Algorytmy Optymalizacji Dyskretnej
Lista 3
Arina Lazarenko

Treść zadania

Zaimplementuj następujące warianty algorytmu Dijkstry dla problemu najkrótszych ścieżek z jednym źródłem w sieci G = (N, A) o n wierzchołkach i m łukach z nieujemnymi kosztami. Algorytmy mają wyznaczyć najkrótsze ścieżki między zadanym źródłem s ∈ N i wszystkimi wierzchołkami
i ∈ N \ {s}:

1. **Wariant podstawowy**: *Zaimplementuj algorytm Dijkstry.*
2. **Algorytm DIALA** *(Directed Improved Algorithm for Labeling): Zaimplementuj algorytm DIALA, który jest ulepszoną wersją algorytmu Dijkstry. Algorytm DIALA wykorzystuje indeksowanie wierzchołków za pomocą kategorii i zapewnia optymalne działanie w sieciach z dużą liczbą krawędzi.*
3. **Implementacja RADIX HEAP**: *Zaimplementuj algorytm Dijkstry, który wykorzystuje strukturę danych o nazwie RADIX HEAP. Ta implementacja zapewnia optymalne działanie zarówno dla małych, jak i dużych sieci. Złożoność czasowa algorytmu jest ograniczona do O(m + n log nC) lub O(m + n log C), gdzie C oznacza maksymalny koszt krawędzi.*

Cel

Celem laboratorium 3 jest porównanie implementacji różnych wariantów algorytmu Dijkstry w kontekście problemu najkrótszych ścieżek z jednym źródłem w sieci o dyskretnych wierzchołkach i krawędziach. W ramach tego zadania należy zaimplementować co najmniej dwa warianty algorytmu Dijkstry oraz przetestować ich działanie.
Warianty algorytmu Dijkstry, które należy zaimplementować, to:

1. *Wariant podstawowy, który może wykorzystywać np. kopiec binarny.*
2. *Algorytm DIALA, którego złożoność wynosi O(m + nC), gdzie m to liczba krawędzi, n to liczba wierzchołków, a C to maksymalny koszt krawędzi.*
3. *Implementacja RADIX HEAP, której złożoność wynosi O(m+n log nC) lub O(m + n log C).*

Przed przystąpieniem do implementacji algorytmów, należy dokonać odpowiedniego wyboru technologii, tak aby zapewnić szybkie i efektywne działanie kodu. Należy również zapewnić możliwość przeprowadzenia testów dla dużych zbiorów danych testowych.
Wymagania dotyczące kompilacji i uruchamiania programów są ściśle określone. Należy dostarczyć plik Makefile umożliwiający kompilację i linkowanie programów. W przypadku języków interpretowanych, należy zamieścić odpowiednie instrukcje uruchamiania w pliku README.
Programy powinny być uruchamiane z linii poleceń z odpowiednimi parametrami. Przykładowe polecenia wywołania programu dla algorytmu Dijkstry obejmują badanie czasu wyznaczania najkrótszych ścieżek oraz obliczanie długości najkrótszych ścieżek między podanymi parami wierzchołków.
Do przeprowadzenia testów należy pobrać odpowiednie dane testowe ze strony 9th DIMACS Implementation Challenge - Shortest Paths. Następnie należy je skompilować i wygenerować dane testowe. Dane testowe zostaną wygenerowane w folderze ./inputs i będą miały określony format.
Wyniki przeprowadzonych eksperymentów należy przedstawić w sprawozdaniu w formacie PDF. Sprawozdanie powinno zawierać opis zaimplementowanych algorytmów, wyniki

eksperymentów w postaci tabel i wykresów, interpretację uzyskanych wyników oraz wnioski. Po przygotowaniu rozwiązania, plik PDF ze sprawozdaniem, pliki z kodem źródłowym, Makefile i README powinny być spakowane w archiwum ZIP, a nazwa archiwum powinna być numerem indeksu studenta.

Algorytm Dijkstry

Opis działania
Algorytm Dijkstry służy do znajdowania najkrótszych ścieżek w grafie ważonym z jednym źródłem. Działa on na zasadzie iteracyjnego rozszerzania obszaru znanych najkrótszych ścieżek, aż do osiągnięcia wszystkich wierzchołków.

Opis kroków algorytmu Dijkstry:

1. **Inicjalizacja**: Ustalamy źródłowy wierzchołek, dla którego chcemy znaleźć najkrótsze ścieżki. Dla tego wierzchołka ustawiamy odległość na 0, a dla pozostałych wierzchołków na nieskończoność. Dodajemy źródłowy wierzchołek do kolejki priorytetowej, gdzie priorytetem jest aktualna odległość.
2. **Wybór wierzchołka**: Z kolejki priorytetowej wybieramy wierzchołek o najmniejszej aktualnej odległości i oznaczamy go jako przetworzony.
3. **Aktualizacja sąsiednich wierzchołków**: Dla każdego sąsiada przetwarzanego wierzchołka, obliczamy nową odległość jako sumę aktualnej odległości przetwarzanego wierzchołka i wagi krawędzi prowadzącej do sąsiada. Jeśli ta nowa odległość jest mniejsza od aktualnej odległości sąsiada, aktualizujemy odległość sąsiada i dodajemy go do kolejki priorytetowej.
4. **Powtarzanie kroków 2 i 3**: Powtarzamy kroki 2 i 3, dopóki istnieją nieprzetworzone wierzchołki w kolejce priorytetowej. Wybieramy kolejny wierzchołek o najmniejszej aktualnej odległości i aktualizujemy odległości jego sąsiadów.
5. **Zakończenie**: Po przetworzeniu wszystkich wierzchołków, otrzymujemy najkrótsze ścieżki z źródła do każdego innego wierzchołka. Możemy również odtworzyć ścieżki, przechodząc od docelowego wierzchołka do źródła przy użyciu informacji o poprzednikach.

Złożoność obliczeniowa
Algorytm o złożoności $O(n^2)$ bo:
- Jak mamy maksymalnie pełny graf to n*(n-1)/2.
- Algorytm Dijkstry wykorzystuje jako strukturę kolejkę priorytetową z listą jednokierunkową, worst case listy to O(n).

Algorytm Dial'a

Opis działania
Tworzymy n*w + 1 pojemników, zwanych "bucketami". Numeracja pojemników odpowiada długościom ścieżek prowadzących od źródła. Na początku umieszczamy źródło w pierwszym pojemniku o labelu 0, ponieważ odległość ścieżki wynosi 0. Następnie w pętli wykonujemy następujące czynności, dopóki wszystkie pojemniki nie będą puste. Przechodzimy po pojemnikach, zaczynając od najmniejszego labela. Jeśli natrafimy na niepusty pojemnik, zatrzymujemy się w nim. W ten sposób wybieramy wierzchołek lub wierzchołki, które mają najkrótszą ścieżkę w danym momencie. Usuwamy wierzchołek z pojemnika. Dla usuniętego

wierzchołka analizujemy jego sąsiadów. Jeśli droga do sąsiada jest krótsza od dotychczasowej, umieszczamy wierzchołek w odpowiednim pojemniku (z odpowiednim labelem). Jeśli w pojemniku jest więcej niż jeden wierzchołek, usuwamy z kolejki ten pierwszy element, który dodaliśmy wcześniej.

Złożoność obliczeniowa
Algorytm jest efektywny dla grafów o niewielkich wagach krawędzi. Złożoność obliczeniowa tego algorytmu wynosi **O(m + n*C)**, gdzie m to liczba krawędzi, n to liczba wierzchołków, a C to koszt krawędzi o największej wadze.

Algorytm Radix Heap

Opis działania

1. Inicjalizacja: Tworzymy pusty radix heap.
2. Wstawianie elementu: Aby wstawić element do radix heap, najpierw dzielimy jego klucz na cyfry (najmniej znaczące cyfry są starsze, a najbardziej znaczące cyfry są młodsze). Następnie umieszczamy element w odpowiednim kuble (bucket) odpowiadającym jego najmniej znaczącej cyfrze. Jeśli kubel ten jest pusty, element staje się jedynym elementem w tym kublu. W przeciwnym razie porównujemy element z innymi elementami w kublu, aby zachować porządek wewnątrz kubla.
3. Znajdowanie najmniejszego elementu: Aby znaleźć najmniejszy element w radix heap, przeglądamy kubły od najmniej znaczącego do najbardziej znaczącego. Zaczynamy od pierwszego niepustego kubła i wybieramy najmniejszy element z tego kubła jako najmniejszy element w całym radix heap. Jeśli kubel nie jest już pusty po wyjęciu najmniejszego elementu, przenosimy się do kolejnego kubła i powtarzamy ten proces.
4. Usuwanie najmniejszego elementu: Aby usunąć najmniejszy element z radix heap, znajdujemy go używając kroku 3, a następnie usuwamy go z odpowiedniego kubła. Jeśli kubel jest teraz pusty, przechodzimy do kolejnego kubła i powtarzamy ten proces.
5. Złączanie dwóch radix heap: Aby połączyć dwa radix heap, porównujemy ich kubły od najmniej znaczących do najbardziej znaczących. Jeśli w obu radix heap znajduje się kubel o tej samej pozycji, łączymy te kubły w jeden. W przeciwnym razie kopiujemy kubel z niepustego radix heap do nowego radix heap.

Działanie radix heap opiera się na fakcie, że klucze są dzielone na cyfry, co pozwala na skuteczne przeglądanie i manipulację elementami w czasie stałym lub logarytmicznym w zależności od implementacji. Pozwala to na wydajne wykonywanie operacji takich jak wstawianie, usuwanie i znajdowanie najmniejszego elementu.

Złożoność obliczeniowa
Zmniejszenie liczby pojemników (bucketów) prowadzi do zmniejszenia liczby pojemników, które musimy odwiedzić. W rezultacie złożoność obliczeniowa wynosi **O(m + n log(n C))**, gdzie m to liczba krawędzi, n to liczba wierzchołków, a C to koszt krawędzi o największej wadze.

Eksperymenty

Poniżej widoczne są wyniki moich testów, które wykonałam. Zastosowałam porównanie złożoności czasowej pomiędzy algorytmami z racji badania efektywności Dijkstry, który należy do tzw. aglorytmów zachłannych.

Random4-n: Instancje "Random4-n"to losowo generowane grafy o określonych cechach. Te grafy mają cztery społeczności lub partycje, a "n"oznacza rozmiar grafu, czyli liczbę wierzchołków w grafie. Krawędzie między wierzchołkami są generowane losowo, zgodnie z określonymi zasadami lub parametrami zdefiniowanymi dla każdej instancji. Celem tych instancji jest dostarczenie zestawu grafów o znanych właściwościach, które mogą być wykorzystane do oceny i porównania wydajności algorytmów podziału grafów.

Instancja	Dijkstra	Dial	Radix Heap	n	m
Random4-n.10.0.gr	9	81118	203	1024	4096
Random4-n.11.0.gr	35	721260	872	2048	8192
Random4-n.12.0.gr	148	7236534	3695	4096	16384
Random4-n.13.0.gr	591		17069	8192	32768
Random4-n.14.0.gr	2932		77880	16384	65536
Random4-n.15.0.gr	13139		298861	32768	131072

Tablica 1: Porównanie czasu działania algorytmów

Square-n: Instancje "Square-n"to grafy reprezentujące kwadratowe siatki. Ponownie "n" oznacza rozmiar grafu, czyli liczbę wierzchołków lub węzłów wzdłuż każdej strony kwadratowej siatki. Na przykład, jeśli "n" wynosi 4, to rezultatem będzie graf mający 16 wierzchołków ułożonych w siatce 4x4. Krawędzie w tych instancjach zazwyczaj łączą sąsiednie wierzchołki w strukturze siatki, tworząc regularny wzór. Kwadratowe grafy siatkowe są przydatne do oceny algorytmów podziału grafów w przypadku danych strukturalnych lub siatek.

Instancja	Dijkstra	Dial	Radix Heap	n	m
Square-n.10.0.gr	7	83364	53	1024	3968
Square-n.11.0.gr	27	601841	134	2025	7920
Square-n.12.0.gr	120	6428085	463	4096	16128
Square-n.13.0.gr	443		1343	8190	32398
Square-n.14.0.gr	1915		4023	16384	65024
Square-n.15.0.gr	7155		14129	32761	130320

Tablica 2: Porównanie czasu działania algorytmów

Wnioski

Najbardziej czasowo wydajnym algorytmem jest podstawowy wariant algorytmu Dijkstry, który korzysta z kolejki priorytetowej. Z kolei algorytm Diala miał bardzo długi czas wykonania ze względu na duże zapotrzebowanie pamięciowe w porównaniu do pozostałych algorytmów. W przypadku tego algorytmu czas wykonania rósł wykładniczo w zależności od rozmiaru testowanej instancji. Okazuje się, że ten algorytm jest efektywny dla grafów o wagach mieszczących się w wąskim zakresie. Radix Heap działał nieznacznie dłużej niż wersja podstawowa, ale zdecydowanie krócej niż algorytm Diala.