Algorytm Floyda-Warshalla

Znajdowanie najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków w grafie ważonym.

Opis algorytmu

Algorytm F-W jest przykładem programowania dynamicznego. Służy do rozwiązania problemu najkrótszych ścieżek między wszystkimi parami wierzchołków w grafie skierowanym $G = (V, E)$. Graf G może posiadać krawędzie z ujemnymi wagami, wtedy algorytm F-W może służyć do wykrywania ujemnych cykli w grafie, to jest takich, że suma wag krawędzi w ścieżce (v_1, v_2, v_1) jest ujemna.

Algorytm F-W korzysta z tego, że jeśli najkrótsza ścieżka pomiędzy wierzchołkami v_1 i v_2 prowadzi przez wierzchołek u, to ta ścieżka jest połączeniem najkrótszych ścieżek prowadzących z v_1 do u oraz tych od u do v_2.

Aktualne wyniki zapisywane są na bieżąco do macierzy `d`, to jest tablicy $n \times n$ (n to ilość wierzchołków). Tablica ta jest wypełniona wartościami ∞ (w kodzie jest to `inf` z modułu `math`), a następnie wartości na diagonali są ustawiane na 0.

Algorytm porównuje wszystkie możliwe ścieżki pomiędzy każdą parą wierzchołków. Używane są trzy zagnieżdzone pętle `for`, każda przechodzi przez wszystkie wierzchołki grafu. Wierzchołki te nazwijmy i, j, k (w kodzie nazwy te zmienione są na `source`, `target` oraz `middle`). Dla każdego wierzchołka i porównywana jest długość ścieżki idąca bezpośrednio (o ile taka istnieje) do wierzchołka j z tą, która przechodzi przez wierzchołek k. Jeżeli ta druga jest krótsza, do

tablicy na pozycji $d_{i,j}$ (`d[source][target]`) wpisywana jest suma wag ścieżek (i, k) i (k, j).

Złożoność

Na czas działania algorytmu najbardziej wpływa potórjnie zagnieżdżona pętla `for`. Kod wewnątrz najbardziej zagnieżdzonej pętli:

```
new_weight = d[source][middle] + d[middle][target]

if target is source and new_weight < 0:
    raise ValueError('Wykryto cykl ujemny: {}~>{}~>{}'.format(
        target, middle, source))

if d[source][target] > new_weight:
    d[source][target] = new_weight
```

jest $\mathcal{O}(1)$. Każda pętla `for` jest $\mathcal{O}(|V|)$. Cała zagnieżdżona struktura pętl jest $\mathcal{O}(|V|_3)$, i jest to najbardziej złożona czasowo część algorytmu, więc cały algorytm też jest $\mathcal{O}(|V|_3)$.

Wyniki dla rzeczywistych danych

W module `main` sprawdzane jest działanie algorytmu Floyda-Warshalla dla danych z bazy danych „CITIES sgb128” przygotowanej przez Johna Burkardta. W bazie znajdują się odległości (drogowe) pomiędzy 128 miastami Stanów Zjednoczonych oraz Kanady.

W programie szukane jest największe bezwzględne skrócenie drogi w porównaniu do pierwotnej. Jest to droga z miasta Steubenville w stanie Ohio prowadząca do miasta Seattle w stanie Washington. Zmiana wynosi 3276 mil i nowa droga to 102 mile. Jest to olbrzymia różnica i po sprawdzeniu rzeczywistej odległości w internecie, okazuje się, że w linii prostej to około 2105 mil, a odległość drogowa wynosi 2531 mile. Po przyjrzeniu się sytuacji okazuje się, że dane są niestety błędne. Zapisana jest tam droga ze Steubenville do Watertown w stanie Nowy Jork, która wynosi rzekomo 72 mile, w rzeczywistości 2719 mil. Prawdopodobnie rzutuje to również na inne wyniki, a skala problemu jest nieznana (należałoby sprawdzić 8128 dróg).

Spis funkcji

W poniższym kodzie zaimportowano moduł `graphutil` jako `gu`.

Moduł `floyd`

```
def floydwarshall(graph):
    """
    Algorytm Floyda-Warshalla, zwraca macierz kosztów dojścia.

    Przyjmuje graf typu dict+dict.
    """
    nodes = gu.list_nodes(graph)
    edges = gu.list_edges(graph)

    # macierz d[n*n], każda wartość zainicjalizowana wartością nieskończoność
    # w postaci słownika
    d = {}
    for source in nodes:
        d[source] = {}
        for target in nodes:
            d[source][target] = inf if source != target else 0

    # uzupełnienie wag przejścia między wierzchołkami o krawędzie
    for edge in edges:
        source, target, weight = edge
        d[source][target] = weight

    for middle in nodes:
        for source in nodes:
            for target in nodes:
                new_weight = d[source][middle] + d[middle][target]

                if target is source and new_weight < 0:
                    raise ValueError('Wykryto cykl ujemny: {}~>{}~>{}'.format(
                        target, middle, source))

                if d[source][target] > new_weight:
                    d[source][target] = new_weight

    return d
```

Moduł `parser`

```
def parse(cities_filename, distances_filename):
    """Przetwarza dane z pliku na graf skierowany.

    Przyjmuje nazwy dwóch plików, jeden z nazwami miast, drugi z odległościami
    pomiędzy miastami.
    """
    # wczytanie miast
    pattern = re.compile(r'\w*, [A-Z][A-Z]')
    with open(cities_filename, 'r') as file:
        data = file.read()
    cities = pattern.findall(data)

    # wczytanie odległości
    with open(distances_filename, 'r') as file:
        data = file.read()
    # rozdzielenie na linie i usunięcie zbędnych
    data = data.splitlines()[7:]

    # stworzenie grafu
    graph = {}
    for i, city in enumerate(cities):
        distances = [int(distance) for distance in data[i].split()]
        gu.add_node(graph, city)
        for j, distance in enumerate(distances):
            target_city = cities[j]
            gu.add_edge_directed(graph, (city, target_city, distance))

    return graph
```

Moduł `graphutil`

```
def add_node(graph, node):
    """Wstawia wierzchołek do grafu."""
    if node not in graph:
        graph[node] = {}
```

```
def add_edge_directed(graph, edge):
    """Dodaje krawędź do grafu skierowanego."""
    source, target, weight = edge
    add_node(graph, source)
    add_node(graph, target)
    # Możemy wykluczyć pętle.
    # if source == target:
    #     raise ValueError("pętle są zabronione")
    if target not in graph[source]:
        graph[source][target] = weight
```

```
def list_nodes(graph):
    """Zwraca listę wierzchołków grafu."""
    return graph.keys()
```

```
def list_edges(graph):
    """Zwraca listę krawędzi (3-krotek) grafu skierowanego ważonego."""
    L = []
    for source in graph:
        for target in graph[source]:
            L.append((source, target, graph[source][target]))
    return L
```

```
def print_graph(graph):
    """Wypisuje postać grafu skierowanego ważonego na ekranie."""
    for source in graph:
        print(source, ": ", end="")
        for target in graph[source]:
            print("{}({})".format(target, graph[source][target]), end=" ")
        print()  # nowa linia
```

```
def random_graph(size, max_weight=10, keys=None):
    """Tworzy losowy graf o alfabetycznych etykietach.

    Parametr size oznacza ilość wierzchołków.
    Parametr max_weight (opcjonalny) oznacza największą możliwą wagę krawędzi.
    Parametr keys (opcjonalny) to własne etykiety wierzchołków.
    """
    if keys is None:
        if size <= 26:
            keys = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
                    'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
        else:
            keys = list(range(1, size+1))

    if len(keys) < size:
        raise ValueError('Zbyt mała ilość etykiet')
    keys = keys[:size]

    graph = {}
    for source in keys:
        graph[source] = {}
        for i in range(size):
            target = keys[randrange(size)]
            # wykluczone krawędzie do źródłowego węzła
            if target == source:
                continue
            weight = randrange(1, max_weight)
            graph[source][target] = weight

    return graph
```

Licencja

W programie używane są dane ze zbioru danych CITIES udostępnianych na licencji GNU LGPL.