Wydział Matematyki i Nauk Informacyjnych
POLITECHNIKA WARSZAWSKA

Algorytmika problemów trudnych obliczeniowo

3-kolorowanie grafu planarnego

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

13 czerwca 2021

Spis treści

1 Sformułowanie problemu

1.1 Wstęp

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.

1.2 Kolorowanie grafów planarnych

Kolorowanie grafów planarnych skrywa istotnie mniej tajemnic od kolorowania grafów nieplanarnych. Dysponujemy całym arsenałem twierdzeń ograniczających liczbę chromatyczną. Możemy chociażby posłużyć się twierdzeniem Heawood'a:

Theorem 1 ***Twierdzenie Heawooda***
Każdy graf planarny jest 5-kolorowalny

Dodatkowo, choć pierwotny dowód był poddawany w wątpliwość przez środowisko naukowe, kolejne dowody (Georges Gonthier, 2005) jedynie dowodzą poprawności twierdzenia sformułowanego przez Appela i Hakena:

Theorem 2 ***Twierdzenie o czterech barwach (Appel, Haken, 1976)***
Każdy graf planarny bez pętli jest 4-kolorowalny

Ponadto Grötzsch w 1959 roku odnalazł właściwość struktury grafów planarnych, która jest wystarczająca do stwierdzenia ich 3-kolorowalności.

Theorem 3 ***Twierdzenie Grötzscha (1959)***
Każdy graf planarny bez trójkątów jest 3 kolorowalny

Zauważmy jednak. że implikacja zachodzi wyłącznie w jedną stronę. Istnieją bowiem grafy planarne 3-kolorowalne zawierające trójkąty. Pomimo, że posiadamy istotnie więcej informacji, niż w przypadku grafów nieplanarnych, wciąż nie mamy narzędzia o rozsądnej złożoności, które jednoznacznie przesądza czy rozpatrywany graf jest 3-kolorowalny.

1.3 Rozwiązanie brute-force

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 $\mathcal{O}^*(3^n)$.

1.4 Rozwiązanie Divide & Conquer

Rozwiązanie brute-force, choć znajduje poprawne 3-kolorowanie (o ile istnieje) nie wydaje się optymalne pod względem czasu. 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 powiedzmy, że separator S dzieli graf na składowe A i B. (rozumowanie można uogólnić na większą ilość składowych). 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. Dodatkowo, do zapamiętania rozpatrywanych kolorowań wykorzystywane są listy kolorów. Bardziej szczegółowy opis algorytmu, analiza złożoności i dowód poprawności zostały przedstawione w kolejnych sekcjach dokumentu.

2 Opis algorytmu

2.1 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).

2.2 Znajdowanie separatora

Pierwszym krokiem algorytmu jest znajdowanie separatora. Odbywa się ono w oparciu o artykuł [2]. Przytoczymy teraz treść twierdzenia stojącego u podstaw procedury:

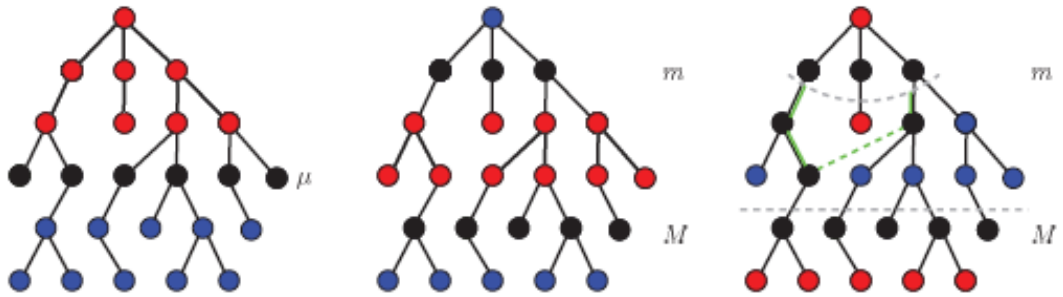
Theorem 4 ***Twierdzenie o separatorze w grafie planarnym (Lipton & Tarjan, 1979)*** *Niech G=(V,E) będzie grafem planarnym o co najmniej 5 wierzchołkach. Wtedy zbiór wierzchołków V może zostać podzielony na trzy zbiory A, B, S takie, że spełnione są następujące warunki:*

- *żaden z wierzchołków ze zbioru A nie jest połączony krawędzią z żadnym wierzchołkiem ze zbioru B*
- *zbiory A i B mają co najwyżej $\frac{2}{3}n$ wierzchołków*
- *S zawiera co najwyżej $\beta\sqrt{n}$ wierzchołków dla pewnej stałej β.*

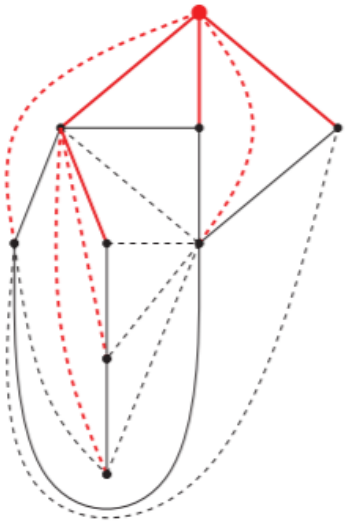
Zauważmy, że twierdzenie wprowadza ograniczenie na wielkość grafu. Jeśli rozpatrywany przez nas graf ma mniejszą liczbę wierzchołków niż 5, nie wiemy nic o istnieniu samego separatora, a tym bardziej o rozmiarze składowych.

Znajdowanie separatora grafu odbywa się przy pomocy funkcji `FindSeparator`. Algorytm składa się z 3 następujących po sobie faz. W każdej fazie wyznaczany jest separator. Jeśli separator wyznaczony przez jedną z dwóch pierwszych faz spełnia nałożone ograniczenia algorytm kończy pracę, zwracając wyznaczony separator. W przeciwnym przypadku rozpatrywana jest kolejna faza.

Faza trzecia kończy się dopiero z chwilą znalezienia separatora spełniającego nałożone ograniczenia. W naszym przypadku wielkość separatora jest istotna. Im mniejszych rozmiarów będzie separator tym mniej złożone będą wywołania rekurencyjne. Jednocześnie balans jest równie istotnym czynnikiem od którego zależy głębokość rekurencji. Ramowa koncepcja algorytmu znajdowania separatora została przedstawiona poniżej. Dokładna postać algorytmu jak i konieczne ograniczenia zostaną wyznaczone empirycznie.




```
1  FindSeparator(G)
2  {
3      //Phase 1
4      Create BFS tree
5      Search first level μ such that levels from 0 to μ contain ⩾ n/2
6      S <- μ
7      if S fits constraints
8          return S
9
10     //Phase 2
11     Find levels m and M such that |m|,|M|<2(√n − D)
12     where D is distance from μ level
13     S <- m ∪ M
14     //Graph is separated into three sets: A, B, C
15     if S fits constraints
16         return m ∪ M
17
18     //Phase 3
19     Reduce(G)
20     T <- Triangulate(G) //Described below
21     do
22     {
23         Pick not visited nontree edge e in T
24         F_c <- FundamentalCycle(G, e)
25         separator <- S ∪ F_c
26     }while(separator does not fit constraints)
27
28     return separator
29 }
30
31 Reduce(G)
32 {
33     Replace levels 0 to m with single node
34     Remove levels below M
35 }
```



```
1  Triangulate(G)
2  {
3      T <- Create BFS tree with arbitrary root
4      while(T is not triangulated)
5      {
6          Pick vertex v from G
7          for each node w not being neighbor of v
8          {
9              e <- triangulation edge {v, w}
10             if(e is not causing any crossings &&
11                 w has not been visited yet)
12             {
13                 Add e to T
14             }
15         }
16     }
17     return T
18 }
```

2.3 Rozpoznawanie składowych

Po znalezieniu separatora należy rozpoznać wszystkie składowe powstałe na wskutek usunięcia separatora. W tym celu skorzystamy z algorytmu znajdowania składowych opartym na algorytmie BFS. Główna pętla algorytmu jest wykonywana tak długo jak w grafie G istnieją wierzchołki nie przypisane jeszcze przez algorytm do żadnej składowej. W pojedynczej iteracji bierzemy wierzchołek v nie przypisany do żadnej składowej, tworzymy pustą listę component (przechowującą wierzchołki należące do aktualnie przeglądanej składowej) i przeglądamy graf wszerz rozpoczynając od v. Wierzchołek v wraz ze swoimi sąsiadami, sąsiadami swoich sąsiadów... stanowi osobną składową grafu. Tworzymy więc kolejkę do której będziemy dodawać wierzchołki należące do składowej, a których sąsiedzi nie zostali jeszcze odwiedzeni. Oczywiście do kolejki dodawany jest również wierzchołek v. W pętli while bierzemy pierwszy wierzchołek z kolejki i przeglądamy wszystkich jego sąsiadów. Jeśli którykolwiek z nich nie znajduje się jeszcze w liście component jest dodawany do kolejki. Pętla while kończy się więc gdy rozpatrzymy wszystkie wierzchołki wraz z ich sąsiadami w obrębie jednej składowej. Równocześnie wszystkie rozpatrywane wierzchołki dodawane są do listy component, która następnie jest dodawana do listy components przechowującej wszystkie rozpoznane składowe i przechodzimy do znajdowania kolejnej składowej.

```
FindComponents(G)
{
    Components <- empty list of components
    while(G has vertices not included in Components)
    {
        component <- empty list of vertices
        queue <- empty queue
        Pick first vertex v not included in any Components
        Add v to component
        Add v to queue
        while(queue is not empty)
        {
            v <- queue.First
            for each neighbor n of v
            {
                if(n is not included in component)
                {
                    Add n to component
                    Add n to queue
                }
            }
        }
        Add component to Components
    }
    return Components
}
```

2.4 Kolorowanie separatora

Do kolorowania separatora zastosowano metodę brute-force. Algorytm polega na kolorowaniu aktualnie rozpatrywanego wierzchołka v wszystkimi możliwymi kolorami i próbie pokolorowania rekurencyjnie pozostałych wierzchołków separatora dla danego kolorowania v. Warunkiem stopu rekurencji jest naturalnie przypadek gdy kolorowany jest ostatni wierzchołek separatora lub gdy dla danego wierzchołka zostaną rozpatrzone wszystkie kolorowania i żadne z nich nie prowadzi do otrzymania poprawnego kolorowania.

Zauważmy, że kolorowania kolejnych separatorów wymuszają na nas branie pod uwagę również kolorowań będących wyłącznie permutacjami kolorowań już wcześniej rozpatrywanych (indeksacja kolorów). Konieczna jest więc analiza wszystkich możliwych (poprawnych) kolorowań. Dodatkowy warunek pomijania niektórych kolorowań został opisany w dalszej części dokumentu.

Algorytm zwraca informacje czy graf (w naszym przypadku separator) jest 3-kolorowalny i zwraca prawidłowe kolorowanie (o ile istnieje).

```
bool BruteForceColoring(S)
{
    Pick vertex v from S
    for each color c avalaible for v
    {
        Color v with c
        for each neigbor n of v
        {
            if(n is not colored and can be colored with c)
            {
                remove color c from n.ColorsList
                if(n has no colors left)
                {
                    Undo changes in current loop iteration
                    Try next v coloring
                }
            }
        }
        if(v is not last vertex of S)
        {
            //Try to color S\v
            if(BruteForceColoring(S\v))
                return true
```

```
24          }
25          else
26          {
27              //Valid coloring found
28              return true
29          }
30          Undo changes in current loop iteration
31          Try next v coloring
32      }
33      //All v colorings have been tested
34      return false
35  }
```

2.5 Przechowywanie informacji o kolorowaniu separatora

Informacja o tym, jak w aktualnie rozpatrywanym kolorowaniu zostały pokolorowane separatory jest przechowywana w listach kolorów. Dla każdego z wierzchołków grafu definiujemy listę kolorów którymi może zostać pokolorowany wierzchołek. Oczywiście początkowo listy dla każdego z wierzchołków są inicjalizowane 3 kolorami. Przy kolorowaniu separatora z list wierzchołków sąsiadujących z wierzchołkami z separatora zabierany jest kolor jakim został pokolorowany jego sąsiad w separatorze.

Użycie list kolorów pozwala również na szybszą weryfikację, czy aktualnie rozpatrywane kolorowanie części grafu może odpowiadać poprawnemu kolorowaniu całego grafu. Jeśli aktualnie rozpatrywane kolorowanie separatora uniemożliwia pokolorowanie dowolnego wierzchołka z sąsiedztwa separatora, przechodzimy do kolejnego kolorowania separatora.

2.6 Badanie poprawności aktualnie rozpatrywanego kolorowania

Zauważmy, że ze względu na zastosowanie list kolorów, sprawdzenie, czy aktualnie rozpatrywane kolorowanie jest poprawne, nie wymaga dodatkowych operacji poza sprawdzeniem przed pokolorowaniem wierzchołka, jakie kolory są jeszcze dla niego dostępne. W przypadku gdy rozpatrywane kolorowanie nie jest poprawne, w grafie będzie istniał jeszcze nie pokolorowany wierzchołek, dla którego lista kolorów będzie pusta.

2.7 Ostateczna postać algorytmu

Po przybliżeniu poszczególnych składowych przejdziemy teraz do opisu zasadniczego algorytmu. Oczywiście graf G może mieć kilka składowych. Zaczynamy więc od wyznaczenia składowych G ($G\_Components$). Poczynimy 2 uwagi:

- Kolorowania składowych są wzajemnie niezależne (wynika wprost z definicji składowej i kolorowania)
- Jeśli dla dowolnej składowej nie istnieje poprawne kolorowanie, nie istnieje również poprawne kolorowanie całego grafu (wynika wprost z definicji składowej i kolorowania)

Na mocy pierwszej uwagi będziemy rozpatrywać kolorowanie każdej składowej z osobna.
Oznaczmy przez `comp` aktualnie rozpatrywaną składową. Zaczynamy od znalezienia dla niej separatora S (`FindSeparator`) i wyznaczeniu jej składowych (Components) (`FindComponents`). Następnie z wykorzystaniem rekurencji rozpatrujemy wszystkie poprawne kolorowania separatora S (wywołujemy `BruteForceColoring`) i dla każdego z nich staramy się pokolorować pozostałe składowe `Components`. W tym celu ponownie dla każdej składowej `comp` z `Components` wyznaczamy jej separator i składowe, po czym rekurencyjnie znajdujemy ich kolorowania.

Poniżej w pseudokodzie przedstawiono najogólniejszą postać algorytmu, zawierającą pewne uproszczenia, które zostały rozwiązane w dalszej części dokumentu. Przede wszystkim założono, że graf wejściowy G jest spójny, więc nie ma potrzeby znajdować jego początkowego podziału na składowe. Ponadto funkcja `BruteForceColorings`, której działanie jest odmienne od tego przedstawionego w rozdziale 2.4 zwraca za każdym razem kolejne prawidłowe kolorowanie grafu wejściowego (separatora)

S. Algorytm przedstawiony w takiej postaci nie różni się znacząco od postaci przedstawionej dalej, natomiast znacząco łatwiej zbadać na nim złożoność obliczeniową (patrz: 4.2).

```
1  Find3Coloring(G)
2      S <- FindSeparator(G)
3      Components <- FindComponents(G\S)
4      for each Coloring in BruteForceColorings(S)
5          allComponentsColored <- true
6          for each Component C in Components
7              if Find3Coloring(C) is false
8                  allComponentsColored <- false
9                  break
10             fi
11         rof
12
13         if allComponentsColored is true
14             return true
15         fi
16     rof
17
18     return false
```

Oczywiście procedura `BruteForceColoring` wymaga stosownych modyfikacji. W przypadku, gdy aktualnie kolorowany wierzchołek jest ostatnim wierzchołkiem separatora, kolorujemy go dostępnym dla niego kolorem (o ile istnieje) i próbujemy pokolorować rekurencyjnie każdą ze składowych. Jeśli dla każdej ze składowych udało się znaleźć poprawne kolorowanie, oznacza to, że znaleźliśmy poprawne kolorowanie jednej ze składowych G (jednej z $G\_Components$). W przeciwnym przypadku (dla którejś ze składowych nie istnieje poprawne kolorowanie) rozpatrujemy kolejne kolorowanie separatora.
Na mocy wspomnianego wcześniej twierdzenia [4] separator spełniający ograniczenia istnieje dla grafów których liczba wierzchołków jest $\geqslant 5$. Jeśli więc rozpatrywana składowa ma mniej niż 5 wierzchołków, nie mamy pewności czy istnieje dla niej separator spełniający ograniczenia. W takim przypadku kolorowana jest ona przy użyciu metody `BruteForceColoring`, tak jak gdyby cała składowa była po prostu separatorem. Oczywiście dla składowych o liczbie wierzchołków większej niż 5 algorytm znajduje separator i dalej rozpatruje jego kolorowanie oraz składowych.

```
1  bool Find3Coloring(G)
2  {
3      G_Components <- FindComponents(G)
4      for each component comp in G_Components
5      {
6          S <- FindSeparator(comp)
7          Components <- FindComponents(comp\S)
8
9          //Check if 3-coloring exists
10         if BruteForceColoring(Components, S) is false
11             return false;
12     }
13     return true
14 }
15
16 bool BruteForceColoring(Components, S)
17 {
18     Pick vertex v from S
19     for each color c available for v
20     {
21         Color v with c
22         for each neighbor n of v
23         {
24             if(n is not colored and c is avaliable for n)
25             {
26                 remove color c from n.ColorsList
27                 if(n has no colors left)
28                 {
29                     Undo changes in current loop iteration
30                     Try next v coloring
31                 }
```

```
32              }
33          }
34          if(v is not last vertex of S)
35          {
36              //Try to color S\v
37              if(BruteForceColoring(Components, S\v))
38                  return true
39          }
40          else
41          {
42              for each component comp in Components
43              {
44                  //Try to color comp
45                  if(comp has less than 5 vertices)
46                  {
47                      if(!BruteForceColoring(empty list, comp))
48                          Try next v coloring
49                  }
50                  else
51                  {
52                      S' <- FindSeparator(comp)
53                      Components' <- FindComponents(comp\S')
54                      if(!BruteForceColoring(Components', S'))
55                          Try next v coloring
56                  }
57              }
58              //All components can be colored
59              return true
60          }
61          Undo changes in current loop iteration
62          Try next v coloring
63      }
64      //All v colorings have been tested
65      return false
66  }
```

3 Analiza poprawności

3.1 FindSeparator

Funkcja `findSeparator` argumentu `G` (typ zbiór wierzchołków grafu planarnego G) zwraca zbiór wierzchołków separatora, który dzieli graf G na spójne składowe o liczbie wierzchołków co najwyżej $\frac{2}{3}|G|$ (typ zbiór wierzchołków pewnego grafu G).

Dowód poprawności algorytmu może zostać bezpośrednio wyprowadzony z dowodu twierdzenia o separatorze planarnym i nie został tutaj umieszczony ze względu na swoją złożoność. [4]

3.2 FindComponents

Funkcja `findComponents` argumentu `G` (typ zbiór wierzchołków pewnego grafu G) zwraca zbiór spójnych składowych (typ zbiór zbiorów wierzchołków pewnego grafu G).

Funkcja korzysta z powszechnie znanego algorytmu BFS, którego poprawność nie została tutaj opisana, gdyż jest to algorytm powszechnie znany.

3.2.1 Warunek stopu

Funkcja wykonuje pętlę, dopóki w podanym zbiorze wierzchołków istnieją wierzchołki nie zawarte w którymkolwiek z `Components`. Jako, że w każdej iteracji do zbioru komponentów dodawany jest dokładnie jeden niepusty komponent (niepusty, bo znajduje się w nim co najmniej wierzchołek, od którego zaczyna się przeszukiwanie BFS) w każdej iteracji liczba wierzchołków nie przypisanych do żadnej składowej zmniejsza się co najmniej o 1. Liczba wierzchołków jest skończona, więc algorytm skończy się po wykonaniu co najwyżej $|G|$ iteracji.

3.2.2 Częściowa poprawność

Rozważmy następujący niezmiennik: Jeśli istnieje wierzchołek $v \in G$, który nie należy do żadnej spójnej składowej ze zbioru `Components`, to istnieje spójna składowa grafu G, do której v należy. Spójna składowa, do której należy v jest wyznaczany poprawnym algorytmem BFS i dodawany do `Components`

Rozpatrzmy teraz niezmiennik przed rozpoczęciem pętli:

- jeśli zbiór `G` jest pusty, to poprzednik implikacji niezmiennika jest fałszywy, stąd implikacja jest prawdziwa.
- jeśli zbiór `G` jest niepusty oraz `Components` jest pusty, to każdy wierzchołek `G` nie należy do żadnego z `Components`.

Wiemy więc, że funkcja `findComponents` jest skończona i częściowo poprawna - jest więc poprawna.

3.3 `BruteForceColoring`

Funkcja `BruteForceColoring` jako argumenty przyjmuje separator `S` (typ: zbiór wierzchołków) pewnego spójnego i planarnego grafu G oraz zbiór wszystkich spójnych składowych `Components` (typ: zbiór zbiorów wierzchołków) grafu $G \setminus S$

Funkcja zwraca wartości logiczne `true`, jeśli graf G jest 3-kolorowalny lub `false`, jeśli nie jest 3-kolorowalny. Dodatkowo, jeśli `BruteForceColoring` zwraca `true`, to efektem ubocznym (side effect) jest poprawne kolorowanie zbiorów przekazanych wierzchołków.

Istotne jest, że w celu uproszczenia rozumowania, wszędzie tam, gdzie `BruteForceColoring` zwraca `false`, tam wszystkie efekty uboczne wywołania danej funkcji są wycofywane.

3.3.1 Warunek stopu

Rozpatrzmy dwa wymiary rekursji w funkcji `BruteForceColoring`. Pierwszy z nich wykonuje się na ustalonym zestawie spójnych składowych - z każdym poziomem rekursji zmienia się jedynie liczba pozostałych niepokolorowanych wierzchołków w zbiorze `S`. Koniec jest w dwóch wypadkach:

- Kiedy nie da się pokolorować kolejnego wierzchołka
- Kiedy wszystkie wierzchołki `S` zostały pokolorowane

Stąd też ten wymiar rekursji (czyli ze stałym argumentem `Components`) spełnia warunek stopu, gdyż początkowa wartość $|S|$ jest stała i skończona, liczba wierzchołków S zmniejsza się o 1 przy każdym kolejnym wywołaniu oraz następuje zatrzymanie, kiedy S już nie ma żadnych wierzchołków.

Kiedy rekursja w pierwszym wymiarze zostanie zakończona pokolorowaniem wszystkich wierzchołków `S`, następuje rekursja w drugim wymiarze: dla każdej ze spójnych składowych zostaje wykonana jedna z dwóch operacji:

- Jeśli rozważana spójna składowa ma mniej niż 5 wierzchołków, funkcja jest rekursywnie wywoływana, tak, że cała spójna składowa jest przekazywana jako argument S, a lista spójnych składowych jest pusta. W ten sposób kolorowanie jest dokańczane w pierwszym wymiarze rekursji.
- W przeciwnym wypadku zostaje wykonany podział komponentu za pomocą separatora na nowy zbiór `S` oraz zbiór komponentów `Components` = spójne składowe G - S

Korzystając z faktu, że największy pod-komponent C' przy każdym rozbiciu komponentu C ma co najwyżej $\frac{2}{3}|C|$ wierzchołków, można stwierdzić, że ten wymiar rekursji również się zatrzyma - najpóźniej po k wywołaniach, gdzie k spełnia nierówność $(\frac{2}{3})^k n < 5$.

Z powyższych rozważań ostatecznie wynika, że dla dowolnego grafu planarnego G funkcja `BruteForceColoring` zatrzyma się po skończonej liczbie wywołań dla dowolnego grafu planarnego G.

3.3.2 Częściowa poprawność

Dla algorytmu `BruteForceColoring` można znaleźć następujący niezmiennik:

Rozważmy graf G, w którym dla każdego wierzchołka została stworzona lista dostępnych kolorów. Weźmy separator S grafu G oraz zbiór wszystkich spójnych składowych C grafu $G \setminus S$. Jeśli G jest kolorowalny z użyciem kolorów dostępnych dla każdego wierzchołka, to istnieje takie kolorowanie separatora S, że

- S został pokolorowany kolorami z listy dostępnych kolorów dla każdego wierzchołka
- w listach dostępnych kolorów wierzchołków sąsiadujących z S zostały usunięte kolory ich pokolorowanych sąsiadów. Jednocześnie listy te pozostają niepuste jeśli wierzchołek nie został jeszcze pokolorowany.
- dla każdej ze spójnych składowych C jest znajdywany separator i spójne składowe. Następnie dla nich szukane są kolorowalna dostępnymi na swoich wierzchołkach kolorami (rekursywne wywołanie) [1]

Fakt istnienia takiego separatora jest sprawdzany poprzez następujący niezmiennik:
Jeśli separator jest kolorowalny dostępnymi kolorami, to jeśli wybierzemy dowolny jego niepokolorowany wierzchołek v i pokolorujemy go dostępnym kolorem c, oraz usuniemy z list jego sąsiadów wybrany kolor, to jeśli separator $S \setminus \{v\}$ jest kolorowalny, to znaczy, że istnieje kolorowanie S, gdzie kolor v to c.

3.4 `Find3Coloring`

`Find3Coloring` jest funkcją, której argumentem jest dowolny graf planarny - jego reprezentacja nie jest istotna z punktu widzenia poprawności algorytmu. Funkcja zwraca wartości logiczne `true`, jeśli graf jest kolorowalny lub `false`, jeśli nie jest kolorowalny za pomocą kolorów dostępnych na początku na wierzchołkach. W szczególności, funkcja może zostać użyta do sprawdzenia, czy dowolny graf G podany w argumencie jest 3-kolorowalny, jeśli na wszystkich jego wierzchołkach zostanie na początku ustawiona dostępność trzech kolorów, tych samych dla każdego wierzchołka.

`Find3Coloring` znajduje wszystkie spójne składowe grafu G, następnie dla każdej z nich ustalane jest, czy jest 3-kolorowalna za pomocą funkcji `BruteForceColoring`.

Poprawność tego sprawdzenia jest trywialna, stąd jedynie krótkie uzasadnienie:

- Pętla jest skończona, gdyż skończony graf posiada skończoną liczbę spójnych składowych
- Funkcja jest poprawna, gdyż zwraca `false`, jeśli którakolwiek ze spójnych składowych nie jest 3-kolorowalna. Funkcja zwraca `true`, jeśli dla wszystkich komponentów istnieje poprawne kolorowanie

4 Analiza złożoności

4.1 `FindSeparator`

Separatory zwrócone przez algorytm w fazie I lub II definiuje się jako **separatory proste**. Te zwrócone w fazie III natomiast – jako **separatory złożone**.
Przy szacowaniu złożoności będziemy korzystać ze znanego faktu o **rzadkości grafów planarnych**, tj:

$$|E| \leqslant 3 \cdot |V| - 6$$

[1] wywołanie rekursywne z podziałem komponentu za pomocą separatora następuje, kiedy liczba wierzchołków $\geqslant 5$. W przeciwnym wypadku cały spójna składowa jest oznaczana jako S i przekazywana do `BruteForceColoring` wraz z pustym zbiorem spójnych składowych, co jest równoważne sprawdzeniu metodą brute-force, czy przekazane S jest kolorowalne dostępnymi kolorami. Ten fakt nie wpływa na niezmiennik, więc został pominięty dla klarowności.

4.1.1 Faza I i II

Budowa drzewa rozpinającego
Faza I rozpoczyna się od stworzenia drzewa metodą BFS (linia 4), której najprostsza i dobrze znana implementacja posiada złożoność czasową: $\mathcal{O}(|V| + |E|)$, co dla grafów planarnych można uprościć (korzystając z *rzadkości*):

$$\mathcal{O}(|V| + 3 \cdot |V| - 6) = \mathcal{O}(|V|) = \mathcal{O}(n)$$

Dla prostszego i szybszego przeprowadzenia kolejnych kroków w Fazach I i II, warto zapamiętywać liczności każdego z poziomów (być może wraz z wierzchołkami do nich należącymi) powstałego drzewa już podczas jego tworzenia. Wymaga to złożoności pamięciowej $\mathcal{O}(h)$, gdzie h – wysokość drzewa. W pesymistycznym przypadku (gdy np. powstałe drzewo jest ścieżką) mamy więc **złożoność pamięciową** $\mathcal{O}(n)$.

Przeszukanie poziomów
Przeszukanie poziomów w celu znalezienia separatora *prostego* (zarówno w fazie I jak i II) odbędzie się, przy wykorzystaniu zapamiętanych informacji, **w czasie** $\mathcal{O}(h) = \mathcal{O}(n)$.

Usprawnienia
Warto wspomnieć o możliwości polepszenia wyników zwróconych przez ten etap. Faza I nie musi zwracać pierwszego poziomu μ spełniającego warunki twierdzenia. Jeżeli istnieje więcej takich poziomów w drzewie, warto porównać ich liczności i wybrać najmniejszy. Podobnie w Fazie II – warto rozważyć przejście po wszystkich poziomach i znalezienie takich m, M, że składający się z tych poziomów separator będzie najmniejszy. Przejście takie może znaleźć wynik bardziej nas satysfakcjonujący przy jednoczesnym niedużym nakładzie czasowym.

4.1.2 Faza III

Redukcja [linia `19`]
Faza III rozpoczyna się od operacji redukcji (linia `19`), którą można sprowadzić do wielokrotnego usuwania wierzchołków oraz "przesuwania" krawędzi. Łatwo zauważyć, że wystarczy przejść po wszystkich wierzchołkach z grafu i, korzystając z informacji zapisanych przy tworzeniu drzewa BFS w fazie I, odpowiednie usunąć bądź zmodyfikować. Stąd złożoność jest ograniczona z góry przez $\mathcal{O}(n)$.

Triangulacja `[20]`
W literaturze (np. [3]) dostępne są algorytmy znajdujące triangulację grafu planarnego w czasie liniowym.

Enumeracja cykli fundamentalnych `[21-26]`
Faza III domyślnie kończy się w momencie znalezienia w triangulacji grafu takiej krawędzi nienależącej do drzewa (*nontree edge*), która spełnia wymagania twierdzenia. W pesymistycznym przypadku algorytm będzie potrzebował zbadać wszystkie takie krawędzie i dla każdej z nich obliczyć długość cyklu fundamentalnego przez nią indukowanego.

Jeżeli przez C oznaczymy złożoność operacji `FundamentalCycle`, złożoność czasowa fazy III wynosić będzie co najwyżej $\mathcal{O}(|E| \cdot C) = \mathcal{O}(nC)$. Przy zaangażowaniu grafu dualnego do rozważanej triangulacji, można zaimplementować ten krok w taki sposób, aby każdy kolejny cykl korzystał z informacji obliczonej przez poprzednie wywołanie funkcji. Wówczas złożoność tej operacji jest pomijalna ([2], rozdział 2.2).

Faza III, podobnie jak poprzednie, nie musi kończyć się w momencie znalezienia pierwszego cyklu fundamentalnego spełniającego kryteria. Przy wykorzystaniu co najwyżej liniowej ilości pamięci komputera, być może warto znaleźć więcej separatorów w tej fazie i porównać ich liczności lub bilans powstałych w podziale składowych. Ponownie, zmiana złożoności czasowej będzie nieznaczna.

4.1.3 Podsumowanie

Przy poszukiwaniu separatora algorytm przechodzi w pesymistycznym przypadku przez wszystkie 3 fazy. Każda z tych faz posiada złożoność liniową. Również wszelkie usprawnienia, które zostały

opisane wyżej, przy odpowiedniej implementacji nie zmieniają klasy złożoności algorytmu.

Złożoność czasowa operacji `FindSeparator`: $\mathcal{O}(n)$

Złożoność pamięciowa operacji `FindSeparator`: $\mathcal{O}(n)$

4.2 `Find3Coloring`

Przyjrzyjmy się ponownie uproszczonej, generalnej wersji algorytmu:

```
1  Find3Coloring(G)
2      S <- FindSeparator(G)
3      Components <- FindComponents(G\S)
4      for each Coloring in BruteForceColorings(S)
5          allComponentsColored <- true
6          for each Component C in Components
7              if Find3Coloring(C) is false
8                  allComponentsColored <- false
9                  break
10             fi
11         rof
12
13         if allComponentsColored is true
14             return true
15         fi
16     rof
17
18     return false
```

4.2.1 Złożoność czasowa

Badając przypadek pesymistyczny, możemy przyjąć, że graf wejściowy jest spójny. Każda kolejna składowa grafu o n wierzchołkach zmniejsza rozmiar problemu, zmniejszając tym samym wysokość drzewa rekurencji, a w konsekwencji i złożoność obliczeniową. Podobnie założymy, że każde wywołanie funkcji `FindSeparator` znajduje separator, którego usunięcie z grafu powoduje, że w grafie pozostaną dokładnie 2 spójne składowe A, B.

`BruteForceColoring` iteruje się po wszystkich możliwych kolorowaniach grafu wejściowego. Ich liczbę można ograniczyć z góry przez $3^{\beta\sqrt{n}}$, gdzie β jest stałą zależną od użytej implementacji algorytmu (dla implementacji Tarjana i Liptona $\beta = \sqrt{8}$). Niech $T(n)$ będzie maksymalną liczbą wywołań rekurencyjnych algorytmu opartego o pseudokod napisany powyżej. Wówczas wielkość tę można oszacować przez następujące równanie rekursywne:

$$\begin{aligned}
T(n) &\leqslant 3^{\beta\sqrt{n}} \left(T\left(\left\lfloor \frac{2}{3} n \right\rfloor \right) + T\left(\left\lfloor \frac{2}{3} n \right\rfloor \right) \right) \leqslant 2 \cdot 3^{\beta\sqrt{n}} T\left(\frac{2}{3} n \right) \\
&= 2 \cdot 3^{\beta\sqrt{n}} \cdot 2 \cdot 3^{\beta\sqrt{\frac{2}{3}}\sqrt{n}} \cdot T\left(\left(\frac{2}{3} \right)^2 n \right) \\
&= 2^2 \cdot 3^{\beta\left(1+\sqrt{\frac{2}{3}}\right)\sqrt{n}} \cdot T\left(\left(\frac{2}{3} \right)^2 n \right) = \ldots \\
&= 2^h \cdot 3^{\beta\left(1+\left(\frac{2}{3}\right)^{\frac{1}{2}}+\left(\frac{2}{3}\right)^{\frac{2}{2}}+\ldots+\left(\frac{2}{3}\right)^{\frac{h-1}{2}}\right)\sqrt{n}} \cdot T\left(\left(\frac{2}{3} \right)^h n \right)
\end{aligned}$$

gdzie $h-1$ to wysokość drzewa rekurencyjnego, tj. taka (maksymalna) liczba wywołań rekurencyjnych, dla których rekursja napotyka warunek stopu - spójne składowe mają wielkość co najwyżej 5. Stąd otrzymujemy:

$$\left(\frac{2}{3} \right)^h n = 5 \Rightarrow h = \log_{\frac{2}{3}} \frac{5}{n}$$

Znajdujący się w wykładniku ciąg jest sumą częściową szeregu geometrycznego. Jego suma wynosi asymptotycznie:

$$s(i) = 1 \cdot \frac{1 - (\sqrt{\frac{2}{3}})^i}{1 - \sqrt{\frac{2}{3}}} \xrightarrow{i \to \infty} \frac{1}{1 - \sqrt{\frac{2}{3}}} \leqslant 5.45$$

Jest to szereg rosnący, w związku z czym złożoność można oszacować z góry:

$$T(n) \leqslant 2^{\log_{\frac{2}{3}} \frac{5}{n}} \cdot 3^{5.45\beta\sqrt{n}} \cdot T(5)$$

Operacja $T(5)$ jest przeprowadzana nierekurencyjnie i jej złożoność jest stała, w związku z czym można ją pominąć przy obliczeniach asymptotycznych.

Po kilku dalszych przekształceniach:

$$2^{\log_{\frac{2}{3}} \frac{5}{n}} = 2^{\frac{\log_2 \frac{5}{n}}{\log_2 \frac{2}{3}}} = \left(2^{\log_2 \frac{5}{n}}\right)^{\frac{1}{\log_2 \frac{2}{3}}} = \left(\frac{5}{n}\right)^{\frac{1}{\log_2 \frac{2}{3}}} = \left(\frac{n}{5}\right)^{A}$$

gdzie $A = -\frac{1}{log_2 \frac{2}{3}} \approx 1.71$, możemy zapisać:

$$T(n) = \mathcal{O}\left(n^{1.71} \cdot 3^{5.45\beta\sqrt{n}}\right) = \mathcal{O}^*\left(3^{\mathcal{O}(\sqrt{n})}\right)$$

4.2.2 Złożoność pamięciowa

Każde wywołanie rekurencyjne funkcji `Find3Coloring` zapamiętuje zarówno wierzchołki należące do separatora jak i do pozostałych komponentów grafu wejściowego (linie 2-3 algorytmu). Można to prosto zaimplementować, przechowując te informacje w tablicach, które łącznie zajmą $\mathcal{O}(n)$ pamięci (ponieważ każdy wierzchołek, wraz powiązanymi informacjami jak jego lista kolorów, musi być zapamiętany w którejś tablicy). Przy takiej implementacji funkcji `BruteForceColorings`, aby kolorowała bezpośrednio graf wejściowy (nie robiąc kopii), jej złożoność pamięciowa będzie stała, tj. pomijalna.

Zauważmy, że każdy kolejny poziom w drzewie rekurencji może wówczas zająć pesymistycznie również $\mathcal{O}(n)$ przestrzeni pamięciowej. W pesymistycznym przypadku separator może być bowiem bardzo mały, a suma wszystkich "wierzchołków" na tym poziomie drzewa rekurencyjnego (reprezentujących spójne składowe grafu będącego ich rodzicem) będzie stanowić (niemal) cały ich graf-rodzic. Stąd wynika złożoność liniowa każdego poziomu rekurencji. Przy liczeniu złożoności czasowej obliczyliśmy głębokość tego drzewa jako:

$$h = \log_{\frac{2}{3}} \frac{5}{n} = \log_{\frac{2}{3}} 5 - \log_{\frac{2}{3}} n = \log_{\frac{2}{3}} 5 + \log_{\frac{3}{2}} n = \mathcal{O}(\log n)$$

Taka sama jest również zajętość pamięci związana z przechowywaniem wywołań funkcji na stosie programu.
Ostatecznie złożoność pamięciowa algorytmu `Find3Coloring` jest rzędu:

$$M(n) = \mathcal{O}(n \log n + \log n) = \mathcal{O}(n \log n)$$

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.