Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Barvení grafu

2023

Do Minh Hieu

Vedoucí práce:
Mgr. Petr Osička, Ph.D.

Studijní program: Informatika,
Specializace: Programování a vývoj software

Bibliografické údaje

Autor:	Do Minh Hieu
Název práce:	Barvení grafu
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Informatika, Specializace: Programování a vývoj software
Vedoucí práce:	Mgr. Petr Osička, Ph.D.
Počet stran:	49
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Do Minh Hieu
Title:	Graph colouring
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Computer Science, Specialization: Programming and Software Development
Supervisor:	Mgr. Petr Osička, Ph.D.
Page count:	49
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Barvení grafu se používá pro plánování rozvrhu, přiřazení zdrojů, optimalizace sítí a mnoho dalšího. V rámci této bakalářské práce tuto disciplínu popisuji. Také se věnuji algoritmům řešící tenhle problém a na základě znalostí z textu je implementována knihovna. Naprogramované algoritmy využívám k experimentálním účelům.

Synopsis

Graph coloring is used for scheduling, resource allocation, network optimization and much more. Within this bachelor thesis I describe this discipline. I also discuss the algorithms solving this problem and a library is implemented based on the knowledge from the text. I use the programmed algorithms for experimental purposes.

Klíčová slova: barvení grafu; graf; greedy; DSatur; RLF; TabuCol; PartialCol; AntCol; HybridEA

Keywords: graph colouring; graph; greedy; DSatur; RLF; TabuCol; PartialCol; AntCol; HybridEA

Rád bych poděkoval Mgr. Petrovi Osičkovi, Ph.D za cenné rady a odborné vedení. Návrhy a doporučení datových struktur pro optimalizaci algoritmu mi velmi pomohly s implementací. Dále bych chtěl poděkovat Matěji Dostálovi za gramatické úpravy.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam tabulek

1 Úvod

Barvení grafu je poutavý a multidisciplinární obor, který již mnoho let přitahuje pozornost matematiků, informatiků a výzkumníků. Koncept barvení grafů je založen na úloze přiřazení barev vrcholům grafu za určitých omezení. I když se zadání problému může zdát jednoduché, jeho složitost se ukáže, jakmile se ponoříme hlouběji do jeho různých aspektů. Tato disciplína představuje řadu výzev a příležitostí ke zkoumání.

Cílem práce bylo nastudovat problematiku, implementovat algoritmy pro barvení grafu a udělat experimentální porovnání jednotlivých algoritmů. Po nastudování této problematiky jsem naprogramoval knihovnu, která obsahuje algoritmy pro barvení grafu. Knihovna je vytvořena v jazyku C a po jejím importování může uživatel používat algoritmy ve svém programu. Knihovna obsahuje 3 konstruktivní a 4 pokročilé algoritmy.

Práce je rozdělena na teoretickou a experimentální část. Teoretická část začíná 2. kapitolou, která přibližuje problém barvení grafů pro pochopení budoucích kapitol. 3 kapitola se věnuje jednotlivým algoritmům, popis jejich činností a je přiložen pseudokód, který čtenář může využít k implementaci. Dále v této kapitole se pojednává o použitých datových strukturách jednotlivých algoritmů. Nakonec je 4 kapitola věnována experimentálnímu porovnávání jednotlivých algoritmů mezi sebou.

2 Barvení grafu

Tato kapitola čerpá z [1], [2].

Barvení grafů je problém, který spočívá v přiřazení barev vrcholům grafu tak, aby žádné dva sousední vrcholy neměly stejnou barvu. Cílem je použít co nejmenší počet barev k obarvení celého grafu. Tento problém má širokou škálu aplikací v informatice, operačním výzkumu a dalších oborech.

2.1 Využití barvení grafu

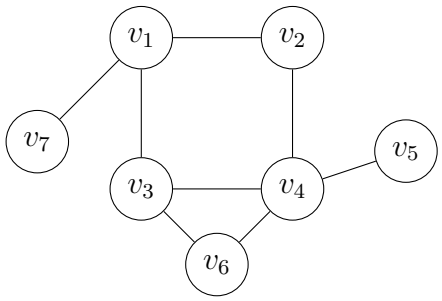
Problém barvení grafu má mnoho praktických aplikací, jako je tvorba jízdních řádů, přidělení registru překladače, navrhování univerzitních rozvrhů, barvení map, sudoku nebo přiřazení frekvence.

Zjednodušeně řečeno si můžeme představit vrcholy grafu jako množinu položek, které je potřeba rozdělit do skupin. Například u tvorby rozvrhu máme k dispozici množinu událostí jako jsou přednášky, zkoušky, semináře a odpovídající množinu dostupných časových úseků. Naším úkolem je rozdělit události do časových úseků tak, aby se dvě události nekřížily. Jedním ze způsobů, jak přistupovat k problému rozvrhování, je nahlížet na něj jako na problém barvení grafu. To lze provést tak, že každou událost reprezentujeme jako vrchol v grafu a mezi každou dvojici vrcholů, které se nemohou vyskytnout současně kvůli konfliktům v plánování, přidáme hrany. Cílem je najít vhodné barevné přiřazení úloh v rozvrhu tak, aby každému dostupnému časovému úseku odpovídala určitá barva a minimalizovat počet barev potřebných pro toto přiřazení tak, aby nepřesáhl počet dostupných časových úseků.

2.2 Teorie grafu

Definice 1 (Neorientovaný graf)

Neorientovaný graf je tvořen neprázdnou množinou vrcholů V a množninou hran E, značeno $G = (V, E)$, kde $E \subseteq \{\{u, v\} | \, u, v \in V, u \neq v\}$.



Obrázek 1: Neorientovaný graf

Vrcholy u a v jsou *sousední*, pokud $\{u, v\} \in E$. Vrcholy u a v nazýváme *koncovými* vrcholy hrany $e = \{u, v\}$, nebo také *indcidentní* s hranou e.

Sousedství vrcholu v, psáno $\Gamma_G(v)$, je množina vrcholů sousedících s ním v grafu G. To znamená, že $\Gamma_G(v) = \{u \in V : \{v, u\} \in E\}$. Sousedství vrcholu v_4 na obrázku 1 je $\Gamma(v_4) = \{v_2, v_3, v_5, v_6\}$.

Stupeň vrcholu v je kardinalita množiny jeho sousedů, teda $|\Gamma_G(v)|$, obvykle se píše *deg(v)*. Stupeň předtím zmíněného vrcholu v_4 je $deg(v_4) = 4$.

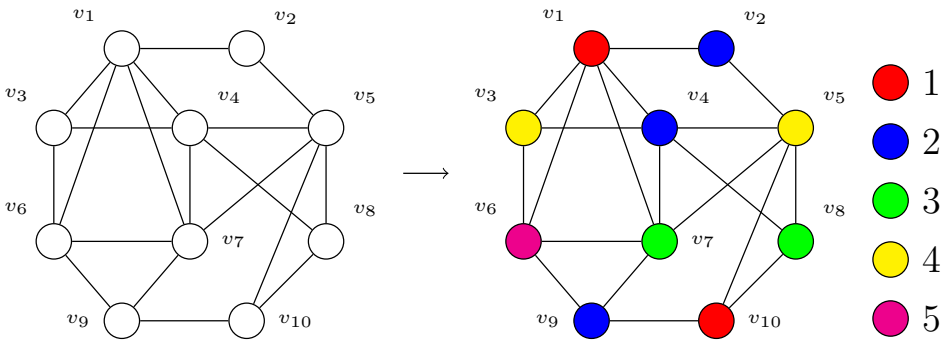
Neorientovaný graf (V_1, E_1) je *podgrafem* grafu (V_2, E_2), právě když $V_1 \subseteq V_2$ a $E_1 \subseteq E_2$. *Podgraf* (V_1, E_1) grafu (V_2, E_2) se nazývá *indukovaný*, právě když E_1 obsahuje každou hranu z E_2, jejíž oba koncové vrcholy patří do V_1. Nechť $W \subseteq V$, pak $G - W$ je podgraf získaný vymazáním vrcholů ve W z G spolu s hranami, které s nimi souvisejí.

Klika je podmnožina vrcholů $C \subseteq V$, které spolu sousedí.

2.3 Popis problému

Definice 2 (Barvení grafu)

Mějme graf G = (V, E), kde V je množina vrcholů. Barvení grafu je zobrazení $c : V \to \mathbb{N}$, které každému vrcholu přiřazuje barvu z množiny přirozených čísel $\mathbb{N}$ tak, že pro libovolné dva sousední vrcholy u, v platí $c(u) \neq c(v)$ a $c(v) \in \{1, 2, \ldots, max(c(v))\}$, kde max(c(v)) počet barev, které bylo použito k obarvení grafů.



Obrázek 2: Neobarvený graf (vlevo) a obarvený graf (vpravo)

Na obrázku 1 máme graf s množinou vrcholů V, která obsahuje $n = 10$ vrcholů a množinu hran E, obsahující $m = 21$ hran a je obarven 5 barvami. Použitím našeho zápisu obarvení lze toto řešení zapsat následovně:

$$c(v_1) = 3, c(v_2) = 2, c(v_3) = 4, c(v_4) = 2, c(v_5) = 4,$$
$$c(v_6) = 5, c(v_7) = 1, c(v_8) = 1, c(v_9) = 2, c(v_{10}) = 3.$$

Definice 3 (Úloha barvení grafu)

Vstup: $G = (V, E)$

Přípustná řešení: Obarvený graf, kde žádné dva sousední vrcholy nemají stejnou barvu.

Cena: max(c(v)), tedy počet barev použitých pro obarvení všech vrcholů.
Cíl: Snažíme se minimalizovat cenu, což znamená použít co nejméně barev.

2.3.1 Vlasnosti obarveného grafu

Obarvený graf na obrázku 1 je *úplný*, protože je všem vrcholům $v \in V$ přiřazena barva $c(v) \in \{1, \ldots, max(c(v))\}$, jinak se obarvení považuje za *částečné*. Částečně obarvený graf má nějaký vrchol, který nemá přiřazenou barvu, tedy $\exists v \in V : c(v) = NULL$.

Konflikt je situace, kdy je dvojici sousedních vrcholů $u, v \in V$ přiřazena stejná barva (tj. $\{u, v\} \in E \ a \ c(v) = c(u)$).

Korektní obarvený graf neobsahuje *konflikty*, pokud je má, tak je považován za *nekorektní*.

Pokud je graf *úplný* a *korektní*, pak ho nazveme *přípustným*. Obarvený graf na obrázku 2 je *přípustný*.

Chromatické číslo grafu G, označené $\chi(G)$, je nejmenší počet barev potřebných pro proveditelné obarvení grafu G. Přípustné obarvení grafu G s použitím přesně $\chi(G)$ barev se považuje za *optimální*. Chromatické číslo grafu na obrázku 2 je 5, protože není možné jej obarvit s méně barvami, aby graf zůstal *přípustný*.

Barevná třída i je množina obsahující všechny vrcholy grafu, kterým je v řešení přiřazena určitá barva. To znamená, že při dané konkrétní barvě $i \in \{1, \ldots, max(c(v))\}$ je třída barev definována jako množina $\{v \in V : c(v) = i\}$.

Nezávislá množina je podmnožina vrcholů $I \subseteq V$, kde žádné dva vrcholy spolu nesousedí. To znamená, že $\forall u, v \in I \ \{u, v\} \notin E$. Barevná třída je nezávislá množina. Proto je užitečné nahlížet na barvení grafu jako na rozklad na nezávislé množiny, kde řešení S je reprezentován množinou barevných tříd $S = \{S_1, \ldots, S_k\}$. Pro připomenutí, aby S byl *přípustný*, je nutné splnit následující omezení:

$$\bigcup_{i=1}^{k} S_i = V, \tag{1}$$

$$S_i \cap S_j = \emptyset \ (1 \leq i \neq j \leq k), \tag{2}$$

$$\forall u, v \in S_i, \{u, v\} \notin E \ (1 \leq i \leq k) \tag{3}$$

Omezení 1 uvádí, že prvek náleží do S pouze pokud patří alespoň do jedné z množin $S_1, \ldots S_k$. Omezení 2 praví, že žádné dvě množiny nemohou mít společné prvky. Poslední omezení 3 stanovuje, že v grafu nemohou sousedit žádné dva vrcholy ze stejné barvené množiny S_i.

2.4 Složitost problému

Úlohu barvení grafu jsme si uvedli jako optimalizační problém, kde se ptáme, jaký je nejmenší počet barev k obarvení grafu G. V oblasti vyčíslitelnosti a složitosti jsou problémy obvykle chápany jako rozhodovací problémy vyžadující odpovědi

„ano“ nebo „ne“. Problém barvení grafu lze snadno formulovat jako rozhodovací problém s otázkou „Existuje providitelne obarvení grafu G, které používá alespoň k barev?“ Rozhodovací problémy, pro které existují algoritmy, jejichž rychlost růstu lze vyjádřit pomocí polynomu, patří do třídy P. Další třídou rozhodovacích problémů je NP, což znamená nedeterministický polynomiální čas. O problému řekneme, že patří do třídy NP, pokud existuje polynomický algoritmus takový, že pro instanci x je odpověd "ano"právě když existuje polynomicky velký certifikát c takový, že tento algoritmus přijímá x a c. Rozhodovací varianta problému barvení grafu patří do NP, protože pro dvojici tvořenou grafem a hodnotou k, existuje barvení s méně než k barvami, právě když je odpověď "ano". Toto barvení můžeme použít jako certifikát, zkontrolovat, že má nejvýše k barev a že je úplné a korektní lze v polynomickém čase.

Je zřejmé, že každý problém patřící do P, patří také do NP, protože pro každý problém v P můžeme k jeho řešení (a tedy ověření) jednoduše použít jeho dostupný algoritmus s polynomiální časovou složitostí. To znamená, že $P \subseteq NP$. Navzdory desítkám let výzkumu však není známo, zda ve skutečnosti P = NP. Obecně uznávanou doměnkou je $P \neq NP$. Tím se dostáváme k třídě NP-úplných problémů. Problém Q nazveme NP-úplným, pokud je NP-těžký a náleží do třídy NP. Problém Q nazveme NP-těžkým, pokud každý problém ve třídě NP lze na problém Q polynomiálně převést, tedy NP-těžké problémy nemusí být ve třídě NP. Má smysl uvažovat o NP-úplných a NP-těžkých problémech, pokud $P \neq NP$.

Za takového předpokladu není možné získat v polynomiálním čase optimální řešení pro všechny vstupy. Proto se uchylujeme k aproximačním algoritmům a heuristickým metodám, které pracují v polynomiálním čase. Cílem těchto technik je najít přípustné obarvení, které se blíží optimálnímu, nikoli zaručit optimální řešení. Tyto algoritmy vytvoří *přípustný* obarvený graf s k barvami G, který nemusí být optimální. V takovém případě můžeme znovu aplikovat algoritmus, který by nám potenciálně mohl poskytnout lepší řešení s použitím méně než k barev. I kdyby náhodou vrátily optimální řešení, nemusíme být schopni toto řešení ověřit dostatečně rychle (museli bychom vyzkoušet všechny obarvení). Dále účinnost aproximačních algoritmů a heuristik pro barvení grafů může záviset na konkrétních vlastnostech uvažovaného grafu. Algoritmus může dobře fungovat na některé typy grafu, ale špatně zas na jiné.

Příklady rozhodovacích problémů souvisejících s barvením grafu:

- Problém nalezení chromatického čísla: Je dán graf G a nějaké celé číslo k, je chromatické číslo grafu G $\chi(G) = k$?
- Problém maximální nezávislé množiny: Je dán graf G a nějaké celé číslo k, obsahuje v grafu G nezávislá množina obsahující k vrcholů?

3 Barvící algoritmy

Celá kapitola a všechny algoritmy čerpá z [1].

V této kapitole se budeme nejprve zabývat třemi rychlými konstruktivními metodami, které pracují tak, že postupně přiřazují barvy každému vrcholu pomocí pravidel a jejich cílem je udržet celkový počet barev co nejmenší. Zbylé čtyři algoritmy jsou vysoce výkonné algoritmy využívající různých metaheuristik a heuristik. Tyto pokročilejší algoritmy využívají kontruktivních algoritmů k vyprodukování počátečního řešení, které se pak zlepšuje.

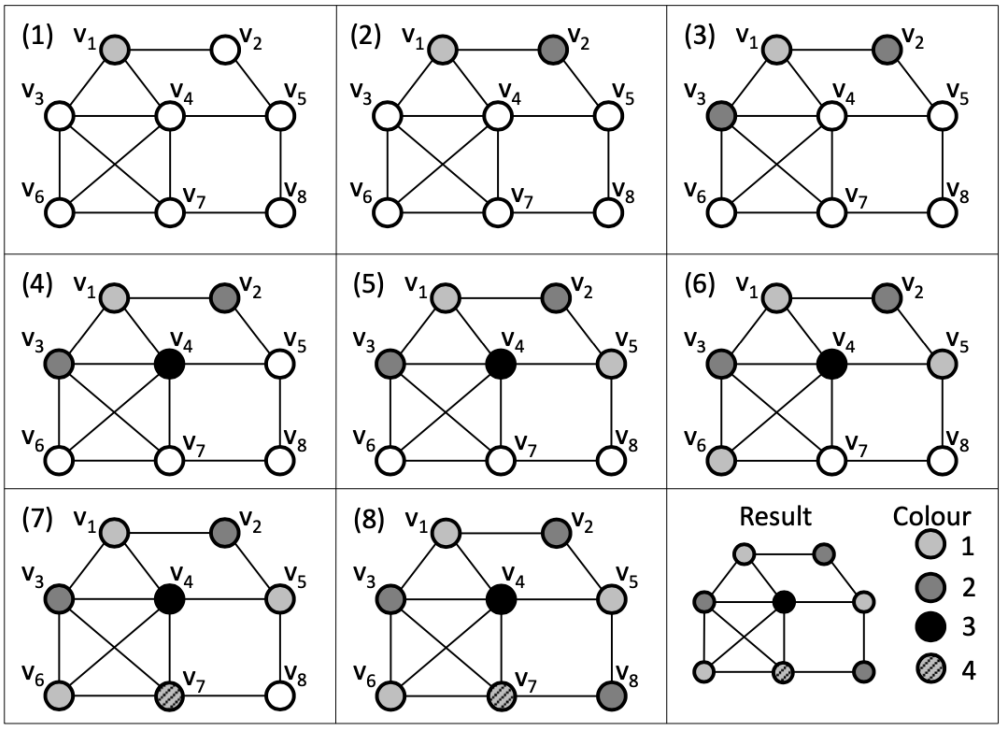
3.1 Greedy algoritmus

Greedy algoritmus je jeden z nejjednodušších a nejzákladnějších heuristických algoritmů pro barvení grafů. Algoritmus pracuje tak, že zpracovává vrcholy v daném uspořádání a každému znich přiřazuje první dostupnou barvu.

Algorithm 1 Greedy ($G = (V, E)$)

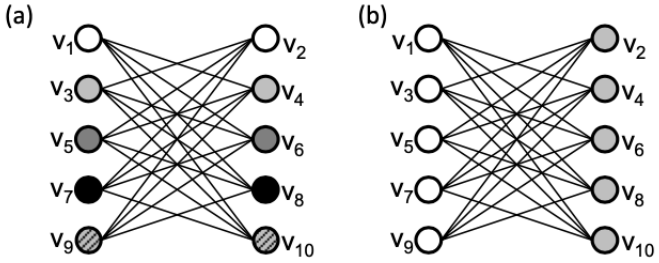
```
1:  S ← ∅
2:  Nechť π je uspořádání vrcholů v ∈ V.
3:  for i ← 1 to |π| do
4:      for j ← 1 to |S| do
5:          if (S_j ∪ v_i) je nezávislá množina then
6:              S_j ← S_j ∪ v_i
7:              break
8:          else
9:              j ← j + 1
10:         end if
11:     end for
12:     if j > |S| then
13:         S_j ← {v_i}
14:         S ← S ∪ S_j
15:     end if
16: end for
17: return množina tříd barev definovaná pomocí S
```

Algoritmus 1 na vstupu přijímá graf G. Na začátku inicializuje prázdné řešení $S = \emptyset$ a libovolné uspořádání vrcholu v. Pro každý vrchol v_i v daném uspořádání se pokusí najít barevnou třídu $S_j \in S$, do které ji lze vložit. Pokud lze vrchol v_i přidat do barevné třídy S_j, aniž by způsobilo konflikt, tak se do ní vrchol přidá a proces pokračuje k dalšímu vrcholu v_{i+1}. Pokud ne, vytvoří se nová barevná třída S_{j+1} a je vrchol do ní přidán. Tento proces zajišťuje, že každému vrcholu je přiřazena barva. Algoritmus se snaží minimalizovat počet použitých barev tím, že každý vrchol pokud možno přiřadí do existující barevné třídy.



Obrázek 3: Příklad použití Greedy

Časová složitost Greedy v nejhorším případě je $O(n^2)$. Nastává v případě, že je graf úplný (každé dva vrcholy jsou spojené hranou). Každý vrchol by byl v konfliktu s vrcholy, které už byly přiřazeny. Musel by pokaždé procházet všechny barevné třídy a zjišťovat nezávislost. V praxi algoritmus poskytuje přípustná řešení poměrně rychle, avšak tato řešení mohou být poměrně špatná z hlediska počtu barev, které algoritmus potřebuje v porovnání s chromatickým číslem. Na obrázku 4 graf a) má 5 barevných tříd a b) jenom 2 třídy. Obarvení grafů se lišilo jenom v úspořádání vrcholů, kde graf a) má $\pi = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$ a graf b) má $\pi = \{v_1, v_3, v_5, v_7, v_9, v_2, v_4, v_6, v_8, v_{10}\}$. Je vidět, že uspořádání vrcholů je velmi důležité.



Obrázek 4: Dva obarvené grafy použitím Greedy algoritmu

Tato část čerpá z [3].

3.1.1 Implementace Greedy algoritmu

Graf je reprezentován jako matice sousednosti. V matici A na pozici A_{ij} je 1, právě když vrchol v_i a v_j spolu sousedí, jinak $A_{ij} = 0$. Tímto způsobem lze rychle zjistit, jestli spolu vrcholy u a v sousedí. Tato reprezentace grafu se používá i u ostatních algoritmů.

```
typedef struct {
    int num_vertices;    // počet vrcholu
    int** adj_matrix;    // dvourozměrné pole |V|x|V|
} Graph;
```

Jednotlivé barevné třídy využívají strukturu seznamu sousedů. Každý prvek seznamu odpovídá spojovému seznamu obsahující všechny vrcholy pro danou barevnou třídu.

```
typedef struct node{
    int value;           // index vrcholu
    struct node* next;   // pointer na další vrchol
    int degree;          // stupeň vrcholu
} node;

typedef struct {
    int num_vertices;    // počet barevných tříd
    node** adj_list;     // spojový seznam vrcholů
} adjacency_list;
```

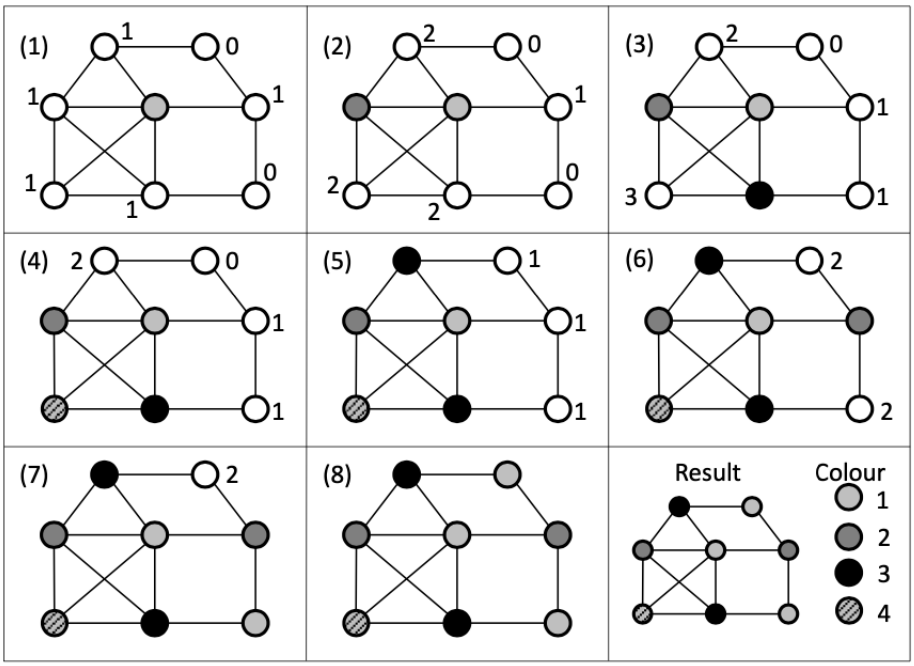
3.2 DSatur algoritmus

Další konstruktivní algoritmus je DSatur, celým názvem „degree of saturation“. Chová se velmi podobně jako Greedy, avšak rozdíl mezi oběma algoritmy spočívá ve způsobu, jakým jsou uspořádání vrcholů generována. V případě Greedy je pořadí určeno předtím, než dojde k jakémukoli barvení. Oproti tomu DSatur vybírá, který vrchol se obarví příště na základě vlastnosti aktuálního částečného obarvého grafu. Tato volba je založena na stupni *sytosti* vrcholů, který je definován jako počet různých barev přiřazených sousedním vrcholům, označený *sat(v)*. To znamená, že $sat(v) = |\{c(u) : u \in \Gamma(v) \wedge c(u) \neq NULL\}|$.

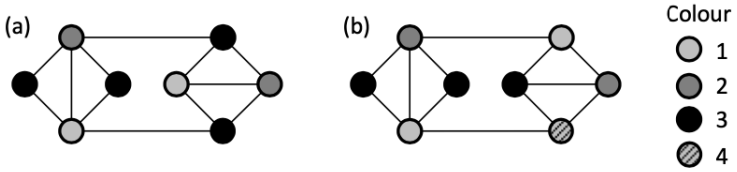
Hlavní rozdíl mezi Greedy a DSatur je vidět na řádcích 2, 3 a 15 pseudokódu 2. Na začátku inicializuje množinu X, která reprezentuje vrcholy, kterým v současné době není přiřazena barva. Na řádku 3 se vybírá vrchol v z X, který má nejvyšší stupeň *nasycení*. Pokud existuje více než jeden vrchol s nejvyšším stupněm *nasycení*, tak se vybere vrchol s nejvyšším stupněm. Myšlenka heuristiky maximálního stupně nasycení spočívá v tom, že dává přednost vrcholům, které mají v současné době k dispozici nejméně barevných možností. Proto se algoritmus nejprve zabývá těmito vrcholy a méně „omezené“ vrcholy může obarvit později.

Algorithm 2 DSatur ($G = (V, E)$)

1: $S \leftarrow \emptyset$, $X \leftarrow V$
2: **while** $X \neq \emptyset$ **do**
3: vybrat $v \in X$
4: **for** $j \leftarrow 1$ to $|S|$ **do**
5: **if** $S_j \cup v$ je nezávislá množina **then**
6: $S_j \leftarrow S_j \cup \{v\}$
7: **break**
8: **else**
9: $j \leftarrow j + 1$
10: **end if**
11: **if** $j > |S|$ **then**
12: $S_j \leftarrow \{v\}$
13: $S \leftarrow S \cup S_j$
14: **end if**
15: $X \leftarrow X - \{v\}$
16: **end for**
17: **end while**
18: **return** S



Obrázek 5: Příklad použití DSatur



Obrázek 6: a) optimální tříbarevné obarvení, b) obarvený graf DSaturnem

Časová složitost v nejhorším případě je stejná jako u Greedy a činí $O(n^2)$. DSatur je přesný pro řadu elementárních topologií grafu, například pro bipartitní a cyklické grafy. Na obrázku 6 máme graf, který je tříbarevný, ale DSatur ho obarvil 4 barvami. Jedná se o nejmenší graf, kde se tato neoptimálnost vyskytuje. [4]

3.2.1 Implementace DSatur

DSatur algoritmus vybírá vrcholy podle *sytosti*, takže oproti normálnímu vrcholu má vlastnost *sytost* navíc.

```
typedef struct {
    int value;          // index vrcholu
    int degree;         // stupeň vrcholu
    int saturation;     // sytost vrcholu
} dsatur_node;
```

Důležitou funkcí je funkce na vybírání vrcholů (*choose_vertex*), která se volá iterativně, aby vybrala další vrchol s nejvyšším stupněm sytosti. Na začátku *qsort* seřadí pole X (pole dsatur vrcholů) sestupně na základě stupně nasycení a stupně vrcholů. Poté se iteruje nad X, aby našla vrchol s nejvyšším stupněm nasycení; vrchol bývá většinou na začátku. Pokud existuje více vrcholů s nejvyšším stupněm nasycení, cyklus porovná jejich stupně. Vybraný vrchol je poté z pole odstraněn nastavením hodnoty nasycení a stupně na -1.

Funkce *update_saturation* iteruje přes sousedy vrcholu a aktualizuje jejich sytost. Funkce používá pomocné pole ke sledování barev používaných sousedy a pomocné pole k efektivnímu přístupu ke stupňům sytosti v poli X a jejich aktualizaci.

3.3 RLF algoritmus (Recursive largest first)

Poslední konstruktivní algoritmus se řídí poněkud jinou strategií. Funguje tak, že obarvuje graf po jedné barvě. V každém kroku algoritmus identifikuje nezávislou množinu vrcholů pomocí heuristiky a přiřadí jim stejnou barvu. Tato nezávislá množina je poté z grafu odstraněna a proces se opakuje na menším podgrafu, dokud nejsou obarveny všechny vrcholy.

V každé vnější smyčce je sestavena i-tá barevná třída S_i. Na začátku se inicializuje množina X obsahující neobarvené vrcholy, které lze aktuálně přidat

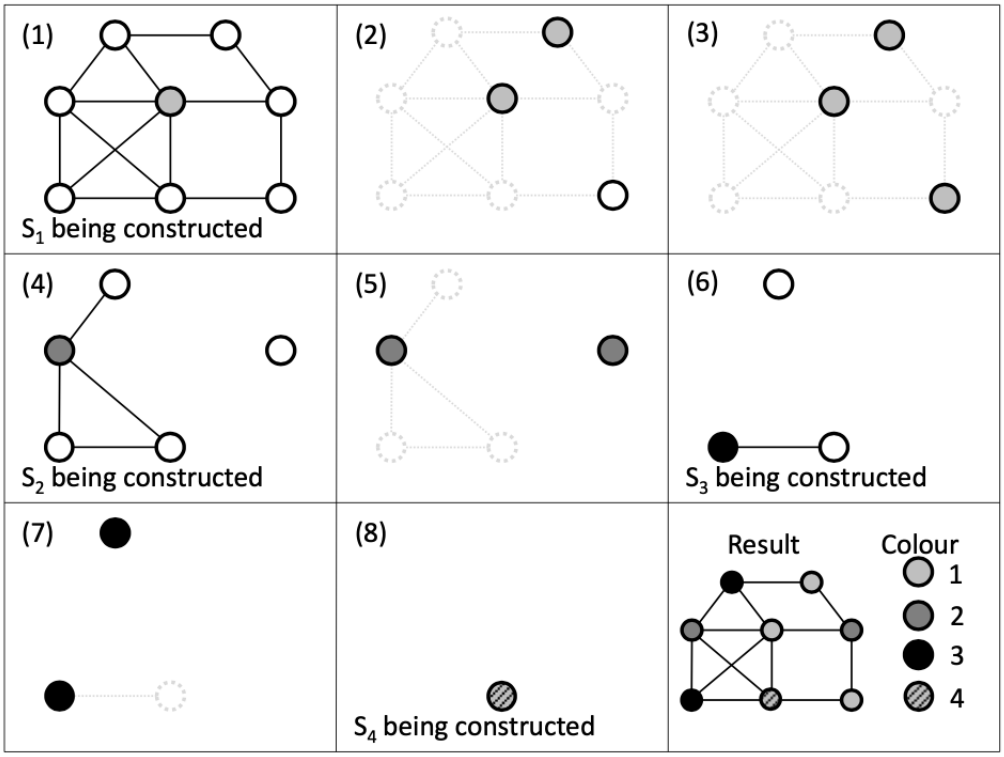
do S_i, aniž by došlo ke konfliktu a Y, která obsahuje neobarvené vrcholy, které nelze do S_i přidat. Od 5. až do 9. řádku v pseudokódu 3 se vytváří barevná třída S_i. Na začátku se vybere vrchol v z X a přidá se do S_i. Poté jsou všechny vrcholy sousedící s v v podgrafu indukovaném X převedeny do Y. Nakonec jsou v a jeho sousedé odstraněni z X, protože nyní nejsou považováni za kandidáty na zařazení do barevné třídy S_i.

Heuristika pro výběr vrcholu na řádku 6 probíhá tak, že se upřednostňují „omezené“ vrcholy. Jako první vrchol se vybere vrchol s nejvyšším stupněm v podgrafu indukovaném X a přidá se do aktuální barevné třídy S_i. Zbývající vrcholy, které se přidají do S_i, se vybírají z množiny X. Postupně jsou vybrány vrcholy, které mají nejvyšší stupěň v podgrafu indukovaném vrcholy z $Y \cup \{v\}$. To znamená, že je vybrán vrchol z X, který sousedí s největším počtem vrcholů z Y. Vybraný vrchol a jeho sousedé jsou odstraněni z X. Jakmile je množina X prázná, do barevné třídy S_i se už další prvek nepřidá a začne se vytvářet další barevná třída S_{i+1}.

Algoritmus RLF má v nejhorším případě časovou složitost $O(n^3)$, což je více, než časová složitost algoritmů Greedy a DSatur, která je $O(n^2)$. Přesto je algoritmus stále polynomiální, což z něj činí efektivní volbu pro mnoho instancí problému barvení grafů.

Algorithm 3 RLF $(G = (V, E))$

```
1:  S ← ∅, X ← V, Y ← ∅, i ← 0
2:  while X ≠ ∅ do
3:      i ← i + 1
4:      S_i ← ∅
5:      while X ≠ ∅ do
6:          vybrat v ∈ X
7:          S_i ← S_i ∪ {v}
8:          Y ← Y ∪ Γ_X(v)
9:          X ← X − (Y ∪ {v})
10:     end while
11:     S ← S ∪ {S_i}
12:     X ← Y
13:     Y ← ∅
14: end while
15: return S
```



Obrázek 7: Příklad použití algoritmu RLF. Zde jsou tečkovanými vrcholy označeny vrcholy aktuálně přiřazené k množině Y. Vrcholy s plnými čarami a bez barvy označují členy množiny X.

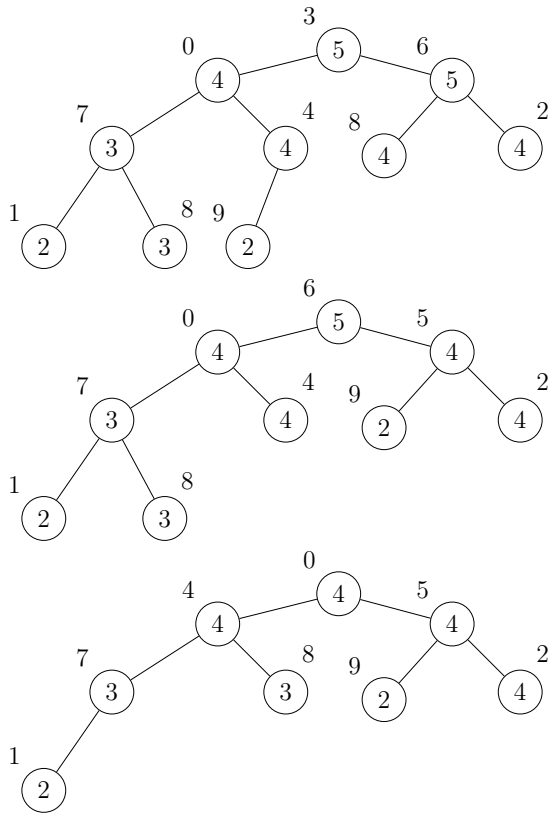
Tato část čerpá z [5].

3.3.1 Implementace RLF

```
typedef struct {
    node* items;        // pole vrcholů
    int size;           // počet vrcholů
    int capacity;       // velikost fronty
    int* indices;       // odkaz na vrchol
} priority_queue;
```

RLF používá datovou strukturu prioritní fronty, aby se nemuselo procházet celé pole při hledání vrcholu s nejvyšším stupněm. Prioritní fronta je abstraktní datový typ podobný běžné datové struktuře fronty nebo zásobníku. Každý prvek prioritní fronty má přiřazenou prioritu. V prioritní frontě jsou prvky s vysokou prioritou vybrány před prvky s nízkou prioritou. Prioritní frontu je implementována pomoci binární haldy. Jedná se o vyvážený binární strom, kde každý vrchol je větší nebo roven všem svým potomkům.

V implementaci prioritní fronty je tedy pole vrcholů, počet vrcholů, kapacita fronty a pole indexů pro přístup k prvkům v prioritní frontě v čase O(1). V ta-



Obrázek 8: Prioritní fronta, kde hodnota ve vrcholech je stupeň (degree) a hodnota mimo vrchol je jeho index (value)

bulce 1 je *value* hodnota vrcholu, *degree* je jeho stupeň a *indices* je index vrcholu v prioritní frontě.

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
indices	1	7	6	0	4	5	2	3	8	9
value	3	0	6	7	4	5	2	1	8	9
degree	5	4	5	3	4	4	4	2	3	2

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
indices	1	7	6	9	4	2	0	3	8	5
value	6	0	5	7	4	9	2	1	8	3
degree	5	4	4	3	4	2	4	2	3	5

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
indices	0	7	6	9	1	2	8	3	4	5
value	0	4	5	7	8	9	2	1	6	3
degree	4	4	4	3	3	2	4	2	5	5

Tabulka 1: Příklad mazání prvku v prioritní frontě

Binární haldou na obrázku 8 je znázorněno mazání prvku s největší sytostí. Při vybírání vrcholu se vždy vybere vrchol, který je na začátku a při smazání se dá na konec. Díky poli *indices* vždy víme, kde vrchol s jakou hodnoutou se nachází. V tabulce 1, která znázorňuje datovou strukturu prioritní fronty, vidíme, že pokud chceme najít vrchol s hodnotou 3, tak v poli indices na pozici 3 najdeme hodnotu 0, která nás odkáže na pozici vrcholu v poli *items*.

3.4 TabuCol algoritmus

TabuCol je založen na metaheuristice *tabu search*, která využívá heuristických metod *lokálního prohledávání*. Metoda lokálního prohledávání začíná u počátečního řešení a poté se opakovaně přesouvá od aktuálního řešení k „sousedním" řešením, aby se pokusilo najít lepší řešení. Přechod od jednoho řešení k dalšímu se nazývá *tah*. Sousedství kandidátního řešení je množina všech kandidátních řešení, která lze nalézt pomoci tahu.

Tabu search na začátku vytvoří kandidátní řešení použitím modifikované verze algoritmu Greedy. Upravený Greedy má stanovený pevný počet k barev, přičemž každý vrchol je přiřazen k barvě podle heuristik Greedy algoritmu. Pokud se tedy vyskytnou vrcholy, které nelze přiřadit k barevné třídě z k-barev, aniž by došlo ke konfliktu, přiřadí se jim náhodně jedna z existujících k barev. Aby řešení nebylo pokaždé stejné, tak je uspořádání vrcholů vždy jiné.

Po přiřazení vznikne obarvený graf s k barvami. Takto obarvený graf je s nejvyšší pravděpodobností nekorektní. Kvalitu řešení zjistíme podle hodnoty *objektové funkce f*, která v tomto případě je počet konfliktů.

Máme-li kandidátní řešení $S = \{S_1, \ldots, S_k\}$, pak tah je proveden tak, že se vybere vrchol $v \in S_i$, který se aktuálně nachází v konfliktu a přiřadí se mu nová barevná třída.

Aby se zabránilo cyklení a povzbuzovalo algoritmus k objevování nových řešení, využívá tabu vyhledávání strukturu zvanou *tabu seznam*, která sleduje dříve navštívená řešení a zakazuje algoritmu se k nim po určitou dobu vracet; takovým prvkům říkáme *tabu*. Tabu seznam je matice $T_{n \times k}$. Pokud je přesunut vrchol v z S_i do S_j, pak se prvek T_{vi} nastaví na $l + t$, kde l je hodnota aktuální iterace a t je kladné celé číslo, které se nazývá *tabu tenure* a je definováno jako hodnota $t = 0.6 * f + r$, kde r je číslo od 0 do 9 včetně[1]. To znamená, že vrchol v nemůže být přesunut zpět do S_i, dokud neproběhne alespoň $l + t$ iterací. Kromě toho TabuCol používá také *kritérium aspirace*, které umožňuje provádět tabu tahy, pokud zlepšují dosud nalezené nejlepší řešení.

Optimalizace kandidátního řešení probíhá ve *while* cyklu na řádcích 4 až 24, který probíhá dokud řešení neobsahuje konflikty anebo dokud nedosáhne maximálního počtu iterací. Proměná *minKonfliktů* značí dosud nalezené řešení s minimálním počtem konfliktů nebo jen počet konfliktů. Na 6. řádku začíná cyklus, který prochází všemi vrcholy, které se nachazí v konfliktu a postupně vrcholy přiřazuje do nové barevné třídy a zjišťuje počet konfliktů, které vzniknou tímto

[1] Podle Galinier a Hao v [6] tato konkrétní nastavení poskytuje dobré výsledky.

Algorithm 4 TABUCOL $(G = (V, E), maxIterations, k)$

```
1:  S ← Greedy(G, k)
2:  tabu_ij ← 0 ∀i, j ∈ V
3:  iter ← 1
4:  while iter < maxIterations do
5:      minKonfliktů ← ∞
6:      for vrcholy ktery jsou v konfliktu do
7:          for barva to k do
8:              numKonfliktů ← početKonfliktů(S, vrchol, barva)
9:              if numKonflikty < minKonfliktů || !tabu[barva][vrchol] then
10:                 bestS ← tah(S, vrchol, barva)
11:                 minKonflikt ← numKonflikty
12:             end if
13:         end for
14:     end for
15:     numKonflikty ← minKonflikty
16:     S ← bestS
17:     if numKonflikty = 0 then
18:         break
19:     end if
20:     if S = NULL then
21:         vyber náhodný tah
22:     end if
23:     aktualizuj tabu seznam
24:     iter ← iter + 1
25: end while
26: return S
```

tahem. Takto se najde řešení obsahující nejmenší počet konfliktů. V případě, že žádný ze sousedních řešení nevylepšuje stávající řešení, tak je vrchol a nová barevná třída vyberá náhodně. Na řádku 10 je proměnná *bestS* nové řešení, které vznikne tahem.

Tato část čerpá z [7] a [8].

3.4.1 Implementace Tabucol

Pro urychlení výpočtu konfliktů se používá pomocná matice C (*adjacent_vertice_color*), kde prvek C_{vj} označuje počet vrcholů v barevné třídě S_j, které sousedí s vrcholem v. Na začátku se vypočítají zjístí hodnoty všech prvků v C. Po provedení tahu vznikne nové řešení a je potřeba aktualizovat C, ale to ovlivní jenom určité prvky. Hodnota objektové funkce nového řešení S' se vypočítá jako:

$$f(S') = f(S) + C_{vj} + C_{vi}. \tag{4}$$

Pro znázornení C, vznikla matice na tabulce 2 podle grafu 2.

vrcholy:	1	2	3	4	5	6	7	8	9	10
S_1	0	1	1	1	0	1	1	1	1	0
S_2	2	0	1	0	2	1	2	1	0	1
S_3	1	0	0	2	2	1	0	0	1	1
S_4	1	1	0	2	0	1	1	1	0	0
S_5	1	0	1	0	0	0	1	0	1	0

Tabulka 2: Matice C podle grafu 2

Díky matici je umožňeno rychle spočítat počet konfliktů v grafu, a podle toho rozhodnout který tah je nejvýhodnější. Vrchol v se nachází v nějaké třídě S_i, pokud na pozici $C_{vi} > 0$, tak vrchol v se nachází v konfliktu a hodnota C_{vi} určuje s kolika vrcholy je v konfliktu. Například pro vrchol 2, který je v barevné třídě S_2, vidíme, že $C_{2,2}$ je hodnota prvku 0, protože sousední prvky nejsou obarveny stejnou barvou.

index:	1	2	3	4	5	6	7	8	9	10
S_1	0	1	1	1	0	1	1	1	1	0
S_2	1	0	0	0	1	1	1	0	0	1
S_3	1	0	0	2	2	1	0	0	1	1
S_4	1	1	0	2	0	1	1	1	0	0
S_5	2	0	2	0	1	0	2	1	1	0

Tabulka 3: Matice C po provedení tahu

Matice 3 je po přesunu vrcholu 4 ze třídy S_2 do S_5. Všechni sousedé vybraného vrcholu jsou aktualizovány. Na řádku původní třídy jsou sousedé dekrementováni o 1. Na řádku nové třídy, do které je vrchol přesunut, jsou sousední prvky inkrementováni. Červeně obarvené prvky jsou sousední prvky, kterým byla dekrementována hodnota a zelené políčka byla naopak inkrementována hodnota o 1.

3.5 PartialCol algoritmus

Tento algoritmus funguje velmi podobně jako TabuCol; taktéž využívá tabu prohledávání. Na začátku vytvoříme částečné korektní řešení. Vrcholy, kterým nelze přiřadit žádnou z k barev, aniž by došlo ke konfliktu, jsou zařazeny do neobarvené množiny vrcholů U. *Objektová funkce* f je kardinalita množiny U. Algoritmus končí až nastane $|U| = 0$. Tah se provede tak, že se vybere neobarvený vrchol $v \in U$ a přiřadí se mu barevná třída S_j, kde $j \leq k$. Poté se vezmou všechny vrcholy $u \in S_j$, které sousedí s v a přesunou se do U. Vybere se takový vrchol, který má nejméně sousedních vrcholů v S_j, aby se přidalo co nejmíň vrcholů do U. Prvky T_{uj} v tabu seznamu jsou pak tabu (nemůžou být přesunuty zpátky do S_j) po t iterací.

Algorithm 5 PARTIALCOL $(G = (V, E), maxIterations, k)$

1: $S \leftarrow$ Greedy(G, k, U)
2: $tabu_{ij} \leftarrow 0 \; \forall i, j \in V$
3: iter $\leftarrow 1$
4: **while** iter $<$ maxIterations **do**
5: **for** $v \in U$ **do**
6: vybereme tah, který minimalizuje U a ulož do S
7: v odstrníme z U
8: **for** $u \in S_j : u \in \Gamma(v)$ **do**
9: přesunem do U
10: **end for**
11: **end for**
12: **if** $|U| = 0$ **then**
13: **break**
14: **end if**
15: pokud nebyl vybrán tah, tak vyberem náhodně
16: aktualizujem tabu seznam
17: iter $\leftarrow$ iter + 1
18: **end while**
19: **return** S

Tato část čerpá z [8].

3.5.1 Implementace PartialCol

PartialCol využívá množinu neobarvených vrcholů U. Proto je potřeba datová struktura, která není omezená velikostí a operace vkládání probíhá v konstantním čase.

```
typerdef struct {
    int num_nodes;       // počet vrcholů
    node* node;          // vrchol
    node* last_node;     // odkaz na poseldní vrchol
} uncoloured_vertices;
```

Jedná se o spojový seznam s ukazatelem na poslední prvek v seznamu. Při operaci přidávání se nemusí procházet seznamem, ale prvek je rovnou napojen na poslední uzel. Struktura je tak upravena, jelikož v každé iteraci se může přidat více prvků, ale odstraněn je vždy jen jeden.

Opět jako v TabuCol je tu pomocná matice C, která zrychluje zjišťování hodnoty objektové funkce sousedních řešení. Po přesunu vrcholu $v \in U$ do barevné třídy S_j se hodnota objektové funkce nového řešení vypočítá následovně:

$$f(S') = f(S) + C_{vj} - 1, \tag{5}$$

kde f je objektová funkce zjišťující kardinalitu množiny U. Jakmile je vrchol $v \in U$ přesunut do S_j a všechny sousední vrcholy v ($\Gamma(v)$) jsou taktéž přesunuty do U, tak pak je matice C aktualizovaná.

Algorithm 6 Update-C(v,j)

```
1: for u ∈ Γ(v) do
2:     C_uj ← C_uj + 1
3:     if c(u) = j then
4:         for w ∈ Γ(u) do
5:             C_wj ← C_wj − 1
6:         end for
7:     end if
8: end for
```

Pokud vezmeme matici C v tabulce 2 a přesuneme vrchol 5 do S_5, sousední vrcholy vrcholu 5 se inkrementují o 1. Sousední vrcholy vrcholu 6 se také inkrementují o 1. Prvek $C_{9,5}$ se inkrementuje o 1 a pak ve vnitřním cyklu se sníží o 1. Prvek 6 se přesune do množiny U, protože je v konlfiktu s vrcholem 7.

index:	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	1	1	1	1	0
2	1	0	0	0	1	1	1	0	0	1
3	0	0	0	2	2	1	0	0	1	1
4	1	1	0	2	0	1	1	1	0	0
5	2	0	0	1	1	0	0	0	1	0

Tabulka 4: Matice C po přesunu vrcholu 5 do S_5.

3.6 Hybridní evoluční algoritmus (HEA)

Evoluční algoritmus je založen na účinném lokálním prohledávání a specializovaném operátoru *rekombinace*. HEA funguje tak, že udržuje populaci kandidátních řešení, kde operátor vytváří nové řešení, které jsou vylepšovány lokálním prohledáváním. Algoritmus začíná vytvořením určitého počtu kandidátních řešení – populace. Každý člen této populace je vytvořen pomocí modifikované verze DSatur, pro který je na začátku pevně stanoven počet barev k. První vrchol je vybrán náhodně a zbytek vrcholů je vybírán podle pravidel DSatur algoritmu. Vrcholy, pro které neexistuje třída, ve kterém by nebyly v konfliktu, jsou přiřazeny do ostatních tříd náhodně. Vznikne tedy nekorektní úplné řešení.

Po vytvoření počáteční populace se pak každé řešení pokusí vylepšit pomocí lokálního prohledávání (TabuCol). Po zbytek běhu se populace vyvíjí tak, že v každé iteraci jsou z populace náhodně vybrána dvě rodičovská řešení S_1 a S_2 a jejich kopie jsou použity ve spojení s operátorem rekombinace k vytvoření

potomka S'. Potomek je poté vylepšen pomocí lokálního prohlédávání (TabuCol) a do populace se dostává nahrazením jednoho ze slabších rodičů.

Rekombinační operátor v HEA se nazývá *Greedy Partition Crossover* (GPX). Největší barevná třída rodiče je vybrána a zkopírována do potomka. Aby se zabránilo výskytu stejných vrcholů v potomkovi, jsou zkopírované vrcholy odstraněny z obou rodičů. Některé vrcholy mohou v potomkovi po skončení cyklu chybět. Tento problém se řeší přiřazením chybějících vrcholů náhodným barevným třídám.

Algorithm 7 GPX ($S_1 = \{s_1, \ldots, s_k\}, S_2 = \{s_1, \ldots, s_k\}$)

$S' \leftarrow \emptyset$
for i to k **do**
    if Pokud i je sudé **then**
        vybereme třídu z S_1, která má nejvíce prvků
    else
        vybereme třídu z S_2, která má nejvíce prvků
    end if
    přidáme vybranou třídu do potomka S'
    odeberou se prvky vybrané třídy z S_1 a S_2
end for
vrcholy, které zbyly náhodně přiřadíme do tříd potomka
return S'

	S_1	S_2	S'
1.	$\{\{v_1, v_2, v_3\}, \{v_4, v_5, v_6, v_7\}, \{v_8, v_9, v_{10}\}\}$	$\{\{v_3, v_4, v_5, v_7\}, \{v_1, v_6, v_9\}, \{v_2, v_8, v_{10}\}\}$	$\{\}$
2.	$\{\{v_1, v_2, v_3\}, \{v_8, v_9, v_{10}\}\}$	$\{\{v_3\}, \{v_1, v_9\}, \{v_2, v_8, v_{10}\}\}$	$\{v_4, v_5, v_6, v_7\}$
3.	$\{\{v_1, v_3\}, \{v_9\}\}$	$\{\{v_3\}, \{v_1, v_9\}\}$	$\{\{v_4, v_5, v_6, v_7\}, \{v_2, v_8, v_{10}\}\}$
4.	$\{\{v_9\}\}$	$\{\{v_9\}\}$	$\{\{v_4, v_5, v_6, v_7\}, \{v_2, v_8, v_{10}\}, \{v_1, v_3\}\}$
5.	$\{\}$	$\{\}$	$\{\{v_4, v_5, v_6, v_7\}, \{v_2, v_8, v_{10}\}, \{v_1, v_3, v_9\}\}$

Tabulka 5: Příklad použití GPX

V tabulce 5 na 2. řádku je do potomka přidána třída $\{v_4, v_5, v_6, v_7\}$, která je poté smázána z S_1 a jednotlivé vrcholy jsou smazány i z S_2. Na řádku 4 vrchol v_9 nebyl přidán nikam, takže je náhodně přiřazen do nějaké třídy.

Jakmile je potomek vytvořen, je upraven lokálním prohledáváním předtím než se začlení do populace. K tomuto účelu se využívá TabuCol algoritmus, který běží po předem stanovený počet iterací. V implementaci je nastaveno $100 \times n$ iterací. Velikost populace je 10, jak doporučují Galinie a Hao [6].

Algorithm 8 HEA $(G = (V, E), maxIterations, k, l, n)$

```
1: S ← ∅
2: populace[n]
3: for i=0 to n do
4:     population[i] ← DSatur(G, k)
5:     Tabucol(populatce[i], l × |V|)
6:     if konflikty(populace[i]) = 0 then
7:         S ← population[i]
8:         break
9:     end if
10: end for
11: iter ← 1
12: while iter < maxIterations do
13:     S_1 ← náhodný člen z populace
14:     S_2 ← náhodný člen z populace
15:     S' ← crossover(S_1, S_2)
16:     Tabucol(S', l × |V|)
17:     if konflikty(S') < konflikty(S_1) || konflikty(S') < konflikty(S_2) then
18:         nahradíme rodiče, který má více konfliktů
19:     end if
20:     if konflikty(S') = 0 then
21:         S ← population[i]
22:         break
23:     end if
24:     iter ← iter + 1
25: end while
26: return S
```

Tato část čerpá z [6].

3.6.1 Implementace HEA

Pro tento algoritmus využíváme datové struktury *adjancency_list* neboli seznam sousedů jako reprezentaci kandidátních řešení v populaci. Díky této struktuře můžeme rychle zjistit velikosti jednotlivých barevných tříd a pracovat s jednotlivými barevnými třídami místo toho, abychom procházeli celé pole v matici.

Hodnota pro parametr *population_size* je 10 a počet iterací pro TabuCol je $100 \times |V|$.

3.7 The AntCol algorithm

Poslední algoritmus v této práci je AntCol, který je založen na metaheuristice optimalizace pomocí mravenčí kolonie. Tato metaheuristika je inspirována způsobem, jakým mravenci využívají feromonovou stopu ke spolupráci při identifikaci zdrojů potravy v blízkosti svého hnízda. Mravenci na začátku slepě hledají potravu. Když však narazí na jídlo, tak odnesou kousek zpátky do kolonie a cestou zpět zanechávají feromonovou stopu. Ostatní mravenci mají tendenci sledovat feromonové stopy položené jinými mravenci. Čím více mravenců bude následovat tuto stopu, tím více mravenců zanechá feromony na této cestě, a tím bude silnější. Feremony na stopě se časem vypařují, což snižuje šanci, že jí mravenec bude sledovat. To znamená, že delší cesta od potravy do kolonie je pro feromonovou cestu horší, než kratší cesta, kde se feromony hromadí rychleji.

Když se toto chování převede na problém barvení grafu, tak hledání potravy pro mravence je pro nás hledání vhodných kandidátních řešení. Mravenec v našem případě reprezentuje jednu iteraci, kde se vytvoří kandidátní řešení. V průběhu algoritmu máme pevný počet mravenců, kteří vytváří řešení ovlivněné předchozími řešeními. Zejména pokud mravenci identifikovali prvky, které podle jejich názoru vedou k lepším než průměrným řešením, je pravděpodobnější, že aktuální mravenec tyto prvky zahrne do svého vlastního řešení, což obecně vede ke snížení počtu barev v průběhu algoritmu.

Na začátku se inicializuje globální matice stopy t. Globální matice stopy nese informaci o tom, zda by vrcholy v a u měly být ve stejné barevné třídě. Pokud v předchozím řešení v a u byly ve stejné barevné třídě a řešení neobsahovalo konflikty, pak bude hodnota t_{uv} vyšší, tudíž je větší šance, že oba vrcholy budou opět spolu ve stejné třídě. V každé iteraci je ale hodnota t_{uv} vynásobena o ρ, což je míra vypařování, a je zvětšena o δ_{uv}. Jedná se o pomocnou matici stopy δ, která se aktualizuje, jakmile je vytvořeno řešení. Prvky δ_{uv} mají vyšší hodnotu, právě když mravenci našli v řešení, že u a v ve společné třídě tvoří řešení bez konfliktů. Pomocná matice stopy δ se vypočítá tak, že je přičtena $F(S)$, což je objektová funkce, která má hodnotu 3, jestliže počet konfliktů řešení S vytvořená mravencem je 0, jinak je rovno $1/počet\ konfliktů$. V matici δ jsou tedy promítnuty více řešení, podle toho kolikrát už bylo vytvořeno řešení.

AntCol pro konstrukci kandidátního řešení využívá modifikovaný RLF. Funguje tak, že je povoleno pouze k tříd a zbývající vrcholy, které by byly v konfliktu, zůstanou neobarveny. První vrchol, který má být přiřazen do S_i, je vždy vybrán náhodně. Zbývající vrcholy, které se přidávají do S_i, jsou vybrány s pravděpodobností:

$$P_{vi} = \begin{cases} \frac{\tau_{vi}^{\alpha} \times \eta_{vi}^{\beta}}{\sum_{u \in X} (\tau_{ui}^{\alpha} \times \eta_{ui}^{\beta})} & \text{if } v \in X \\ 0 & \text{if } v \notin X \end{cases}$$

$$\tau_{vi} = \frac{\sum_{u \in S_i} t_{uv}}{|S_i|} \tag{6}$$

Parametr τ_{vi} se vypočítá tak, že se pro daný vrchol v a barevnou třídu i (S_i) sečtou hodnoty t_{uv}, kde u jsou všechny vrcholy v barevné třídě S_i a vydělí se počtem prvků v dané třídě ($|S_i|$). Pokud v předchozích řešeních vrchol v s vrcholy u nebyl v konfliktu a tvořil přípustné řešení, tak tau_{vi} bude nabývat vyšší hodnoty. Hodnota η_{vi} je mezitím spojena s heuristickým pravidlem, které je v tomto případě stupněm vrcholu v v grafu indukovaném množinou aktuálně neobarvených vrcholů $X \cup Y$. Pro připomenutí – množina X obsahuje neobarvené vrcholy, které lze přidat do S_i, aniž by došlo ke konfliktu. Množina Y obsahuje neobarvené vrcholy, které není možné přidat do S_i. Vyšší hodnota η_{vi} nutí algoritmus vybrat vrchol s nejvyšším stupněm. Čím vyšší hodnoty pro τ a η, tím je větší pravděpodobnost, že je vrchol v přiřazen do S_i. Parametry α a β jsou konstanty, které kontrolují sílu τ a η v rovnici. Parametr α mocní hodnotu τ a β mocní η. Vyšší hodnota α klade důraz na hodnoty v globální matici stop t. Naopak vyšší hodnota β dává větší váhu „omezeným“ vrcholům.

Algorithm 9 AntCol $(G = (V, E), maxIterations, k, l, nants)$

```
1:  S ← ∅
2:  t_uv ← 1 ∀u, v ∈ V : u ≠ v
3:  iter ← 1
4:  while iter < maxIterations do
5:      δ_uv ← 0 ∀u, v ∈ V : u ≠ v
6:      min ← k
7:      konec ← false
8:      for mravenec ← 1 to nants do
9:          S ← RLF(G, k, multiset, alfa, beta)
10:         if S je částečné řešení then
11:             náhodně přiřadíme neobarvené vrcholy k barevným třídám v S
12:             Tabucol(S, l × |V|)
13:         end if
14:         if S je přípustný then
15:             konec ← true
16:             if |S| ≤ min then
17:                 min ← |S|
18:             end if
19:         end if
20:         δ_uv ← δ_uv + F(S) ∀u, v : c(u) = c(v)  &  u ≠ v
21:     end for
22:     t_uv ← ρ × δ_uv ∀u, v ∈ V : u ≠ v
23:     if konec = true then
24:         k ← best - 1
25:     end if
26: end while
27: return S
```

Protože je vytváření třídy náhodný proces, je při vytváření řešení použita konstanta *multiset*, která určuje počet pokusů konstrukce každé barevné třídy. Z těchto pokusů se vybere ten, který má nejvíce vrcholů. Například pokud je $multiset = 5$, tak se třída S_i vytvoří pětkrát a vybere se takové S_i, které má nejvíce vrcholů, protože takové řešení ma tendenci mít nižší počet barevných tříd. Po skončení RLF vznikne řešení. Takové řešení může být částečné, v takovém případě se neobarvené vrcholy náhodně přiřadí do různých tříd. Poté se na řešení spustí Tabucol na $l \times |V|$ iterací.

Nants je počet mravenců a určuje kolik řešení je zkonstruováno v jedné iteraci. Souběžná konstrukce řešení více mravenci umožňuje zkoumat různé oblasti prostoru řešení. Větší počet mravenců obecně vede k rozsáhlejšímu zkoumání prostoru řešení, protože jsou objeveny více řešení. Čím více řešení jsou navštíveny, tím lépe se identifikuje, které vrcholy by měly být spolu ve stejné barevné třídě. Zvýšení počtu mravenců však také zvyšuje výpočetní složitost algoritmu.

Tato část čerpá z [9].

3.7.1 Implementace AntCol

AntCol při vytváření počátečního řešení využívá modifikovaného RLF. V původním RLF je použita prioritní fronta, ale není zde potřeba vybrat první vrchol ve frontě, vrchol se vybírá vždy náhodně. Proto se zde používá hashovací tabulka. Operace vkládání probíhá pouze na začátku a po zbytek algoritmu se prvky jenom odebírají z hashovací tabulky. Hashovací tabulka je velmi podobná struktuře prioritní fronty, jen *items* je pole *ht_item*.

```
typedef struct {
    int key;            // index vrcholu
    int value;          // stupeň vrcholu
} ht_item;

typedef struct {
    ht_item* items;     // pole vrcholů
    int* positions;     // odkaz na vrchol
    int size;           // počet vrcholů v tabulce
    int capacity;       // velikost tabulky
} HashTable;
```

Parametr *positions* funguje stejně jako *indices* u prioritní fronty u RLF 3.3.1. U operace mazání se vybraný prvek vymění s posledním prvkem, zmenší se velikost tabulky a aktualizuje se pole *positions*.

Hodnoty parametrů jsou následující: $multiset = 2$, $alpha = 2$, $beta = 3$, $nants = 5$, $tabu\_iteration = 1000$.

4 Experimenty s algoritmy

Algoritmy byly testovány na procesoru 1,1 GHz Dual-Core Intel Core i3, operační systém macOS Ventura. Výstupem jednotlivých algoritmů je textový soubor a funkce vrací pole celých čísel. V souboru jsou zaznamenány informace o počtu barevných tříd použité pro řešení, počet iterací a doba trvání v sekundách. Návratová hodnota funkce je pole, kde index značí vrchol a hodnota na indexu je barevná třída.

Výsledky byly získány výpočtem průměru z 10 pokusů. V této sekci porovnáme jednotlivé konstruktivní grafy mezi sebou a poté porovnáme pokročilejší algoritmy.

4.1 Testovací grafy

Testování probíhá na grafech v DIMACS [2] formátu. Jedná se o formát pro neorientované grafy, který se používá jako standart pro úlohy v neorientovaných grafech.

graf	**vrcholy**	**hrany**	**hustota**	$\boldsymbol{k/\chi}$
DSJC_1000.1.col	1000	49629	0.1	20/?
DSJC_250.5.col	250	31336	0.5	28/?
DSJC_250.9.col	250	27897	0.9	72/?
le450_25a.col	450	8260	0.08	25/25
le450_25d.col	450	17425	0.1	25/25
le450_15b.col	450	8169	0.08	15/15
flat300_28_0.col	300	21695	0.5	28/28
flat300_26_0.col	300	21633	0.5	26/26
flat300_20_0.col	300	21375	0.5	20/20

Hustota grafu je definována jako poměr počtu hran |E| vzhledem k maximálnímu možnému počtu hran. Hodnota k je nejmenší počet barevných tříd, které dosud byly u instance objeveny. U některých instancí chromatické číslo χ dosud nebylo dokázáno. K testování používáme 3 různé kategorie grafu:

- **DSJC_V.H** jsou náhodné grafy, které byly používány v [10]. Tyto grafy jsou hojně využívány pro algoritmy barvení grafu. Počet vrcholů je označen prvním číslem V, zatímco druhá číslice udává hustotu.
- **le450_K** jsou Leigtonovy grafy se 450 vrcholy a známým chromatickým číslem K. Všechny grafy mají kliku velikosti K. [5]
- **flatV_K** grafy vznikly rozdělením množiny vrcholů na K téměř stejně velkých tříd a následným výběrem hran pouze mezi vrcholy různých tříd. Nalezení nejlepšího přípustného K-obarvení je ekvivalentní obnovení tohoto

[2] Discrete mathematics and Theoretical Computer Science

počátečního rozdělení. Druhé číslo K je tedy chromatické číslo a V je počet vrcholů.

4.2 Porovnání konstruktivních algoritmů

Kontruktivní algoritmy málokdy vytváří optimální řešení. Slouží spíše ke vytváření počátečního řešení, které se pak optimalizuje.

4.2.1 Výsledky pro grafy DSJC

	Greedy	**DSatur**	**RLF**
barvy	31	26	32
průmerný čas [s]	0,0028	0,083	0,066
nejlepší čas [s]	0,0022	0,064	0,054

Tabulka 6: Porovnání konstruktivních algoritmů na instanci DSJC1000.1.col

	Greedy	**DSatur**	**RLF**
barvy	43	36	43
průmerný čas [s]	0.000239	0.00501	0.0121
nejlepší čas [s]	0.000164	0.00377	0.0103

Tabulka 7: Porovnání konstruktivních algoritmů na instanci DSJC250.5.col

	Greedy	**DSatur**	**RLF**
barvy	99	88	59
průmerný čas [s]	0.000523	0.009900	0.07143
nejlepší čas [s]	0.000425	0.007925	0.041671

Tabulka 8: Porovnání konstruktivních algoritmů na instanci DSJC250.9.col

Ve všech případech DSatur vytvoří řešení s nejmenším počtem tříd, ale časově je nejnáročnější. Naopak RLF potřebuje k obarvení grafů nejvíce tříd. Greedy je nejrychlejší ze všech algoritmů a počtem tříd je horší než DSatur. U hustějších grafů je rozdíl mezi algortimy méně patrný.

4.2.2 Výsledky pro Leigtonovy grafy

	Greedy	**DSatur**	**RLF**
barvy	28	25	25
průmerný čas [s]	0.0005	0.018	0.006
nejlepší čas [s]	0.0005	0.015	0.004

Tabulka 9: Porovnání konstruktivních algoritmů na instanci le450_25a.col

	Greedy	**DSatur**	**RLF**
barvy	35	28	33
průmerný čas [s]	0.0007	0.023	0.01
nejlepší čas [s]	0.0005	0.018	0.009

Tabulka 10: Porovnání konstruktivních algoritmů na instanci le450_25d.col

	Greedy	**DSatur**	**RLF**
barvy	22	16	20
průmerný čas [s]	0.0004	0.017	0.005
nejlepší čas [s]	0.0004	0.015	0.004

Tabulka 11: Porovnání konstruktivních algoritmů na instanci le450_15b.col

Zde, pro Leigtonovy grafy, si opět vede nejlépe DSatur a u instance le450_15b skoro dosáhl chromatického čísla (15). Časově je opět náročnější než ostatní algoritmy. U instance le450_25a pouze Greedy nedosáhl chromatického čísla, ale časově je nejrychlejší.

4.2.3 Výsledky pro Flat grafy

	Greedy	**DSatur**	**RLF**
barvy	46	42	50
průmerný čas [s]	0.0004	0.009	0.041
nejlepší čas [s]	0.0003	0.006	0.025

Tabulka 12: Porovnání konstruktivních algoritmů na instanci flat300_28_0.col

	Greedy	**DSatur**	**RLF**
barvy	45	43	49
průmerný čas [s]	0.0003	0.007	0.0018
nejlepší čas [s]	0.0003	0.005	0.015

Tabulka 13: Porovnání konstruktivních algoritmů na instanci flat300_26_0.col

	Greedy	**DSatur**	**RLF**
barvy	47	41	48
průmerný čas [s]	0.0004	0.01	0.026
nejlepší čas [s]	0.0004	0.009	0.025

Tabulka 14: Porovnání konstruktivních algoritmů na instanci flat300_20_0.col

Opět DSatur má nejmenší cenu ze všech, ale nejsou zde takové rozdíly jako u předešlých grafů. Žádný algoritmus nedosáhl chormatického čísla a všechny jsou od optimálního řešení daleko.

4.3 Porovnání pokorčilých algoritmů

Většina tabulek jsou přiloženy v příloze. Všechny algoritmy měly limit 10 000 000 iterací.

4.3.1 Výsledky pro grafy DSJC

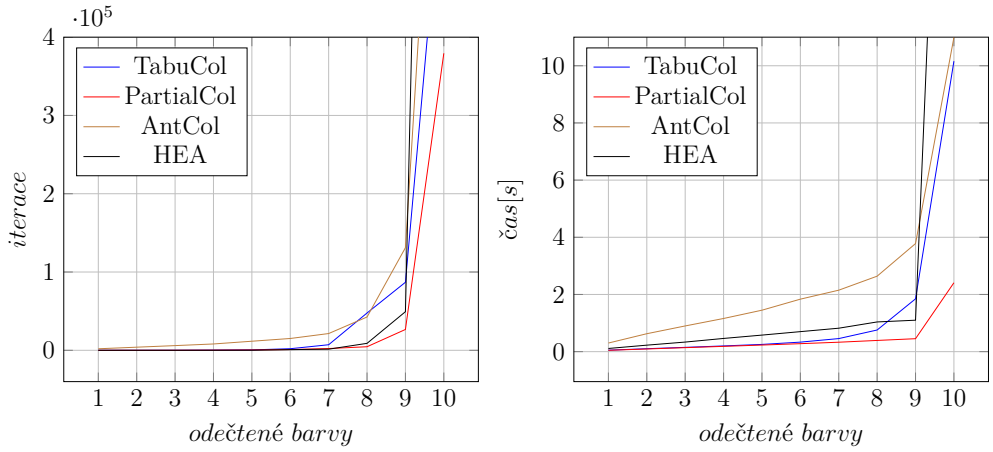
V tabulce 15 jde vidět, že PartialCol produkuje nejlepší výsledky co se týče času a počtu provedených iterací. Naopak HEA algoritmus je nejpomalejší z uvedených algoritmů. TabuCol a AntCol mají podobné výsledky. Parametry u AntCol jsou $nants = 1,\ \rho = 0.75,\ \alpha = 2,\ \beta = 3,\ multiset = 2,\ tabu\_iteration = 1000$. Používáme jenom jednoho mravence, protože algoritmus není paralelizován. U HEA je $populace = 10$ a $tabu\_iteration = 100$. Pro všechny algoritmy je tu hranice 3 000 000 iterací.

	TabuCol		**PartialCol**		**AntCol**		**HEA**	
barvy	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]
31	2	0,051	2	0,047	1953	0,301	1	0,111
30	11	0,054	13	0,046	3984	0,326	1	0,116
29	47	0,047	57	0,047	5994	0,272	1	0,108
28	238	0,050	181	0,045	8094	0,261	21	0,123
27	537	0,053	435	0,046	11 595	0,290	82	0,117
26	2032	0,080	968	0,048	15 090	0,386	868	0,121
25	7109	0,124	2247	0,051	21 407	0,314	1440	0,120
24	47 660	0,300	4564	0,062	42 536	0,492	8880	0,219
23	86 898	1,09	26 612	0,157	131 093	1,14	49 312	0,621
22	635 324	8,23	379 374	1,72	924 442	7,2	2 151 486	33,3
celkový čas[s]	10,16		2,41		10,6		33,0	

Tabulka 15: Porovnání pokročilých algoritmů na DSJC_1000.1

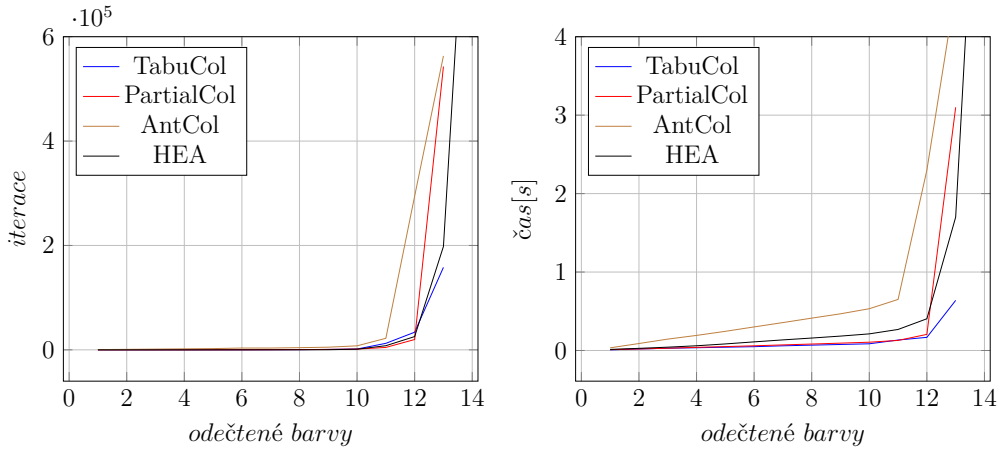
Na obrázku 9 graf nalevo představuje počet iterací jednotlivých algoritmů. Na ose x je počet barev které byly odečteny z původního k-obarvení. Vidíme, že HEA od 23. barevné třídy přerostl všechny algoritmy. Důvodem je, že algoritmus musí provést mnoho rekombinací v populaci, aby našlo přípustné řešení. Zbylé

algoritmy rostou velmi podobně, ale PartialCol má méně iterací než zbylé dva algoritmy po celý běh algoritmu a zároveň je nerychlejší.



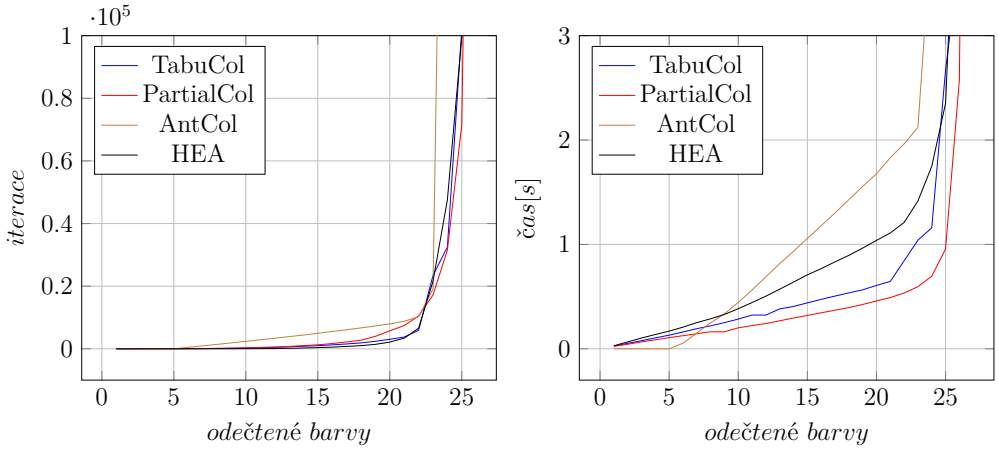
Obrázek 9: Graf pro tabulku 15

Pro instanci DSJC250.5 v tabulce 10 si nejlépe vedl algoritmus HEA, který nalezl řešení s nejnižším počtem barev. TabuCol sice má nejméně iterací a je nejrychlejší, ale oproti ostatním algoritmům by mu trvalo déle snížit jeho cenu. Optimální cena řešení je 28 a při zvýšení limitu by algoritmus HEA dokázal najít optimální řešení rychleji než ostatní.



Obrázek 10: Graf pro tabulku 18

Na grafu DSJC250.9 v tabulce 11, který je velmi hustý, si nejlépe vedl HEA, který našel optimální řešení v poměrně rychlém čase. TabuCol si vedl nejhůře. AntCol oproti ostatním algoritmům začal s počátečním řešením s méně třídami.



Obrázek 11: Graf pro tabulku 19

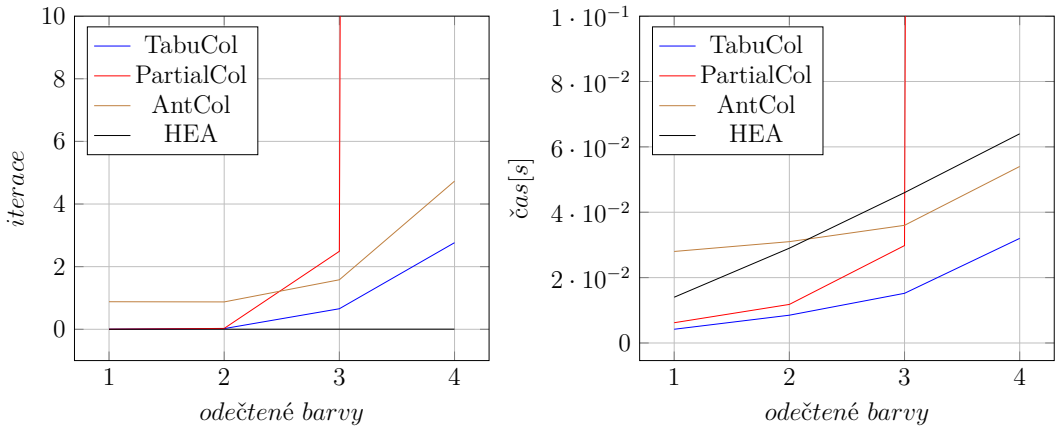
4.3.2 Výsledky pro grafy Leigtonovy grafy

PartialCol má oproti ostatním algoritmům podstatně horší výsledky. V polovině případů z 20 000 000 iterací PartialCol nebyl schopný najít chromatické číslo grafu. Naopak HEA algoritmus oproti předchozímu grafu produkuje výsledky velmi rychle. Nejrychlejší algoritmus, který najde přípustné řešení, je TabuCol. AntCol je zajímavý v tom, že nezačíná vždy na stejném počtu barevných tříd, ale u 4 z 10 pokusů začal s 28 barvama, ve 3 z 10 případů na 27 a ve zbylých případech začal s 26 barevnými třídami. V případě paralelizace algoritmu, by tak AntCol produkoval nejlepší výsledky, protože by byla velká šance, že by existoval mravenec co by našel lepší počáteční řešení než u ostatních algoritmů a k chromatickému číslu by se dostal rychleji. HEA díky GPX, najde řešení v 1 iteraci a nemusí spouštět lokální prohledávání na řešení.

	TabuCol		**PartialCol**		**AntCol**		**HEA**	
barvy	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]
28	2	0,0042	2	0,0062	880	0,028	1	0,014
27	22	0,0043	28	0,0056	874	0,031	2	0,015
26	656	0,0067	2490	0,018	1581	0,036	3	0,017
25	2765	0,013	7000000	15,8	4730	0,054	4	0,017
celkový čas[s]	0,032		16,0		0,126		0,064	

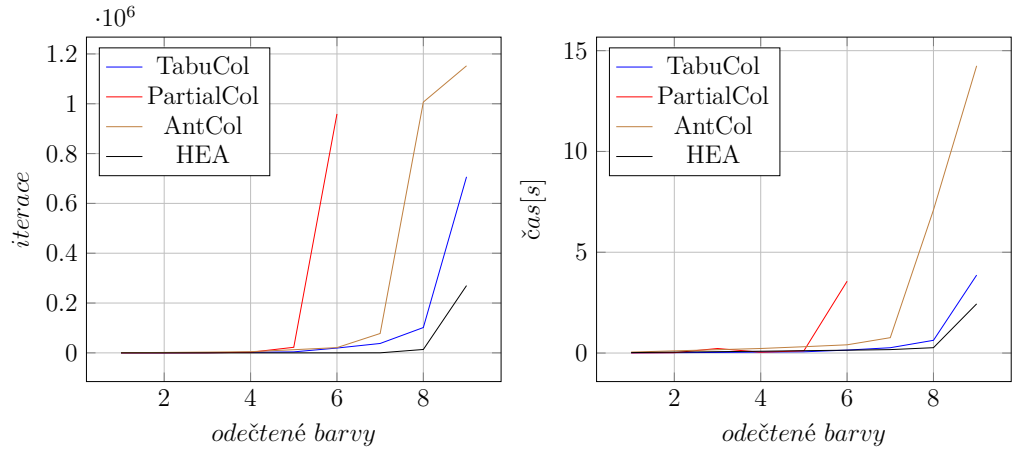
Tabulka 16: Porovnání pokročilých algoritmů na le450_25a.col

Na obrázku 12 lze vidět, že PartialCol není ideální algoritmus pro hledání chromatického čísla u Leigtonových grafů. Od 26 barevných tříd počet iterací rapidně převyšuje ostatní algoritmy. HEA má nejméně iterací, ale TabuCol je ze všech nejrychlejší.



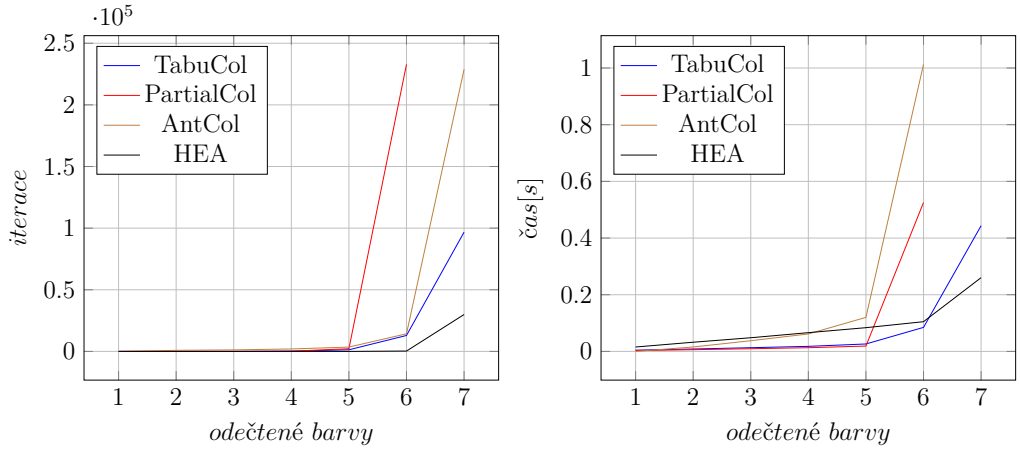
Obrázek 12: Graf pro tabulku 16

Na obrázku 13 jsou algoritmy porovnány na instanci le450_25d.col 20. Graf má chromatické číslo 25 a všechny algoritmy, až na PartialCol, dokázaly najít řešení s 27 třídami. PartialCol se nedokázal dostat pod 30 barevných tříd ani v jednom běhu.



Obrázek 13: Graf pro tabulku 20

Pro graf le450_15b na obrázku 14, který má chromatcké číslo 15, si algoritmy vedly stejně jak u le450_25d 20.



Obrázek 14: Graf pro tabulku 21

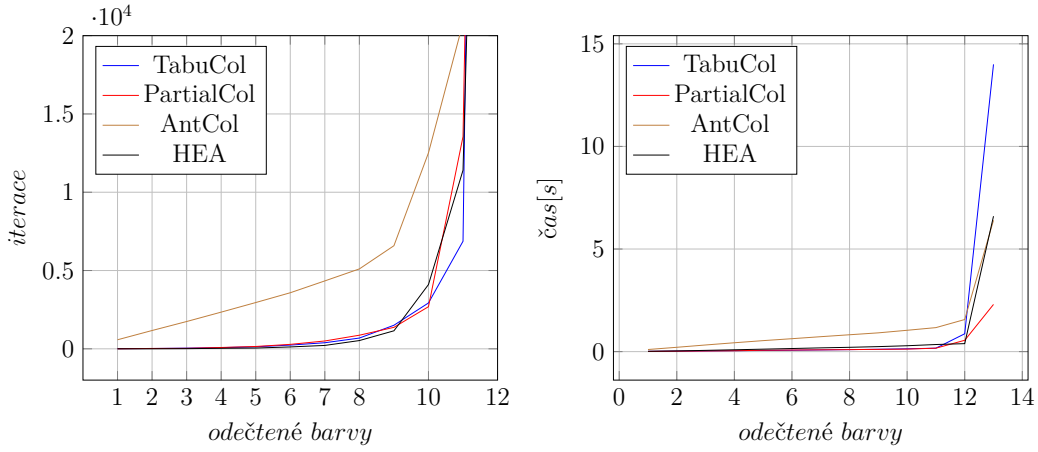
Celkově pro Leigtonovy grafy si dobře vedl HEA a TabuCol. PartialCol není ideální pro tento typ grafu a ve všech případech byl nejhorší ze všech algoritmů.

4.4 Výsledky pro Flat grafy

Pro flat graf flat300_28_0 algoritmy nedokázaly najít chromatické číslo. HEA algoritmus našel nejnižší počet barevných tříd ze všech v rozumném počtu iterací. V jednom případě byl spuštěn s limitem 100 000 000 iterací a dostal se na 32 barevných tříd. S limitem 10 000 000 iterací se v polovině případů dostal na 33 barevných tříd a ve všech případech dosáhl alespoň 34 barevných tříd. Dalším algoritmem, který se opakovaně dostal na 34 barevných tříd, je AntCol. TabuCol a PartialCol se jenom v jednom případě z deseti dostaly na 34 barevných tříd.

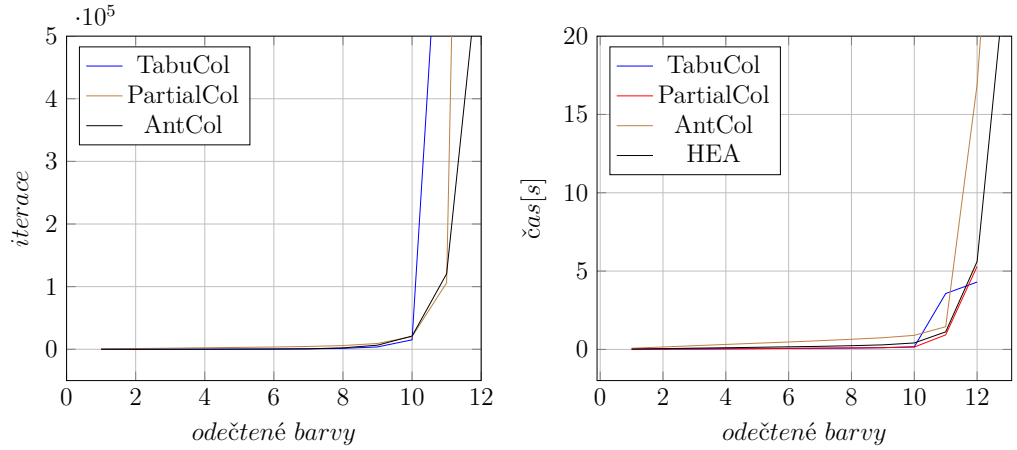
	TabuCol		**PartialCol**		**AntCol**		**HEA**	
barvy	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]
46	8	0,0118	2	0,0105	578	0,103	1	0,0162
45	20	0,0124	24	0,0112	1169	0,108	1	0,0171
44	43	0,0114	31	0,0113	1743	0,113	10	0,0277
43	76	0,0111	85	0,0095	2343	0,108	33	0,0289
42	130	0,0088	149	0,0239	2955	0,104	50	0,0301
41	239	0,0129	284	0,0112	3581	0,094	120	0,0311
40	388	0,0152	501	0,0098	4337	0,098	216	0,0319
39	687	0,0142	869	0,0107	5103	0,094	527	0,0275
38	1505	0,0179	1371	0,0111	6584	0,093	1150	0,0342
37	2929	0,0213	2690	0,0139	12 517	0,126	4104	0,0432
36	6870	0,0318	13 555	0,0449	20 824	0,130	11 408	0,0588
35	170 127	0,702	133 306	0,384	91 709	0,393	92 052	0,0429
34	2 258 989	13,1	427 351	1,3	955 515	5,3	620 490	2,8
33							3 968 345	19,7
32							10 373 551	40,6
celkový čas[s]	15,3		2,6		6,4		66,9	

Tabulka 17: Porovnání pokročilých algoritmů na flat300_28_0.col



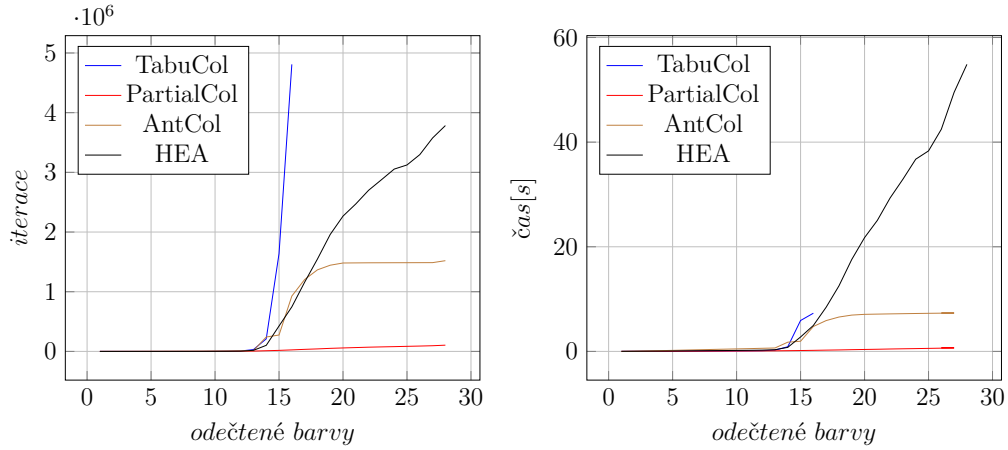
Obrázek 15: Graf pro tabulku 17

Chromatické číslo flat300_26_0.col je 26 a nejvíce se k němu přiblížil HEA. TabuCol si vedl nejhůř s PartialCol. Hybridní a populační algortimy jsou lepší pru tuto instanci



Obrázek 16: Graf pro tabulku flat300_26_0.col 22

Pro graf 17 dokázaly všechny algoritmy najít chormtické číslo, což je 20, až na TabuCol. Řešení TabuCol byl počtem tříd velmi daleko od optimálního. PartialCol si vedl výrazně lépe než všechny ostatní algoritmy. HEA byl pomalý, ale dokázal najít optimální řešení.



Obrázek 17: Graf pro tabulku 23

4.5 Shrnutí výsledků

Greedy mezi konstruktivním algoritmy vytváří nejrychleji řešení, což je ideální když nám jde o rychlost při vytváření počátečního řešení. Pokud chceme počáteční řešení, kde máme co nejméně barevných tříd, tak se více hodí DSatur algoritmus.

TabuCol algoritmus je velmi rychlý pro Leigtonovy grafy a dokázal najít stejně dobrá řešení jako ostatní algoritmy. Na grafy DSJC nedokázal najít optimální řešení a oproti ostatním algoritmům měl často horší cenu. Flat grafy byly nejhorší a v jednom případě jeho řešení bylo cenově značně špatné oproti ostatním algoritmům. Celkově TabuCol patří mezi nejdůležitější algoritmy, protože jeho metody jsou používány v jiných algoritmech.

Algoritmus PartialCol si vedl u DSJC grafů lépe než TabuCol a AntCol. Nedokázal však najít optimální řešení. Naopak u Leigtonových grafů byl nejhorší a ve všech případech cena jeho řešení byla horší než ostatních. U jedné instance (le450_25a.col) kdy nalezl stejný počet tříd jako ostatní algoritmy, byl podstatně pomalejší než ostatní algoritmy.

AntCol algoritmus je u všech grafů průměrný, ale má velký potenciál. Výhoda tohoto algoritmu je, že dokáže přeskočit několik barevných tříd. Ostatní algoritmy vždy nalézají správné řešení o jednu barevnou třídu míň než v předchozím řešení, ale mravenci v AntCol dokáží redukovat více barevných tříd a ne pouze o jednu barevnou třídu.

Nakonec Hybridní Evoluční Algoritmus (HEA) oproti ostatním algoritmů vždy našel řešení s nižším počtem tříd. Časově je pomalejší než ostatní, ale cena výsledku byla vždy lepší. Celkově je HEA nejlepší z uvedencýh algoritmů, vytváří totiž u ruzných grafů řešení se skoro optimálním řešením a je nejspolehlivější.

Závěr

V bakalářské práci jsem se věnoval problému barvení grafů. V první kapitole je představena úloha barvení grafu, jeho využití a složitost problému. V další části jsou algoritmy podrobně popsány a je přiložen pseudokód. Ve stejné kapitole jsou rozebrány implementační záležitosti, především jaké datové struktury byly použity. Popsané algoritmy jsou naprogramované v jazyce C a výstupem je knihovna s algoritmy. V poslední části je analýza jednotlivých algoritmů na různých instancí a porovnání výkonu mezi sebou.

Conclusions

In my bachelor's thesis I dealt with the problem of graph colouring. The first chapter introduces the problem of graph coloring, its applications and the complexity of the problem. In the next section, the algorithms are described in detail and pseudocode is included. In the same chapter, implementation issues are discussed, in particular what data structures were used. The algorithms described are programmed in C and the output is a library of algorithms. The last section analyses the different algorithms on different instances and compares the performance between them.

A Ukázka použití knihovny

```
#include "libcol.h"

int main(){

   max_iteration.iterations = 1000000;
   ant_default_param.nants = 1;
   ant_default_param.alpha = 2;
   ant_default_param.beta = 3;
   ant_default_param.multiset = 2;
   ant_default_param.tabu_iteration = 1000;

   Graph* graph = col2graph("DSJC250.9.col");

   int* solution = Antcol(graph, "antcol_results.txt");

   return 0;
```

Zdrojový kód 1: Ukázka použití knihovny

Ve zdrojovém kódu na začátku importuji knihovnu *libcol.h*, která obsahuje algoritmy. Explicitně nastavuji parametry funkce AntCol. Všechny funkce mohu zavolat bez nastavení parametrů. Algoritmy mají dva povinné parametry – soubor anebo už datovou strukturu grafu a název výstupního souboru. Po zavolání funkce algoritmus pracuje tiše a po skončení se vytvoří soubor. Soubor antcol_results.txt obsahuje výsledky ve formě tabulky.

ANTCOL

colour	time	total time	iteration
31	0.339859	0.361628	1990
30	0.346428	0.725462	3997
29	0.285976	1.027599	5982
28	0.314798	1.358603	8022
27	0.306804	1.680862	11447
26	0.286297	1.981855	15413
25	0.344872	2.340253	22797
24	0.351774	2.705897	33777
23	0.597516	3.317524	63482
22	7.971871	11.307188	684383

Obrázek 18: Výsledky v souboru effort.txt

Na obrázku 18 vidíme že v souboru jsou 4 sloupečky. Sloupec colours je

informace o počtu barevných tříd. Time je informace o tom jak dlouho se trvalo dostat z předešlého řešení na řešení o jednu barevnou třídu míň. Total time je celkový čas od spuštění algoritmu. Iteration je počet iterací, které bylo vykonáno.

B tabulky

	TabuCol		**PartialCol**		**AntCol**		**HEA**	
barvy	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]
43	1	0,009	2	0,011	467	0,034	1	0,013
42	3	0,014	3	0,008	937	0,054	1	0,013
41	16	0,008	9	0,009	1417	0,056	1	0,015
40	30	0,007	37	0,011	1900	0,047	1	0,017
39	63	0,006	84	0,009	2389	0,052	9	0,023
38	114	0,006	159	0,010	2892	0,056	34	0,026
37	196	0,010	321	0,011	3446	0,055	86	0,025
36	327	0,007	524	0,011	4170	0,056	147	0,023
35	798	0,007	851	0,011	5085	0,055	265	0,024
34	1543	0,005	1453	0,011	7697	0,06	1269	0,028
33	12 466	0,04	4617	0,023	22117	0,11	7921	0,056
32	33 761	0,03	19 793	0,077	294565	1,6	25 404	0,13
31	158 023	0,4	542 895	2,6	563184	2,3	198 106	1,2
30							1 115 912	6,7
celkový čas[s]	0,7		2,41		10,6		33,0	

Tabulka 18: Porovnání pokročilých algoritmů na DSJC250.5.col

	TabuCol		**PartialCol**		**AntCol**		**HEA**	
barvy	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]
99	1	0,022	1	0,023			1	0,029
98	8	0,020	2	0,019			1	0,036
97	10	0,021	11	0,022			1	0,042
96	17	0,022	21	0,019			1	0,045
95	22	0,021	36	0,020			1	0,041
94	45	0,020	59	0,018	470	0,053	2	0,042
93	68	0,020	93	0,019	948	0,097	5	0,036
92	112	0,021	141	0,018	1420	0,097	10	0,037
91	169	0,022	208	0,019	1894	0,086	15	0,042
90	244	0,020	301	0,018	2381	0,109	31	0,055
89	364	0,022	422	0,022	2870	0,118	56	0,061
88	475	0,021	567	0,018	3379	0,127	81	0,064
87	637	0,022	758	0,026	3888	0,125	158	0,067
86	803	0,025	1003	0,026	4400	0,118	250	0,064
85	1005	0,024	1273	0,024	5526	0,123	388	0,071
84	1259	0,020	1683	0,025	6073	0,121	529	0,062
83	1550	0,022	2183	0,024	6679	0,122	763	0,074
82	1886	0,021	2741	0,026	7296	0,127	1004	0,074
81	2394	0,023	3979	0,029	7946	0,126	1424	0,077
80	3037	0,026	5739	0,033	8803	0,135	2138	0,073
79	3755	0,026	7410	0,032	10 201	0,139	3390	0,072
78	5893	0,035	10 500	0,043	19 071	0,132	6574	0,100
77	23 390	0,140	17 184	0,061	302 684	1,9	21 529	0,201
76	32 509	0,056	31 521	0,099	1 131 327	5,3	47 593	0,331
75	102 010	0,420	71 606	0,26	1 530 390	3,1	99 769	0,604
74	223 287	1,12	373 311	1,6	2 444 274	14,7	3 699 934	3,0
73			3 322 719	13,1	4 012 513	26,7	1 054 471	6,7
72							2 950 792	16,8
celkový čas[s]	3,7		15,6		32,4		24,0	

Tabulka 19: Porovnání pokročilých algoritmů na DSJC250.9.col

	TabuCol		**PartialCol**		**AntCol**		**HEA**	
barvy	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]
35	59	0.009540	10	0.008775	856	0.048256	1	0.022255
34	73	0.008872	76	0.008621	1728	0.060743	1	0.022342
33	574	0.011276	242	0.009420	3017	0.057163	1	0.022751
32	2353	0.016918	2379	0.014376	4807	0.059653	1	0.022512
31	3958	0.014640	22384	0.069932	12702	0.087815	3	0.026529
30	19 320	0.084989	959 042	3.449951	21078	0.094674	9	0.025155
29	37 718	0.117665			77786	0.354488	207	0.028649
28	101 666	0.367305			1006764	5.376503	13626	0.095584
27	706699	3.236377			1152295	6.334454	270025	1.996651
celkový čas[s]	3.8		3.5		14.2		2.4	

Tabulka 20: Porovnání pokročilých algoritmů na le450_25d.col

	TabuCol		**PartialCol**		**AntCol**		**HEA**	
barvy	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]
22	1	0.004	1	0.003	421	0.0156	1	0.015
21	2	0.004	5	0.002	876	0.0220	1	0.016
20	10	0.004	23	0.003	1236	0.0238	1	0.015
19	211	0.004	173	0.003	1873	0.0382	1	0.017
18	1307	0.008	2158	0.005	3521	0.058	1	0.017
17	12 924	0.058	232 864	0.506	14 286	0.268	279	0.021
16	96 739	0.358			228 791	0.892	29 926	0.015
celkový čas[s]	0.443		0.525		1.01		0.260	

Tabulka 21: Porovnání pokročilých algoritmů na le450_15b.col

	TabuCol		**PartialCol**		**AntCol**		**HEA**	
barvy	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]
45	18	0.009870	19	0.011240	581	0.074791	9	0.024683
44	39	0.009908	57	0.011413	1160	0.081566	16	0.024481
43	75	0.009286	119	0.011907	1750	0.079240	40	0.030523
42	144	0.008442	208	0.011052	2357	0.082681	91	0.027401
41	295	0.011314	378	0.011650	2985	0.078558	169	0.028364
40	470	0.011428	610	0.012164	3621	0.076822	270	0.030794
39	708	0.012040	899	0.012487	4459	0.083484	688	0.029885
38	1494	0.015065	1848	0.012059	5912	0.091025	2258	0.040417
37	3717	0.024002	3744	0.016505	9256	0.097356	6521	0.054575
36	15071	0.057666	12716	0.039459	20310	0.152681	21049	0.121066
35	903867	3.398848	191073	0.779656	106160	0.550394	120547	0.709504
34	1261297	0.838691	806123	3.982208	2828678	15.458260	649871	4.490496
33					4458086	25.860063	2962249	18.779760
32							7418989	25.372442
celkový čas[s]	4.3		5.6		33.5		51.2	

Tabulka 22: Porovnání pokročilých algoritmů na flat300_26_0.col

	TabuCol		**PartialCol**		**AntCol**		**HEA**	
barvy	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]	iterace	čas[s]
47	1	0.010154	2	0.008084	568	0.048256	1	0.013489
46	4	0.009382	6	0.008621	1131	0.060743	1	0.013236
45	10	0.008592	15	0.007490	1732	0.057163	1	0.015367
44	23	0.008466	38	0.007419	2315	0.059653	8	0.015837
43	48	0.008332	81	0.007102	2906	0.040016	24	0.021263
42	95	0.009284	163	0.007057	3542	0.046812	51	0.025155
41	179	0.008203	283	0.007925	4151	0.055933	107	0.020322
40	276	0.008533	455	0.008274	4869	0.054733	200	0.023964
39	726	0.010680	713	0.007807	6411	0.059834	322	0.019091
38	2290	0.014836	1131	0.008121	7838	0.056800	920	0.023743
37	4623	0.015356	2016	0.009879	10295	0.054570	1638	0.032101
36	7358	0.018260	3680	0.012513	14022	0.055331	3926	0.081195
35	30135	0.082684	5563	0.012524	240926	0.056145	17978	0.444881
34	210503	0.643297	11363	0.026705	274523	0.053361	101650	1.983967
33	1638786	1.952115	17219	0.028422	927273	2.826249	425891	2.204931
32	4809304	4.720424	25463	0.036176	1204021	1.128570	746199	3.486733
31			34420	0.041696	1366393	0.945670	1159311	5.042739
30			41945	0.037202	1444216	0.174682	1547032	5.042739
29			51101	0.049988	1481206	0.369876	1959418	5.042739
28			58031	0.039624	1482919	0.045285	2266787	4.177741
27			64320	0.039538	148424	0.678443	2470974	3.295484
26			70828	0.049283	1485167	0.369876	2699765	4.297344
25			75529	0.039416	1485720	0.143200	2875649	3.580618
24			79831	0.035567	1485720	0.033802	3054531	3.819299
23			84975	0.042403	1486265	0.027953	3123790	1.584615
22			89669	0.041618	1486825	0.037116	3296814	4.136418
21			95085	0.049781	1487385	0.037437	3575975	7.063073
20			104516	0.082965	1520009	0.027458	3783749	5.343064
celkový čas[s]	7.3		0.7		7.3		54.8	

Tabulka 23: Porovnání pokročilých algoritmů na flat300_20_0.col

C Obsah elektronických dat

Tato data jsou nedílnou součástí práce a tvoří (datovou) přílohu textu práce. Povinné položky struktury dat jsou:

`text/`
: Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu (případně v ZIP archivu), tj. zdrojový text textu a příloh, vložené obrázky, apod.

`src/`
: Obsahuje zdrojový kód knihovny.

`README.md`
: Textový soubor s informacemi o používání knihovny.

`output/`
: Obsahuje textový soubor s výsledky běhu jednotlivých algoritmů.

`DIMACS_input/`
: Obsahuje DIMACS soubory, které byly použity pro testování.

Literatura

[1] LEWIS, R.M.R. *A Guide to Graph Colouring*. 2016. Dostupný také z: ⟨https://doi.org/10.1007/978-3-319-25730-3⟩.

[2] RADIM BĚLOHLÁVEK, Vilém Vychodil. *Diskrétní matematika pro informatiky II*. Dostupný z: ⟨http://belohlavek.inf.upol.cz/vyuka/dm2.pdf⟩.

[3] KOSOWSKI, Adrian; MANUSZEWSKI, Krzysztof. *Classical coloring of graphs*. 2004. Dostupný také z: ⟨https://doi.org/10.1090/conm/352/06369⟩.

[4] JANCZEWSKI, R.; KUBALE, M.; MANUSZEWSKI, K.; PIWAKOWSKI, K. The smallest hard-to-color graph for algorithm DSATUR. *Discrete Mathematics*. 2001, roč. 236, č. 1-3, s. 151–165. Dostupný také z: ⟨https://doi.org/10.1016/s0012-365x(00)00439-8⟩.

[5] LEIGHTON, F.T. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*. 1979, roč. 84, č. 6, s. 489. Dostupný také z: ⟨https://doi.org/10.6028/jres.084.024⟩.

[6] GALINIER, Philippe; HAO, Jin-Kao. *Journal of Combinatorial Optimization*. 1999, roč. 3, č. 4, s. 379–397. Dostupný také z: ⟨https://doi.org/10.1023/a:1009823419804⟩.

[7] HERTZ, A.; WERRA, D. de. Using tabu search techniques for graph coloring. *Computing*. 1987, roč. 39, č. 4, s. 345–351. Dostupný také z: ⟨https://doi.org/10.1007/bf02239976⟩.

[8] BLÖCHLIGER, Ivo; ZUFFEREY, Nicolas. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers &amp Operations Research*. 2008, roč. 35, č. 3, s. 960–975. Dostupný také z: ⟨https://doi.org/10.1016/j.cor.2006.05.014⟩.

[9] DOWSLAND, Kathryn A.; THOMPSON, Jonathan M. An improved ant colony optimisation heuristic for graph colouring. *Discrete Applied Mathematics*. 2008, roč. 156, č. 3, s. 313–324. Dostupný také z: ⟨https://doi.org/10.1016/j.dam.2007.03.025⟩.

[10] JOHNSON, David S.; ARAGON, Cecilia R.; MCGEOCH, Lyle A.; SCHEVON, Catherine. Optimization by Simulated Annealing: An Experimental Evaluation Part II, Graph Coloring and Number Partitioning. *Operations Research*. 1991, roč. 39, č. 3, s. 378–406. Dostupný také z: ⟨https://doi.org/10.1287/opre.39.3.378⟩.