     vytvoř zdi (levá ... pravá)
28     náhodně vytvoř cestu v (levá ... pravá)
29
30     rdivision(levá, pravá, horní, místo)
31     rdivision(levá, pravá, místo, spodní)
32 end
```

Pokud jsou pozice hrany rozdělující bludiště voleny náhodně se stejnou pravděpodobností, potom vygenerovaná bludiště mají tendenci obsahovat množství cest délky jedna, nezanedbatelné množství dlouhých rovných cest a málo křižovatek. Možností jak tento problém

odstranit je několik. Například změnou pravděpodobnosti výběru směru rozdělení plochy, tedy bude-li plocha rozdělena horizontálně či vertikálně. Druhou možností je přednostně vybírat pro pozici hrany plochu s delší hranou. Také lze ovlivnit poměr rozdělení plochy zvýšením pravděpodobnosti výběru pozice směrem do středu plochy. Algoritmus je znázorněn na obrázku Obr. 22.



Obr. 22 – Vytvoření bludiště algoritmem půlení intervalů

### 7.3 Algoritmy řešící bludiště

Implementace řešících algoritmů vycházejí z předpokladu, že jsou předem definované počáteční (startovací) a konečná (cílová) buňky do pozic  $(0, 0)$  respektive  $(W-1, H-1)$ , kde  $W$  je šířka a  $H$  je výška bludiště. Toto umístění bylo zvoleno proto, aby byla řešícími algoritmy prohledávána co největší část bludiště, což by se při náhodném umístění těchto buněk nedalo zajistit.

#### 7.3.1 Algoritmus sledování zdi

Princip algoritmus je podobný k v kapitole 7.2.1 uvedenému algoritmu prohledávání do hloubky, jen o směru dalšího postupu prohledávání grafu se nerozhoduje náhodně, ale směr postupu je pevně daný. V implementaci toho algoritmu bylo zvoleno pravidlo pravé ruky, tedy na každé křižovatce se zatáčí doprava.

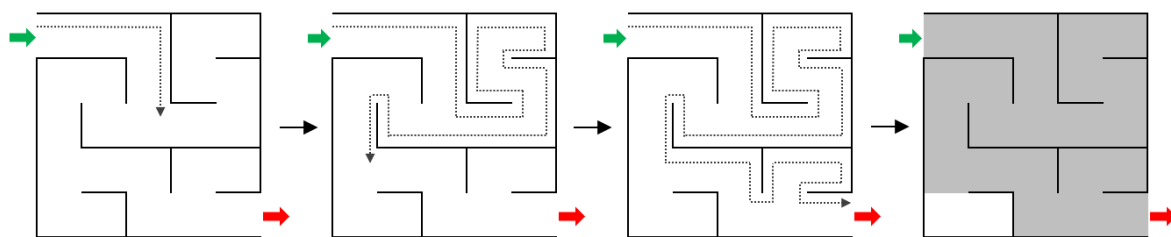
Pseudokód tohoto algoritmu potom vypadá následovně:

```

1   vyber počáteční buňku a výchozí směr pohybu
2   while (buňka != cíl) do
3       if (soused vpravo je dostupný) then
4           buňka = soused
5       else if (soused rovně je dostupný) then
6           buňka = soused
7       else if (soused vlevo je dostupný) then
8           buňka = soused
9       else
10          buňka = předchozí buňka
11  end

```

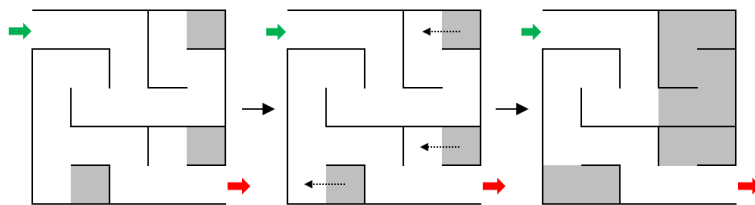
Výstupem tohoto algoritmu není trasa od vstupního do výstupního bodu bludiště, ale jen zjištění zdali taková trasa existuje. Konkrétní trasu lze na výstup algoritmu dostat například průběžným ukládáním každé navštívené buňky. Také pokud by se i eliminovaly buňky navštívené více než jednou, nalezená trasa bude zároveň trasou nejkratší. Princip algoritmu je znázorněn na obrázku Obr. 23.



Obr. 23 – Řešení bludiště algoritmem sledování zdi (pravidlo levé ruky)

### 7.3.2 Algoritmus vyplňování slepých konců

Algoritmus není opět složitý. V první cyklu se projdou všechny buňky bludiště a označí se slepé konce. Tedy ty buňky, ze kterých se dá odejít jen jedním směrem. Následně se označí každá buňka trasy vedoucí od slepého konce k první křižovatce. Tím se označí všechny buňky bludiště až na hledanou trasu.



Obr. 24 – Řešení bludiště algoritmem vyplňování slepých konců

Algoritmus je možno popsat následujícím pseudokódem (viz také jeho znázornění na obrázku Obr. 24):

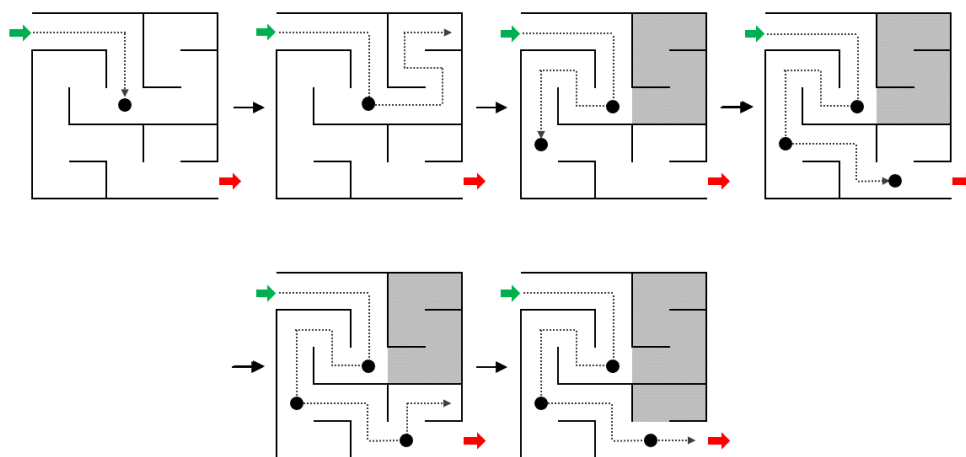
```

1   vytvoř frontu Q = 0
2
3   foreach (buňka v bludišti)
4       if (buňka B má tři zdi) then
5           přidej B do Q
6       end
7   end
8
9   while (Q není prázdná) do
10      vyber B z Q
11      označ B jako VISITED
12      while (soused B není křižovatka) do
13          označ B jako VISITED
14          načti souseda B
15      end
16  end
17
18  řešení bludiště = buňky not VISITED

```

### 7.3.3 Trémauxův algoritmus

Algoritmus prochází bludiště, a na zásobník jsou ukládány buňky cesty (vrcholy grafu) od vstupního bodu bludiště. Na křižovatce se volí směr dalšího postupu náhodně po hraně, která ještě nebyla navštívena. V případě, že cesta končí slepým koncem, algoritmus se vrací zpět k poslední křižovatce a odebírá buňky ze zásobníku. Jakmile se vrátí zpět na křižovatku, označí si slepou cestu jako nesprávnou a opět pokračuje náhodným směrem, dokud nejsou všechny možné směry křižovatky vyčerpány.



Obr. 25 – Řešení bludiště Trémauxovým algoritmem

Pro nalezení řešení není nutné projít celé bludiště, ukončovací podmínkou algoritmu je nalezení cíle, kdy v zásobníku zůstává trasa řešící bludiště. Naopak pokud se zásobník vyprázdní před dosažením cíle, znamená to, že výstupní buňka bludiště neexistuje. Algoritmus je znázorněn na obrázku Obr. 25.

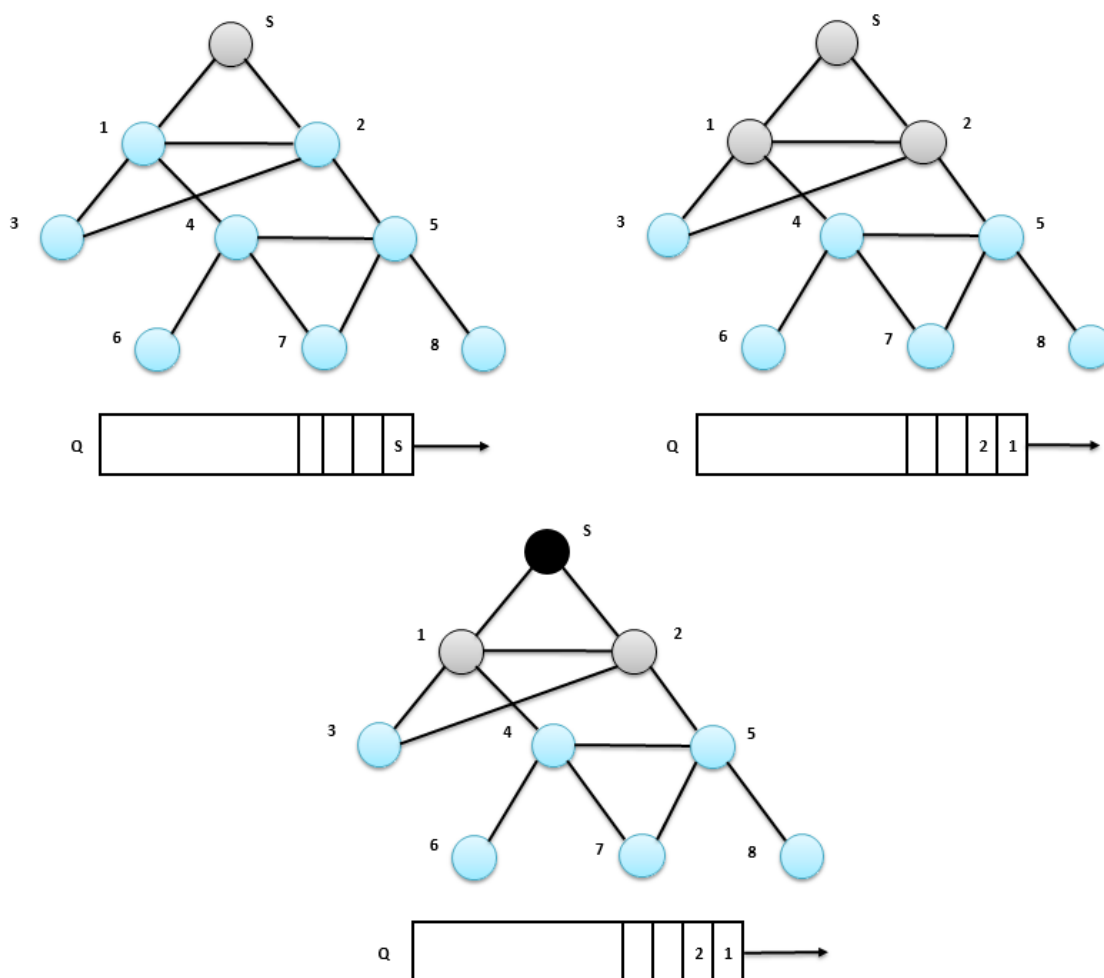
### 7.3.4 Algoritmus hledání nejkratších cest

Jak již bylo uvedeno v kapitole 4.4, algoritmus prochází celý graf ve vlnách, a to následovně (viz také znázornění na obrázku Obr. 26):

1. Při inicializaci se všem vrcholům grafu nastaví příznak *READY*
2. Vstupní vrchol se umístí do fronty a jeho příznak se změní na *WAITING*
3. Dokud není fronta prázdná, tak se z ní vybírají vrcholy. S vybraným vrcholem se provedou dvě operace: nastaví se mu příznak *PROCESSED* a do fronty jsou vloženy všechny s ním sousedící vrcholy, které mají příznak *READY* a změní se jim příznak na *WAITING*.

Pseudokód implementace algoritmu potom vypadá následovně [5]:

```
1   označ všechny vrcholy grafu G jako READY
2   procedure BFS(graf G, vrchol V)
3       vytvoř frontu Q
4       přidej V do Q
5       označ V jako WAITING
6       while (Q není prázdná) do
7           odeber V z fronty Q
8           označ V jako PROCESSED
9           if (V == cíl) then
10              exit procedure
11          else
12              foreach soused N vrcholu V
13                  if (N == READY) then
14                      označ N jako WAITING
15                      přidej N do Q
16                  end
17              end
18              označ V jako PROCESSED
19          end
20      end
21  end
```



Obr. 26 – Znázornění průchodu grafem do šířky

### 7.3.5 Dijkstrův algoritmus

Dijkstrův algoritmus si uchovává všechny vrcholy v prioritní frontě řazené dle vzdálenosti od zdroje, v první iteraci má pouze zdroj vzdálenost 0 a všechny ostatní vrcholy nekonečno. Algoritmus v každém svém kroku vybere z fronty vrchol s nejvyšší prioritou (tedy nejmenší vzdáleností od již zpracované části) a zařadí jej mezi zpracované vrcholy. Poté projde všechny jeho dosud nezpracované potomky a přidá je do fronty, pokud tam již nejsou obsaženi, přičemž ověří, jestli nejsou blíže zdroji, než byli před zařazením právě vybraného vrcholu mezi zpracované. To znamená, že pro všechny potomky ověřuje:

$$vzdálenost_{zpracováváný} + cena_{hrany_{zpracováváný, potomek}} < vzdálenost_{potomek}$$

Pokud výše uvedená nerovnost platí, tak danému potomkovi nastaví novou vzdálenost a označí za jeho předka zpracováváný vrchol. Po průchodu přes všechny potomky algoritmus



vybere z fronty vrchol s nejvyšší prioritou a celý krok opakuje. Princip je znázorněn na obrázku Obr. 27.

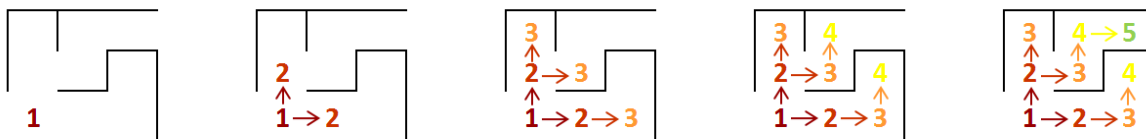
Algoritmus končí v okamžiku, kdy jsou zpracovány všechny vrcholy, a prioritní fronta je prázdná. Pro určení nejkratší trasy potom stačí zpětně projít předky všech vybraných vrcholů.

Pseudokód tohoto algoritmu vypadá následovně [7].:

```

1   cena každého vrcholu grafu = nekonečno
2   cena počátečního vrcholu = 0
3
4   while (existují nezpracované vrcholy v grafu) do
5       vyber nezpracovaný vrchol s nejnižší cenou n
6       označ n jako zpracovaný
7       foreach (soused s vrcholu n)
8            $cena(s) = \min(cena(s), cena(n) + cena(n, s))$ 
9           označ všechny s jako zpracované
10      end
11  end

```



Obr. 27 – Průchod bludištěm s využitím Dijkstrova algoritmu

## 7.4 Implementace ACO

Při implementaci algoritmů optimalizace mravenčí kolonií pro řešení vygenerovaného bludiště bylo využito jednoduchého S-ACO algoritmu, jehož teoretický základ je položen v kapitole 5.2. V následujících podkapitolách bude popsána problematika nalezení nejkratší cesty v bludišti, způsob implementace algoritmu v této práci a změny, které byly provedeny oproti původnímu algoritmu.

### 7.4.1 Problém hledání nejkratší cesty

Na problematiku nalezení cesty v bludišti, lze nahlížet jako na optimalizační problém nalezení nejkratší cesty (v originále *Shortest Path Problem*, zkráceně *SPP*), v našem případě tedy tu nejkratší cestu ze všech možných cest mezi dvěma buňkami bludiště.

Obecný popis SPP je následující: mějme orientovaný graf  $G = (V, E)$ , kde množina  $V$  obsahuje uzly grafu  $G$  a množina  $E$  hrany grafu  $G$ . Každé hraně  $(i, j) \in E$  potom přiřadíme cenu  $a_{ij}$ . Alternativně může být tato cena označena jako délka  $a_{ij}$ . Pro délku cestu mezi uzly grafu  $a_{ij} = (n_1, n_2, \dots, n_k)$  platí vztah:

$$a_{ij} = \sum_{i=1}^{k-1} a_{n_i n_{i+1}}$$

Cestu  $a_{ij}$  je potom nazýváme nejkratší, když má nejmenší délku ze všech možných cest mezi počátečním a koncovým uzlem, kdy počáteční uzel je definován jako uzel  $s$  a koncový uzel jako uzel  $t$ .

Na problém nalezení nejkratší cesty potom může být nahlíženo jako na:

- Nalezení nejkratší cesty mezi dvěma uzly grafu.
- Nalezení nejkratší cesty je-li dán pouze počáteční uzel.
- Nelezení nejkratší cesty je-li dán pouze koncový uzel.
- Nalezení nejkratší cesty mezi všemi uzly.

V rámci této práce je potom řešením bludiště myšleno nalezení nejkratší cesty mezi dvěma uzly grafu, kdy, jak již bylo uvedeno v kapitole 2.2, má každá cesta mezi dvěma sousedními buňkami bludiště cenu (délku) rovnu 1.

#### 7.4.2 Pseudokód S-ACO

Použití algoritmu S-ACO lze potom zapsat následujícím pseudokódem:

```
1   nastav parametry, inicializuj feromonové stopy
2   while (nejsou splněny podmínky pro ukončení) do
3       mravenci vytvářejí řešení
4       aktualizují se feromonové stopy
6   end
```

#### 7.4.3 Třídy algoritmu

V implementaci algoritmu S-ACO jsou definovány následující třídy popisující graf reprezentující bludiště:

- třída **uzlu** grafu (class **Vertex**),
- třída **hrany** grafu tvořená dvojicí uzlů (class **Edge**),
- třída **cesty**, tedy posloupností hran, (class **Path**),
- a třída **světa**, popisující graf (class **World**).

Dále jsou definovány dvě třídy zajišťující chod samotného algoritmu:

- třída definující jednotlivé umělé **mravence** a jejich pohyb (class **Ant**),
- a třída organizující jednotlivé iterace **kolonie** (class **Colony**).

#### 7.4.4 Umělý mravenec

Pro implementaci optimalizačních algoritmů je třídou **Ant** vytvořen tzv. umělý mravenec. Jeho chování je inspirováno chováním skutečného mravence v mravenčí kolonii, ale z důvodu zlepšení výsledků konkrétních optimalizačních problémů má některé vlastnosti posíleny.

Následující vlastnosti má shodné se skutečnými mravenci:

- jednotlivci kooperují v rámci kolonie,
- nepřímo komunikují pomocí feromonů,
- po nalezení potravy (cíle) vytvářejí feromonové stopy,
- strategie jejich rozhodování je lokální,
- a jejich rozhodování se děje na základě pravděpodobnosti.

Oproti skutečným mravencům se ale umělý mravenec liší v tom, že:

- obor, ve kterém pracuje, je diskrétní,
- umělý mravenec má osobní paměť zaznamenávající doposud vykonané akce (vykonanou trasu),
- doba uložení feromonu je závislá na řešeném problému.

Umělí mravenci potom v algoritmu fungují jako stochastické konstruktivní procedury vytvářející řešení interaktivním přidáváním komponent do částečného řešení. Při výběru každé další komponenty zvažují heuristickou informaci o řešeném problému, která směřuje výpočet ke slibným řešením a zároveň zkušenosti získané všemi mravenci od začátku výpočtu. Ty jsou reprezentované právě feromonovými stopami, které se během výpočtu neustále adaptují. Stochastická složka zajistí vygenerování velkého počtu různých řešení.

#### 7.4.5 Inicializace algoritmu

Implementace algoritmu používá následující parametry:

- $m$  – počet mravenců v kolonii (parametr je nastavitelný v rozsahu 1 .. 500),

- $\rho$  – parametr zajišťující rychlost vypařování feromonu v rámci jedné iterace (parametr je nastavitelný v rozsahu 0,0001 .. 1),
- $\tau_0$  – výchozí úroveň feromonu na hranách grafu (výchozí hodnota tohoto parametru je  $\tau_0 = 0$ ),
- $\Delta\tau$  – množství přidaného feromonu k množství feromonu aktuálně se nacházející na uzlu grafu (výchozí hodnota tohoto parametru je  $\Delta\tau = 0,5$ ),
- $\tau_{min}, \tau_{max}$  – minimální a maximální akceptovatelná úroveň feromonu na hranách grafu, určující parametry pro vizualizaci ukládání a vypařování feromonů (výchozí hodnota pro tyto parametry je  $\tau_{min} = 0$  a  $\tau_{max} = 1$ ),
- $i$  – počet iterací, ve kterých budou mravenci procházet bludiště (parametr je nastavitelný v rozsahu 1 .. 10000),
- $T$  – počet nalezených cest, které mají délku stejnou nebo větší než doposud nejlepší nalezená cesta (parametr je nastavitelný v rozsahu 1 .. 500).

Počet mravenců  $m$  značně ovlivňuje přesnost řešení získaného algoritmem. Na druhou stranu ale negativně ovlivní výpočetní čas algoritmu. Proto je nutné, aby tento parametr odpovídal předmětu řešení. Obecně lze říci, že čím větší bludiště je řešeno, tím více mravenců je třeba použít.

Parametr  $\rho$  rozhoduje o rychlosti vypařování feromonu, tedy o rychlosti snižování úrovně feromonu na jednotlivých hranách grafu. Čím rychleji je feromon vypařován, tím větší prostor grafu (větší plochu bludiště) můžou mravenci prozkoumat a tím pádem objevit nová řešení problému. Zatímco čím pomaleji se feromon vypařuje, tím více mravenci tíhnou k použití již nalezených řešení problému. Čím menší hodnota tohoto parametru je v implementaci algoritmu nastavena, tím pomaleji se feromony vypařují.

Počet iterací  $i$  je první z ukončujících podmínek algoritmu. Jedná se o počet nezávislých životních cyklů kolonie, ve kterých algoritmus provádí výpočetní úkony. Po uplynutí  $i$  životních cyklů je nezávisle na nalezeném nejlepším řešení běh algoritmu ukončen.

Druhým ukončujícím parametrem  $T$ , je počet nalezených cest, které mají délku stejnou nebo větší než doposud nejlepší nalezená cesta. Pokud mravenci naleznou souvisle po sobě  $T$  delších cest než je doposud nejlepší nalezená cesta, algoritmus se ukončí.

Oba terminační parametry ovlivňují především dobu, po kterou bude algoritmus pracovat a tím i nepřímo ovlivňují kvalitu nalezeného řešení.

#### 7.4.6 Metoda nalezení cesty

Jedním z důležitých prvků libovolného algoritmu, který je založen na použití algoritmů mravenčí kolonie, je způsob vytváření cesty z počátečního do koncového uzlu grafu. Zjednodušeně řečeno pravidla, kterými se jednotlivý mravenci řídí při svém putování bludištěm.

V implementaci S-ACO algoritmu byl použit způsob hledání cesty každým mravencem zvlášť. Tzn., že každý mravenec se při hledání cesty bludištěm (grafem) posunuje zvlášť v rámci jedné iterace s ohledem na definovaná pravidla a na konci každé iterace se uskuteční vypařování feromonu nad celým grafem najednou.

#### 7.4.7 Výběr následujícího uzlu

Způsob výběru následujícího uzlu, kterým se bude mravenec pohybovat je v případě použití algoritmu S-ACO dán jednoduchým pravidlem uvedeným v kapitole 5.2. V implementaci algoritmu byl výpočet pravděpodobnosti  $p_{ij}^k$  výběru sousedního uzlu  $j$  mravencem  $k$  nacházejícího se na uzlu  $i$  oproti původnímu pravidlu zjednodušen tak, že tato pravděpodobnost je dána jen množstvím feromonu nacházejícím se na uzlu  $e \in N_i$  (kde  $N_i$  jsou všechny sousední uzly k uzlu  $i$ ), tedy:

$$p_{ij}^k = \begin{cases} \tau_{ij} & \dots \text{ když } j \in N_i \\ 0 & \dots \text{ když } j \notin N_i \end{cases}$$

Využití toho jednoduchého pravidla nicméně nebylo dostačující. Pro zvýšení pravděpodobnosti výběru doposud nenavštívené buňky a tím pádem nalezení nové trasy, do pravidla pro výběr následujícího uzlu byly implementovány následující podmínky:

- pravděpodobnost výběru konkrétní buňky zvyšuje fakt, že tato ještě nebyla mravencem navštívena (tzn., že se kontroluje, zdali se v lokální paměti mravence o jím navštívených uzlech možný sousední uzel nenachází a pokud ano, je tento uzel vynechán),
- pokud se mravenec dostane na křižovatku, kde všechny sousední buňky mají stejnou pravděpodobnost postupu a zatím nebyly mravencem navštíveny, vybírá se následující buňka náhodně,
- a pokud se mravenec dostane na křižovatku, kde všechny sousední buňky mají stejnou pravděpodobnost postupu a již byly mravencem navštíveny, upřednostní se výběr té buňky, která byla navštívena nejdříve (byla uložena do seznamu již navštívených buněk dříve).



Princip této varianty není složitý: jakmile mravenec nalezne cílovou buňku, vezme se jeho optimalizovaná trasa (tzn., eliminují se v ní zacyklení) a po ní se mravenec vrací zpět ke startovací buňce, přičemž se v každém jeho kroku aktualizují hodnoty feromonů na uzlech o  $\Delta\tau^k$ , tedy přidávaná intenzita feromonu zohledňuje, který mravenec danou cestu našel. Způsob eliminace těchto opakujících se uzlů je nastíněn na obrázku Obr. 28.

#### 7.4.9 Vypařování feromonů

Mechanismus vypařování feromonů je sám o sobě také jednoduchý a intuitivní, hodnota feromonů na každém uzlu se násobí vhodně zvoleným parametrem  $\rho$ . Funkci lze popsat následujícím pseudokódem:

```
1   foreach (buňka v bludišti)
2        $\tau_{ij} \leftarrow \tau_{ij}(1 - \rho)$ 
3   end
```

Důležitým faktorem vypařování feromonů je stanovení momentu, kdy se vypařování provede. Má se za to, že nejvhodnější okamžik pro vypařování feromonů je na konci aktuální iterace, tedy v okamžiku, kdy všechny ostatní akce již byly dokončeny. Tak je tomu i v implementaci S-ACO algoritmu v rámci této práce.

#### 7.4.10 Ukončení algoritmu

S ohledem na to, že uvedený optimalizační algoritmus negarantuje stoprocentní nalezení optimálního řešení, je důležité a nutné nadefinovat podmínky, při jejichž splnění se algoritmus ukončí.

Obecně vzato, nejlepší podmínkou pro ukončení algoritmu je jeho konvergence, tedy že se charakteristiky nalezeného řešení všemi mravenci s časem blíží k požadovanému řešení, tzn. nejkratší cestě. Tato situace nastává, když feromony indikující řešení mají tak vysokou hodnotu, že každá další iterace nepřináší žádnou další změnu. Toto řešení bylo implementováno za pomoci již dříve zmíněného ukončovacího parametru  $T$ .

Druhým způsobem jak algoritmus ukončit je nastavení určitého počtu iterací, které algoritmus provede a pak se ukončí. Toto řešení je implementováno pomocí parametru  $i$ .

V praxi se ukázalo, že čím větší je prohledávané bludiště a větší počet mravenců, tím více iterací a také počtu nalezených cest se stejnou délkou je třeba nastavit.

## 8 POROVNÁNÍ ALGORITMŮ

Vytvořená počítačová aplikace pro generování bludišť a řešení základních úloh v bludištích umožňuje přehledné znázornění fungování principů jednotlivých algoritmů vytváření i řešení bludišť. Jsou nadefinovány čtyři velikosti bludišť (10x10, 15x15, 20x20 a 60x30 buněk). Bylo by teoreticky možné vytvářet a řešit i bludiště daleko větší, ale z praktického hlediska byla zvolena maximální velikost právě 60x30 buněk, která je ještě únosná z pohledu vizualizace tvorby.

Aplikace je vytvořena tak, že u každého implementovaného algoritmu, s výjimkou řešících BFS a Dijkstrova algoritmu, dokáže při podstatné změně v datové struktuře bludiště tuto změnu názorně vykreslit. Děje se tak za pomoci pozastavení komponenty zajišťující animaci appletu v rámci prostředí internetového prohlížeče. Toto pozastavení je definováno v milisekundách a je optimalizované pro každý algoritmus zvlášť, proto je nemožné přesně kvantifikovat jejich časovou náročnost v případě, že je jejich běh vizualizován. U dvou výše uvedených algoritmů je zobrazen až výsledný stav bludiště po provedení výpočtů.

Pro výpočet časové náročnosti je pak v aplikaci vytvořena metoda, která sleduje a následně vypisuje čas potřebný pro běh jednotlivých algoritmů. Jako počáteční časový moment se bere spuštění algoritmu (v aplikaci tlačítka pro zahájení vytváření nebo řešení bludiště, viz příloha P I) a koncový časový moment se bere poslední vykreslení datové struktury bludiště.

Aby bylo možné porovnat reálnou časovou náročnost implementovaných algoritmů (zjednodušeně rychlost vytváření nebo řešení bludiště), existuje v aplikaci možnost zrušit vykreslování jednotlivých změn v datové struktuře bludiště a vykreslit stav bludiště před a po vytvoření či vyřešení.

Jak se ale ukázalo v praxi, časová kvantifikace generování bludišť je i v případě největšího definovaného rozměru (tedy 60x30 buněk) bludiště v podstatě zbytečná. Všechny pět vytvářecích algoritmů generuje bludiště v řádech stovek milisekund. Přehled naměřených hodnot u jednotlivých algoritmů při generování bludiště je uveden v tabulce Tab. 4.

Metodika tohoto měření byla použita následující: vytvářené bylo bludiště největšího definovaného rozměru (60x30 buněk) a každým algoritmem se vytvořilo 10 bludišť a pro statistické porovnání byl použit medián. Aplikace byla pro měření spuštěna v Applet Vieweru.



Tab. 4 – Časová náročnost algoritmů vytvářejících bludiště

Řešící algoritmus	Čas vytváření [ms]
Prohledávání do hloubky	59
Primův algoritmus	324
Kruskalův algoritmus	506
Ellerův algoritmus	48
Půlení intervalů	26

U řešících algoritmů bylo nutné rozlišit, zdali je bludiště vytvořeno se zacyklením nebo ne. Tři algoritmy řešící bludiště (tedy algoritmus sledování zdí, algoritmus vyplňování slepých konců a Trémauxův algoritmus) totiž nejsou schopny bludiště obsahující cykly vůbec řešit, algoritmy končí v nekonečné smyčce. Zbylé tři řešící algoritmy (algoritmus hledání nejkratších cest, Dijkstrův algoritmus a S-ACO) tento typ bludiště řeší bez problémů. Přehled naměřených hodnot u jednotlivých algoritmů při řešení perfektního bludiště pro každý vytvářecí algoritmus je uveden v tabulce Tab. 5.

Metodika tohoto měření byla použita následující: vytvářené bylo perfektní bludiště největšího definovaného rozměru (60x30 buněk). Každým algoritmem se vytvořilo 5 bludišť a bylo hledáno jejich řešení. Pro algoritmus S-ACO byly po sérii pokusů stanoveny parametry na: počet mravenců 300, koeficient vypařování na 0,075, počet iterací 3000 a ukončovací podmínku 300. Pro statistické porovnání byl použit medián. Aplikace byla také spuštěna v Applet Vieweru.

Tab. 5 – Časová náročnost řešení bludiště (perfektní bludiště)

	Čas vytváření [ms]				
	DFS	Prim	Kruskal	Eller	Půlení
<b>Sledování zdí</b>	83	86	84	71	65
<b>Vyplňování slepých konců</b>	78	87	87	81	83
<b>Trémauxův algoritmus</b>	74	73	71	71	75
<b>BFS</b>	96	83	92	62	64
<b>Dijkstrův algoritmus</b>	105	94	95	65	70
<b>S-ACO</b>	12406	6585	3944	9633	17654

Z tabulky Tab. 5 vyplývá, že algoritmu S-ACO trvalo nejdéle řešení bludišť vytvořených algoritmem půlení intervalů. To je způsobeno tím, že při vytváření bludiště jeho plocha dělena na podplochy a v rozdělení je vytvořena jen jedna spojnice do další části bludiště a to zpomaluje umělým mravencům prohledávání bludiště.

V případě řešení bludiště obsahující cykly byly tedy testovány jen řešící algoritmy algoritmus hledání nejkratších cest, Dijkstrův algoritmus a S-ACO. Přehled naměřených hodnot u jednotlivých algoritmů při řešení bludiště obsahujícího cykly pro každý vytvářecí algoritmus a průměrná odchylka algoritmu S-ACO od nejkratší trasy (tedy nepřesnost nalezení trasy) je uveden v tabulce Tab. 6.

Metodika tohoto měření byla použita následující: vytvářené bylo bludiště největšího definovaného rozměru (60x30 buněk) obsahující cykly. Každým algoritmem se vytvořilo 5 bludišť a bylo 10x hledáno jejich řešení. Pro algoritmus S-ACO byly po sérii pokusů stanoveny parametry na: počet mravenců 400, koeficient vypařování na 0,095, počet iterací 7000 a ukončovací podmínku 500. Pro statistické porovnání byl použit aritmetický průměr. Aplikace byla také spuštěna v Applet Vieweru.

*Tab. 6 – Časová náročnost řešení bludiště (bludiště obsahující cykly)*

	Čas vytváření [ms]				
	DFS	Prim	Kruskal	Eller	Půlení
<b>BFS</b>	61	69	68	65	69
<b>Dijkstrův algoritmus</b>	66	76	69	68	65
<b>S-ACO</b>	12341	9749	10429	49681	12648
<b>Průměrná odchylka [%]</b>	15	7	10	10	6

Všechny tři algoritmy tedy jsou schopny řešit vytvořená bludiště obsahující cykly. Nejméně vhodným se ukázal algoritmus S-ACO, což je dáno složitostí prohledávaného grafu, a také tím, že algoritmus neobsahuje žádné heuristické informace o řešeném problému

## ZÁVĚR

Tato diplomová práce byla mojí první zkušeností s podobným druhem výzkumu, který se nakonec ukázal mnohem zajímavějším, než jsem čekal. Zejména proto, že bludiště obecně a jejich vytváření a řešení, je velmi rozmanité a zábavné studijní téma.

Hlavním cílem této práce bylo poskytnout ucelený přehled teoretických informací o bludištích a labyrintech a pomocí vybraného evolučního algoritmu nastínit řešení základních úloh v bludištích.

V teoretické části je po krátkém úvodu do problematiky bludišť uvedena kapitola obsahující jejich základní klasifikaci. Poté je popsána matematická teorie grafů potřebná pro řešení základních úloh v bludištích, kterými jsou vytváření bludiště a hledání jeho řešení. V další části se práce zabývá popisem nejznámějších tvořících a řešících algoritmů. Je uvedeno pět algoritmů vytvářejících bludiště a šest algoritmů hledajících v bludišti cestu. V závěru teoretické části práce je uvedena biologická inspirace, historický a teoretický základ pro vybraný evoluční algoritmus, kterým je Optimalizace mravenčí kolonií. V celé teoretické části je kladen důraz na obecné vysvětlení každé dílčí části.

Po důkladném prostudování teorie grafů a zvoleného evolučního algoritmu byla navržena aplikace v jazyku Java simulující jednotlivé tvořící a řešící algoritmy. Jejich podrobný popis včetně pseudokódů je pak uveden v praktické části této práce. V jejím závěru je potom uvedeno porovnání jednotlivých algoritmů.

Z hlediska předepsaného zadání diplomové práce proto považuji konkrétní cíle za splněné. V teoretické i praktické části práce je zde prostor pro jejich další rozšíření, například o nové vytvářející a řešící algoritmy, případně o rozšíření stávajících algoritmů pro tvorbu a řešení bludišť vyšších dimenzí či složitějších typů mozaiky.

Implementovaná aplikace je veřejně dostupná na webové adrese: [www.vecerka.net/maze](http://www.vecerka.net/maze).

## SEZNAM POUŽITÉ LITERATURY

- [1] DOOB, Penelope Reed. *The Idea of the Labyrinth: From Classical Antiquity through the Middle Ages*. New York: Cornell University Press, 1992. ISBN 0-80-148000-0.
- [2] FISHER, Adrian. *The Amazing Book of Mazes*. New York: Harry N. Abrams, 2006. ISBN 978-08-109-4311-7.
- [3] MATĚJKA, Petr. *Algoritmy pro generování a řešení bludišť*. Brno, 2012. Diplomová práce. Masarykova Univerzita, Fakulta Informatiky.
- [4] ŠPAŇHELOVÁ, Kateřina. *Labyrint*. Praha, 2006. Diplomová práce. Univerzita Karlova, Pedagogická fakulta, Katedra výtvarné výchovy.
- [5] ČERNÝ, Jakub. *Základní grafové algoritmy*. In: Základní grafové algoritmy [online]. 2010. Dostupné z: <http://kam.mff.cuni.cz/~kuba/ka/ka.pdf>
- [6] KOLÁŘ, Josef. *Teoretická informatika*. 2. vyd. Praha: Česká informatická společnost, 2004. ISBN 80-900853-8-5.
- [7] BUCK, Jamis. *Mazes for Programmers*. The Pragmatic Programmers LLC, 2015. ISBN 978-1-68050-055-4.
- [8] ZELINKA, I., Z. OPLATKOVÁ, M. ŠEDA, P. OŠMERA a F. VČELAŘ. *Evoluční výpočetní techniky, principy a aplikace*. Praha: BEN, 2008. ISBN 80-7300-218-3.
- [9] DORIGO, Marco. *Ant colony optimization and swarm intelligence: 4th international workshop, ANTS 2004, Brussels, Belgium, September 5 - 8, 2004 : proceedings*. Berlin: Springer, 2004, xii, 434 s. ISBN 3540226729.
- [10] DORIGO, Marco a Thomas STÜTZLE. *Ant Colony Optimization*. Cambridge: The MIT Press, 2004. ISBN 978-02-620-4219-2.
- [11] HEROUT, Pavel. *Učebnice jazyka Java*. České Budějovice: KOPP, 2000. ISBN 80-7232-115-3.
- [12] HEROUT, Pavel. *Java - grafické uživatelské prostředí a čeština*. České Budějovice: KOPP, 2001. ISBN 80-7232-150-1.

## SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ACO	Metoda optimalizace mravenčí kolonií (Ant Colony Optimization).
S-ACO	Zjednodušená (didaktická) varianta optimalizace mravenčí kolonií (Simple Ant Colony Optimization).
DFS	Grafový algoritmus prohledávání do hloubky (Deep-first Search).
LIFO	Způsob práce s obecnou datovou strukturou používanou pro dočasné ukládání dat, tzv. zásobníkem (Last In – First Out).
BFS	Grafový algoritmus prohledávání do šířky (Breadth-first Search).
FIFO	Způsob práce s obecnou datovou strukturou používanou pro dočasné ukládání dat, tzv. frontou (First In – First Out).
TSP	Obtížný optimalizační problém, matematicky vyjadřující a zobecňující úlohu nalezení nejkratší možné cesty procházející všemi zadanými body na mapě (Traveling Salesman Problem).
JVM	Sada počítačových programů a datových struktur, která využívá modul virtuálního stroje ke spuštění dalších počítačových programů a skriptů vytvořených v jazyce Java (Java Virtual Machine).
AWT	Nástroj pro tvorbu grafického uživatelského rozhraní v jazyce Java (Abstract Window Toolkit).
JRE	Rozhraní potřebné nejen pro spouštění aplikací vytvořených v jazyce Java (Java Runtime Environment).
HTML	Značkovací jazyk používaný pro tvorbu internetových stránek, které jsou propojeny hypertextovými odkazy (HyperText Markup Language).
SPP	Optimalizační problém, matematicky vyjadřující a zobecňující úlohu nalezení cesty mezi dvěma uzly grafu takovou, aby součet vah všech hran v ní obsažených byl minimální (Shortest Path Problem).

**SEZNAM OBRÁZKŮ**

Obr. 1 – Labyrint na ostrově Knóssos .....	8
Obr. 2 – Ukázka dvou dimenzionálního bludiště .....	12
Obr. 3 – Typy mozaiky .....	13
Obr. 4 – Typy směřování .....	14
Obr. 5 – Ukázka grafu.....	17
Obr. 6 – Izomorfismus .....	18
Obr. 7 – Typy grafů .....	19
Obr. 8 – Strom T .....	19
Obr. 9 – Příklady metrik .....	20
Obr. 10 – Klasická reprezentace bludiště grafem .....	20
Obr. 11 – Reprezentace bludiště grafem.....	21
Obr. 12 – Příklad zápisu grafu pomocí spojové reprezentace .....	21
Obr. 13 – Příklad zápisu grafu pomocí matice sousednosti.....	22
Obr. 14 – Znázornění zpracování grafu ve vlnách.....	29
Obr. 15 – Uspořádání „experimentu dvou mostů“ .....	32
Obr. 16 – Mravenci hledající optimální cestu v grafu .....	33
Obr. 17 – Příklad konstrukčního grafu .....	36
Obr. 18 – Vytvoření bludiště algoritmem prohledávání do hloubky .....	44
Obr. 19 – Vytvoření bludiště Primovým algoritmem .....	46
Obr. 20 – Vytvoření bludiště Kruskalovým algoritmem .....	48
Obr. 21 – Vytvoření bludiště Ellerovým algoritmem .....	49
Obr. 22 – Vytvoření bludiště algoritmem půlení intervalů.....	51
Obr. 23 – Řešení bludiště algoritmem sledování zdí .....	52
Obr. 24 – Řešení bludiště algoritmem vyplňování slepých konců .....	52
Obr. 25 – Řešení bludiště Trémauxovým algoritmem.....	53
Obr. 26 – Znázornění průchodu grafem do šířky.....	55
Obr. 27 – Průchod bludištěm s využitím Dijkstrova algoritmu .....	56
Obr. 28 – Princip odstranění zacyklení v cestě.....	61

**SEZNAM TABULEK**

Tab. 1 – Typy buněk vyskytujících se v bludišti .....	11
Tab. 2 – Složitost reprezentací.....	22
Tab. 3 – Neúplný seznam variant ACO algoritmů .....	33
Tab. 4 – Časová náročnost algoritmů vytvářejících bludiště.....	64
Tab. 5 – Časová náročnost řešení bludiště (perfektní bludiště).....	64
Tab. 6 – Časová náročnost řešení bludiště (bludiště obsahující cykly) .....	65

## SEZNAM PŘÍLOH

P I:     Stručný popis vytvořené aplikace



## PŘÍLOHA P I: STRUČNÝ POPIS VYTVOŘENÉ APLIKACE

The screenshot shows a web application interface for maze generation and solving. It features several control panels and a central maze grid.

- Top Left Panel:**
  - 1. "Vyberte velikost bludiště" (Select maze size): A dropdown menu showing "15 x 15 ... střední".
  - 2. "Vyberte algoritmus, kterým bude vytvořeno bludiště" (Select algorithm for maze generation): A dropdown menu showing "Prohledávání do hloubky".
  - 3. "Vyberte algoritmus, kterým bude bludiště vyřešeno" (Select algorithm for maze solving): A dropdown menu showing "Sledování zdí".
- Top Center Panel:**
  - 4. "RESET" button.
  - 5. "GENERUJ" button.
  - 6. "VYŘEŠ" button.
- Top Right Panel:**
  - 7. "Zobrazit animace?" (Show animation?) checkbox, which is checked.
  - 8. "Vytvořit zacyklení?" (Create cycles?) checkbox, which is unchecked.
  - 9. "Počet mravenců" (Number of ants): A text input field with "30".
  - "Koeficient vypařování" (Evaporation coefficient): A text input field with "0.1".
  - "Počet iterací" (Number of iterations): A text input field with "1000".
  - "Ukončovací podmínka" (Termination condition): A text input field with "100".
- Maze Grid:**
  - A 15x15 grid representing the maze. A green square at the top left indicates the start cell. A red square at the bottom right indicates the end cell.
  - 10. A circular callout with the number "10" points to the right side of the maze grid.
- Bottom Text:**
  - Generování bludiště: HOTOVÓ
  - Generující algoritmus: prohledávání do hloubky

Vytvořená počítačová aplikace pro generování bludišť a řešení základních úloh v bludištích umožňuje přehledné znázornění fungování principů jednotlivých algoritmů. Aplikace pracuje na základě jednoduchého workflow:

1. vygenerování bludiště zvolené velikosti zvoleným vytvářecím algoritmem,
2. a řešení bludiště (tzn. nalezení cesty mezi počáteční a cílovou buňkou) zvoleným řešícím algoritmem.

Aplikace obsahuje následující ovládací prvky:

1. Rozbalovací nabídka pro výběr velikosti bludiště.
2. Rozbalovací nabídka pro výběr algoritmu, kterým bude bludiště vytvořeno.
3. Rozbalovací nabídka pro výběr algoritmu, kterým bude bludiště řešeno.
4. Tlačítko pro zrušení vytvořeného či vyřešeného bludiště.
5. Tlačítko pro zahájení vytváření bludiště.
6. Tlačítko pro zahájení řešení bludiště.
7. Zaškrtnutí pole zobrazující / skrývající animaci fungování principů jednotlivých algoritmů.
8. Zaškrtnutí pole přidávající generujícím algoritmům vlastnost vytváření zacyklení.

9. Textová pole pro úpravu vstupních parametrů algoritmu S-ACO.
10. Vlastní plocha s animací znázorňující fungování principů jednotlivých algoritmů.

V případě, že velikost bludiště není 60x30 je plocha s animací pro řešící algoritmus S-ACO rozšířena o matici čísel reprezentující aktuální hodnoty feromonů v jednotlivých buňkách bludiště.