Kolorowanie grafu

Kolorowanie wierzchołków grafu polega na przyporządkowaniu każdemu wierzchołkowi tego grafu pewnej wartości – liczby naturalnej, intuicyjnie rozumianej jako kod odnoszący się do pewnego koloru. Kolorowanie grafu jest legalne (dozwolone, prawidłowe) jeżeli żadnemu z dwóch sąsiadujących wierzchołków nie przypisano tej samej wartości. Optymalne kolorowanie ma miejsce gdy liczba użytych wartości (wierzchołków) równa jest liczbie chromatycznej $\chi(G)$ grafu G , to znaczy gdy użyto najmniejszej możliwej liczby kolorów.

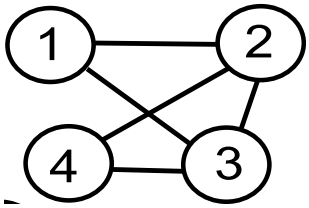
Na problem kolorowania grafów po raz pierwszy natknięto się w XIX wieku gdy wraz z rozwojem wojskowej kartografii próbowano obniżyć koszty masowego druku map w dużej skali. Hipoteza twierdzenia o czterech barwach, według którego każdy graf planarny można pokolorować co najwyżej czterema barwami, postawiona została już w 1840r. a jej pierwszy dowód przedstawiono dwanaście lat później.

Algorytm dokładny

Problem kolorowania grafu to typowy przykład problemu NP-trudnego. Nie jest znany algorytm pozwalający na optymalne pokolorowanie grafu w czasie wielomianowym co oznacza że znajdowanie liczby chromatycznej i decyzyjny wariant problemu – to znaczy odpowiedź czy graf można legalnie pokolorować co najwyżej k kolorami – również mają wykładniczą złożoność obliczeniową. Wciąż jedynym sposobem pozwalającym na uzyskanie dokładnego rozwiązania jest podejście „bruteforce”, to znaczy wygenerowanie i przejrzenie wszystkich możliwych rozwiązań i wybór tego które będzie jednocześnie charakteryzowane przez najniższą liczbę chromatyczną i dozwolone. Oczywistą wadą jest wysoka złożoność obliczeniowa: znalezienie najlepszego poprawnego przyporządkowania n wierzchołkom co najwyżej k kolorów wymaga k^n porównań. W pesymistycznym wypadku grafów bardzo gęstych lub wręcz pełnych, ilość kolorów będzie równa ilości wierzchołków $k=n$ co sprowadza złożoność obliczeniową do jeszcze wyższej klasy $O(n^n)$. Czas działania takiego algorytmu jest niepraktycznie długi jeśli liczba wierzchołków dochodzi do 10, co w większości wypadków właściwie wyklucza jego praktyczne stosowanie. Pamięciowa złożoność jest niska: $O(N)$.

```
Pseudokod 1. Algorytm dokładny – bruteforce
Dane: graf o N wierzchołkach,
policz_kolory(kolorowanie) – funkcja podająca ilość wartości użytych kolorów
legalne_kolorowanie(kolorowanie) - funkcja informująca czy podane kolorowanie
jest dozwolone
inkr_z_przeniesieniem(kolorowanie) – funkcja zwraca następne możliwe
kolorowanie

procedure koloruj_bruteforce(graf):
      kolorowanie := {0, … ,0} // wszystkie kolory ustawione na 0
      licznik := 0
      while licznik < N^N:
      //jeżeli kolorowanie jest prawidłowe
            if legalne_kolorowanie(kolorowanie):
                  //jeżeli kolorowanie ma mniej kolorów niż najlepsze
                  if policz_kolory(kolorowanie) <
                  policz_kolory(najlepsze_kolorowanie):
                        najlepsze_kolorowanie := kolorowanie
            //znajdź następne kolorowanie
            kolorowanie := inkr_z_przeniesieniem(kolorowanie)
      return najlepsze_kolorowanie
```

Niech dany będzie graf G postaci: $\{1{:}[2, 3], 2{:}[1, 3, 4], 3{:}[1, 2, 4], 4{:}[3, 2]\}$ o 4 wierzchołkach. Kolejne sprawdzane kolorowania dla algorytmu bruteforce: {1: 0, 2: 0, 3: 0,



4: 0}, {1: 1, 2: 0, 3: 0, 4: 0}, {1: 2, 2: 0, 3: 0, 4: 0}, {1: 3, 2: 0, 3: 0, 4: 0},{1: 0, 2: 1, 3: 0, 4: 0}, ... , {1: 0, 2: 2, 3: 1, 4: 0} – najlepsze kolorowanie grafu G , użyte zostały 3 kolory $\chi(G)=3$.

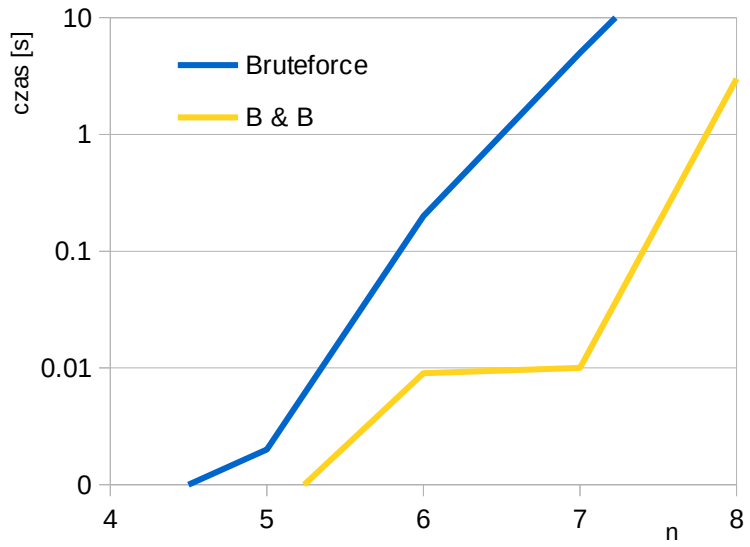
Można podjąć próbę optymalizacji poprzez heurystyczne wyznaczenie na początku działania algorytmu maksymalnej możliwej wartości $\mathrm{k\_max}$ – górnego ograniczenia liczby chromatycznej – przy użyciu innego podejścia, na przykład algorytmu zachłannego, po czym standardowo wyznaczyć wszystkie możliwe warianty. Redukuje to ich liczbę z n^n do $\mathrm{k\_max}^n$; dodatkowo po znalezieniu rozwiązania o niższej liczbie kolorów można ponownie zmniejszać $\mathrm{k\_max}$ i dodatkowo ograniczać zakres przeszukiwanych rozwiązań. Dla grafów rzadkich może to stanowić bardzo istotne usprawnienie co pokażą testy szybkości porównujące go z algorytmem bruteforce. Jednakże klasa czasowej złożoności obliczeniowej algorytmu w dalszym ciągu pozostaje wykładnicza co ogranicza

jego zastosowanie do raczej niewielkich, rzadkich grafów.

Pseudokod 2. **Zmodyfikowany algorytm dokładny z ograniczeniem.**

```
procedure koloruj_branch_and_bound(graf):
      kolorowanie := {0,...,0};
      najlepsze_kolorowanie := koloruj_zachlannie(graf)
      k_max := policz_kolory(najlepsze_kolorwanie)
      kontynuuj := True
      while (kontynuuj == True):
            if legalne_kolorowanie(obecne_kolorowanie):
                  if policz_kolory(kolorowanie) < k_max:
                        najlepsze_kolorowanie := obecne_kolorowanie
                        k_max = policz_kolory(kolorowanie)
            kontynuuj := inkr_z_przeniesieniem(kolorowanie, k_max);
      return kolorowanie
```

Dla przykładowego grafu G kolejność generowania pierwszych sprawdzanych rozwiązań jest identyczna jak w podstawowym algorytmie, rozwiązane optymalne również zostaje znalezione jako 36., lecz w tym momencie następuje przypisanie $\mathrm{k\_max}=3$ i program zwróci wynik już po sprawdzeniu nowych $\mathrm{k\_max}^n$ (zamiast wszystkich n^n) rozwiązań.



Rys. 1 Porównanie szybkości działania algorytmów dokładnych.

Algorytm zachłanny

Wszystkie pozostałe omawiane algorytmy to algorytmy przybliżone, nie gwarantujące dokładnego rozwiązania problemu (uzyskania optymalnego kolorowania) a jedynie pewne jego przybliżenie. Pierwszym z nich jest najprostszy z omawianych, algorytm zachłanny. Inicjalizuje on strukturę danych (na przykład słownik lub tablicę haszującą) kolorowania z tą samą wartością (na przykład kodem 0) dla wszystkich wierzchołków. W następnym kroku iteruje po kolejnych wierzchołkach, inkrementując przypisaną im wartość aż do momentu w którym żaden wierzchołek sąsiadujący z aktualnie przetwarzanym nie ma przypisanej takiej samej wartości. Rozwiązanie uzyskiwane jest szybko, w wielomianowym czasie, jednak jak wspomniano może nie być optymalne (dokładne).

Pseudokod 3. **Algorytm zachłanny.**

```
Dane: graf o N wierzcholkach

procedure koloruj_zachlannie(graf):
      int kolorowanie := {0,...,0}
      for i = 0 to N:
            v = graf.wierzcholki[i]
            lista sasiednie_kolory := [];
            for j = 1 to N:
                  if graf.zawiera_krawedz(v, j):
                        sasiednie_kolory.dodaj(kolorowanie[j])
            while sasiednie_kolory.zawiera(kolorowanie[v]):
                        kolorowanie[v] += 1;
```

Pamięciowa złożoność $O(N)$ jest analogiczna do algorytmu dokładnego. Dużą zaletą jest niska złożoność czasowa $O(n^2)$, osiągana jest ona jednak kosztem rezygnacji z dokładnego rozwiązania i zadowolenia się jego przybliżeniem. Dokładność uzyskanego przybliżenia zależy od badanego zbioru danych oraz szczegółów implementacji.

Algorytm przybliżony: LF

Algorytm Largest First jest modyfikacją algorytmu zachłannego. Pierwszym krokiem jest stworzenie listy wierzchołków, posortowanej według ich stopnia. Następnie według uzyskanej kolejności zostają one pokolorowane zachłannie.

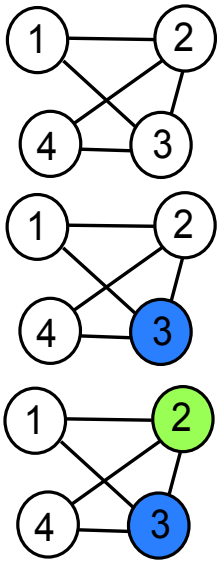
Pamięciowa złożoność obliczeniowa znów wynosi $O(N)$. Wprawdzie przed stworzeniem struktury danych odpowiedzi konieczne jest dodatkowo stworzenie listy wierzchołków posortowanej według ich stopnia, również o długości N , lecz w notacji asymptotycznej taka różnica nie ma znaczenia. Utrzymana zostaje również wielomianowa złożoność czasowa. Pierwszym etapem jest sortowanie listy o długości N , dla którego można przyjąć złożoności średnią $O(n \cdot \log n)$ i optymistyczną $O(n)$. Założenie jest słuszne dla algorytmu sortowania timsort z biblioteki standardowej użytego języka – Python 3. Kroki te i tak nie są trudniejsze niż następujące po nich zachłanne kolorowanie o klasie $O(n^2)$.

Pseudokod 4. **Algorytm LF.**

```
procedure koloruj_lf(graf):
      int v_malejaco[N] = sortuj(graf.wierzcholki, key=graf.stopien(v))
      kolorowanie := {0, ..., 0}
      for i = N to 0:
            v := v_malejaco[i]
            lista sasiednie_kolory := []
            for j = 1 to N:
                  if graf.zawiera_krawedz(v, j):
                        sasiednie_kolory.dodaj(kolorowanie[j])

            while sasiednie_kolory.zawiera(kolorowanie[v]):
                        kolorowanie[v] += 1
      return kolorowanie
```

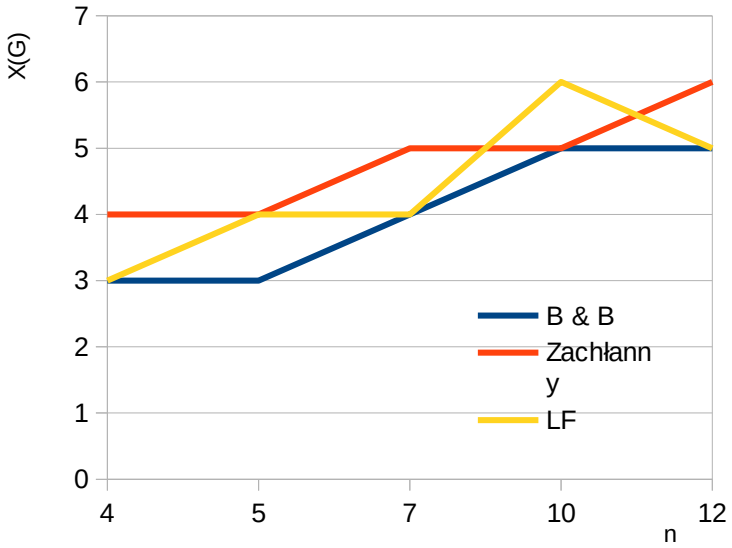
Algorytm ten, tak samo jak wyjściowy zachłanny, w przypadku ogólnym daje tylko przybliżone rozwiązanie. Jednak jak pokaże przypadek testowy, zastosowana prosta modyfikacja znacząco zwiększa dokładność otrzymywanego przybliżenia.

Schemat działania algorytmu dla przykładowego grafu G . Wartości dla jego kolorowania: 0 – białe, 1 – niebieskie, 2 – zielone, 3- żółte.



- lista v_malejaco ma postać 3, 2, 4, 1
- wszystkie wierzchołki sąsiadujące z 3. mają przypisane wartości 0 więc kolorowanie[3] zostaje zwiększone do 1.
- wierzchołki sąsiednie do 2. są pokolorowane 0, 1, 0, kolorowanie[2] zostaje inkrementowane do 2.
- sąsiedzi 4. mają przypisane kolejno 1, 2, więc kolorowanie[4] pozostaje równe 0.
- wreszcie wierzchołki sąsiadujące z 1 pokolorowane są jako 2, 1, stąd kolorowanie[1] również pozostaje równe 0.

Liczba użytych kolorów równa jest 3, rozwiązanie jest więc w tym szczególnym przypadku optymalne. W przypadku ogólnym nie musi to mieć miejsca, jednak średni błąd przybliżenia i tak jest zazwyczaj mniejszy niż dla wyjściowego prostego algorytmu zachłannego



Rys. 2 Porównanie dokładności osiąganych przez algorytmy przybliżone z algorytmem dokładnym

Algorytm genetyczny

Idea algorytmu genetycznego bazuje na biologicznej teorii ewolucji sterowanej doborem[1,2]. Zdefiniowane zostają funkcje lub operatory krzyżowania i mutacji osobników - zakodowanych, zwykle binarnie, rozwiązań. W pierwszym kroku zostaje zainicjalizowana populacja, to jest pula losowo wygenerowanych osobników. Zostają one ocenione pod kątem poprawności i liczby użytych kolorów, po czym następuje selekcja rodziców i tworzenie osobników nowego pokolenia. Im dane rozwiązanie jest lepsze, tym większe ma szanse na bycie rodzicem osobnika który pojawi się w następnym pokoleniu. Cykl selekcji rodziców i „hodowania” nowego pokolenia powtarzany jest pewną liczbę razy, po czym zwracane zostaje najlepsze rozwiązanie z ostatniego pokolenia.

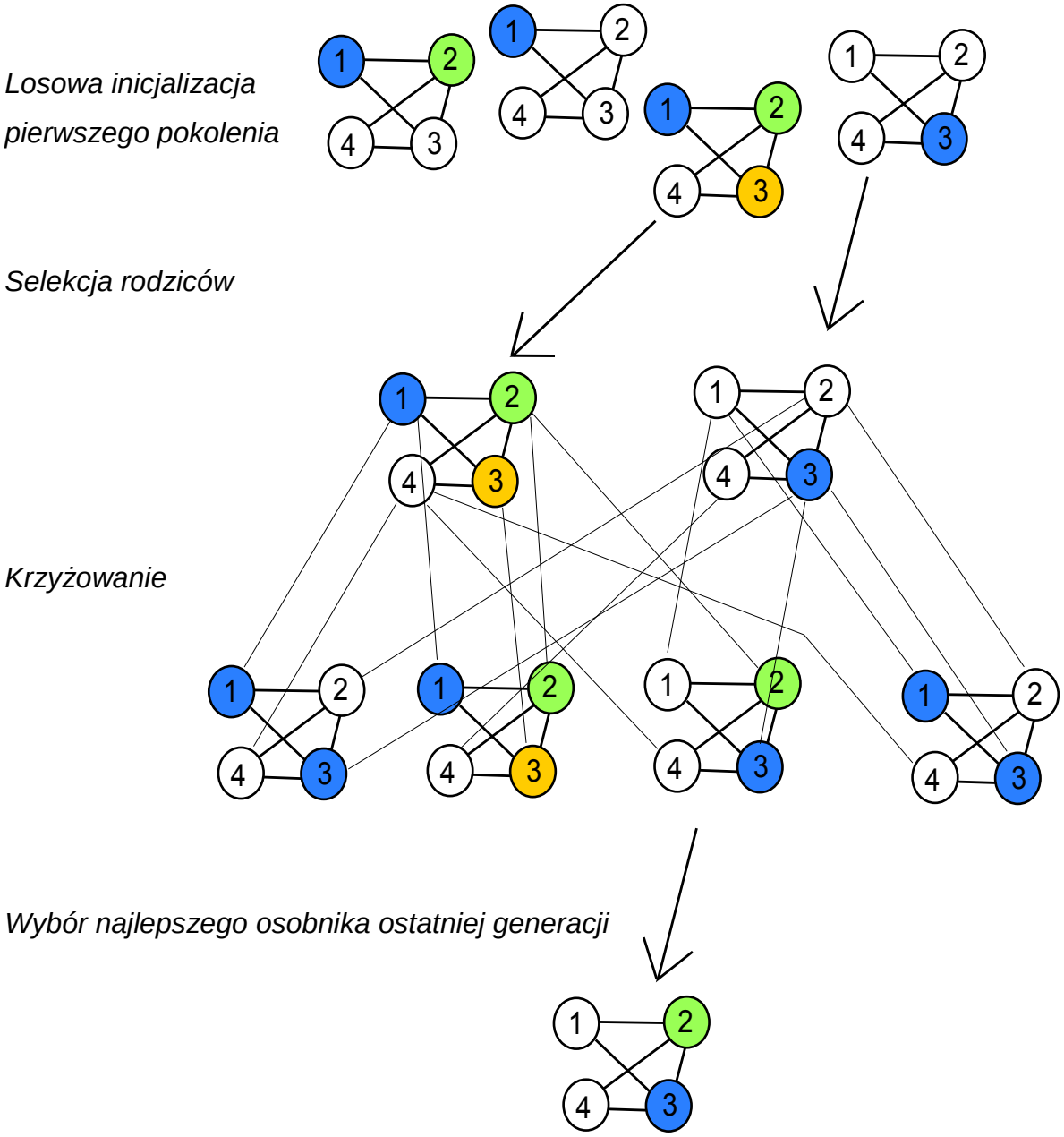
Pseudokod 5. **Algorytm genetyczny**

```
Dane: graf,
l – liczebność populacji,
p – ilość pokoleń,
m – częstość mutacji

procedure kolorowanie_genetyczne(graf, l, p, m):
      kolorowanie populacja[l]
      k_max := policz_kolory(koloruj_zachlannie(graf))
      for i=1 to l:
            // losowa inicjalizacja osobników
            populacja[i] := losowe_kolorowanie(graf, k_max)

      for i=1 to p:
            // wybór populacji rodzicielskiej
            populacja_rodzicow := selekcja_rodzicow(populacja)
            // 'hodowla' następnego pokolenia
            populacja := krzyzowanie(populacja_rodzicow)
            // losowe zmiany osobników z określoną częstością
            mutacje(populacja, m)

      wybierz_najlepszego(populacja)
```

Nietypową cechą są złożoności: pamięciowa $O(N \cdot p)$ i czasowa $O(n \cdot l \cdot O(f))$ algorytmu genetycznego zależące nie tylko od zadanej liczby optymalizowanych elementów (wierzchołków grafu), ale także od liczby pokoleń p i ich liczebności l ustalanych przez programistę, oraz złożoności funkcji oceny dopasowania $O(f)$. Przy założeniu stałych parametrów p i l dawałoby to wręcz złożoność liniową $O(n)$ jednak notacja O zastosowana do oceny złożoności tego typu algorytmów może być myląca, gdyż ukrywa duży stały narzut obliczeniowy obecny nawet dla małych wartości *n.*

W niektórych zastosowaniach optymalizacyjnych zamiast sztywnego zadawania liczby pokoleń możliwe jest zatrzymanie algorytmu genetycznego gdy rozwiązanie osiągnie pewną wyznaczoną wcześniej progową wartość funkcji celu. W wypadku kolorowania grafu również można wyznaczyć taką progową ilość kolorów której osiągnięcie będzie satysfakcjonujące. To rozwiązanie jest jednak obarczone ryzykiem z jednej strony przedwczesnego zatrzymania algorytmu i otrzymania niewystarczająco dokładnego przybliżenia, z drugiej – zapętlenia się grafu jeśli wybrana wartość będzie niższa od liczby chromatycznej grafu. Jest to o tyle prawdopodobne że przeważnie wartość ta musi być wybrana bez wcześniejszej znajomości $\chi(G)$, jako że samo wyznaczenie liczby chromatycznej jest równie trudne co znalezienie optymalnego kolorowania grafu. Dokładne wartości p i l zazwyczaj muszą zostać ustalone empirycznie, tak aby z jednej strony uzyskać satysfakcjonującą dokładność przybliżenia wyniku, z drugiej otrzymywać je w odpowiednio krótkim czasie; ich dobór nie jest trywialnym zagadnieniem.

Zaimplementowany graf wykorzystuje selekcję osobników rodzicielskich spośród najlepszych 25% osobników metodą koła ruletki oraz całkowite zastępowanie osobników przez potomne w każdym kolejnym pokoleniu.



Poniższy schemat przedstawia przykładowy przebieg szukania kolorowania wcześniej danego grafu G przez silnie uproszczony algorytm genetyczny, z dwoma pokoleniami liczącymi cztery osobniki i z pominięciem mechanizmu mutacji. Z racji niedeterministycznej inicjalizacji osobników pierwszego pokolenia i nie determinizmu jednego z etapów selekcji w każdym pokoleniu, przetwarzanie grafu G przebiega w inny sposób przy każdym uruchomieniu algorytmu.

Sieć neuronowa

Algorytm wielowartościowej sieci neuronowej[3]. Każdy stan S_i neuronu może przyjąć jedną z k różnych wartości (kolorów) tzn.: $S_i \in \{1, \dots, k\}$.

Mając graf **G**={**V**, **E**}, N = |**V**| - liczba wierzchołków, budujemy odpowiadającą mu sieć neuronową – taką jak graf, gdzie każdemu wierzchołkowi przyporządkujemy neuron. $\{A_{ij}\}$ - macierz sąsiedztwa dla tego grafu: A_{ij} = 1 gdy $\{v_i, v_j\} \in$ **E**, lub 0 w przeciwnym wypadku. Każdy wierzchołek zwraca wartość odpowiadającą jego stanowi S_i, wejściem są stany sąsiadujących wierzchołków. W zależności od problemu definiuje się różne funkcje energii E(**S**). Celem algorytmu jest zlezienie takiego stanu **S** (kolorowania) aby odpowiadająca mu energia była minimalna.

Aby znaleźć lokalne minimum energetyczne będziemy zmieniali kolor c pojedynczego neuronu i: S_i <- c w taki sposób by energia układu malała:

$$\Delta E(i,c) = E(\boldsymbol{S}) - E(\boldsymbol{S}_{old}) \quad ,$$

gdzie **S** to stan $\mathbf{S}_{old}$ po zamianie S_i <- c.

Dla problemu znalezienia minimalnego kolorowania korzystamy z funkcji energii określonej wzorem:

$$E(\boldsymbol{S},\gamma) = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{ij}\,\delta(S_i,S_j) + \gamma\sum_{i=1}^{N} S_i$$

gdzie δ - delta Kroeneckera – zwraca 1 gdy argumenty są takie same, 0 w przeciwnym wypadku. Pierwszy składnik sumy zlicza liczbę wystąpień tego samego koloru na sąsiednich wierzchołkach – jest on odpowiedzialny za to by rozwiązanie było prawidłowe. Drugi składnik to suma wszystkich kolorów – odpowiada za minimalizację ilości użytych kolorów. Gdy współczynnik γ jest odpowiednio mały γ < γ*, wtedy wszystkie lokalne minima zawierają prawidłowe kolorowanie. Im większy jest współczynnik γ tym mniej różnych kolorów zawiera lokalne minimum. W algorytmie najpierw liczymy **S** dla γ > γ* - otrzymując małą liczbę kolorów, a następnie optymalizujemy **S** dla γ < γ* by otrzymać prawidłowe kolorowanie.

Pseudokod 5. **Algorytm Sieci Neruonowej**

```
2 najważniejsze pętle:
ineer_loop(S,γ)
      do
            // policz macierz Delta_E(i,c) – zmian energii gdy zamienimy kolor
            //wierzchołka i na c
            Delta_E := compute_DeltaE(S, γ)
            //wybierz parę [i,c] odpowiadającą największej ujemnej różnicy
            //energii Delta_E(i,c) < 0 lub zwróć 0 gdy niemożliwe
            [i,c] := F_greedy(Delta_E)
            S[i] := c //pokoloruj wierzchołek i na c
      while (i != 0) //powtarzaj dopóki zmniejszamy energię
      return S //kolorowanie odpowiadające lokalnemu minimum

annealing_loop(γL, γH)
      S := randomS()  //losowe kolorowanie
      S := inner_loop(S,γL) //pierwsze prawidłowe kolorowanie
      do
            S_prev := S       //zapisz prawidłowe kolorowanie
            //znajdź nieprawidłowe kolorowanie o małej ilości kolorów
            S := inner_loop(S,γH)
            //powróć do prawidłowego kolorowania
            S := inner_loop(S,γL)
      while(E(S) < E(S_prev)) //powtarzaj dopóki rozwiązanie się poprawia
      return S_prev     // zwróć prawidłowe kolorowanie
```

1) `inner_loop(S, γ)` – zaczynając od kolorowania **S** szuka **S** odpowiadającemu lokalnemu minimum. W zależności od γ jest to rozwiązanie poprawne lub z małą ilością kolorów.

2) `annealing_loop(γL,γH)` – zaczynając od losowego kolorowania **S** naprzemiennie oblicza kolorowanie prawidłowe (γL < γ*) i o małej ilości kolorów dopóki energia prawidłowego kolorowania spada. Zwraca najlepsze znalezione prawidłowe kolorowanie.

Współczynnik γH jest obliczany na początku tak by liczba znalezionych dzięki niemu kolorów była pewnym ułamkiem (np. f = 3/4) liczby kolorów znalezionych początkowo przy γL < γ*. W naszym algorytmie współczynnik γ* = 1/(DG*(DG+1)), gdzie DG to maksymalny stopień wierzchołka w grafie.

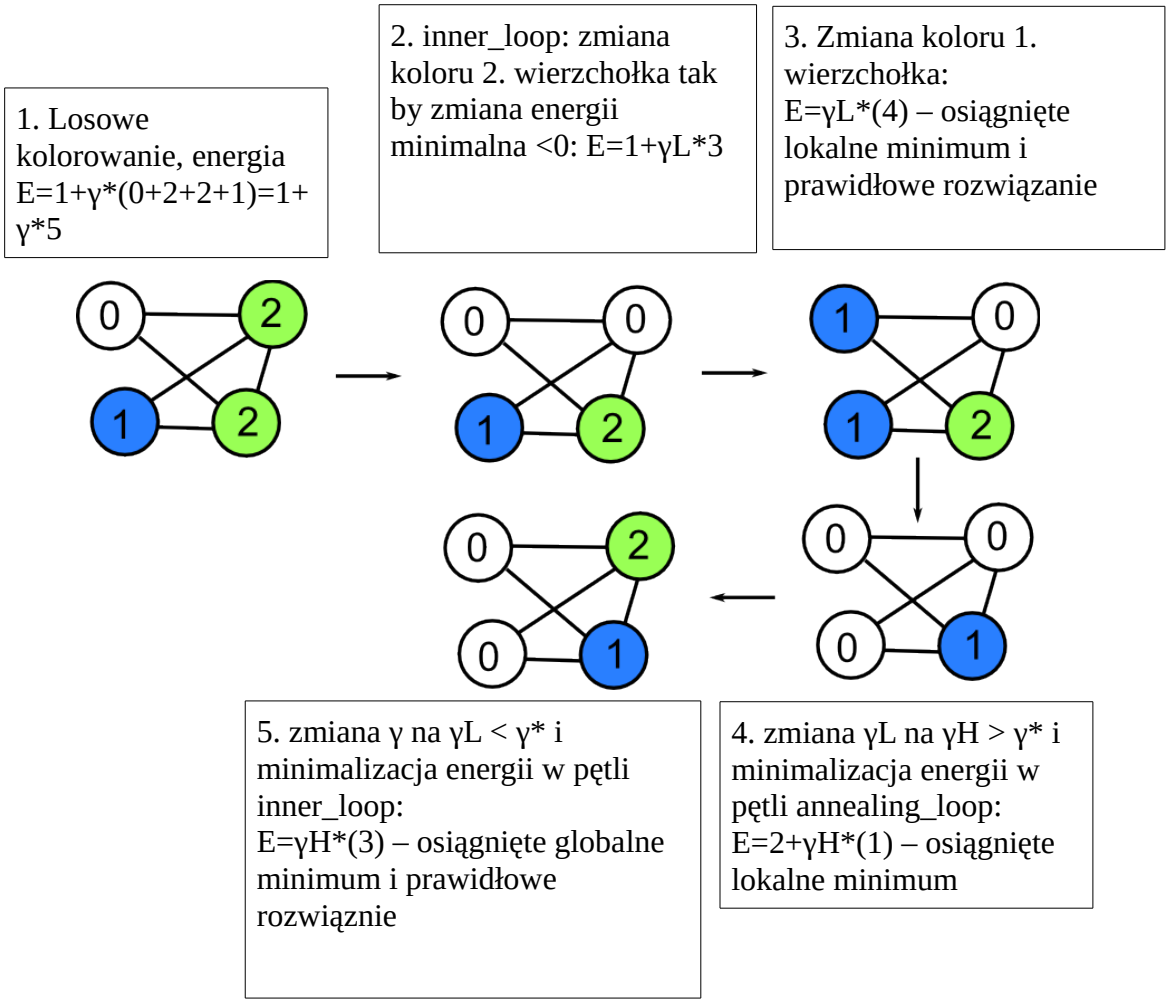
Algorytm ma wielomianową złożoność obliczeniową. Funkcja `annealing_loop` jest wywoływana ustaloną z góry liczbę razy od której zależy dokładność kolorowania. Pętla w `annealing_loop` wykona się $O(N)$ razy wywołując funkcję `inner_loop`. W

funkcji tej obliczana jest $O(N)$ razy funkcja `compute_delta` o złożoności $O(N^2k)$. Funkcję `compute_delta` można zastąpić bardziej skomplikowaną funkcją `update_delta` o mniejszej złożoności $O(Nk)$, gdzie k - maksymalna liczba kolorów. Tak więc cały algorytm osiąga złożoność obliczeniową $O(N^2k)$. Przy czym czynnik stały jest duży.

Złożoność pamięciowa algorytmu to $O(Nk)$ z małą stałą– wynika z wielkości tablicy ΔE(i,c).

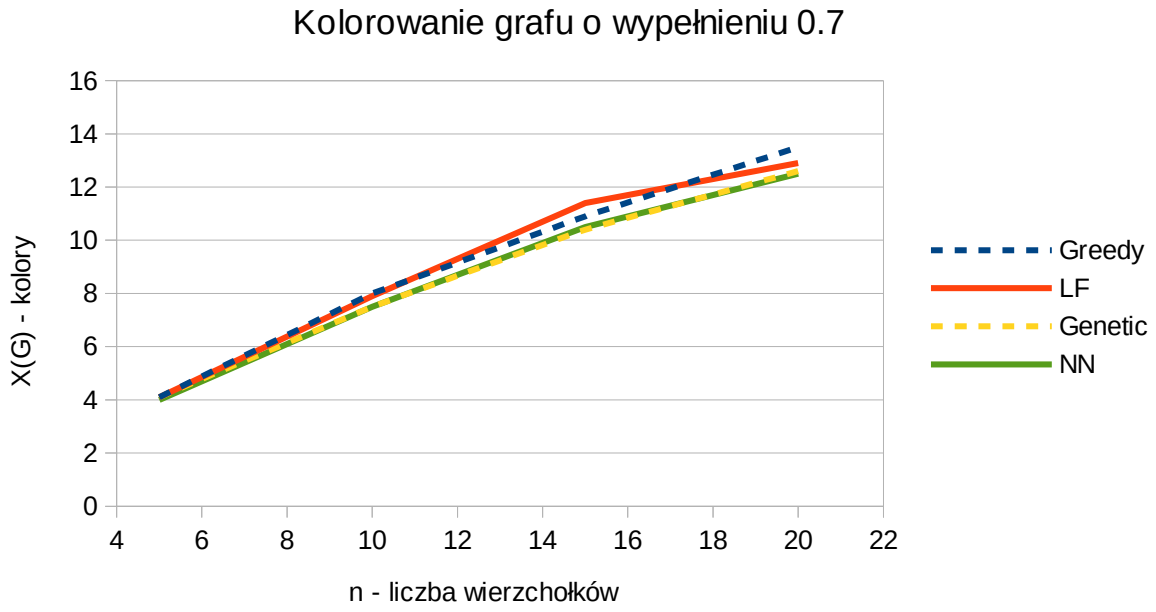
Przykład działania algorytmu

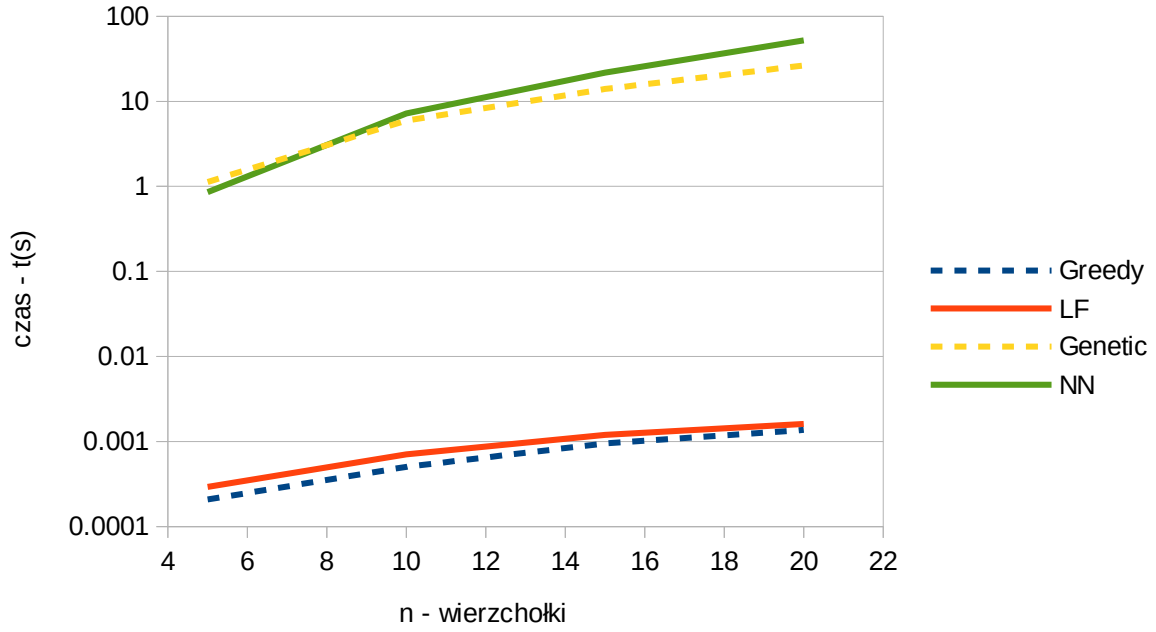
Liczby 0,1,2,3 na wierzchołkach - oznaczają tutaj kolory i są argumentami funkcji energii:



Porównanie skuteczności algorytmów przybliżonych

Na poniższym wykresie pokazano jakie średnio najlepsze kolorowanie i w jakim czasie znajduje każdy z algorytmów Greedy - zachłanny, LF - zachłanny largest first, Genetic - genetyczny, NN - Sieci neuronowe. Użyto losowych grafów o wypełnieniu 0.7.





Podsumowanie

Do rozwiązania problemu kolorowania grafu wykorzystano algorytmy reprezentujące trzy podejścia. Algorytmy dokładne pozwalają na uzyskanie każdorazowo najlepszego możliwego rozwiązania problemu, jednak jest to osiągane kosztem wykładniczej złożoności obliczeniowej. Nawet przy użyciu heurystycznego określenia maksymalnej liczby niezbędnych wartości (kolorów) i dalszej redukcji kroków innymi metodami, dla bardzo dużych, gęstych grafów zbyt długi czas działania nie pozwala na ich praktyczne użycie. Algorytmy zachłanne pozwalają na szybkie otrzymanie przybliżonego wyniku, jednak w zależności od danych wejściowych i samej implementacji dokładność uzyskanego przybliżenia może pozostawiać wiele do życzenia. Jednak nawet jeżeli taki wynik jest niewystarczająco dokładny jako ostateczny rezultat sam z siebie, to krótki czas jego otrzymania pozwala na użycie takich algorytmów do ograniczania liczby niepotrzebnych kroków które musiałby wykonać bardziej dokładny algorytm. Ostatnie dwa użyte algorytmy - genetyczny i sieci neuronowych, korzystające z mechanizmów uczenia maszynowego, pozwalają – w pewnym zakresie – na uzyskiwanie dokładniejszych przybliżeń przy zachowaniu względnie szybkiego działania nawet dla dużej liczby danych. Odbywa się to kosztem bardziej złożonej implementacji, dużego stałego narzutu obliczeniowego, co uwidacznia się na bardzo małych instancjach problemu, wreszcie może pojawić się potrzeba 'dostrajania' – empirycznego dobrania pewnych parametrów algorytmu.

Bibliografia

[1]Holland, J.; „Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence”, *U Michigan Press*. (1975), 183

[2]Kwedło, W.; „Algorytmy ewolucyjne”, *Materiały wykładowe, Politechnika Białostocka*

[3]*Di Blas, A.; Jagota, A.; Hughey, R., "Energy function-based approaches to graph coloring," Neural Networks, IEEE Transactions on , vol.13, no.1, pp.81,91, Jan 2002 doi: 10.1109/72.977273*