

Grafy i Sieci. Sprawozdanie 3.

SK11 Kolorowanie grafu za pomocą przeszukiwania z tabu.

Michał Aniserowicz, Jakub Turek

Temat projektu

SK11 Kolorowanie grafu za pomocą przeszukiwania z tabu.

Uruchamianie programu

Program uruchamiany jest z linii poleceń. Wywoływany jest za pomocą komendy `python main.py <parametry>` (należy uprzednio przejść do folderu z projektem). Podane `<parametry>` muszą być zgodne z opcjami aplikacji:

```
main.py [-h] [-o output_file] [-v] -i maximum_iterations
-s maximum_iterations_without_score_change -m memory_size [--dimacs-compatible]
input_file [input_file ...]
```

Opis parametrów:

- `-h` wyświetla pomoc do aplikacji (jest to treść zamieszczona powyżej).
- `-o` pozwala na przekierowanie wyjścia do pliku. Standardowo wszystkie komunikaty programu wypisywane są na konsoli. Podanie flagi `-o output.txt` spowoduje przekierowanie wyjścia programu do pliku `output.txt`.
- `-v` jest opcjonalnym argumentem, który uruchamia tryb „rozmowny”¹. W tym trybie prezentowane są informacje o poszczególnych iteracjach algorytmu (wraz z wynikami iteracji, czasem wykonania i zawartością pamięci „tabu”). W trybie standardowym aplikacja przedstawia jedynie wynik działania programu.
- `-i` wymagany argument, który specyfikuje wielkość jednego z kryteriów stopu: maksymalną liczbę iteracji.
- `-s` wymagany argument, który specyfikuje wielkość jednego z kryteriów stopu: maksymalną liczbę iteracji bez zmiany wyniku.
- `-m` wymagany argument, który specyfikuje rozmiar pamięci „tabu”.
- `--dimacs-compatible` opcjonalna flaga, która pozwala na czytanie danych z plików w formacie DIMACS². Standardowo program obsługuje wejście we właściwym dla siebie formacie.
- `input_file` specyfikuje nazwę pliku wejściowego, który zawiera definicję grafu, na którym przeprowadzone zostaną obliczenia. Plików wejściowych może być wiele, natomiast wymagany jest przynajmniej jeden.

Przykładowe wywołanie programu:

```
python main.py -v -o queen6_6_mem01.txt --dimacs-compatible -i 200 -s 50 -m 5 queen6_6.txt
```

Powyższe wywołanie programu uruchamia aplikację w trybie „rozmownym”, przekierowuje wyjście programu do pliku `queen6_6_mem01.txt`, pobiera dane z pliku `queen6_6.txt` w formacie DIMACS i uruchamia aplikację dla maksymalnie dwustu iteracji, pięćdziesięciu iteracji bez zmiany wyniku oraz rozmiarem pamięci „tabu” równym pięć.

¹ang. verbose.

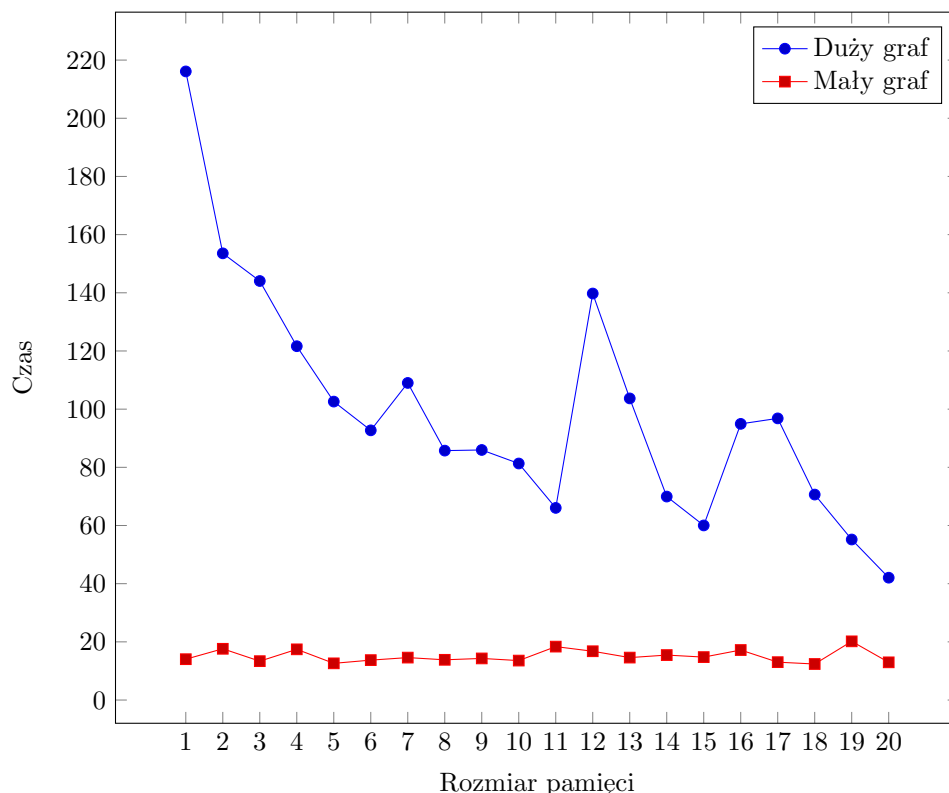
²<http://mat.gsia.cmu.edu/COLOR/general/ccformat.ps>.

Poszukiwanie optymalnego rozmiaru „tabu”

Przy poszukiwaniu optymalnego rozmiaru tabu zostały przeprowadzone dwa badania:

1. Badanie czasu obliczeń. Dla dwóch grafów, dla których znane jest optymalne kolorowanie, przeprowadzono pomiar czasu wyznaczania poprawnego kolorowania dla identycznych kryteriów stopu. Kryteria stopu były dobrane w taki sposób, aby wygaszanie algorytmu było powodowane przez osiągnięcie maksymalnej liczby iteracji bez zmiany rezultatu. Pod uwagę brane były wyłącznie próby, które kończyły się poprawnym obliczeniem rozwiązania.
2. Badanie poprawności obliczeń. Dla grafu, dla którego znane jest optymalne kolorowanie, przeprowadzono pomiar prawdopodobieństwa poprawnego obliczenia rezultatu dla identycznych kryteriów stopu.

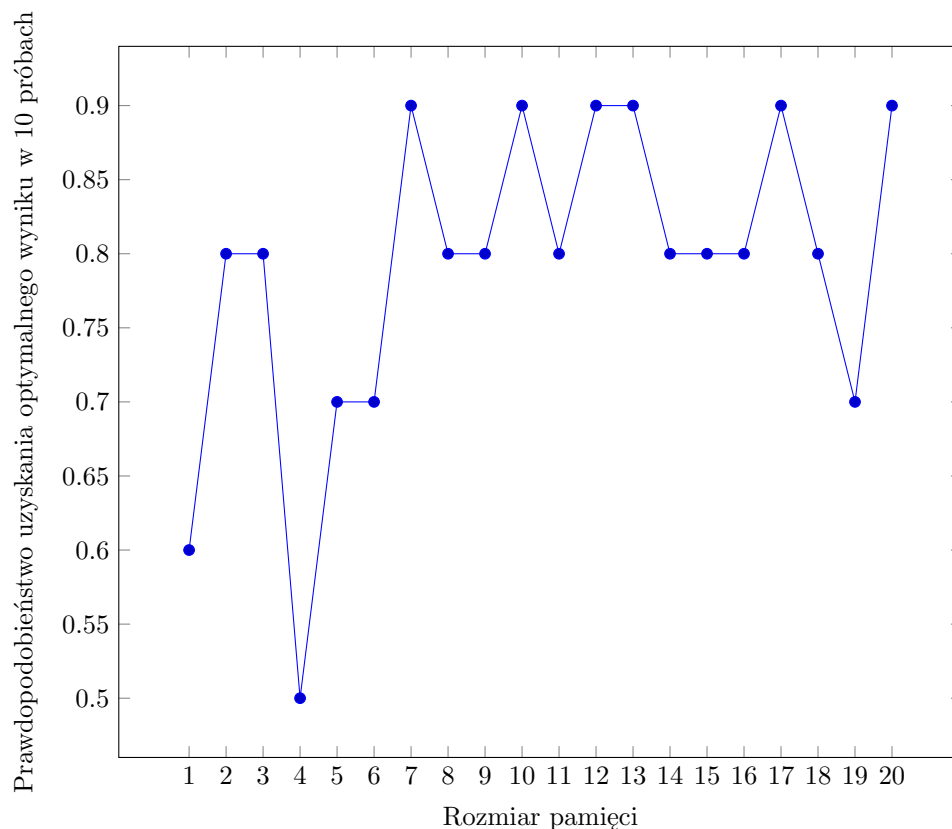
W obu przypadkach początkowe pokolorowanie grafu dobierane było w sposób losowy. Wyniki ilustrują poniższe wykresy:



Rysunek 1: Czas wykonania algorytmu w zależności od rozmiaru pamięci.

Analizując rysunki 1 oraz 2 możemy wyciągnąć następujące wnioski:

- Dla dużego grafu i małych rozmiarów pamięci „tabu” czas wykonywania algorytmu jest znacznie dłuższy niż dla większych rozmiarów.
- Dla dużego grafu czas wykonywania algorytmu drastycznie maleje dla rozmiarów „tabu” z przedziału [1; 6]. Dla większych rozmiarów pamięci krzywa czasu wykonania programu nie jest monotoniczna. Dla tych rozmiarów pamięci istotniejszy jest czynnik związany z początkowym pokolorowaniem grafu.
- Analiza czasu wykonania algorytmu dla małego grafu nie daje nam żadnych dodatkowych informacji. Dla testowanego grafu czasy wykonywania programu były zbliżone dla każdej wielkości pamięci „tabu”.



Rysunek 2: Optymalność wyniku w zależności od rozmiaru pamięci.

- Analiza prawdopodobieństwa uzyskania optymalnego wyniku pozwala stwierdzić, że optymalnym rozmiarem pamięci „tabu” jest 7. Powyżej tego progu średnie prawdopodobieństwo uzyskania optymalnego wyniku wynosi 83,57%. Poniżej tego progu średnie prawdopodobieństwo uzyskania optymalnego wyniku wynosi 68,33%.

Badanie wydajności algorytmu

Wydajność algorytmu została zmierzona dla optymalnego rozmiaru pamięci „tabu” wyznaczonego we wcześniejszej części raportu (7). Testy zostały przeprowadzone na grafach, dla których znane jest optymalne pokolorowanie. Pozostałe parametry algorytmu zostały dobrane tak, aby odnaleźć optymalne pokolorowanie.

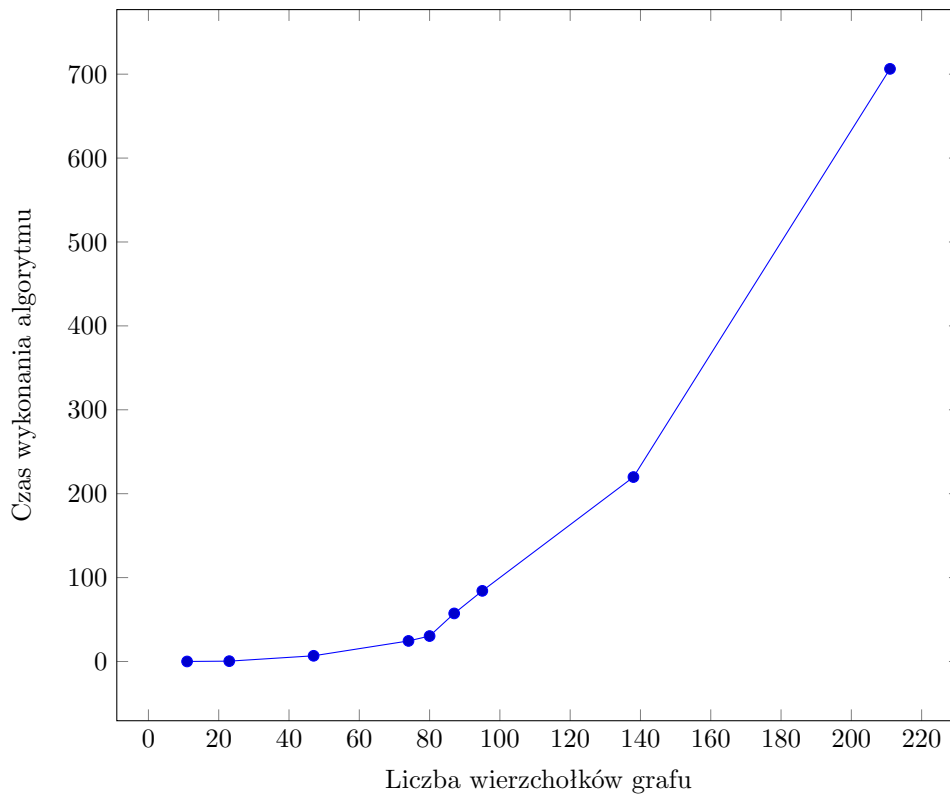
Zgodnie z przewidywaniami w raporcie drugim, z wykresu 3 wynika, że algorytm ma złożoność czasową wykładniczą.

Zwiększenie wydajności aplikacji byłoby możliwe poprzez zmianę algorytmu poruszania się po grafie. Wykorzystany w projekcie DFS można zastąpić zwykłą iteracją po liście, która będzie uporządkowana np. według rosnących identyfikatorów wierzchołków.

Poprawność algorytmu

Program został stworzony przy użyciu techniki TDD³, co zapewnia poprawność implementacji algorytmu. Kod odpowiedzialny za algorytm jest pokryty testami jednostkowymi niemalże w 100%.

³Test-Driven Development. Technika programowania, w której programista najpierw tworzy automatyczne testy jednostkowe do danej funkcjonalności, a następnie implementuje właściwą funkcjonalność.



Rysunek 3: Czas wykonania algorytmu w zależności od liczby wierzchołków grafu.

Poza testami jednostkowymi aplikacja była testowana dla grafów o określonym optymalnym pokolorowaniu, zaczerpniętych ze strony <http://mat.gsia.cmu.edu/COLOR/instances.html>. Algorytm przeszedł testy dla następujących zbiorów danych:

- `zero.in.i.1.col`, `zero.in.i.2.col`, `zero.in.i.3.col`,
- `anna.col`, `david.col`, `huck.col`, `jean.col`,
- `games120.col`,
- `queen5_5.col`,
- `myciel3.col`, `myciel4.col`, `myciel5.col`, `myciel6.col`, `myciel7.col`.

Dla każdego z grafów z powyższych zbiorów danych udało się znaleźć takie wartości kryteriów stopu, żeby dla optymalnej wielkości pamięci „tabu” aplikacja odnalazła prawidłowe pokolorowanie, które jest optymalne (optymalność stwierdzana była na podstawie danych znajdujących się w przytoczonym zbiorze danych).

Wnioski

Algorytm przeszukiwania przestrzeni z „tabu” jest wydajnym algorytmem poszukiwania jak najbardziej optymalnego pokolorowania grafu. Dzięki zastosowaniu pamięci krótkoterminowej algorytm dobrze radzi sobie z opuszczaniem lokalnych minimów dla zdefiniowanej funkcji celu. Pamięć „tabu” pozwala również ograniczyć czas niezbędny do przeprowadzenia obliczeń. Iteracyjna natura algorytmu sprawia, że jest on dobrze skalowalny - pozwala na wyznaczenie suboptymalnego rozwiązania w krótkim czasie.

Problemem algorytmu kolorowania grafu z „tabu” jest zdefiniowanie i wyznaczanie wartości konfiguracyjnych kryteriów stopu. Nie da się wyznaczyć uniwersalnej miary, która będzie właściwa dla wszystkich typów i rozmiarów grafów. W trakcie badań przeprowadzanych na potrzeby projektu wielokrotnie

zdarzały się sytuacje, gdy zadane kryteria stopu powodowały zbyt szybkie zakończenie algorytmu. Konsekwencją tego była konieczność ponownego wykonywania obliczeń. Z tego powodu algorytm nie może być stosowany do wyznaczania optymalnego pokolorowania, jeżeli nie znamy z góry liczby barw, które składają się na to kolorowanie. Pozwala on natomiast łatwo wyznaczyć kolorowanie grafu n kolorami, jeżeli takie pokolorowanie istnieje.

Ze względu na brak jasno określonych kryteriów stopu i ich wielkości algorytm wyznaczania kolorowania grafu z „tabu” mógłby znaleźć zastosowanie w systemie typu *supervised*. Decyzję o zakończeniu algorytmu podejmowałby nadzorca systemu na podstawie jakości wyniku uzyskanego na wyjściu i -tej iteracji.

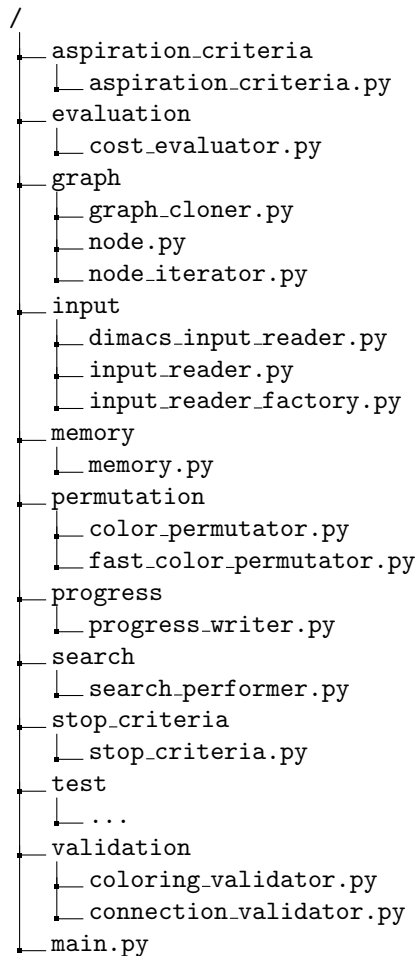
W ramach badań przeprowadzonych w trakcie projektu udało się wyznaczyć uniwersalny rozmiar pamięci „tabu”, który może być wykorzystywany dla różnych rozmiarów i typów grafów.

Przeprowadzone badania dowodzą, że szybkość i skuteczność działania algorytmu jest ściśle związana ze wstępnym pokolorowaniem grafu. Wstępne kolorowanie ma znaczący wpływ na liczbę iteracji algorytmu potrzebnych do wyznaczenia pokolorowania optymalnego, co rzutuje na konfigurację kryteriów stopu, które używane są do konkretnego typu grafu.

Dokumentacja kodu źródłowego

Kod źródłowy projektu został stworzony w języku Python. Program jest kompatybilny z wersją 2.7.x interpretera. Aplikacja testowana była w Pythonie w wersji 2.7.5, pod kontrolą systemu OS X 10.9 (Mavericks). Do uruchomienia testów jednostkowych wymagane jest zainstalowanie biblioteki `Mock`⁴ w wersji 1.0.1.

Ogólna struktura kodu źródłowego została przedstawiona na poniższym diagramie.



⁴Biblioteka została wcielona do specyfikacji języka począwszy od wersji 3.3.

Reprezentacja grafu

Graf reprezentowany jest z wykorzystaniem klasy `Node` reprezentującej wierzchołek. Ponieważ, z założenia, aplikacja operuje wyłącznie na grafach spójnych nie ma znaczenia, od którego wierzchołka rozpoczynamy analizę struktury.

```
class Node:
    Id = 0

    def __init__(self, color=None,
                  node_id=None, previous_color=None):

        self.edges = []
        self.color = color

        if node_id is not None:
            self.node_id = node_id
        else:
            self.node_id = Node.Id
            Node.Id += 1

        self.previous_color = self.color

        if previous_color is not None:
            self.previous_color = previous_color

    def add_edges(self, nodes):
        for node in nodes:
            if node not in self.edges:
                self.edges.append(node)

            if self not in node.edges:
                node.edges.append(self)

    def iterator(self):
        return NodeIterator(self)

    def get_node_of_id(self, node_id):
        for node in self.iterator():
            if node.node_id == node_id:
                return node

    def node_count(self):
        return sum(1 for _ in self.iterator())

    def get_colors_count(self):
        colors = set()

        for node in self.iterator():
            colors.add(node.color)

        return len(colors)
```

Metoda `init` służy do konstrukcji węzła. Węzeł posiada następujące składowe:

- `edges` lista wierzchołków połączonych z danym węzłem,
- `color` kolor wierzchołka,
- `node_id` identyfikator wierzchołka,
- `previous_color` poprzedni kolor wierzchołka używany do wyznaczania permutacji.

Identyfikator, jak również kolor wierzchołka, mogą być dowolnego typu (liczba, ciąg znaków...). Identyfikatory mogą, ale nie muszą być nadawane automatycznie - są wtedy typu liczbowego. Kolejne identyfikatory pobierane są ze zmiennej „statycznej” `Id`.

Metoda `add_edges` pozwala na łączenie wierzchołka z innymi wierzchołkami. Implementacja została przygotowana dla grafów nieskierowanych, a więc podczas dodawania krawędzi tworzone jest od razu wiązanie dwustronne.

Do poruszania się po grafie wykorzystywany jest iterator, który korzysta z algorytmu DFS.

Metoda `get_node_of_id` pozwala na dojście do dowolnego wierzchołka po identyfikatorze.

Metoda `node_count` zlicza liczbę wierzchołków w grafie.

Metoda `get_colors_count` zwraca liczbę kolorów, którymi w chwili obecnej pokolorowany jest graf.

Klasa `NodeIterator` dostarcza interfejs iteratora dla wierzchołka grafu. Udostępnia ona metodę `next`, która dla danego wierzchołka zwraca kolejny w porządku przeszukiwania w głąb. Przeszukiwanie w głąb

oznacza, że w pierwszej kolejności przechodzimy do pierwszego dziecka danego wierzchołka, a dopiero po powrocie algorytmu do tego samego wierzchołka przeglądamy jego kolejne dziecko. Wykorzystanie wzorca iteratora pozwala na przeglądanie grafu w wygodny sposób - używając do tego pętli `for`.

Oprócz narzędzia do przeglądania grafu zaimplementowana została też metoda do kopiowania całego grafu. Jest ona zawarta w metodzie `clone` klasy `GraphCloner`. Klonowanie grafu jest przydatne podczas wyznaczania możliwych permutacji kolorów. Wystarczy powielić cały graf i zmienić barwę analizowanego wierzchołka.

Funkcja kosztu

```
class CostEvaluator:
    def evaluate(root_node, color_set):
        c, e = self.evaluate_score_for_colors(
            root_node)
        return self.evaluate_cost(color_set, c, e)

    def evaluate_score_for_colors(root_node):
        inspected_edges, c, e = [], {}, {}

        for node in root_node.iterator():
            if node.color not in c:
                c[node.color] = 0

            c[node.color] += 1

            for child_node in node.edges:
                if {node, child_node} not in
                    inspected_edges and
                    color == child_node.color:
                    if node.color not in e:
                        e[node.color] = 0

                    e[node.color] += 1

                inspected_edges.append(
                    {node, child_node})

        return c, e

    def evaluate_cost(color_set, c, e):
        cost = 0

        for color in color_set:
            c_i, e_i = 0, 0

            if color in c:
                c_i = c[color]
            if color in e:
                e_i = e[color]

            cost += -1 * c_i ** 2 + 2 * c_i * e_i

        return cost
```

Metoda `evaluate` oblicza wartość funkcji kosztu dla danego grafu. Algorytm wykonywany jest w dwóch krokach.

W pierwszym kroku obliczane są wartości C_i oraz E_i dla każdego koloru. Metoda `evaluate_score_for_colors` wykonuje niezbędne obliczenia. Istotne jest, że wszystkie wartości wyznaczone są w czasie pojedynczego przejścia przez graf, dzięki czemu metoda jest wydajna.

Następnie zliczane są wyniki dla wszystkich kolorów znajdujących się w zbiorze. Funkcja `evaluate_cost` oblicza wartość na podstawie wzoru $f(G) = -\sum_{i=1}^k C_i^2 + \sum_{i=1}^k 2C_i E_i$, gdzie C_i oznacza liczbę wierzchołków o kolorze i , natomiast E_i oznacza liczbę krawędzi, która łączy dwa wierzchołki o kolorze i .

Ponadto klasa `CostEvaluator` posiada metodę `evaluate_score_for_permutation`. Pozwala ona na szybkie obliczanie funkcji celu dla permutacji pokolorowania grafu. Metoda przyjmuje parametry:

- `node` wierzchołek, którego kolorowanie ulegnie zmianie w trakcie permutacji.
- `target_color` docelowy kolor dla wierzchołka (po permutacji).
- `base_c` słownik wartości C_i przed wykonaniem permutacji.
- `base_e` słownik wartości E_i przed wykonaniem permutacji.
- `color_set` zbiór wszystkich kolorów.

Korzystając z powyższych parametrów metoda wyznacza funkcję kosztu dokonując pojedynczego przejścia po wierzchołku oraz wszystkich jego sąsiadach, a nie po całym grafie. Pozwala to znacząco zredukować czas szacowania funkcji kosztu dla permutacji.

Pamięć

<pre>class Memory: def __init__(self, short_term_memory_size): self.memory = [] self.short_term_memory_size = short_term_memory_size def add_to_memory(self, node, color): self.memory.append((node.node_id, color)) def clear_memory(self): self.memory = [] def get_short_term_memory(self): return self.memory[-self.short_term_memory_size:] def get_long_term_memory(self): return self.memory def is_in_short_term_memory(self, node, color): return (node.node_id, color) in self.get_short_term_memory() def is_in_long_term_memory(self, node, color): return (node.node_id, color) in self.get_long_term_memory()</pre>	<p>Klasa <code>Memory</code> realizuje pamięć poprzez przechowywanie par $(id_{wierzchołka}, kolor)$ w liście <code>memory</code>. Pamięć krótkoterminowa i długoterminowa jest realizowana z wykorzystaniem jednej pamięci fizycznej.</p> <p>Dodanie wpisu do pamięci polega na dopisaniu pary $(id_{wierzchołka}, kolor)$ na końcu pamięci.</p> <p>Metoda <code>clear_memory</code> czyści zawartość pamięci.</p> <p>Pamięć krótkoterminowa to n ostatnich wpisów listy, gdzie n to rozmiar tabu i jest definiowany zmienną <code>short_term_memory_size</code>.</p> <p>Pamięć długoterminowa to cała zawartość pamięci.</p> <p>Metoda <code>is_in_short_term_memory</code> sprawdza, czy dana kombinacja znajduje się w pamięci krótkoterminowej.</p> <p>Metoda <code>is_in_long_term_memory</code> oferuje analogiczną funkcjonalność dla pamięci długoterminowej.</p>
--	--

Kryteria stopu

Zgodnie z założeniami przedstawionymi w poprzednich raportach zaimplementowane zostały dwa kryteria stopu:

- maksymalna liczba iteracji,
- maksymalna liczba iteracji bez zmiany najlepszego wyniku.


```

class StopCriteria:
    def __init__(self, max_iters,
                  max_iters_without_change):
        self.max_iters = max_iters
        self.max_iters_without_change =
            max_iters_without_change
        self.current_iters = 0
        self.current_iters_without_change = 0
        self.previous_score = None

    def reset(self):
        self.current_iters = 0
        self.current_iters_without_change = 0
        self.previous_score = None

    def next_iteration(self, score):
        self.current_iters += 1

        if self.previous_score is None or
            self.previous_score != score:
            self.previous_score = score
            self.current_iters_without_change = 1
        else:
            self.current_iters_without_change += 1

    def should_stop(self):
        return self.current_iters >= self.max_iters
            or self.current_iters_without_change >=
                self.max_iters_without_change

```

Klasa `StopCriteria` realizuje kryteria stopu. Zlicza ona liczbę iteracji algorytmu oraz liczbę iteracji bez zmiany wyniku i porównuje je z wartościami konfiguracyjnymi ze zmiennych `max_iters` oraz `max_iters_without_change`.

Metoda `next_iteration` jest wywoływana przy każdej iteracji przeszukiwania z tabu. Parametrem tej metody jest najlepsza znaleziona wartość funkcji celu. Funkcja sprawdza czy oszacowanie uległo zmianie.

Metoda `should_stop` orzeka czy wykonywanie algorytmu powinno zakończyć się na podstawie kryteriów stopu.

Kryteria aspiracji

Kryteria aspiracji orzekają, kiedy wolno pominąć restrykcje „tabu”. Zaimplementowane zostało proste kryterium aspiracji, które pomija restrykcje „tabu” wtedy i tylko wtedy, gdy dana permutacja posiada lepsze oszacowanie funkcji celu niż najlepsze dotychczas odnalezione.

```

class AspirationCriteria:
    def __init__(self, banned_trans, best_score):
        self.banned_trans = banned_trans
        self.best_score = best_score

    def is_allowed(self, node, color, cost):
        if (node.node_id, color) not in
            self.banned_trans:
            return True

        return self.best_score is not None and
            cost < self.best_score

```

Klasa `AspirationCriteria` orzeka czy należy wziąć pod uwagę restrykcje „tabu”. Przecho-wuje listę zabronionych przejść `banned_trans` oraz najlepszy znaleziony wynik funkcji celu `best_score`.

Metoda `is_allowed` stwierdza czy wolno doko-nać dane przejście w permutacji. Jeżeli przej-ście nie jest objęte restrykcją „tabu” to za-wsze można dokonać tego przejścia. W prze-ciwnym wypadku można go dokonać tylko wte-dy, gdy oszacowanie funkcji celu dla permutacji jest lepsze niż najlepsze znalezione dotychczas oszacowanie.

Weryfikacja pokolorowania

```
class ColoringValidator:
    def is_coloring_valid(root_node):
        for node in root_node.iterator():
            for child_node in node.edges:
                if node.color == child_node.color:
                    return False

        return True
```

Metoda `is_coloring_valid` klasy `ColoringValidator` sprawdza czy w grafie nie istnieje krawędź łącząca dwa wierzchołki identycznego koloru.

Wyznaczanie permutacji pokolorowań

Wyznaczenie permutacji pokolorowań to część algorytmu, która wymagała największej optymalizacji. W przypadku podejścia naiwnego, które polegało na wyznaczeniu wszystkich możliwych sąsiedztw, a następnie oszacowania dla nich wartości funkcji celu, pojedyncze iteracje przeszukiwania z „tabu” (nawet dla stosunkowo małego grafu) trwały około 20 sekund.

Optymalizacja polegała na spostrzeżeniu, że do poprawnego działania algorytmu nie jest potrzebna pełna informacja o całym możliwym sąsiedztwie. W kolejnym kroku algorytmu odcinane były bowiem wszystkie permutacje, które w danej iteracji nie uzyskały maksymalnej wartości funkcji celu. Optymalizacja została dokonana poprzez złączenie ze sobą dwóch kroków algorytmu:

- Dla każdej badanej permutacji od razu dokonywane jest obliczanie wartości funkcji celu. Można zrobić to w bardzo wydajny sposób. Przed dokonaniem permutacji należy jednokrotnie obliczyć wartość funkcji celu dla całego grafu i zachować cząstkowe wartości C_i oraz E_i dla wszystkich kolorów. Następnie, po wykonaniu permutacji koloru jednego wierzchołka, należy przeanalizować w jaki sposób zmiana ta wpłynęła na wartość funkcji celu. Ponieważ permutacja ma zasięg lokalny, wystarczy zbadać wyłącznie sąsiadów analizowanego wierzchołka, a nie cały graf.
- Badając każde kolejne sąsiedztwo przechowujemy informację o wartości funkcji celu dla najlepszej znalezionej do tej pory permutacji. Jeżeli obliczona dla nowej permutacji wartość funkcji celu jest gorsza niż dla już znalezionej, odrzucamy takie sąsiedztwo. W przypadku, gdyby wartości okazały się identyczne, należy dopisać kolejną permutację do listy zwracanych wartości, a gdyby nowa permutacja okazała się najlepsza, należy wyczyścić listę pozostałych i wstawić w ich miejsce wyłącznie ostatnią. Na wyjściu otrzymujemy wyłącznie najlepsze permutacje dla danej iteracji.

```
class FastColorPermutator:
    def permute(self, node, color_set, criteria):
        self.permutations = []
        self.best_score = None
        self.c, self.e = CostEvaluator.
            evaluate_score_for_colors(node)

        self.find_permutations(node, color_set,
                               criteria)

        return self.permutations, self.best_score
```

Klasa `FastColorPermutator` wyznacza wszystkie najlepsze sąsiedztwa dla danej iteracji, które respektują restrykcje narzucone przez „tabu” oraz spełniają kryteria aspiracji.

Metoda `permute` inicjalizuje zmienne, które przechowują znalezione sąsiedztwa (`permutations`), najlepszy wynik (`best_score`) oraz wartości komponentów funkcji celu C_i oraz E_i (`c` i `e`) dla wszystkich kolorów przed wykonaniem permutacji. Następnie wywołuje właściwe poszukiwanie najlepszego sąsiedztwa.

<pre>def find_permutations(self, root_node, color_set, aspiration_criteria): for node in root_node.iterator(): for color in color_set: if node.color == color: continue cost = CostEvaluator. evaluate_score_for_permutation(node, color, self.c, self.e, color_set) if not aspiration_criteria.is_allowed(node, color, cost): continue if self.current_best_score is None or cost <= self.current_best_score: cloned_node = GraphCloner.clone(node) cloned_node.color = color if cost == self.current_best_score: self.permutations.append(cloned_node) else: self.permutations = [cloned_node] self.current_best_score = cost</pre>	<p>Metoda <code>find_permutations</code> odnajduje wszystkie najlepsze sąsiedztwa dla danej iteracji. Pierwsze dwie pętle <code>for</code> służą do wyznaczenia wszystkich możliwych kombinacji wierzchołków wraz ze wszystkimi kolorami.</p> <p>Następnie, szybkim algorytmem, obliczana jest wartość funkcji celu dla wyznaczonej iteracji.</p> <p>Sprawdzone jest czy dane przejście nie jest objęte restrykcjami „tabu”. Jeśli tak jest to przerywamy analizę permutacji.</p> <p>Jeżeli permutacja należy do najlepszych w danej iteracji to tworzymy jej kopię ze zmienionym kolorem wierzchołka.</p> <p>Jeśli permutacja jest równie dobra co pozostałe, dopisujemy ją na listę. Jeśli jest najlepsza, usuwamy wszystkie pozostałe z listy.</p> <p>Uaktualniamy oszacowanie dla najlepszej iteracji.</p>
--	--

Przeszukiwanie z „tabu”

Ostatnim ogniwnem implementacji jest klasa, która spaja wszystkie poprzednie moduły. Dokonuje ona właściwego przeszukiwania z „tabu” i zwraca graf reprezentujący najlepsze znalezione pokolorowanie dla zadanych parametrów.

<pre>class GraphColoringSearchPerformer: def __init__(self, stop_criteria, memory_size): self.stop_criteria = stop_criteria self.color_permutator = FastColorPermutator() self.memory = Memory(memory_size) self.best_score = None def search(self, root_node, color_set): self.memory.clear_memory() self.best_score = (root_node, CostEvaluator .evaluate(root_node, color_set)) return self.recursive_search(root_node, color_set)</pre>	<p>Klasa <code>GraphColoringSearchPerformer</code> przy pomocy metody <code>search</code> dokonuje przeszukiwania z „tabu”.</p> <p>Najlepszy znaleziony wynik przechowywany jest w parze <code>best_score</code>, na którą składają się graf oraz wartość funkcji celu dla tego grafu.</p> <p>Metoda <code>search</code> wywołuje właściwą funkcję rekurencyjną.</p>
---	--

```

def recursive_search(self, node, color_set):
    #wywołaj metodę find_permutations do
    #wyznaczenia najlepszego sąsiedztwa

    #wybierz najlepszy wynik dla iteracji
    #przy pomocy metody
    #get_best_score_for_iteration

    #dodaj do pamięci najlepszy wynik
    #dla danej iteracji

    #jeżeli najlepszy wynik dla danej
    #iteracji jest równocześnie najlepszym
    #globalnym wynikiem to uaktualnij
    #najlepszy globalny wynik

    #uaktualnij liczbę iteracji w kryteriach
    #stopu

    #jeżeli kryteria stopu są spełnione to
    #przerwij algorytm i zwróć wynik metody
    #return_score

    #wywołaj rekurencyjnie metodę
    #recursive_search

```

```

def find_permutations(self, node, color_set):
    #wyznacz permutacje przy pomocy obiektu
    #FastColorPermutator()

    #jeżeli nie ma żadnych permutacji to
    #skrót pamięć tabu o jeden element,
    #a następnie dla tych danych wyznacz
    #permutacje przy pomocy obiektu
    #FastColorPermutator()

    #powtarzaj powyższe aż do znalezienia
    #permutacji lub wyczerpania limitu
    #skracania tabu

```

```

def return_score(self):
    #zwróć graf dla najlepszego wyniku

```

```

def get_best_score_for_iteration(self,
    permutations_to_scores):
    #spośród wszystkich znalezionych permutacji
    #o tej samej wartości celu wybierz jedną,
    #która występowała do tej pory najrzadziej

```

Metoda `recursive_search` jest nieco zbyt skomplikowana, aby przedstawić ją w pełnej postaci w dokumencie. Stąd zostanie ona udokumentowana w formie opisowej, z zachowaniem podziału na metody.

Metoda `find_permutations` realizuje skracanie pamięci krótkoterminowej w przypadku, gdy jej wykorzystanie uniemożliwia wyznaczenie przynajmniej jednej dozwolonej permutacji (aspiracja domniemana).

Metoda `get_best_score_for_iteration` wykorzystuje pamięć długoterminową. W przypadku, gdy poszukiwanie sąsiedztwa zwróci wiele sąsiedztw o jednakowej wartości funkcji celu, metoda ta wybierze takie sąsiedztwo, które wymaga wykorzystania najrzadziej stosowanego przejścia. Ma to umożliwić opuszczenie minimum lokalnego.