Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Hamiltonovské grafy

Diplomová práce

Autor: Radim Krátký
Studijní obor: AI-2

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Odborný konzultant: RNDr. Andrea Ševčíková
Budova J, místnost 93200

Hradec Králové Duben 2020

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury. Souhlasím se zapůjčováním práce.

V Hradci Králové dne 27. 4. 2020 Radim Krátký

Poděkování:

Děkuji všem, kteří se podíleli na vzniku této práce. Především děkuji prof. RNDr. PhDr. Antonínovi Slabému, CSc. za metodické vedení práce a RNDr. Andree Ševčíkové za cenné rady a připomínky při zpracování diplomové práce.

Anotace

Jezdcova procházka, problém kterým se zabývali různí matematici už v minulých stoletích, je populární úlohou rekreační matematiky. Patří mezi příklady obecného a známého NP-úplného problému nalezení hamiltonovské kružnice, resp. cesty. Tato diplomová práce se na jezdcovu procházku zaměřuje. Práce je rozdělena na dvě části. Část teoretickou, která se věnuje hamiltonovským grafům, a na část praktickou, jejímž cílem bylo vytvoření desktopové aplikace CheGra. Jsou popsány algoritmy řešení jezdcovy procházky, včetně jejich testování a porovnání k usnesení závěru, jak výrazný rozdíl je při hledání řešení mezi těmito algoritmy.

Klíčová slova: Hamiltonovské grafy, Jezdcova procházka, Neuronová síť, Backtracking, Warnsdorffův algoritmus

Annotation

Title: Hamiltonian graphs

The Knight's tour, a problem that has been dealt with by various mathematicians in past centuries, is a popular task of recreational mathematics. These include examples of the general and well-known NP-complete problem of finding a Hamiltonian circle, respectively path. This diploma thesis focuses on the Knight's tour. The work is divided into two parts. The theoretical part, which deals with Hamiltonian graphs, and the practical part, which aimed to create a desktop application CheGra. Algorithms for solving the Knight's tour are described, including their testing and comparison to resolve the conclusion of how significant is the difference between these algorithms in finding a solution.

Keywords: Hamiltonian graph, Knight's tour, Neural network, Backtracking, Warnsdorff's algorithm

Obsah

1 Přehled použitého značení 7
2 Úvod 8
3 Cíl práce 12
4 Metodika zpracování 13
5 Teoretický rámec diplomové práce 14
5.1 Hamiltonovský graf 14
Postačující podmínky 15
Nutná podmínka 17
Nutná a postačující podmínka 17
5.2 Hledání hamiltonovské kružnice 18
5.2.1 Obecný algoritmus 19
5.2.2 Heuristické algoritmy 21
5.3 Aplikace hamiltonovské kružnice a cesty 23
5.3.1 Problém obchodního cestujícího 24
5.3.2 Jezdcova procházka 24
6 Aplikace CheGra 35
6.1 Obecně o aplikaci 35
6.2 Hlavní nabídka 36
6.3 Lišty aplikace 37
6.3.1 Horní lišta 37
6.3.2 Dolní lišta 38
6.4 Popis problému 38
6.5 Hra – Najdi procházku 39
6.6 Simulace 41
6.7 Hra – Krycí hra 41
6.8 Hra – Hamilton 42

6.9 Generování řešení ... 44

6.10 Generování neuronovou sítí ... 46

7 Implementace ... 48

7.1 Okno aplikace ... 48

7.2 Grafické objekty ... 49

7.3 Hlavní menu ... 50

7.4 Informační cedulka (nápověda) ... 50

7.5 Implementace – Popis problému ... 51

7.6 Implementace – Najdi procházku ... 52

7.7 Implementace – Simulace ... 53

7.8 Implementace – Krycí hra ... 53

7.9 Implementace – Hra Hamilton ... 53

7.10 Implementace – generování řešení ... 54

7.11 Implementace neuronové sítě ... 55

8 Implementace algoritmů ... 58

8.1 Backtracking ... 58

8.2 Warnsdorffův algoritmus ... 60

8.3 Hopfieldova neuronová síť ... 61

9 Testování aplikace ... 63

10 Shrnutí výsledků ... 67

11 Závěry a doporučení ... 68

12 Seznam použité literatury ... 69

1 Přehled použitého značení

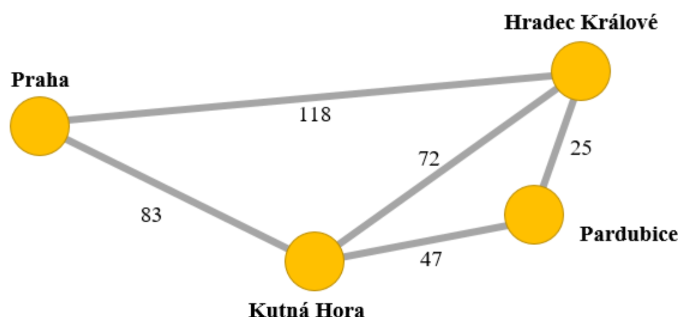
$A, B, C, \ldots$	prvky množiny vrcholů
$a, b, c, \ldots$	prvky množiny hran
$\{A, B\}$	neorientovaná hrana mezi vrcholy A a B
$A - B$	jiné zadání neorientované hrany
(A, B)	orientovaná hrana z vrcholu A do vrcholu B
$X \subseteq Y$	množina X je podmnožinou množiny Y
$X \cup Y$	sjednocení množin X a Y
$x \in X$	prvek x je prvkem množiny X
$deg(X)$	počet hran obsahujících vrchol X
K_n	úplný graf na n vrcholech

2 Úvod

Pod pojmem graf si většina lidí představí sloupcové grafy používané u statistik nebo grafy znázorňující průběh funkcí.

Grafy z oblasti teorie grafů si lze představit jako zjednodušení reálného světa, které se znázorňuje pomocí bodů, jenž reprezentují reálné objekty a čar představující vztahy mezi nimi. Body se nazývají vrcholy grafů a čáry hrany grafu.

Aniž by si to člověk uvědomoval, setkává se s teorií grafů poměrně často. Například v podobě navigace při hledání nejkratší cesty do cíle mezi vybranými městy, kde města jsou reprezentovány vrcholy a silnice hranami, je použit Dijkstrův algoritmus na hledání nejkratších cest v nezáporně ohodnoceném grafu, viz Obrázek 1.



Obrázek 1 - Příklad grafu v praxi

Nejedná se o jediný způsob použití, grafy mohou znázorňovat prakticky cokoli. Typickým příkladem aplikace teorie grafů je řešení infrastruktury měst například pro řízení dopravy a její optimalizace, kde vrcholy grafů reprezentují křižovatky a hrany silnice, viz Obrázek 2. [1]

Obdobně je to tak i v herním průmyslu, který je dnes velmi rozšířený. V podstatě každá hra s otevřeným světem využívá těchto grafů.

Obrázek 2 - Využití grafů v městské infrastruktuře (zdroj: [2])

Ať už při pohybu samotného hráče, tak i při pohybu NPC[1]. Díky tomu se každá postava ve hře dokáže přemístit z jednoho místa na druhé.

Grafy mají také využití v programování, kde jsou algoritmy často zapsány pomocí vývojových diagramů nebo při znázornění relací v databázi. Velké využití má i v rekreační matematice při řešení různých logických rébusů. [1]

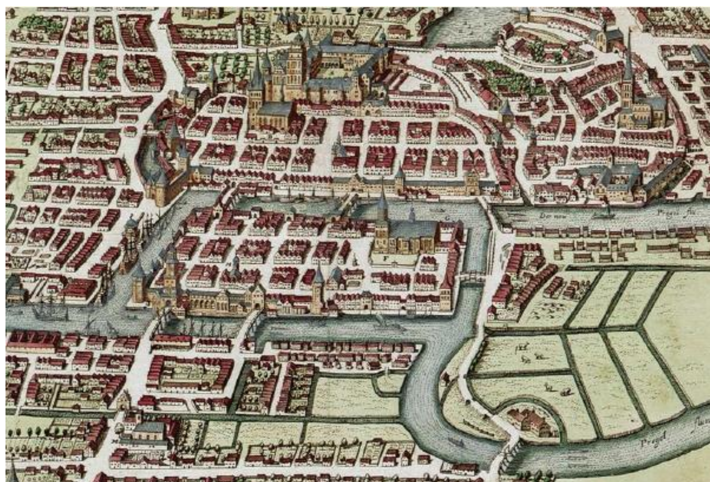
Teorie grafů, která je oborem diskrétní matematiky a která zkoumá vlastnosti grafů, je velmi mladá matematická disciplína. Její kořeny sice nacházíme v 18. a 19. století, ale teprve v roce 1936 vyšla první kniha, která byla celá věnována teorii grafů. Rozvoj této matematické disciplíny nastal v druhé polovině 20. století, kdy výsledky teorie grafů našly uplatnění především ve fyzice, elektrotechnice, chemii či ekonomice. [3]

Počátky grafů můžeme spatřovat již ve starověkém Egyptě a Římě v kontextu her nebo ve formě rodokmenu jako rodinného stromu. Nicméně za opravdový počátek teorie grafů se považuje rok 1736, kdy švýcarský matematik a fyzik Leonhard Euler (1707 – 1783) řešil úlohu **Sedmi mostů města Königsbergu** (též Královec, viz obrázek 3). [4]

Zadání znělo, zda je možné přejít přes všechny mosty právě jednou a vrátit se zpět do výchozího místa. Euler si úlohu představil tak, že vrcholy označovaly břehy

[1] NPC (non-playable character) – postava neovládaná hráčem nýbrž počítačem

a hrany grafu jako mosty, které břehy spojují. Nicméně nakonec dokázal, že úloha nemá řešení. [5]



Obrázek 3 – Historická mapa Královce (zdroj: [6])

„Předmět Diskrétní matematika (dále DIMA) vyučovaný na Fakultě informatického managementu Univerzity Hradec Králové (dále FIM UHK) zahrnuje především oblast teorie grafů. Teorie grafů nám poskytuje elegantní nástroj k popisu různých situací nebo problému ze skutečného života [7]. Také často poskytuje i návod na jejich řešení. Předmět DIMA je pro studenty přínosný do následujícího života, a proto je důležité, aby chápali témata a příslušné procesy vyučující v daném předmětu.“ [8]

Výuka na školách základních, středních i vysokých, se v současné době neobejde bez počítačů, interaktivních tabulí a přístupu k internetu. Pedagogové i žáci využívají všech dostupných moderních technologií zlepšujících kvalitu a srozumitelnost výuky. Vytvářejí svoje vlastní prezentace, dokážou třídit informace a používat je při svojí práci. Školství je obor, který dokáže využívat pokrok velmi účinně.

S cílem podpořit efektivitu výuky předmětu DIMA byly na FIM UHK vyvinuté specifické podpůrné multimediální nástroje, např. GrAlg [9] a ADIMA [10], které byly vytvořené pro podporu výuky grafových algoritmů, GraPro [11] pro podporu výuky matematických důkazů v teorii grafů a GraphScore [12], vytvořený autorem v rámci bakalářské práce pro podporu tématu skóre grafu. Tyto aplikace jsou

používané jako doplněk k existujícímu výkladu na přednáškách, k zlepšení představivosti a celkovému pochopení náročných témat. Vizualizace pomocí multimediálních nástrojů může zlepšit schopnost soustředit se na konkrétní cíl a na jeho dosažení, a proto jsou navrhované a vyvíjené nové aplikace pro různé oblasti teorie grafů vyučované v předmětu DIMA.

Tato diplomová práce má za cíl vytvoření jedné takové podpory, a to výuky hamiltonovských grafů. Skládá se z teoretické a praktické části. V teoretické části bude čtenář zasvěcen do problematiky hamiltonovských grafů a s nimi spojené NP - úplné problémy. Práce je zaměřená na jeden z těchto problémů - jezdcova procházka. Čtenáři je vysvětlen samotný problém i jeho řešící algoritmy. V praktické části se práce věnuje především samotné podpůrné aplikaci výuky hamiltonovských grafů, nazvané CheGra, konkrétně popisem jednotlivých her, funkcí, nabízených možností a zajímavých algoritmů, které byly při vývoji použity. V závěru praktické části je testování aplikace, kde autor porovnává efektivnost algoritmů řešících jezdcovu procházku na různých velikostech šachovnice. Nedílnou součástí práce jsou také přílohy obsahující seznamy obrázků, tabulek, algoritmů nacházející se v textu práce a také tam lze najít některé zajímavé kompletní algoritmy.

3 Cíl práce

Práce je zaměřena na matematickou oblast teorie grafů, konkrétně na hamiltonovské grafy. Cílem bylo vytvoření desktopové aplikace obsahující různé hry zasazené do této matematické oblasti, která by sloužila jako podpora výuky předmětů DIMA a DMO (Diskrétní metody a optimalizace) na FIM UHK a jako účelová aktivita na různých akcích fakulty organizovaných pro středoškolské studenty nebo veřejnost, např. Den Pí, Noc vědců. Součástí práce je teoretická studie hamiltonovských grafů, jejich problematiky nalezení řešení, postup implementace vyvíjené aplikace CheGra, její popis a návod k použití.

Veškeré hry obsažené v aplikaci byly navrženy samotným autorem a následně schváleny vedoucím a konzultantem diplomové práce.

4 Metodika zpracování

Vytvářená aplikace byla sestavená s použitím pouze vlastních znalostí. Veškerá teoretická východiska byla získána studiem zdrojů z referencích uvedených na konci práce. Bylo dbáno především na sestavení vlastního algoritmu v jazyce JAVA pouze se znalostmi z uvedených zdrojů. Nebylo využito žádného kopírování kódů z internetu.

Na základě znalostí programátora byl pro tento úkol vybrán jazyk JAVA (více v [13]). Technologie byly zvoleny na základě zkušeností autora - programátora. Autor práce vypracoval už několik projektů s využitím stejných technologií, v rámci předmětu Počítačová grafika a také v rámci předešlé bakalářské práci na UHK, ve které implementoval program GraphScore, který slouží pro podporu výuky v oblasti skóre grafu [12].

Pokud není uveden zdroj v popisku obrázku, jedná se o vlastnoručně vytvořená znázornění pro účely této práce a k snazšímu pochopení problematiky. K vytvoření obrázků autor použil pouze program CheGra a internetový online editor obrázků PIXLR [14].

5 Teoretický rámec diplomové práce

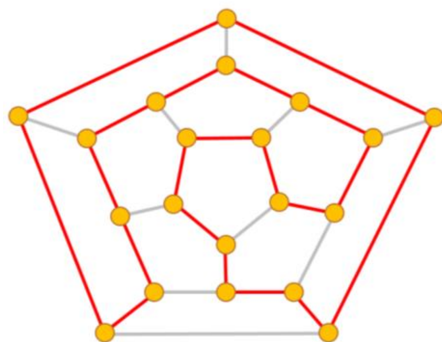
Základem teoretického rámce této diplomové práce jsou hamiltonovské grafy, algoritmy pro hledání hamiltonovských kružnic a matematická úloha jezdcova procházka. Primární koncepty používané v následujícím textu při definování základních pojmů a formulování vět zásadních pro tuto práci by neměly být pro čtenáře neznámé, lze je najít v [15], [16].

5.1 Hamiltonovský graf

Hamiltonovský graf je takový graf, ve kterém existuje uzavřená cesta procházející všemi vrcholy grafu. Tato uzavřená cesta se nazývá hamitonovská kružnice. Obdobně cesta, která je otevřená a vede všemi vrcholy grafu, se nazývá hamiltonovskou cestou. Na první pohled se hledání hamitonovské kružnice může jevit jako podobná úloha hledání uzavřeného eulerovského tahu, ve kterém hledáme uzavřený tah obsahující všechny hrany grafu. Při hamiltonovské kružnici potřebujeme projít všechny vrcholy grafu, při eulerovském tahu všechny hrany grafu. I když v obou úlohách je zjevná podoba, snadnost řešení tak analogická není. Na rozdíl od úlohy eulerovského tahu, kde máme jednoduchou nutnou a postačující podmínku pro existenci eulerovského tahu, pro existenci hamiltonovské kružnice v grafu žádná jednoduchá nutná a postačující podmínka neexistuje. [4]

Hamiltonovská kružnice je pojmenována po siru Williamovi Rowanovi Hamiltonovi (1805 – 1865), irském matematikovi, fyzikovi a astronomovi. Jeden z jeho nejpozoruhodnějších objevů z roku 1856 je nalezení modelu, který představoval cesty v grafu pravidelného dvanáctistěnu, nazvaného The Icosian Calculus. Jedná se o nekomutativní algebru, která zahrnuje cesty grafické interpretace pravidelného dvanáctistěnu, viz obrázek 4.

Grafickou podobu tohoto modelu prodal výrobci a od roku 1859 se tato hra začala běžně prodávat pod názvem The Icosian Game, viz obrázek 5. Cílem této hry bylo natáhnout vlákno, které by procházelo kolem všech kolíků a tvořilo kružnici.



Obrázek 4 – The Icosian Calculus s vyznačenou hamiltonovskou kružnicí

Díky oblíbenosti hry dostal tento typ grafu název hamiltonovská kružnice. [3]

Obrázek 5 – Ukázky her The Icosian Game a Cestovatel dvanáctistěnem (zdroj: [17])

Logicky lze předpokládat, že hamiltonovskou kružnici lze snadněji najít v grafech s hustou sítí hran než v grafu s malým průměrným stupněm.

Jedny z postačujících nebo nutných podmínek existence hamiltonovského grafu vyslovují věty blíže popsané v nadcházejícím textu. Důkazy vět je možné najít v [4].

Postačující podmínky

Jednu z prvních postačujících podmínek v neorientovaném grafu přinesl v roce 1960 Oystein Ore a přitom ukázal příklady tzv. maximálních grafů bez hamiltonovských cest.

Oreho věta

Mějme graf G s n vrcholy, kde $n \geq 3$. Jestliže pro každé dva nesousední vrcholy u, v grafu G platí $deg(u) + deg(v) \geq n$, tak graf G je hamiltonovský. [4]

Následující Diracova věta je důsledkem Oreho věty.

Diracova věta

Mějme graf G s n vrcholy, kde $n \geq 3$. Je-li $\delta(G) \geq n/2$, tak graf G je hamiltonovský, kde $\delta(G)$ je minimální stupeň grafu. [4]

Nejobecnější větu dokázal v roce 1962 maďarský matematik Pósa.

Pósova věta

Mějme graf G s n vrcholy, kde $n \geq 3$. Jestliže pro každé přirozené číslo $j < n/2$ obsahuje graf G méně než j vrcholů stupně menšího nebo rovného j, tak graf G je hamiltonovský. [4]

V roce 1972 odvodil Václav Chvátal, český matematik a emeritní profesor Montrealské univerzity Concordia, postačující podmínku založenou na posloupnosti stupňů vrcholů grafu. [3]

Chvátalova věta

Mějme graf G s n vrcholy, kde $n \geq 3$. Vrcholy grafu G označme $v_1, v_2, \dots, v_n$ tak, aby platilo $deg(v_1) \leq deg(v_2) \leq \cdots \leq deg(v_n)$. Jestliže pro každé $i < n/2$ platí $deg(v_i) > i$ nebo $deg(v_{n-i}) \geq n - i$, tak graf G je hamiltonovský. [4]

Není obtížné si uvědomit, že Diracova věta je speciálním případem Oreho věty a Oreho věta je speciálním případem Pósovy věty. Stejně tak je ale i Pósova věta speciálním případem Chvátalovy věty. [4] Tyto postačující podmínky však nemusí každý hamiltonovský graf splňovat. Například každý graf C_n (kružnice velikosti n) pro $n \geq 5$ nesplňuje předpoklad uvedených postačujících podmínek, a přece je hamiltonovským grafem, protože sám graf je hamiltonovskou kružnicí. [4]

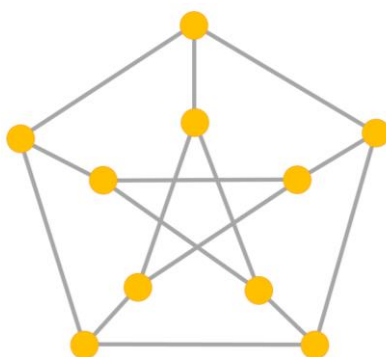
Nutná podmínka

Dle definice hamiltonovského grafu je zřejmé, že každý takový graf musí být souvislý. Dokonce platí, že každý hamiltonovský graf je alespoň 2-souvislý, tedy neobsahuje artikulaci. [3] Následující nutná podmínka říká:

Je-li graf G hamiltonovský graf, tak pro každou neprázdnou vlastní podmnožinu S vrcholové množiny $V(G)$ platí $\omega(G - S) \leq |S|$, kde ω je počet komponent. [4]

Neboli z každé komponenty grafu existuje hrana do jiného vrcholu množiny S, proto množina S obsahuje alespoň tolik vrcholů. Jako je komponent grafu $G - S$. U bipartitních grafů zase platí, že každý hamiltonovský graf má partity stejné velikosti, avšak neplatí, že každý bipartitní graf, který má stejné partity, je hamiltonovský. [4]

Obecně postačující podmínky jsou používány k potvrzení, že graf hamiltonovský je, naopak nutné podmínky jsou využívány k jeho vyvrácení. Jestliže graf nesplňuje některou z nutných podmínek, lze hledání uzavřít s tím, že hamiltonovská kružnice v grafu neexistuje. Existují také grafy, které splňují všechny nutné podmínky, a hamiltonovskými grafy nejsou, příkladem může být Petersenův graf, viz obrázek 6. [4]



Obrázek 6 – Petersenův graf

Nutná a postačující podmínka

Elegantní a rozřešitelná v krátkém časovém rozmezí, nutná a postačující podmínka pro rozhodnutí, zda graf je hamiltonovský, neexistuje. Bude zde uvedena

věta, která je formulovaná jako nutná postačující podmínka pro zjištění, zda graf je hamiltonovský, ale jak pak bude možné z věty vidět, není to tak úplně snadné.

Před jejím uvedením je potřeba říct, co je uzávěr grafu. Uzávěrem grafu G se nazývá graf $cl(G)$, který vznikne postupným přidáváním všech hran mezi nesousední vrcholy u a v, pro které platí, že jejich součet stupňů je alespoň n. [18]

Bondy-Chvátalova věta

Graf G je hamiltonovský právě tehdy, když je hamiltnovský jeho uzávěr $cl(G)$. [18]

Tato nutná a postačující podmínka, ze které vyplývají Oreho a Diracovy postačující podmínky, jenom převádí určení hamiltonicity na jiný graf.

5.2 Hledání hamiltonovské kružnice

Jak je vidět z předchozí kapitoly 5.1, určit, zda je graf hamiltonovský, není jednoduché. Jednoduché není ani najít hamiltonovskou kružnici v hamiltonovském grafu. Tento problém patří do skupiny tzv. NP-úplných problémů. To znamená, že se jedná o matematický problém, který je považován za jeden z nejtěžších z množiny problémů, jejíchž časová složitost je v lepším případě exponenciální (v horším faktoriálová) a které lze řešit v polynomiálně omezeném čase na nedeterministickém Turingově stroji. [19]

Turingův stroj

Turingův stroj byl navržen Alanem Turingem v roce 1936. Využívá se pro modelování algoritmů v teorii vyčíslitelnosti. Deterministický Turingův stroj se skládá z procesorové jednotky, tvořené konečným automatem, programu ve tvaru pravidel přechodové funkce a nekonečné pásky pro zápis výsledků. Z této pásky může číst i zapisovat. Čtecí hlava se pohybuje nad páskou oběma směry. Stroj má příjímací a zamítací stav, ale nemusí těchto stavů dosáhnout, v takovém případě se zacyklí. Nedeterministický Turingův stroj, na rozdíl od deterministického, nemá jednoznačně

definovanou přechodovou funkci, ale pracuje se zobrazením. Výpočet se tak dělí do více větví. [19]

Mezi nejznámější matematické úlohy, které jsou NP-úplné, patří již zmíněné nalezení hamiltonovské kružnice, nalezení úplného podgrafu (problém kliky – viz [20]) či hledání nezávislé množiny (problém splnitelnosti booleovské formule – viz [21]).

Složitost NP-úplných problémů má i své výhody. Právě jejich obtížná řešitelnost nachází uplatnění v moderní kryptografii. Jedná se o případy, kdy je ověření správnosti řešení provedeno velmi rychle (přihlášení na účet), nicméně samotné nalezení řešení trvá velmi dlouho (hackování účtu). [19]

V následujících částech bude představen obecný algoritmus pro nalezení hamiltonovské kružnice a několik heuristických algoritmů řešící daný problém.

5.2.1 Obecný algoritmus

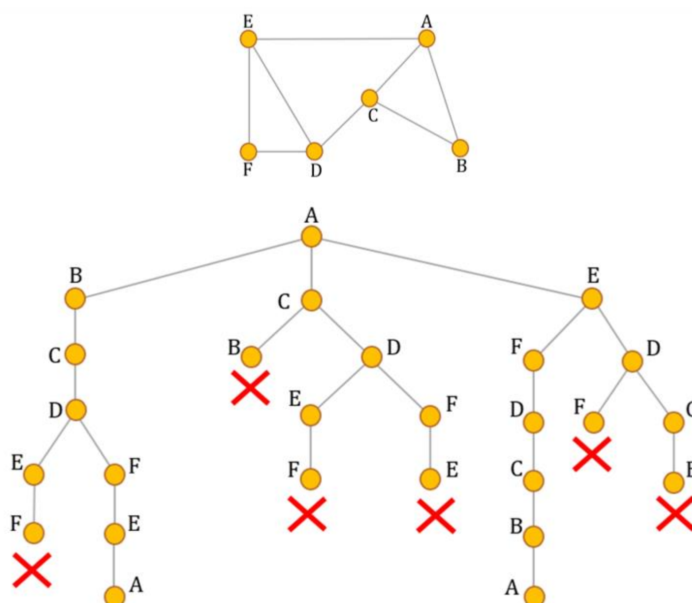
Nejjednodušším algoritmem pro hledání hamiltonovské kružnice je algoritmus „řešení hrubou silou" (anglicky brute-force algorithm). Algoritmus spočívá v systematickém procházení celého vyhledávacího prostoru daného problému. Vyhledávací prostor problému je množina všech možných řešení problému. Velikost vyhledávacího prostoru pro problém hamiltonovské kružnice je faktoriál počtu vrcholů daného grafu. [22]

Během procesu hledání metodou hrubé síly v úplném grafu je nejdříve vybrán jeden vrchol z n možných, následující vrchol grafu může být vybrán $n - 1$ možnými způsoby a další $n - 2$ způsoby atd. [22]. To znamená, že v nejhorším případě může tento algoritmus mít faktoriální složitost (složitost algoritmu vyjadřuje, kolik algoritmus provede operací se vstupními daty) a je tedy pro vyšší počet vrcholů nepoužitelný. Algoritmus řešení hrubou silou, dle [23], je možné sepsat do následujících bodů:

1. *Zvolí se libovolný uzel, který bude kořenem rozhodovacího stromu. Z kořenového uzlu se zvolí úsek, který je s tímto uzlem spojen hranou, do libovolného uzlu.*

2. *Z každého uzlu stromu vychází tolik větví, kolik existuje dalších možných pokračování v trase.*
3. *Trasu nelze rozšířit o již navštívený uzel s výjimkou kořenového uzlu, který je výchozím a zároveň koncovým uzlem konstruovaného stromu.*
4. *Řešení je dosaženo právě tehdy, když byly všechny uzly navštíveny a zároveň koncovým uzlem je uzel výchozí.*

Na obrázku 7 je znázorněn příklad nalezení hamiltonovské kružnice metodou hrubé síly pomocí rozhodovacího stromu. Algoritmus zkouší veškeré možnosti a nalezne kružnice ABCDFEA a AEFDCBA. Druhá zmíněná je převrácenou variantou první kružnice.



Obrázek 7 – Příklad řešení hamiltonovské kružnice hrubou silou (zdroj: [24])

Jak už bylo zmíněno, složitost obecného algoritmu je vysoká a pro vyšší počet vstupů je neaplikovatelný. Tabulka 1 ukazuje hrubý časový odhad pro různé vstupy u algoritmů s exponenciální a faktoriální složitostí oproti například lineární či kvadratické složitosti. Jak je možné vidět, při vstupu 50 trvá výpočet několik let.

$g(n)$	$n = 10$	$n = 20$	$n = 50$	$n = 100$
n	$10\ ns$	$20\ ns$	$50\ ns$	$100\ ns$
$n * \log_2(n)$	$33\ ns$	$86\ ns$	$282\ ns$	$664\ ns$
n^2	$100\ ns$	$400\ ns$	$900\ ns$	$100\ \mu s$
n^3	$1\ \mu s$	$8\ \mu s$	$27\ \mu s$	$1\ ms$
2^n	$1\ \mu s$	$1\ ms$	$10^{21}s \cong 3 * 10^{13} let$	$10^{292}s$
$n!$	$3\ ms$	$10^9 s \cong 31\ let$	$10^{23}s \cong 3 * 10^{15} let$	$10^{2558}s$

Tabulka 1 – Příklad doby výpočtu funkce $g(n)$ (zdroj: [25])

Výhodami obecného algoritmu je skutečnost, že nalezne opravdu nejlepší řešení dané úlohy, a jednoduchost jeho implementace. Nicméně kvůli své složitosti se v praxi příliš nepoužívá. Výjimkou jsou úlohy s malým prostorem. Úlohy, kde je přesnost důležitější než rychlost výpočtu, nebo může posloužit jako nástroj pro heuristiky, které dokážou zmenšit velikost prostoru. [26]

5.2.2 Heuristické algoritmy

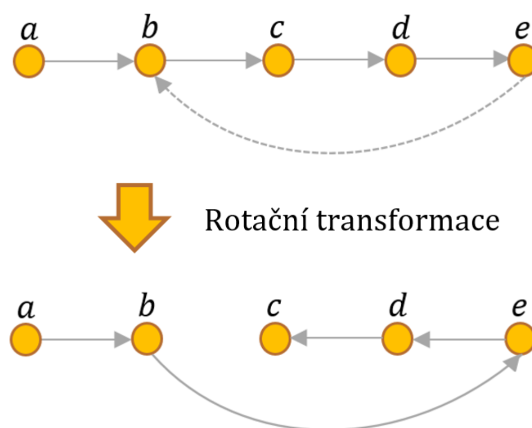
V praxi, při hledání hamiltonovských kružnic a taktéž jiných NP-problémů se především využívá různých heuristických algoritmů. Jedná se o algoritmy, které dokáží úlohy vyřešit v polynomiálním (přijatelném) čase. Jsou založeny na myšlence, jak by danou úlohu mohlo jít vyřešit. Z toho důvodu může být jejich výsledek o něco horší, než výsledek přesného algoritmu, ale také může najít i optimální řešení. NP-úloha může být řešena různým počtem heuristických algoritmů, kdy každý z nich bude mít jinou časovou složitost i přesnost výsledků. [27]

Výhodou heuristických algoritmů je tedy jejich polynomiální časová složitost výpočtu. Naopak nevýhodou je pak jejich přesnost výsledků získaných těmito algoritmy. Následují popisy nejzákladnějších heuristických algoritmů, řešící problém hamiltonovské kružnice. [27]

Hladový algoritmus je založen na principu rozhodovacího stromu. Hlavní rozdíl spočívá ve výběru vrcholů v grafu. Hladový algoritmus začíná hamiltonovskou

cestu od vrcholu nejvyššího stupně. Následující vrcholy vybírá na základě jejich stupně. Algoritmus totiž upřednostňuje vrcholy s menším stupněm, protože jsou hůře dostupné. Díky tomu se nedostává často do slepých uliček, jako obecný algoritmus.

Další heuristický algoritmus - Rotační transformace - je založen na postupném přidávání vrcholů za účelem sestavení hamiltonovské cesty a tu poté uzavřít v kružnici. Pokud dojde k situaci, kdy nelze přidat další vrchol, pak použije záměnu pořadí vrcholů a snaží se dále hledat jinou variantu cesty, viz obrázek 8. Aby se předešlo hledání stejných variant, ukládá si množinu S, do které se vkládají zaměněné vrcholy. [28]



Obrázek 8 – Rotační transformace (zdroj: [29])

Postup rotační transformace je popsán níže: [29]

1. *Najít sousední vrchol s koncovým vrcholem e v cestě* P *(např. vrchol b)*
2. *Vytvoření nové cesty* P*:*
    a. *Spojením vrcholů e a b*
    b. *Obrácením směru cesty od c do e*

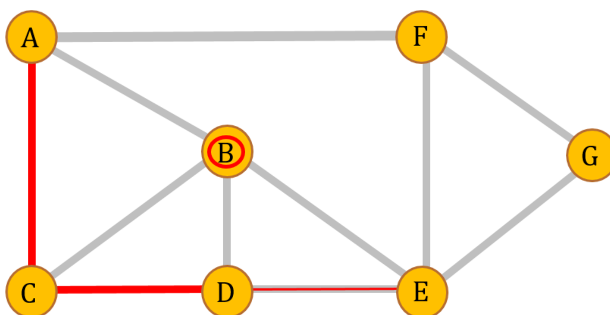
Rotační technika, vyvinutá Pósou, nalezne řešení v polynomiálním čase v hustých grafech [30]. Lizhi Du tuto techniku zobecnil pro všechny obecné grafy ve svém článku „An Efficient Algorith for Hamilton Cycle Based on the Enlarged Rotation-Extension Technique“ [30].

Třetí velmi známý heuristický algoritmus je algoritmus nedosažitelného vrcholu. Tato heuristika se při vytváření hamiltonovské kružnice snaží snížit šanci dosažení slepé uličky, tedy bodu, ze kterého není možné pokračovat dál. [29]

Pravidla heuristiky: [29]

1. *Necht* P *je částečná cesta a* X *je koncový vrchol na částečné cestě*
2. *Vyber další sousední vrchol* Y *vrcholu* X *takový, že počet všech jeho nenavštívených sousedních vrcholů je větší než jedna*

Vrchol je považovaný za nedosažitelný, pokud jsou všechny jeho sousední vrcholy součástí cesty P. V takovém případě neexistuje žádný způsob, jak vrcholu dosáhnout a stane se nedosažitelným. Pokud výběr vrcholu způsobí, že některý jeho nenavštívený sousední vrchol je nedosažitelný, pak nebude tento vrchol přidán do cesty P. [29]



Obrázek 9 – Heuristika nedosažitelného vrcholu (zdroj: [29])

Kombinací těchto tří algoritmů zkonstruovala Seeja ve svém článku [29] hybridní algoritmus HYBRIDHAM. Tento algoritmus dosahuje kubické složitosti, tedy $O(n^3)$. [29]

5.3 Aplikace hamiltonovské kružnice a cesty

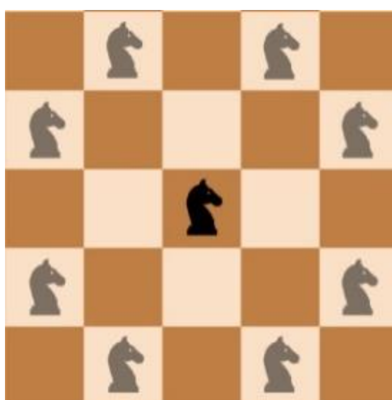
Dvě nejznámější úlohy, které lze přeformulovat na problém hledání hamiltonovské kružnice, jsou problém obchodního cestujícího a jezdcova procházka na šachovnici.

5.3.1 Problém obchodního cestujícího

Obchodní cestující má za úkol navštívit všechna města v dané oblasti. Každé město musí navštívit právě jednou a musí se vrátit do výchozího města. Tedy hledá se hamiltonovská kružnice. Při řešení takové úlohy se požaduje, aby celková vzdálenost, kterou obchodní cestující urazí, byla co nejmenším. Jde tedy o nalezení minimální hamiltonovské kružnice v ohodnoceném grafu. [31]

5.3.2 Jezdcova procházka

Druhou velmi známou hádankou popisující hamiltonovskou kružnici je jezdcova procházka (anglicky Knight's tour). Jezdec zde představuje šachovou figurku, jejíž pohyb je dán písmenem L, podle pravidel šachu (obrázek 10). [32]



Obrázek 10 - Pohyb šachového jezdce

Problém, známým po staletí, je navštívit všechna pole šachovnice pouze jednou, přičemž se jezdec může, ale nemusí, vrátit do své startovací pozice. Úloha má tedy dva druhy řešení – otevřená procházka a uzavřená procházka. U otevřených řešení jezdec navštíví každé pole šachovnice právě jednou, v tom případě se jedná o hamiltonovskou cestu, kdežto u uzavřeného řešení jezdec sice navštíví každé pole šachovnice právě jednou, ale s výjimkou počátečního pole, ze kterého procházku začínal. Uzavřené řešení má tedy o jeden krok navíc a to je krok, kterým se jezdec dokáže jedním tahem vrátit na výchozí pole. V tom případě se jedná o kružnici, protože jezdec tímto tahem uzavře svoji procházku do kružnice. Platí, že každá hamiltonovská kružnice obsahuje hamiltonovskou cestu. Opak ale neplatí,

prodloužením každé hamiltonovské cesty o jeden krok nemusí vytvořit hamiltonovskou kružnici.

Právě uzavřené řešení jezdcovy procházky spadá do problému nalezení hamiltonovské kružnice. Každé uzavřené řešení lze provést dvěma směry (jedná se o kružnici). Považujeme-li tato dvě řešení za různá, říkáme, že jsou to řešení orientovaná. V opačném případě, považujeme-li tato řešení za totožná, nazýváme řešení neorientovaná. [32], [33]

První specifické řešení tohoto problému našli De Montmort a Abraham De Moivre, každý zvlášť na počátku 18. století. Moivrova metoda spočívala v pohybu jezdce po okraji šachovnice, jakmile nebyl tento pohyb dále možný, provedl tah do středního čtverce, který tvoří 16 polí, jehož vyřešení není již složité. [4]

Podrobněji byl tento problém řešen až v polovině 18. století, kdy se jím zabýval Leonhard Euler, který poznamenal, že tento problém není možné vyřešit na šachovnicích NxN o liché velikosti N (např. 5×5). [3]

V roce 1771 Alexandre – Theopile Vandermonde našel algebraické řešení jezdcovy procházky na šachovnici velikosti 8×8. Ve své metodě využíval šachových souřadnic a symetrie šachovnice, což přispělo k zobecnění řešení pro vícerozměrné šachovnice. [3]

S efektivním řešením jezdcovy procházky přišel v roce 1923 Warnsdorff. Tato metoda spočívá v upřednostňování políček, ze kterých je menší možnost dalších tahů. Nejdříve jsou tedy navštívena místa, co nejblíže krajům šachovnice. [34]

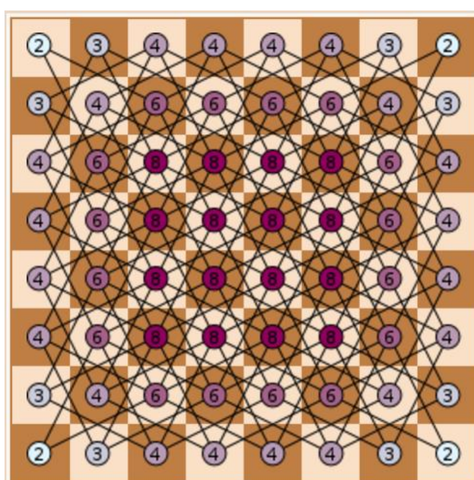
Nejčastější otázka, kolik takových řešení jezdce na šachovnici existuje, však dodnes nebyla zcela vyřešena i přesto, že existují metody stanovení počtu hamiltonovských kružnic pro některé typy grafů. V roce 1946 C. A. B. Smith dokázal, že v pravidelném grafu 3. stupně bez smyček a násobných hran je počet hamiltonovských kružnic sudý. Důsledkem je skutečnost, že pokud graf obsahuje hamiltonovskou kružnici, pak obsahuje nejméně tři takové kružnice. Dále Juraj Bosák v roce 1967 dokázal, že pravidelné bipartitní grafy 3. stupně mají sudý počet hamiltonovských kružnic. [3]

Právě úloha jezdcovy procházky byla vybrána k implementaci pro aplikaci CheGra. Na rozdíl od obecnějšího problému (hledání hamiltonovské kružnice) je

problém jezdcovy procházky řešitelný v lineárním čase. V porovnání s úlohou obchodního cestujícího jezdcova procházka nebere ohled na vzdálenosti mezi jednotlivými políčky, a tak není hledána nejkratší trasa. Samotnou implementací algoritmů pro nalezení procházky se práce zabývá v kapitole Implementace algoritmů. [31] [32]

Graf šachovnice je reprezentován následovně: Jednotlivá pole šachovnice jsou vrcholy grafu a hrana mezi vrcholy existuje právě tehdy, když se jezdec může dostat jediným tahem z jednoho vrcholu na druhý.

Jezdec má v každém kroku nejvýše 8 možných tahů. Počet cest, které může šachový jezdec na šachovnici NxN je $4(n-2)(n-1)$ [34]. Šachovnici o velikosti $8x8$ znázorňující všechny možné tahy jezdce pro řešení jezdcovy procházky lze nalézt na Obrázek 11. Čísla v každém uzlu označují počet možných pohybů jezdce, které lze z této pozice učinit.



Obrázek 11 – Všechny možné tahy jezdce na šachovnici $8x8$

Existence uzavřeného řešení na šachovnici stanovuje Schwenkova věta:

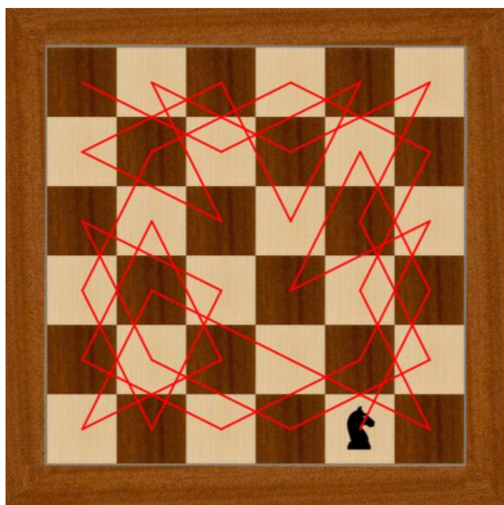
Schwenkova věta

Pro libovolnou šachovnici o rozměrech MxN polí je uzavřené řešení jezdcovy procházky vždy možné, pokud platí $M \leq N$ a neplatí, že: [35]

1) *Čísla M a N jsou lichá*
2) *$M = 1, 2$ nebo 4; M a N nejsou současně rovna 1*

3) $M = 3 \; a \; N = 4, 6 \; nebo \; 8$

Úloha má mnoho variant. Jezdcovu procházku lze nalézt na různých velikostech šachovnic. Nejtypičtější variantou je šachovnice o velikosti $8x8$ polí, ale úlohu lze řešit i na větších/menších šachovnicích, které mohou/nemusejí být souměrné, viz ukázka řešení úlohy na šachovnici 6x6 na Obrázek 12. Právě velikost šachovnice výrazně komplikuje hledání jezdcovy procházky.



Obrázek 12 – Příklad otevřeného řešení jezdcovy procházky na šachovnici $6x6$

5.3.2.1 Popis problému

Obecně platí, že čím větší šachovnice je, tím více řešení má. Při malých rozměrech šachovnic dostáváme ještě reálné množství řešení, nicméně rozšíření takové šachovnice o jedno další pole má za následek exponenciální vzrůst počtu řešení.

Počet neorientovaných uzavřených procházek na šachovnici $8x8$ byl nezávisle vypočítán v 90. letech McKayem a Wegenerem, kdy se zjistilo, že takových cest je 13 267 364 410 532, což je přibližně 10^{13} (tedy 26 534 728 821 064 orientovaných cest). [33]

Nyní se nabízí otázka, kolik je otevřených řešení na šachovnici $8x8$. Metodou vzorkování důležitosti kombinovanou s Warnsdorffovým pravidlem přišli Cancela a Mordecki v roce 2006 k domněnce, že jich je přibližně $1{,}22 * 10^{15}$ cest, které jsou

neorientované, nesymetrické a bez rotací. Přesný výsledek nebyl znám, proto se používal 99% interval spolehlivosti $< 1{,}220\,;1{,}225 > * 10^{15}$. [33]

Přesný počet otevřených řešení jezdcovy procházky na šachovnici $8x8$ byl zveřejněn až v roce 2015 Alexandrem Chernovem, který uvedl, že konečný počet je 9 795 914 085 489 952 neorientovaných cest se symetrií a rotacemi (tedy 19 591 828 170 979 904 orientovaných). Nicméně tento výsledek nebyl doposud ověřen, zda je správný. Počty dnes známých řešení jsou zapsány v Tabulce 2. [33]

Není náhodou, že vyhledání všech řešení je obtížným úkolem. Jedná se o nereálný úkol i v dnešní době výkonných počítačů. K provedení takové úlohy je zapotřebí vyzkoušet všechny možné cesty jezdce po šachovnici. Takových cest je na šachovnici $8x8$ přesně 64!, což je přibližně $1{,}268 * 10^{89}$. Takový výpočet by trval několik let.

Tento problém se neformálně nazývá kombinatorická exploze, nebo také prokletí dimenzionality („Curse of Dimensionality"). Jedná se o situaci, kdy složitost výpočtu silně vzrůstá se vzrůstajícím počtem na vstupu. Tudíž se vzrůstající velikostí šachovnice silně vzrůstá i složitost nalezení řešení úlohy. Kombinatorická exploze je často spojována s problémem nalezení nejlepší strategie šachů.

N	Počet řešení šachovnice NxN
1	1
2	0
3	0
4	0
5	1 728
6	6 637 920
7	165 575 218 320
8	19 591 828 170 979 904

Tabulka 2 – Počet řešení jezdcovy procházky (zdroj: [33])

5.3.2.2 Řešení úlohy

Existuje několik algoritmů, které umí vyhledat jezdcovu procházku na zadané šachovnici. Některé z nich byly implementovány v aplikaci CheGra. Konkrétně Backtracking, Warnsdorffův algoritmus a algoritmus řešící jezdcovu procházku pomocí Hopfieldovy neuronové sítě.

Všechny algoritmy mají jedno společné a to reprezentaci šachovnice. Každé pole je uloženo v nějaké kolekci (v případě této práce je to ArrayList) a má své identifikační číslo (id), kterým se odlišuje od ostatních (viz Obrázek 13).

Obrázek 13 – Číselná reprezentace polí šachovnice

Následující části se zabývají popisu implementovaných algoritmů.

5.3.2.3 Backtracking

Nejjednodušším způsobem řešení problému je Backtracking. Algoritmus spočívá v náhodném skákání jezdce na doposud neobsazená pole, dokud se neocitne v situaci, kdy se nemůže dále posunout. V takové situaci se jezdec vrátí na předešlé pole a pokračuje jinou cestou. Naivní algoritmus má ovšem tendenci být velmi pomalý, protože se dostává snadno do těchto slepých uliček. [34] Všimněte si analogie s řešením hrubou silou. Backtracking je nadstavbou obecného algoritmu, který dokáže vyřadit potenciálně špatná řešení dříve, než budou plně vyzkoušena díky omezením, které jsou po každém tahu vyhodnoceny.
Algoritmus, dle [36], je možné napsat v těchto základních krocích:

K1: Zvolí se výchozí pole

K2: Přidá se jeden z osmi možných tahů jezdce, který ještě nebyl vyzkoušen, do řešení

K3: Pokud byla všechna pole navštívena -> krok K6

K4: Pokud jezdec skončí ve slepé uličce -> krok K7

K5: Zpět ke kroku K2

K6: Hamiltonovská cesta/kružnice byla nalezena; algoritmus končí

K7: Odstraní se tento tah z řešení; zpět ke kroku K2

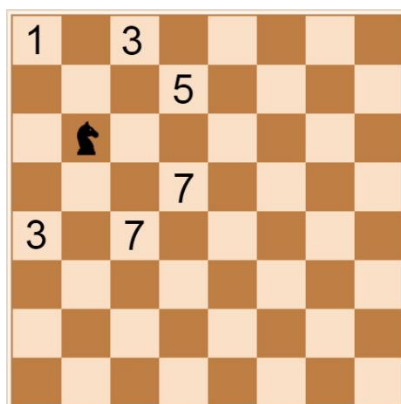
5.3.2.4 Warnsdorffův algorimus

V roce 1823 přišla optimalizace Backtrackingu – Warnsdorffův algoritmus. Algoritmus spočívá v nalezení cesty bez jakéhokoli zpětného sledování pomocí výpočtu ohodnocení pro následující kroky, které ještě nebyly navštíveny a mohou být dosaženy jedním jediným pohybem jezdce z dané pozice. Jezdec se vždy vydá přednostně na to pole, ze kterého může pokračovat nejméně způsoby. Tímto způsobem jsou nejprve navštěvována špatně dostupná pole, zatímco ta snadno dostupná pole jsou odložena na později, viz následující popis algoritmu dle [37]:

K1: Zvolí se výchozí pole
K2: Provedení výpočtu ohodnocení nenavštívených sousedních vrcholů
K3: Přidá se jeden z osmi možných tahů jezdce s nejnižším ohodnocením, který ještě nebyl vyzkoušen, do řešení
K4: Pokud byla všechna pole navštívena -> krok K7
K5: Pokud jezdec skončí ve slepé uličce -> krok K8
K6: Zpět ke kroku K2
K7: Hamiltonovská cesta/kružnice byla nalezena; algoritmus končí
K8: Odstraní se tento tah z řešení; zpět ke kroku K2

Warnsdorffův algoritmus je typickým příkladem hladového algoritmu [38]. Čas potřebný pro tento algoritmus roste zhruba lineárně s počtem čtverců šachovnice, nicméně počítačová implementace ukazuje, že tento algoritmus naráží na slepé uličky pro šachovnice větší než $76x76$, přestože funguje dobře na menších šachovnicích. [34]

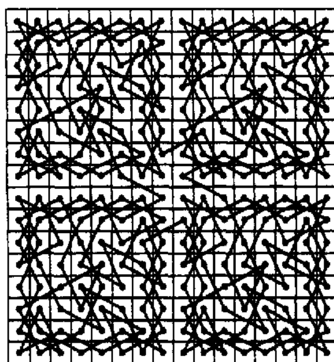
Na Obrázek 14 lze nalézt grafické znázornění Warnsdorffova pravidla. Každý čtverec obsahuje číslo udávající počet možných pohybů, které by mohl jezdec udělat z tohoto pole. V tomto případě pravidlo říká, přesuň jezdce na pole s nejmenším číslem, tedy 1.



Obrázek 14 – Warnsdorffové pravidlo

5.3.2.5 Conradův algoritmus

První algoritmus efektivní i pro šachovnice větší než $76x76$ je Conradův algoritmus z roku 1994. Algoritmus spočívá v rozložení šachovnice na menší šachovnice (nemusí být čtvercové), viz obrázek 15, pro něž je řešení jezdcovy procházky známé. [39]



Obrázek 15 – Conradův algoritmus na šachovnici $16x16$ (zdroj: [39])

5.3.2.6 Neuronová síť

Poslední možností, jak jezdcovu procházku vyřešit, je pomocí neuronové sítě.

Neuronové sítě dělíme na dva typy – biologické a umělé. Biologické neuronové sítě představují hlavní část nervového systému – mozku. Lidský mozek je nesmírně komplexní a v současnosti se jedná o nejvíce složitý objekt ve známém vesmíru.

Jeho složitost je dána počtem neuronů počtem propojení, které mezi sebou tyto neurony navzájem mají. Grafické znázornění neuronu lze vidět na Obrázek 16. [40]

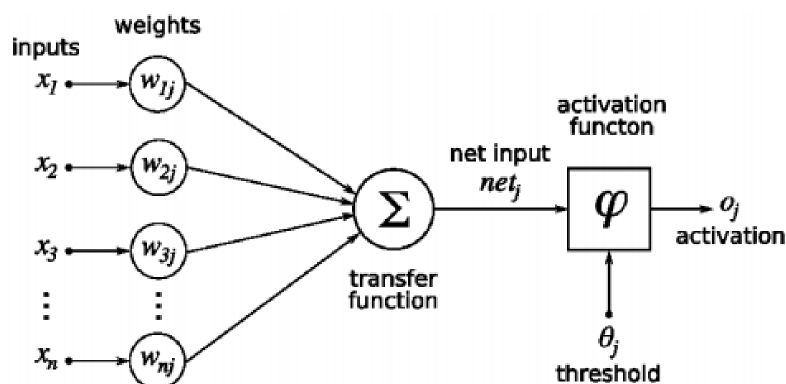
Umělé neuronové sítě jsou inspirovány principem fungování neuronů v lidském mozku. Sítě se skládají z neuronů, které jsou seskupené do několika vrstev. Jedná se o velmi populární nástroje pro řešení a modelování složitých problémů. Své popularity však dosáhly především v analýze obrazu nebo regresních a klasifikačních úlohách. [41]

Obrázek 16 – Grafické znázornění neuronu (zdroj: [42])

Umělý neuron napodobuje funkci biologického neuronu. Nicméně pořád se jedná pouze o jednoduchý matematický model, který nedosahuje reálné složitosti biologické neuronové buňky. Umělé neuronové sítě se tak pouze inspirují biologickými a nejedná se o přímé modely. [43], [41]

Stejně jako u biologických neuronových sítí, tak rovněž u umělé neuronové sítě se skládají z jednotlivých neuronů, které také fungují na principu předávání informace [41]. Nejpoužívanějším modelem neuronu je tzv. formální neuron (Obrázek 17) publikovaný autory W. S. McCulloch a W. Pittson. Jedná se o první reálně použitelný matematický model neuronu. [43]

Mezi hlavní výhody umělých neuronových sítí patří schopnost učit se, a to bez nutnosti explicitní znalosti algoritmu řešení a schopnost generalizovat, tedy správně si zařazovat poznatky, které se nenacházely v trénovací sadě. [41], [43]



Obrázek 17 – Formální neuron (1943) (zdroj: [43])

Mezi nevýhody patří náročnost na volbu parametrů sítě (počty vrstev, počty neuronů, aktivační funkce, velikost optimalizačního kroku), sklony k přetrénování a vysoká výpočetní náročnost při trénování rozsáhlých sítí. [43]

5.3.2.7 Neuronová síť pro řešení jezdcovy procházky

Pro neuronovou síť v aplikaci CheGra byla vybrána Hopfieldova architektura. Tato síť je reprezentována tak, že její velikost odpovídá velikosti šachovnice, každý tah šachového jezdce je reprezentován neuronem, který je inicializován náhodně, buď je aktivní, nebo neaktivní (jeho výstup je 1 nebo 0). Aktivní neuron znamená, že příslušný krok jezdce je součástí jezdcovy procházky. Každý neuron má svoji stavovou funkci, která je inicializována na 0. Po spuštění sítě jsou neurony aktualizovány dle pohybu jezdce. Mohou tak změnit svůj stav a výstup na základě stavů a výstupů svých sousedů. Každou iterací se tak hledají stabilní stavy všech neuronů, aby ve výsledku byla celá neuronová síť stabilní. [44]

Postup algoritmu neuronové sítě dle [39]:

K1: Vytvoření neuronové sítě o velikosti šachovnice
K2: Každému neuronu inicializován stav na 0
K3: Každému neuronu inicializován výstup náhodně na 0 nebo 1
K4: Aktualizace neuronů matematickou funkcí, dle pohybu jezdce
K5: Pokud je překročen počet iterací -> krok K2
K6: Je-li hamiltonovská kružnice nalezena -> konec algoritmu
K7: Zpět ke kroku K4

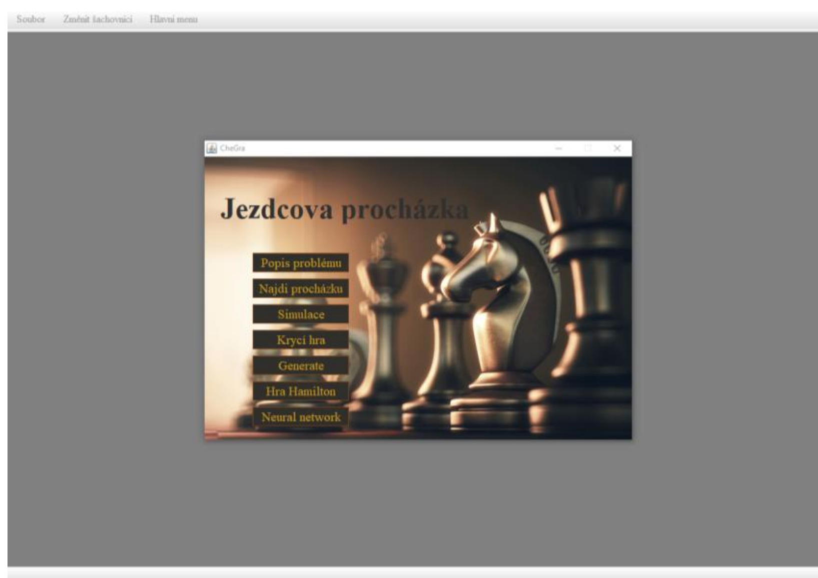
Nicméně tato poslední varianta není považována za standard pro řešení jezdcovy procházky, přestože je považována za smart technologii. [45] Proč tomu tak je, se pokusil autor v této práci zjistit. Zda je běžný algoritmus (Warnsdorffův) při řešení jezdcovy procházky o tolik efektivnější než řešení pomocí neuronové sítě, případně o kolik, jak velký rozdíl to je. Či který algoritmus bude více ovlivněn zvětšováním velikosti šachovnice pro $N > 8$, viz kapitola 8 – Implementace algoritmů.

6 Aplikace CheGra

Vytvořená desktopová aplikace pojmenovaná CheGra (Obrázek 18) měla obsahovat různé hry na šachovnici, případně hry týkající se hamiltonovských grafů. Po konzultacích s konzultantem práce byly pro aplikaci zvoleny následující hry:

1) Hledání hamiltonovské cesty (Najdi procházku)
2) Krycí hra
3) Hledání hamiltonovské kružnice (Hra Hamilton)

Konečná aplikace nenabízí pouze zmíněné tři hry. Autor práce program rozšířil o další zajímavé funkce, především pro jeho zvědavost a pro rozšíření jeho diplomové práce. Konkrétně implementoval algoritmy řešící jezdcovu procházku – Backtracking, Warnsdorffův algoritmus a algoritmus založený na Hopfieldově neuronové síti.



Obrázek 18 – Aplikace CheGra

6.1 O aplikaci

Aplikace CheGra je určena především pro podporu výuky teorie grafů v předmětech DIMA a DMO. Slouží k snazšímu pochopení problematiky NP-úplných

problémů – konkrétně hledání hamiltonovské cesty a kružnice v typické úloze jako je jezdcova procházka.

Pro akce fakulty organizované pro středoškolské studenty byla aplikace zjednodušena za účelem zpřehlednění aplikace. Liší se pouze nabídkou hlavního menu. Odlehčená varianta je ochuzena o tlačítka v hlavním menu (Generate, Neural network), které spouští rozšířené funkce. To znamená, funkce jsou implementovány, ale uživatel se k nim nedostane (viz Obrázek 19).

Obrázek 19 – Nabídka hlavního menu odlehčené verze pro studenty (vlevo) a plné verze (vpravo)

6.2 *Hlavní nabídka*

Po spuštění aplikace se uživateli nabídne hlavní menu aplikace. Obsahuje seznam hlavních funkcí aplikace. Od popisu matematického problému jezdcovy procházky, přes možnost vyzkoušení si vyřešit tento problém, až po ostatní hry a funkce pro hledání jezdcovy procházky různými algoritmy, viz Obrázek 19.

Některé hry a funkce nabízejí své různé varianty. Proto po kliknutí na danou hru se hlavní nabídka změní a odhalí uživateli varianty zvolené hry. Pro příklad je na Obrázek 20 zobrazena nabídka, která se zobrazí po zvolení hry „Najdi procházku“. Z této nabídky si uživatel může zvolit, pro kterou variantu šachovnice bude procházku hledat. Samozřejmě, uživatel může toto rozhodnutí kdykoli změnit pomocí tlačítka „Změnit šachovnici“, více v 6.3.1. Hlavní menu v tomto případě dále nabízí tlačítko zpět, které uživateli umožní se vrátit do hlavní nabídky.

Obrázek 20 – Nabídka variant hry „Najdi procházku“

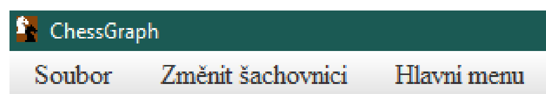
6.3 Lišty aplikace

Aplikace nabízí dvě lišty – horní a dolní. Horní lišta umožňuje uživateli orientaci v aplikaci. Obsahuje tři fixní tlačítka, která se nemění, ale v některých hrách mohou být jejich funkce nedostupné. Dolní lišta obsahuje ovládací prvky k jednotlivým hrám a má proměnlivou nabídku. Tlačítka se na této liště zobrazují v závislosti na zvolené hře.

6.3.1 Horní lišta

Horní lišta (viz Obrázek 21) slouží převážně pro orientaci uživatele. Obsahuje následující záložky:

1) Soubor
2) Změnit šachovnici
3) Hlavní menu



Obrázek 21 – Horní lišta aplikace

V první záložce „Soubor“ lze zde nalézt informace o samotném programu, za jakým účelem byl vytvořen a kdo je jeho autorem. Nachází se zde také tlačítko pro vypnutí aplikace.

Druhá záložka „Změnit šachovnici“ je dostupná pouze u her, které mají více variant rozložení šachovnice, konkrétně u Krycí hry a Najdi procházku. Záložka nabízí seznam variant této hry, díky čemuž se uživatel může snadno přepnout na obtížnější potažmo lehčí variantu hry. Ve výsledku se změní šachovnice, ale hra zůstává stejná.

Poslední záložka „Hlavní menu“ otevírá hlavní nabídku aplikace. Uživateli slouží především pro přepnutí z jedné hry na jinou, případně k opětovnému spuštění hry. Zaručuje tak snadnou orientaci uživatele v aplikaci.

6.3.2 Dolní lišta

Dolní lišta (viz Obrázek 22) se přizpůsobuje zvolené hře. Každá hra má na této liště jiné tlačítko. Funkce těchto tlačítek budou podrobně vysvětleny v příslušných kapitolách popisující konkrétní hry.

Obrázek 22 – Dolní lišta Krycí hry

6.4 Popis problému

První položka v hlavní nabídce „Popis problému“ přibližuje hráči problém jezdcovy procházky. Problém je popsán jednoduše. Snahou bylo přiblížit problém studentům jak středoškolským navštěvující akce UHK, tak studentům navštěvující předmět DIMA zajímavější formou.

Funkce popisující problém jezdcovy procházky zobrazí okno, které se skládá ze dvou částí. Teoretické části, která je umístěna v levé části okna. Ta popisuje pravidla matematické úlohy a grafické části zobrazující nápomocné obrázky k lepšímu pochopení úlohy, viz Obrázek 23.

Obrázek 23 – Popis problému

Pod teoretickou částí popisující jezdcovu procházku lze nalézt dvě tlačítka pro listování mezi stránkami. Při každé změně stránky se změní i grafická část. To znamená, že každá stránka má svůj originální obrázek, který slouží pro lepší představu teoretické myšlenky.

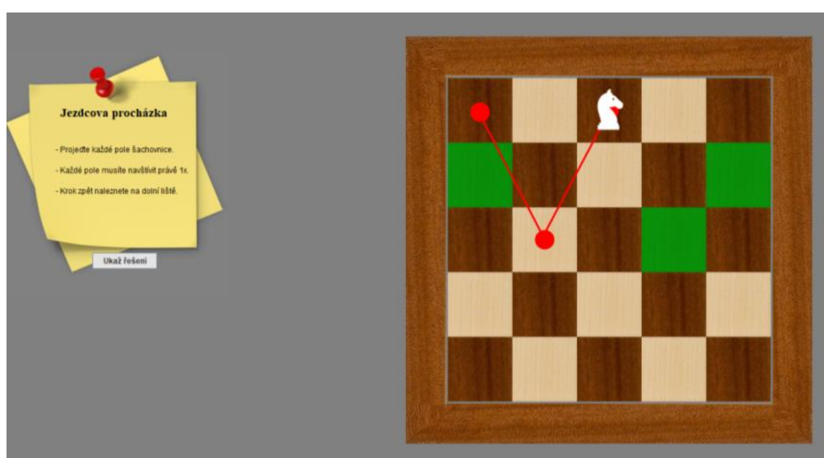
Unikátní stránkou je ta poslední, která nabízí nová tlačítka „Chci to zkusit" a „Spustit ukázku" umístěné opět pod teoretickou částí. První z tlačítek přesune uživatele z teoretického okna do okna hry „Najdi procházku" (viz kapitola Hra – Najdi procházku), kde si může zkusit vyřešit problém jezdcovy procházky. Druhé tlačítko „Spustit ukázku" po kliknutí spustí animaci znázorňující příklad, jak může být tato matematická úloha vyřešena. Během animace nelze aplikaci dále používat, veškerá tlačítka jsou nepřístupná. Uživatel tak musí nechat animaci doběhnout a poté může aplikaci dále normálně používat.

6.5 Hra – Najdi procházku

Cílem hry je nalézt na zvolené šachovnici jezdcovu procházku, viz Obrázek 24. Výchozí pozice koně je v levém horním rohu šachovnice a hráč se snaží navštívit každé pole šachovnice právě jednou dle pravidel pohybu šachového jezdce, tedy do písmena L.

Tato hra nabízí tři varianty šachovnice: 4x4, 5x5 a 6x6. I když na šachovnici 4x4 žádná hamiltonovská cesta neexistuje, je tato šachovnice tady zařazena s cílem přimět studenty přimět zapřemýšlet, že ne každá šachovnice takové řešení má.

Ovládání této hry je intuitivní. Hráč může buď koně přesouvat tahem, nebo může klikat na zeleně zvýrazněná pole, které označují možné směry, kudy se může jezdec aktuálně vydat. Cesta koně po šachovnici je zakreslena červenými cestami a na navštívených polích jsou vykreslena červené kolečka. Uživatel tak lehce pozná, které pole již bylo navštíveno a které nikoli.



Obrázek 24 – Hra „Najdi procházku“

Hra má více variant z pohledu velikosti šachovnice, kterou lze volit tlačítkem „Změnit šachovnici“ v horní liště aplikace. Hráč si může vybrat z velikostí: 4x4, 5x5 a 6x6.

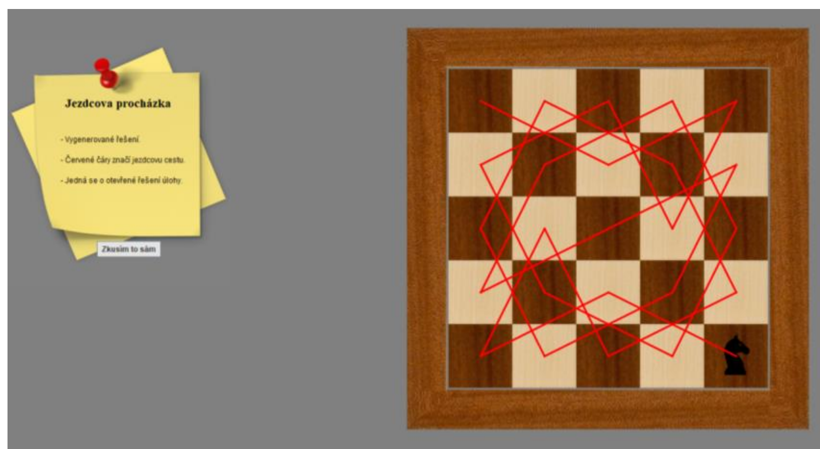
Na dolní liště lze nalézt tlačítko „Krok zpět“, které, jak naznačuje název, vrátí koně na předchozí pole šachovnice, na kterém se nacházel.

V levé části aplikace se nachází informační cedulka popisující hráči pravidla hry a pod ní se nachází tlačítko „Ukaž řešení“, které umožňuje uživateli, jenž si neví rady, jak hru dokončit, zobrazit jedno ukázkové řešení, které daná šachovnice má. Výjimkou je varianta velikosti šachovnice 4x4, která slouží jako „chyták“. Tato akce je pouhým přemístěním hráče ze hry „Najdi procházku“ na funkci „Simulace“ pro danou šachovnici. Hráč se může vrátit zpět pomocí tlačítka „Zkusím to sám“ (viz kapitola Simulace). Po úspěšném nalezení procházky se hráči zobrazí okénko, informující hráče o splnění cíle hry.

6.6 Simulace

Nejedná se o hru, nýbrž o funkci aplikace, která částečně navazuje na předchozí hru hledání jezdcovy procházky. Uživatelé, kteří totiž nepřijdou na to, jak tuto matematickou úlohu vyřešit na zadané šachovnici, se mohou zajímat o to, jak takové řešení tedy vypadá. Právě proto se zde nachází funkce simulace řešení jezdcovy procházky.

Výsledkem simulace je šachovnice, na které je vyobrazena jezdcova procházka červenými čarami (viz Obrázek 25). Simulace se drží pravidel hledání jezdcovy procházky, tedy včetně počátečního umístění koně v levém horním rohu šachovnice. V levé části se nachází opět stejná informační cedulka jako ve hře „Najdi procházku“. Jedinou změnou je odlišné tlačítko umístěné pod touto cedulkou. Zde je umístěno tlačítko „Zkusím to sám“, které uživatele přemístí do hry „Najdi procházku“ se stejnou variantou šachovnice.



Obrázek 25 – Simulace řešení šachovnice 5x5

6.7 Hra – Krycí hra

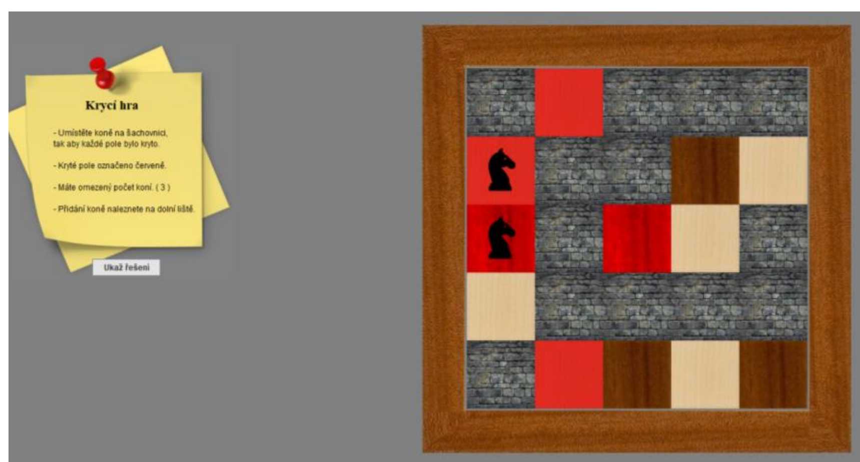
Další v pořadí je hra nesoucí název „Krycí hra“, jejíž cílem je umístit na šachovnici koně tak, aby každé pole šachovnice bylo těmito koni kryto. Tedy, aby každé pole bylo ohroženo některým z umístěných koní. Pole, která jsou koněm ohrožována, jsou na šachovnici zvýrazněna červeně.

Přemístění koně provede uživatel tahem šachového jezdce na příslušné pole. Koně může postavit pouze na neobsazené pole šachovnice, které zároveň není zdí.

Každá varianta hry má omezený počet koní, které může uživatel na šachovnici umístit. Přesný počet těchto poskytnutých koní lze nalézt v levé části aplikace, kde se také, mimo jiné, nachází i pravidla této hry. Jakmile je cíl hry splněn, zobrazí se oznamující okénko. Nového koně uživatel přidá pomocí tlačítka „Nový kůň“ na dolní liště aplikace.

Tato hra nabízí různé varianty šachovnic a to nejen svoji velikostí, ale také svoji strukturou. V některých variantách krycí hry jsou na šachovnici umístěny zdi, které nedovolují na dané pole postavit šachového koně. Díky tomu se úloha zdá být studentovi složitější (Obrázek 26).

Pro přechody mezi variantami šachovnice slouží tlačítko „Změnit šachovnici“. Úspěšné splnění úkolu je oznámeno hráči informativním okénkem. Opět pro případ nezpůsobilosti najít řešení hry nabízí jedno ukázkové řešení pod tlačítkem „Ukaž řešení“.



Obrázek 26 – Hra „Krycí hra“

6.8 Hra – Hamilton

Poslední nabízenou hrou aplikace CheGra je hra Hamilton. Tato hra je založena na hledání hamiltonovské kružnice na zadaných grafech. Uživateli je na obrazovce zobrazen graf složený z vrcholů a hran. Cílem je navštívit každý vrchol grafu a vrátit se do počátečního vrcholu.

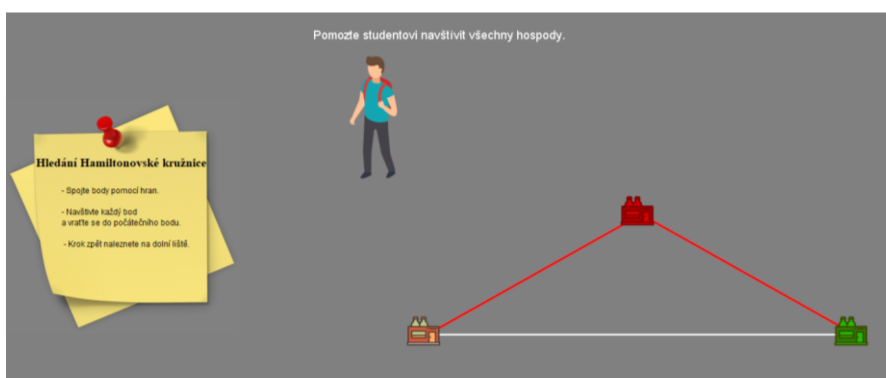
Ovládání je založeno na kreslení cesty přes vrcholy v zakresleném grafu. Počáteční vrchol cesty si může uživatel libovolně zvolit. Cestu, kterou mezi dvěma

vrcholy chce vyznačit, provede tahem myši z jednoho vrcholu na druhý. Vyznačené cesty uživatele jsou znázorněné červeně. Volné cesty jsou vykresleny bílou barvou. Vrcholy jsou rozděleny barevně následovně (Obrázek 27):

1) Výchozí (bez zvýraznění) – nenavštívený vrchol
2) Zelenou – vrchol, ve kterém se aktuálně nacházíme
3) Červenou – vrchol, který byl již navštíven

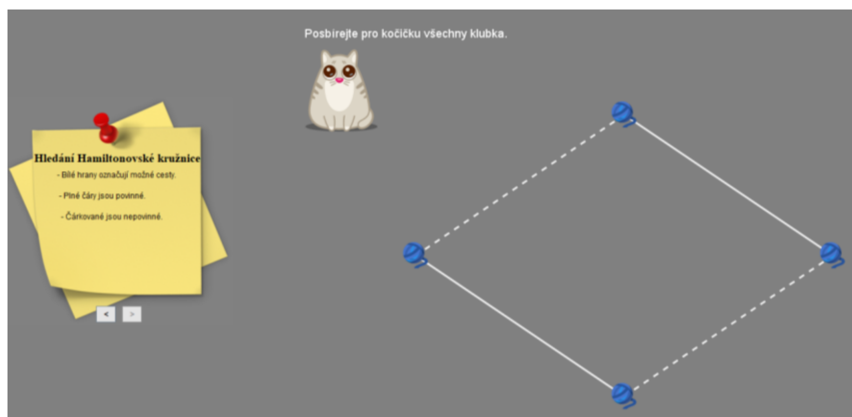
Obrázek 27 – Barevné varianty vrcholů grafu

Hra obsahuje devět úrovní. Jakmile hráč najde správně hamiltonovskou kružnici, posouvá se o úroveň výše se složitějším grafem. Po splnění všech devíti úrovní hra končí a hráč je přesunut do hlavního menu aplikace. Každá úroveň je založena na nějakém jednoduchém problému, jak ukazuje Obrázek 28, kde je vidět úkol „Pomozte studentovi navštívit všechny hospody“.



Obrázek 28 – Hra Hamilton – příklad 1. úrovně

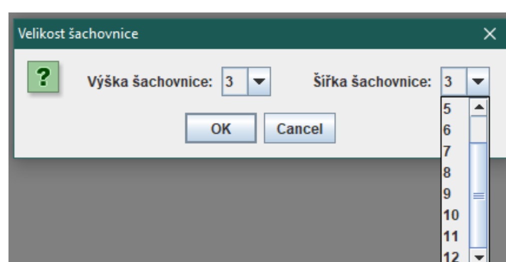
Od druhé úrovně je odemknut nový typ cest a to nepovinná cesta. Tato cesta je čárkovaná a pro splnění hlavního cíle hry není nutné tuto cestu projet. Nicméně v některých případech se jejich navštívení stejně nelze vyhnout. Ukázkovým příkladem je hned druhá úroveň hry Hamilton sbírání klubek pro smutnou kočičku (Obrázek 29). Jsou dány dvě povinné a dvě nepovinné cesty, každopádně hráč musí projet všechny čtyři cesty, aby splnil cíl hry a vrátil se i do výchozího vrcholu grafu.



Obrázek 29 - Hra Hamilton – příklad 2. úrovně

6.9 *Generování řešení*

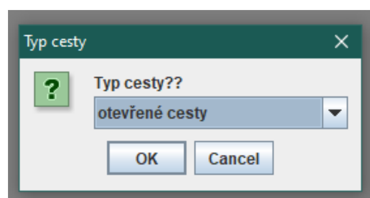
Funkce spočívá v generování různých řešení matematické úlohy jezdcovy procházky. Uživatel si na začátku zvolí algoritmus. Na výběr je ze dvou možných – Backtracking a Warnsdorffův algoritmus. Poté vybere velikost šachovnice, pro kterou chce řešení generovat (Obrázek 30). Avšak nelze zadat libovolnou velikost šachovnice. Po výběru velikosti totiž dojde k validaci vstupů, které uživatel zadá. Šachovnice nesmí být menší než 5x5, jelikož na menších řešení neexistují. Dále pokud šachovnice je větší než 35x35, v případě Warnsdorffova algoritmu, nebo větší než 6x6, v případě Backtrackingu, je zobrazená varující hláška o delší době výpočtu řešení. Pokud uživatel zadá šachovnici větší než 8x8 algoritmu Backtrackingu, zobrazí se chybové hlášení, že u větších šachovnic nedojde algoritmus v brzké době k řešení.



Obrázek 30 – Volba velikosti šachovnice

Po zvolení velikosti šachovnice musí ještě uživatel zvolit, jaký typ cest chce generovat, zda otevřené či uzavřené (Obrázek 31). V případě, kdy uživatel zadá

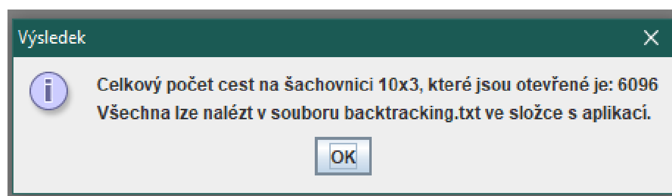
šachovnici lichých rozměrů (např. 5x5) pak nemá uživatel na výběr a může nechat generovat pouze otevřená řešení, jelikož na těchto šachovnicích uzavřená řešení neexistují. V poslední fázi uživatel zadá, kolik řešení chce vygenerovat.



Obrázek 31 – Volba typu cest pro generování

Poté začne samotné generování. Tato úloha může trvat delší dobu, jelikož algoritmus hledá veškeré možné cesty v zadané šachovnici pomocí algoritmu Backtrackingu, který nepatří mezi nejefektivnější algoritmy, ale zato je velmi jednoduchý.

Výsledkem generování je informativní okénko, které na uživatele vyskočí, jakmile jsou prohledány všechny možné cesty na zadané šachovnici (Obrázek 32). Tato zpráva uživatele informuje, kolik řešení bylo vygenerováno a také o tom, že všechna tato řešení může uživatel najít v souboru umístěným ve složce s aplikací CheGra.



Obrázek 32 – Výsledek generování cest na šachovnici 10x3

V případě zvoleného Backtracking algoritmu budou řešení vypsána v souboru „backtracking.txt“, naopak u Warnsdorffova algoritmu budou uloženy v souboru „warnsdorff.txt“. Samotné soubory obsahují popis, jak jsou řešení vypsána a dále už výpis jednotlivých řešení, kde každé je originální, viz Obrázek 33.

```
Tento soubor obsahuje všechna možná řešení vygenerovaná algoritmem Backtracking.
Řešení jsou vypsána v podobě čísel ve tvaru velikosti šachovnice 10x3.
Každé číslo reprezentuje krok jezdcovy procházky.
Číslo 0 je políčko, kde jezdec začíná
Číslo 1 je políčko, kam se jezdec přesunul z kroku 0 atd...

1. řešení (Backtracking)
0 21 24 15 18 3 28 11 6 9
23 16 19 2 25 14 5 8 27 12
20 1 22 17 4 29 26 13 10 7

2. řešení (Backtracking)
0 27 24 15 18 3 20 11 6 9
25 16 29 2 23 14 5 8 21 12
28 1 26 17 4 19 22 13 10 7

3. řešení (Backtracking)
```

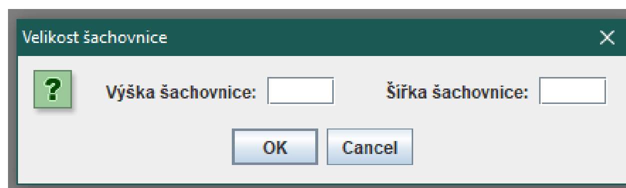
Obrázek 33 – Soubor obsahující vygenerovaná řešení

6.10 Generování neuronovou sítí

Slouží pro hledání řešení jezdcovy procházky s využitím Hopfieldovy neuronové sítě. Princip je velmi podobný funkci generování řešení. Uživatel nejdříve zadá do formuláře (Obrázek 34), kolik řešení požaduje a poté určí velikost šachovnice, ve které chce řešení najít, přičemž musí dodržet pravidla zadávání, na které upozorňuje aplikace. Pravidla jsou následující:

- Vloženým znakem musí být kladné sudé číslo různé od nuly.
- Rozměry šachovnice nesmí být obě lichá čísla, alespoň jeden rozměr musí být sudý.
- Šachovnice musí být větší než 6x6, u menších žádná řešení neexistují.

Čísla mohou být různě veliká, ale z testů, které byly provedeny v kapitole Testování aplikace, je známo, že algoritmus je efektivní pouze do velikosti šachovnice 24x24. U větších šachovnic bylo jedno řešení dosaženo průměrně za necelou půl hodinu. Z tohoto důvodu je uživatel při zadávání upozorněn. Zadá-li šachovnici větší než 19x19, je varován, že nalezení řešení může trvat několik minut, ale zadá-li šachovnici větší než 23x23, je výrazně varován na fakt, že nalezení řešení může trvat řádově v hodinách. Algoritmus dokáže vyhledat pouze uzavřená řešení šachovnic.

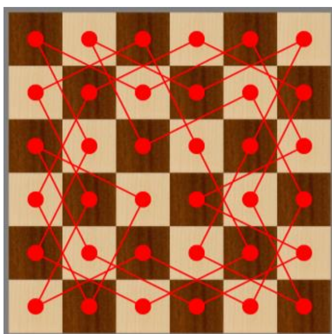


Obrázek 34 – Styl zadávání velikosti šachovnice

Poté uživatel zadá, kolik řešení chce najít a může začít generování. U vkládání počtu řešení, kolik chce uživatel vygenerovat, je upozorněn, že tato úloha může trvat velmi dlouho, v případě, kdy zadá velký počet řešení nebo šachovnici o velké velikosti (viz kapitola Testování aplikace).

Při tomto generování je jezdec ve výchozím stavu umístěn na první pole šachovnice, tedy do levého horního rohu. Jakmile je generování dokončeno, uživatel je informován zprávou o tom, kolik bylo řešení vygenerováno a také, že tato řešení nalezne v souboru „solutions.txt" v adresáři, kde je umístěna aplikace.

Výstupem je rovněž graficky vykreslená ukázková jezdcova procházka na šachovnici dané velikosti, ve které není počáteční ani koncový bod cesty vykreslen, jelikož se jedná v tomto případě o bezvýznamnou informaci, viz Obrázek 35.



Obrázek 35 – Grafický výstup generování pomocí neuronové sítě

Tento způsob hledání jezdcovy procházky je sice efektivnější než Backtracking, nicméně jeho složitost je exponenciální, a tak délka jeho výpočtu u větších šachovnic je nereálná i s využitím výkonných počítačů dnešní doby.

7 Implementace

V této části budou popsány jednotlivé použité algoritmy a zajímavé implementace, kterých bylo využito při vytváření aplikace CheGra. Aplikace je naprogramována v jazyce JAVA s použitím potřebných knihoven v programu Eclipse [46]. Autor dílo verzoval a sdílel na portál GitHub, kde je poskytnuto veřejnosti [47].

7.1 Okno aplikace

Aplikace je ve výchozím stavu nastavena na fullscreen. To znamená, že aplikace se přizpůsobí velikosti rozlišení monitoru a je tedy roztáhlá přes celou obrazovku uživatele. Veškeré objekty jsou v aplikaci umístěny relativně, nikoli fixně. Díky tomu má aplikace responzivní design a lze ji tak bez problému používat na většině počítačích.

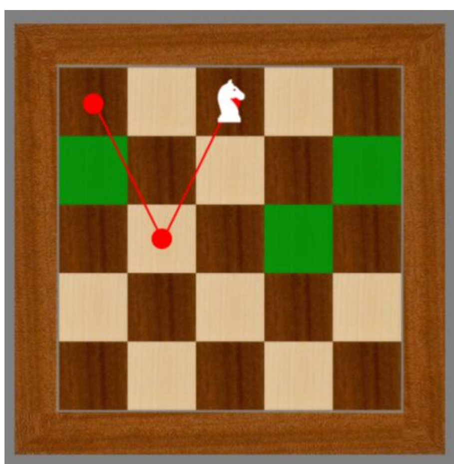
Celé okno aplikace implementuje rozhraní MouseListener a MouseMotionListener, která umožňují programu reagovat na práci uživatele myší (kliknutí, tažení…). Zaručují tak funkčnost ovládání myší.

Okno obsahuje JComponentu nazvanou Canvas, která se skládá z Serializable třídy Image. Může se to zdát býti složitější, ale opak je pravdou. Image si lze představit jako kontejner obsahující jednotlivé grafické položky, které se mají vykreslit. Tím může být čára, tečka či celý obrázek. Canvas zase zajišťuje samotné vykreslení těchto grafických objektů do okna aplikace. O vykreslení se stará třída Graphics2D již zmíněné komponenty Canvas. Jedná se o velmi rychlou a jednoduchou variantu kreslení, která je postačující. Aplikace nevyužívá žádných vláken. Tento způsob by byl zbytečný a náročný, z tohoto důvodu je plátno překreslováno pouze po určitém uživatelském úkonu. Pro hladké vykreslení grafických objektů je zapnuta funkce antialiasingu. Jedná se o funkci zamaskování nebo odstranění aliasu. Alias je chyba při vzorkování spojité funkce. Výsledkem chyby je schodovité vykreslení šikmých čar.

Okno dále obsahuje horní a dolní lištu nabízející další ovládací prvky aplikace. Tyto lišty jsou typu JMenuBar, kde horní lišta je v rozmístění umístěna na sever a dolní na jih.

7.2 Grafické objekty

Okno slouží ke grafické reprezentaci dvou odlišných tříd objektů – vrcholů a hran. Na Obrázek 36 lze vidět dvě hrany (červené cesty koně), všechno ostatní je typu vrcholů (Vertex).



Obrázek 36 – Příklad Vertexů a Edgů

Třída „Vertex" slouží především k bodové reprezentaci. Patří sem jak vykreslení jednoduchých grafických elementů (body), tak i ty složité (externí obrázky). Pomocí této třídy je například vykreslena celá šachovnice, kde každé pole šachovnice je reprezentováno jednou instancí třídy Vertex. Pomocí této třídy jsou dále vykresleny i grafické texty, koně či vrcholy ve hře Hamilton. Jedná se tedy o třídu, která umožní vzít nějaký grafický objekt a vykreslit ho na určené místo. Jeho hlavními atributy jsou: pozice na plátně (souřadnice X a Y), případně barva a velikost. Barva a velikost jsou po spuštění aplikace nastaveny na defaultní hodnoty, které mohou být změněny za běhu aplikace.

Třída „Edge" slouží především k reprezentaci hran a cest. Hrany jsou definovány dvěma vertexy (počáteční a koncový), barvou a tloušťkou hrany. Na rozdíl od bodové reprezentace Vertexu se jedná o grafickou reprezentaci, která je

dána dvěma body (vertexy). Počáteční a koncový vertexy nemohou být jeden a ten samý vrchol. Barva a tloušťka hrany jsou po spuštění aplikace nastaveny na defaultní hodnoty, které opět mohou být za běhu aplikace změněny.

7.3 Hlavní menu

První, čeho si uživatel po spuštění aplikace všimne, je hlavní nabídka. Jedná se nové okno typu JFrame, kterému je nastavena vysoká priorita, to znamená, že je umístěno před vším ostatním co nejblíže k uživateli a nelze používat zbytek aplikace, dokud toto hlavní menu bude otevřené. Cílem hlavní nabídky je snadná navigace uživatele v aplikaci. Proto byl dán zřetel na její jednoduchost a přehlednost.

Jeho velikost je fixní (672x470 pixelů) a nelze ji měnit. Ostatní úkony jsou však dostupné, jako přemístění nebo minimalizace okna. Hlavní nabídka se skládá z jednotlivých tlačítek pro zapnutí daných funkcí/her. Tato tlačítka jsou typu JButton.

7.4 Informační cedulka (nápověda)

Jedná se sice o detail, nicméně je to jedna z nejdůležitějších částí aplikace. Jejím cílem je totiž uživateli vysvětlit, co má dělat, respektive, jaká jsou pravidla ve zvolené hře.

Z pohledu implementace se jedná o JTextArea, která se skládá z dalších instancí typu JTextArea a také z instancí typu JTextField. Tyto části slouží pro zobrazení samotného textu nápovědy. Instance JTextField jsou použity jako nadpisy a instance JTextField jako popisující text.

Nápověda obsahuje pro každou hru právě jeden nadpis a popisující text. V závislosti na zvolené hře nápověda pouze přepíná, který nadpis a který popisující text se má v ní vykreslit. Výjimku tvoří pouze hra Hamilton, která má dvě strany popisujícího textu. Tudíž bylo nutné pro tento případ vytvořit dvojici tlačítek typu JButton pro listování v tomto popisujícím textu. Tato tlačítka se uživateli zobrazují pouze tehdy, kdy je spuštěna hra Hamilton a hráč se nachází v druhé či vyšší úrovni (Obrázek 37). Na první úrovni hry Hamilton je vykreslena pouze první strana

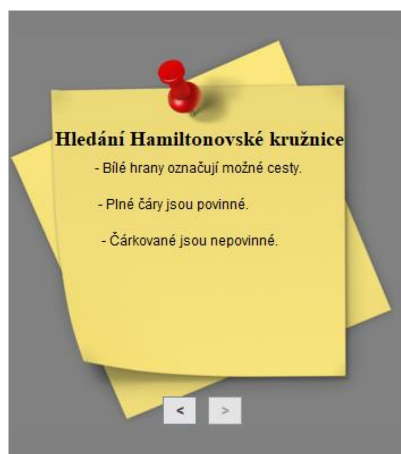
nápovědy, jelikož druhá strana obsahuje informace důležité pouze v pozdější fázi hry.

V některých hrách a funkcích jsou pod cedulkou umístěna i jiná tlačítka typu JButton. Konkrétně tomu tak je:

1) Najdi procházku – tlačítko „Ukaž řešení“
2) Krycí hra – „Ukaž řešení“
3) Simulace – „Zkusím to sám“

Tato tlačítka mají pouze jedinou funkci – přepnout hráče z jedné hry/funkce do jiné. Výjimkou je pouze krycí hra, kde toto tlačítko hráče nikam nepřepne, nýbrž za něj umístí koně do šachovnice tak, jak by mohlo vypadat jedno z možných řešení.

Vzhled nápovědy, aby vypadala jako připíchnutá cedulka, je proveden přepsáním původní metody paintComponent, kdy pro vykreslení této komponenty není využito výchozího nastavení, nýbrž byl použit lokální obrázkový soubor, který byl načten do aplikace jako instance typu BufferedImage. Ten již umožní metodou drawImage vykreslení obrázku do této komponenty.



Obrázek 37 – Informační cedulka ve hře Hamilton (2. úroveň)

7.5 Implementace – Popis problému

Jak už bylo řečeno, popis problému je rozdělen na část teoretickou a část grafickou. V implementaci jsou označeny jako TheoryTable a TheoryPictures. Může se zdát, že jsou velice odlišné, faktem ale je, že obě dědí z třídy JTextArea. Jedná se

o komponenty, které do sebe umožňují psát text, a jejich pozadí může být editováno. Toho také autor využil. Teoretická část (TheoryTable) obsahuje text, který popisuje danou problematiku, a její pozadí je nastaveno fixně na obrázek tabule. Kdežto grafická část (TheoryPictures) text neobsahuje, nicméně její pozadí mění obrázky v závislosti, na jaké stránce teorie se uživatel nachází.

Na poslední straně teoretické části jsou zobrazeny dvě speciální JButton tlačítka. „Chci to zkusit“ ukončí „Popis problému“ a spustí hru „Najdi procházku“ pro šachovnici 5x5. Tlačítko „Spustit ukázku“ pouze spustí cyklus, který vykreslí jeden krok jezdcovy procházky, následně se na 0.5 sekundy pozastaví a poté vykreslí další krok. Takto pokračuje, dokud nevykreslí všechny kroky jezdcovy procházky. Během tohoto cyklu je uživateli v aplikaci všechno nepřístupné, jelikož by mohlo dojít k chybě programu.

7.6 Implementace – Najdi procházku

U hry „Najdi procházku“ je kladen důraz především na jednoduché ovládání. Uživatel může koně posouvat tahem myši či kliknutím na zeleně zvýrazněné pole šachovnice. Využívá se tedy rozhraní MouseListener a MouseMotionListener. Zeleně zvýrazněná pole jsou v každém kroku aktualizována vzhledem k aktuální pozici koně a vzhledem k pravidlu pohybu šachového jezdce (do písmena L).

Během hraní se ukládá pohyb koně. Jakmile jsou všechna pole šachovnice navštívena, hra končí, hráč správně našel jezdcovu procházku a aplikace následně informuje hráče o této skutečností pomocí informačního okna. K jinému závěru dojít nemůže, jelikož kůň nelze umístit na pole, na které nemůže vstoupit. Platí, že kůň nemůže být umístěn na pole, které bylo již navštíveno nebo které je nedosažitelné dle pravidel jeho pohybu.

Jak bylo řečeno, cesta, kterou hráč provádí, se ukládá. Konkrétně do ArrayListu Vertexů. Jedná se o kolekci, která nemá pevně danou délku. Díky této kolekci lze provádět důležitý ovládací prvek této hry – „Krok zpět“. Toho je docíleno smazáním posledního vloženého kroku v Listu obsahující cestu koně. Po tomto úkonu stačí pouze aktualizovat umístění koně na předchozí pozici včetně aktualizování zeleně zvýrazněných polí, kudy se může z této pozice vydat. V případě, že si hráč neví rady,

jak hru dokončit, může si zobrazit řešení JButton tlačítkem „Ukaž řešení“, které hráče přesune do „Simulace“. Po úspěšném nalezení jezdcovy procházky se uživateli zobrazí informativní okno JOptionPane typu INFORMATION_MESSAGE.

7.7 Implementace – Simulace

Simulace hledá jezdcovu procházku na zadané šachovnici pomocí algoritmu Backtracking. Šachovnice je reprezentována ArrayListem, kde každé pole má svoje id (identifikační číslo). Je-li šachovnice velikosti NxN pak ArrayList obsahuje celkem n^2 polí. Samotnou implementací Backtrackingu se práce zabývá v kapitole 8 - Implementace algoritmů. Tento algoritmus se provádí při každém spuštění „Simulace“ pro danou šachovnici. Jakmile algoritmus jezdcovu procházku nalezne, je následně uživateli vykreslena na šachovnici pomocí červených cest.

7.8 Implementace – Krycí hra

Krycí hra je velmi podobná hře „Najdi procházku“, co se týče implementace. V tomto případě ale je možné koně umístit na jakékoli pole, které není zdí. Nemusí se tedy při posouvání koně dodržovat pravidlo pohybu do písmena L. Uživatel se tak může s koněm pohybovat dle své libosti. Nicméně jak už bylo zmíněno, výjimku tvoří zdi, které se nacházejí ve variantách „Level 1“, „Level 2“ a „Level 3“. Zdi mají specifickou texturu. Lze je tak jednoduše odlišit od jiných polí šachovnice.

V případě, že si hráč neví rady, jak hru dokončit, může si zobrazit řešení JButton tlačítkem „Ukaž řešení“, které za hráče level vyřeší umístěním koní na předem definovaná místa. Po úspěšném nalezení jezdcovy procházky se uživateli zobrazí informativní okno JOptionPane typu INFORMATION_MESSAGE.

7.9 Implementace – Hra Hamilton

Tato hra nabízí celkem devět úrovní. Průběh hrou je naimplementován tak, že po každé splněné úrovni je hráč přesunut na další úroveň, a tak se mu vykreslí nový graf pro řešení. Grafy jsou vždy vykresleny k nějakému tématu či problému.

Jednotlivé grafické obrázky/objekty jsou načteny z lokálních souborů a v implementaci jsou reprezentovány jako typ BufferedImage, který právě umožňuje vykreslení těchto obrázků na obrazovku uživatele.

Hra Hamilton využívá především práce s myší. Z tohoto důvodu je zde využíváno dvou rozhraní a to MouseListener a MouseMotionListener. Díky těmto rozhraním může hráč provádět na grafu cestu a splnit tak cíl hry.

Po slnění úkolu dané úrovně hry vyskočí na uživatele oznamovací okénko o úspěšně zvládnutém levelu. Stejně tomu tak je i na konci hry, kdy hráč splnil všech devět úrovní, s tím rozdílem, že po zavření tohoto okna je hráč přesunut do hlavního menu aplikace. Opět se jedná o okno JOptionPane typu INFORMATION_MESSAGE.

7.10 Implementace – generování řešení

Tato funkce vygeneruje uživatelem zadaný počet řešení jezdcovy procházky, která jsou navzájem různá pro zadanou šachovnici.

K samotnému generování lze využít algoritmu Backtracking či Warnsdorffova. Principy obou algoritmů budou probrány v kapitole 8 – Implementace algoritmů.

Výsledkem těchto generování je grafický výstup pro uživatele aplikace, který informuje o tom, kolik řešení bylo na šachovnici nalezeno. K tomu využívá okno JOptionPane typu INFORMATION_MESSAGE, které také uživatele informuje, že jeho vygenerovaná řešení jsou vypsána v souboru umístěném v adresáři s aplikací.

Tento souborový výstup slouží pro uživatele, aby si mohl řešení zkontrolovat a dále s nimi pracovat. Řešení jsou vypsána ve tvaru šachovnice a jsou tvořena z identifikačních čísel jednotlivých polí šachovnice, viz Obrázek 33. Soubor obsahuje veškeré informace o provedeném generování (kolik bylo vygenerováno, jakým algoritmem) a také návod, jak daná vygenerovaná data číst. Tento výpis do textového souboru byl zaručen přenastavením systémového výstupu System.setOut(), kdy cílové místo výstupu bylo z konzole nastaveno na textový soubor v adresáři s aplikací, viz kód 1.

```
PrintStream out = null;
try {
    out = new PrintStream(new FileOutputStream("backtracking.txt"));
} catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
System.setOut(out);
```

Kód 1 – Nastavení systémového výstupu na soubor backtracking.txt

Jednou z důležitých vlastností této funkce je možnost generovat zvlášť otevřené a uzavřené cesty. Lze tak jednoduše určit, které varianty řešení chce uživatel vygenerovat. Implementace tohoto rozdílu spočívá především v počáteční a koncové lokalizaci šachového jezdce. Začíná-li jezdec na jednom poli a končí na odlišném, jedná se o otevřenou cestu.

U hledání uzavřených řešení se postupuje velmi podobně. Jedinou výjimkou je skutečnost, že za správné řešení se považuje ta možnost, která začíná v jednom poli a končí v jiném poli, které je přesně jeden tah vzdálené od počátečního pole, ze kterého jezdec vyjížděl. Ve výsledku každé uzavřené řešení je i otevřeným řešením, rozdíl mezi nimi je pouze jeden krok navíc u uzavřeného řešení, který umístí jezdce na počáteční pole.

V obou případech je počáteční pole jezdce během generování postupně měněno skrz celou šachovnici. To zaručí nalezení všech otevřených cest, ať už jezdec začíná na kterémkoli poli šachovnice. U uzavřeného řešení dochází k nalezení jedné cesty dvakrát. Je to z toho důvodu, že uzavřené řešení je vlastně kružnice, tedy lze ji nalézt dvěma směry, přičemž se jedná o jednu a tu samou cestu. Proto je celkový počet uzavřených řešení ve finále dělen dvěma.

7.11 Implementace neuronové sítě

Po spuštění neuronové sítě se uživateli nabídne možnost zadat, kolik řešení se má vygenerovat a poté možnost určit velikost šachovnice, přičemž počty řádků a sloupců nesmí být obě lichá čísla (např. 7x15), alespoň jeden rozměr musí být sudý (7x14, 16x16...). Neuronová síť vyhledává pouze uzavřená řešení a ty neexistují na šachovnicích lichých velikostí.

Po zadání všech těchto povinných údajů začne samotný proces generování, po jehož skončení je uživatel informován zprávou okna JOptionPane typu INFORMATION_MESSAGE, kolik řešení bylo vygenerováno a že tato řešení lze nalézt v souboru „solution.txt“ v adresáři s aplikací. Tento výpis je proveden stejným stylem jako výpis řešení funkce generování (viz kapitola Implementace – generování řešení).

Pro neuronovou síť byla vybrána Hopfieldova architektura. Tato síť je reprezentována tak, že její velikost odpovídá velikosti šachovnice, každý tah šachového jezdce je reprezentován neuronem, který je inicializován náhodně, buď je aktivní, nebo neaktivní (jeho výstup je 1 nebo 0). Aktivní neuron znamená, že příslušný krok jezdce je součástí jezdcovy procházky. Každý neuron dále má svoji stavovou funkci, která je inicializována na 0. Po spuštění sítě jsou neurony aktualizovány dle pohybu jezdce. Mohou tak změnit svůj stav a výstup na základě stavů a výstupů svých sousedů. Každou iterací se tak hledají stabilní stavy všech neuronů, aby ve výsledku byla celá neuronová síť stabilní. [44]

Jakmile je síť spuštěna, je každý aktivní neuron nakonfigurován tak, aby dosáhl stabilního stavu, a to pouze tehdy, pokud má aktivní dva sousední neurony (jinak se stav neuronu změní). Když je celá síť stabilní, získá se řešení. Úplná pravidla přechodu jsou následující: [48]

$$\text{když } d_{ij} = 1 \text{ pak} \quad \frac{dU_{ij}}{dt} = -\left(\sum_{k=1}^{p} V(N_{ik})d_{ik} - 2\right) - \left(\sum_{k=1}^{p} V(N_{kj})d_{kj} - 2\right) \qquad \text{jinak} \quad \frac{dU_{ij}}{dt} = 0 \tag{1}$$

Kde d_{ij} je roven jedné, pokud existuje validní tah z pole i do pole j, jinak je roven nule, dU_{ij} je přechodová funkce a $V(N_{ij})$ je výstup neuronu od i do j. [48] Aktualizace neuronů poté probíhá následovně:

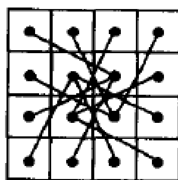
$$U_{t+1}(N_{i,j}) = U_t(N_{i,j}) + dU_{ij} \tag{2}$$

$$V_{t+1}(N_{ij}) = \begin{cases} 1 & když\ U_{t+1}(N_{ij}) > 3 \\ 0 & když\ U_{t+1}(N_{ij}) < 0 \\ V_t(N_{ij}) & jinak \end{cases} \quad (3)$$

Kde t představuje čas (přírůstek v diskrétních intervalech) a $U(N_{ij})$ je stav neuronu spojujícího pole i s polem j[48]

Zpočátku (v $t = 0$) je stav každého neuronu nastaven na 0 a výstup každého neuronu je nastaven náhodně buď na 0, nebo 1. Neurony jsou poté postupně aktualizovány počítáním polí na šachovnici v řádkové pořadí a výpočtem neuronů, které představují jezdcovy pohyby z každého pole.

Síť je v zásadě nakonfigurována tak, aby generovala podgrafy stupně 2, které jsou v rámci jezdcovy procházky. Existuje však mnoho dalších řešení, která by uspokojila síť, ale zároveň by nebyla jezdcovými procházkami. Například síť mohla objevit dva nebo více malých nezávislých cest v jezdcově procházce. Neboli jednalo by se o nesouvislé grafy a cesta by se tak skládala ze dvou či více komponent (Obrázek 38), z tohoto důvodu byla implementována kontrola, zda se jedná o souvislý graf (z každého bodu se lze dostat do libovolného druhého bodu). Kromě toho problému existují určité případy, které způsobí, že se síť odkloní (nikdy nebude stabilní).



Obrázek 38 - Stabilní síť tvořená několika komponentami (zdroj: [39])

Pravděpodobnost získání jezdcovy procházky na šachovnici NxN ve skutečnosti klesá s rostoucím N. Některé starší publikace dokonce uvádějí, že autoři získali řešení pouze pro $N < 20$, viz [48]. Nebo například Parberry dokázal získat jednu jedinou jezdcovu procházku ze 40 000 pokusů pro $N = 26$. [39]

8 Implementace algoritmů

V této kapitole budou popsány ty nejdůležitější prvky implementovaných algoritmů. Kompletní algoritmy lze najít v příloze 3, nebo je lze dohledat na GitHubu [47]. Porovnání efektivity těchto algoritmů je prezentováno v kapitole 9.

8.1 Backtracking

Implementace začíná nastavením proměnných dle velikosti šachovnice. Šachovnice je reprezentována jako dvourozměrné pole, proto je na začátku nutné nastavit každé pole na „nenavštíveno“ a poté se může spustit hledání procházky (Kód 2).

```
for (int i = 0; i < ySize; i++) {
    for (int j = 0; j < xSize; j++) {
        solutionBoard[i][j] = NOT_VISITED;
    }
}
solve();
```

Kód 2 – Inicializace pole

Metoda solve spustí algoritmus hledání jezdcovy procházky. Řešení jsou vyhledávána postupně od pole s indexem jedna až po poslední pole na šachovnici, viz Kód 3. Nalezne-li algoritmus všechna řešení z počátečního pole o souřadnicích (0,0), je zavoláno rekurzivní resetování tohoto pole a spustí se vyhledávání řešení úlohy z počátečního pole (0,1).

```
public void solve() {
    // hledání řešení od všech možných výchozích pozic jezdce
    for (int i = 0; i < ySize; i++) {
        for (int j = 0; j < xSize; j++) {
            takeTurn(j, i, 0);
            solutionBoard[i][j] = NOT_VISITED; // rekurzivní reset pole
        }
    }
}
```

Kód 3 – Spuštění algoritmu

Následuje metoda takeTurn, která provede krok jezdce na zadanou pozici. Avšak jezdec se nepohybuje zcela náhodně. Zkouší nejdřív ta pole, která jsou v kolekci uložena na prvním místě. Během provádění těchto tahů je číslo tahů ukládáno do pole solutionBoard, které slouží pro vypsání řešení do výstupu. V každém tahu jezdce se kontroluje, zda již nebyla nalezena veškerá řešení, jejichž počet uživatel zadal. V takovém případě se algoritmus přeruší. Jinak se dále pokračuje kontrolou, zda nebyla všechna pole šachovnice navštívena. Pokud ano, řešení bylo nalezeno a provede se jeho výpis do výstupu. V opačném případě se pokračuje dalším tahem. Následující tah jezdce může být proveden na pole dle pravidel šachu a také toto pole nesmí být již navštíveno. Pokud jezdec nemá kam tah udělat (všechny možnosti provedení tahu již vyzkoušel), pak se předešlý tah jezdce z řešení odstraní a volá se rekurzivní resetování příslušného pole, viz Kód 4.

```
private void takeTurn(int x, int y, int turnNr) {
    // pokud máme všechna řešení co jsme chtěli -> konec
    if (pocetReseni == solutionsCount) {
        return;
    }
    solutionBoard[y][x] = turnNr;

    // pokud bylo nalezno řešení
    if (turnNr == (xSize * ySize) - 1) {
        // podmínka pro výstup uzavřených řešení
        if (uzav) {
            for (Coords c : getFields(x, y)) {
                if (solutionBoard[c.getY()][c.getX()] == 0) {
                    podm = true;
                }
            }
        }

        printSolution(); // tisk řešení
        return;

    // pokud ještě nebylo řešení nalezeno
    } else {
        // provedení tahu na ještě nenavštívené pole dle pravidel šachu
        for (Coords c : getFields(x, y)) {
            if (solutionBoard[c.getY()][c.getX()] == NOT_VISITED) {
                takeTurn(c.getX(), c.getY(), turnNr + 1);
                // rekurzivní reset pole
                solutionBoard[c.getY()][c.getX()] = NOT_VISITED;
            }
        }
    }
}
```

Kód 4 – Rekurzivní provádění tahu jezdce

Pohyby šachového jezdce jsou definovány pomocí ArrayListu souřadnic (Coords) v následujícím Kód 5:

```
private List<Coords> getFields(int x, int y) {
    List<Coords> l = new ArrayList<Coords>();
    if (x + 2 < xSize && y - 1 >= 0)
        l.add(new Coords(x + 2, y - 1)); // doprava nahoru
    if (x + 1 < xSize && y - 2 >= 0)
        l.add(new Coords(x + 1, y - 2)); // nahoru doprava
    if (x - 1 >= 0 && y - 2 >= 0)
        l.add(new Coords(x - 1, y - 2)); // nahoru doleva
    if (x - 2 >= 0 && y - 1 >= 0)
        l.add(new Coords(x - 2, y - 1)); // doleva nahoru
    if (x - 2 >= 0 && y + 1 < ySize)
        l.add(new Coords(x - 2, y + 1)); // doleva dolu
    if (x - 1 >= 0 && y + 2 < ySize)
        l.add(new Coords(x - 1, y + 2)); // dolu doleva
    if (x + 1 < xSize && y + 2 < ySize)
        l.add(new Coords(x + 1, y + 2)); // dolu doprava
    if (x + 2 < xSize && y + 1 < ySize)
        l.add(new Coords(x + 2, y + 1)); // doprava dolu
    return l;
}
```

Kód 5 – Reprezentace pohybů jezdce

Neexistuje zde žádný speciální rozdíl mezi hledáním uzavřených a otevřených řešení. Postup je stejný, pouze u uzavřených řešení je výstup zkrácen. Vypisují se pouze ta řešení, kde koncové pole otevřeného řešení úlohy je vzdáleno jedním tahem jezdce (tedy, že se může jezdec jedním tahem vrátit na počáteční pole).

8.2 Warnsdorffův algoritmus

Jak už bylo řečeno, optimalizací Backtrackingu je Warnsdorffův algoritmus, který je založen na stejném principu jako předešlý algoritmus. Při pohybu jezdec zvýhodňuje ta pole, která jsou obtížnější k jejich dosažení. V každém tahu jezdce jsou tedy všechna pole pohybu jezdce ohodnocena. Toto ohodnocení reprezentuje, na kolik polí se lze z daného políčka vydat. Čím menší ohodnocení, tím více je pole zvýhodněno pro další krok. V případě, kdy jsou pole ohodnocena stejně, je pak vybráno to, které je v reprezentaci (ArrayList polí) na prvním místě. Inicializace všech polí a ohodnocení jejich možných kroků je popsáno v následujícím algoritmu, viz kód 2:

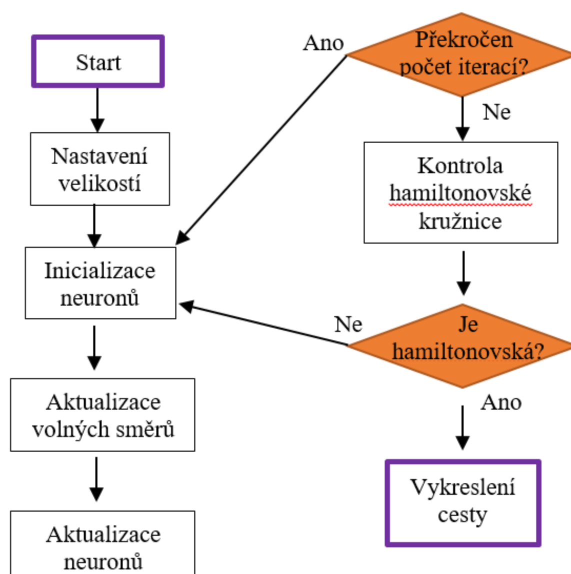
```
// pro každé pole
for (i = 0; i < n; i++){
    for (j = 0; j < n; j++){
        // pro každý směr
        for (t = 0; t < 8; t++){
            // DX a DY obsahují pohyby jezdce po ose X a Y
            u = i + DX[t];
            v = j + DY[t];
            // když se při pohybu nedostanou mimo šachovnici
            // -> zvýší se ohodnocení
            if ((u >= 0) && (u < n) && (v >= 0) && (v < n)){
                deg[i][j]++;
            }
        }
    }
}
```

Kód 6 – Ohodnocení polí (Warnsdorffovo pravidlo)

8.3 Hopfieldova neuronová síť

Neuronová síť na Hopfieldově architektuře pro jezdcovu procházku je randomizovaným algoritmem (dává různé výstupy při různých provedeních). Toho je docíleno pomocí prvotního nastavení, kdy je výchozí stav neuronu $U_0(N_{i,j})$ nastaven pomocí generátoru pseudonáhodných čísel a výchozí výstup neuronu $V_0(N_{i,j})$ je nastaven na nulu pro všechny $1 \leq i < j \leq n^2$. Standardní Hopfieldovy sítě mají zaručenou konvergenci, pokud jsou aktualizovány v sekvenčním režimu (jeden neuron najednou), ale nemusí tak tomu být, pokud jsou aktualizovány v paralelním režimu (všechny najednou). V případě této práce byla použita aktualizace neuronů postupně pro každé pole S šachovnice v řádkovém pořadí, kde neurony reprezentují pohyby z pole S. Aktualizace neuronů byly implementovány dle předešlé kapitoly Implementace neuronové sítě. Čas potřebný k aktualizaci každého neuronu v každém tahu je v oblasti neuronových sítí nazýván epochou. Algoritmus byl implementován pouze pro hledání uzavřených řešení. To znamená, že vyhledává hamiltonovskou kružnici, tedy do každého z polí šachovnice

vedou dvě cesty (hrany). Očekávalo se, že vyhledání otevřeného řešení by mělo za výsledek, že by neuronová síť nikdy nedošla do stabilního bodu. Flowchart algoritmu znázorněn na Obrázek 39.



Obrázek 39 - Flowchart neuronové sítě

9 Testování aplikace

Tato část je věnována testování vybraných algoritmů řešících jezdcovu procházku. Pro testy byly vybrány algoritmy: Backtracking, Warnsdorffův algoritmus a Hopfieldova neuronová síť.

Pro testování byl každý z algoritmů testován na následující počítačové sestavě: i7-7700K, GeForce 1060 6GB, 16GB RAM. Nejedná se o aktuálně nejsilnější sestavu, proto v době, kdy vychází tato práce, se lze setkat s mnohem výkonnějšími sestavami, které budou mít o poznání lepší výsledky.

Při testování byly zkoumány různé velikosti šachovnic, aby se dalo porovnat nejen, který algoritmus je efektivnější, ale také jak velký to má dopad na výpočet, nebo také aby se dala určit prognóza, jak se bude algoritmus chovat ve větších velikostech šachovnice.

Komplikací testování byl tak zvaný „infinity loop“, kterého bylo dosaženo u algoritmů Backtrackingu a Warnsdorffova, kdy docházelo k tomu, že jezdec se snadno dostával do slepých uliček. Konkrétně tomu tak bylo u Backtrackingu při $N > 8$ a u Warnsdorffova algoritmu při $N \geq 30$. I přes tuto komplikaci byla pomocí Warnsdorffova pravidla nalezena řešení i daleko větších šachovnic metodou pokus omyl. K tomuto problému častých slepých uliček docházelo nejspíše z důvodu kombinace volby počátečního bodu jezdcovy procházky a volby následujícího tahu, který by měl být náhodný, ale není, jelikož se jezdec rozhodne pro první možnou cestu z ArrayListu. Tuto chybu by neměl být problém vyřešit, nicméně jedná se pouze o domněnku, že bude chyba právě na tomto místě. Tak či onak jedná se o záležitost, která bude v budoucnosti řešena.

Výsledky testování Backtrackingu jsou znázorněny v tabulce 3.

	Backtracking – otevřená řešení		
N	1. řešení	20 řešení	≅ na 1 řešení
5	0,006 s	0,063 s	0,0032 s
6	0,016 s	0,112 s	0,0056 s
7	0,007 s	0,022 s	0,0011 s
8	1,793 s	8,636 s	0,4318 s

Tabulka 3 - Čas řešení jezdcovy procházky - Backtracking

Jak je z tabulky patrné, Backtracking dosáhl velmi dobrých výsledků u malých šachovnic, nicméně již u šachovnice $8x8$ se pomalu schyluje k nevýhodě tohoto algoritmu (problém slepých uliček). U šachovnice $9x9$ už došlo k „infinity loopu" a výsledku nebylo dosaženo.

V následující tabulce (Tabulka 4) lze nalézt časovou náročnost řešení jezdcovy procházky s využitím Warnsdorffova algoritmu.

	Warnsdorff – otevřená řešení		
N	1. řešení	20 řešení	≅ na 1 řešení
5	0,001 s	0,005 s	0,00025 s
6	0,001 s	0,005 s	0,00025 s
7	0,001 s	0,007 s	0,00035 s
8	0,001 s	0,008 s	0,0004 s
12	0,001 s	0,014 s	0,0007 s
23	0,001 s	0,038 s	0,0019 s
28	0,001 s	0,053 s	0,00265 s
42	0,001 s	0,114 s	0,0057 s
57	0,002 s	0,182 s	0,0091 s
68	0,002 s	0,267 s	0,01335 s
73	0,002 s	0,311 s	0,01555 s
90	0,002 s	0,45 s	0,0225 s
118	0,003 s	0,738 s	0,0369 s
143	0,004 s	1,132 s	0,0566 s
181	0,004 s	1,804 s	0,0902 s
	Warnsdorff – uzavřená řešení		
N	1. řešení	20 řešení	≅ na 1 řešení
6	0,001 s	0,022 s	0,0011 s
8	0,001 s	0,009 s	0,00045 s
10	0,001 s	0,012 s	0,0006 s
12	0,001 s	0,016 s	0,0008 s
14	0,001 s	0,022 s	0,0011 s
16	0,001 s	0,024 s	0,0012 s
18	0,001 s	0,025 s	0,00125 s
28	0,001 s	0,052 s	0,0026 s
48	0,001 s	0,149 s	0,00745 s
58	0,001 s	0,205 s	0,01025 s
72	0,001 s	0,299 s	0,01495 s
92	0,001 s	0,49 s	0,0245 s
104	9,316 s	9,892 s	0,4946 s

Tabulka 4 - Čas řešení jezdcovy procházky - Warnsdorff

Warnsdorffův algoritmus představuje zlepšení oproti klasickému Backtrackingu. A to nejen v časech hledání, ale také v možnostech hledání jezdcovy procházky na větších rozměrech šachovnice. Jak je z tabulky patrné, každé zvětšení šachovnice o další pole neovlivňuje razantně složitost výpočtu. Nicméně při testování docházelo k již zmíněnému problému častých slepých uliček, při $N > 76$.

Z tabulky není vidět výrazný rozdíl při hledání otevřeného či uzavřeného řešení a to i přes to, že vyhledání otevřeného řešení by mělo být daleko jednodušší, protože každé uzavřené řešení je zároveň otevřeným řešením. Rozdílem je pouze hrana navíc, která spojuje první pole s posledním. Výrazný nárůst výpočetní doby je vidět až při hledání uzavřeného řešení při velikosti $N = 104$.

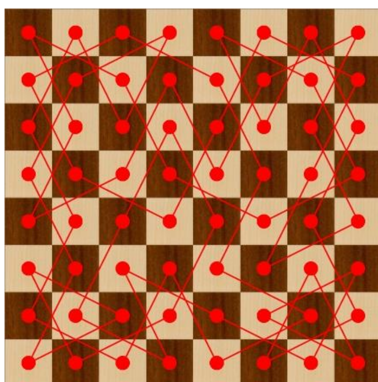
Posledním algoritmem hledající jezdcovu procházku je neuronová síť založená na Hopfieldově architektuře. Neuronová síť není nijak omezená velikostí šachovnice, nicméně dokáže hledat pouze uzavřená řešení. Výsledky testů jsou znázorněny v Tabulka 5. Příklady nalezeného řešení šachovnic 8x8 a 28x28 lze nalézt na Obrázek 40 a Obrázek 41.

	Neuronová síť		
N	1. řešení	20 řešení	≅ na 1 řešení
6	0,002 s	0,067 s	0,00335 s
8	0,022 s	0,395 s	0,1975 s
10	0,052 s	0,721 s	0,03605 s
12	0,245 s	4,372 s	0,2186 s
14	0,768 s	14,353 s	0,71765 s
16	2,765 s	47,041 s	2,35205 s
	1. řešení	3 řešení	≅ na 1 řešení
18	27 s	59 s	19,7 s
20	1 m 25 s	4 m 40 s	1 m 33 s
22	1 m 23 s	11 m 46 s	3 m 55 s
24	4 m 30 s	9 m 34 s	3 m 21 s
26	34 m 31 s	1 h 14 m	24 m 40 s
28	12 m 23 s	5 h 7 m	1 h 42 m

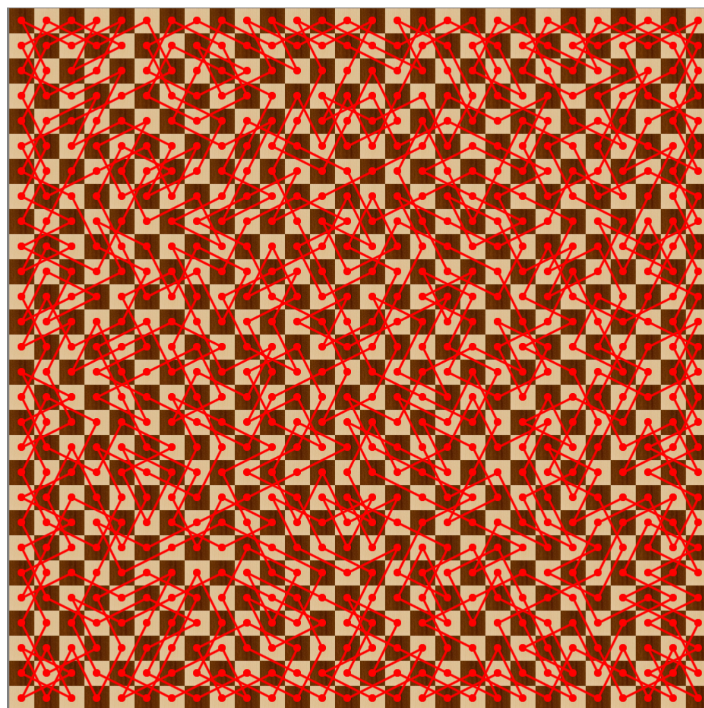
Tabulka 5 - Čas řešení jezdcovy procházky - Neuronová síť

Již na první pohled je vidět, že se časové jednotky výpočtu neuronovou sítí pohybují úplně někde jinde. Pro příklad se lze podívat na typickou šachovnici s rozměry $8x8$. Backtracking najde 20 řešení za necelých devět sekund, Warnsdorff

pod jednu setinu sekundy a Hopfieldova neuronová síť za 0,4 sekundy. Oproti backtrackingu tedy došlo ke zlepšení, bohužel neuronová síť nedosahuje tak dobrých výsledků jako Warnsdorffův algoritmus. To je způsobeno především tím, že neuronová síť je v řešení jezdcovy procházky náhodná, kdežto Warnsdorff využívá svého pravidla ohodnocení, který výrazně zrychluje výpočet.



Obrázek 40 - Jezdcova procházka (8x8) vyřešená Hopfieldovou neuronovou sítí



Obrázek 41 - Jezdcova procházka (28x28) vyřešená Hopfieldovou neuronovou sítí

10 Shrnutí výsledků

Autor si při psaní diplomové práce osvojil znalosti z oblasti matematické disciplíny teorie grafů a programování v jazyce JAVA. Největší časovou náročnost si vyžádala praktická část, kdy se bylo potřeba vypořádat s několika implementačními problémy.

Byla vytvořena aplikace, která je intuitivní a nabízí různé hry na šachovnicích, včetně popisu matematického problému jezdcovy procházky a třech možnostech generování řešení této úlohy. Program tak splňuje veškeré cíle, které byly na začátku práce určeny.

Samotná aplikace CheGra je odrazem programátorských znalostí autora. Veškeré funkcionality programu byly testovány autorem. Během testování nedocházelo k žádným chybám, avšak nějaké obsahovat může. Každý program se vyladí až průběžným používáním.

Během testování algoritmů bylo zjištěno, že nejméně efektivním algoritmem byl Backtracking, který dosahoval nejen nejhorších časových výsledků, ale také byl použitelný pouze pro malé šachovnice - velikostí nanejvýše 8x8. Druhým testovaným algoritmem byl Warnsdorffův algoritmus. Tato metoda dosahovala nejlepších výsledků ze všech testů. Algoritmus byl použitelný pro šachovnice do velikosti 76x76 a jeho časová náročnost výpočtu byla malá. Poslední algoritmus založený na Hopfieldově neuronové síti dostával velmi příjemné výsledky u menších šachovnic, avšak díky jeho exponenciální složitosti, byl velmi neefektivní u větších šachovnic (větších než 24x24), kde jedno řešení bylo získáno řádově v hodinách.

Aplikace nevyžaduje velké nároky na vybavení počítače. Stačí mít na počítači nainstalovaný software JAVA minimální verze 8.

11 Závěry a doporučení

Cíl práce byl splněn, byla vypracována teoretická studie hamiltonovských grafů. Současně byla vytvořena aplikace pro podporu výuky předmětů DIMA a DMO na FIM UHK a jako účelná aktivita na různých aktivitách fakulty. Práce popisuje jak použité algoritmy, tak i rozvržení tlačítek v aplikaci pro přehledné ovládání.

Pro popis problematiky hamiltonovských grafů autor vybral matematickou úlohu jezdcovy procházky. Tímto problémem se zabýval během celé práce, včetně jeho možností řešení.

Byly provedeny veškeré očekávané testy, z kterých se dal usoudit závěr nad jednotlivými algoritmy. U všech naimplementovaných algoritmů je třeba ještě zapracovat na optimalizaci, tak aby bylo dosaženo ještě lepších výsledků.

Ze všech algoritmů bylo nejzajímavější řešení pomocí Hopfieldovy neuronové sítě. Bohužel jeho výsledky v testech ověřily skutečnost, proč tento algoritmus není využíván v praxi při řešení problému jezdcovy procházky a jiných podobných úlohách. U neuronové sítě sice došlo k zrychlení výpočtu vzhledem k algoritmu Backtrackingu a k zlepšení možností výpočtu větší šachovnice, nicméně v porovnání s Warnsdorffovým algoritmem značně zaostává. Toto jenom potvrzuje, proč Warnsdorffův algoritmus zůstává světovým standardem při řešení takovýchto matematických problémů.

Cíl, který byl stanoven pro diplomovou práci, byl splněn. Teorii grafů, konkrétně hamiltonovské grafy, si autor vybral z důvodu předchozích zkušeností. Autor v něm našel zalíbení již při studiu předmětu Diskrétní matematika na Univerzitě Hradec Králové, oboru aplikovaná informatika.

12 Seznam použité literatury

[1] L. Jirovský, „Teorie grafů", Bakalářská práce, Univerzita Karlova, Praha, 2008.

[2] R. Krátký, „Hradec Králové - Pražské Předměstí", *Google Maps*. https://goo.gl/maps/2KWEejHVywD2.

[3] E. Hillebrandová, „On-line hry a teorie grafů", Bakalářská práce, Univerzita Karlova, Praha, 2018.

[4] P. Kovář, „Teorie grafů". 2019, [Online]. Dostupné z: http://homel.vsb.cz/~kov16/files/skriptum_teorie_grafu_rozsirene.pdf.

[5] „Leonard Euler's Solution to the Konigsberg Bridge Problem | Mathematical Association of America". https://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem (viděno úno. 28, 2020).

[6] „Město Královce". http://www.hs-augsburg.de/~harsch/germanica/Chronologie/18Jh/Zedler/zed_15k1.jpg (viděno bře. 27, 2020).

[7] E. Milková, Univerzita Hradec Králové, a Fakulta informatiky a managementu, *Teorie grafů a grafové algoritmy*. Hradec Králové: Gaudeamus, 2013.

[8] A. Ševčíková a E. Milková, *2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI).* S.l.: IEEE., 2016.

[9] J. Šitina, „Grafové algoritmy a jejich využití", Diplomová práce, Univerzita Hradec Králové, 2010.

[10] A. Hübner, „Aplikace pro podporu výuky předmětu Disktrétní matematika", Bakalářská práce, Univerzita Hradec Králové, 2014.

[11] M. Růtová-Šťastná, „Vizuální podpora výuky předmětů zabývajících se teorií grafů a grafovými algoritmy", Diplomová práce, Univerzita Hradec Králové.

[12] R. Krátký, „Skóre grafu", Univerzita Hradec Králové, 2018.

[13] „The Java Language Environment". https://www.oracle.com/technetwork/java/intro-141325.html (viděno úno. 28, 2020).

[14] „Photo Editor : Pixlr X - free image editing online", *Pixlr*. https://pixlr.com/x/ (viděno bře. 29, 2020).

[15] J. Matoušek a J. Nešetřil, *Kapitoly z diskrétní matematiky*. Praha: Karolinum, 2000.

[16] M. Mareš a T. Valla, *Průvodce labyrintem algoritmů*, 2007. vyd. Praha.

[17] "Ukázky her The Icosian Game", Dostupné z: https://www.puzzlemuseum.com/month/picm02/200207icosian.htm.

[18] T. Hulcová, „Hamiltonovské kružnice v Kneserových grafech", Bakalářská práce, Univerzita Karlova, Praha, 2017.

[19] prof. RNDr. J. Hynek, MBA, Ph.D., „Teoretická informatika - Teorie vyčíslitelnosti", Hradec Králové.

[20] M. Demlová, *Teorie algoritmů*, 2020. vyd. .

[21] J. Lomitzki, „Řešení problému splnitelnosti booleovské formule (SAT)", Bakalářská práce, České vysoké učení technické v Praze, Praha, 2008.

[22] S. Kaminani, *Finding Hamiltonian Cycles*, 2005. vyd. Western Kentucky University.

[23] J. Hastík, „Využití metod operačního výzkumu pro optimalizaci trasy při zásobování", Technická univerzita Ostrava, Ostrava, 2008.

[24] „Business Efficiency", Dostupné z: http://www.math.wisc.edu/~robbin/141dir/propp/COMAP/Guidefor1stTimeInstructors/c_FAPP07_FTI_02.pdf .

[25] L. Jurčík, „Evoluční algoritmy při řešení problému obchodního cestujícího", Diplomová práce, Vysoké učení technické v Brně, Brno, 2014.

[26] E. Fox a C. Guestrin, „Complexity of brute force search - Nearest Neighbor Search | Coursera", *Coursera*, 2018. https://www.coursera.org/lecture/ml-clustering-and-retrieval/complexity-of-brute-force-search-5R6q3 (viděno bře. 27, 2020).

[27] J. Žerovnik, „Heuristics for NP-hard optimization problems - simpler is better!?", *Logist. Sustain. Transp.*, roč. 6, č. 1, s. 1–10, lis. 2015, doi: 10.1515/jlst-2015-0006.

[28] A. Večerka, „GRAFY A GRAFOVÉ ALGORITMY". Univerzita Palackého v Olomouci", *Právnická Fak.*, 2007, [Online]. Dostupné z: http://phoenix.inf.upol.cz/esf/ucebni/grafy_a_grafove_al.pdf.

[29] K. R. Seeja, „HybridHAM: A Novel Hybrid Heuristic for Finding Hamiltonian Cycle", *J. Optim.*, roč. 2018, s. 1–10, říj. 2018, doi: 10.1155/2018/9328103.

[30] L. Du, „An Efficient Algorithm for Hamilton Cycle Based on the Enlarged Rotation-Extension Technique", *Hong Kong*, s. 4, 2014.

[31] „Eulerovské a hamiltonovské grafy". https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=23498 (viděno úno. 28, 2020).

[32] M. Pranav, S. Nithin, a N. Guruprasad, „A Comparison of Warnsdorff's Rule and Backtracking for Knight's Tour on Square Boards", in *Emerging Research in Electronics, Computer Science and Technology*, roč. 545, V. Sridhar, M. C. Padma, a K. A. R. Rao, Ed. Singapore: Springer Singapore, 2019, s. 171–185.

[33] H. Cancela a E. Mordecki, „On the number of open knight's tours", s. 4.

[34] „Knight Graph", *Wolfram MathWorld*, 1999 2020. http://mathworld.wolfram.com/KnightGraph.html.

[35] A. J. Schwenk, „Which Rectangular Chessboards Have a Knight's Tour?", *Math. Mag.*, roč. 64, č. 5, s. 325, pro. 1991, doi: 10.2307/2690649.

[36] „The Knight's tour problem | Backtracking", *GeeksforGeeks*. https://www.geeksforgeeks.org/the-knights-tour-problem-backtracking-1/.

[37] „Warnsdorff's algorithm for Knight's tour problem", *GeeksforGeeks*. https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/.

[38] S. L. Marateck, „How good is the Warnsdorff's knight's tour heuristic?", *ArXiv08034321 Cs*, bře. 2008, Viděno: dub. 28, 2020. [Online]. Dostupné z: http://arxiv.org/abs/0803.4321.

[39] I. Parberry, „Scalability of a neural network for the knight's tour problem", *Neurocomputing*, roč. 12, č. 1, s. 19–33, čvc. 1996, doi: 10.1016/0925-2312(95)00027-5.

[40] J. Trejbal, „Implementace neuronových sítí v objektově orientovaném jazyce", Diplomová práce, Univerzita Hradec Králové, Hradec Králové, 2016.

[41] M. Macůrek, „Šachy a umělá inteligence", Diplomová práce, Vysoké učení technicné v Brně, Brno, 2019.

[42] "Grafické znázornění neuronu", Dostupné z: https://www.elektroprumysl.cz/images/stories/Kategorie/Technologicke_novinky/Siemens/Neuronova_sit_1.jpg.

[43] T. Urbanová, „Neuronové sítě pro ovládání robotických systémů", Bakalářská práce, Univerzita Hradec Králové, Hradec Králové, 2016.

[44] R. Li, J. Qiao, a W. Li, „A modified hopfield neural network for solving TSP problem", in *2016 12th World Congress on Intelligent Control and Automation (WCICA)*, Guilin, China, čer. 2016, s. 1775–1780, doi: 10.1109/WCICA.2016.7578744.

[45] R. Singh a S. Kansal, „Performance evaluation of neural network based spectrum sensing in cognitive radio", in *2016 International Conference on Internet of Things and Applications (IOTA)*, Pune, India, led. 2016, s. 368–372, doi: 10.1109/IOTA.2016.7562754.

[46] E. F. Inc, „The Platform for Open Innovation and Collaboration | The Eclipse Foundation". https://www.eclipse.org/ (viděno bře. 03, 2020).

[47] R. Krátký, *manas08/Chess-graph*. 2020.

[48] Y. Takefuji a K. C. Lee, „Neural network computing for knight's tour problems", *Neurocomputing*, roč. 4, č. 5, s. 249–254, srp. 1992, doi: 10.1016/0925-2312(92)90030-S.

Seznam obrázků

Obrázek 1 - Příklad grafu v praxi ... 8
Obrázek 2 - Využití grafů v městské infrastruktuře (zdroj: [2]) ... 9
Obrázek 3 – Historická mapa Královce (zdroj: [6]) ... 10
Obrázek 4 – The Icosian Calculus s vyznačenou hamiltonovskou kružnicí ... 15
Obrázek 5 – Ukázky her The Icosian Game a Cestovatel dvanáctistěnem (zdroj: [17]) ... 15
Obrázek 6 – Petersenův graf ... 17
Obrázek 7 – Příklad řešení hamiltonovské kružnice hrubou silou (zdroj: [24]) ... 20
Obrázek 8 – Rotační transformace (zdroj: [29]) ... 22
Obrázek 9 – Heuristika nedosažitelného vrcholu (zdroj: [29]) ... 23
Obrázek 10 - Pohyb šachového jezdce ... 24
Obrázek 11 – Všechny možné tahy jezdce na šachovnici $8x8$ ... 26
Obrázek 12 – Příklad otevřeného řešení jezdcovy procházky na šachovnici $6x6$ ... 27
Obrázek 13 – Číselná reprezentace polí šachovnice ... 29
Obrázek 14 – Warnsdorffové pravidlo ... 31
Obrázek 15 – Conradův algoritmus na šachovnici $16x16$ (zdroj: [39]) ... 31
Obrázek 16 – Grafické znázornění neuronu (zdroj: [42]) ... 32
Obrázek 18 – Formální neuron (1943) (zdroj: [43]) ... 33
Obrázek 19 – Aplikace CheGra ... 35
Obrázek 20 – Nabídka hlavního menu odlehčené verze pro studenty (vlevo) a plné verze (vpravo) ... 36
Obrázek 21 – Nabídka variant hry „Najdi procházku“ ... 37
Obrázek 22 – Horní lišta aplikace ... 37
Obrázek 23 – Dolní lišta Krycí hry ... 38
Obrázek 24 – Popis problému ... 39
Obrázek 25 – Hra „Najdi procházku“ ... 40
Obrázek 26 – Simulace řešení šachovnice 5x5 ... 41
Obrázek 27 – Hra „Krycí hra“ ... 42
Obrázek 28 – Barevné varianty vrcholů grafu ... 43

Obrázek 29 – Hra Hamilton – příklad 1. úrovně 43
Obrázek 30 - Hra Hamilton – příklad 2. úrovně 44
Obrázek 31 – Volba velikosti šachovnice 44
Obrázek 32 – Volba typu cest pro generování 45
Obrázek 33 – Výsledek generování cest na šachovnici $10x3$ 45
Obrázek 34 – Soubor obsahující vygenerovaná řešení 46
Obrázek 35 – Styl zadávání velikosti šachovnice 47
Obrázek 36 – Grafický výstup generování pomocí neuronové sítě 47
Obrázek 37 – Příklad Vertexů a Edgů 49
Obrázek 38 – Informační cedulka ve hře Hamilton (2. úroveň) 51
Obrázek 39 - Stabilní síť tvořená několika komponentami (zdroj: [39]) 57
Obrázek 41 - Flowchart neuronové sítě 62
Obrázek 42 - Jezdcova procházka (8x8) vyřešená Hopfieldovou neuronovou sítí 66
Obrázek 43 - Jezdcova procházka (28x28) vyřešená Hopfieldovou neuronovou sítí 66

Seznam algoritmů

Kód 1 – Nastavení systémového výstupu na soubor backtracking.txt......55
Kód 2 – Inicializace pole......58
Kód 3 – Spuštění algoritmu......58
Kód 4 – Rekurzivní provádění tahu jezdce......59
Kód 5 – Reprezentace pohybů jezdce......60
Kód 6 – Ohodnocení polí (Warnsdorffovo pravidlo)......61

Backtracking

```
package cz.uhk.diplom.prochazka;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.List;

import javax.swing.JOptionPane;

public class KnightsTour {

	/**
	 *
	 * @author manas08
	 *
	 * Backtracking algoritm
	 * for searching Knight's tours in chessboard
	 *
	 */

	/**
	 * Priznak nenavstivenosti policka
	 */
	private static int NOT_VISITED = -1;
	/**
	 * Velikost sachovnice na ose x
	 */
	private int xSize;
	/**
	 * Velikost sachovnice na ose y
	 */
	private int ySize;
	/**
	 * Pocet reseni
	 */
	private int solutionsCount, pocetReseni;
	/**
	 * Pole pro reseni 0 -> pocatecni pozice kone 1 -> prvni tah 2 -> druhy tah
	 * . . . n -> n-ty tah
	 */
	private int[][] solutionBoard;

	private boolean podm = true, uzav = false;

	/**
	 * Konstruktor resitele jezdcovy prochazky
	 *
	 * @param xSize
	 *            velikost sachovnice na ose x
	 * @param ySize
	 *            velikost sachovnice na ose y
	 * @param n2
	 *            typ hledane cesty
	 */
```

```
public KnightsTour(int xSize, int ySize, String n2, int pocetReseni) {
	this.xSize = xSize;
	this.ySize = ySize;
	this.pocetReseni = pocetReseni;
	if (n2 == "uzavřené cesty") {
		uzav = true;
		podm = false;
		n2 = "uzavřené";
	} else {
		n2 = "otevřené";
	}
	solutionsCount = 0;

	solutionBoard = new int[ySize][xSize];

	PrintStream originalStdout = System.out;
	PrintStream out = null;
	try {
		out = new PrintStream(new FileOutputStream("backtracking.txt"));
	} catch (FileNotFoundException e) {
		e.printStackTrace();
	}
	System.setOut(out);

	System.out.println("Tento soubor obsahuje " + pocetReseni + " řešení vygenerovaná
	algoritmem Backtracking.");
	System.out.println("Řešení jsou vypsána v podobě čísel ve tvaru velikosti
	šachovnice " + xSize + "x" + ySize + ".");
	System.out.println("Každé číslo reprezentuje krok jezdcovy procházky.");
	System.out.println("Číslo 0 je políčko, kde jezdec začíná");
	System.out.println("Číslo 1 je políčko, kam se jezdec přesunul z kroku 0 atd...");
	System.out.println();

	for (int i = 0; i < ySize; i++) {
		for (int j = 0; j < xSize; j++) {
			solutionBoard[i][j] = NOT_VISITED;
		}
	}
	solve();

	System.out.println("<td>" + solutionsCount + "</td>");

	if (uzav) {
		solutionsCount = solutionsCount / 2;
	}

	System.setOut(originalStdout);
	JOptionPane.showMessageDialog(null, "Právě bylo vygenerováno " + pocetReseni + "
	cest jezdcovy procházky " + "\n na šachovnici " + xSize + "x" + ySize + ", jedná se
	o " + n2 + ".\nLze je nalézt v souboru backtracking.txt ve složce s aplikací.",
	"Hotovo", JOptionPane.INFORMATION_MESSAGE, null);
}

public void solve() {
	// hledání řešení od všech možných výchozích pozic jezdce
```

```
        for (int i = 0; i < ySize; i++) {
            for (int j = 0; j < xSize; j++) {
                takeTurn(j, i, 0);
                solutionBoard[i][j] = NOT_VISITED; // rekurzivní reset pole
            }
        }
}

/**
 * Vrati policka, na ktera muze kun skocit
 *
 * @param x
 *            souradnice kone x
 * @param y
 *            souradnice kone y
 * @return souradnice, na ktere muze kun skocit
 */

private List<Coords> getFields(int x, int y) {
        List<Coords> l = new ArrayList<Coords>();
        if (x + 2 < xSize && y - 1 >= 0)
            l.add(new Coords(x + 2, y - 1)); // doprava nahoru
        if (x + 1 < xSize && y - 2 >= 0)
            l.add(new Coords(x + 1, y - 2)); // nahoru doprava
        if (x - 1 >= 0 && y - 2 >= 0)
            l.add(new Coords(x - 1, y - 2)); // nahoru doleva
        if (x - 2 >= 0 && y - 1 >= 0)
            l.add(new Coords(x - 2, y - 1)); // doleva nahoru
        if (x - 2 >= 0 && y + 1 < ySize)
            l.add(new Coords(x - 2, y + 1)); // doleva dolu
        if (x - 1 >= 0 && y + 2 < ySize)
            l.add(new Coords(x - 1, y + 2)); // dolu doleva
        if (x + 1 < xSize && y + 2 < ySize)
            l.add(new Coords(x + 1, y + 2)); // dolu doprava
        if (x + 2 < xSize && y + 1 < ySize)
            l.add(new Coords(x + 2, y + 1)); // doprava dolu
        return l;
}

/**
 * Provede tah konem
 *
 * @param x
 *            cilova souradnice x
 * @param y
 *            cilova souradnice y
 * @param turnNr
 *            cislo tahu
 */

private void takeTurn(int x, int y, int turnNr) {
        // pokud máme všechna řešení co jsme chtěli -> konec
        if (pocetReseni == solutionsCount) {
            return;
        }
        solutionBoard[y][x] = turnNr;
```

```
        // pokud bylo nalezno řešení
        if (turnNr == (xSize * ySize) - 1) {
            // podmínka pro výstup uzavřených řešení
            if (uzav) {
                for (Coords c : getFields(x, y)) {
                    if (solutionBoard[c.getY()][c.getX()] == 0) {
                        podm = true;
                    }
                }
            }

            printSolution(); // tisk řešení
            return;

        // pokud ještě nebylo řešení nalezeno
        } else {
            // provedení tahu na ještě nenavštívené pole dle pravidel šachu
            for (Coords c : getFields(x, y)) {
                if (solutionBoard[c.getY()][c.getX()] == NOT_VISITED) {
                    takeTurn(c.getX(), c.getY(), turnNr + 1);
                    // rekurzivní reset pole
                    solutionBoard[c.getY()][c.getX()] = NOT_VISITED;
                }
            }
        }
    }

    /**
     * Vypise reseni
     */
    private void printSolution() {
        if (podm) {

            solutionsCount++;
            podm = false;

            System.out.println(solutionsCount + ". řešení (Backtracking)");
            for (int i = 0; i < solutionBoard.length; i++) {
                for (int j = 0; j < solutionBoard[i].length; j++) {
                    System.out.print(solutionBoard[i][j] + " ");
                }
                System.out.println("");
            }
            System.out.println("");

            if (!uzav) {
                podm = true;
            }

        }
    }

    /**
     * @return the solutionsCount
     */
    public int getSolutionsCount() {
        return solutionsCount;
```

```
	}

	/**
	 * Reprezentuje souradnici
	 */
	private class Coords {
		private int x;
		private int y;

		public Coords(int x, int y) {
			this.x = x;
			this.y = y;
		}

		/**
		 * @return the x
		 */
		public int getX() {
			return x;
		}

		/**
		 * @return the y
		 */
		public int getY() {
			return y;
		}
	}
}
```

Warnsdorffův algoritmus

```
package cz.uhk.diplom.prochazka;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;

import javax.swing.JOptionPane;

public class KnightTest2 {


	/**
	 *
	 * @author manas08
	 *
	 * Warnsdorff algoritm
	 * for searching Knight's tours in chessboard
	 *
	 */

	public static PrintStream originalStdout;
	public static int n;
	public static int numberOfBoards;
	public static String n2;
	public static String response;
	public static int[][] label = new int[1000][1000];
	public static int[][] deg = new int[1000][1000];
	public static int[] DX = { 1, 2, 2, 1, -1, -2, -2, -1 };
	public static int[] DY = { -2, -1, 1, 2, 2, 1, -1, -2 };


	private static int numberofsolution = 0;

	public static void output_label() {
		for (int j = 0; j < n; j++) {
			System.out.print(label[0][j]-1);
			for (int i = 1; i < n; i++) {
				System.out.print(" ");
				System.out.print(label[i][j]-1);
			}
			System.out.print("\n");
		}
	}

	public static void output() {
		if (numberofsolution == numberOfBoards) {
			return;
		}
		numberofsolution++;
		System.out.println();
		System.out.println(numberofsolution + ". řešení (Warnsdorff)");
		output_label();

		if (numberofsolution == numberOfBoards) {
```

```
            System.setOut(originalStdout);
            JOptionPane.showMessageDialog(null, "Právě bylo vygenerováno " +
            numberofsolution + " cest jezdcovy procházky " + "\n na šachovnici " + n +
            "x" + n + ", jedná se o " + n2 + ".\nLze je nalézt v souboru warnsdorff.txt
            ve složce s aplikací.", "Hotovo", JOptionPane.INFORMATION_MESSAGE, null);
        }
    }

    public static void init() {
        int i;
        int j;
        int u;
        int v;
        int t;

        // pro každé pole
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                // pro každý směr
                for (t = 0; t < 8; t++) {
                    // DX a DY obsahují pohyby jezdce po ose X a Y
                    u = i + DX[t];
                    v = j + DY[t];
                    // když se při pohybu nedostanou mimo šachovnici
                    // -> zvýší se ohodnocení
                    if ((u >= 0) && (u < n) && (v >= 0) && (v < n)) {
                        deg[i][j]++;
                    }
                }
            }
        }
    }

    public static void BackTracking(int number, int i, int j) {
        if (numberofsolution == numberOfBoards) {
            return;
        }

        int u;
        int v;
        int t;
        int nChoices;
        int[] d = new int[8];
        int[] next = new int[8];

        label[i][j] = number;

        if (number < n * n) {
            nChoices = 0;
            for (t = 0; t < 8; t++) {
                u = i + DX[t];
                v = j + DY[t];
                if ((u >= 0) && (u < n) && (v >= 0) && (v < n)) {
                    if (label[u][v] == 0) {
                        d[nChoices] = deg[u][v];
                        next[nChoices] = t;
                        nChoices++;
```

```
                        deg[u][v]--;
                    }
                }
            }

            for (u = 0; u < nChoices; u++) {
                for (v = u + 1; v < nChoices; v++) {
                    if (d[u] > d[v]) {
                        int pom1 = d[u];
                        d[u] = d[v];
                        d[v] = pom1;

                        pom1 = next[v];
                        next[v] = next[u];
                        next[u] = pom1;
                    }
                }
            }

            for (t = 0; t < nChoices; t++) {
                BackTracking(number + 1, i + DX[next[t]], j + DY[next[t]]);
            }

            for (t = 0; t < 8; t++) {
                u = i + DX[t];
                v = j + DY[t];
                if ((u >= 0) && (u < n) && (v >= 0) && (v < n)) {
                    if (Label[u][v] == 0) {
                        deg[u][v]++;
                    }
                }
            }
        } else {
            // Case 1: Need to find a route
            if ((response.charAt(0) == 'r') || (response.charAt(0) == 'R')) {
                output();
            }

            // Case 2: Need to find a Hamilton circle
            else if ((i != 0) && (j != 0)) {

                for (t = 0; t < 8; t++) {
                    u = i + DX[t];
                    v = j + DY[t];
                    if ((u == 2) && (v == 2)) {
                        output();
                    }
                }
                if (i + j == 3) {
                    output();
                }
            }
        }

        Label[i][j] = 0;
    }
```

```
	public void Main(int n, String n2, int numberOfBoards) {
		this.n = n;
		this.n2 = n2;
		this.numberOfBoards = numberOfBoards;

		originalStdout = System.out;
		PrintStream out = null;
		try {
			out = new PrintStream(new FileOutputStream("warnsdorff.txt"));
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		}
		System.setOut(out);
		System.out.println("Tento soubor obsahuje " + numberOfBoards + " vygenerovaných
		řešení Warnsdorffovým algoritmem.");
		System.out.println("Řešení jsou vypsána v podobě čísel ve tvaru velikosti
		šachovnice " + n + "x" + n + ".");
		System.out.println("Každé číslo reprezentuje krok jezdcovy procházky.");
		System.out.println("Číslo 0 je políčko, kde jezdec začíná");
		System.out.println("Číslo 1 je políčko, kam se jezdec přesunul z kroku 0 atd...");

		if (n2 == "uzavřené cesty") {
			response = "c";
		}else {
			response = "r";
		}
		init();

		if ((response.charAt(0) == 'r') || (response.charAt(0) == 'R')) {
			for (int i = 0; i < n; i++) {
				for (int j = 0; j < n; j++) {
					BackTracking(1, i, j);
				}
			}
		} else {
			BackTracking(1, n - 3, n - 3); // 2,2
		}

		System.out.print("There is no solution.");
		System.out.print("\n");
	}
}
```

Hopfieldova neuronová síť

```
package cz.uhk.diplom.prochazka;

import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.image.BufferedImage;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.List;

import javax.imageio.ImageIO;

import cz.uhk.diplom.MainWindow;
import cz.uhk.diplom.model.Image;
import cz.uhk.diplom.model.Vertex;
import cz.uhk.diplom.tangible.RandomNumbers;
import cz.uhk.diplom.utils.Link;

/**
 *
 * @author manas08
 *
 * Algoritm based on Hopfield's neural network
 * for searching Knight's tours in chessboard
 *
 */

public class NeuralNetworkTour {

	int CSIZE;
	int DSIZE;
	int NSIZE;
	boolean hamiltonian;
	int number;
	int totalTrials = 0;
	int XSIZE = 80;
	int YSIZE = 80;
	int numberOfBoards;

	int[][] U;
	int[][] V;
	int[][] D;
	int[][] A;

	List<Link> links;
	List<Integer[]> path;

	BufferedImage img3 = null;
	BufferedImage img4 = null;
	BufferedImage img5 = null;
	BufferedImage img6 = null;
	BufferedImage img7 = null;
	BufferedImage img8 = null;
```

```
MainWindow main;

public NeuralNetworkTour() {
        try {
                img3 = ImageIO.read(getClass().getResourceAsStream("/textures/brick.jpg"));
                img4 = ImageIO.read(getClass().getResourceAsStream("/textures/brick2.jpg"));
                img5 = 
                ImageIO.read(getClass().getResourceAsStream("/textures/smallbricks.jpg"));
                img6 = 
                ImageIO.read(getClass().getResourceAsStream("/textures/smallbricks2.jpg"));
                img7 = 
                ImageIO.read(getClass().getResourceAsStream("/textures/verysmallbricks.jpg"))
                ;
                img8 = 
                ImageIO.read(getClass().getResourceAsStream("/textures/verysmallbricks2.jpg")
                );

        } catch (IOException e) {
                e.printStackTrace();
        }
}

public void cmdGoClick(MainWindow main, int x, int y) {
        this.CSIZE = x;
        this.DSIZE = y;
        this.NSIZE = CSIZE * DSIZE;
        this.main = main;
        int numHamiltonian = 0;
        boolean stopped = false;

        U = new int[NSIZE][NSIZE];
        V = new int[NSIZE][NSIZE];
        D = new int[NSIZE][NSIZE];
        A = new int[NSIZE][NSIZE];
        links = new ArrayList<>();

        PrintStream originalStdout = System.out;
        PrintStream out = null;
        try {
                out = new PrintStream(new FileOutputStream("solutions.txt"));
        } catch (FileNotFoundException e) {
                e.printStackTrace();
        }
        System.setOut(out);

        System.out.println("Tento soubor obsahuje " + numberOfBoards + " vygenerovaných
        řešení Hopfieldovou neuronovou sítí.");
        System.out.println("Řešení jsou vypsána v podobě čísel ve tvaru velikosti
        šachovnice " + x + "x" + y + ".");
        System.out.println("Každé číslo reprezentuje krok jezdcovy procházky.");
        System.out.println("Číslo 0 je políčko, kde jezdec začíná");
        System.out.println("Číslo 1 je políčko, kam se jezdec přesunul z kroku 0 atd...");
        System.out.println();

        do {
```

```
Initialize();

int n;
int t = 0;
int diag = 1;
int dU;
int k;
int sum_row;
int sum_col;
int[] U_;
int[] V_;
int[] D_;

while (diag > 0) {
        diag = 0;
        n = 1;

        for (int i = 0; i < NSIZE; i++) {
                U_ = U[i];
                V_ = V[i];
                D_ = D[i];
                for (int j = n; j < NSIZE; j++) {

                        if (D_[j] == 1) {
                                sum_row = 0;
                                sum_col = 0;
                                for (k = 0; k < NSIZE; k++) {
                                        if (D_[k] != 0) {
                                                sum_row += V_[k];
                                        }
                                        if (D[k][j] == 1) {
                                                sum_col += V[k][j];
                                        }
                                }
                                dU = -(sum_row - 2) - (sum_col - 2);
                        } else {
                                dU = 0;
                        }

                        U_[j] += dU;

                        if (U_[j] > 10) {
                                U_[j] = 10;
                        }
                        if (U_[j] < -10) {
                                U_[j] = -10;
                        }
                        if (U_[j] > 3) {
                                V_[j] = 1;
                        }
                        if (U_[j] < 0) {
                                V_[j] = 0;
                        }

                        if (V_[j] == 1) {
                                V[j][i] = 1;
                        }
```

```
                                if (V_[j] == 0) {
                                    V[j][i] = 0;
                                }

                                diag += (dU == 0 ? 0 : 1);
                            }
                            n++;
                        }

                        if (t > 1000) {
                            break;
                        }
                        t++;

                    }

                    if (stopped) {
                        break;
                    }

                    if (t > 1000) {
                        hamiltonian = false;
                    } else if (CheckHamiltonian()) {
                        numHamiltonian++;
                        hamiltonian = true;
                        DrawNeurons();
                    } else {
                        hamiltonian = false;
                    }

                    // totalTrials < targetTrials
                } while (numHamiltonian < numberOfBoards);

                System.setOut(originalStdout);
                stopped = true;
            }

            public boolean CheckHamiltonian() {
                int p = 1;
                int linkNum = 0;
                int linkCount;
                for (int m = 0; m < CSIZE; m++) {
                    for (int n = p; n < DSIZE; n++) {
                        if (V[m][n] == 1) {
                            Link l = links.get(linkNum++);
                            l.i = m;
                            l.j = n;
                            l.visited = false;
                        }
                    }
                    p++;
                }

                linkCount = linkNum;
                linkNum = 0;
                int numTraversed = 0;
                Link l = links.get(linkNum);
```

```
        int startPt = l.i;
        int nextPt = l.j;
        boolean found;
        while (true) {
            l.visited = true;
            found = false;
            for (int i = 0; i < linkCount; i++) {
                Link l2 = links.get(i);
                if (l2.visited) {
                    continue;
                }
                if (l2.i == nextPt) {
                    l = l2;
                    nextPt = l2.j;
                    found = true;
                    numTraversed++;
                    break;
                }
                if (l2.j == nextPt) {
                    l = l2;
                    nextPt = l2.i;
                    found = true;
                    numTraversed++;
                    break;
                }
            }
            if (!found) {
                break;
            }
            if ((nextPt == startPt) || (numTraversed >= (linkCount - 1))) {
                break;
            }
        }
        if ((nextPt == startPt) && (numTraversed >= (linkCount - 1))) {
            if (oneComponent()) {
                return true;
            } else {
                return false;
            }
        }
        return false;
    }

    public void Initialize() {
        int n;
        int m;
        int count;

        // clear old arrays
        U = new int[NSIZE][NSIZE];
        V = new int[NSIZE][NSIZE];
        D = new int[NSIZE][NSIZE];
        A = new int[NSIZE][NSIZE];
        links = new ArrayList<>();

        int numLinks;
        if (CSIZE > DSIZE) {
```

```
        numLinks = 4 * (CSIZE - 2) * (CSIZE - 1) + 1;
} else {
        numLinks = 4 * (DSIZE - 2) * (DSIZE - 1) + 1;
}
for (int i = 0; i < numLinks; i++) {
        Link l = new Link();
        links.add(l);
}

for (m = 0; m < NSIZE; m++) {
        for (n = 0; n < NSIZE; n++) {
                U[m][n] = -(int) (RandomNumbers.nextNumber() % CSIZE);
                V[m][n] = 0;
        }
}
for (m = 0; m < NSIZE; m++) {
        for (n = 0; n < NSIZE; n++) {
                D[m][n] = 0;
        }
}

// directions
for (int i = 0; i < NSIZE; i++) {
        if (i + 1 < CSIZE * (i / CSIZE + 1) && i + 1 - (2 * CSIZE) >= 0) {
                D[i][i + 1 - (2 * CSIZE)] = 1;
        }
        if (i + 2 < CSIZE * (i / CSIZE + 1) && i + 2 - CSIZE >= 0) {
                D[i][i + 2 - CSIZE] = 1;
        }
        if (i + 2 < CSIZE * (i / CSIZE + 1) && i + 2 + CSIZE < NSIZE) {
                D[i][i + 2 + CSIZE] = 1;
        }
        if (i + 1 < CSIZE * (i / CSIZE + 1) && i + 1 + (2 * CSIZE) < NSIZE) {
                D[i][i + 1 + (2 * CSIZE)] = 1;
        }
        if (i - 1 >= CSIZE * (i / CSIZE) && i - 1 + (2 * CSIZE) < NSIZE) {
                D[i][i - 1 + (2 * CSIZE)] = 1;
        }
        if (i - 2 >= CSIZE * (i / CSIZE) && i - 2 + CSIZE < NSIZE) {
                D[i][i - 2 + CSIZE] = 1;
        }
        if (i - 2 >= CSIZE * (i / CSIZE) && i - 2 - CSIZE >= 0) {
                D[i][i - 2 - CSIZE] = 1;
        }
        if (i - 1 >= CSIZE * (i / CSIZE) && i - 1 - (2 * CSIZE) >= 0) {
                D[i][i - 1 - (2 * CSIZE)] = 1;
        }
}

count = -1;
for (int i = 0; i < NSIZE; i++) {
        if (i % CSIZE == 0) {
                count++;
                A[i][0] = XSIZE + XSIZE / 2;
                A[i][1] = YSIZE + YSIZE / 2 + YSIZE * count;
        } else {
                A[i][0] = XSIZE + XSIZE / 2 + XSIZE * (i % CSIZE);
```

```
                A[i][1] = YSIZE + YSIZE / 2 + YSIZE * count;
            }
        }
    }

    public void DrawNeurons() {

        int p = 1;

        List<Integer> points = new ArrayList<>();
        drawBoard();

        for (int m = 0; m < NSIZE; m++) {
            for (int n = p; n < NSIZE; n++) {
                if (V[m][n] == 1) {

                    int x1 = A[m][0] - 120;
                    int y1 = A[m][1] - 120;
                    int x2 = A[n][0] - 120;
                    int y2 = A[n][1] - 120;

                    points.add(x1 / 80);
                    points.add(y1 / 80);
                    points.add(x2 / 80);
                    points.add(y2 / 80);

                    main.drawTest(x1 / 80, y1 / 80, x2 / 80, y2 / 80, CSIZE);
                }
            }
            p++;
        }

        Integer[] vertex = new Integer[3];
        vertex[0] = 0;
        vertex[1] = 0;
        vertex[2] = 0;
        path = new ArrayList<Integer[]>();
        path.add(vertex);
        int w = 0;
        p = 0;
        boolean b = false;
        totalTrials++;
        while (path.size() < NSIZE) {

            for (int i = 0; i < points.size(); i = i + 4) {

                if (path.size() == w) {
                    main.clear();
                    cmdGoClick(main, CSIZE, DSIZE);
                    return;
                }

                if ((path.get(w)[0] == points.get(i)) && (path.get(w)[1] ==
                points.get(i + 1))) {
                    vertex = new Integer[3];
                    vertex[0] = points.get(i + 2);
                    vertex[1] = points.get(i + 3);
```

```
                    for (Integer[] integer : path) {
                        if (integer[0] == vertex[0] && integer[1] == vertex[1]) {
                            b = true;
                        }
                    }
                    if (!b) {
                        vertex[2] = path.size();
                        path.add(vertex);
                        break;
                    }
                } else if ((path.get(w)[0] == points.get(i + 2)) && (path.get(w)[1] ==
                points.get(i + 3))) {
                    vertex = new Integer[3];
                    vertex[0] = points.get(i);
                    vertex[1] = points.get(i + 1);
                    for (Integer[] integer : path) {
                        if (integer[0] == vertex[0] && integer[1] == vertex[1]) {
                            b = true;
                        }
                    }
                    if (!b) {
                        vertex[2] = path.size();
                        path.add(vertex);
                        break;
                    }
                }
                b = false;
            }
            w++;
        }

        number++;


        System.out.println(number + ". řešení (Neural network)");
        if (CSIZE == DSIZE) {
            for (int i = 0; i < CSIZE; i++) {
                for (int j = 0; j < DSIZE; j++) {
                    for (Integer[] integer : path) {
                        if (integer[0] == i && integer[1] == j) {
                            System.out.print(integer[2] + " ");
                        }
                    }
                }
                System.out.println();
            }
            System.out.println();
            System.out.println();
        } else {
            for (int i = 0; i < DSIZE; i++) {
                for (int j = 0; j < CSIZE; j++) {
                    for (Integer[] integer : path) {
                        if (integer[1] == i && integer[0] == j) {
                            System.out.print(integer[2] + " ");
                        }
                    }
                }
```

```
                System.out.println();
            }
            System.out.println();
            System.out.println();
        }


    }

    private void drawBoard() {
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        double width = screenSize.getWidth();
        double height = screenSize.getHeight();
        double pom1 = 0, pom2 = 0;
        if (CSIZE >= 10) {
            pom1 = ((width / 2) - CSIZE * 16.5) / 33.0;
            pom2 = ((height / 2) - ((DSIZE * 16.5) + 50.0)) / 33.0;
        } else {
            pom1 = ((width / 2) - CSIZE * 50) / 100.0;
            pom2 = ((height / 2) - ((DSIZE * 50) + 50)) / 100.0;
        }

        Image obrazek = this.main.getObrazek();
        List<Vertex> vertices = this.main.getVertices();
        Vertex vertex;

        boolean obr = false;
        int row = 1;
        for (double i = pom2; i < pom2 + DSIZE; i++) {
            int collumn = 1;
            if (obr) {
                obr = false;
            } else {
                obr = true;
            }

            for (double j = pom1; j < pom1 + CSIZE; j++) {
                if (obr) {
                    if (CSIZE >= 10) {
                        vertex = new Vertex((int) (j * 33), (int) (i * 33), img5,
                        2);
                    } else if (CSIZE > 28) {
                        vertex = new Vertex((int) (j * 15), (int) (i * 15), img7,
                        2);
                    } else {
                        vertex = new Vertex((int) (j * 100), (int) (i * 100),
                        img3, 2);
                    }
                    if (CSIZE == DSIZE) {
                        vertex.setCollumn(collumn);
                        vertex.setRow(row);
                    } else {
                        vertex.setCollumn(row);
                        vertex.setRow(collumn);
                    }
                    vertex.setWhite(false);
                    collumn++;
```

```
                        vertices.add(vertex);
                        obrazek.pridej(vertex);
                        obr = false;
                    } else {
                        if (CSIZE >= 10) {
                            vertex = new Vertex((int) (j * 33), (int) (i * 33), img6,
                            2);
                        } else if (CSIZE > 28) {
                            vertex = new Vertex((int) (j * 15), (int) (i * 15), img8,
                            2);
                        } else {
                            vertex = new Vertex((int) (j * 100), (int) (i * 100),
                            img4, 2);
                        }
                        if (CSIZE == DSIZE) {
                            vertex.setCollumn(collumn);
                            vertex.setRow(row);
                        } else {
                            vertex.setCollumn(row);
                            vertex.setRow(collumn);
                        }
                        vertex.setWhite(true);
                        collumn++;
                        vertices.add(vertex);
                        obrazek.pridej(vertex);
                        obr = true;
                    }
                }
                row++;
            }
            main.setObrazek(obrazek);
            main.setVertices(vertices);
    }

    public boolean oneComponent() {

            List<Integer> points = new ArrayList<>();
            int p = 1;
            for (int m = 0; m < NSIZE; m++) {
                for (int n = p; n < NSIZE; n++) {
                    if (V[m][n] == 1) {
                        int x1 = A[m][0] - 120;
                        int y1 = A[m][1] - 120;
                        int x2 = A[n][0] - 120;
                        int y2 = A[n][1] - 120;

                        points.add(x1 / 80);
                        points.add(y1 / 80);
                        points.add(x2 / 80);
                        points.add(y2 / 80);
                    }
                }
                p++;
            }

            Integer[] vertex = new Integer[3];
            vertex[0] = 0;
```

```
        vertex[1] = 0;
        vertex[2] = 0;
        path = new ArrayList<Integer[]>();
        path.add(vertex);
        int w = 0;
        p = 0;
        boolean b = false;
        totalTrials++;
        while (path.size() < NSIZE) {

            for (int i = 0; i < points.size(); i = i + 4) {

                if (path.size() == w) {
                    return false;
                }
                if ((path.get(w)[0] == points.get(i)) && (path.get(w)[1] ==
                points.get(i + 1))) {
                    vertex = new Integer[3];
                    vertex[0] = points.get(i + 2);
                    vertex[1] = points.get(i + 3);
                    for (Integer[] integer : path) {
                        if (integer[0] == vertex[0] && integer[1] == vertex[1]) {
                            b = true;
                        }
                    }
                    if (!b) {
                        vertex[2] = path.size();
                        path.add(vertex);
                        break;
                    }
                } else if ((path.get(w)[0] == points.get(i + 2)) && (path.get(w)[1] ==
                points.get(i + 3))) {
                    vertex = new Integer[3];
                    vertex[0] = points.get(i);
                    vertex[1] = points.get(i + 1);
                    for (Integer[] integer : path) {
                        if (integer[0] == vertex[0] && integer[1] == vertex[1]) {
                            b = true;
                        }
                    }
                    if (!b) {
                        vertex[2] = path.size();
                        path.add(vertex);
                        break;
                    }
                }
                b = false;
            }
            w++;
        }
        return true;
    }

    public void setNumberOfBoards(int jml1) {
        this.numberOfBoards = jml1;
    }
```

```
	public String getNumberOfBoards() {
		return String.valueOf(numberOfBoards);
	}
}
```

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor: **Bc. Radim Krátký**

Studium: I1800113

Studijní program: N1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název diplomové práce: **Hamiltonovské grafy**
Název diplomové práce AJ: Hamiltonian Graphs

Cíl, metody, literatura, předpoklady:

ZÁSADY PRO VYPRACOVÁNÍ:

- Rešerše dané problematiky

- Nastudování potřebné oblasti z teorie grafů a grafových algoritmů

- Teoretické zpracování vybraného problému z teorie grafů ? Hamiltonovské grafy a využití/význam v praxi

- Návrh aplikace pro hledání Hamiltonovské kružnice a její využití

- Výběr technologie, ve které bude technologie vyvíjena

- Vytvoření aplikace

- Popis aplikace a její ovládání

- Testování aplikace

- Navržení rozvoje aplikace do budoucna

SEZNAM DOPORUČENÉ LITERATURY:

DEMEL, Jiří: Grafy a jejich aplikace, Academia, 2002.

KUČERA, Luděk: Kombinatorické algoritmy, SNTL, 1983.

MATOUŠEK, J.: NEŠETŘIL, J.: Kapitoly z diskrétní matematiky, Karolinum, Praha 2000.

MILKOVÁ, Eva: Teorie grafů a grafové algoritmy, Gaudeamus, Hradec Králové, 2013.

Garantující pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Datum zadání závěrečné práce: 14.1.2018