Kolorowanie grafów

Sarna Piotr

Maciej Dzierwa

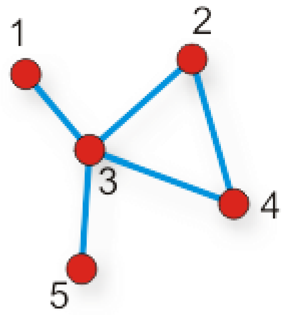
Mikołaj Zając

Konrad Ligas

Charakterystyka problemu

Graf – zbiór wierzchołków, połączonych krawędziami. W naszych algorytmach będziemy wykorzystywać grafy nieskierowane nieważone – krawędzie nie mają ani kierunku ani wagi.

Graf nieskierowany nieważony można przedstawić za pomocą macierzy sąsiedztwa w następujący sposób:



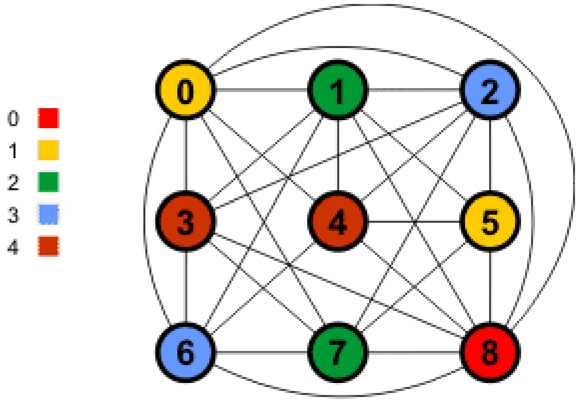
0	0	1	0	0
0	0	1	1	0
1	1	0	1	1
0	1	1	0	0
0	0	1	0	0

Źródło: https://www.if.pw.edu.pl/~agatka/moodle/obiekty.html

Indeks każdego wiersza w macierzy oznacza aktualnie rozpatrywany wierzchołek, a kolumna wierzchołek, z którym chcemy się połączyć. „1” oznacza krawędź między nimi dwoma, „0” oznacza brak krawędzi.

Kolorowanie grafu to przypisanie kolorów wierzchołkom w taki sposób, aby żadne dwa sąsiednie wierzchołki nie miały tego samego koloru. Celem jest minimalizacja liczby kolorów – tzw. liczby chromatycznej grafu.

Liczba chromatyczna – najmniejsza liczba kolorów potrzebna do prawidłowego pokolorowania grafu.



Źródło: https://eduinf.waw.pl/inf/alg/001_search/0142.php

Graf nieskierowany nieważony:

- Liczba wierzchołków: 8
- Liczba chromatyczna: 5

Algorytm z nawrotami (ang. backtracking) to metoda systematycznego sprawdzania wszystkich możliwych rozwiązań problemu poprzez próbowanie różnych opcji i wycofywanie się z tych, które nie prowadzą do poprawnego rozwiązania.

Zasada działania:

- Rekurencja – Algorytm próbuje zbudować rozwiązanie krok po kroku
- Sprawdzanie warunków – W każdym kroku sprawdza, czy aktualna częściowa konfiguracja spełnia ograniczenia problemu
- Nawrót – Jeśli nie spełnia, algorytm porzuca tę ścieżkę i wraca do poprzedniego kroku (backtrack), próbując następną opcję
- Znalezienie rozwiązania – Proces kontynuowany jest aż do znalezienia poprawnego rozwiązania lub wyczerpania wszystkich możliwości

Zalety:

- Gwarantuje znalezienie najbardziej optymalnego rozwiązania (jeśli istnieje)

Wady:

- Może być bardzo wolny dla dużych instancji problemu (złożoność wykładnicza)

Algorytm zachłanny (ang. Greedy Algorithm) to metoda rozwiązująca problem w sposób przybliżony, często szybciej niż tradycyjne metody dokładne, ale bez gwarancji optymalności lub poprawności rozwiązania. Wybierający on lokalnie najbardziej optymalne rozwiązanie w każdym pojedynczym kroku a nie po przejściu całego algorytmu.

Zasada działania:
- Lokalnie optymalny wybór – W każdym kroku wybiera najlepszą dostępną opcję
- Brak nawrotów – Po podjęciu decyzji nie cofa się

Zalety:
- Szybki i prosty w implementacji

Wady:
- Nie zawsze prowadzi do optymalnego rozwiązania globalnego

Algorytm dokładny

Algorytm dokładny to algorytm kolorowania grafu z nawrotami.

Przebieg algorytmu:

- Wybierz kolejny niepokolorowany wierzchołek, spróbuj przypisać mu kolor z listy wszystkich dostępnych kolorów (zaczynając od 1 koloru).
- Jeśli da się pokolorować, wykorzystując któryś z dostępnych kolorów:
    - Jeśli żaden sąsiad nie ma tego samego koloru – przypisz go i przejdź do następnego wierzchołka.
    - Jeśli kolor jest zajęty – spróbuj następnego koloru
- Jeśli nie da się pokolorować:
    - Cofnij się (backtrack) do poprzedniego wierzchołka i zmień jego kolor na następny możliwy
    - Jeśli wyczerpano wszystkie kolory – zwiększ maksymalną liczbę kolorów i zacznij od nowa
- Powtarzaj aż do pokolorowania wszystkich wierzchołków, wypisz pokolorowany graf

Opis kodu algorytmu dokładnego

Funkcja exactGraphColoring()

```
void exactGraphColoring(int** graphMatrix, const int& vertexCount) {
    std::vector<int> colors(vertexCount, 0);

    for (int maxColors = 1; maxColors <= vertexCount; maxColors++) {
        if (solveColoring(graphMatrix, vertexCount, colors, 0, maxColors)) {
            std::cout << "Minimalna liczba kolorow: " << maxColors << "\n";
            for (int i = 0; i < vertexCount; i++) {
                std::cout << "Wierzcholek " << i << ": Kolor " << colors[i] << "\n";
            }
            return;
        }
    }
}
```

Funkcja inicjalizująca proces:

- Rozpoczyna od 1 koloru i stopniowo zwiększa ich liczbę
- Dla każdej liczby kolorów wywołuje solveColoring()
- Gdy znajdzie poprawne kolorowanie, wyświetla wynik i kończy działanie

Funkcja solveColoring()

```
bool solveColoring(int** graphMatrix, const int& vertexCount, std::vector<int>& colors, const int& vertex, const int& maxColors) {
    if (vertex == vertexCount) {
        return true;
    }

    for (int color = 1; color <= maxColors; color++) {
        if (isSafe(graphMatrix, vertexCount, colors, vertex, color)) {
            colors[vertex] = color;
            if (solveColoring(graphMatrix, vertexCount, colors, vertex + 1, maxColors)) {
                return true;
            }
            colors[vertex] = 0;
        }
    }
    return false;
}
```

Główna funkcja rekurencyjna, która próbuje pokolorować graf:

- Jeśli wszystkie wierzchołki są pokolorowane (vertex == vertexCount), zwraca „true”
- Dla każdego koloru sprawdza bezpieczeństwo (funkcja isSafe())
- Jeśli kolor jest bezpieczny, przypisuje go i rekurencyjnie próbuje pokolorować kolejne wierzchołki
- W przypadku niepowodzenia (backtracking) cofa przypisanie koloru (colors[vertex] = 0)

Funkcja isSafe()

```cpp
bool isSafe(int** graphMatrix, const int& vertexCount, const std::vector<int>& colors, const int& vertex, const int& color) {
    for (int i = 0; i < vertexCount; i++) {
        if (graphMatrix[vertex][i] == 1 && colors[i] == color) {
            return false;
        }
    }
    return true;
}
```

Sprawdza, czy można przypisać dany kolor do wierzchołka, analizując kolory sąsiadów w macierzy sąsiedztwa. Zwraca „false”, jeśli któryś sąsiad ma ten sam kolor.

Algorytm LF (ang. **L**argest **F**irst)

Algorytm LF to algorytm zachłanny kolorowania grafu.

Przebieg algorytmu:

- Sortowanie wierzchołków:
  - Oblicz liczbę sąsiadów każdego wierzchołka
  - Posortuj wierzchołki malejąco według liczby sąsiadów (najpierw wierzchołki o najwyższym stopniu)
- Weź kolejny wierzchołek według posortowanej kolejności, spróbuj przypisać mu najniższy dostępny kolor (zaczynając od 1), który:
  - Nie jest użyty u żadnego sąsiada
  - Należy do aktualnego zakresu kolorów
- Jeśli nie znaleziono odpowiedniego koloru:
  - Zwiększ zakres dostępnych kolorów
  - Przypisz nowy kolor nowemu wierzchołkowi
- Powtarzaj aż do pokolorowania wszystkich wierzchołków, wypisz pokolorowany graf

Opis kodu algorytmu LF

Struktura Vertex

```
struct Vertex {
    int index, neighboursCount;
};
```

Struktura odpowiadająca pojedynczemu wierzchołkowi, zawiera jego indeks z macierzy sąsiedztwa oraz jego stopień

Funkcja getSortedVertexVector()

```cpp
std::vector<Vertex> getSortedVertexVector(int** graphMatrix, const int& vertexCount) {
    std::vector<Vertex> vertexVector(vertexCount);

    for (int i = 0; i < vertexCount; i++) {
        int neighboursCount = 0;
        for (int j = 0; j < vertexCount; j++) {
            if (graphMatrix[i][j] == 1) {
                neighboursCount++;
            }
        }
        vertexVector[i] = {i, neighboursCount};

    }

    for (int i = 0; i < vertexCount; i++) {
        for (int j = 0; j < vertexCount; j++) {
            if (vertexVector[j].neighboursCount > vertexVector[i].neighboursCount) {
                std::swap(vertexVector[i], vertexVector[j]);
            }
        }
    }

    return vertexVector;
}
```

Przygotowuje posortowaną listę wierzchołków:

- Dla każdego wierzchołka zlicza liczbę sąsiadów
- Tworzy wektor struktur Vertex zawierających indeks wierzchołka i stopień wierzchołka
- Sortuje wierzchołki malejąco według liczby sąsiadów (najpierw te z największą liczbą)

Funkcja LFgraphColoring()

```
void LFgraphColoring(int**& graphMatrix, const int& vertexCount) {
    std::vector<int> colors(vertexCount, 0);
    std::vector<Vertex> vertexVector = getSortedVertexVector(graphMatrix, vertexCount);

    int maxColors = 1;
    for (int i = 0; i < vertexCount; i++) {
        int vertexIndex = vertexVector[i].index;
        bool colored = false;

        for (int j = 1; j <= maxColors; j++) {
            if (isSafe(graphMatrix, vertexCount, colors, vertexIndex, j)) {
                colors[vertexIndex] = j;
                colored = true;
                break;
            }
        }

        if (!colored) {
            colors[vertexIndex] = ++maxColors;
        }
    }

    std::cout << "Minimalna liczba kolorow: " << maxColors << "\n";
    for (int i = 0; i < vertexCount; i++) {
        std::cout << "Wierzcholek " << i << ": Kolor " << colors[i] << "\n";
    }
}
```

Główna funkcja implementująca algorytm:

- Inicjalizuje tablicę kolorów
- Pobiera posortowaną liczbę wierzchołków
- Dla każdego wierzchołka (w kolejności posortowanej) próbuje przypisać najniższy możliwy kolor
- Jeśli nie znajdzie dostępnego koloru, zwiększa pulę dostępnych kolorów
- Na końcu wyświetla minimalną liczbę kolorów i przypisanie kolorów do wierzchołków

Funkcja isSafe()

```
bool isSafe(int** graphMatrix, const int& vertexCount, const std::vector<int>& colors, const int& vertex, const int& color) {
    for (int i = 0; i < vertexCount; i++) {
        if (graphMatrix[vertex][i] == 1 && colors[i] == color) {
            return false;
        }
    }
    return true;
}
```

Sprawdza, czy można przypisać dany kolor do wierzchołka, analizując kolory sąsiadów w macierzy sąsiedztwa. Zwraca „false”, jeśli któryś sąsiad ma ten sam kolor.

Porównanie złożoności algorytmów

Algorytm dokładny

Złożoność obliczeniowa

W najgorszym przypadku O(k^n), gdzie:

- k – maksymalna liczba kolorów (w praktyce sprawdzane od 1 do liczby wierzchołków)
- n – liczba wierzchołków w grafie

Złożoność obliczeniowa wykładnicza wynika z działania algorytmu dokładnego. Algorytm ten sprawdza wszystkie możliwe kombinacje kolorowania wierzchołków, co w najgorszym przypadku prowadzi do pełnego przeglądu drzewa możliwości (każdy wierzchołek może mieć do k kolorów).

Algorytm LF

Złożoność obliczeniowa

W najgorszym przypadku O(n^2), gdzie:

- Sortowanie wierzchołków (getSortedVertexVector()): O(n^2) – dla każdego wierzchołka (n) zliczamy sąsiadów (n operacji).
- Główne kolorowanie (LFgraphColoring()): O(n^2) – Dla każdego wierzchołka (n) sprawdzamy kolorowy (do maxColors, gdzie maxColors ≤ n).

Algorytm LF dzięki złożoności wielomianowej działa dobrze dla dużych grafów, ale nie gwarantuje minimalnej liczby kolorów.

Porównanie działania algorytmów

Prezentacja działania obu algorytmów dla tego samego grafu o 5ciu wierzchołkach:

Algorytm dokładny

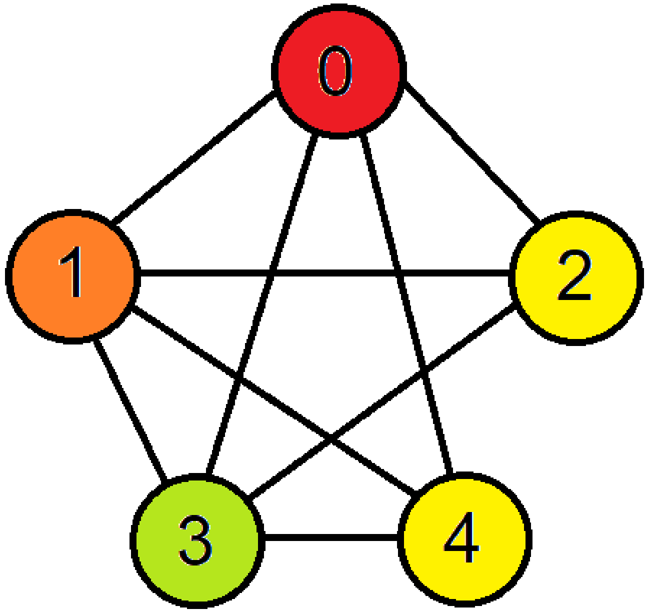
```
Problem kolorowania grafow
Wybierz graf do pokolorowania:
1. Graf 3-wierzcholkowy
2. Graf 5-wierzcholkowy
3. Graf 10-wierzcholkowy
2
0 1 1 1 1
1 0 1 1 1
1 1 0 1 0
1 1 1 0 1
1 1 0 1 0
1. Algorytm dokladny
2. Algorytm LF
Wybierz algorytm:
1
Minimalna liczba kolorow: 4
Wierzcholek 0: Kolor 1
Wierzcholek 1: Kolor 2
Wierzcholek 2: Kolor 3
Wierzcholek 3: Kolor 4
Wierzcholek 4: Kolor 3
```

Algorytm LF

```
Problem kolorowania grafow
Wybierz graf do pokolorowania:
1. Graf 3-wierzcholkowy
2. Graf 5-wierzcholkowy
3. Graf 10-wierzcholkowy
2
0 1 1 1 1
1 0 1 1 1
1 1 0 1 0
1 1 1 0 1
1 1 0 1 0
1. Algorytm dokladny
2. Algorytm LF
Wybierz algorytm:
2
Minimalna liczba kolorow: 4
Wierzcholek 0: Kolor 2
Wierzcholek 1: Kolor 4
Wierzcholek 2: Kolor 1
Wierzcholek 3: Kolor 3
Wierzcholek 4: Kolor 1
```

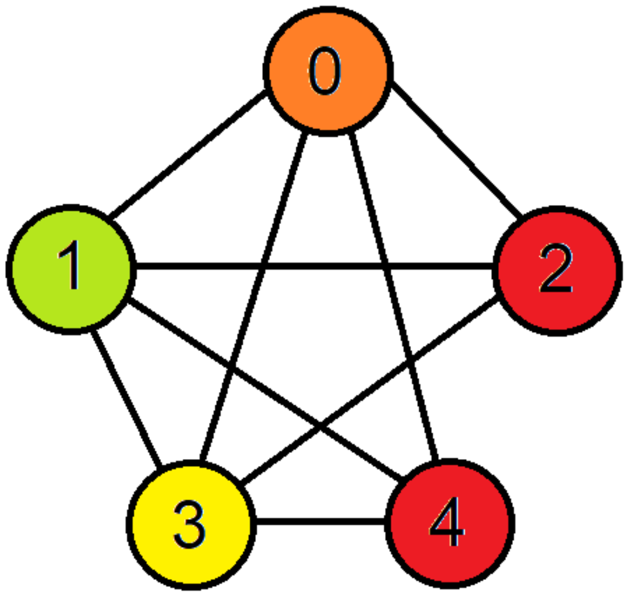
Zwizualizowanie działania algorytmów

Algorytm dokładny



Kolor 1
Kolor 2
Kolor 3
Kolor 4

Algorytm LF



Jak widać, oba algorytmy działają poprawnie, mimo innego działania pokolorowały graf w identyczny sposób, wykorzystując do tego maksymalnie 4 kolory.

Różnica w działaniu algorytmów

Prezentacja działania obu algorytmów dla tego samego grafu o 10ciu wierzchołkach:

Algorytm dokładny

```
Problem kolorowania grafow
Wybierz graf do pokolorowania
1. Graf 3-wierzcholkowy
2. Graf 5-wierzcholkowy
3. Graf 10-wierzcholkowy
3
0 1 0 0 0 1 0 0 0 0
1 0 1 0 0 0 1 0 0 0
0 1 0 1 0 0 0 1 0 0
0 0 1 0 1 0 0 0 1 0
0 0 0 1 0 1 0 0 0 1
1 0 0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
1. Algorytm dokladny
2. Algorytm LF
Wybierz algorytm:
1
Minimalna liczba kolorow: 2
Wierzcholek 0: Kolor 1
Wierzcholek 1: Kolor 2
Wierzcholek 2: Kolor 1
Wierzcholek 3: Kolor 2
Wierzcholek 4: Kolor 1
Wierzcholek 5: Kolor 2
Wierzcholek 6: Kolor 1
Wierzcholek 7: Kolor 2
Wierzcholek 8: Kolor 1
Wierzcholek 9: Kolor 2
```

Algorytm LF

```
Problem kolorowania grafow
Wybierz graf do pokolorowania:
1. Graf 3-wierzcholkowy
2. Graf 5-wierzcholkowy
3. Graf 10-wierzcholkowy
3
0 1 0 0 0 1 0 0 0 0
1 0 1 0 0 0 1 0 0 0
0 1 0 1 0 0 0 1 0 0
0 0 1 0 1 0 0 0 1 0
0 0 0 1 0 1 0 0 0 1
1 0 0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
1. Algorytm dokladny
2. Algorytm LF
Wybierz algorytm:
2
Minimalna liczba kolorow: 4
Wierzcholek 0: Kolor 1
Wierzcholek 1: Kolor 3
Wierzcholek 2: Kolor 2
Wierzcholek 3: Kolor 3
Wierzcholek 4: Kolor 4
Wierzcholek 5: Kolor 2
Wierzcholek 6: Kolor 1
Wierzcholek 7: Kolor 1
Wierzcholek 8: Kolor 1
Wierzcholek 9: Kolor 1
```

Ten przykład pokazuje, jak zachłanność algorytmu LF może dać gorszy wynik niż optymalne rozwiązanie. Graf ten jest celowo skonstruowany tak, aby zachłanność algorytmu LF podjęła nieoptymalne decyzje.

Podsumowanie

Algorytm dokładny

Złożoność:

- Czasowa: O(k^n)
- Pamięciowa: O(n)

Zalety:

- Znajduje optymalne rozwiązanie

Wady:

- Bardzo wolny dla grafów >20/30 wierzchołków
- Niepraktyczny w zastosowaniach wymagających szybkości

Algorytm LF

Złożoność:

- Czasowa: O(n^2)
- Pamięciowa: O(n)

Zalety:

- Szybki – nadaje się do dużych grafów
- Łatwy w implementacji

Wady:

- Może dać gorsze wyniki niż algorytm dokładny
- Zależny od kolejności wierzchołków – wymaga ich posortowania

Bibliografia

https://inf.ug.edu.pl/~hanna/grafy/14_kolorowanie.pdf

https://mattomatti.com/pl/a0298

https://www.if.pw.edu.pl/~agatka/moodle/obiekty.html

https://eduinf.waw.pl/inf/alg/001_search/0142.php

https://ufkapano.github.io/algorytmy/lekcja14/color.html