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

Hamiltonovské grafy

Bakalářská práce

Autor: Ondřej Brekl
Studijní obor: Aplikovaná informatika

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

Hradec Králové listopad 2021

Prohlášení:

Prohlašuji, že jsem bakalářskou/diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 9.11.2021 Ondřej Brekl

Poděkování:

Děkuji vedoucímu bakalářské práce prof. RNDr. PhDr. Antonínovi Slabému, CSc. za metodické vedení práce a konzultaci odborných otázek.

Anotace

Práce má za cíl vytvořit program pro řešení hamiltonovských grafů. To zahrnuje následující. Shromáždění teoretických poznatků o hamiltonovských grafech. Definice pojmů z teorie grafů, nutné, postačující podmínky hamiltonovských grafů. Představení stručné historie hamiltonovských grafů. Rozebrání problematiky P, NP a NP-úplných tříd složitosti. Na těchto teoretických základech je vytvořen program pro edukační účely. Program má sloužit jako pomůcka při výkladu problematiky hamiltonovských grafů a k případnému samostatnému procvičení. V programu je možné grafy kreslit nebo si je nechat vygenerovat. Program také obsahuje předpřipravené ukázkové příklady. Nad všemi těmito částmi lze spustit prohledávací algoritmus, který určí, zda je graf hamiltonovský. Algoritmus je postaven na backtrackingu. Kód je napsán v jazyce Java za pomoci grafické knihovny Swing. Program je multiplatformní. Podporuje český a anglický jazyk.

Klíčová slova: hamiltonovský graf, prohledávání do hloubky, NP-úplný

Annotation

Title: Hamiltonian graphs

The aim of this work is to create a program for solving Hamiltonian graphs. That includes the following. Gathering theoretical knowledge about Hamiltonian graphs. Definition of graph theory concepts, necessary, sufficient conditions of Hamiltonian graphs. Brief introduction of Hamiltonian graphs' history. Discussion of P, NP and NP-complete complexity classes. Building on said theoretical foundations to create a program for educational purposes. The program is intended to serve as an aid in explaining the subject matter of Hamiltonian graphs and for students' independent exercise. Within the program, users can draw their own graphs or have them generated. It also contains pre-made examples. A search

algorithm can be run over all these parts to determine whether the graph is Hamiltonian. The algorithm is based on backtracking. Source code is written in Java using Swing graphics library. The program is cross-platform. It supports localization for Czech and English language.

Keywords: hamiltonian graph, backtracking, NP-complete

Obsah

1 Úvod

Práce se zabývá hamiltonovskými grafy. Nejdříve je popsána definice hamiltonovského grafu, na jakých grafech je možné hamiltonovské kružnice hledat a také představení dalších pojmů z teorie grafů, se kterými se v textu pracuje. Například úplný graf, bipartitní graf, most a artikulace. Dále jsou zmíněny nutné, postačující podmínky hamiltonovských grafů. Žádná nutná a zároveň postačující podmínka hamiltonovského grafu není známa. V další kapitole je popsána stručná historie grafů. Nejstarší zmínky o těchto grafech v podobě jezdcovy procházky. Přínos Thomase Kirkmana a jeho práce na průchodech mnohostěnů, kde jeho nejznámějším grafem je takzvaná „včelí plástev“. Na Kirkmanovu práci navázal William Hamilton, který problém zpopularizoval, když vymyslel společenskou hru Icosian game. Poté je zmíněno rozdělení algoritmů do tříd složitosti. Vysvětlení, co jsou P, NP problémy a NP-úplné problémy, do kterých hamiltonovské grafy spadají. Jsou uvedeni i jiní zástupci NP – úplných problémů. A prezentovány ukázky příkladů převoditelnosti těchto problémů.

Je představen algoritmus backtrackingu, který byl použit pro prohledávání hamiltonovských grafů. Na ukázkovém příkladu je znázorněno jeho fungování při průchodu grafem. Dále jsou popsány technologické nástroje použité při vytváření vlastního programu. Tento program slouží jako výuková pomůcka při výkladu hamiltonovských grafů. V kapitole o edukačním rámci programu je představeno, pro jaké konkrétní věci by bylo program vhodné použít. Program samotný je popsán z hlediska vnitřní funkcionality, rozložení uživatelského ovládání a hlavních funkcí programu (volné kreslení, generování, předpřipravené příklady). Nakonec je uvedeno, jak byl program testován.

2 Cíl práce

Cílem práce je vytvořit program, který by řešil hamiltonovské grafy. Ten má sloužit jako učební pomůcka, která pomůže vizualizovat problematiku hamiltonovských grafů. Bude možné kreslit vlastní grafy a také generovat náhodné grafy. Není nutné, aby prohledávací algoritmus fungoval pro velké množství uzlů, jde o to, aby díky němu bylo možné prezentovat vlastnosti grafů, které umožňují/zabraňují existenci hamiltonovského grafu (například spojitost grafu, závislost na stupních vrcholů, existence mostů). Je ale potřeba, aby program bez problémů zvládal pracovat s grafy o minimálně osmi uzlech. Program musí být spolehlivý a jeho rozhraní uživatelsky přístupné. Bude podporovat anglický a český jazyk.

3 Metodika zpracování

Nejdříve byly shromážděny teoretické poznatky o hamiltonovských grafech a jejich složitosti. Jako zdroje rešerše nejčastěji sloužily výzkumné práce a učební materiály univerzit. Dále bylo nutné prozkoumat, jaké technologické nástroje budou vhodné pro vytvoření programu na prohledávání grafů. Následně byl vytvořen prototyp, který implementoval prohledávací algoritmus, ale neměl grafické rozhraní. Toto řešení se ukázalo jako proveditelné a dostatečné funkční. Na základě prototypu byl napsán výsledný program. Ten byl průběžně testován a opravován. Program byl vytvořen ve vývojovém prostředí IntelliJ IDEA. Ke kreslení topologických obrázků byly použity Google Drawings a vlastní výsledný program odevzdávaný s bakalářskou prací. Pokud v práci není uveden zdroj obrázku nebo tabulky jedná se o vlastní zpracování.

4 Hamiltonovské grafy v Teorii grafů

4.1 Definice grafu

Hamiltonovské grafy spadají do oboru diskrétní matematiky, která se nazývá Teorie grafů. Ta se zabývá grafy, popisem jejich uzlů a relacemi mezi uzly vyjadřovanými pomocí hran. Uzly, mohou reprezentovat jakýkoli objekt (entitu) z reálného světa a hrany reprezentují vztahy, které mezi sebou jednotlivé objekty mají. Místo pojmu *uzel* se také používá označení *vrchol*. Matematická definice grafu: *„Graf G (také jednoduchý graf nebo obyčejný graf) je uspořádaná dvojice G = (V,E), kde V je neprázdná množina vrcholů a E je množina hran – množina (některých) dvouprvkových podmnožin množiny V.“* [22]

Na těchto grafech lze hledat hamiltonovské grafy. Graf je hamiltonovský, pokud obsahuje hamiltonovskou kružnici (též hamiltonovský cyklus). Pokud je možné projít graf přes všechny vrcholy, každý navštívit právě jednou a vrátit se do výchozího bodu, kde každá hrana byla použita maximálně jednou, jedná se o hamiltonovskou kružnici.

„Hamiltonovský cyklus v grafu je takový cyklus, který prochází všemi vrcholy daného grafu. Graf, ve kterém existuje hamiltonovský cyklus, se nazývá hamiltonovský graf.“ [22]

4.2 Další matematické pojmy používané v práci

komponenta grafu – Komponenta je maximální souvislý (propojený hranami) podgraf. Počet komponent je tedy počet samostatných „oblastí“ grafu, které nejsou navzájem propojeny. Pokud se graf skládá z více komponent, nemůže být hamiltonovský.

artikulace – Takový vrchol grafu G, že po jeho vyjmutí a hran s ním incidentních z G se zvětší počet komponent grafu. [8]

most - hrana, po jejímž odstranění se zvětší počet komponent grafu [8]

bipartitní graf – *„Bipartitní grafy jsou grafy, jejichž vrcholy lze rozdělit do dvou množin tak, že všechny hrany vedou jen mezi těmito množinami.“* [18] Tedy dva vrcholy ze stejné množiny, nikdy nemohou být propojeny hranou.

stupeň vrcholu - Udává počet hran, které jsou na uzel napojeny. Hrana, která začíná i končí v jednom uzlu, se počítá za dva.

orientovaný graf – Hrany grafu mají svůj směr. Hrana má svůj počáteční a koncový vrchol. Tyto vrcholy nelze zaměnit a lze jimi projít pouze v tomto pořadí. V opačném směru nelze hranou projít.

neorientovaný graf – Hrany grafu jsou nesměrové. Při průchodu grafem lze projít jedním nebo druhým směrem.

úplný graf – *„Graf na n vrcholech, kde $n \in N$, který obsahuje všech $\binom{n}{2}$ hran se nazývá úplný nebo také kompletní graf a značí se K_n.“* [22] Tedy graf obsahuje všechny hrany, které se v něm mohly vyskytnout.

ohodnocený graf – Hrany (případně vrcholy) ohodnoceného grafu mají svoji cenu, se kterou se pracuje při průchodu grafem. Typicky se takto reprezentuje vzdálenost nebo obtížnost trasy. Pokud jsou si všechny hrany a vrcholy rovny, jedná se o **neohodnocený graf.**

K určení, zda daný graf je hamiltonovský, do jisté míry mohou pomoci logické výroky nazývané postačující podmínky a nutné podmínky hamiltonovských grafů.

4.3 Nutné a postačující podmínky

Logické výroky mohou mít mezi sebou určité vztahy. S jejich pomocí lze určit, na základě daných vlastností, jestli se jedná o danou zkoumanou entitu či nikoliv. Tyto podmínky se dělí na nutné, postačující, nutné a zároveň postačující. Vysvětlení těchto podmínek je vztaženo (v této práci) na rozhodování, zda existuje hamiltonovský graf v obyčejném grafu.

Nutná podmínka musí být splněna, jakmile není splněna, graf neexistuje. Ale pokud je podmínka splněna, tak to neznamená, že graf existuje, jen pro svoji potenciální existenci splňuje nějakou nezbytnou podmínku.

Pokud je splněna *postačující podmínka*, graf existuje, ale neznamená to, že pokud podmínka není splněna, graf nemůže existovat.

Nutná a (zároveň) postačují podmínka je kombinací předchozích podmínek. Pokud je splněna, tak graf existuje, pokud není splněna, graf neexistuje.

Pro hamiltonovské grafy není známa žádná *nutná a zároveň postačující podmínka*.

Některé *postačující podmínky* jsou:

1. Diracova podmínka – *„Je-li G obyčejný graf s n uzly (n >= 3) a jestliže stupeň každého uzlu je nejméně n , pak G obsahuje hamiltonovskou kružnici."* [37]
2. Oreho podmínka – *„K tomu, aby graf G s n uzly (n >= 3) obsahoval hamiltonovskou kružnici, stačí, aby pro každé dva nesousední uzly u, v platilo $d_G(u) + d_G(v) >= n$."* [37] Symbol d_G značí stupeň uzlu, tedy kolik hran je na uzel napojeno. Součet stupňů dvou nesousedních uzlů musí být větší nebo roven celkovému počtu uzlů.
3. Pósova podmínka – *„Nechť G je graf s n uzly (n >= 3) takový, že pro každé celé číslo k splňující nerovnost*

   $$1 <= k < \frac{1}{2}(n - 1)$$

   je počet uzlů grafu G, jejichž stupeň není vyšší než k, menší než k a pro liché n počet uzlů stupně $\frac{1}{2}(n - 1)$ není vyšší než $\frac{1}{2}(n - 1)$. Potom G obsahuje hamiltonovskou kružnici." [37]

Některé *nutné podmínky* jsou:
Graf musí být obyčejný, musí být souvislý (všechny uzly musí být propojeny hranami), musí být konečný (graf má konečný počet uzlů a hran), musí mít minimálně 3 uzly, dále - *„Je-li G hamiltonovský graf, je jeho uzlový stupeň souvislosti u(G) ≥ 2."* [13] Tedy z každého uzlu musí vést minimálně 2 hrany. Pokud by byl jakýkoli uzel odebrán, graf zůstane souvislý.

Nutných nebo postačujících podmínek lze využít pro zefektivnění algoritmů při hledání hamiltonovských grafů. Například algoritmus před začátkem prohledávání může prověřit Diracovu podmínku. Pokud má graf v každém uzlu stejný počet hran jako je počet všech uzlů, je splněna postačující podmínka. Hamiltonovský graf určitě existuje a nemá smysl obyčejný graf prohledávat. Tato zefektivnění ovšem nestačí a není znám obecný algoritmus pro řešení hamiltonovských grafů v polynomiálním čase. Proč to není možné, je popsáno v kapitole *Třídy složitosti - P, N, NP-úplné*.

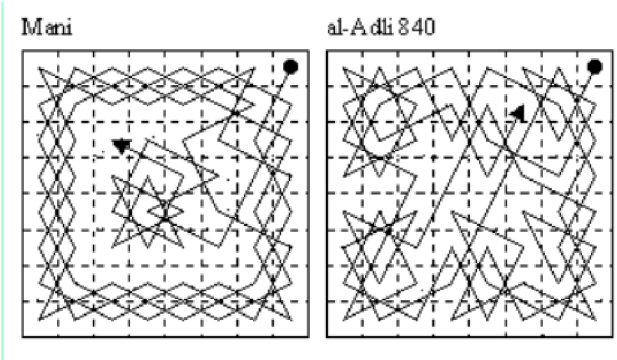
5 Historie hamiltonovských grafů

5.1 Nejstarší zmínky – Jezdcova procházka

Nejstarší známým případem, kdy se lidé zabývali hamiltonovskými grafy, je matematický a šachovnicový problém jezdcovy procházky (v angličtině knight´s tour). Jezdec se pohybuje po šachovnici podle šachových pravidel. Tedy podle velkého písmene L, o dvě pole dopředu a jedno do strany, případně jedno dopředu a dvě do strany. Jezdec se pohybuje po čtvercové šachovnicové síti, typicky 8x8 polí. Z jeho výchozí pozice se snaží pohybovat tak, aby navštívil všechna pole právě jednou a posledním tahem se vrátil do výchozího bodu. Existují různé variace tohoto problému, čtvercová síť může mít různý počet polí. Eventuálně může být síť obdélníková.

Mezi nejstarší známé příklady jezdcovy procházky se řadí báseň *Kavyalankara od básníka Rudrata*, žijícího v 9. století našeho letopočtu v oblasti Kašmíru nacházejícího se na severu dnešní Indie. Jeho báseň je rozdělena do slabik a slova dávají smysl, jen pokud se nalezne správný směr podle pohybu jezdce. [25]

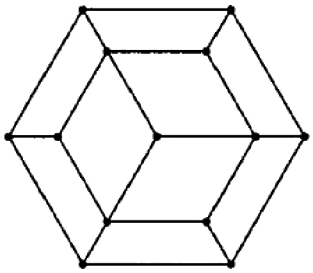
Další příkladem je arabský rukopis, který napsal *Abu Zakariya Yahya ben Ibrahim al-Hakim*, s názvem *Nuzhat al-arbab al-'aqulfi'sh-shatranj al-manqul* (Rozkoš inteligentních, popis šachů). Obsahuje dva tahy jezdcovy procházky o rozměrech šachovnice 8x8 polí, jeden od Alího C. Maniho, jinak neznámého šachisty, a druhou od al-Adli ar-Rumiho, který žil kolem roku 840 a o němž je známo, že napsal knihu o šatrandži (tehdy populární formě šachu), která se však dochovala pouze ve výňatcích v tomto a dalších rukopisech. [21]



Obr. 1: Jezdcova procházka od arabských šachistů [21]

5.2 Thomas Kirkman – Včelí plástev

Reverend Thomas Penyngton Kirkman (1806-1895) byl farářem ve farnosti Croft-with-Southworth v anglickém Lancashiru. Ve volném čase se věnoval matematice. Byl vynikajícím matematikem, jehož významný přínos k teorii grup (group theory) i teorii blokových konstrukcí (the theory of block design) zůstal z velké části nedoceněn. Mezi jeho další zájmy patřilo studium mnohostěnů. Zabýval se také myšlenkou, která by určila, pro které mnohostěny existuje uzavřený cyklus procházející každým vrcholem. Uvedl postačující podmínku pro existenci takového cyklu, ta se však ukázala jako nesprávná. [45] Ukázal také, že každý mnohostěn se sudým počtem stěn, ale lichým počtem vrcholů, takový cyklus nemá. Takový mnohostěn lze získat, „rozříznutím buňky včelí plástve na dvě části", čímž vznikne diagram (Obr. 2).



Obr. 2: Kirkmanova plástev [45]

Takový diagram je bipartitní graf s lichým počtem vrcholů, a pro takový graf nemůže existovat žádný cyklus procházející každým jeho vrcholem, s návratem do výchozího vrcholu. [45] Další kdo se touto problematikou zabýval, byl William Hamilton.

5.3 William Hamilton – Icosian game

Sir William Rowan Hamilton byl irský matematik a astronom. Narodil se roku 1805 a zemřel 1865 na problémy spojené se dnou. [28] Od raného věku se projevoval jako geniální dítě. Své schopnosti pravděpodobně zdědil po matce. Zpočátku se věnoval především studiu jazyků. S pomocí svého strýce se naučil latinsky, řecky a hebrejsky. Než mu bylo 12, věnoval se také francouzštině, italštině,

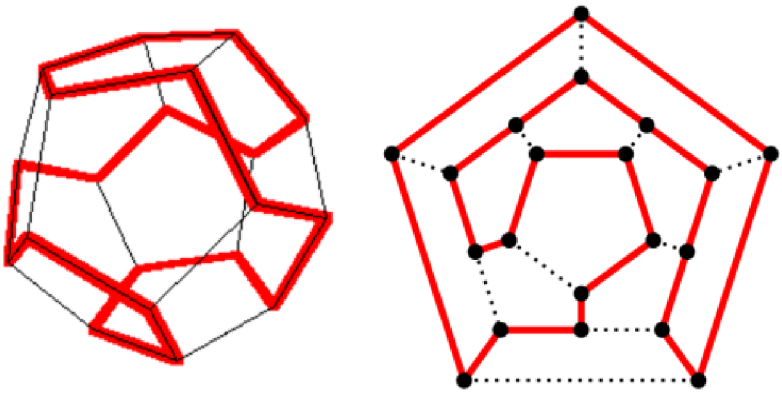
arabštině, syrštině, perštině a sanskrtu. [43] Byl vynikající student. V osmi letech se zúčastnil matematické soutěže proti o rok staršímu Zerahu Colburnovi. Kde prohrál, a to ho motivovalo se více věnovat matematice. [28] V 18 letech začal studovat na univerzitě Trinity College v Dublinu, kde naprosto exceloval. V 21 letech dostal ještě jako nepromovaný student titul a místo profesora astronomie v observatoři Dunsink. Zde působil po zbytek života. Nejvíce se však věnoval matematice a skrze ni přinesl významné poznatky do optiky a fyziky. V roce 1835 byl povýšen do šlechtického stavu. [43] Osobně za největší svůj přínos považoval svůj objev a následující práci na kvaternionech. Hamilton chtěl matematiku komplexních čísel rozšířit o čísla s vyššími řády. Na komplexní čísla pohlížel jako body v dvojrozměrné rovině. Dlouho se mu nedařilo dosáhnout pokroku s trojrozměrnými čísly. Následně měl však úspěch s čísly čtyřrozměrnými. K tomu se váže slavná historka. Při procházce se ženou v Dublinu napadl Hamiltona vzorec, jak řešit kvaterniony. Neměl však čím si vzorec poznamenat, a aby ho nezapomněl vyryl ho do kamenné zdi mostu. Dnes se kvaterinony využívají například v počítačové grafice pro manipulaci s 3D objekty za pomoci čtyřrozměrných matic. [14]

Obr. 3: Fotografie Williama Hamiltona [44]

Obr: 4: Pamětní deska se vzorcem na mostě [7]

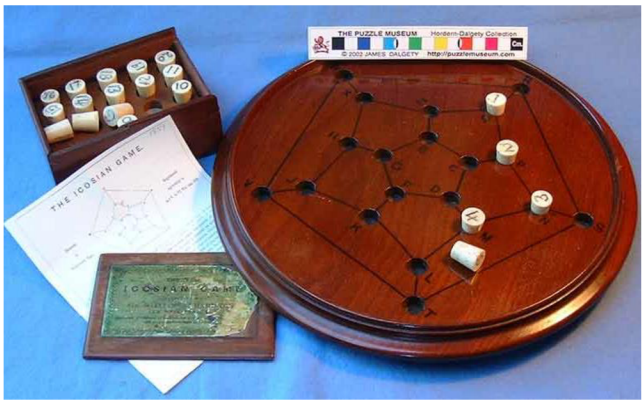
Hamilton se v návaznosti na Kirkmanovu práci zabýval jednotahovými cykly ve dvanáctistěnu. V roce 1857 vymyslel hru *Icosian game*, která spočívala v hledání cyklů (dnes známé jako hamiltonovské grafy). Licenci na výrobu hry prodal londýnské firmě John Jacques a synové. Ta se zabývala výrobou hraček a šachových

setů vysoké kvality. Prodal ji za 25 britských liber. [5] Tehdy bylo Irsko součástí Velké Británie (bylo její součástí až do roku 1921). Na dnešní peníze (k 29.9.2021) je to přibližně 3 335 amerických dolarů, tedy necelých 73 000 CZK. [9]

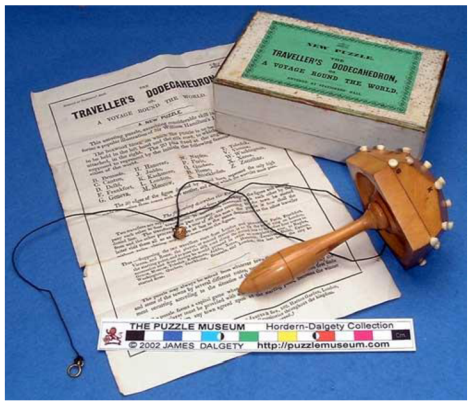


Obr. 5: Hamiltonovská kružnice v trojrozměrném a dvojrozměrném modelu dvanáctistěnu [15]

Základem hry byl rovinný síťový model dvanáctiúhelníku. Ten je tvořen 20 vrcholy a 30 hranami. Od počtu vrcholů je odvozeno jméno hry. V latině *Icosa* znamená dvacet. [6] První verze hry se hrála na ploché herní desce ze dřeva. Na ní byly otvory pro slonovinové kolíčky, a otvory byly propojeny čarami. Podle osazení kolíčků se měnil výchozí prohledávaný graf. V pravidlech bylo uvedeno 15 příkladů hlavolamů. Jeden hráč zadal hlavolam a druhý hráč ho řešil. V pozdější verzi hry nazvané Cestovatelův dvanáctistěn (The Traveler's Dodecahedron), byla deska nahrazena hřibovitým tvarem. Třicet hran představuje jediné cesty, po kterých je možné projít. Ty spojovali 20 slonovinových kolíků, které představují města. Dva cestovatelé se vydají na návštěvu 4 sousedních měst. Jeden se vrací domů a druhý pokračuje v cestě kolem světa a snaží se navštívit všechna zbývající města pouze jednou. Provázkem se vyznačila cesta, po které se cestovatel vypravil. [4] [37]



Obr. 6: Původní verze hry [4]



Obr. 7: Vylepšená verze hry [4]

Ačkoliv se před Hamiltonem daným tématem zabývali i jiní vědci, jsou dnes hamiltonovské grafy pojmenovány po něm. *„V pozdější době vznikly spory o to, kdo byl autorem myšlenky zkoumat kružnice dvanáctistěnu. Je třeba říci, že zatímco Euler, Vandermonde a Hamilton zkoumali konkrétní případy grafů, Kirkman byl první, kdo se pokusil o jistá zobecnění. Nicméně na počest Hamiltonových prací dnes hovoříme o hamiltonovské kružnici, resp. hamiltonovském grafu.“* [37]

6 Třídy složitosti - P, NP, NP- úplné

Hamiltonovské grafy spadají do takzvaných NP-úplných problémů (NP-complete). [40] Obecný algoritmus pro řešení hamiltonovských grafů v polynomiálním čase pravděpodobně neexistuje. [32]

Algoritmy lze dělit do skupin podle toho, jak obtížné je s jejich pomocí vyřešit určitý problém. Samozřejmě musí být použitý takový algoritmus, kterým je daný problém, alespoň teoreticky řešitelný. Tyto skupiny se nazývají třídy složitosti. Ty udávají, kolik času a paměti je potřeba pro nalezení řešení. Často se složitost uvádí pro nejhorší možnou situaci, která může při řešení nastat. Tedy nejhorší časová složitost říká, jakou maximální dobu bude algoritmus pracovat, než nalezne řešení. (Řešení nutně neznamená, že algoritmus nalezne graf, ale také může dojít k výsledku, že graf nemůže existovat.)

K nalezení řešení se používá Turingův stroj. Turingův stroj je teoretický počítač. Má nekonečnou zapisovací pásku, která slouží jako paměť. Na ní se pohybuje hlava, která má možnost z ní číst a také na ní zapisovat. Hlava obsahuje výpočetní jednotku

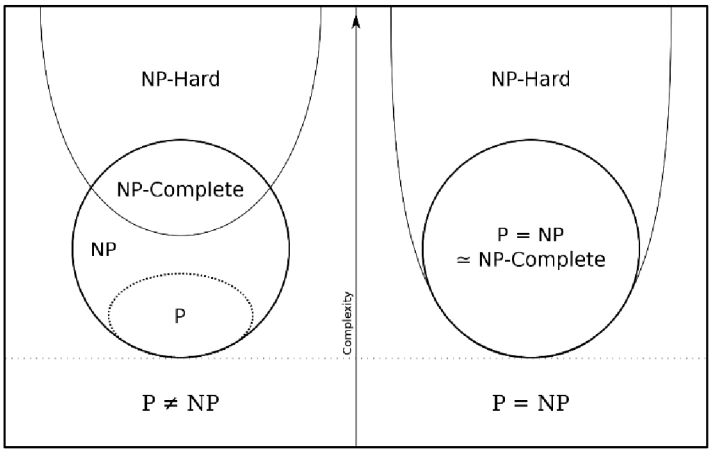
s programem. [39] Ten se chová jako konečný automat. Konečný automat, nemá žádnou vlastní paměť. Zná pouze svůj aktuální stav. Na základě vstupu prochází určité stavy, ve kterých provádí úkony a výsledek vrací na výstup. Turingův stroj může být různě rozšířen – může využívat obě strany pásky, může mít více pásek, a tím pádem i hlav, které pásky čtou. Nebo může být nedeterministický. To znamená, že může řešit více větví, do kterých se algoritmus může dostat současně. Například pří prohledávání hamiltonovského grafu se nedeterministický stroj při každém rozhodování, kterou cestou se vydat, rozvětví a jednotlivé větve se počítají odděleně, ale současně. Veškeré modifikované Turingovy stroje jsou ale převeditelné na základní Turingův stroj s jednou hlavou a jednou páskou.

P (polynomiální) problémy, jsou takové úlohy, které lze vyřešit v polynomiálním čase. Jedná se o třídu časové složitosti *O*, pro kterou platí *O(n^k),* kde *k* je konstanta a *n* je množství vstupních dat, například počet uzlů v grafu. Pro časovou složitost *O(1)* je jedno, kolik je vstupních dat, čas pro vyřešení je pokaždé stejný – konstantní. Některé další složitosti, které jsou jednodušší než polynomiální, jsou logaritmická složitost *O(log n)*, lineární *O(n)*, lineametrická *O(n log n)*, kvadratická *O(n^2)*. [34]

Obecně lze říct, že se vyplatí používat algoritmy s maximálně polynomiální složitostí. Algoritmy vyšších složitostí jsou většinou použitelné jen pro nízká čísla. *P problémy* tedy mají menší složitost než exponenciální problémy – *O(k^n)*. Neznamená to ale, že pro jakýkoliv P problém lze snadno nalézt řešení s vynaložením rozumného množství výkonu a času. S dostatečně vysokými proměnnými může být i *P problém* stejně nevyřešitelný. *NP problémy* jsou složitější na vyřešení. K jejich vyřešení je zapotřebí neúměrně více paměti nebo času. U *NP problémů* je výrazně náročnější najít správné řešení problému než ověřit, zdali je nějaké potenciální předložené řešení správné, či nikoliv. Pokud by se řešily na výše zmíněném nedeterministickém Turingově stroji, kdy při každém větvení se řeší každá další větev současně se všemi ostatními, aniž by se jednotlivé větve musely dělit o paměť nebo výpočetní výkon, což by platilo pro neomezený počet větví, byly by *NP problémy* také vyřešeny v polynomiálním čase. Ovšem reálné výpočetní jednotky nic takového nedokážou. *NP* (nedeterministicky polynomiální) *problémy* lze tedy ověřit v polynomiálním čase, ale ne je vyřešit. Lze to přiblížit na příměru

řešení s věštcem. Pokud by se řešící algoritmus při každém větvení zeptal věštce, který by byl pokaždé schopný uhodnout správnou cestu, tak by se k výsledku dostal v polynomiálním čase. Obecně se předpokládá, že platí tvrzení $P \subseteq NP$. P je podmnožinou *NP*, ale *P != NP*, tedy že třída *P* se nerovná třídě *NP*. [32] Z toho lze vyvodit, že pravděpodobně neexistuje algoritmus, který by dokázal řešit *NP problémy* v polynomiálním čase za pomoci Turingova stroje, aniž by byl nedeterministický. Doposud ovšem nebyl představen důkaz, který by tuto nerovnost potvrzoval. Vyřešení této otázky je zařazeno mezi 7 problémů tisíciletí - Millennium Prize Problems. Jedná se o sedm nejdůležitějších otevřených matematických problémů. Seznam vytvořil Clayův matematický ústav sídlící v USA v roce 2000. [2] Volně navazuje na 23 Hilbertových problémů představených v roce 1900 Davidem Hilbertem.

Pokud by ovšem platila rovnice *P = NP*, znamenalo by to, že i problémy třídy *NP* jsou rozumně řešitelné. To by představovalo problém například pro šifrovací algoritmy současné kryptografie. Bylo by je možné řešit ve stejném čase, jako je ověřovat (vložení privátního klíče a ověření jeho správnosti).



Obr. 8: Diagram možných vztahů P, NP a NP – úplných problémů [10]

NP-úplné problémy jsou ty nejsložitější problémy ze třídy *NP*. Musí splňovat dvě podmínky. Za prvé musí náležet do třídy *NP* a za druhé jakýkoliv problém ze třídy *NP* musí být převoditelný na *NP-úplný* problém. Pokud platí pouze druhá podmínka, nazývá se *NP-těžký* problém. [32] Tedy všechny NP-úplné problémy jsou

zároveň i NP-těžké. NP-těžké problémy nemusí spadat do NP problémů, ale mohou být složitější.

NP-úplné problémy je díky převoditelnosti možné převést na jakýkoli jiný *NP-úplný* problém. Převoditelnost znamená, že pokud máme problémy T a U, tak můžeme problém T převést za pomoci deterministického polynomiálního algoritmu na problém U. [27] Pokud by tedy platilo *P = NP,* mělo by to za následek: „*Pokud by byl některý NP-úplný problém řešitelný v polynomiálním čase, pak existuje polynomiální algoritmus pro každý problém z NP.*“ [32] Jak již bylo řečeno, hamiltonovské grafy spadají do třídy *NP-úplných* problémů.

6.1 Další NP–úplné problémy

SAT (SATISFIABILITY) splnitelnost booleovských formulí – Problém rozhodnutí, jestli je pro daný logický výraz (například $\phi = \neg x_1 \wedge (x_2 \vee x_3) \wedge x_4$) možné, za určité kombinace proměnných, aby nabyl hodnoty TRUE. Jedná se o první určený NP – úplný problém. Popisuje ho Cook-Levinova věta, publikovaná v roce 1973. [3]

3-SAT – Varianta problému SAT. Výraz musí být v konjunktivní normální formě. To znamená, že výraz je konjunkcí (∧) klauzulí. Klauzule je disjunkcí (∨) literálů. V každé klauzuli musí být právě tři literály. Literál nabývá podoby *x* nebo ¬*x*. Příkladem 3-SAT výrazu je $\phi = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_4)$.

problém nezávislé množiny (IS) - Určení, jestli v grafu existuje počet *k* uzlů, které nejsou mezi sebou spojeny žádnou hranou.

problém batohu (Knapsack problem) – Existuje množina věcí, z nichž každá má svoji hodnotu (cenu) a svoji hmotnost. Dále existuje batoh, ten má svoji maximální nosnost, která nesmí být překročena. Úkolem je zjistit, jakou kombinaci věcí lze v batohu odnést tak, aby celková cena odnesených věcí byla co nejvyšší.

barvení grafu (Graph coloring) – Problém spočívá v tom, zda lze obarvit vrcholy grafu předem daným počtem barev. Přičemž žádné dva sousední vrcholy nesmějí být obarveny stejnou barvou.

problém kliky (CLIQUE) – Klika je úplný podgraf grafu. Určuje se, jestli existuje v grafu klika s minimálním počtem *k* vrcholů.

vrcholové pokrytí (Vertex cover) – Určení, jestli v grafu existuje množina vrcholů o počtu *k* tak, aby každá hrana grafu měla alespoň jeden svůj vrchol náležící této množině.

problém obchodního cestujícího (Traveling salesman problem) – Města jsou propojena cestami, která mají různou délku (cenu). Úkolem obchodního cestujícího je projít co nejkratší cestou všechna města a každé z nich navštívit minimálně jednou, a nakonec se vrátit do počátečního města. Od hamiltonovského grafu se úloha liší tím, že je možné navštívit uzel vícekrát.

Subset sum– Určení, jestli je možné z dané množiny čísel vybrat podmnožinu takovou, aby součet všech čísel z podmnožiny dával výsledek *k*. Například z množiny {1,2,4,7,8} vybrat čísla, která dávají výsledek *k* = 13. Zde řešení existuje (2+4+7).

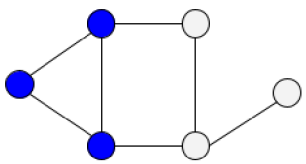
N-puzzle – Jedná se o logickou hru. Číslované kameny se přesouvají ve čtvercovém poli, jeden kámen chybí. Cílem je určit, jestli existuje taková sada přesunů, aby na konci byly kameny seřazeny za sebou podle jejich čísel.

Příklady jsou převzaty z: [11] [35]

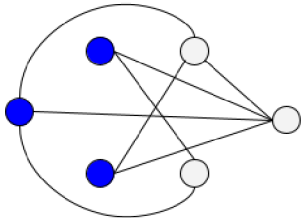
6.2 Převoditelnost – příklady

6.2.1 Převod kliky na nezávislou množinu

Výchozí graf obsahuje kliku velikosti $k = 3$. V grafu budou prohozeny hrany a nehrany (pomyslná hrana, která by mezi dvěma uzly grafu mohla existovat, ale v současnosti neexistuje). Tedy vznikne doplněk původního grafu. Z grafu kliky vznikne problém nezávislé množiny *k*. [23]



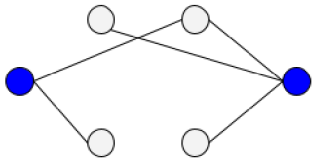
Obr. 9: Graf kliky



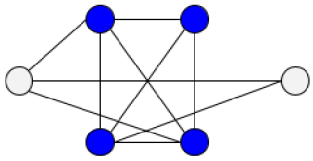
Obr. 10: Graf nezávislé množiny

6.2.2 Převod vrcholového pokrytí na problém kliky

Graf má vrcholové pokrytí 2. Doplněk tohoto grafu obsahuje kliku velikosti k = 4. Pokud budou odečteny od celkového počtu vrcholů grafu všechny vrcholy kliky doplňkového grafu, budou získány vrcholy, které tvoří vrcholové pokrytí grafu původního. [46]



Obr. 11: Graf vrcholového pokrytí



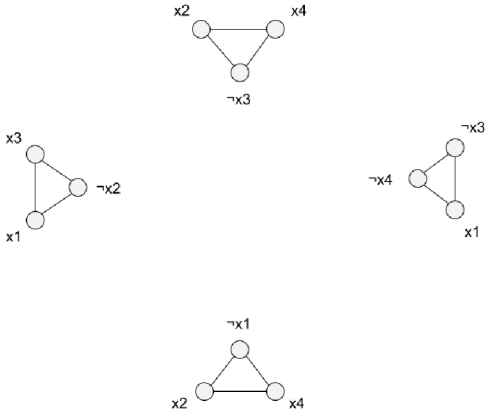
Obr. 12: Graf kliky

6.2.3 Převod 3-SAT na problém IS

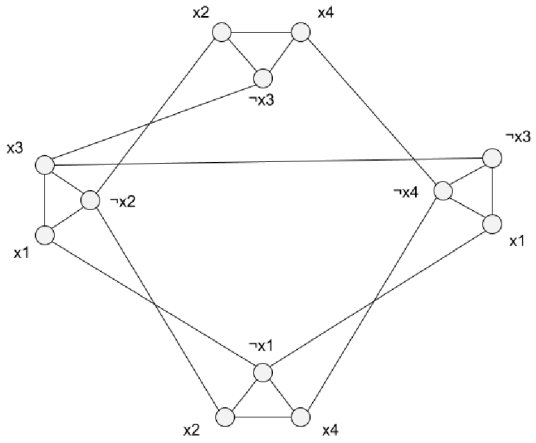
Následující 3-SAT výrok bude transformován na problém nezávislé množiny.

$$(x1 \vee \neg x2 \vee x3) \wedge (x2 \vee \neg x3 \vee x4) \wedge (x1 \vee \neg x3 \vee \neg x4) \wedge (\neg x1 \vee x2 \vee x4)$$

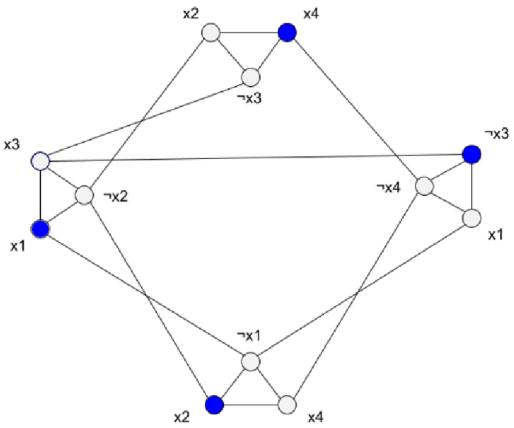
Za každý výskyt literálu je přidán do grafu jeden vrchol. Vrcholy odpovídající výskytům literálů patřící do stejné klauzule budou propojeny hranami. (Obr. 13) Také budou propojeny literály stejného čísla a opačných hodnot (x1 spojeno s ¬x1). (Obr. 14) Číslo *k* bude odpovídat počtu klauzulí. V tomto případě *k* = 4. Jestliže je výrok splnitelný, musí v každé klauzuli mít alespoň jeden literál hodnotu 1. To splňuje například: x1 = 1, x2 = 1, x3 = 0, x4 =1. Z každé klauzule bude vybrán jeden literál, který má při ohodnocení hodnotu 1. Tyto vybrané vrcholy poté tvoří nezávislou množinu velikosti *k*. (Obr. 15) [35]



Obr. 13: Spojení literálů v rámci klauzule



Obr. 14: Spojení klauzulí



Obr. 15: Vrcholy nezávislé množiny $k = 4$

7 Prohledávací algoritmus

Prohledávací algoritmus prochází a vyhodnocuje zkoumaný graf, a na základě tohoto průchodu určí, jestli je graf hamiltonovský.

7.1 Backtracking

Algoritmus backtrackingu je v češtině známý jako algoritmus *prohledávání do hloubky* nebo algoritmus *zpětného vyhledávání*. Jde o jeden ze základních algoritmů pro prohledávání grafů. Jeho přednosti jsou relativní jednoduchost

implementace a názornost postupu při jednotlivých krocích. Jelikož je takto jednoduchý, jeho postup řešení není moc efektivní. Řadí se mezi algoritmy, které nalézají řešení takzvaně hrubou silou. To znamená, že zkouší hledat řešení metodou pokus-omyl. Vybere si jakoukoli z možných cest, tou pokračuje do té doby, dokud je to možné. Jestliže tato cesta byla správná, je algoritmus ukončen. Pokud ne, vrátí se na poslední rozhodování, kde zbyla ještě nějaká nevyzkoušená cesta a tou pokračuje. V nejhorším možném případě algoritmus projde všechny možné cesty, to ho činní značně neefektivním. Níže je uveden slovní popis backtrackingu pro nalezení hamiltonovského grafu.

7.2 Algoritmus backtrackingu v krocích

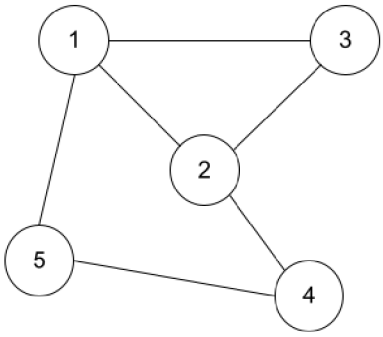
Legenda:
Pokud není uvedeno jinak, jednotlivé kroky na sebe navazují odshora dolů. Příklad průchodu algoritmem je znázorněn na tabulkách 1 až 14.

1. Začni ve výchozím uzlu a ulož ho do zásobníku – pokud není předem daný výchozí uzel, většinou se vybere první uzel ze seznamu, případně se vybere náhodně.
2. Je možné pokračovat dál? Ano – bod 3, Ne – bod 5
3. Pokračuj do prvního dalšího nenavštíveného možného neslepého uzlu.
4. Ulož současný uzel do zásobníku. bod 2
5. Byly všechny uzly navštíveny právě jednou? Ano – bod 9, Ne – bod 6
6. Odeber poslední uzel ze zásobníku. Lze se vrátit na předchozí uzel? (v zásobníku se ještě nějaký uzel nachází?) Ano – bod 7, Ne – bod 11
7. Pokud jsou v současném řádku některé uzly označeny jako slepé, označ je zpět na normální stav. A vrať se o uzel zpět, tedy na uzel, co je v zásobníků navrchu.
8. Na stávajícím řádku označ uzel, ze kterého proběhl návrat jako slepý. bod 2
9. Vede ze současného uzlu nenavštívená hrana do výchozího uzlu? Ano – bod 10, Ne – bod 6
10. Projdi do výchozího uzlu, výsledek = hamiltonovský graf EXISTUJE, bod 12

11. Hamiltonovská kružnice nebyla nalezena, výsledek = hamiltonovský graf NEEXISTUJE
12. Předej výsledek a skonči

7.3 Ukázkový příklad s tabulkami

Aplikace backtrackingu na konkrétním grafu: Graf je neorientovaný a všechny hrany mají stejnou váhu. Úkolem algoritmu je ověřit, zda lze v grafu nalézt hamiltonovskou kružnici. Pokud je nalezena, jedná se o hamiltonovský graf. Algoritmus, pokud bude mít na výběr, zvolí vždy uzel s nižším číslem. Jednotlivé kroky prohledávání jsou znázorněny tabulkami níže.



Obr. 16: Vizuální znázornění prohledávaného grafu

Legenda k tabulkám:
Na osách jsou vypsány všechny uzly grafu. Jedničky představují hrany mezi uzly. Nuly značí neexistenci hran. Písmena *x* říkají, že se jedná o ten samý uzel. Žlutá barva značí, v jakém uzlu se právě nacházím. Modře jsou označeny uzly, které už byly navštíveny a nelze do nich vkročit znovu. Zelené uzly znázorňují současnou cestu. Červená značí slepé uličky, ze kterých se algoritmus právě vrátil. Slepé uzly se postupně odmazávají, podle toho, jak daleko zpět se algoritmus vrací.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 1: Z výchozího uzlu 1 se pokračuje první možnou hranou. Tedy do uzlu 2.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 2: Z uzlu 2 se jde do uzlu 3.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 3: Z uzlu 3 není možné pokračovat, je nutný návrat do předchozího bodu.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 4: V uzlu 2 vybrána další hrana, pokračuje se do uzlu 4.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 5: Z uzlu 4 se pokračuje do uzlu 5.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 6: Z uzlu 5 již nelze dále pokračovat, nebyly však navštíveny všechny uzly. Návrat do 4.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 7: Nelze dále pokračovat. Návrat do 2.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 8: Nelze dále pokračovat. Návrat do 1.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 9: Cesta do uzlu 2 se ukázala jako slepá. Postup do uzlu 3.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 10: Postup z uzlu 3 do uzlu 2.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 11: Postup z uzlu 2 do uzlu 4.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 12: Postup z uzlu 4 do uzlu 5.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 13: Z uzlu 5 již nelze dále pokračovat, byly navštíveny všechny uzly. Kontrola jestli existuje nepoužitá hrana do výchozího bodu 1.

	1	2	3	4	5
1	x	1	1	0	1
2	1	x	1	1	0
3	1	1	x	0	0
4	0	1	0	x	1
5	1	0	0	1	x

Tab. 14: Nevyužitá hrana mezi uzly 1 a 5 existuje. Algoritmus se navrátil do výchozího bodu. Hamiltonovská kružnice existuje. Jedná se o hamiltonovský graf.

Tento příklad ukazuje hlavní výhodu backtrackingu. Jeho relativní jednoduchost a názornost. Největší nevýhodou je jeho složitost *O(N!)*. Tedy složitost N faktoriál, kde N je počet uzlů. Z tabulky 15 je patrné, že stačí malé množství prohledávaných uzlů, a kvůli faktoriálové funkci, počty kroků velmi rychle rostou. Sice se jedná o nejhorší možné průchody, ale algoritmus musí fungovat vždy.

N	N!		N	N!
1	1		9	362880
2	2		10	3628800
3	6		11	39916800
4	24		12	479001600
5	120		13	6227020800
6	720		14	87178291200
7	5040		15	1,30767E+12
8	40320		16	2,09228E+13

Tab. 15: Znázornění rychlosti růstu faktoriálu

7.4 Další prohledávací algoritmy

Warnsdorffův algoritmus - Jedná se o vylepšení backtrackingu. Rozdíl je, že Warnsdorffův algoritmus si vybírá, do kterého uzlu bude pokračovat. Upřednostní ty, které jsou jen obtížně dostupné a mají tendenci být nenavštíveny. Tím se algoritmus častěji vyvaruje procházení dlouhých slepých uliček.

Branch and bound B&B - Algoritmus větvení a mezí. Převedení prohledávaného grafu na graf stromu, kde kořen je výchozí vrchol původního grafu. A

následně jeho prohledávání jako stromu. Tento algoritmus ve svém základu není příliš efektivní.

Zkoumáním, klasifikací a případným zefektivněním algoritmů pro řešení hamiltonovských grafů, se zabývá projekt *Flinders Hamiltonian Cycle Project* z Flinders University. Věnuje se algoritmům hledajícím hamiltonovské kružnice na grafech do 5000 uzlů. Jako vcelku efektivní algoritmy uvádějí například: **Determinant Interior Point Algorithm, Snakes and Ladders Heuristic, Wedged-MIP heuristic** [12]

8 Použité technologické nástroje

Vlastní program je napsaný v jazyku Java s využitím grafické knihovny Swing. Použitý algoritmus backtrackingu využívá rekurzivní volání a zásobník. V této kapitole jsou popsány vlastnosti těchto technologických nástrojů.

8.1 Java

Jazyk Java vydala americká společnost *Sun* na začátku roku 1996. Vznikla přejmenováním programovacího jazyka Oak, poté co vývojáři zjistili, že již programovací jazyk s takovým jménem existuje. Oak byl ve vývoji od roku 1991. Jednalo se o jednouchý programovací jazyk, který měl původně sloužil pro vývoj aplikací obsluhující drobnou domácí elektroniku (například dálkové ovladače). O toto využití jazyka však nebyl zájem. Poté byl jazyk (již Java) využit pro vývoj webového prohlížeče HotJava. Java (ve verzi 1.0) se však ukázala jako nedostatečná pro vývoj aplikací, a musela být značně rozšířena. Nejzásadnější nedostatky vyřešila verze 1.1. Na jazyku se dále pracovalo. Verze 5.0 z roku 2004 například přidala výčtové datové typy nebo konstrukci *for-each*. Verze Java SE 8 (SE = standard edition) z roku 2014 podporuje "funkční" styl jazyka programování, který

usnadňuje vyjadřování výpočtů, které lze provádět souběžně. [19] Java je stále ve vývoji, k roku 2021 vyšla zatím poslední verze Java SE 17.

Version	Year	New Language Features	Number of Classes and Interfaces
1.0	1996	The language itself	211
1.1	1997	Inner classes	477
1.2	1998	The `strictfp` modifier	1,524
1.3	2000	None	1,840
1.4	2002	Assertions	2,723
5.0	2004	Generic classes, “for each” loop, varargs, autoboxing, metadata, enumerations, static import	3,279
6	2006	None	3,793
7	2011	Switch with strings, diamond operator, binary literals, exception handling enhancements	4,024
8	2014	Lambda expressions, interfaces with default methods, stream and date/time libraries	4,240
9	2017	Modules, miscellaneous language and library enhancements	6,005

Obr. 17: Vývojové verze Javy [19]

8.1.1 Základní vlastnosti jazyka Java

Java je objektově orientovaný jazyk. Objekty (třídy) jsou reprezentací jakékoliv požadované entity. Objekty mají vlastnosti popsané a uložené v atributech. Třídy manipulují s atributy a komunikují s ostatními třídami za pomoci metod(funkcí). Java však není čistě objektově orientovaný jazyk (ne všechno je třída). Protože obsahuje primitivní datové typy (byte, short, int, long, float, double, char, boolean). Ty jsou používány, protože jsou rychlejší a méně náročné na paměť oproti standartním objektům, které v Javě také existují (Integer, Char, Double atd.) [20]

Syntaxe vychází z jazyka *C* a *C++*. Na rozdíl od C++ Java nepodporuje vícenásobnou dědičnost, přetěžování operátorů, ukazatele (pointer) nebo strukturu *goto*. To pomáhá zvýšit jednoduchost a pochopitelnost jazyka.

Java je statický silně typovaný jazyk. Statické typování vyžaduje deklaraci proměnných uvedením datového typu. (Datový typ určuje množinu hodnot, jakých

může proměnná nabývat. Například proměnná typu integer může obsahovat pouze celé číslo.) Díky vyžadování datových typů je jasné již při překladu programu, že každá proměnná používá pouze datotypově odpovídající operace. Zatímco u dynamicky typovaných programů probíhá typová kontrola až za běhu programu, což je více výkonnostně a paměťově náročné. Silné typování zakazuje funkcím kombinovat argumenty různých datových typů. Například nelze „číslo“ ve Stringu násobit číslem typu integer. To pomáhá přehlednosti výsledného kódu.

8.1.2 Spuštění aplikací v Javě

Pro spuštění aplikací je nutné převést zdrojový kód (kód napsaný programátorem) na kód, se kterým dokáže pracovat konkrétní elektronické zařízení. To především závisí na hardwaru a operačním systému. Programovací jazyky se dělí na kompilované, interpretované, nebo jejich kombinace.

U kompilovaných jazyků je zdrojový kód přeložen překladačem do strojového kódu, poté je možné program spustit. Strojový kód je spustitelný soubor. Po zavedení do paměti ho může přímo provádět procesor počítače. Kompilované programy jsou mnohem rychlejší a efektivnější než ty spouštěné interpretem. Jsou nepostradatelné zejména v případech, kde jsou požadavky na co nejvyšší výkon aplikací. Jejich nevýhodou je, že nejsou platformově nezávislé. [36]

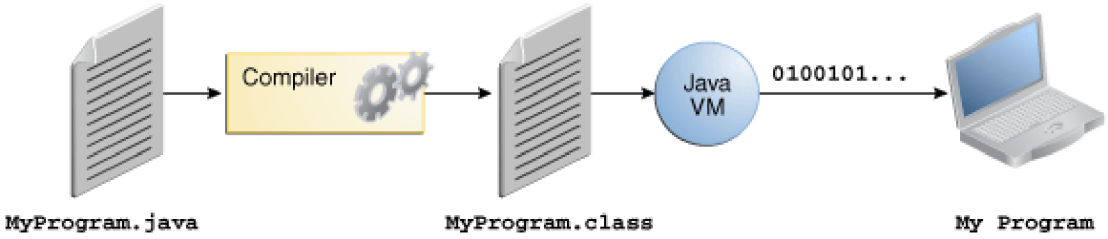
Pro interpretované jazyky je nezbytný zdrojový kód a zvláštní program - *interpret*, který zdrojový kód provádí (interpretuje). Výhodou je obvykle dobrá přenositelnost programu na jinou platformu (pokud pro ni existuje interpret příslušného jazyka). [36]

Kombinovaný překlad – zdrojový kód překladač přeloží do mezikódu, ten je předán interpretu, který ho spustí. [36]

Java využívá kombinovaný překlad. Kompilátor *javac* zkompiluje původní zdrojový kód (.java) na bytecode do souboru (.class). Bytecode je podobný assembleru, skládá se z jednoduchých instrukcí. Tento soubor je přenositelný na jakoukoli platformu. JMV (Java Virtual Machine) interpretuje soubor *class* na takový soubor, kterému bude daná platforma rozumět. Samozřejmě JMV musí danou platformu podporovat. V současnosti je mezikód zpočátku interpretován a na

základě statistik je později proveden překlad často používaných částí do strojového kódu (včetně dalších dynamických optimalizací). [29]

JVM je součástí balíků Java. Ty se dělí na uživatelské (JRE) a vývojářské (JDK). JRE (Java Runtime Environment) pro běžné uživatele slouží ke spouštění javovských programů. JDK (Java Developer Kit) je balík nástrojů pro psaní programů v jazyce Java. Oproti uživatelskému balíku obsahuje několik nástrojů navíc, například kompilátor nebo debugger. [16]



Obr. 18: Kombinovaný překlad Javy [29]

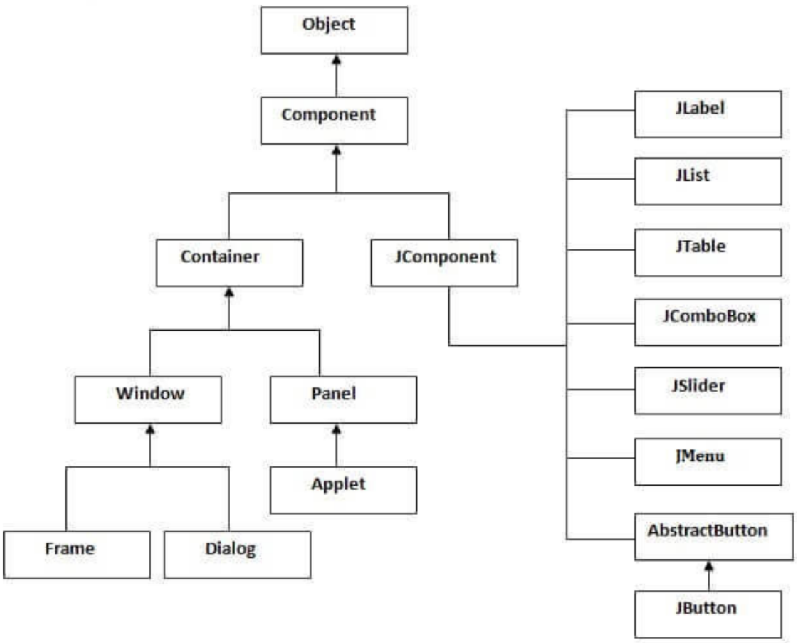
Java podporuje automatickou správu paměti. O tu se stará garbage collector. Programátor si inicializuje objekty, těm je přiděleno místo v paměti (alokace paměti). Pokud objekt již není používán (nevede na něj žádná reference), je nepotřebný, ale stále zabírá místo v paměti. Pokud by někdo objekt neodstranil a tím paměť neuvolnil (dealokace paměti), vznikaly by úniky paměti (memory leak). Během běhu programu by se úniky paměti mohly postupně zvětšovat, což by ubíralo paměť, kterou by mohly využít ostatní programy. Nebo by to mohlo vést až k pádu programu, protože by vyžadoval více paměti, než je mu možné přidělit, a operační systém by ho musel ukončit. To je problém zejména u dlouho běžících programů (například serverové aplikace). V jazyku Java se o dealokaci nemusí starat programátor, ale automaticky ji provádí garbage collector. Ten hlídá každý objekt, jestli na něj vede reference. Pokud ne, je collectorem odstraněn. Nevýhodou je, že garbage collector na svůj provoz spotřebovává část výpočetního výkonu. Většinou to nevadí, ale u aplikací, které potřebují dosáhnout co nejvyššího výpočetního výkonu, se garbage collector stává přítěží.

8.2 Swing

Grafická knihovna Swing. Vychází ze starší knihovny AWT (Abstract Window Toolkit) a řeší její nedostatky. „*Swing komponenty jsou označovány jako lehké, zatímco AWT komponenty jsou označovány jako těžké*“. [33] Těžké (heavyweight) komponenty jsou kresleny operačním systémem. Lehké (lightweight) komponenty se starají o své vykreslení samy. To zajistí, že na všech systémech vypadají stejně. *„Jedna z největších výhod Swingu je jeho platformová nezávislost. Swingové komponenty jsou napsány ze 100 % v Javě. To znamená, že se komponenty chovají a vypadají vždy stejně nezávisle na platformě. Snižuje to potřebu testovat a ladit aplikace na každé cílové platformě*.“ [33] O změnu vzhledu se ve Swingu stará sada *Look and Feel*. Dalším rozdílem mezi těžkými a lehkými komponenty je práce s osou *Z,* která řeší hloubku scény. Lehké s ní pracují samy, zatímco těžké to nechávají na systému. Knihovny se navzájem doplňují. Některé komponenty plní stejnou funkci a jsou zastoupeny v obou knihovnách. Pro lepší rozlišitelnost veškeré swingovské komponenty začínají písmenem *J,* například JLabel, JComboBox. [33]

JComponent je rodičovský objekt pro všechny Swing komponenty. JComponent rozšiřuje AWT třídu Container. Z tohoto důvodu, Swing nejlépe popsán jako vrstva nad AWT, nikoli jako náhrada za něj. [33] Swing přidal podporu pro dvojitou vyrovnávací paměť (double buffering). Nejdříve se objekty postupně vykreslí do vyrovnávací paměti mimo obrazovku a následně se celý obraz vykreslí

uživateli najednou. Zlepšuje to plynulost vykreslení, pomáhá zamezit blikání objektů. [30] Na rozdíl od AWT knihovna Swing využívá MVC architekturu.



Obr. 19: Dědičnost Swing komponent [17]

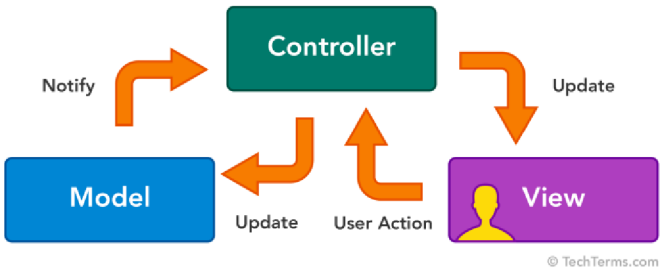
8.3 MVC architektura

Model-view-controller architektura. Skládá se ze tří logických částí: model, view (pohled) a controller (řadič nebo ovladač). Architektura rozděluje program do samostatných částí a snaží se je co nejvíce oddělit. To zvyšuje přehlednost programu, pomáhá při testování, údržbě a případném dalším vývoji programu. Také lze snáze nahradit celé logické bloky, například vyměnit celé uživatelské rozhraní. Jednotlivé logické části mezi sebou spolupracují. Každá z těchto tří částí je zaměřena na jiné aspekty budoucího programu.

Model má za úkol práci s daty. Zajišťuje správu, seskupení a utřídění dat. Data je schopen na vyžádání předávat jiným částem programu, ukládat příchozí data, případně je na vyžádání měnit. Ověřuje přitom, jestli dané požadavky jsou oprávněné, ale neposuzuje, jestli jsou smysluplné. Například controller se chystá dělit a potřebuje od modelu dělitel. Controller zažádá o poslední prvek z pole. Model zkontroluje, že pole není prázdné, a prvek předá. Je na controlleru, aby ověřil, zda obdržené číslo je použitelné pro dělení (například není nula).

View je v kontaktu s uživatelem. Stará o správné zobrazení dat, která mu byla předána controllerem. Spravuje zobrazovací prvky, jejich rozložení a podobu. Přijímá data a požadavky od uživatele (někdy tuto funkci vykonává přímo controller) a posílá je k dalšímu zpracování.

Controller je propojovacím prvkem mezi Model a View. Zde probíhají výpočty a operace. Jako jediný „rozumí“ datům (a jejich významu), se kterými pracuje. Například uživatel zadá požadavek na výpis součtu všech čísel ve sloupci tabulky. Controller vezme data z Modelu, zpracuje je a výsledek své práce předá View, který ho prezentuje uživateli. [1]



Obr. 20: Vztahy v MVC modelu [26]

8.4 Rekurze

Rekurze znamená, že daný prvek volá (a obsahuje) sám sebe. Rekurzivní funkce se musí skládat ze dvou základních prvků. První je příkaz, kterým volá sebe sama před svým dokončením. Druhá je ukončovací podmínka. Ta stanovuje, kdy se program dostal do bodu, kdy už není nutné pokračovat, případně to není možné a další rekurzivní volání již nenastane. Funkce se ukončí a běh programu se vrátí zpět s případným výsledkem.

Rekurzivní volání se dělí na přímé a nepřímé. Přímá rekurze znamená, že se funkce volá ve svém těle sama. Nepřímá rekurze spočívá v tom, že funkce sama neobsahuje rekurzi, ale volá jinou funkci nebo posloupnost funkcí, které ale před svým dokončením rekurzivně volá funkci původní. Například funkce A může volat funkci B a funkce B zase funkci A. Každá rekurze lze změnit na nerekurzivní volání. Rekurzivní volání funkce je nahrazeno vlastním programovým zásobníkem a cyklem v programu. Zápis programu bude komplikovanější a méně přehledný, ale samotný výpočet bude efektivnější. [38]

V tomto programu, je použita rekurze při prohledávání uzlů. Prohledávací funkce začne ve výchozím uzlu, vybere volný uzel, kam bude pokračovat, a na tomto novém uzlu se zavolá rekurzivní instance funkce. Takto se rekurze zanořuje stále hlouběji dokud má algoritmus volné uzly, do kterých je možné pokračovat. Pokud již nelze pokračovat, provede se kontrola, jestli byli všechny uzly navštíveny právě jednou, a jestli je možné pokračovat nepoužitou hranou do výchozího uzlu. Pokud tomu tak není, program se vrátí o uzel zpět. Pokud to možné je, funkce vrátí výsledek, že hamiltonovský graf existuje. Funkce se postupně dokončují. Po každém dokončení funkce, se program vrátí do funkce předešlé. Takto to pokračuje dál, až se jako poslední dokončí funkce, která byla volána jako první. Maximální počet zanoření v rekurzi závisí na velikosti interního zásobníku.

8.5 Zásobník

Rekurze využívá zásobník, anglicky stack. Jedná se o způsob ukládání dat. Manipuluje s daty způsobem LIFO – last in, first out. To znamená, že prvek, který bude do zásobníku vložen, přijde jako první na řadu, když se budou ze zásobníku data brát. Lze si ho představit jako nábojový zásobník u pistole. Poslední vložený náboj bude vystřelen jako první. Žádný náboj nemůže „předbíhat“, bude vystřelen až po všech nábojích, co byly vloženy do zásobníku po něm. Vkládané prvky se číselně indexují, to pomáhá orientaci v zásobníku a další práci s prvky. Čím vyšší indexové číslo prvek má, tím byl vložen později. Typicky se k obsluze zásobníku využívají dvě hlavní funkce (nebo jejich obdoby) PUSH a POP. Funkce PUSH uloží prvek na vrchol zásobníku. Funkce POP odebere prvek z vrchu zásobníku. Dále může zásobník podporovat další rozšiřující funkce.

- Prohledávací funkce, kde se zadá hledaný prvek a funkce vrátí index prvku.
- Funkce, která vrátí prvek z vrcholu zásobníku bez toho, aby byl prvek odebrán.
- Rozhodovací funkce, která určí, jestli je zásobník prázdný.
- Funkce, která vrátí počet prvků v zásobníku.

- Rotační funkce, které přesunou všechny prvky v zásobníku o určitý počet pozic nahoru nebo dolů.

9 Program pro práci s hamiltonovský grafy ve výuce

Program vytvořený v rámci této práce by měl sloužit jako doplňková aplikace při výuce. Při výkladu hamiltonovských grafů by pomohl vizualizovat problematiku, případně by mohl sloužit k domácímu procvičení grafů. Aby aplikace dokázala splnit svůj účel a byla co možná nejužitečnější pro výuku, měla by splňovat několik níže uvedených zásad. Užití těchto zásad je vztaženo k vytvořenému programu.

9.1 Zásady edukační aplikace

názornost – Musí být zdůrazněny ty nejpodstatnější prvky programu, v tomto případě vykreslení grafu a následné vyhodnocení. Ovládací prvky nesmějí být dominantním prvkem, nesmějí překrývat vykreslené grafy, ale zároveň musejí být neustále k dispozici, aby bylo možné s grafy pohodlně pracovat.

přehlednost – Aplikace umožňuje přesouvání uzlů grafu po kreslící ploše, to pomáhá zpřehlednit někdy velmi nepřehledné grafy, u kterých dochází k častému překrývání hran. Použité barvy jsou kontrastní, aby grafy vynikly na pozadí, a výsledný hamiltonovský graf byl dobře odlišitelný od zbytku grafu.

rychlost demonstrace problému – Je důležité, aby program mohl být rychle připravený k použití (například při přednášce), a tak šlo v co nejkratším čase demonstrovat určitý problém. Nejrychlejší je ukázat některý z předpřipravených příkladů, eventuálně ručně nakreslit předem vymyšlený graf.

jednoduchost obsluhy – Uživatelské prostředí musí být vizuálně přehledné, tlačítka musí být srozumitelně popsána. Je nutné, aby bylo na první pohled jasné, jakou funkci plní. Jednotlivá tlačítka jsou k sobě seskupena a seřazena podle toho, jak na sebe logicky navazují. Podrobněji popsáno v podkapitole *Grafické rozvržení programu (layout).*

návod k obsluze – Pokud s programem přijde uživatel poprvé do styku, je důležité, aby mohl někde v programu nalézt nápovědu, která mu sdělí, co má od programu očekávat a jak daný program ovládat.

jazyková přístupnost – Aplikace podporuje dva jazyky češtinu a angličtinu. Tyto jazyky jsou vybrány z důvodu, že pravděpodobně nejčastějšími uživateli aplikace by byli čeští vysokoškolští studenti, případně zahraniční studenti z programu Erasmus. Za pomoci tlačítka lze jazyk kdykoli přepnout. Aplikaci by nebyl problém rozšířit o další jazyky (například španělštinu, němčinu, italštinu). Překládají se popisky tlačítek, přepínačů a odpovědi, zda se jedná o hamiltonovský graf, či nikoli. V okně nápovědy jsou české a anglické návody uvedeny současně.

snadná spustitelnost na zařízeních – Je žádoucí, aby byl program spustitelný na co nejvíce zařízeních, aby se potenciálně mohl dostat k co největšímu počtu studentů. Jedná se o desktopovou aplikaci. Program je napsán v programovacím jazyce Java, ten má tu výhodu, že je multiplatformní. Neměl by být problém spustit aplikaci na aktuálních desktopových operačních systémech (Windows, MacOS, Linux), na kterých je nainstalovaný aktuální JDK (Java Developer Kit). Je určitou nevýhodou, že je nutné mít JDK na zařízení nainstalované. Uživatelé ho občas mívají již nainstalovaný pro spouštění jiných programů. Jeho případná instalace není nijak náročná. Odkaz ke stažení a případný návod, jak instalovat JDK, by mohl být umístěn na webu u odkazu na stažení hamiltonovské aplikace.

Způsob využití programu se liší z pohledu toho, jestli je uživatel vyučující nebo student.

9.2 Pro studenta

Program je určen pro studenty, kteří jsou seznámeni, případně se právě z jiných zdrojů seznamují s principy a problematikou hamiltonovských grafů. Aplikace sama podrobně nevysvětluje, co jsou hamiltonovské grafy zač a jaké mají vlastnosti. Dává možnost praktické vizualizace těchto grafů. Buď prezentuje hamiltonovské grafy na některých ukázkových příkladech nebo za pomoci volného kreslení. To lze využít k překreslení příkladů ze skript nebo jiných učebních materiálů. Za pomoci přetahování uzlů se tyto grafy stanou mnohem přehlednější. Případně je možné (pokud jsou rozumného rozsahu), nechat si tyto příklady programem vyřešit. Také si lze procvičit hledání hamiltonovských grafů za pomoci náhodně generovaných obyčejných grafů.

9.3 Pro vyučujícího

Program využije vyučující, zejména při přednáškách nebo pro zadání domácího cvičení. Při přednáškách se pro rychlou ukázku určité vlastnosti grafu nejvíce hodí předpřipravené příklady. Pokud jsou příklady nedostatečné, vyučující může nakreslit vlastní graf. Aplikace nedovoluje uživatelské upravení nebo vytvoření vlastních předpřipravených příkladů. Jejich vytvoření není složité, ale je nutné zasahovat přímo do kódu programu a následně ho nechat nově zkompilovat. Je potřeba základní znalost Javy, pokud jí vyučující disponuje může předpřipravené příklady sám vytvořit. Stačí upravit již vytvořený příklad ve třídě *Example*. Vyplní se počet uzlů grafu, souřadnice jednotlivých uzlů a hrany mezi uzly zanesené do tabulky (dvojrozměrného pole). Vše je ve třídě popsáno a okomentováno.

10 Vlastní program

10.1 Obecný popis programu

Program umožňuje kreslit a generovat obyčejné grafy. Pomocí algoritmu backtrackingu je schopen určit, jestli je tento graf hamiltonovský. Případnou hamiltonovskou kružnici vykreslí. Program vykresluje grafy ve dvourozměrném prostoru. Pracuje pouze s neorientovanými grafy. To znamená, že hrany lze projít, buď jedním nebo druhým směrem. Grafy nemají hodnocené ceny hran. Respektive všechny hrany mají stejnou váhu průchodu. Jsou si tedy před prohledávacím algoritmem rovny.

V první fázi vývoje byl vytvořen pouze řešící algoritmus bez grafického rozhraní s výstupem výsledku na konzoli. Cílem bylo vytvořit prototyp, pomocí kterého by bylo možné ověřit, jestli bude Java a její základní práce s rekurzí a zásobníkem (stack) alespoň uspokojivě schopna řešit hamiltonovské grafy za pomoci backtrackingu.

Java byla zvolena pro svou univerzálnost, multiplatformnost, širokou komunitní podporu, manuály a integraci grafické knihovny Swing. Dalším důvodem je autorova předchozí zkušenost s jazykem Java a knihovnou Swing.

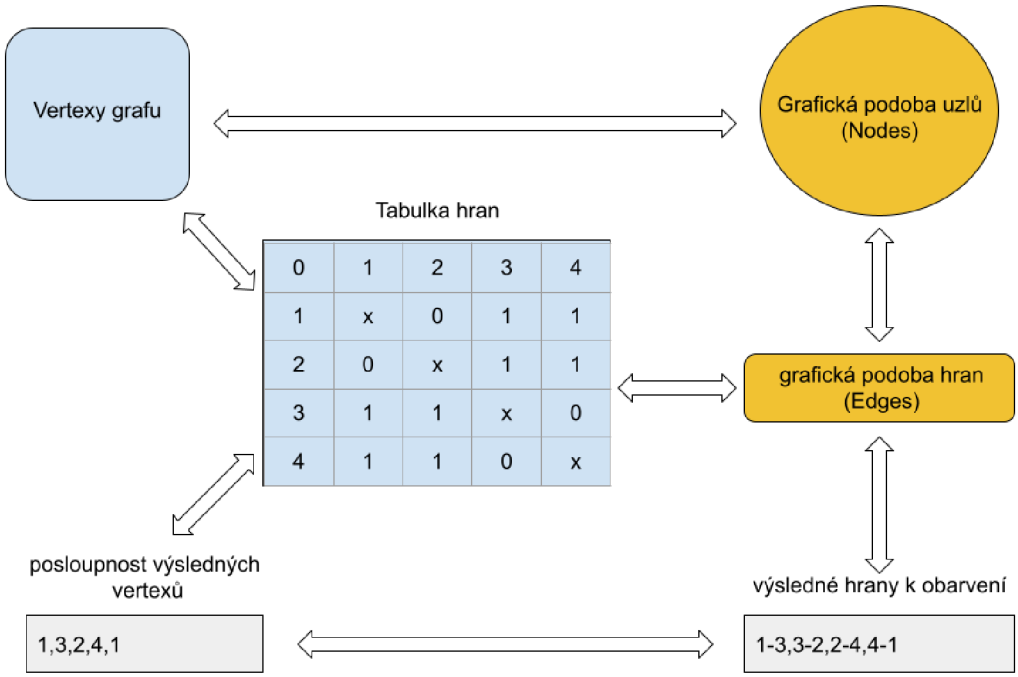
Pokud by se použití backtrackingu v Javě ukázalo jako nevhodné, například by uspokojivě řešil hamiltonovské grafy jen pro příliš nízký počet uzlů (maximálně šest), použil by se nějaký sofistikovanější algoritmus. V případě, že by se ukázalo, že z nějakého technického aspektu Javy její použití není možné, musel by být zvolen jiný jazyk. Pravděpodobně by byl vybrán C++, který dovoluje více podrobné alokování paměti, čímž dovoluje více šetřit paměť a výkon. C++ na rozdíl od Javy nemá garbage collector, ale programátor musí sám žádat o přidělení paměti a následně ji uvolnit. Tím se ušetří výkon, ale je to méně komfortní na naprogramování. Případně by bylo nutné využít kooperace několika jazyků, jeden hlavní by se staral o uživatelské rozhraní, přijímal od uživatelů požadavky, ty by připravil a odeslal druhému programu v jiném jazyce, který by velmi dobře zvládal prohledávací algoritmus, ten by úlohu vyřešil, a výsledky odeslal zpět předchozímu jazyku, který by výsledky prezentoval uživateli. S myšlenkou na úsporu paměti a výkonu byla snaha co nejvíce oddělit samotný algoritmus prohledávání hamiltonovského grafu od uživatelského rozhraní.

10.2 Vnitřní logika programu

V této podkapitole je popsáno, jakým způsobem jsou reprezentovány vrcholy a hrany grafu v kódu programu.

Nejdříve jsou vytvořeny vertexy. Ty reprezentují uzly, ale ne jejich grafickou podobu. K nim jsou vytvořeny hrany. Každá hrana je přiřazena k právě dvěma vertexům. Hrany jsou uloženy v tabulce. Souběžně s tím jsou vytvořeny grafické uzly (Nodes) a grafické hrany (Edges). Vertexy jsou propojeny s Nodes a hrany s Edges. Pouze nad tabulkou hran probíhá prohledávací algoritmus. Při průchodu tabulkou si algoritmus poznamenává, jakými vertexy prošel. Vertexy jsou uloženy v pořadí tak, jak na sebe navazují. Každý vertex je spojen s přechozím i následujícím vertexem. Tedy první vertex je spojen hranou se druhým, druhý se třetím a tak dále. Pokud hamiltonovský graf existuje, je vždy poslední vertex spojen s prvním. Na

konci průchodu je algoritmus schopen určit pouze to, zda hamiltonovský graf existuje nebo ne. Pokud ano, předá poznamenané vertexy dál. Díky jejich pořadí je možné určit výsledné hrany hamiltonovské kružnice, které se mají přebarvit. Tyto hrany jsou společně s poznamenanými uzly přebarveny nazeleno, aby uživatel viděl výsledný graf.



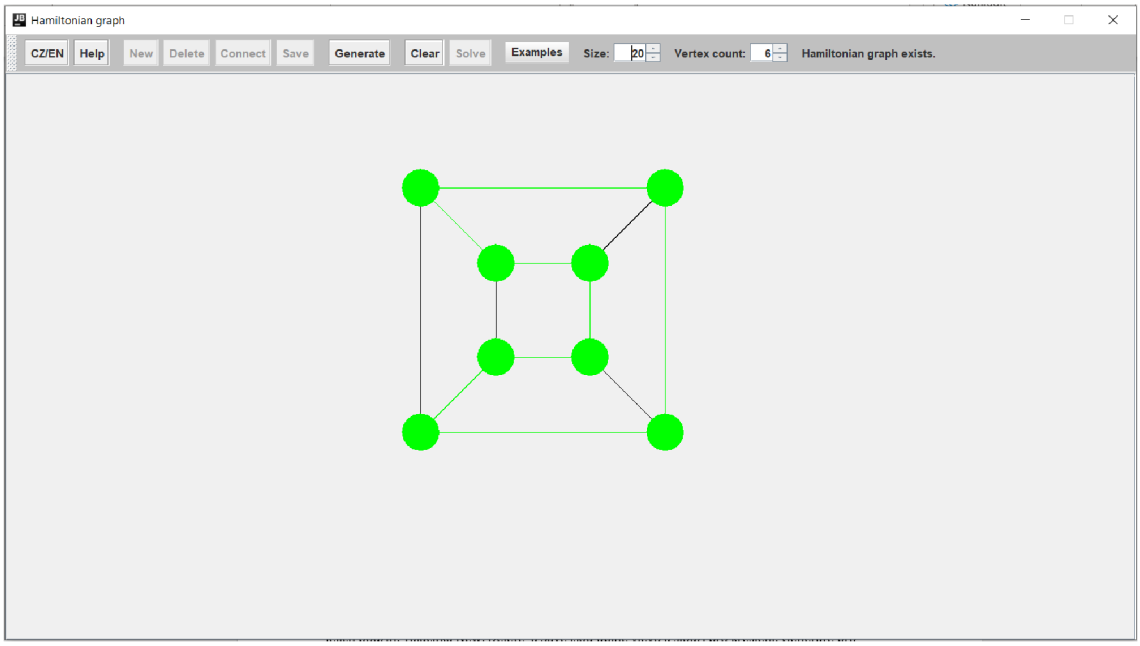
Obr. 21: Schéma hran a uzlů programu

Základ grafického rozhraní programu byl převzat z aplikace *GraphPanel*. Dostupné na: [24]. Autorem programu je *John B. Matthews,* program byl zveřejněn v roce 2008 a je distribuován pod licencí GPL. Je zaměřen na kreslení 2D geometrických tvarů a jejich barvení. Nepotřebné věci byly odstraněny, některé prvky zůstaly takřka nezměněny (tvorba Nodes a Edges, označování uzlů), a další prvky byly doplněny nebo upraveny pro potřeby stávající aplikace (listenery, kreslicí plátno).

Z pohledu funkčnosti je program rozdělen na tři hlavní části – ukázkové příklady, generátor grafů a volné kreslení grafu uživatelem. Nad každou z těchto částí lze zavolat prohledávací algoritmus k určení, zda se jedná o hamiltonovský graf.

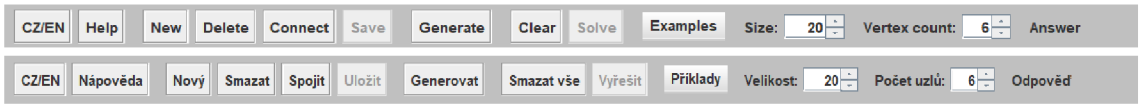
10.3 Grafické rozvržení programu (layout)

Layout je rozdělen na dvě hlavní části. První je lišta v horní části okna programu. Na liště se nacházejí ovládací tlačítka, nastavitelná pole a výsledkové pole pro odpověď na hledání grafu. Druhá část je kreslicí plátno, na kterém je možné vytvářet a generovat grafy.



Obr. 22: Okno programu

V aplikaci je možné kdykoli přepínat mezi češtinou a angličtinou. Dále v textu je na tlačítka referováno pod jejich anglickým názvem. Ale tlačítka samozřejmě mají stejné vlastnosti, i když jsou zrovna popsána česky.



Obr. 23: Anglické a české popisky ovládacích prvků

Pro větší intuitivnost ovládání jsou tlačítka seřazena zleva doprava, podle jejich využití, dále jsou seskupena do skupin podle příbuznosti jejich použití. Jako první zleva jsou tlačítka *CZ/EN* a *Help.* Na tomto přednostním místě jsou umístěna z důvodu, že pokud k aplikaci přijde zcela nový uživatel, pravděpodobně si bude chtít

přečíst nápovědu, případně změnit jazyk. Tlačítka *New, Delete, Connect* a *Save* slouží pro volné kreslení. *Generate* je samostatné, protože samo generuje náhodné grafy. *Clear* a *Solve* jsou u sebe, protože se jedná o koncová tlačítka (použijí se na již nakreslený graf). Jedno vyhodnotí graf a případně vykreslí existují hamiltonovskou kružnici, druhé vyčistí celé plátno. Examples obsahuje rozbalovací nabídku s předpřipravenými příklady.

Tlačítka se postupně deaktivují a znovu aktivují, podle smyslu dalšího ovládání. Například Save bude neaktivní, dokud je plátno prázdné (není co ukládat). Nebo po stisknutí Solve a vyhodnocení grafu zůstane tlačítko neaktivní, dokud vyhodnocený graf nebude nahrazen jiným grafem (nemá smysl prohledávat, již prohledaný graf). Neaktivita tlačítek usnadňuje obsluhu programu a zabraňuje výskytu některých chyb.

10.3.1 Popis jednotlivých tlačítek:

CZ/EN – První tlačítko programu. Přepíná texty z angličtiny do češtiny a naopak. Překládá se vše - tlačítka, popisky nastavitelných polí a textové odpovědi.

Help (Nápověda) – Základní nápověda pro obsluhu programu. Stručný popis, co je hamiltonovský graf. Popis vlastností tlačítek. Text je v češtině a angličtině.

New (Nový)- Vytvoří nový uzel. Uzel se ve výchozím stavu vytvoří zhruba uprostřed kreslícího pole. Po kliknutí tlačítkem myši do kreslícího pole, se příští uzel vytvoří na pozici, kam bylo naposledy kliknuto. Je ošetřeno, aby se nově vytvořené uzly nepřekrývaly.

Delete (Smazat) – Smaže označené uzly. Společně s nimi smaže také hrany, které byly napojeny na označené mazané uzly.

Save (Uložit) – Uloží nakreslené uzly a hrany. Z grafických prvků vytvoří vertexy a tabulku hran připravenou pro prohledávací algoritmus. Bez tohoto uložení není možné nakreslený graf nechat prohledat.

Connect (Spojit) – Propojí všechny označené uzly hranami.

Generate (Generovat) – Generování náhodných grafů. V poli *Vertex count* uživatel, nastaví, kolik uzlů má generovaný graf obsahovat.

Solve (Vyřešit) – Je společné tlačítko pro všechny tři segmenty. Pro volné kreslení, pro automatické generování i pro předpřipravené příklady. Po jeho stisknutí se spustí prohledávací algoritmus, jehož výsledek se následně vypíše do pole pro odpověď. V případě, že hamiltonovský graf existuje, obarví se uzly a výsledné hrany zeleně tak, jak byl nalezen výsledný graf.

Clear (Smazat vše) – Smaže vše nakreslené na kreslicím plátně. Odstraní všechny uložené uzly a hrany. Znovu inicializuje všechny *array listy* na pozadí, vyprázdní zásobník. Smaže stávající odpověď o ne/existenci hamiltonovského grafu.

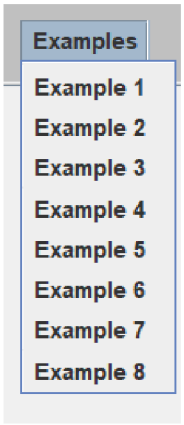
Size (Velikost) – Nastaví vizuální velikost uzlu. Tu lze nastavit v rozsahu 5 až 100. Výchozí hodnota je 20. Každý uzel může mít jinou velikost. Větší uzly najdou využití, pokud by se grafy promítaly na plátno, které by bylo ve velké vzdálenosti od studentů.

Vertex count (Počet uzlů) – Nastaví počet uzlů, které se mají vygenerovat. Minimální počet uzlů je 3. Nižší počet nemůže tvořit hamiltonovskou kružnici a je tedy zbytečné ho generovat a následně prohledávat. Maximální počet je 10. Prohledávací algoritmus by pro více uzlů nemusel vždy fungovat správně.

Pole pro odpověď – Po stisknutí tlačítka Solve, zobrazí anglickou/českou větu, sdělující jestli hamiltonovský graf existuje, či nikoli. Po stisknutí tlačítka Clear a smazání grafu se smaže rovněž *odpověď*. Ta bude nahrazena výchozím slovem Answer/Odpověď.

Hamiltonian graph exists.	Hamiltonovský graf existuje.
Too few vertexes. Hamiltonian graph does not exist.	
Příliž málo uzlů. Hamiltonovský graf nemůže existovat.	
Hamiltonian graph does not exist.	Answer
Hamiltonovský graf neexistuje.	Odpověď

Obr. 24: Všechny texty, které se v poli pro odpověď mohou objevit

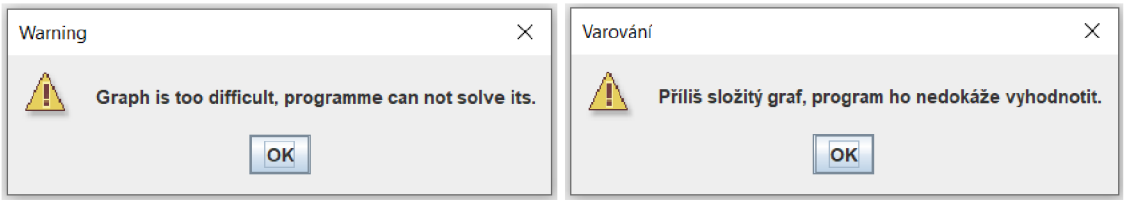


Obr. 25: Rozbalovací tlačítko s připravenými grafy

10.4 Hlavní části programu

10.4.1 Volné kreslení

Uživatel má možnost volně kreslit grafy a nechat si je prohledávat. Doporučená maximální velikost kresleného grafu je deset uzlů. Prohledávací algoritmus by měl tyto grafy bez problému řešit. Pro grafy o více uzlech, už algoritmus nemusí fungovat správně. Čím vyšší počet uzlů, tím pravděpodobněji algoritmus selže. Pokud algoritmus nedokáže graf prohledat, objeví se okno s varováním. To uživateli sdělí, že graf je příliš složitý a program ho neumí vyřešit. Hláška je buď anglicky nebo česky podle zrovna zvoleného jazyka. Po zavření okna a vymazání grafu lze ovšem pokračovat dál v práci. Není nutné program restartovat. Pokud by uživatel potřeboval pracovat s rozsáhlejšími grafy, lze program alespoň využít jako vizualizační pomůcku. Uzly grafu lze přesouvat, tím graf zpřehlednit a pomoci uživateli odhalit hamiltonovské kružnice od pohledu.



Obr. 26: Upozornění, že algoritmus nedokáže graf vyřešit. Anglická a česká verze.

Nový uzel se vykreslí v tom místě, kam uživatel naposledy klikl levým tlačítkem myši. Pokud je několikrát za sebou vytvořen nový uzel a pozice pro vykreslení mezitím není změněna, nový uzel se vždy o kus odsadí od předchozího. To zabrání překrývání nových uzlů. Tažením myši lze označit více uzlů. Po stisknutí pravého tlačítka myši se objeví vertikální lišta s tlačítky New, Delete, Connect. Využití této lišty usnadní a urychlí práci při kreslení rozsáhlejších grafů.

10.4.2 Generování

Generátor náhodně generuje obecné grafy. Ty je dále možné nechat algoritmus posoudit, jestli jsou hamiltonovské. Uživatel si může zvolit počet uzlů, které má vygenerovaný graf obsahovat. Výchozí hodnota je šest. Minimum jsou tři a maximum je deset. Méně uzlů nemá cenu generovat, protože nemohou tvořit hamiltonovský graf. Použití více uzlů už by algoritmus, z důvodu paměťové náročnosti, nemusel korektně zpracovat. Pokud mezi nastavením počtu uzlů a generováním proběhlo volné kreslení, případně kreslení předpřipravených příkladů, vygeneruje se požadovaný, dříve nastavený počet uzlů pro generování, nezávisle na úlohách, které mezitím proběhly.

10.4.2.1 Postup generování

Nejdříve se vygenerují vertexy. Pro každý z nich se vygeneruje náhodná pozice ve virtuální mřížce. Každý vertex má odlišné poziční souřadnice. Tato pozice slouží pro jejich pozdější identifikaci a nemá návaznost na polohu vykreslovaných uzlů, které pro tyto vertexy budou později vytvořeny. Dále se generují hrany propojující vertexy. Je potřeba stanovit, kolik hran pro n uzlů může být vygenerováno. To nastane, když bude graf úplný. Znamená to, že každý uzel bude propojený se všemi ostatními uzly. Počet hran úplného grafu se spočítá podle rovnice:

$$\max = \frac{n * (n-1)}{2}$$

Kde n je počet uzlů a *max* je maximální počet hran grafu. Tímto se stanoví maximální počet hran. Hrany se generují podle jednoho ze tří různých módů. Módy se od sebe liší rozsahem počtu hran, které dovolí vygenerovat. První má rozsah *0 až max/2*, druhý má rozsah *max/2 až 2/3max* a třetí *2/3max až max*. Každý z určitých módů má danou procentuální šanci, se kterou může nastat. Původně bylo procentuální rozdělení šancí pro každý počet uzlů stejné. Během testování se však ukázalo, že čím více má graf uzlů, tím častěji se v *módu 3* s rostoucím počtem hran hamiltonovský graf objeví. Bylo to tak časté, že to škodilo rozmanitosti generovaných grafů. Proto

jsou vytvořeny skupiny podle počtu grafů s rozdílnými procentuálními šancemi. V tabulce je uvedeny procentuální šance módů pro jednotlivé uzly.

	rozsah módu	počet vertexů grafu			
		3	4;5	6	7;8;9;10
mód 1	0 - 1/2max	14 %	18 %	22 %	25 %
mód 2	1/2max - 2/3 max	21 %	27 %	33 %	38 %
mód 3	2/3 max - max	64 %	55 %	44 %	38 %

Tab. 16: Pravděpodobnost vygenerovaní určitého počtu hran v závislosti na počtu vertexů

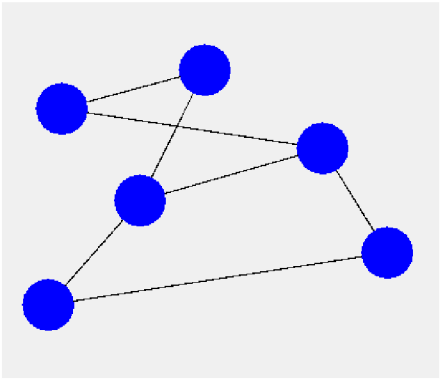
Počet uzlů je v rozsahu 3 až 10. Jiný počet program neumožní vygenerovat. Módům 1 a 2 s navyšujícím se počtem uzlů procentuální šance roste. Zatímco módu 3 procenta klesají. Z toho vyplývá, že čím méně uzlů vygenerovaný graf má, tím potřebuje podpořit více hranami. To pomůže docílit toho, aby každý generovaný graf měl zhruba 50% šanci, že bude hamiltonovský.

10.4.3 Připravené příklady

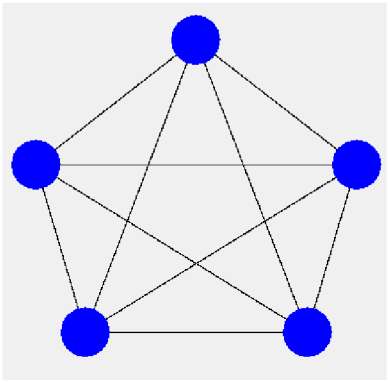
Příklady jsou navržené tak, aby ukázaly některé vlastnosti grafů. Například pokud je graf úplný, je vždy hamiltonovský, nebo pokud je bipartitní s lichým počtem vrcholů, nemůže být nikdy hamiltonovský. Také poukazuje na vizuální podobnost mezi některými, vlastnostmi zcela odlišnými, grafy. Níže jsou popsány jednotlivé grafy, které jsou v aplikaci připraveny.

Příklad 1 – Základní příklad na seznámení. Hamiltonovský graf existuje. Hamiltonovská kružnice je zde dobře vidět. Jedna hrana je v grafu navíc (po vyřešení zůstane neobarvená). To pomůže uživateli si uvědomit, že pokud hamiltonovská kružnice existuje nemusí procházet všemi hranami grafu.

Příklad 2 – Jedná se o úplný graf K_5 . Úplné grafy jsou vždy hamiltonovské. Hamiltonovský graf existuje.



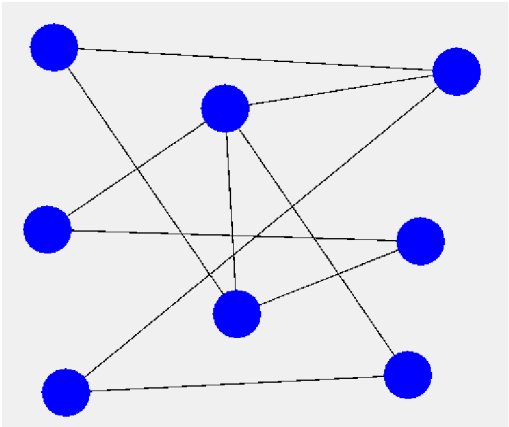
Obr. 27: Příklad 1



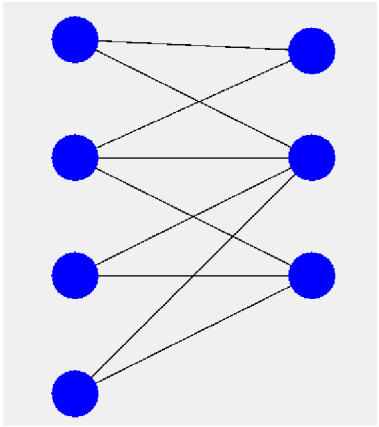
Obr. 28: Příklad 2

Příklad 3 – Graf je poměrně nepřehledný, dochází k mnoha křížení hran. Na první pohled, kvůli rozložení uzlů, může vypadat jako bipartitní. Bipartita je zde však několikrát porušena, například u prostředních uzlů. Hamiltonovský graf existuje.

Příklad 4 – Zde se již jedná o bipartitní graf s lichým počtem vrcholů. Hamiltonovská kružnice v těchto grafech nemůže existovat. Důkaz: [31]



Obr. 29: Příklad 3

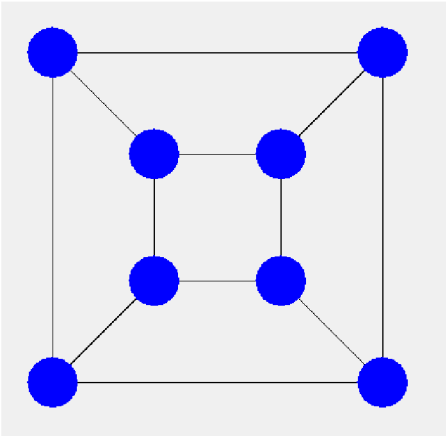


Obr. 30: Příklad 4

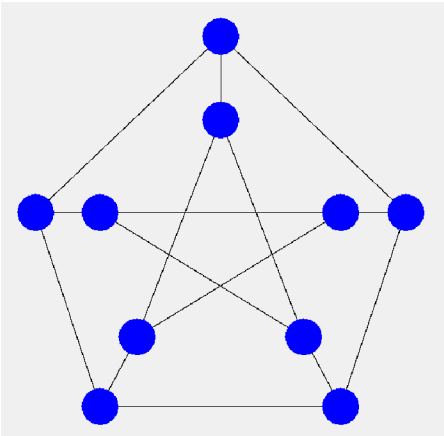
Příklad 5 – Graf vnořených čtverců. Jde o rovinný síťový model krychle. Jedná se o typ grafu, kdy je trojrozměrné těleso převedeno do dvojrozměrné podoby, stejně jako u Hamiltonova dvanáctistěnu (kapitola *William Hamilton*). Hamiltonovský graf zde existuje.

Příklad 6 – Petersonův graf. Jedná se o 3-regulární graf (každý uzel má stupeň 3), který neobsahuje most. Také je to nejmenší hypohamiltonovský graf. Takový graf

není hamiltonovský, ale po odebrání jakéhokoli uzlu je výsledný graf hamiltonovský. [41] [42]



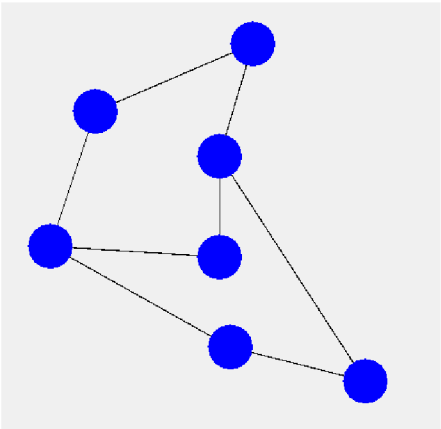
Obr. 31: Příklad 5



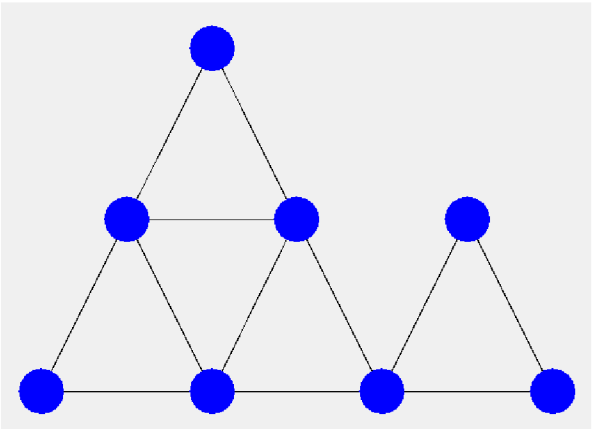
Obr. 32: Příklad 6

Příklad 7 – Graf obsahuje tři zcela různé kružnice, přesto však dohromady netvoří žádnou hamiltonovskou kružnici. Hamiltonovský graf neexistuje.

Příklad 8 – V tomto grafu je jasně viditelná artikulace. Pokud graf obsahuje most nebo artikulaci nemůže být hamiltonovský.



Obr. 33: Příklad 7



Obr. 34: Příklad 8

10.5 Testování

Před začátkem vývoje byly stanoveny podmínky, které musí program splňovat. Během vývoje programu byly průběžně doplňovány. Výsledná aplikace byla testována, jestli tyto podmínky splňuje. Pro tyto podmínky nejsou vytvořeny automatické testy. Stanovené podmínky často závisejí na vizuální kontrole, proto by

automatické testy byly obtížně kontrolovatelné, nebo nedostatečné. Například snadno by se hledala *Exception* chyba ve výpisu na konzoli, ale to by nekontrolovalo vizuální výstup. Pokud by *Exception* chyby nenastávaly, neznamenalo by to, že vykreslené grafy jsou v pořádku. Podmínky nejsou tolik rozsáhlé, aby byly nutné alespoň částečně automatizované testy. Jsou velmi různorodé a muselo by se napsat mnoho druhů testů. Chyby byly často natolik závažné, že zabránily v pokračování běhu programu. Při ladění chyb nejvíce posloužily ukázkové příklady. Zejména na nich proběhlo ladění prohledávacího algoritmu. Protože dokážou připravit vždy stejný vstupní graf, usnadňovalo to zopakování vyskytujících se chyb a jejich následné odstranění.

Jedná se o desktopovou aplikaci. Aplikace není určenas pro mobilní zařízení (pokud na nich neběží adekvátní desktopová verze operačního systému). Kód je napsaný v Javě. Díky její multiplatformnosti by aplikace měla běžet na každém operačním systému s aktuálním JDK (Java Developer Kit). Aplikace byla testována na Windows 10 a Linux Ubuntu 20 (příloha 1 a 2). Testy jsou rozděleny do několika kategorií.

10.5.1 Testy pro volné kreslení

- grafy s jedním nebo dvěma uzly nikdy nemohou být vyhodnoceny jako hamiltonovské
- ke každému vykreslenému prvku musí být vytvořeny odpovídající prvky na pozadí programu, například každý vertex je přiřazen k node, každá hrana k edge
- pokud bude nějaký uzel/hrana vizuálně smazána, musí být smazána i ze všeho kde byla uložena
- hrana musí propojovat pouze dva různé uzly, samostatná hrana nebo hrana mezi jiným počtem uzlů než dva nemůže existovat
- mezi dvěma uzly nemůže existovat více než jedna hrana
- pokud bude nakreslen a prohledáván příliš složitý graf, který prohledávací algoritmus ho nedokáže vyhodnotit, musí o tom být uživatel informován

10.5.2 Testy pro generování

- generované grafy by měly mít rovnoměrně rozdělenou řešitelnost. Nesmí se stávat, že hamiltonovský graf bude jeden z deseti vygenerovaných grafů. Je žádoucí, aby hamiltonovské a nehamiltonovské grafy byly generovány zhruba ve stejné míře.
- musí se vždy vygenerovat požadovaný počet uzlů uvedený v poli Vertex count
- při přechodu z volného kreslení nebo příkladů na generování, pokud není nastaven jiný počet uzlů, musí se vygenerovat stejný počet uzlů jako při posledním generování, ne počet, který byl použit obecně (například při kreslení). Zabrání to moc vysokému počtu uzlů grafu, který pokud je správně nakreslený, je v pořádku, ale při náhodném generování by mohl selhat. Kdyby se tak stalo, objeví se *varování o neřešitelnosti*. Je však žádoucí selhání předejít.

10.5.3 Testy předpřipravených příkladů

- všechny nabízené příklady musí vygenerovat nějaký graf
- grafy od sebe musejí být různé, nemá smysl pod více položkami mít stejné grafy
- grafy mají být různorodé, mají reprezentovat určité důležité vlastnosti grafů
- připravený graf musí být vždy tak složitý, aby prohledávací algoritmus byl vždy schopný určit, jestli se jedná o hamiltonovský graf
- všechny příkladové grafy musejí být v programu zadány správně a konzistentně (obsahovat počet uzlů, pozice uzlů a symetrickou matici hran)

10.5.4 Celkové testy

- výsledek prohledávání musí být vždy sdělen uživateli a výsledek musí být vždy správný

- tlačítka musí vždy plnit svoji funkci nehledě na předchozí konfiguraci. Pokud to z důvodu vnitřní logiky není možné, případně to nedává smysl, tlačítka budou neaktivní do doby, dokud znovu nenabydou svojí relevance.
- výsledek prohledávání se musí shodovat s grafem (nesmí se stát, že by se vykreslila hamiltonovská kružnice, ale výsledek by hlásil její neexistenci)
- pokud je vyhodnoceno, že hamiltonovský graf existuje, musí být barevně vyznačena výsledná hamiltonovská kružnice, která byla nalezena
- při přepínání jazyků se změnit všechny texty, co se mají změnit, nesmějí zůstávat pozůstatky předešlého jazyka
- pole pro změnu velikosti a počtu vertexů musí dovolovat pouze takové vstupy od uživatele, které je program schopný obsloužit

11 Závěr a doporučení

Vytvořený program je funkční a splňuje očekávané požadavky. Jeho největší slabinou je jeho prohledávací algoritmus. Využití backtrackingu pro hledání hamiltonovských grafů je možné, avšak velmi omezené. Vzhledem k tomu, že hamiltonovské grafy patří do třídy složitosti NP-úplných problémů a samotný algoritmus backtrackingu je vcelku neefektivní, je program limitován počtem uzlů grafu, který bude prohledávat. Bezpečně zvládá grafy o 10 uzlech. Lze kreslit a vyhodnocovat i složitější grafy, ale čím větší počet uzlů, v tím více případech prohledávání selže. Dojde k přetečení zásobníku. To znamená, že zásobník vyčerpá veškerou paměť, která mu byla přidělena. Díky ošetření nedojde k nějaké fatální chybě a s programem lze dále pracovat, ale program daný graf nedokáže vyhodnotit.

V případě potřeby navýšení počtu prohledávaných uzlů je jednoznačným doporučením využití efektivnějšího algoritmu. Samozřejmě by šel zefektivnit i současný kód, například nahradit rekurzi cyklem a vlastním zásobníkem. Případně přidělit vnitřnímu zásobníku více paměti (což na různých počítačích funguje různě, občas to zabrání samotnému spuštění programu). Ale v porovnání s nasazením jiného algoritmu, bude zvýšení efektivity zanedbatelné. Program má však sloužit jako učební pomůcka. Vzhledem k zaměření programu je nutné, aby byl dostupný pro co největší počet lidí. Program svoje edukační požadavky splňuje. Je přehledný, není složitý na obsluhu, je připraven k rychlému použití, podporuje dva jazyky a díky tomu, že je napsaný v Javě, je multiplatformní.

Možnosti dalšího rozšíření lze rozdělit do několika částí. Zaprvé je to již výše zmíněná implementace efektivnějšího algoritmu. Další oblastí je zlepšení interakce s uživatelem. Program by v případě nenalezení grafu mohl vypsat nějaké základní příčiny, proč graf není hamiltonovský. Například, že graf není spojitý, že na vrchol je napojena pouze jedna hrana , že porušuje nutnou podmínku minimálního počtu hran. V případě nalezení hamiltonovské kružnice by program mohl zobrazovat alternativní průchody, nyní vždy vykreslí stejnou kružnici. Také by mohl umět vyhodnocovat hamiltonovské cesty (graf, který lze projít všemi body bez nutnosti návratu do výchozího bodu). Ovšem pravděpodobně nejpřínosnější pro rozvinutí

edukačního aspektu programu by byla implementace dalších prohledávacích algoritmů pro jiné druhy grafů. Například pro *eulerovský graf* (graf, ve kterém je nutné projít všechny hrany jedním tahem), *stromové grafy* (spojité grafy, které neobsahují žádnou kružnici) nebo přidat *hodnocené grafy* a implementovat například Dijkstrův algoritmus. Všechna tato navrhovaná rozšíření a případné další úpravy programu, by ovšem měly vycházet z požadavků, které vzniknou při praktickém používání programu.

12 Seznam zdrojů

[1] Bernard, Borek. Úvod do architektury MVC. In: Zdroják.cz [online]. 7.5.2009 [cit. 2021-9-12] Dostupné z: https://zdrojak.cz/clanky/uvod-do-architektury-mvc/

[2] Clay Mathematics Institute [online]. Peterborough, 18.8.2021 [Citace: 20.8.2021]. Dostupné z: https://www.claymath.org/millennium-problems

[3] Cook–Levin theorem or Cook's theorem. In: Geeks for geeks [online] 18. 6. 2021 [cit. 2021-10-30] Dostupné z: https://www.geeksforgeeks.org/cook-levin-theorem-or-cooks-theorem/

[4] DALGETY, James. The Icosian Game. In: Puzzlemuseum.com [online]. 2017 [cit. 2021-9-17] Dostupné z: https://www.puzzlemuseum.com/month/picm02/200207icosian.htm

[5] DARLING, David. Icosian game. In: Daviddarling.info [online] [cit. 2021-10-15] Dostupné z: https://www.daviddarling.info/encyclopedia/I/Icosian_Game.html

[6] Dictionary. Wordsense.eu [online]. [cit. 2021-9-6] Dostupné z: https://www.wordsense.eu/icosa-/

[7] DOLAN, Brian. Plaque on Broome Bridge, Dublin. In: Researchgate.net [online] [cit. 2021-9-20] Dostupné z: https://www.researchgate.net/figure/Plaque-on-Broome-Bridge-Dublin-commemorating-Hamiltons-discovery-of-quaternions_fig4_305078486

[8] DVOŘÁK, Tomáš. Souvislost v grafech [online prezentace] Praha: KSVI UK 1. 3. 2005 [cit. 2021-10-1] Dostupné z: http://ksvi.mff.cuni.cz/~dvorak/vyuka/UIN009/Souvislost_tisk.pdf

[9] ELIASEN, Alan. Historical Currency Conversions. [online] [cit. 2021-9-29] Dostupné z: https://futureboy.us/fsp/dollar.fsp?quantity=25&currency=pounds&fromYear=1857

[10] ESFAHBOD, Behnam. Euler diagram for P, NP, NP-Complete, and NP-Hard set of problems. In: Wikipedia.org [online] 1. listopadu 2007 [cit. 2021-10-14] Dostupné z : https://cs.wikipedia.org/wiki/Soubor:P_np_np-complete_np-hard.svg

[11] ERLEBACH, Pavel. Vybrané třídy složitosti. Brno: VUT FIT [online]. [cit. 2021-10-10] Dostupné z: http://www.fit.vutbr.cz/~meduna/mti/2004_05/erlebach.pdf

[12] Flinders Hamiltonian Cycle Project. Sites at Flinders University [online]. Copyright © 2021. [cit. 01.11.2021]. Dostupné z: https://sites.flinders.edu.au/flinders-hamiltonian-cycle-project/

[13] FUCHS, Eduard. Teorie grafů.[online] Brno: Masarykova univerzita, 2003. [Citace: 18. 8. 2021] Dostupné z: https://is.muni.cz/el/sci/podzim2005/M5140/um/graf.pdf

[14] GÜNAŞTI, Gökmen. Quaternions Algebra, Their Applications in Rotations and Beyond Quaternions. Digitala Vetenskapliga Arkivet [online]. 3.9. 2016 [cit. 2021-10-2] Dostupné z: https://www.diva-portal.org/smash/get/diva2:535712/FULLTEXT02.pdf

[15] A Hamiltonian cycle on a dodecahedron. In: Thatsmaths.com [online] 19. 12. 2012 [cit. 2021-9-4] Dostupné z: https://thatsmaths.com/2012/12/20/santas-tsp-algorithm/icosian-game-2d3d

[16] HARTMAN, James. What is Java? Definition, Meaning & Features of Java Platforms. In: Guru99.com [online]. 7. října 2021 [cit. 2021-10-17] Dostupné z: https://www.guru99.com/java-platform.html

[17] Hierarchy of Java Swing classes. In: Javatpoint.com [online] [2021-9-18] Dostupné z: https://www.javatpoint.com/java-swing

[18] HLINĚNÝ, Petr. Základy teorie grafů pro (nejen) informatiky. [online] Brno: Masarykova univerzita. 2010 [cit. 2021-10-20] Dostupné z: https://is.muni.cz/elportal/?id=878389

[19] HORSTMANN, Cay S. *Core Java: Volume I–Fundamentals*. Eleventh Edition. Pearson Education, 2019. ISBN 978-0-13-516630-7.

[20] Java Primitives versus Objects. Baeldung.com [online]. 20. 9. 2019 [cit. 2021-10-6] Dostupné z: https://www.baeldung.com/java-primitives-vs-objects

[21] JELLISS, G.P. Early History of Knight's Tours. In: Mayhematics.com [online] [cit. 2021-9-18] Dostupné z: https://www.mayhematics.com/t/1a.htm

[22] KOVÁŘ, Petr. Úvod do Teorie grafů. [online] 2016. [Citace: 20. 8. 2021] Dostupné z: https://homel.vsb.cz/~kov16/files/uvod_do_teorie_grafu.pdf

[23] MAREŠ, Martin. Převody problémů a NP-úplnost. In: The Home Page of Martin Mareš [online]. 16. 2. 2012 [cit. 2021-9-17] Dostupné z: https://mj.ucw.cz/vyuka/1112/ads2/8-prevody.pdf

[24] MATTHEWS, John B. Graph Panel. [online] 2008 [cit. 2021-10-2] Dostupné z: https://sites.google.com/site/drjohnbmatthews/graphpanel

[25] MURTHY, G. S. S. The Knight's Tour Problem and Rudrata's Verse. In: Indian Academy of Sciences [online] srpen 2020 [cit. 2021-10-5] Dostupné z: https://www.ias.ac.in/article/fulltext/reso/025/08/1095-1116

[26] MVC diagram. In: Techterms.com [online] [cit. 2021-9-4] Dostupné z: https://techterms.com/definition/mvc

[27] NETRVALOVÁ, Arnoštka. NP-úplné problémy [online] [Citace: 19. 8. 2021] Dostupné z: http://www.kiv.zcu.cz/~netrvalo/vyuka/ppa2-14/ekniha/online/HTML/56/text.htm

[28] O'CONNOR, J. J. ROBERTSON, E. F. William Rowan Hamilton. In: MacTutor History of Mathematics Archive [online] červen 1998 [cit. 2021-9-28] Dostupné z: https://mathshistory.st-andrews.ac.uk/Biographies/Hamilton/

[29] An overview of the software development process. In: Oracle.com [online] [cit. 2021-10-13] Dostupné z: https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

[30] Painting in AWT and Swing. In: Oracle.com [online] 4. 3. 2019 [cit. 2021-10-12] Dostupné z: https://www.oracle.com/java/technologies/painting.html

[31] Prove that a bipartite graph with an odd number of vertices is not hamiltonian. In: slaystudy.com [online] [cit. 2021-10-1] Dostupné z: https://slaystudy.com/prove-that-a-bipartite-graph-with-an-odd-number-of-vertices-is-not-hamiltonian/

[32] RICHTA, Karel a kolektiv. NP-úplnost a další.[online prezentace] 2018. [Citace: 13.8. 2021] Dostupné z: https://cw.fel.cvut.cz/b182/_media/courses/b6b36dsa/dsa-13-np-uplnost.pdf

[33] ROBINSON, Matthew a VOROBIEV, Pavel. Swing. Second edition. Greenwich: Manning Publications Co., 2003. Chapter 1. ISBN 1930110-88-X.

[34] RYBIČKA, Jiří. Časová a prostorová složitost algoritmů[online prezentace] Brno: PEF MENDELU[Citace: 26.8.2021] Dostupné z: https://akela.mendelu.cz/~rybicka/prez/pt/p9_slozitost.pdf

[35] SAWA, Zdeněk. NP-úplné problémy. Ostrava: VŠB TUO [online prezentace]. 13. prosince 2020 [cit. 2021-10-16] Dostupné z: http://www.cs.vsb.cz/sawa/ti/slides-video/19-np-completeness.pdf

[36] STRMECKI, Daniel. Is Java a Compiled or Interpreted Language?. In: Baeldung.com [online]. 25. srpna 2021 [cit. 2021-10-21] Dostupné z: https://www.baeldung.com/java-compiled-interpreted

[37] ŠIŠMA, Pavel. Vznik a vývoj teorie grafů. (Czech) [The rise and development of graph theory]. Pokroky matematiky, fyziky a astronomie, vol. 43 (1998), issue 2, pp. 89-99 Dostupné z: https://dml.cz/handle/10338.dmlcz/137535

[38] TÖPFER, Pavel. Rekurze. Praha: KSVI MFF UK [online]. [cit. 2021-9-6] Dostupné z: https://ksvi.mff.cuni.cz/~topfer/Texty/TextRekurze.pdf

[39] VITANYI, Paul. Turing machine. *Scholarpedia* [online]. 2009. doi:10.4249/scholarpedia.6240 [cit. 2021-8-26] Dostupné z: http://www.scholarpedia.org/article/Turing_machine

[40] WEISSTEIN, Eric W. Hamiltonian Cycle. MathWorld--A Wolfram. [online] [Citace: 13. 8. 2021.] Dostupné z: https://mathworld.wolfram.com/HamiltonianCycle.html

[41] WEISSTEIN, Eric W. Hypohamiltonian Graph. In: MathWorld--A Wolfram Web Resource [online]. [cit. 2021-10-3] Dostupné z: https://mathworld.wolfram.com/HypohamiltonianGraph.html

[42] WEISSTEIN, Eric W. Petersen Graph. In: MathWorld--A Wolfram Web Resource [online]. [cit. 2021-10-3] Dostupné z: https://mathworld.wolfram.com/PetersenGraph.html

[43] WILKINS, R. David. William Rowan Hamilton: mathematical genius. In: Physics world [online] 3. srpna 2005 [cit. 2021-9-24] Dostupné z: https://physicsworld.com/a/william-rowan-hamilton-mathematical-genius/

[44] William Rowan Hamilton portrait oval 2. In: Wikimedia.org [online] [cit. 2021-10-2] Dostupné z: https://commons.wikimedia.org/wiki/File:William_Rowan_Hamilton_portrait_oval_2.png

[45] WILSON, R. J. A Brief History of Hamiltonian Graphs. *Annals of Discrete Mathematics 41*. North-Holland: Elsevier Science Publishers B.V., 1989, s. 487–496. DOI: 10.1016/S0167-5060(08)70484-9.

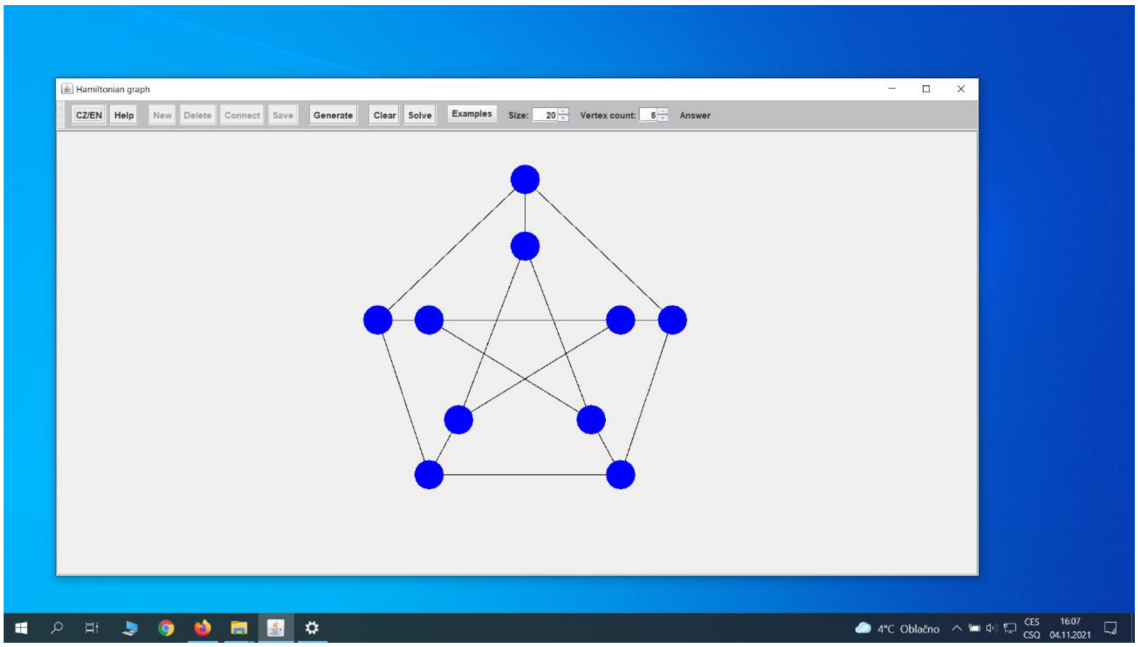
[46] ZHU, Eric. VERTEX-COVER → CLIQUE. [online]. 2015 [cit. 2021-9-12] Dostupné z: https://www.clear.rice.edu/comp487/VC_Clique.pdf

13 Seznam obrázků

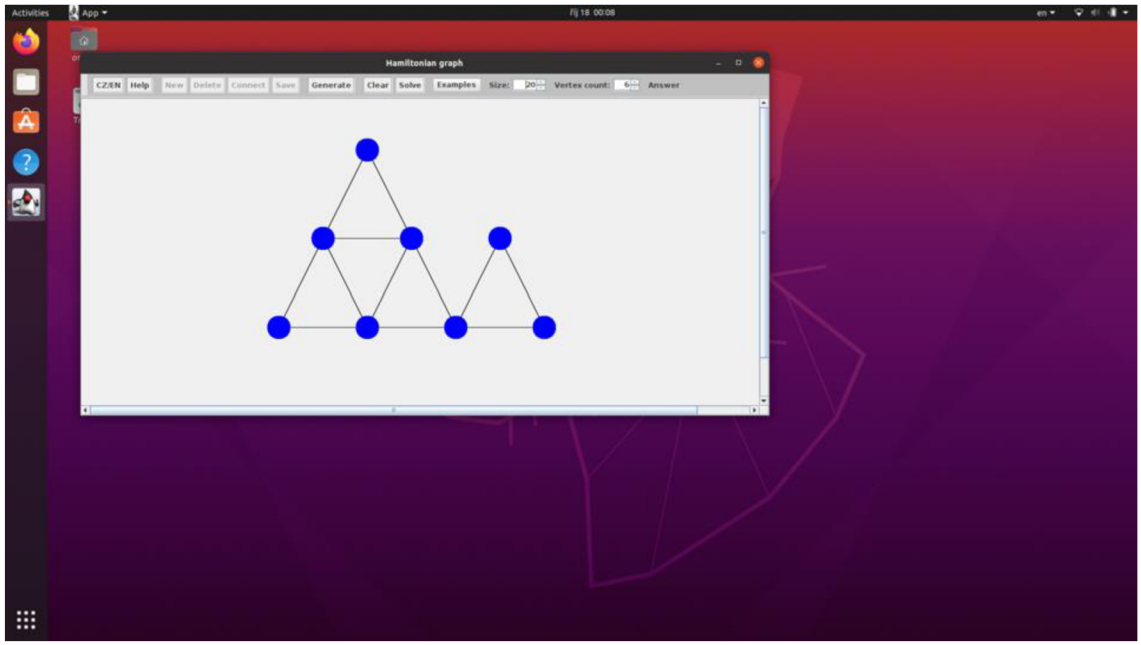
14 Seznam tabulek

15 Přílohy

1) Vytvořený program spuštěný na Windows 10



2) Vytvořený program spuštěný na Linux Ubuntu 20



3) zdrojový kód vytvořeného programu

```java
import hamiltonianGraph.GraphPanel;

import javax.swing.*;
import java.awt.*;

public class App {

        public static void main(String[] args) {
          EventQueue.invokeLater(() -> {
                  JFrame f = new JFrame( title: "Hamiltonian graph");
                  f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                  GraphPanel gp = new GraphPanel();
                  f.add(gp.getControlPanel(), BorderLayout.NORTH);
                  f.add(new JScrollPane(gp), BorderLayout.CENTER);
                  f.pack();
                  f.setLocationRelativeTo(null);
                  f.setVisible(true);
                  f.setResizable(true);
          });
}
}
```

```java
package hamiltonianGraph;

import java.awt.*;
import java.util.List;
import java.util.Random;


//generovani nahodnych  cisel
public class Generator {

    private static final Random r = new Random();

    public Generator(){

    }
    //vrátí náhodné celé číslo ze zadaného rozsahu
    public int getRandomIntegerBetweenRange(int min, int max) { return (int)(Math.random()*((max-min)+1))+min; }

    //zabrání překrývání uzlů při generování a náhodně je rozmístí po mapě
    public Point noCoverInGenerate(int WIDE, int HIGH, List<Node> nodes ) {

        int a, b, d, e;
        boolean generateAgain;
        do  {
            generateAgain = false;

            //ty čísla 40 zabrání vykreslení mimo okno
            d = r.nextInt( bound: WIDE - 80 ) + 40;
            e = r.nextInt( bound: HIGH - 80 ) + 40;

            for (Node node : nodes) {
                a = node.getLocation().x;
                b = node.getLocation().y;

                //aby se uzly neprekryvaly
                if (Math.abs(d - a) < 51 && Math.abs(e - b) < 51) {
                    generateAgain = true;
                    break;
                }
            }
        } while (generateAgain);

        return new Point(d,e);
    }

}
```

```java
package hamiltonianGraph;

//interní reprezenace uzlu, slouží pro prohledávání
public class Vertex {

    private int x;
    private int y;
    private boolean visited = false;

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

    public int getX() { return x; }

    public int getY() { return y; }

    public boolean isVisited() { return visited; }

    public void setVisited(boolean visited) { this.visited = visited; }
}
```

```java
package hamiltonianGraph;
import java.awt.*;

public class Edge {

    public Node getN1() {
        return n1;
    }

    public Node getN2() {
        return n2;
    }

    private Node n1;
    private Node n2;

    public Edge(Node n1, Node n2) {
        this.n1 = n1;
        this.n2 = n2;
    }

    public void draw(Graphics g, Color color) {
        Point p1 = n1.getLocation();
        Point p2 = n2.getLocation();
        g.setColor(color);
        g.drawLine(p1.x, p1.y, p2.x, p2.y);
    }

    public void draw(Graphics g) {
        Point p1 = n1.getLocation();
        Point p2 = n2.getLocation();
        g.setColor(Color.black);
        g.drawLine(p1.x, p1.y, p2.x, p2.y);
    }
}
```

```java
package hamiltonianGraph;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Stack;

// vertexy a hrany "na pozadí" - prohledávací algorimus
public class Controller {

    Generator generator;
    ArrayList<Vertex> vertexList;
    ArrayList<Edge> lineList;

    int vertexCount = 6;        //výchozí počet vertxů
    boolean exist = true;       // existence ham. grafu
    boolean InStepflag = false;      //true pokud nalezena spojnice mezi body
    int[][] path;               //pole hran
    int[][] vertexDuplicateChecker;
    int visitCount = 0;         //počet navštívených uzlů
    int[] result;
    int backStepFlag = -2;
    Stack<Integer> stack = new Stack<>();
    ArrayList<Vertex> solvedVertexes = new ArrayList<>(); //list výsledných uzlů ham. grafu

    public Controller() { init(); }


    public void init() {
        generator = new Generator();
        vertexList = new ArrayList<>();
        lineList = new ArrayList<>();
        stack = new Stack<>();
        visitCount = 0;
        exist = true;
        result = new int[vertexCount + 30];
    }

    // generování uzlů - automatické
    public void generate() {

        initTables();
        int x, y;
        for (int i = 0; i < vertexCount; i++) {
            x = generator.getRandomIntegerBetweenRange(0, 10);
            y = generator.getRandomIntegerBetweenRange(0, 10);

            if (vertexDuplicateChecker[x][y] == 0) {
                vertexDuplicateChecker[x][y] = 1;
                Vertex vertex = new Vertex(x, y);
                vertexList.add(vertex);
            } else {
                i--;
            }
        }
        generateLines();
    }

    // generování uzlů s vlastními uzly
    public void generateCustom(List<Node> nodes, List<Edge> edges) {

        vertexCount = nodes.size();
        initTables();
```

```
        for (int i = 0; i < nodes.size(); i++) {
            vertexList.add(new Vertex(i, i));
            nodes.get(i).setVertex(new Vertex(i, i));
        }
        customLines(nodes, edges);
    }

    // generování hran s vlastními hranami
    private void customLines(List<Node> nodes, List<Edge> edges) {

        int m = 0;
        int n = 0;
        Node n1, n2;


        for (Edge edge : edges) {
            n1 = edge.getN1();
            n2 = edge.getN2();

            for (int j = 0; j < nodes.size(); j++) {
                if (nodes.get(j) == n1) {
                    m = j;
                } else if (nodes.get(j) == n2) {
                    n = j;
                }
            }
            path[m][n] = 1;
            path[n][m] = 1;
        }
    }
    public void initTables() {
        path = new int[vertexCount][vertexCount];
        for (int[] row : path) {
            Arrays.fill(row, val: 0);
        }

        //velikost podle generatoru, nyni 0 - 9
        vertexDuplicateChecker = new int[30][30];
        for (int[] row : vertexDuplicateChecker) {
            Arrays.fill(row, val: 0);
        }
    }


    // generování hran - automatické
    private void generateLines() {
        //úplný graf - max hran
        int max = (vertexCount * (vertexCount - 1)) / 2;

        int mode;
        if (vertexCount < 4) {
            mode = generator.getRandomIntegerBetweenRange(1, 14);
        } else if (vertexCount < 6) {
            mode = generator.getRandomIntegerBetweenRange(1, 11);
        } else if (vertexCount < 7) {
            mode = generator.getRandomIntegerBetweenRange(1, 9);
        } else {
            mode = generator.getRandomIntegerBetweenRange(1, 8);
        }
```

```
        //počet hran poodle vybraneho modu
        int numberOfLines;
        if (mode < 3) {
            numberOfLines = generator.getRandomIntegerBetweenRange(0, max / 2);

        } else if (mode > 3 && mode < 6) {
            numberOfLines = generator.getRandomIntegerBetweenRange(max / 2, max * 2 / 3);

        } else {
            numberOfLines = generator.getRandomIntegerBetweenRange(max * 2 / 3, max);
        }

        for (
                int i = 0;
                i < numberOfLines; i++) {
            int m = generator.getRandomIntegerBetweenRange(0, vertexCount - 1);
            int n = generator.getRandomIntegerBetweenRange(0, vertexCount - 1);

            if ((m != n) && (path[m][n] != 1) && (path[n][m] != 1)) {

                path[m][n] = 1;
                path[n][m] = 1; // vytvori symetrickou tabulku hran
            } else {
                i--;
            }
        }
    }
    // prohledávání grafu - řídící funkce
    public boolean search() {

        int i = 0;
        int j = 0;
        vertexList.get(0).setVisited(true);
        solvedVertexes.add(new Vertex(vertexList.get(0).getX(), vertexList.get(0).getY()));
            rowInspect(i, j);

            if (exist) {
                //graf existuje
                solvedVertexes.add(new Vertex(vertexList.get(0).getX(), vertexList.get(0).getY()));
                return true;
            } else {
                //graf neexistuje
                return false;
            }
    }
    // prohledávání grafu - rekurzivní funkce
    private void rowInspect(int i, int j) {
        InStepflag = false;

        //zkouška jestli je kam pokračovat
        while (j < vertexCount) {
            if ((path[i][j] == 1) && (!vertexList.get(j).isVisited())) {
                InStepflag = true;
                break;
            }
            // ošetření, aby se "neodešlo" z tabulky hran
            if (j != vertexCount - 1) {
                j++;
            } else {
                break;
            }
        }

        //přenastavení hodnot, při návratu
        if (backStepFlag != -2) {
            vertexList.get(backStepFlag).setVisited(false);
            backStepFlag = -2;
            visitCount--;
        }
```

```
        // je možné zanoření
        if (InStepflag) {
            stack.push(i);
            stack.push(j);
            vertexList.get(j).setVisited(true);

            solvedVertexes.add(new Vertex(vertexList.get(j).getX(), vertexList.get(j).getY()));
            visitCount++;
            rowInspect(j,  j: 0);

            // kontrola existence poslední hrany vedoucí do výchozího uzlu
        } else if ((visitCount + 1 == vertexCount) && (path[i][0] == 1)) {

        } else {
            // návrat
            if (!stack.isEmpty()) {
                backStepFlag = i;

                j = stack.pop();
                i = stack.pop();

                if (solvedVertexes.size() != 0) {
                    solvedVertexes.remove( index: solvedVertexes.size() - 1);
                } rowInspect(i, j);

            } else {
                //ham. graf neexistuje
                exist = false;
            }
        }
    }

    public void setVertexCount(int vertexCount) { this.vertexCount = vertexCount; }
}
```

```
package hamiltonianGraph;
import java.awt.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
import javax.swing.*;

public class GraphPanel extends JComponent {
    private static final int WIDE = 1250;
    private static final int HIGH = 600;
    private static final int RADIUS = 20;
    private ControlPanel control = new ControlPanel();
    private Generator generator = new Generator();
    private int radius = RADIUS;
    private List<Node> nodes = new ArrayList<>();       //vizuální uzly
    private List<Node> selected = new ArrayList<>();    //vybrané uzly
    private List<Edge> edges = new ArrayList<>();       //vizuální hrany
    private Point mousePt = new Point( x: WIDE / 2,  y: HIGH / 2);
    private Rectangle mouseRectangle = new Rectangle();     //pro označování uzlů
    private boolean selecting = false;
    private Controller controller = new Controller();
    private Example example = new Example(controller);
    private Point lastMousePt;
    private int shiftPosition = (int)(radius * 1.5); // aby se nove uzly neprekryvaly
    private int vertexCountForGenerate = 6;
    private String result;
    private List<Edge> resultEdges = new ArrayList<>();     //výsledné hrany ham. kružnice
    public int switchLanguage = 0;
```

```
    public GraphPanel() {
        this.setOpaque(true);
        this.addMouseListener(new MouseHandler());
        this.addMouseMotionListener(new MouseMotionHandler());
    }

    @Override
    public Dimension getPreferredSize() { return new Dimension(WIDE, HIGH); }

    //vykreslování
    @Override
    public void paintComponent(Graphics g) {
        g.setColor(new Color( rgb: 0x00f0f0f0));
        g.fillRect( i: 0,  i1: 0, getWidth(), getHeight());
        for (Edge e : edges) {
            e.draw(g);
        }

        //pro obarvení hran výsledku
        if (resultEdges.size() != 0) {
            for (Edge en : resultEdges) {
                en.draw(g, Color.green);
            }
        }

        for (Node n : nodes) {
            n.draw(g);
        }
        if (selecting) {
            g.setColor(Color.darkGray);
            g.drawRect(mouseRectangle.x, mouseRectangle.y,
                    mouseRectangle.width, mouseRectangle.height);
        }
    }
    private class MouseHandler extends MouseAdapter {

        @Override
        public void mouseReleased(MouseEvent e) {
            selecting = false;
            mouseRectangle.setBounds( x: 0,  y: 0,  width: 0,  height: 0);
            if (e.isPopupTrigger()) {
                showPopup(e);
            }
            e.getComponent().repaint();
        }

        @Override
        public void mousePressed(MouseEvent e) {
            mousePt = e.getPoint();

            if (e.isPopupTrigger()) {
                Node.selectOne(nodes, mousePt);
                showPopup(e);
            } else if (Node.selectOne(nodes, mousePt)) {
                selecting = false;
            } else {
                Node.selectNone(nodes);
                selecting = true;
            }
            e.getComponent().repaint();
        }

        private void showPopup(MouseEvent e) {
            control.popup.show(e.getComponent(), e.getX(), e.getY());
        }
    }
```

```
    private class MouseMotionHandler extends MouseMotionAdapter {

        Point delta = new Point();

        @Override
        public void mouseDragged(MouseEvent e) {
            if (selecting) {
                mouseRectangle.setBounds(
                        Math.min(mousePt.x, e.getX()),
                        Math.min(mousePt.y, e.getY()),
                        Math.abs(mousePt.x - e.getX()),
                        Math.abs(mousePt.y - e.getY()));
                Node.selectRectangle(nodes, mouseRectangle);
            } else {
                delta.setLocation(
                         x: e.getX() - mousePt.x,
                         y: e.getY() - mousePt.y);
                Node.updatePosition(nodes, delta);
                mousePt = e.getPoint();
            }
            e.getComponent().repaint();
            lastMousePt = mousePt;
            shiftPosition = (int)(radius * 1.5);
        }
    }

    public JToolBar getControlPanel() {
        return control;
    }
```

```
    public class ControlPanel extends JToolBar {
        //ovládací prvky
        private Action newNode = new NewNodeAction( name: "New");
        public Action clearAll = new ClearAction( name: "Clear");
        private Action generate = new GenerateAction( name: "Generate");
        private Action delete = new DeleteAction( name: "Delete");
        private Action save = new SaveAction( name: "Save");
        private Action solve = new SolveAction( name: "Solve");
        private Action connect = new ConnectAction( name: "Connect");
        private Action prepared = new PreparedAction( name: "Examples");
        private Action example1 = new Example1Action( name: "Example 1");
        private Action example2 = new Example2Action( name: "Example 2");
        private Action example3 = new Example3Action( name: "Example 3");
        private Action example4 = new Example4Action( name: "Example 4");
        private Action example5 = new Example5Action( name: "Example 5");
        private Action example6 = new Example6Action( name: "Example 6");
        private Action example7 = new Example7Action( name: "Example 7");
        private Action example8 = new Example8Action( name: "Example 8");
        private Action help = new HelpAction( name: "Help");
        private Action language = new LangAction( name: "CZ/EN");
        private JPopupMenu popup = new JPopupMenu();
        private JLabel resultL = new JLabel( text: "Answer");
        private JMenuBar menuBar = new JMenuBar();
        private JMenu menu = new JMenu(prepared);
        private JLabel sizeLabel = new JLabel( text: "Size:");
        private JLabel countLabel = new JLabel( text: "Vertex count:");

        ControlPanel() {
            this.setLayout(new FlowLayout(FlowLayout.LEFT));
            this.setBackground(Color.lightGray);
            JSpinner jsSize = new JSpinner();
            jsSize.setModel(new SpinnerNumberModel(RADIUS,  minimum: 5,  maximum: 100,  stepSize: 5));
            jsSize.addChangeListener(e -> {
                JSpinner s = (JSpinner) e.getSource();
                radius = (Integer) s.getValue();
                Node.updateRadius(nodes, radius);
                GraphPanel.this.repaint();
            });
```

```
        JSpinner jsVertex = new JSpinner();
        jsVertex.setModel(new SpinnerNumberModel( value: 6,  minimum: 3,  maximum: 10,  stepSize: 1));
        jsVertex.addChangeListener(e -> {
            JSpinner s = (JSpinner) e.getSource();
            controller.setVertexCount((Integer) s.getValue());
            vertexCountForGenerate = controller.vertexCount;
            GraphPanel.this.repaint();
        });
        this.add(new JButton(language));
        this.add(new JButton(help));
        this.add(Box.createRigidArea(new Dimension( width: 5,  height: 0)));
        this.add(new JButton(newNode));
        this.add(new JButton(delete));
        this.add(new JButton(connect));
        this.add(new JButton(save));
        this.add(Box.createRigidArea(new Dimension( width: 5,  height: 0)));
        this.add(new JButton(generate));
        this.add(Box.createRigidArea(new Dimension( width: 5,  height: 0)));
        this.add(new JButton(clearAll));
        this.add(new JButton(solve));
        this.add(Box.createRigidArea(new Dimension( width: 5,  height: 0)));

        this.add(menuBar);
        menuBar.add(menu);
        menu.add(new JMenuItem(example1));
        menu.add(new JMenuItem(example2));
        menu.add(new JMenuItem(example3));
        menu.add(new JMenuItem(example4));
        menu.add(new JMenuItem(example5));
        menu.add(new JMenuItem(example6));
        menu.add(new JMenuItem(example7));
        menu.add(new JMenuItem(example8));
        this.add(Box.createRigidArea(new Dimension( width: 5,  height: 0)));
        this.add(sizeLabel);
        this.add(jsSize);
        this.add(Box.createRigidArea(new Dimension( width: 5,  height: 0)));
        this.add(countLabel);
        this.add(jsVertex);
        this.add(Box.createRigidArea(new Dimension( width: 5,  height: 0)));
        this.add(resultL);
        popup.add(new JMenuItem(newNode));
        popup.add(new JMenuItem(connect));
        popup.add(new JMenuItem(delete));
    }
}
// nový uzel
private class NewNodeAction extends AbstractAction {

    public NewNodeAction(String name) {
        super(name);
    }

    public void actionPerformed(ActionEvent e) {
        // kresleni pres New, odsazuje nove body, aby nebyly na stejnem miste
        if (nodes.size() == 0) {
            createNode(mousePt.getLocation());
            shiftPosition = (int)(radius * 1.5);

        } else if (lastMousePt.x == (mousePt.getLocation().x) && lastMousePt.y == (mousePt.getLocation().y)) {
            Point p = new Point( x: mousePt.getLocation().x + shiftPosition,  y: mousePt.getLocation().y + shiftPosition);
            createNode(p);

            shiftPosition = (int)(radius * 1.5) + shiftPosition;
        } else {
            createNode(mousePt.getLocation());
            shiftPosition = (int)(radius * 1.5);
        }
```

```
            lastMousePt = mousePt;
            control.save.setEnabled(true);
            control.solve.setEnabled(false);
        }
    }

    //ukázkové příklady
    private static class PreparedAction extends AbstractAction {
        public PreparedAction(String name) { super(name); }
        public void actionPerformed(ActionEvent e) {
        }
    }

    private class Example1Action extends AbstractAction {
        public Example1Action(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            clearAll();
            example.getExample1();
            btnEnableConfig( exampleSet: true);

        }
    }
```

```
    private class Example2Action extends AbstractAction {
        public Example2Action(String name) { super(name); }

        public void actionPerformed(ActionEvent e) {
            clearAll();
            example.getExample2();
            btnEnableConfig( exampleSet: true);
        }
    }

    public class Example3Action extends AbstractAction {
        public Example3Action(String name) { super(name); }

        public void actionPerformed(ActionEvent e) {
            clearAll();
            example.getExample3();
            btnEnableConfig( exampleSet: true);
        }
    }

    private class Example4Action extends AbstractAction {
        public Example4Action(String name) { super(name); }

        public void actionPerformed(ActionEvent e) {
            clearAll();
            example.getExample4();
            btnEnableConfig( exampleSet: true);
        }
    }
    private class Example5Action extends AbstractAction {
        public Example5Action(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            clearAll();
            example.getExample5();
            btnEnableConfig( exampleSet: true);
        }
    }
```

```
    private class Example6Action extends AbstractAction {
        public Example6Action(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            clearAll();
            example.getExample6();
            btnEnableConfig( exampleSet: true);
        }
    }

    private class Example7Action extends AbstractAction {
        public Example7Action(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            clearAll();
            example.getExample7();
            btnEnableConfig( exampleSet: true);
        }
    }

    private class Example8Action extends AbstractAction {
        public Example8Action(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            clearAll();
            example.getExample8();
            btnEnableConfig( exampleSet: true);
        }
    }
    //nápověda
    private class HelpAction extends AbstractAction {
        public HelpAction(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            helpFrame();
        }
    }
    // změna jazyka
    private class LangAction extends AbstractAction {
        public LangAction(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            if (switchLanguage == 0) {
                control.help.putValue(Action.NAME,  o: "Nápověda");
                control.newNode.putValue(Action.NAME,  o: "Nový");
                control.delete.putValue(Action.NAME,  o: "Smazat");
                control.save.putValue(Action.NAME,  o: "Uložit");
                control.generate.putValue(Action.NAME,  o: "Generovat");
                control.prepared.putValue(Action.NAME,  o: "Příklady");
                control.connect.putValue(Action.NAME,  o: "Spojit");
                control.clearAll.putValue(Action.NAME,  o: "Smazat vše");
                control.example1.putValue(Action.NAME,  o: "Příklad 1");
                control.example2.putValue(Action.NAME,  o: "Příklad 2");
                control.example3.putValue(Action.NAME,  o: "Příklad 3");
                control.example4.putValue(Action.NAME,  o: "Příklad 4");
                control.example5.putValue(Action.NAME,  o: "Příklad 5");
                control.example6.putValue(Action.NAME,  o: "Příklad 6");
                control.example7.putValue(Action.NAME,  o: "Příklad 7");
                control.example8.putValue(Action.NAME,  o: "Příklad 8");
                control.solve.putValue(Action.NAME,  o: "Vyřešit");
                control.sizeLabel.setText("Velikost:");
                control.countLabel.setText("Počet uzlů:");
```

```
            if (control.resultL.getText().equals("Answer")) {
                control.resultL.setText("Odpověď");
            } else if (control.resultL.getText().equals("Too few vertexes. Hamiltonian graph does not exist.")) {
                control.resultL.setText("Příliž málo uzlů. Hamiltonovský graf nemůže existovat.");
            } else if (control.resultL.getText().equals("Hamiltonian graph exists.")) {
                control.resultL.setText("Hamiltonovský graf existuje.");
            } else if (control.resultL.getText().equals("Hamiltonian graph does not exist.")) {
                control.resultL.setText("Hamiltonovský graf neexistuje.");
            }
            switchLanguage = 1;
        } else if (switchLanguage == 1) {
            control.help.putValue(Action.NAME,  o: "Help");
            control.newNode.putValue(Action.NAME,  o: "New");
            control.delete.putValue(Action.NAME,  o: "Delete");
            control.save.putValue(Action.NAME,  o: "Save");
            control.generate.putValue(Action.NAME,  o: "Generate");
            control.prepared.putValue(Action.NAME,  o: "Examples");
            control.connect.putValue(Action.NAME,  o: "Connect");
            control.clearAll.putValue(Action.NAME,  o: "Clear");
            control.example1.putValue(Action.NAME,  o: "Example 1");
            control.example2.putValue(Action.NAME,  o: "Example 2");
            control.example3.putValue(Action.NAME,  o: "Example 3");
            control.example4.putValue(Action.NAME,  o: "Example 4");
            control.example5.putValue(Action.NAME,  o: "Example 5");
            control.example6.putValue(Action.NAME,  o: "Example 6");
            control.example7.putValue(Action.NAME,  o: "Example 7");
            control.example8.putValue(Action.NAME,  o: "Example 8");
            control.solve.putValue(Action.NAME,  o: "Solve");
            control.sizeLabel.setText("Size:");
            control.countLabel.setText("Vertex count:");
            if (control.resultL.getText().equals("Odpověď")) {
                control.resultL.setText("Answer");
            } else if (control.resultL.getText().equals("Příliž málo uzlů. Hamiltonovský graf nemůže existovat.")) {
                control.resultL.setText("Too few vertexes. Hamiltonian graph does not exist.");
            } else if (control.resultL.getText().equals("Hamiltonovský graf existuje.")) {
                control.resultL.setText("Hamiltonian graph exists.");
            } else if (control.resultL.getText().equals("Hamiltonovský graf neexistuje.")) {
                control.resultL.setText("Hamiltonian graph does not exist.");
            }
            switchLanguage = 0;
        }
    }

}
// smazání všeho
private class ClearAction extends AbstractAction {
    public ClearAction(String name) {
        super(name);
    }

    public void actionPerformed(ActionEvent e) {
        clearAll();
    }
}
```

```
    private void clearAll() {
        controller.vertexList.clear();
        nodes.clear();
        edges.clear();
        resultEdges.clear();
        controller.solvedVertexes.clear();
        controller.backStepFlag = -2;
        if (switchLanguage == 0) {
            control.resultL.setText(result = "Answer");
        } else {
            control.resultL.setText(result = "Odpověď");
        }
        repaint();

        control.save.setEnabled(false);
        control.delete.setEnabled(true);
        control.newNode.setEnabled(true);
        control.connect.setEnabled(true);
    }

    //náhodné generování
    private class GenerateAction extends AbstractAction {

        public GenerateAction(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            clearAll();
            // pomuze vygenerovat tolik uzlu, jako je uvedeno v uzivatelskem rozhrani
            if (controller.vertexCount != vertexCountForGenerate) {
                controller.setVertexCount(vertexCountForGenerate);
            }
            controller.generate();
            prepareNode( IsRadomize: true);
            prepareEdges();
            control.solve.setEnabled(true);
            btnEnableConfig( exampleSet: false);
        }
    }
    //spustí prohledávání grafu
    private class SolveAction extends AbstractAction {

        public SolveAction(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            boolean errorFlag = false;
            if (nodes.size() == 1 || nodes.size() == 2) {
                if (switchLanguage == 0) {
                    result = ("Too few vertexes. Hamiltonian graph does not exist.");
                } else {
                    result = ("Příliž málo uzlů. Hamiltonovský graf nemůže existovat.");
                }

                //pokud neni uzel nedelej nic, pokud 1 až 2 - ham. neexistuje nehledej, jinak hledej
            } else if (nodes.size() != 0) {
                boolean exist = false;
                try {
                    exist = controller.search();
                } catch (StackOverflowError en) {
                    if (switchLanguage == 0) {
                        JOptionPane.showMessageDialog(control,
                                message: "Graph is too difficult, programme can not solve its.",
                                title: "Warning", JOptionPane.WARNING_MESSAGE);
```

```
                } else {
                    JOptionPane.showMessageDialog(control,
                            message: "Příliš složitý graf, program ho nedokáže vyhodnotit.",
                            title: "Varování", JOptionPane.WARNING_MESSAGE);
                }
                errorFlag = true;
            }

            if (!errorFlag) {
                if (exist) {
                    if (switchLanguage == 0) {
                        result = ("Hamiltonian graph exists.");
                    } else {
                        result = ("Hamiltonovský graf existuje.");
                    }

                    for (int i = 0; i < nodes.size(); i++) {
                        setColorNode(i, Color.green);
                    }
                } else {
                    if (switchLanguage == 0) {
                        result = ("Hamiltonian graph does not exist.");
                    } else {
                        result = ("Hamiltonovský graf neexistuje.");
                    }
                }
            }
        }

        if (!errorFlag) {
            control.resultL.setText(result);
            coloredEdges();
        }
        control.solve.setEnabled(false);
        control.save.setEnabled(false);
        controller.init();
    }
}

// propojení hranami
private class ConnectAction extends AbstractAction {

    public ConnectAction(String name) {
        super(name);
    }

    public void actionPerformed(ActionEvent e) {
        boolean duplicityFlag = false;
        Node.getSelected(nodes, selected);
        if (selected.size() > 1) {
            for (int i = 0; i < selected.size() - 1; ++i) {
                Node n1 = selected.get(i);
                Node n2 = selected.get(i + 1);

                // zabrani duplicitnim hranam mezi stejnymi body
                for (int d = 0; d < edges.size() - 1; d++) {
                    if ((n1 == edges.get(d).getN1() && n2 == edges.get(d).getN2())
                            || (n2 == edges.get(d).getN1() && n1 == edges.get(d).getN2())) {
                        duplicityFlag = true;
                    }
                }
                if (!duplicityFlag) {
                    edges.add(new Edge(n1, n2));
                }
                duplicityFlag = false;
            }
        }
        repaint();
```

```
        }
    }

    //smazání označených uzlů
    private class DeleteAction extends AbstractAction {
        public DeleteAction(String name) {
            super(name);
        }

        public void actionPerformed(ActionEvent e) {
            control.solve.setEnabled(false);
            ListIterator<Node> iter = nodes.listIterator();
            while (iter.hasNext()) {
                Node n = iter.next();
                if (n.isSelected()) {
                    deleteEdges(n);
                    iter.remove();
                    shiftPosition -= (int)(radius * 1.5);
                    controller.vertexCount--;
                }
            }
            repaint();
        }

        private void deleteEdges(Node n) {
            edges.removeIf(e -> e.getN1() == n || e.getN2() == n);
        }
    }

    // přípravení nakresleného grafu pro prohledávání
    private class SaveAction extends AbstractAction {
        public SaveAction(String name) {
            super(name);
        }
        public void actionPerformed(ActionEvent e) {
            btnEnableConfig( exampleSet: false);

            if (nodes.size() != 0) {
                controller.generateCustom(nodes, edges);
            }
        }
    }

    // vytvoření uzlu
    public void createNode(Point point) {
        Node.selectNone(nodes);
        Color color = Color.GRAY;
        Node n = new Node(point, radius, color);
        n.setSelected(true);
        nodes.add(n);
        repaint();
    }

    // vytvoření uzlu a napojení na vertex
    public void createNode(Point point, Vertex vertex) {
        Node.selectNone(nodes);
        Color color = Color.blue;
        Node n = new Node(point, radius, color, vertex);
        n.setSelected(true);
        nodes.add(n);
        repaint();
    }
```

```java
    // barva uzlu
    public void setColorNode(int positionInList, Color color) {
        Node.selectNone(nodes);
        nodes.get(positionInList).setColor(color);
        repaint();
    }

    // propojení node s vertexem
    public void prepareNode(boolean IsRadomize) {
        int xv;
        int yv;
        for (int i = 0; i < controller.vertexCount; i++) {
            xv = controller.vertexList.get(i).getX();
            yv = controller.vertexList.get(i).getY();

            if (IsRadomize) {
                Point p = new Point(generator.noCoverInGenerate(WIDE, HIGH, nodes));
                xv = p.x;
                yv = p.y;
            }
            createNode(new Point(xv, yv), controller.vertexList.get(i));
        }
    }

    // vykresleni hran
    public void prepareEdges() {
        for (int i = 0; i < controller.vertexCount; i++) {
            for (int j = 0; j < controller.vertexCount; j++) {
                if (controller.path[i][j] == 1) {
                    edges.add(new Edge(nodes.get(i), nodes.get(j)));
                }
            }
        }
    }
    // obarveni hran
    public void coloredEdges() {
        for (int i = 0; i < (controller.solvedVertexes.size() - 1); i++) {

            for (Edge edge : edges) {
                if (edge.getN1().getVertex().getX() == controller.solvedVertexes.get(i).getX()
                        && edge.getN1().getVertex().getY() == controller.solvedVertexes.get(i).getY()
                        && edge.getN2().getVertex().getX() == controller.solvedVertexes.get(i + 1).getX()
                        && edge.getN2().getVertex().getY() == controller.solvedVertexes.get(i + 1).getY()) {
                    resultEdges.add(edge);
                    break;
//
                } else if (edge.getN2().getVertex().getX() == controller.solvedVertexes.get(i).getX()
                        && edge.getN2().getVertex().getY() == controller.solvedVertexes.get(i).getY()
                        && edge.getN1().getVertex().getX() == controller.solvedVertexes.get(i + 1).getX()
                        && edge.getN1().getVertex().getY() == controller.solvedVertexes.get(i + 1).getY()) {
                    resultEdges.add(edge);
                    break;
                }
            }
        }
    }

    // nastavení dostupnosti tlačítek
    private void btnEnableConfig(boolean exampleSet) {
        // donastavení věcí pro ukázkové příklady, když false je to pro generate, jinak pro example
        if (exampleSet) {
            prepareNode( IsRadomize: false);
            prepareEdges();
        }
        control.solve.setEnabled(true);
        control.delete.setEnabled(false);
        control.newNode.setEnabled(false);
        control.connect.setEnabled(false);
    }
```

```java
    // okno napovedy
    private void helpFrame() {
        EventQueue.invokeLater(() -> {
            JFrame frame = new JFrame( title: "Help");
            frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

            JPanel panel = new JPanel();
            panel.setLayout(new BorderLayout());
            JTextArea textArea = new JTextArea( rows: 25,  columns: 50);
            textArea.setEditable(false);
            textArea.setFont(textArea.getFont().deriveFont(14f));
            textArea.setText(getHelpString());

            panel.add(textArea, BorderLayout.LINE_START);

            JScrollPane scroller = new JScrollPane(textArea);
            scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED);
            scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);

            panel.add(scroller, BorderLayout.CENTER);
            frame.getContentPane().add(BorderLayout.CENTER, panel);
            frame.pack();
            frame.setSize( width: 960,  height: 560);
            frame.setLocationRelativeTo(null);
            frame.setVisible(true);

        });
    }
//text nápovědy
private String getHelpString() {
    return ("\n  Aplikace slouží pro hledání hamiltonovských grafů za pomoci backtrackingu." +
            " Hamiltonovský graf existuje, když lze projít všemi body právě jednou \n" +
            "  a vrátit se do výchozího bodu. Grafy lze kreslit nebo generovat. Případně lze použít některý z předpřipravených příkladů. \n" +
            "\n  Ovládání:\n  Levé tlačítko myši označí jeden uzel. Více uzlů lze označit tažením myši." +
            " Stisknutí pravého tlačítka myši otevře vertikální menu" +
            "\n  s tlačítky New, Delete, Connect.\n" +
            "\n  Vlastnosti tlačítek:\n" +
            "  CZ/EN - změna jazyka: angličtina nebo čeština, Nový - vytvoří nový uzel, Smazat - smaže označené uzly," +
            " Spojit - propojí označené uzly hranami,\n" +
            "  Uložit - uloží nakreslený graf, aby mohl být prohledán, Generovat - vygeneruje náhodný graf o tolika uzlech," +
            " kolik je uvedeno v poli Počet uzlů,\n" +
            "  Smazat vše - vyčistí kreslicí plátno, Vyřešit - prohledá a vyhodnotí nakreslený graf, Příklady - sada připravených příkladů\n\n" +

            "  The application is used to search for Hamiltonian graphs using backtracking. A Hamiltonian graph exists" +
            " if it is possible to go through the graph \n" +
            "  in a loop and visit each node exactly once and return to the starting node. Graphs can be drawn or" +
            " generated within the application. \n" +
            "  It also contains a set of prepared examples.\n" +
            "\n  Controls:\n" +
            "  Left mouse button  is used to select either a singular node or multiple nodes by dragging the mouse around them.\n" +
            "  Right mouse button opens vertical menu with buttons: New, Delete, Connect.\n" +
            " \n  Buttons functions:\n" +
            "  CZ/EN - switches languages: English or Czech, New - creates new node, Delete - deletes selected nodes,\n" +
            "  Connect - connects selected nodes with edges, Save - saves the drawn graph so that it can be searched,\n" +
            "  Generate - generates a random graph with as many nodes as specified in the Vertex count field,\n" +
            "  Clear - removes all from drawing screen, Solve - searches and solves drawn graph," +
            "\n  Example - opens menu with prepared examples");
}
```

```java
package hamiltonianGraph;

import java.awt.*;
import java.util.List;

//grafická reprezentace uzlu
public class Node {

    private Point p;
    private int r;
    private Color color;
    private boolean selected = false;
    private Rectangle rectangle = new Rectangle();
    private Vertex vertex;

    public Node(Point p, int r, Color color, Vertex vertex) {
        this.p = p;
        this.r = r;
        this.color = color;
        this.vertex = vertex;
        setBoundary(rectangle);
    }

    public Node(Point p, int r, Color color) {
        this.p = p;
        this.r = r;
        this.color = color;
        setBoundary(rectangle);
    }

    public void setColor(Color color) { this.color = color; }

    public Vertex getVertex() { return vertex; }

    // výpočet hranice výběrového obdélníku
    private void setBoundary(Rectangle b) { b.setBounds( x: p.x - r,  y: p.y - r,  width: 2 * r,  height: 2 * r); }

    public void draw(Graphics g) {
        g.setColor(this.color);
        g.fillOval(rectangle.x, rectangle.y, rectangle.width, rectangle.height);
        if (selected) {
            g.setColor(Color.darkGray);
            g.drawRect(rectangle.x, rectangle.y, rectangle.width, rectangle.height);
        }
    }
    //vrátí označené uzly
    public static void getSelected(List<Node> list, List<Node> selected) {
        selected.clear();
        for (Node n : list) {
            if (n.isSelected()) {
                selected.add(n);
            }
        }
    }

    public static void selectNone(List<Node> list) {
        for (Node n : list) {
            n.setSelected(false);
        }
    }
```

```java
    public static boolean selectOne(List<Node> list, Point p) {
        for (Node n : list) {
            if (n.contains(p)) {
                if (!n.isSelected()) {
                    Node.selectNone(list);
                    n.setSelected(true);
                }
                return true;
            }
        }
        return false;
    }

    // výběr uzlu z označovacího obdélníku
    public static void selectRectangle(List<Node> list, Rectangle r) {
        for (Node n : list) {
            n.setSelected(r.contains(n.p));
        }
    }

    //změna pozice
    public static void updatePosition(List<Node> list, Point d) {
        for (Node n : list) {
            if (n.isSelected()) {
                n.p.x += d.x;
                n.p.y += d.y;
                n.setBoundary(n.rectangle);
            }
        }
    }
    //změna poloměru uzlu
    public static void updateRadius(List<Node> list, int r) {
        for (Node n : list) {
            if (n.isSelected()) {
                n.r = r;
                n.setBoundary(n.rectangle);
            }
        }
    }

    public Point getLocation() {
        return p;
    }

    public boolean contains(Point p) { return rectangle.contains(p); }

    public boolean isSelected() { return selected; }

    public void setSelected(boolean selected) { this.selected = selected; }
}
```

```java
package hamiltonianGraph;

public class Example {

    private Controller controller;

    public  Example(Controller controller) { this.controller = controller; }

    /* Úprava příkladu
    - controller.setVertexCount(6); - nastaví počet vertexů grafu,
    - controller.initTables(); - nastavuje program pro vložení příkladu, musí být volán
    - controller.vertexList.add(new Vertex(150,150)); - vytvoří nový vertex se souřadnicemi,
     které jsou v závorce
        počet vložených vertexů musí přesně odpovídat, počtu vertexů, které byly nastaveny výše
    - controller.path[0][1] = 1; - vytvoří hranu, zde mezi nultým a prvním vertexem, hrana musí
    být vytvořena i v opačném směru
     */

    public void getExample1() {
        controller.setVertexCount(6);
        controller.initTables();
        controller.vertexList.add(new Vertex( x: 150, y: 150));
        controller.vertexList.add(new Vertex( x: 260, y: 120));
        controller.vertexList.add(new Vertex( x: 210, y: 220));
        controller.vertexList.add(new Vertex( x: 350, y: 180));
        controller.vertexList.add(new Vertex( x: 400, y: 260));
        controller.vertexList.add(new Vertex( x: 140, y: 300));

        controller.path[0][1] = 1;
        controller.path[1][2] = 1;
        controller.path[2][3] = 1;
        controller.path[3][0] = 1;
        controller.path[3][4] = 1;
        controller.path[5][4] = 1;
        controller.path[5][2] = 1;
        controller.path[1][0] = 1;
        controller.path[2][1] = 1;
        controller.path[3][2] = 1;
        controller.path[0][3] = 1;
        controller.path[4][3] = 1;
        controller.path[4][5] = 1;
        controller.path[2][5] = 1;

    }

    public void getExample2() {
        controller.setVertexCount(5);
        controller.initTables();
        controller.vertexList.add(new Vertex( x: 160, y: 375));
        controller.vertexList.add(new Vertex( x: 120, y: 240));
        controller.vertexList.add(new Vertex( x: 250, y: 140));
        controller.vertexList.add(new Vertex( x: 380, y: 240));
        controller.vertexList.add(new Vertex( x: 340, y: 375));


        controller.path[0][1] = 1;
        controller.path[0][2] = 1;
        controller.path[0][3] = 1;
        controller.path[0][4] = 1;
        controller.path[1][2] = 1;
        controller.path[1][3] = 1;
        controller.path[1][4] = 1;
        controller.path[2][3] = 1;
        controller.path[2][4] = 1;
        controller.path[3][4] = 1;
```

```
        controller.path[1][0] = 1;
        controller.path[2][0] = 1;
        controller.path[3][0] = 1;
        controller.path[4][0] = 1;
        controller.path[2][1] = 1;
        controller.path[3][1] = 1;
        controller.path[4][1] = 1;
        controller.path[3][2] = 1;
        controller.path[4][2] = 1;
        controller.path[4][3] = 1;

    }

    public void getExample3() {
        controller.setVertexCount(8);
        controller.initTables();
        controller.vertexList.add(new Vertex( x: 50, y: 50));
        controller.vertexList.add(new Vertex( x: 380, y: 70));
        controller.vertexList.add(new Vertex( x: 190, y: 100));
        controller.vertexList.add(new Vertex( x: 200, y: 270));
        controller.vertexList.add(new Vertex( x: 350, y: 210));
        controller.vertexList.add(new Vertex( x: 45, y: 200));
        controller.vertexList.add(new Vertex( x: 60, y: 335));
        controller.vertexList.add(new Vertex( x: 340, y: 320));

        controller.path[0][1] = 1;
        controller.path[1][2] = 1;
        controller.path[2][3] = 1;
        controller.path[3][0] = 1;
        controller.path[3][4] = 1;
        controller.path[5][4] = 1;
        controller.path[5][2] = 1;
        controller.path[6][7] = 1;
        controller.path[7][2] = 1;
        controller.path[6][1] = 1;
        controller.path[1][0] = 1;
        controller.path[2][1] = 1;
        controller.path[3][2] = 1;
        controller.path[0][3] = 1;
        controller.path[4][3] = 1;
        controller.path[4][5] = 1;
        controller.path[2][5] = 1;
        controller.path[7][6] = 1;
        controller.path[2][7] = 1;
        controller.path[1][6] = 1;

    }

    public void getExample4() {
        controller.setVertexCount(7);
        controller.initTables();
        controller.vertexList.add(new Vertex( x: 200, y: 100));
        controller.vertexList.add(new Vertex( x: 400, y: 110));
        controller.vertexList.add(new Vertex( x: 200, y: 200));
        controller.vertexList.add(new Vertex( x: 400, y: 200));
        controller.vertexList.add(new Vertex( x: 200, y: 300));
        controller.vertexList.add(new Vertex( x: 400, y: 300));
        controller.vertexList.add(new Vertex( x: 200, y: 400));

        controller.path[0][1] = 1;
        controller.path[0][3] = 1;
        controller.path[1][2] = 1;
        controller.path[2][3] = 1;
        controller.path[2][5] = 1;
        controller.path[3][4] = 1;
        controller.path[5][4] = 1;
        controller.path[3][6] = 1;
        controller.path[5][6] = 1;
```

```
        controller.path[1][0] = 1;
        controller.path[3][0] = 1;
        controller.path[2][1] = 1;
        controller.path[3][2] = 1;
        controller.path[5][2] = 1;
        controller.path[4][5] = 1;
        controller.path[4][3] = 1;
        controller.path[6][3] = 1;
        controller.path[6][5] = 1;
    }

    public void getExample5() {
        controller.setVertexCount(8);
        controller.initTables();
        controller.vertexList.add(new Vertex( x: 200, y: 150));
        controller.vertexList.add(new Vertex( x: 300, y: 150));
        controller.vertexList.add(new Vertex( x: 300, y: 250));
        controller.vertexList.add(new Vertex( x: 200, y: 250));
        controller.vertexList.add(new Vertex( x: 120, y: 70));
        controller.vertexList.add(new Vertex( x: 380, y: 70));
        controller.vertexList.add(new Vertex( x: 380, y: 330));
        controller.vertexList.add(new Vertex( x: 120, y: 330));

        controller.path[0][1] = 1;
        controller.path[1][2] = 1;
        controller.path[2][3] = 1;
        controller.path[3][0] = 1;
        controller.path[0][4] = 1;
        controller.path[1][5] = 1;
        controller.path[2][6] = 1;
        controller.path[3][7] = 1;
        controller.path[4][7] = 1;
        controller.path[4][5] = 1;
        controller.path[5][6] = 1;
        controller.path[6][7] = 1;
        controller.path[1][0] = 1;
        controller.path[2][1] = 1;
        controller.path[3][2] = 1;
        controller.path[0][3] = 1;
        controller.path[4][0] = 1;
        controller.path[5][1] = 1;
        controller.path[6][2] = 1;
        controller.path[7][3] = 1;
        controller.path[7][4] = 1;
        controller.path[5][4] = 1;
        controller.path[6][5] = 1;
        controller.path[7][6] = 1;
    }

    public void getExample6() {
        controller.setVertexCount(10);
        controller.initTables();
        controller.vertexList.add(new Vertex( x: 50, y: 240));
        controller.vertexList.add(new Vertex( x: 250, y: 50));
        controller.vertexList.add(new Vertex( x: 450, y: 240));
        controller.vertexList.add(new Vertex( x: 380, y: 450));
        controller.vertexList.add(new Vertex( x: 120, y: 450));
        controller.vertexList.add(new Vertex( x: 160, y: 375));
        controller.vertexList.add(new Vertex( x: 120, y: 240));
        controller.vertexList.add(new Vertex( x: 250, y: 140));
        controller.vertexList.add(new Vertex( x: 380, y: 240));
        controller.vertexList.add(new Vertex( x: 340, y: 375));
```

```
    public void getExample6() {
        controller.setVertexCount(10);
        controller.initTables();
        controller.vertexList.add(new Vertex( x: 50, y: 240));
        controller.vertexList.add(new Vertex( x: 250, y: 50));
        controller.vertexList.add(new Vertex( x: 450, y: 240));
        controller.vertexList.add(new Vertex( x: 380, y: 450));
        controller.vertexList.add(new Vertex( x: 120, y: 450));
        controller.vertexList.add(new Vertex( x: 160, y: 375));
        controller.vertexList.add(new Vertex( x: 120, y: 240));
        controller.vertexList.add(new Vertex( x: 250, y: 140));
        controller.vertexList.add(new Vertex( x: 380, y: 240));
        controller.vertexList.add(new Vertex( x: 340, y: 375));

        controller.path[0][1] = 1;
        controller.path[0][4] = 1;
        controller.path[0][6] = 1;
        controller.path[1][2] = 1;
        controller.path[1][7] = 1;
        controller.path[2][3] = 1;
        controller.path[2][8] = 1;
        controller.path[3][4] = 1;
        controller.path[3][9] = 1;
        controller.path[4][5] = 1;
        controller.path[5][7] = 1;
        controller.path[5][8] = 1;
        controller.path[6][8] = 1;
        controller.path[6][9] = 1;
        controller.path[7][9] = 1;

        controller.path[1][0] = 1;
        controller.path[4][0] = 1;
        controller.path[6][0] = 1;
        controller.path[2][1] = 1;
        controller.path[7][1] = 1;
        controller.path[3][2] = 1;
        controller.path[8][2] = 1;
        controller.path[4][3] = 1;
        controller.path[9][3] = 1;
        controller.path[5][4] = 1;
        controller.path[7][5] = 1;
        controller.path[8][5] = 1;
        controller.path[8][6] = 1;
        controller.path[9][6] = 1;
        controller.path[9][7] = 1;
    }

    public void getExample7() {
        controller.setVertexCount(7);
        controller.initTables();

        controller.vertexList.add(new Vertex( x: 160, y: 160));
        controller.vertexList.add(new Vertex( x: 300, y: 100));
        controller.vertexList.add(new Vertex( x: 270, y: 200));
        controller.vertexList.add(new Vertex( x: 400, y: 400));

        controller.vertexList.add(new Vertex( x: 280, y: 370));
        controller.vertexList.add(new Vertex( x: 270, y: 290));
        controller.vertexList.add(new Vertex( x: 120, y: 280));
        controller.path[0][1] = 1;
        controller.path[0][6] = 1;
        controller.path[1][2] = 1;
        controller.path[2][3] = 1;
        controller.path[2][5] = 1;
        controller.path[3][4] = 1;
        controller.path[4][6] = 1;
        controller.path[5][6] = 1;
```

```
        controller.path[1][0] = 1;
        controller.path[6][0] = 1;
        controller.path[2][1] = 1;
        controller.path[3][2] = 1;
        controller.path[5][2] = 1;
        controller.path[4][3] = 1;
        controller.path[6][4] = 1;
        controller.path[6][5] = 1;

    }

    public void getExample8() {
        controller.setVertexCount(8);
        controller.initTables();
        controller.vertexList.add(new Vertex( x: 100, y: 400));
        controller.vertexList.add(new Vertex( x: 250, y: 400));
        controller.vertexList.add(new Vertex( x: 400, y: 400));
        controller.vertexList.add(new Vertex( x: 550, y: 400));
        controller.vertexList.add(new Vertex( x: 175, y: 250));
        controller.vertexList.add(new Vertex( x: 325, y: 250));
        controller.vertexList.add(new Vertex( x: 475, y: 250));
        controller.vertexList.add(new Vertex( x: 250, y: 100));

        controller.path[0][1] = 1;
        controller.path[0][4] = 1;
        controller.path[1][2] = 1;
        controller.path[1][4] = 1;
        controller.path[1][5] = 1;
        controller.path[2][3] = 1;
        controller.path[2][5] = 1;
        controller.path[4][5] = 1;
        controller.path[2][6] = 1;
        controller.path[3][6] = 1;
        controller.path[4][7] = 1;
        controller.path[5][7] = 1;
        controller.path[1][0] = 1;
        controller.path[4][0] = 1;
        controller.path[2][1] = 1;
        controller.path[4][1] = 1;
        controller.path[5][1] = 1;
        controller.path[3][2] = 1;
        controller.path[5][2] = 1;
        controller.path[5][4] = 1;
        controller.path[6][2] = 1;
        controller.path[6][3] = 1;
        controller.path[7][4] = 1;
        controller.path[7][5] = 1;
    }
}
```

Oskenované zadání práce

Zadání bakalářské práce

Autor:	**Ondřej Brekl**
Studium:	I1800668
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Hamiltonovské grafy**
Název bakalářské práce AJ:	Hamiltonian graphs

Cíl, metody, literatura, předpoklady:

Hamiltonovské grafy

Osnova práce

1.Historické souvislosti ve výzkumu oblasti Hamiltonských grafů.

2.Hamiltonovské grafy - Pojmový aparát, základní definice, elementy teorie.

3. NP úplné problémy, hlavní existující problémy a jejich převoditelnost. Existence hamiltonovské kružnice jako NP úplný problém.

4. Hamiltonovské grafy ve výuce.

5. Vlastní přístup k řešení problému malého rozsahu backtrackingem

- Popis algoritmu
- Programové řešení a jeho popis/dokumentace
- Testování, ukázka výstupu

6. Závěr

Kovář, Petr. Úvod do Teorie grafů

Wilson, J. Robin. A Brief History of Hamiltonian Graphs - Annals of Discrete Mathematics

Matthew Robinson, Pavel Vorobiev. Swing

Cay S. Horstmann. Core Java - volume 1 – Fundamentals

další literatura bude upřesněna

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	prof. RNDr. PhDr. Antonín Slabý, CSc.
Datum zadání závěrečné práce:	15.10.2021