Wydział Matematyki i Nauk Informacyjnych
POLITECHNIKA WARSZAWSKA

Algorytmika problemów trudnych obliczeniowo

3-kolorowanie grafu planarnego - raport końcowy

Artur Michalski, Dawid Maksymowski, Arkadiusz Noster
opiekun: dr Paweł Rzążewski

13 czerwca 2021

Spis treści

1 Sformułowanie problemu

Problem kolorowania wierzchołkowego polega na znalezieniu odwzorowania przypisującego każdemu z wierzchołków kolor w taki sposób, aby sąsiadujące ze sobą wierzchołki otrzymały różne kolory. Jest to w ogólności problem trudny obliczeniowo. Sama decyzja czy graf może zostać pokolorowany przy użyciu k kolorów (tzn. jest k-kolorowalny) należy do klasy problemów NP-zupełnych (poza przypadkami 1- i 2-kolorowalności). Niniejszy dokument przedstawia algorytm 3-kolorowania grafu planarnego.

1.1 Rozwiązanie BruteForce

Jednym ze sposobów na znalezienie 3-kolorowania grafu (o ile istnieje) jest rozwiązanie brute-force. Idea algorytmu opiera się na próbie rozpatrzenia wszystkich możliwych kolorowań grafu wykorzystując 3 kolory. Oczywiście jest to rozwiązanie poprawne. Jeśli graf jest 3-kolorowalny to z pewnością przeglądając wszystkie możliwe 3-kolorowania znajdziemy poprawne kolorowanie. Problemem jest jednakże złożoność czasowa takiego rozwiązania. Spróbujmy określić liczbę wszystkich możliwych kolorowań. Naturalnie jest ich 3^n. Dla każdego z wierzchołków rozpatrujemy pokolorowanie go jednym z 3 kolorów, a decyzję taką podejmujemy n razy. Dodatkowo dla każdego z rozpatrywanych kolorowań grafu należy sprawdzić czy jest ono poprawne. Rozwiązaniem może być na przykład iteracja po wszystkich krawędziach grafu i sprawdzenie czy dla żadnej krawędzi wierzchołki z nią incydente nie zostały pokolorowane tym samym kolorem. Złożoność czasowa rozwiązania brute-force wynosi, więc $\mathcal{O}^*(3^n)$.

1.2 Rozwiązanie Divide And Conquer

Rozwiązanie brute-force, choć znajduje poprawne 3-kolorowanie (o ile istnieje), nie wydaje się być optymalne pod względem złożoności czasowej. Z tego powodu prezentujemy rozwiązanie pozwalające na znajdowanie 3-kolorowania w oczekiwanym czasie $\mathcal{O}^*\left(3^{\mathcal{O}(\sqrt{n})}\right)$.

Redukcja złożoności jest możliwa dzięki wykorzystaniu metody *Divide and Conquer*. Takie podejście wymaga od nas rozbicia problemu na mniejsze – w tym celu wykorzystujemy separator planarny. Bez straty ogólności możemy założyć (rozumowanie można uogólnić na większą ilość składowych), że separator S dzieli graf na składowe A i B. Zauważmy, że wtedy kolorowania składowych A i B nie są od siebie zależne. Wynika to wprost z definicji S i kolorowania. Kolorowania A i B zależą więc wyłącznie od tego w jaki sposób zostanie pokolorowany separator. Analizując więc wszystkie możliwe kolorowania separatora (analogicznie do metody brute-force) co najmniej jedno z nich odpowiada kolorowaniu wierzchołków z separatora w jednym z poprawnych 3-kolorowań całego grafu (o ile istnieje). Po pokolorowaniu separatora otrzymujemy więc dwa wzajemnie niezależne problemy kolorowania składowych A oraz B.

2 Szkic algorytmu

Algorytm jest rekurencyjny. W każdym kroku znajdujemy separator planarny S aktualnie rozpatrywanego grafu G. Następnie metodą brute-force rozpatrujemy wszystkie możliwe kolorowania wyznaczonego separatora. Dla każdego poprawnego kolorowania wywołujemy rekurencyjnie powyższą procedurę dla każdej z nowo powstałych składowych grafu G. Oczywiście kolorowanie separatora ma wpływ na kolorowanie pozostałych składowych grafu. Informacja o tym, jak zostały pokolorowane separatory w aktualnie rozpatrywanym kolorowaniu jest przechowywana poprzez listy kolorów. Dla każdego z wierzchołków będącego sąsiadem wierzchołka z separatora z listy kolorów zabierany jest kolor jakim został pokolorowany jego sąsiad w separatorze. W przypadku, gdy dla dowolnego jeszcze nie pokolorowanego wierzchołka zostanie stwierdzony brak dostępnych kolorów, rozpatrywane jest kolejne kolorowanie. Algorytm kończy pracę, gdy zgodnie z listami kolorów zostaną prawidłowo pokolorowane wszystkie wierzchołki (tj. zostanie znalezione poprawne 3-kolorowanie) lub gdy dla dowolnego z separatorów żadne z kolorowań nie pozwoli na poprawne kolorowanie pozostałego grafu (dla dowolnego separatora zakończy się iteracja po wszystkich możliwych kolorowaniach).

3 Trudności napotkane przy implementacji algorytmu

Podczas implementacji algorytmu pojawiły się następujące komplikacje:

- Stwierdzenie, ile wierzchołków znajduje się wewnątrz cyklu fundamentalnego
- Triangulacja otrzymanego drzewa BFS w separatorze

Rozwiązania powyższych kwestii zostały szczegółowo opisane w kolejnej sekcji raportu.

4 Modyfikacje względem pierwotnej dokumentacji

4.1 Triangulacja

Niezbędnym do implementacji trzeciej fazy algorytmu znajdowania separatora jest algorytm triangulizacji drzewa BFS stworzonego w poprzednich etapach `FindSeparator`. W ufności informacjom odnalezionym w literaturze, nasz zespół był przekonany o łatwej, powszechnej dostępności prostego algorytmu triangulizującego, który działałby w czasie liniowym. Algorytmy takie rzeczywiście istnieją, wymagają jednak 2-spójności grafu wejściowego. Drzewa BFS oczywiście tego wymagania nie spełniają. Ogólny problem „u-dwóspójnienia" dowolnego grafu wymaga zgłębienia się w **drzewa blokowe** oraz **drzewa SQPR** ([3] – rozdział 6 oraz 4), co ze względu na ograniczenia czasowe oraz obszerność tego tematu wybiega poza niniejszy projekt.

Okazuje się jednak, że mając do dyspozycji **zanurzenie kombinatoryczne** grafu, możliwe jest odnalezienie jego triangulacji bardzo prostym algorytmem:

```
1  in: graph G, embedding E of G
2  out: triangulated G
3  ----------------------
4  for each vertex v in G:
5      for each pair of consecutive neighbors (u, w) of v from E:
6          if G does not contain edge (u, w):
7              add edge (u, w) to G and E
```

4.1.1 Zanurzenie kombinatoryczne drzewa

Zanurzeniem kombinatorycznym dowolnego grafu G nazywamy taką strukturę, reprezentującą zanurzenie G w płaszczyźnie, która dla każdego wierzchołka należącego do tego grafu przechowuje listę wierzchołków incydentnych do niego w zadanej z góry kolejności (np. przeciwnej do ruchu wskazówek zegara).

Warto zauważyć, że stworzenie zanurzenia dla drzewa rozpinającego jest zazwyczaj bardzo proste, jeżeli zostało ono zbudowane metodą BFS. Wówczas tworząc zanurzenie, wystarczy dla każdego wierzchołka dodawać jego sąsiadów do listy w takiej kolejności, w jakiej wierzchołki są przechowywane wewnętrznie – najpierw zostanie dodany *rodzic*, a następnie kolejno wszystkie jego *dzieci*. Stąd wynikają 2 implementacje interfejsu `ITriangulation`: `InternalTriangulation` oraz `BFSTriangulation`. Obie implementacje posiadają złożoność liniową (`InternalTriangulation` z lepszym współczynnikiem).

4.1.2 Dodanie krawędzi do zanurzenia

O ile dodanie krawędzi do grafu możliwe jest do zrealizowania w czasie stałym, tak dodanie jej do zanurzenia nie jest zadaniem trywialnym i może wymagać dłuższego czasu. Najprostszą metodą jest dodawanie krawędzi zawsze w ten sam sposób. Mając do dyspozycji taki fragment zanurzenia `E`:

```
1  ...
2  E[v] := [a1 ... u  w ... ak]
3  E[u] := [b1 ... bi v ... bm]
4  E[w] := [c1 ... v cj ... cn]
5  ...
```

i chcąc dodać krawędź `u-w`, należy zmodyfikować `E` w następujący sposób:

```
1  ...
2  E[v] := [a1 ... u  w ... ak]
3  E[u] := [b1 ... bi w v ... bm]
4  E[w] := [c1 ... v u cj ... cn]
5  ...
```

Słabym punktem algorytmu może wydawać się znalezienie pozycji wierzchołka v na listach $E[u]$ i $E[w]$. W pesymistycznym przypadku przeszukanie to jest liniowe, co przekłada się na asymptotyczną złożoność algorytmu $\mathcal{O}(n^2)$. Można się jednak spodziewać (z własności rzadkości grafów planarnych), że stała ukryta wewnątrz tej złożoności jest niewielka. Ponadto w perspektywie całości implementacji różnica między złożonością liniową oraz kwadratową jest niezauważalna.

4.2 Separator - faza III

Modyfikacji została poddana również faza trzecia (złożona) algorytmu znajdowania separatora. W pierwszym kroku dokonywana jest triangulizacja otrzymanego we wcześniejszych fazach algorytmu drzewa BFS. Następnie wybierana jest arbitralnie jedna z krawędzi otrzymanych podczas triangulacji- oznaczmy ją przez e. Zauważmy, że wierzchołki incydentne z e indukują ścieżki w stronę korzenia drzewa, które spotykają się w jednym z wierzchołków (w pesymistycznym wariancie jest to korzeń drzewa). Rozpatrywane ścieżki wraz z krawędzią e tworzą więc *cykl fundamentalny*. Rozpatrywany cykl indukuje podział pozostałych wierzchołków (nie należących do cyklu) na część *wewnętrzną* oraz *zewnętrzną* stanowiące 2 (lub więcej) odrębne składowe.

Algorytm rozpatruje kolejne krawędzie otrzymane z triangulacji tak długo, jak liczność wierzchołków części wewnętrznej i zewnętrznej przekracza $\frac{2n}{3}$, gdzie n – liczba wierzchołków grafu wejściowego. Aby zweryfikować warunek stopu należy znać liczność części wewnętrznej lub zewnętrznej. Problem ten jest o tyle trudny, że położenie wierzchołków względem rozpatrywanego cyklu jak i właściwa triangulacja zależy bezpośrednio od konkretnego zanurzenia grafu.

4.2.1 Liczba wierzchołków wew. cyklu

Do ustalenia liczby wierzchołków wewnątrz cyklu wykorzystano rozważania oparte o graf dualny do znalezionej triangulacji, a przeniesione na następujące fakty:
Rozważmy ogólny przypadek dodawania krawędzi w algorytmie przedstawionym w 4.1. Niech $V_{wew}(a, b)$ oznacza liczbę wierzchołków wewnętrznych cyklu fundamentalnego indukowanego przez krawędź (a, b). Algorytm, próbując dodać do grafu krawędź (u, w), gdzie u, w są kolejnymi sąsiadami wierzchołka v, może łatwo sprawdzić, która z następujących sytuacji jest prawdziwa:

- obie krawędzie (u, v), (v, w) są krawędziami drzewa wejściowego (nie zostały dodane w trakcie działania algorytmu) – (u, w) są dziećmi v
  $V_{wew}(u, w) = 0$„
- tylko jedna z krawędzi (u, v), (v, w) (ozn. e) jest krawędzią drzewa wejściowego
  $V_{wew}(u, w) = V_{wew}(e)$,
- żadna z krawędzi (u, v), (v, w) nie jest krawędzią drzewa wejściowego (obie zostały dodane w trakcie działania algorytmu)
  $V_{wew}(u, w) = V_{wew}(u, v) + V_{wew}(v, w) + 1$.

Dzięki temu, wykorzystując odpowiednią strukturę danych, możliwe jest bardzo szybkie ustalenie, ile wierzchołków znajdzie się wewnątrz cyklu dla każdej rozpatrywanej krawędzi triangulacji. Algorytm polega na iteracji po wszystkich takich krawędziach, wyznaczeniu indukowanego cyklu fundamentalnego, a następnie sprawdzeniu, czy otrzymane w wyniku podziału składowe spełniają górne ograniczenie na liczność zbioru wierzchołków.

4.2.2 Długość cyklu fundamentalnego

W aktualnej implementacji długość każdego cyklu fundamentalnego jest obliczana indywidualnie. Rozważając podgraf grafu dualnego do triangulacji, składający się tylko z tych krawędzi, które

odpowiadają krawędziom triangulacji spoza grafu wejściowego (podgraf ten okazuje się być drzewem) intuicja podpowiada, że również długość cyklu można obliczyć krokowo, opierając się na obliczeniach z krawędzi dodanych wcześniej. Intuicje te potwierdzone są przez [2] (rozdział 2.2). Wymagałoby to nieznacznego zwiększenia złożoności pamięciowej, natomiast uzyskanie długości cyklu możliwe byłoby do uzyskania w czasie stałym.

4.2.3 Liczba rozpatrywanych krawędzi

Nie wszystkie krawędzie dodane do drzewa warto rozpatrywać. Długości cykli fundamentalnych oraz liczby wierzchołków wewnątrz nich może być identyczna dla wielu z krawędzi. Biorąc jako przykład drzewa będące gwiazdą: jego triangulacja posiada wiele krawędzi, które indukują cykle o identycznej długości (= 3) oraz posiadają identyczną liczbę wierzchołków wewnętrznych (= 0). Można rozpatrywać tylko jedną krawędź z każdej takiej rodziny krawędzi.

4.3 Implementacja właściwej rekurencji

Pewnej optymalizacji i refaktoryzacji została również poddana część stanowiąca główny punkt algorytmu - rekurencyjne kolorowanie separatora.

- Odseparowano kolorowanie kolejnych wierzchołków separatora od wyznaczania separatorów i próby kolorowania kolejnych składowych,
- zmieniono liczbę wierzchołków dla której algorytm koloruje składową metodą brute-force z 5 (minimalna liczba) na 15 (wyznaczona eksperymentalnie),
- wprowadzono dodatkową strukturę danych `verticesWithTakenColor` w postaci listy wierzchołków, które właśnie straciły kolor w wyniku aktualnego pokolorowania *sąsiada*.

W przypadku, gdy rozpatrywany graf jest 3-kolorowalny, informacja o jednym z możliwych poprawnych kolorowań jest zwracana poprzez tablicę o długości liczności zbioru wierzchołków grafu indeksowaną numerem wierzchołka, której elementy mówią o kolorowaniu i-tego wierzchołka. Jeśli natomiast kolorowanie nie istnieje, zwracany jest `null`. W celu zapobiegnięcia ryzyku modyfikacji grafu wejściowego wszystkie działania wykonywane są na przygotowanej na początku wywołania funkcji kopii grafu.

5 Zastosowane optymalizacje

5.1 Wycofywanie się z niepoprawnych kolorowań

W trakcie implementacji zaobserwowano zachowanie wpływające negatywnie na czas działania algorytmu. Rozważmy sytuację, w której zdołaliśmy pokolorować już wszystkie wierzchołki należące do separatora i przystępujemy do kolorowania otrzymanych na wejściu składowych. Załóżmy, że dla n-tej rozpatrywanej składowej nie udało się znaleźć poprawnego kolorowania. Algorytm wraca więc do kolorowania wierzchołków należących do separatora, przechodzi do kolejnego poprawnego kolorowania (o ile istnieje) i próbuje ponownie pokolorować składowe. Rozpatrzmy przypadek, gdy zmienione kolorowanie nie wpłynęło w żaden sposób na kolorowanie składowej n (tj. wierzchołki należące do separatora, które w aktualnie rozpatrywanym kolorowaniu otrzymały nowy kolor nie należą do bezpośredniego sąsiedztwa wierzchołków należących do składowej n). Oczywiście aktualnie rozpatrywane kolorowanie również skazane jest na niepowodzenie przy próbie znalezienia kolorowania dla składowej n. W celu wyeliminowania opisanego zachowania, przy próbie znalezienia kolorowań składowych zwracamy informacje o indeksie składowej dla której znajdowanie kolorowania nie powiodło się. Następnie wycofujemy się z rozpatrywanego kolorowania separatora tak długo, aż nie zostanie stwierdzone, że dla wierzchołków ze składowej n należących do bezpośredniego sąsiedztwa separatora stało się dostępne nowe kolorowanie. Aby cały proces mógł odbywać się bez istotnego zwiększenia złożoności obliczeniowej przy znajdowaniu składowych grafu po usunięciu wierzchołków należących do separatora, wprowadzono tworzenie dodatkowej struktury danych w postaci słownika. Jej celem jest przechowywanie dostępnej w czasie stałym informacji, do której składowej należy dany wierzchołek.

6 Potencjalne kierunki optymalizacji

6.1 Wycofywanie się z niepoprawnych kolorowań

Optymalizację opisaną w 5.1 można dodatkowo usprawnić. Zamiast cofać się z kolorowaniem separatora aż do napotkania pierwszego wierzchołka należącego do sąsiedztwa $N(A_i)$ feralnej składowej A_i, można zacząć modyfikować kolorowanie *od razu* od wierzchołków należących do $N(A_i)$. Gdyby okazało się, że niemożliwe jest 3-kolorowanie grafu $G[A_i \cup N(A_i)]$ (np. jeśli zawarty jest w nim podgraf K_4), możliwe jest natychmiastowe zakończenie działania programu, zwracając informację o nieistnieniu poprawnego 3-kolorowania.

Wprowadzenie optymalizacji wymaga m.in. wprowadzenia możliwości:

- zmiany kolejności, w jakiej znajdują się wierzchołki w zbiorze separatora,
- odróżnienia powrotu wywołania programu w górę rekurencji (cofanie się kolorowania) od przedwczesnego zakończenia (kolorowanie podgrafu niemożliwe).

6.2 Zrównoleglenie kolorowania składowych grafu

Po pokolorowaniu wszystkich wierzchołków należących do separatora, kolejnym krokiem algorytmu jest próba pokolorowania otrzymanych składowych. Przypomnijmy, że z definicji separatora wynika wprost, że kolorowanie dowolnej składowej nie ma bezpośredniego wpływu na kolorowanie pozostałych składowych. W implementacji można wykorzystać tę właściwość poprzez zrównoleglenie znajdowania kolorowań tych składowych. Jeśli którakolwiek z nich nie może zostać pokolorowana, próba pokolorowania pozostałych jest przerywana. Podczas badania poprawności działania algorytmu, dla jednego przypadku testowego zaobserwowano jednak zwrócenie niepoprawnego wyniku. Wszystkie załączone analizy tej metody powinny być w związku z tym traktowane jedynie jako wskazówka oraz zobrazowanie potencjalnych korzyści wynikających z zastosowania tego podejścia. Należy zwrócić uwagę, że wprowadzenie do algorytmu obliczeń równoległych wprowadza równocześnie całą klasę potencjalnych trudności, które należy rozpatrzyć. Warto jednak wspomnieć o tym potencjalnie znacznym usprawnieniu – w szczególności zauważalnym dla grafów, których separatory rozspójniają je na liczne składowe. Ten rodzaj optymalizacji jest bardzo trudny do zastosowania dla rozwiązania naiwnego opisywanego problemu.

7 Testy

7.1 Generator grafów planarnych

Ze względu na małą dostępność i różnorodność gotowych grafów planarnych w dostępnych zasobach internetowych, został stworzony generator grafów planarnych. Jako, że poszczególne spójne składowe odpowiadają oddzielnym problemom kolorowania, algorytm generuje grafy spójne, by ujednolicić trudność zadania kolorowania.

Algorytm generuje graf kombinatorycznie, w przeciwieństwie do powszechniej używanych algorytmów geometrycznych. Generacja może zostać opisana następującymi krokami:

1. Pobierz gęstość oraz oczekiwaną liczbę wierzchołków jako argument
2. Stwórz graf nieskierowany złożony z dwóch wierzchołków i krawędzi
3. Stwórz listę ścian i dodaj do niej ścianę zewnętrzną
4. Dopóki graf nie posiada oczekiwanej liczby wierzchołków wykonuj następujące kroki:
   (a) Dodaj nowy wierzchołek
   (b) Wylosuj ścianę, do której będzie należeć
   (c) Dla każdego wierzchołka tej ściany wylosuj z prawdopodobieństwem równym argumentowi gęstości, czy utworzony wierzchołek zostanie połączony do niego.

(d) Jeśli żaden wierzchołek nie został wylosowany, przejdź do (b).

(e) Dodaj krawędzie pomiędzy nowym wierzchołkiem i wylosowanymi wierzchołkami. Zaktualizuj listę ścian, tak, by odzwierciedlała nowy stan.

Zarówno manualna analiza, jak i różnorodność wyników pojawiających się przy próbach uruchomienia algorytmów na wygenerowanych grafach wskazują, że powstałe w ten sposób grafy są różnorodne, przez co są dobrymi kandydatami do przeprowadzenia badań złożoności algorytmów.

7.2 Parametry testów

Zostały przeprowadzone trzy testy o różnej charakterystyce. Każda z przeprowadzonych prób algorytmu może posiadać jeden z następujących wyników:

- "timeout" – algorytm wywołany na danym grafie zajął więcej niż arbitralnie ustalona granica, więc został przerwany. Takie ograniczenie jest konieczne, gdyż przy wielu wywołaniach górna granica czasu trwania alogrytmu jest większa niż czas do końca semestru.
- "error" – kiedy algorytm kończy się błędem lub zwrócone przez niego kolorowanie jest niepoprawne.
- "colorable" – kiedy algorytm zwrócił kolorowanie, które jest poprawne.
- "uncolorable" – kiedy algorytm nie znalazł żadnego kolorowania.

Należy zwrócić uwagę, że poprawność wyniku "colorable" jest łatwa do roztrzygnięcia. Za to o poprawności wyniku "uncolorable" wnioskujemy jedynie porównując z innymi implementacjami rozwiązania problemu.

Wyniki wszystkich przeprowadzonych testów są dostępne jako arkusz google [LINK].

Na wykresach umieszczonych na arkuszu znalazły się linie trendu oparte o szereg potęgowy. Wydaje się, że odpowiadają one najlepiej temu, jaki trend jest widoczny dla otrzymanych danych. Należy jednak mieć pod uwagą, że są one przekłamane z powodu zastosowania ograniczenia górnego czasu wykonania algorytmu. Asymptotycznie nie należy oczekiwać, by ten trend się utrzymał.

7.3 Testy wszystkich algorytmów

Testy zostały przeprowadzone na następujących danych: grafy o liczbach wierzchołków od 10 do 100 z krokiem 10, wygenerowanych z parametrem gęstości od 0,05 do 0,3 z krokiem 0,05. Łączna liczba wygenerowanych grafów wynosi więc 60.

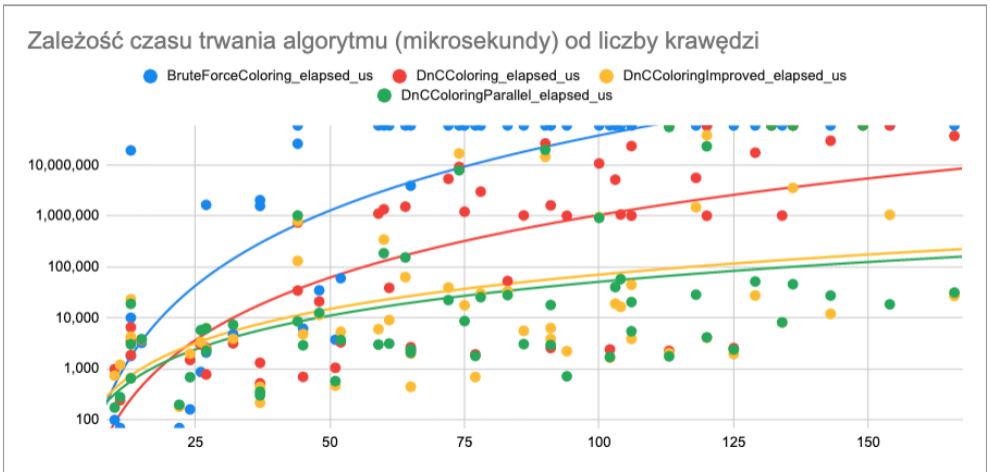
Algorytmy, które zostały uruchomione na tych grafach to:

- BruteForce – zgodny z opisem w sekcji 1.1,
- DivideAndConquerBasic – wersja pierwotna bez usprawnień, opisana przy okazji analizy teoretycznej,
- DivideAndConquerImproved - wersja z usprawnieniem opisanym w 5.1,
- DivideAndConquerParallel - próba zrównoleglenia algorytmu zgodnie z opisem w 6.2. W obecnej wersji posiada błąd implementacyjny, który powoduje, że algorytm nie jest poprawny. Wersja ta została umieszczona jako ciekawostka i nie została podjęta jej głębsza analiza.

Na każde uruchomienie algorytmu zostało ustawione ograniczenie czasowe 1 minuty. Dodatkowo na wywołanie wszystkich algorytmów dla wszystkich grafów zostało ustawione ograniczenie godzinne.

Zgodność wyników zaimplementowanej metody DivideAndConquer dla mniejszych rozmiarów grafów wskazuje, że algorytm działa poprawnie. Wyraźnie widoczny jest również zysk wynikający z zastosowania opisanego algorytmu - BruteForce nie wykonywał się do końca już przy 40 wierzchołkach, niezależnie od wybranego przy generacji parametru gęstości.

Pozostałe implementacje DivideAndConquer wykazują znacznie niższy średni czas działania. Należy mieć jednak na uwadze, że ciężko jest ten czas ograniczyć z konkretnym przedziale. Dla podobnych wielkości i gęstości grafów różnice w czasie wykonania algorytmu wynoszą nawet 4-5 rzędów wielkości.



7.4 Długi test DivideAndConquerImproved

Celem tego testu jest sprawdzenie, dla jakiego zakresu liczby wierzchołków DivideAndConquerImproved ma znaczne szanse na znalezienie rozwiązania lub określenie nierozwiązywalności w rozsądnym (czyli takim poniżej kilku minut) czasie.

Wygenerowane grafy posiadają liczbę wierzchołków od 2 do 300 oraz ich parametr gęstości został wybrany na wartość 0.1 - decyzja ta jest umotywowana różnorodnością wyników, jakie były otrzymywane dla tej wartości przy poprzednich próbach. Ograniczenie czasu wywołania algorytmu zostało wybrane na wartość 10 sekund, gdyż wcześniejsze testy pokazały, że powyżej tej granicy szansa, że algorytm zakończy pracę w rozsądnym czasie jest bardzo mała.

Wyniki, podobnie jak poprzednio są bardzo eratyczne. Można z nich wywnioskować, 80-100 wierzchołków jest granicą, powyżej występują grafy, na których algorytm wykonuje się długo (> 10 sekund). Należy jednak zwrócić uwagę, że udział tych grafów jest stosunkowo niewielki i ich odsetek rośnie stosunkowo powoli wraz z liczbą wierzchołków. Sugeruje to, że algorytm może mieć zastosowanie nawet dla bardzo dużych grafów, szczególnie w połączeniu z innymi algorytmami i heurystykami.

7.5 Dokładny test DivideAndConquerImproved

Celem tego testu jest dokładniejsze sprawdzenie, jak stworzony algorytm zachowuje się dla grafów, w zakresie, w którym możemy oczekiwać, że wynik uda się szybko otrzymać. Stąd też test został wykonany dla grafów o liczbie wierzchołków od 2 do 100, gdzie dla każdej liczby wierzchołków jest generowanych 10 grafów. Ponownie wybrana gęstość to 0.1, gdyż daje ona różnorodne wyniki (grafy kolorowalne przeplatają grafy niekolorowane).

Podobnie, jak miało to miejsce w poprzednich próbach, wyniki są eratyczne. Okazuje się, że faktycznie grafy, których rozwiązanie nie jest szybko znajdywalne pojawiają się wcześniej niż można by wnioskować z poprzednich rozważań. Pojawił się przypadek grafu o 32 wierzchołkach, przez który algorytm nie przeszedł w wyznaczobnych 10 sekundach.

Na ogół jednak algorytm spisał się dobrze, znajdując kolorowanie lub zwracając informację o braku kolorowania w 943 przypadkach z 1000.

8 Podsumowanie oraz wyciągnięte wnioski

Algorytm na pewno posiada wiele stopni, w których możliwa jest poprawa jego działania. Implementację można rozwijać w następujących kierunkach:

- analiza przypadków, które zajmują obecnej implementacji znacznie więcej czasu w porównaniu ze średnim przypadkiem
- modyfikacja algorytmu triangulizacji tak, aby działał liniowo,

- obliczanie długości cyklu fundamentalnego na podstawie zbudowanego drzewa dualnego,
- redukcja liczby rozpatrywanych krawędzi triangulacji,
- dodanie innych metod poszukiwania optymalnego separatora, w tym heurystycznych
- lepsze wycofywanie się z niepoprawnych kolorowań składowych,
- wcześniejsze kończenie działania algorytmu w przypadku znalezienia podgrafu nie-3-kolorowalnego,
- wykorzystanie obliczeń równoległych.

Podczas wspólnej pracy nad implementacją wyciągnęliśmy następujące wnioski:

- problemy trudne obliczeniowo są trudne, *pierwotnie nie wydawało się to być oczywiste*,
- nawet problemy trudne posiadają wiele miejsc do potencjalnej optymalizacji, których nie widać na pierwszy rzut oka,
- potencjalnie dobrym rozwiązaniem może być posiadanie kilku algorytmów rozwiązujących dany problem i próbowanie każdego z nich, jeśli wykonywanie poprzedniego trwa zbyt długo.
- w przypadku braku pomysłu na dalsze ulepszenia, warto pomyśleć o zrównolegleniu obliczeń.

Literatura

[1] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Springer, 2010.

[2] Wagner Prasinos Zaroliagis Holzer, Schulz. Engineering planar separator graphs. 14, 2009.

[3] Goosen Kant. Algorithms for drawing planar graphs, 1993.

[4] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.