Kolorowanie grafów za pomocą metod heurystycznych (SK16) - GIS sprawozdanie 2

Jakub Mazur, Maciej Purski

17 maja 2019

1 Temat projektu

Projekt ma na celu implementację algorytmu rozwiązującego problem kolorowania grafu. Przez kolorowanie rozumiemy podział zbioru wierzchołków V na k rozłącznych klas C_k w taki sposób, że jeśli dowolne dwa wierzchołki u i v należą do jednej klasy C_k , to wierzchołki te nie są połączone krawędzią.

Celem algorytmu będzie wyznaczenie takiego podziału na klasy, by liczba k była jak najmniejsza oraz, by nie występowały konflikty tj. nie można było znaleźć dwóch wierzchołków połączonych krawędzią, które byłyby w jednej klasie. Do rozwiązania problemu zostanie wykorzystana metoda heurystyczna **przeszukiwania z tabu**.

2 Algorytm

Przeszukiwanie z tabu jest heurystyczną metodą służącą do rozwiązywanie problemów optymalizacyjnych. Opiera się na przeszukiwaniu lokalnego sąsiedztwa danego rozwiązania. Tego typu metody posiadają silną tendencję do zatrzymywania się w optimum lokalnym. Algorytm przeszukiwania z tabu zakłada przechowywanie pewnego zbioru ostatnio odwiedzonych rozwiązań, tak aby przeszukiwanie nie *utknęło* w optimum lokalnym.

Algorytm może być używany do różnego typu problemów optymalizacyjnych. Jego postać ogólna została przedstawiona w [1]. Dla zastosowania algorytmu do rozwiązania problemu kolorowania grafu, należy uszczegółowić jego poszczególne elementy, co jest tematem kolejnych podrozdziałów.

2.1 Reprezentacja rozwiązania

Rozwiązaniem będziemy nazywali podział zbioru wierzchołków *V* na *k* niepustych podzbiorów, które można interpretować jako *klasy* lub *kolory*.

2.2 Warunki początkowe

Początkowa liczba kolorów k będzie stanowiła parametr algorytmu. Wierzchołki zostaną pokolorowane w sposób losowy z gwarancją, że każdy z k kolorów zostanie użyty co najmniej raz - czyli innymi słowy zbiór wierzchołków grafu zostanie podzielony na k niepustych podzbiorów. Takie początkowe kolorowanie nie musi być wcale kolorowaniem poprawnym.

2.3 Sąsiedztwo

Zgodnie z [3] **sąsiednim** rozwiązaniem $y \in N(x)$ do rozwiązania x, nazywamy takie rozwiązanie *y*, które można uzyskać poprzez zmianę przynależności jednego z wierzchołków rozwiązania *x*. Generacja jednego sąsiada polega na wylosowaniu jednej z niepustych klas C_i, następnie wylosowaniu wierzchołka $v \in C_i$ oraz liczby $j \in \langle 1, k \rangle$, gdzie k to liczba klas. Następnie wierzchołek v jest kolorowany na kolor j. Jeżeli wylosowano tę samą klasę, do której należy wierzchołek, losowanie jest powtarzane.

W każdym kroku generowane jest g sąsiadów *punktu roboczego*[1], gdzie g stanowi parametr algorytmu. Spośród zbioru sąsiadów wybierany jest najlepszy element względem funkcji kosztu, który nie należy do listy tabu. Będzie on nowym punktem roboczym.

2.4 Lista tabu

W naszym projekcie skorzystamy z metody zarządzania oraz reprezantacji listy tabu przedstawionej w artykule [5]. Lista tabu zostanie zaimplementowana jako kolejka FIFO o rozmiarze n zadanym jako parametr algorytmu. Będzie ona przechowywała zakazane *ruchy*. *Ruchem* nazywamy zmianę wierzchołka *v* z koloru *i* na kolor *j*. W każdej iteracji algorytmu wybierany jest najlepszy ruch (pod względem wartości funkcji celu), który nie znajduje się na liście tabu. Stworzy on nowy punkt roboczy. Na listę trafi natomiast ruch *odwrotny* do wykonanego, a zatem będzie to para składająca się z wierzchołka *v* oraz koloru *i*, będącego *starym* kolorem. Kiedy lista tabu się zapełni, zostanie z niej usunięty ostatni element.

[1] tj. aktualnie przetwarzanego

2.5 Funkcja kosztu

Aby możliwe było porównywanie poszczególnych rozwiązań, należy zdefiniować funkcję kosztu, która ma być zminimalizowana przez algorytm przeszukiwania z tabu. Pochodzi ona z artykułu [3]. Ma ona postać:

$$cost(x) = -\sum_{i=1}^{k} |C_i|^2 + \sum_{i=1}^{k} 2 \cdot |C_i| \cdot |E_i| \tag{1}$$

gdzie C_i oznacza i-tą klasę kolorowania, natomiast E_i oznacza zbiór konfliktujących krawędzi w tej klasie. Pierwszy człon tej funkcji (pierwsza suma) odpowiada za minimalizację liczby klas poprzez *premiowanie* jak najbardziej licznych klas. Drugi człon zas odpowiada za minimalizację liczby konfliktujących krawędzi.

Jedną z głównych obserwacji odnośnie tej funkcji, przedstawionych we wspomnianym artykule, jest to, że jej minima lokalne odpowiadają podziałom bez konfliktów. Należy zauważyć, że liczba k zawarta we wzorze nie zakłada z góry ilości użytych kolorów. Minimalizacja liczby użytych kolorów będzie niejako „efektem ubocznym" minimalizacji wartości funkcji celu.

2.6 Kryterium stopu

Algorytm nie ma naturalnego kryterium stopu. Jedną z możliwości, na którą zdecydowaliśmy się w naszej implementacji jest zatrzymanie algorytmu po zadanej liczbie iteracji t, która jest parametrem agorytmu.

2.7 Parametry

Reasumując, algorytm będzie przyjmował następujące parametry:

- Początkowa liczba kolorów
- Liczba iteracji
- Rozmiar sąsiedztwa
- Rozmiar listy tabu

3 Implementacja

Algorytm zostanie zaimplementowany w języku C++. Będzie on odczytywał graf w formie tekstowej ze standardowego wejścia i wypisywał rozwiązanie na standardowe wyjście. Możliwe będzie wypisywanie historii wartości funkcji kosztu

aktualnego punktu roboczego w celu zbadania jej zmienności w ciągu kolejnych iteracji algorytmu oraz wypisanie najlepszego znalezionego rozwiązania.

Do automatyzacji procesu ewaluacji rozwiązania posłuży skrypt w języku Python, który będzie powtarzał eksperymenty odpowiednią liczbę razy oraz rysował wykresy obrazujące zmianę wartości funkcji kosztu.

3.1 Struktury danych

3.1.1 Reprezentacja grafu

Graf zostanie zaimplementowany jako **macierz sąsiedztwa**. Z punktu widzenia naszego algorytmu istotna jest bowiem możliwość sprawdzania w czasie stałym, czy dana krawędź istnieje. W tym celu użyta zostanie klasa *boost::adjacency_matrix* z bilbioteki *Boost Graph*.

3.1.2 Lista tabu

Lista tabu zostanie zaimplementowana jako bufor cykliczny o stałym rozmiarze, który umożliwia dodawanie elementów na koniec oraz usuwanie z początku w czasie stałym. Do implementacji listy tabu zostanie użyta klasa *boost::circular_buffer* z biblioteki *Boost Container*.

3.1.3 Punkt roboczy

Punkt roboczy, czyli aktualnie najlepsze rozwiązanie będzie reprezentowany jako *wektor klas*, gdzie *klasą* nazywamy podzbiór wierzchołków. Liczba klas k jest stała, w wektorze są zatem przechowywane także takie klasy, które są puste. *Klasa* składa się z listy wierzchołków do niej należących oraz liczby konfliktów w danej klasie.

3.2 Pseudokod

$G \leftarrow odczytaj()$
$x \leftarrow zainicjujPunktRoboczy(k)$ ▷ parametr k - początkowa liczba klas
$globalnieNajlepszy \leftarrow x$
$tabu \leftarrow zainicjujListeTabu(t)$ ▷ parametr t - rozmiar listy tabu
$i \leftarrow 0$
while $i < iters$ **do** ▷ parametr iters - liczba iteracji
    $ruchyVek \leftarrow zainicjujWektorRuchow()$
    for $j \leftarrow 0; j < N; j++$ **do** ▷ Generacja losowych sasiadow
        $cPocz \leftarrow rand(0, k)$
        $v \leftarrow rand(0, nWierzcholkow(cPocz))$
        $cKon \leftarrow rand(0, k)$
        $ruchyVek.push\_back(cPocz, v, cKon)$
    end for
    $najlepszyDot \leftarrow null$
    for $g \in ruchyWek$ **do** ▷ Wybór najlepszego sąsiada
        if $koszt(x, g) < koszt(x, najlepszyDot)$ **and** g **not in** $tabu$ **then**
            $najlepszyDot \leftarrow g$
        end if
    end for
    $x.wykonajRuch(najlepszyDot)$
    $tabu.dodajRuchOdwrotny(najlepszyDot)$
    $i++$
    if $koszt(x) < koszt(globalnieNajlepszy)$ **then**
        $globalnieNajlepszy \leftarrow x$
    end if
end while
Zwróć $globalnieNajlepszy$

Powyższy pseudokod opisuje ogólne działanie algorytmu. Zaniedbane są pewne szczegóły w tym obliczanie funkcji kosztu na podstawie wzoru 1. Możliwa jest optymalizacja tej funkcji poprzez obliczanie jedynie zmiany jej wartości po wykonaniu potencjalnego ruchu. Wiadomo, że wartość $|C_{cPocz}|$ spadnie o 1 a wartość $|C_{cKon}|$ wzrośnie o 1. Liczba konfliktów wewnątrz klasy zmieni się także jedynie dla klas $cPocz$ oraz $cKon$. Aby sprawdzić o ile zmniejszy się wartość $|E_{cPocz}|$ wystarczy policzyć ile wierzchołków z tej klasy konfliktowało z wierzchłkiem v. Podobnie dla E_{cKon} wystarczy policzyć ile wierzchołków byłoby w konflikcie z wierzchołkiem v po dodaniu go do klasy.

4 Badanie rozwiązania

4.1 Dobór parametrów

Przed rozpoczęciem właściwych eksperymentów i oceny efektywności rozwiązania, konieczne jest określenie optymalnych wartości parametrów tj. przede wszystkim długości listy tabu oraz rozmiaru sąsiedztwa. Parametry te zostaną ustalone doświadczalnie – algorytm zostanie uruchomiony dla kilku standardowych przypadków grafów, np. grafu losowego $G_{n,p}$[2] z różnymi wartościami parametrów. Pozwoli to ustalić, dla jakich wartości parametrów algorytm działa efektywnie.

4.2 Optymalizacja funkcji kosztu

Ważnym elementem oceny rozwiązania jest zbadanie, jak zmienia się wartość funkcji kosztu wraz z kolejnyi iteracjami. Aby można było zbadać jej zmienność w ciągu kolejnych iteracji, program będzie logował wartości funkcji celu w kolejnych iteracjach. Dane te zostaną zebrane w formie wykresów. Pozwoli to sprawdzić, czy wartość funkcji zbiega do minimum, a także, ile iteracji jest do tego potrzebne.

4.3 Ewaluacja rozwiązania

Najważniejszym kryterium ewaluacji rozwiązania dla konkretnego przypadku grafu *G* będzie uzyskana liczba chromatyczna *k*. Aby można było w sposób miarodajny określić efektywność implementacji, należy przetestować algorytm na wielu różnorodnych przypadkach grafów. W tym celu zostaną wykorzystane standardowe grafy wykorzystywane w wielu artykułach traktujących o tematyce kolorowania grafów. Doskonały przegląd tych grafów, a także wyników uzyskanych przez najlepsze algorytmy można znaleźć w [2].

Dla potrzeb ewaluacji rozwiązania algorytm zostanie uruchomiony na przypadkach grafów opisanych we wspomnianym artykule. Uzyskana liczba chromatyczna oraz czas działania i liczba iteracji programu zostaną porównane z wynikami przedstawionymi w artykule.

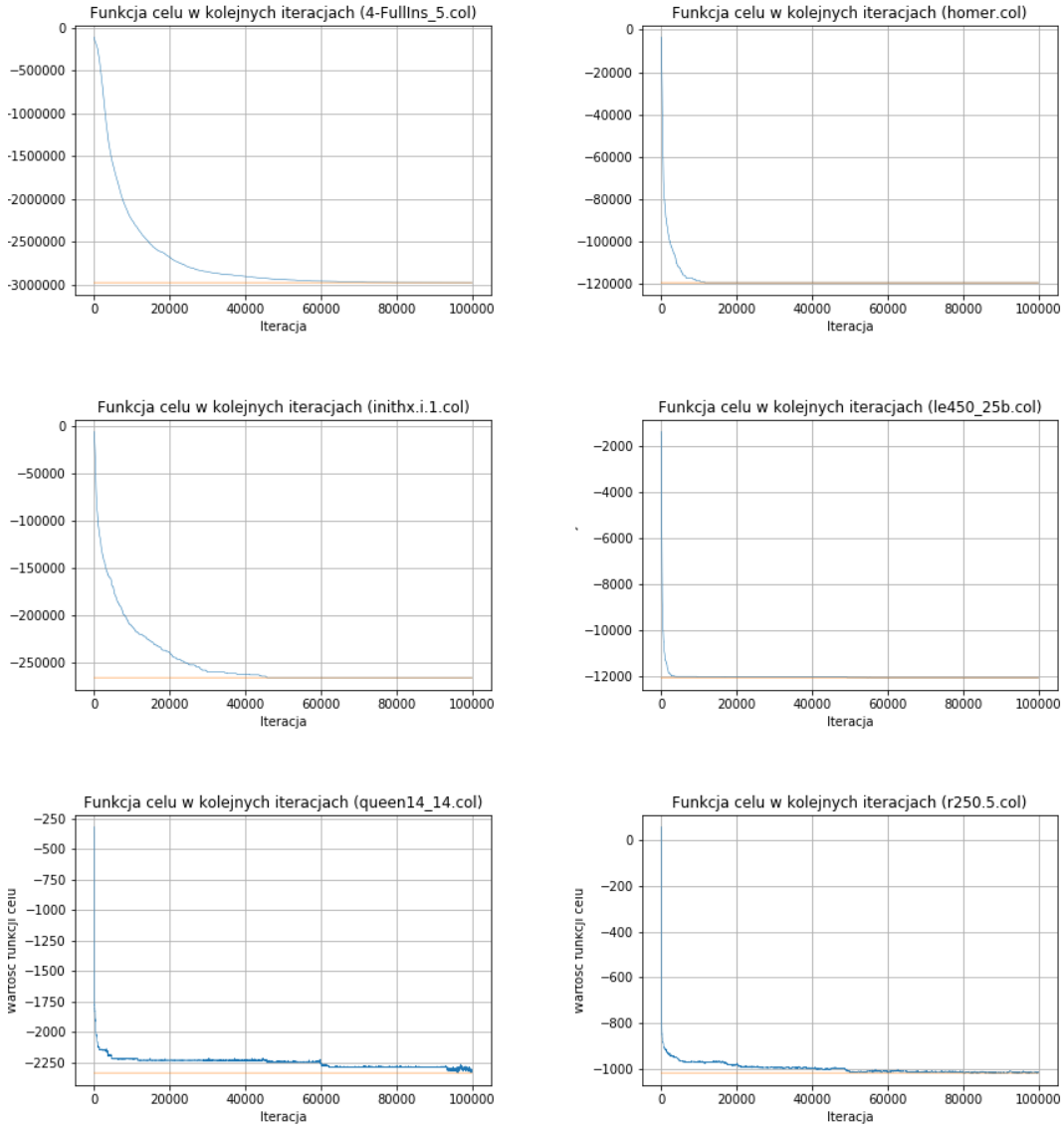
[2]Jest to graf losowy, generowany według modelu Erdősa–Rényi [4], gdzie n oznacza ilość wierzchołków, natomiast p oznacza prawdopodobieństwo, że dowolna z możliwych krawędzi zostanie włączona do grafu.

5 Eksperymenty

Zostały przeprowadzone dwie fazy eksperymentów: badanie parametrów oraz poszukiwanie najlepszych rozwiązań - jako najlepsze uznajemy rozwiązanie o jak najniższej liczbie chromatycznej bez konfliktów. Eksperymenty polegały na wielokrotnym uruchamianiu algorytmu dla różnych rodzajów grafów, przy kilku zestawach parametrów startowych. Podczas eksperymentów wykorzystano zestaw grafów stworzonych na potrzeby konkursu *The Second DIMACS Implementation Challenge*[6]. Dotyczył on implementacji algorytmów rozwiązujących problemy NP-trudne na grafach, m.in. problem kolorowania wierzchołków. Powstały dla celów konkursu zbiór problemów obejmuje grafy o różnej budowie, dlatego świetnie nadaje się do przetestowania przygotowanej implementacji.

5.1 Zmienność funkcji kosztu

Na rysunku (1) widoczna jest zmienność kosztu aktualnie najlepszego kolorowania. Poziomą żółtą linią oznaczono na wykresach wartość minimalną uzyskaną przez algorytm. Dla grafów większych zbieżność jest znacznie wolniejsza (widoczne na trzech pierwszych wykresach). Na dwóch ostatnich wykresach widoczna jest *skokowość* zmian wartości funkcji kosztu.



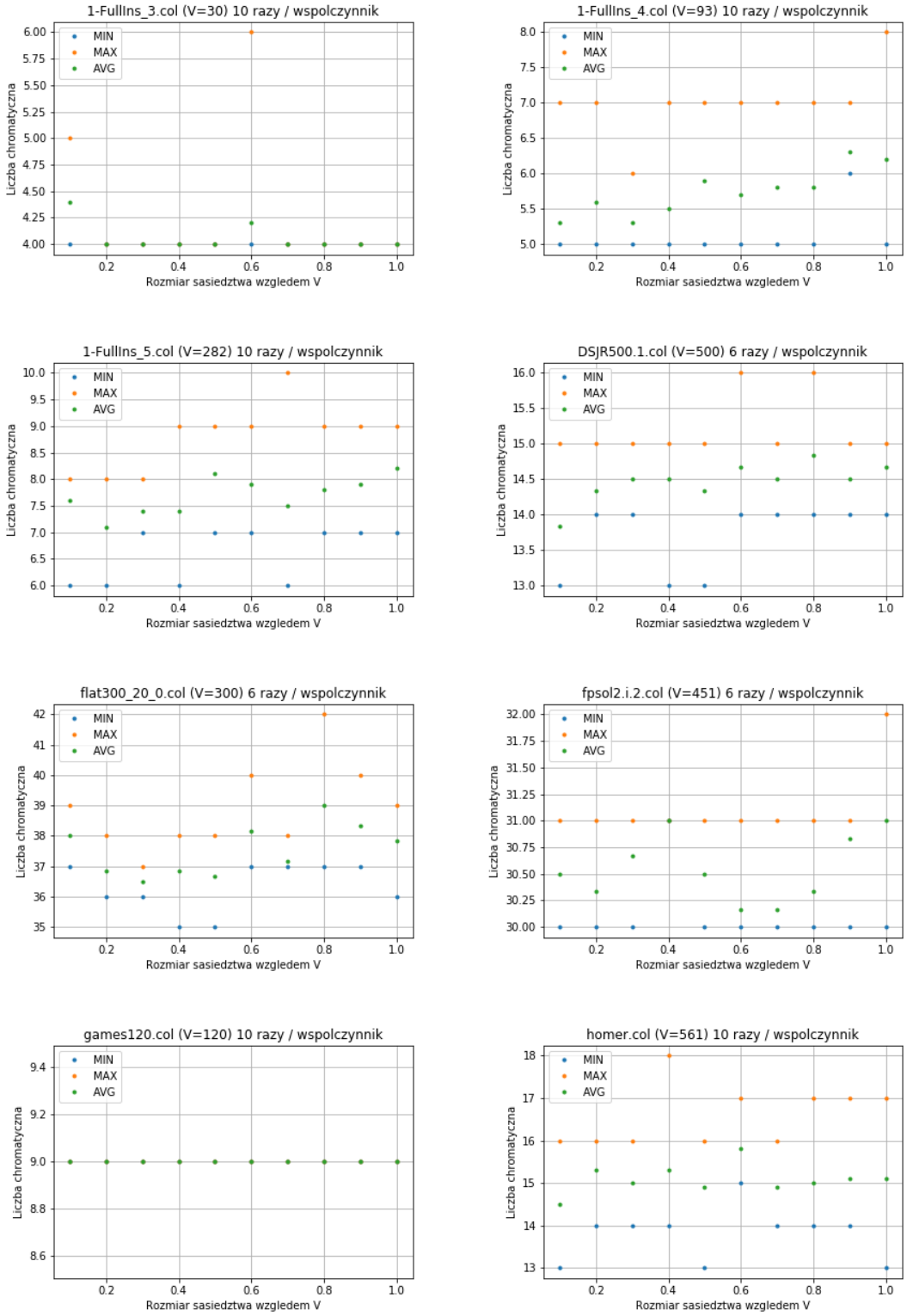
Rysunek 1: Badanie zmienności funkcji kosztu na różnych grafach

5.2 Badanie parametrów

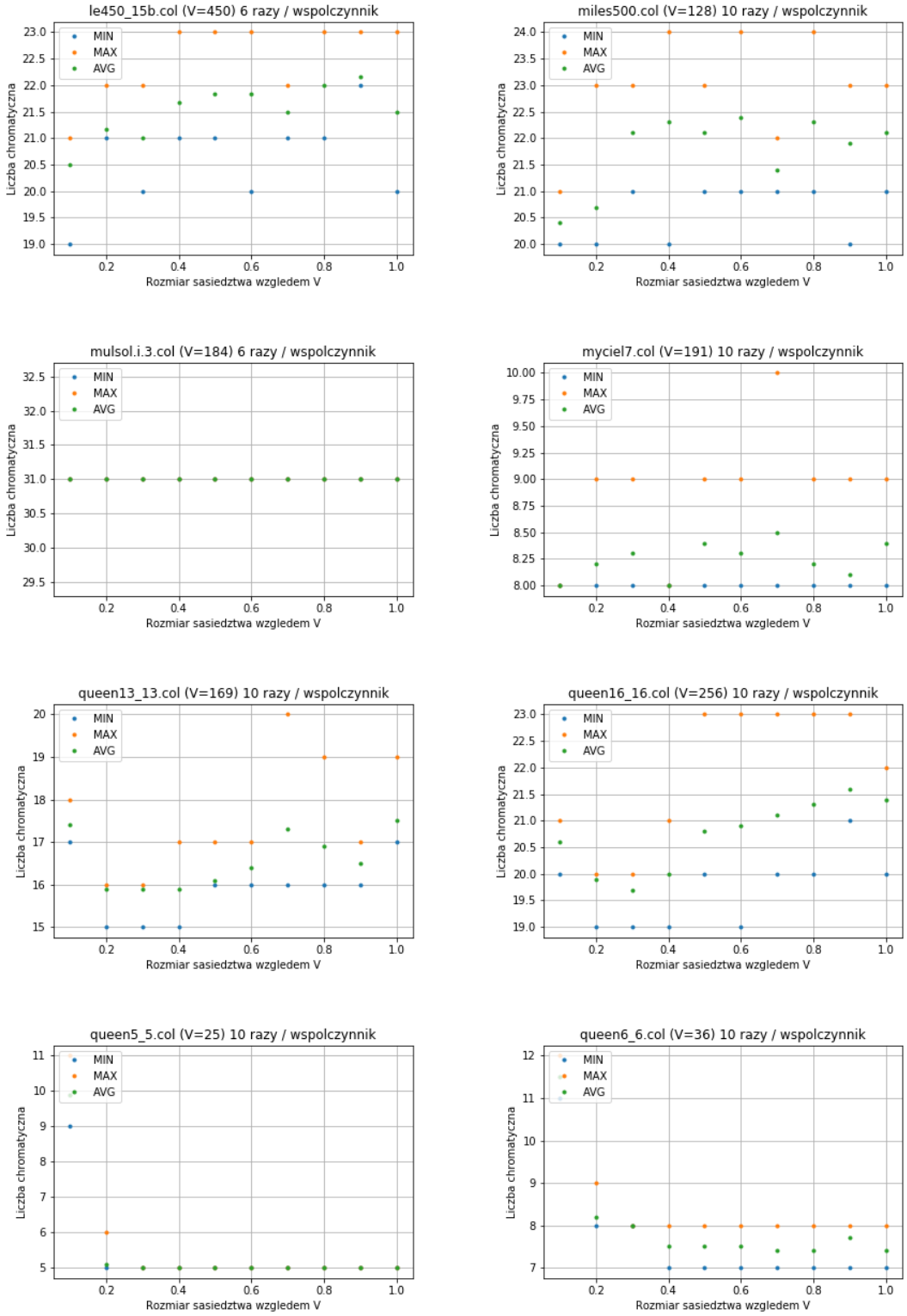
Badaniom podlegało zachowanie algorytmu względem rozmiaru sąsiedztwa oraz rozmiaru listy tabu. Celem badania było sprawdzenie, czy rozmiar sąsiedztwa i długość listy tabu zależą silnie od charakterystyki konkretnego grafu. Poszukiwany był taki zestaw parametrów, który byłby na tyle ogólny, by dało się go zastosować do dowolnego grafu, którego charakterystyka nie jest znana.

5.2.1 Rozmiar sąsiedztwa

Na rysunkach (2) oraz (3) przedstawiono badanie rozmiaru sąsiedztwa. Aby sprawdzić, jak rozmiar sąsiedztwa wpływa na działanie algorytmu, z różnych rodzin grafów wybrano po kilku przedstawicieli. Rozmiar sąsiedztwa był wyznaczany jako iloczyn liczby wierzchołków i pewnego współczynnika. Wartości tego współczynnika zmieniały się w zakresie $< 0.1, 1.0 >$, z krokiem równym 0.1. Na każdy współczynnik przyjęto 10 lub 6 uruchomień algorytmu. Otrzymywana liczba kolorów była zapisywana do logu, by następnie wyznaczyć minimalną, maksymalną i średnią liczbę kolorów uzyskaną przy danym współczynniku. Długość listy tabu pozostawała niezmienna, natomiast początkowa liczba kolorów stanowiła około 0.4 liczby wierzchołków.



Rysunek 2: Badanie rozmiaru sąsiedztwa



Rysunek 3: Badanie rozmiaru sąsiedztwa c.d.

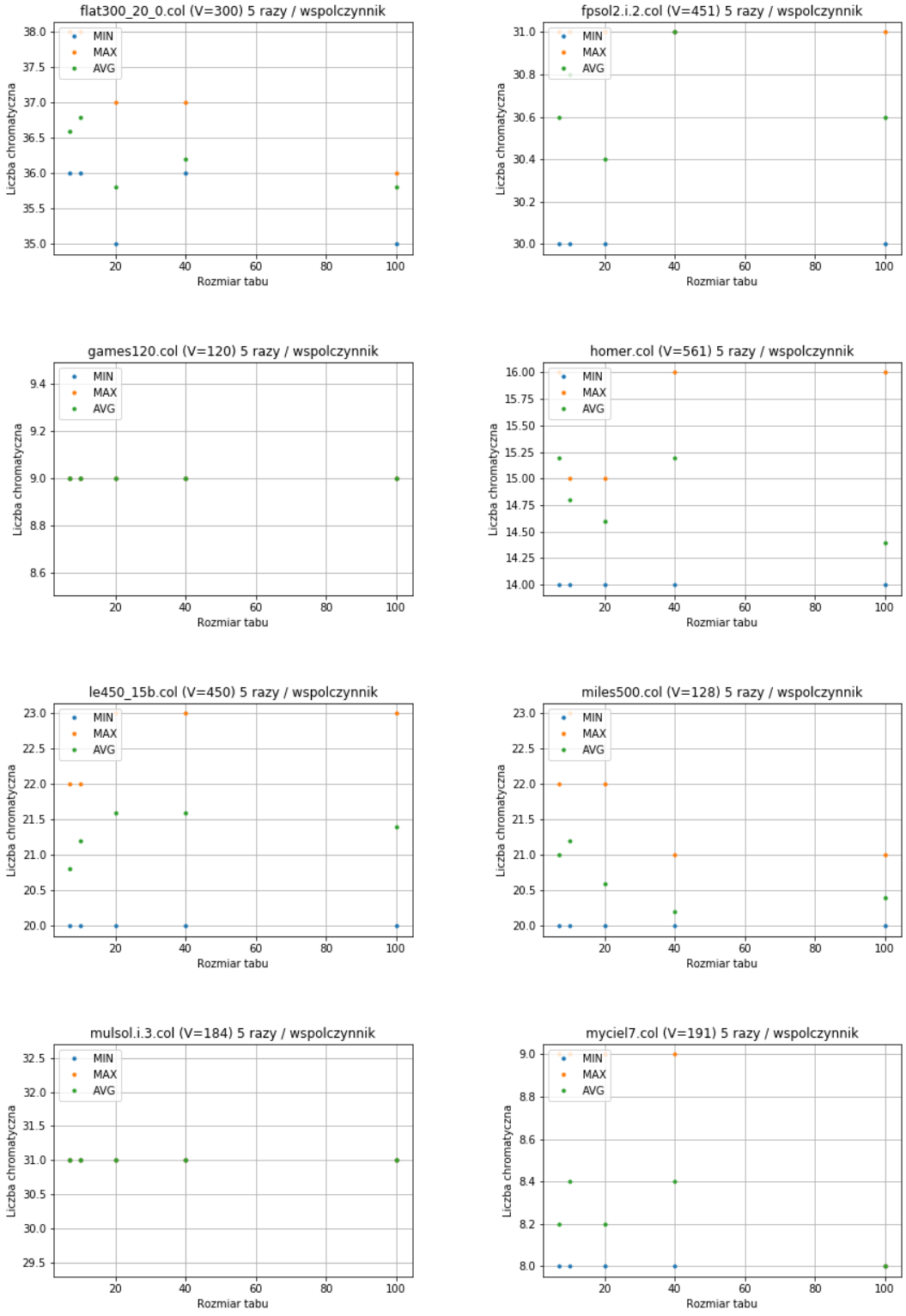
Zgodnie z przewidywaniami, w większości przypadków rozmiar sąsiedztwa miał duży wpływ na zwracaną liczbę kolorów. Jak można zaobserwować na wykresach, usilne zwiększanie rozmiaru nie owocowało znaczącą poprawą rozwiązania. Przeciwnie, w wielu przypadkach zarówno średnia jak i minimalna liczba kolorów stawały się coraz gorsze wraz ze wzrostem sąsiedztwa. Z analizy wyników można odczytać, że rozmiary sąsiedztwa nie przekraczające wartości $0.5V$ sprawdzały się najlepiej. Należy jednak zauważyć, że optymalny współczynnik, dający najmniejszą średnią liczbę kolorów jest mocno zależny od konkretnego przypadku grafu.

Na podstawie wyników eksperymentu, w trakcie wykonywania poszukiwań liczby chromatycznej, opisanych w kolejnych rozdziałach, jako optymalny rozmiar sąsiedztwa przyjęto wartość około 0.5. Potwierdzenie tej wartości można też odnaleźć w [5].

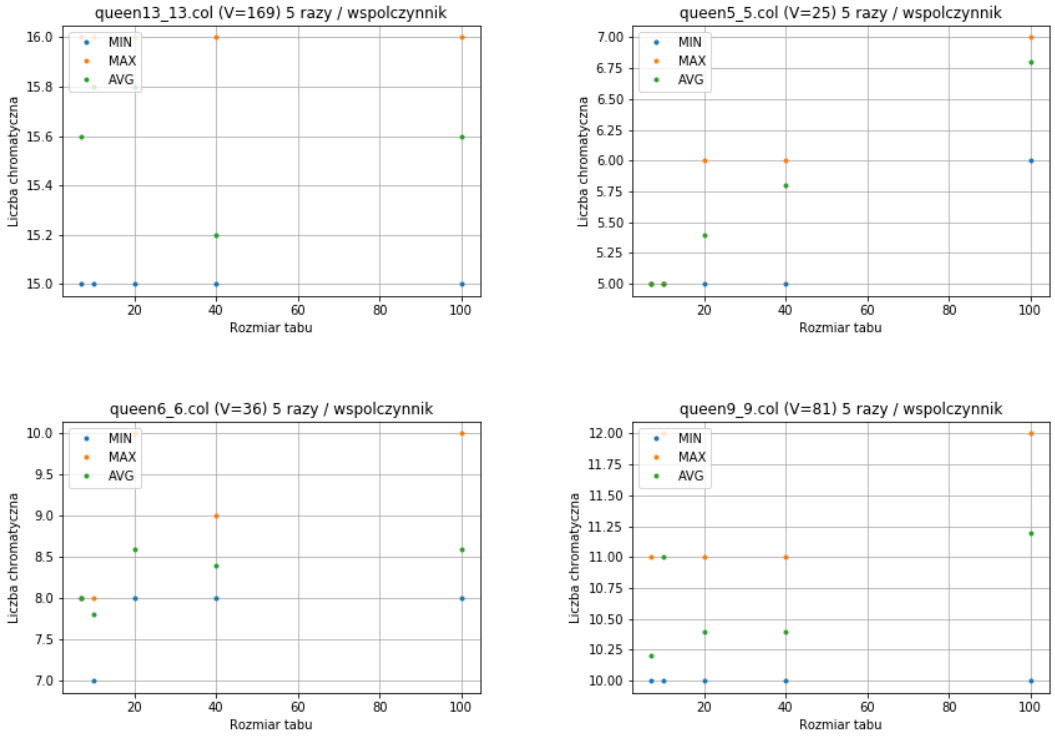
Warto również wspomnieć, że rozmiar sąsiedztwa bardzo silnie wpływa na czas wykonania programu. Z tego właśnie powodu, rozpoczynanie poszukiwań z dużym sąsiedztwem może okazać się nieopłacalne.

5.2.2 Badanie rozmiaru listy tabu

Badanie wpływu rozmiaru listy tabu przebiegało analogicznie do badania rozmiaru sąsiedztwa. Podobnie jak poprzednio, dla różnych rodzajów grafów uruchamiano algorytm z kilkoma długościami listy tabu. Długości te pochodziły ze zbioru $\{7, 10, 20, 40, 100\}$. Dla każdej długości tabu uruchomiono algorytm 5 razy, za każdym razem logując zwracaną liczbę kolorów. Wyniki w postaci minimalnej, maksymalnej i średniej liczby kolorów zaprezentowano na wykresach widocznych na rysunkach (4) oraz (5). Pozostałe parametry, tj. początkowa liczba kolorów i rozmiar sąsiedztwa ustalono odpowiednio na $0.4V$ i $0.5V$.



Rysunek 4: Badanie rozmiaru tabu



Rysunek 5: Badanie rozmiaru tabu c.d.

Podobnie jak w przypadku rozmiaru sąsiedztwa, długość listy tabu w większości przypadków miała wpływ na zwracaną liczbę kolorów. Analiza wyników wskazuje, że krótsze listy tabu generują lepsze rezultaty. Obserwując wszystkie wykresy można zauważyć, że tabu o długości 7 niemal w każdym wypadku zwróciło minimalną liczbę kolorów. Wartość ta jest również postulowana w [5]. Z tego powodu, do wyznaczania liczby chromatycznej w kolejnych rozdziałach przyjęto listę tabu o długości 7.

5.2.3 Początkowa liczba kolorów

Ważnym elementem algorytmu jest początkowa liczba kolorów. Oczywistym jest, że aby algorytm odnalazł kolorowanie bez konfliktów, liczba ta nie może być mniejsza od rzeczywistej liczby chromatycznej grafu. Zazwyczaj jednak nie posiadamy dobrego oszacowania tej liczby. Testy algorytmu wykazały jednak dużą odporność na początkową liczbę kolorów tj. nawet startując od liczby kolorów wprost równej liczbie wierzchołków, algorytm jest w stanie znacząco zredukować liczbę użytych kolorów.

W przypadku rozważanych tutaj grafów, liczby chromatyczne były dokładnie znane. Z tego też powodu, dla celu automatyzacji przeprowadzanych testów, przyjęto bezpieczną wartość $0.4V$, która w każdym przypadku była większa od liczby chromatycznej.

Należy tutaj zauważyć, że początkowa liczba kolorów ma istotne znaczenie dla zmiany ciężaru pomiędzy eksploracją i eksploatacją przestrzeni rozwiązań. Duża liczba możliwych kolorów będzie wywierać presję w stronę eksploracji. Wynika to ze sposobu generowania sąsiedniego rozwiązania, które w swoim ostatnim kroku przypisuje wierzchołkowi losowy kolor z zakresu 1 do k.

Temat balansu pomiędzy eksploracją i eksploatacją oraz dobierania początkowej liczby k zostanie jeszcze omówiony w rozdziałach prezentujących wyniki poszukiwań liczby chromatycznej dla testowanych grafów.

5.3 Wyniki poszukiwań liczby chromatycznej

Jak opisano w poprzednich rozdziałach, na potrzeby testów wykorzystano zbiór grafów przygotowanych dla konkursu DIMACS. Aby ocenić poprawność przygotowanej implementacji, dokonano porównania z wynikami klasycznych algorytmów (m.in. DSATUR i RLF) przedstawionymi w artykule [2]. W artykule tym również wykorzystano grafy z konkursu DIMACS. Zbiór testowanych grafów został podzielony na kilka grup, różniących się charakterystykami :

- Grafy Mycielskiego i SGB - w tej grupie można znaleźć klasyczne grafy Mycielskiego oraz grafy ze Stanford GraphBase (SGB), reprezentujące m.in. połączenia między miastami w USA (grafy *miles*) czy też relacje pomiędzy postaciami danej książki (grafy *david*, *jean*, *anna*, *homer*, *huck*),

- grafy queens - grafy w których wierzchołki reprezentują pola szachownicy, natomiast krawędzie dodawane są tylko wtedy, gdy królowe postawione na polach odpowiadających wierzchołkom atakują się wzajemnie,

- grafy CAR - grafy inspirowane grafami Mycielskiego, jednak trudniejsze w kolorowaniu,

- grafy losowe i flat - trudne algorytmicznie grafy losowe stworzone przez Davida Johnsone'a, oraz grafy flat zaproponowane przez Culbersone'a,

- grafy RAC - reprezentują rzeczywisty problem alokacji zmiennych z kodu pośredniego do rejestrów procesora przez kompilator,

- grafy Leighton - każdy z nich składa się z 450 wierzchołków. Cechą charakterystyczną jest zmieniająca się liczba chromatyczna przy niezmiennej gęstości.

Dla każdego grafu opisanego w artykule uruchamiano algorytm od 10 do 15 razy, z następującymi parametrami:

- $c = 0.4V$
- $n = 0.5V$
- $t = 7$
- $$i = \begin{cases} 250000, & |V| < 500 \\ 500000, & |V| >= 500 \end{cases} \quad (2)$$

Następujące tabele prezentują najlepsze uzyskane wyniki, porównane z rzeczywistą liczbą chromatyczną oraz wynikami algorytmów opisanymi w artykule.

Graf	V	E	Faktyczne χ	Najlepsze χ z artykułu	**Tabu** χ
queen5_5	25	160	5	5	**5**
queen6_6	36	290	7	8	**7**
queen7_7	49	476	7	9	**7**
queen8_12	96	1368	12	13	**12**
queen8_8	64	728	9	11	**9**
queen9_9	81	1056	10	12	**10**
queen10_10	100	2940	11	13	**11**
queen11_11	121	3960	11	14	**12**
queen12_12	144	5192	13	15	**14**
queen13_13	169	6656	13	16	**15**
queen14_14	196	8372	16	17	**17**

Rysunek 6: Pomiary dla grafów typu queens

Graf	V	E	Faktyczne χ	Najlepsze χ z artykułu	**Tabu** χ
le450_15b	450	8169	15	16	**18**
le450_25a	450	8260	25	25	**26**
le450_25b	450	8263	25	25	**26**
le450_25c	450	17343	25	28	**30**
le450_5c	450	9803	5	5	**5**
le450_5d	450	9757	5	6	**5**

Rysunek 7: Pomiary dla grafów Leightona

Graf	V	E	Faktyczne χ	Najlepsze χ z artykułu	**Tabu** χ
myciel3	11	20	4	4	**4**
myciel4	23	71	5	5	**5**
myciel5	47	236	6	6	**6**
myciel6	95	755	7	7	**7**
myciel7	191	2360	8	8	**8**
miles1000	128	3216	42	42	**42**
miles1500	128	5198	73	73	**73**
miles500	128	1170	20	20	**21**
miles750	128	2113	31	31	**32**
anna	138	493	11	11	**11**
david	87	406	11	11	**11**
homer	561	1629	13	13	**13**
huck	74	301	11	11	**11**
jean	80	254	10	10	**10**
games120	120	638	9	9	**9**

Rysunek 8: Pomiary dla grafów Mycielskiego

Graf	V	E	Faktyczne χ	Najlepsze χ z artykułu	**Tabu** χ
fpsol2.i.1	496	11654	65	65	**65**
fpsol2.i.2	451	8691	30	30	**30**
fpsol2.i.3	425	8688	30	30	**30**
mulsol.i.1	197	3925	49	49	**49**
mulsol.i.2	188	3885	31	31	**31**
mulsol.i.3	184	3916	31	31	**31**
mulsol.i.4	185	3946	31	31	**31**
mulsol.i.5	186	3973	31	31	**31**
inithx.i.1	864	18707	54	54	**54**
inithx.i.2	645	13979	31	31	**31**
inithx.i.3	621	13969	31	31	**31**
zeroin.i.1	211	4100	49	49	**49**
zeroin.i.2	211	3541	30	30	**30**
zeroin.i.3	206	3540	30	30	**30**

Rysunek 9: Pomiary dla grafów rac

Graf	V	E	Faktyczne χ	Najlepsze χ z artykułu	**Tabu** χ
DSJC125.1	125	736	5	6	**6**
DSJC125.5	125	3891	17	21	**18**
DSJC125.9	125	6961	44	49	**49**
DSJC250.1	250	3218	8	10	**11**
DSJC250.5	250	15668	28	35	**32**
DSJR500.1	500	3555	12	12	**14**
r250.5	250	14849	65	68	**72**
flat300_20_0	300	21375	20	38	**35**
flat300_26_0	300	21633	26	39	**36**
flat300_28_0	300	21695	28	38	**35**

Rysunek 10: Pomiary dla grafów losowych

Graf	V	E	Faktyczne χ	Najlepsze χ z artykułu	**Tabu** χ
1-FullIns_4.col	93	593	5	5	**5**
1-FullIns_5.col	282	3247	6	6	**6**
1-Insertions_4.col	67	232	5	5	**5**
1-Insertions_5.col	202	1227	6	6	**6**
1-Insertions_6.col	607	6337	7	7	**8**
2-FullIns_3.col	52	201	5	5	**5**
2-FullIns_4.col	212	1621	6	6	**7**
2-FullIns_5.col	852	12201	7	7	**9**
2-Insertions_4.col	149	541	5	5	**5**
2-Insertions_5.col	597	3936	6	6	**6**
3-FullIns_3.col	80	346	6	6	**6**
3-FullIns_4.col	405	3524	7	7	**8**
3-FullIns_5.col	2030	33751	8	8	**11**
3-Insertions_3.col	56	110	4	4	**4**
3-Insertions_4.col	281	1046	5	5	**5**
3-Insertions_5.col	1406	9695	6	6	**7**
4-FullIns_3.col	114	541	7	7	**7**
4-FullIns_4.col	690	6650	8	8	**9**
4-FullIns_5.col	4146	77305	9	10	**13**
4-Insertions_3.col	79	156	4	4	**4**
4-Insertions_4.col	475	1795	5	5	**5**
5-FullIns_3.col	154	792	8	8	**8**
5-FullIns_4.col	1085	11395	9	9	**10**

Rysunek 11: Pomiary dla grafów car

5.4 Wnioski

Jak wynika z analizy wyników, algorytm uzyskał bardzo dobre wyniki dla szerokiej gamy grafów. W większości przypadków był on w stanie znaleźć rzeczywistą liczbę chromatyczną lub liczbę bardzo jej bliską. Możemy to zaobserwować w przypadku grafów *queens, rac, car* a także grafów Mycielskiego i SGB.

Największe rozbieżności można zaobserwować w grupie grafów losowych. Grafy te wykazują szczególną trudność, jednakże i w tym przypadku przeszukiwanie z tabu okazało się lepsze od klasycznych algorytmów takich jak DSATUR czy RLF. Wyjątkiem może tu być graf R250.5, w którym uzyskany wynik jest gorszy od wyników zaprezentowanych w artykule.

Nieco słabsze wyniki można zaobserwować również w przypadku grafów Leightona. Należy jednak pamiętać, że parametry algorytmu nie były dostosowane

pod konkretny graf. Analiza dwóch ostatnich grafów leigthona tj. *le450_5c* oraz *le450_5d* wykazała, że da się poprawić wynik algorytmu tak, by osiągnął rzeczywistą liczbę chromatyczną równą 5. Należało w tym celu zmniejszyć liczbę kolorów k do 5, podnieść rozmiar sąsiedztwa oraz liczbę iteracji.

5.4.1 Dobór parametrów

Analiza otrzymanych wyników wykazała, że choć przyjęte w testach wartości parametrów pozwalają na otrzymanie dość dobrego rezultatu, w wielu przypadkach można poprawić otrzymany wynik, uzyskując rzeczywistą liczbę chromatyczną. Wymaga to jednak manipulacji parametrami.

Jednym z najważniejszych parametrów jest liczba kolorów. Wpływa ona bardzo mocno na balans pomiędzy eksploracją i eksploatacją przestrzeni rozwiązań. W początkowym stadium działania algorytmu liczba ta może być znaczna, jednak powinna się zmniejszać wraz z poprawą rozwiązania, aby wywierać większą presję na eksploatację przestrzeni wokół optimum. W przygotowanej implementacji jednak liczba ta pozostaje stała w trakcie działania algorytmu. Powoduje to, że wraz z poprawą liczby kolorów, coraz więcej klas pozostaje pustych. Zwiększa się więc prawdopodobieństwo wygenerowania w sąsiedztwie ruchu, który nigdy nie zostanie przyjęty, ponieważ będzie usiłował rozbić liczną klasę kolorowania.

Z tego powodu warto, by początkowa liczba k nie odbiegała znacznie od rzeczywistej liczby chromatycznej. Zazwyczaj jednak nie dysponujemy dobrym oszacowaniem tej liczby. Możemy jednak użyć do tego samego algorytmu. Można wtedy wyróżnić 2 fazy:

- Faza eksploracji - uruchamiamy algorytm rozpoczynając od liczby k równej liczbie wierzchołków. Ograniczamy znacznie sąsiedztwo i liczbę iteracji. Ponieważ funkcja oceny premiuje liczne klasy, algorytm bardzo szybko ograniczy liczbę kolorów. Po otrzymaniu wyniku bez konfliktów uruchamiamy algorytm na niewielką liczbę iteracji kolejny raz, przyjmując k równe otrzymanemu w poprzednim uruchomieniu. Postępujemy tak aż do pojawienia się kolorowania z konfliktami.

- Faza eksploatacji - uruchamiamy algorytm rozpoczynając od liczby k, która zawierała konflikty lub liczby nieznacznie większej. Zwiększamy znacznie rozmiar sąsiedztwa np. ponad $0.5V$ oraz podnosimy liczbę iteracji.

6 Errata do sprawozdania 2

Pseudokod przedstawiony w rozdziale 3.2 powinien być uzupełniony o *kryterium aspiracji*. Ruch będący ruchem tabu powinien być wybrany jako najlepszy, jeżeli

prowadzi do optymalizacji globalnego optimum. W rzeczywistej implementacji zostało to uwzględnione i wszystkie testy zostały wykonane z poprawnie zaimplementowanym kryterium aspiracji.

W rozdziale 4.3 wspomniano o porównaniu czasu działania naszego algorytmu z wynikami przedstawionymi w artykule. Nie dokonaliśmy takiego porównania, ponieważ nie mieliśmy dostępu do implementacji algorytmów opisanych w artykule, a zatem niemożliwe było miarodajne porównanie czasu działania algorytmów. W naszych eksperymentach skupiliśmy się jedynie na porównaniu uzyskanych wartości liczby chromatycznej.

Literatura

[1] J. Arabas. Wykład z przedmiotu Algorytmy Heurystyczne. `https://elektron.elka.pw.edu.pl/~jarabas/ALHE/wyklad5.pdf`, 2019. Politechnika Warszawska.

[2] Murat Aslan and Nurdan Baykan. A performance comparison of graph coloring algorithms. *International Journal of Intelligent Systems and Applications in Engineering*, 4:1–1, 12 2016.

[3] L. A. McGeoch C. Schevon D. S. Johnson, C. R. Aragon. Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 1991.

[4] P. Erdös and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290, 1959.

[5] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, Dec 1987.

[6] David J. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, Boston, MA, USA, 1996.