

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.

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 Mock¹ w wersji 1.0.1.

Ogólna struktura kodu źródłowego została przedstawiona na poniższym diagramie.

```
/
├── aspiration_criteria
│   └── aspiration_criteria.py
├── evaluation
│   └── cost_evaluator.py
├── graph
│   ├── graph_cloner.py
│   ├── node.py
│   └── node_iterator.py
├── input
│   ├── dimacs_input_reader.py
│   ├── input_reader.py
│   └── input_reader_factory.py
├── memory
│   └── memory.py
├── permutation
│   ├── color_permutator.py
│   └── fast_color_permutator.py
├── progress
│   └── progress_writer.py
├── search
│   └── search_performer.py
├── stop_criteria
│   └── stop_criteria.py
├── test
│   └── ...
├── validation
│   ├── coloring_validator.py
│   └── connection_validator.py
└── main.py
```

¹Biblioteka 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 częściowe 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.

```

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

```

Metoda `find_permutations` odnajduje wszystkie najlepsze sąsiedztwa dla danej iteracji. Pierwsze dwie pętle `for` służą do wyznaczenia wszystkich możliwych kombinacji wierzchołków wraz ze wszystkimi kolorami.

Następnie, szybkim algorytmem, obliczana jest wartość funkcji celu dla wyznaczonej iteracji.

Sprawdzone jest czy dane przejście nie jest objęte restrykcjami „tabu”. Jeśli tak jest to przerywamy analizę permutacji.

Jeżeli permutacja należy do najlepszych w danej iteracji to tworzymy jej kopię ze zmienionym kolorem wierzchołka.

Jeśli permutacja jest równie dobra co pozostałe, dopisujemy ją na listę. Jeśli jest najlepsza, usuwamy wszystkie pozostałe z listy.

Uaktualniamy oszacowanie dla najlepszej iteracji.