Systemy Równoległe i Rozproszone

Dokumentacja projektu

Algorytm Dijkstry

Implementacja RMI

Autorzy:
Dominik Czarnota
Magdalena Jaroszyńska

Kraków, 13 czerwca 2016

Spis treści

1 Algorytm Dijkstry

Algorytm Dijkstry służy do wyszukiwania w grafie ścieżek o najmniejszym koszcie (sumie wag krawędzi) z jednego wierzchołka do pozostałych. Jest wykorzystywany do odnalezienia najkrótszej ścieżki między dwoma wybranymi wierzchołkami – np. dwoma miastami w grafie połączeń międzymiastowych.

Działanie algorytmu jest następujące:

1. Wczytanie macierzy sąsiedztwa.
2. Wypełnienie tablicy najkrótszych ścieżek wartością ∞.
3. Wczytanie wierzchołka początkowego i przypisanie mu wartości 0 w tablicy najkrótszych ścieżek.
4. Dla każdego z sąsiadujących wierzchołków zapisanie w tablicy najkrótszych ścieżek mniejszej z wartości:
   - dystans do danego sąsiedniego wierzchołka, przechowywany w tablicy najkrótszych ścieżek
   - suma dystansu do bieżącego wierzchołka i dystansu z bieżącego wierzchołka do danego sąsiedniego wierzchołka.
5. Oznaczenie rozpatrywanego wierzchołka jako odwiedzonego i wybranie kolejnego wierzchołka do rozpatrzenia – będzie to wierzchołek o najmniejszym dystansie w tablicy najkrótszych ścieżek, spośród wierzchołków jeszcze nieodwiedzonych.
6. Powtarzanie 4-5 do momentu odwiedzenia wszystkich wierzchołków lub odwiedzenia wierzchołka docelowego, czyli odnalezienia najkrótszej ścieżki do niego.

Pseudokod algorytmu:

```
Dijkstra(V, E, w, s)
    V_T = {s}
    for all v ∈ (V − V_T):
        if (s, v) exists:
            l[v] := w(s, v)
        else:
            set l[v] = ∞
    while V_T ≠ V:
        find a vertex u such that l[u] = min( l[v], v ∈ (V − V_T) )
        V_T = V_T ∪ {u}
        for all v ∈ (V − V_T):
            l[v] = min( l[v], l[u] + w(u, v) )
```

Dla każdego wierzchołka u $\in$ (V - V_T) algorytm zapisuje w tablicy l koszt najkrótszej ścieżki, łączącej wierzchołek początkowy V_T z u.

1.1 Zrównoleglony algorytm Dijkstry

W wersji zrównoleglonej, tablica zawierająca najkrótsze ścieżki zostaje podzielona pomiędzy procesy – każdy z p procesów oblicza $\frac{n}{p}$ kolejnych wartości najkrótszych ścieżek w tablicy l. Następnie proces główny scala te fragmenty i wybiera wierzchołek, który zostanie odwiedzony w następnej kolejności. Schemat ten powtarza się do momentu znalezienia najkrótszej ścieżki do wierzchołka docelowego.

2 Implementacja

Projekt został zaimplementowany w języku Java z wykorzystaniem technologii RMI (Remote Method Invocation). Aplikacja wczytuje zadany przypadek testowy i odnajduje najkrótsze ścieżki z pierwszego wierzchołka do wszystkich pozostałych.

2.1 Struktura projektu

- `Client` – aplikacja klienta – jednostki zarządzającej
  - `Client` – punkt wejścia do programu klienta
  - `DijkstraClient` – klasa implementująca algorytm Dijkstry, uruchamiająca go na wczytanych danych i przekazująca zadania serwerom
  - `Map` – klasa wczytująca plik z macierzą przyległości grafu i zapisująca ją do struktury danych
- `Server` – aplikacja serwera – jednostki wykonującej obliczenia
  - `Server` – klasa odbierająca zadanie od klienta i zwracająca jego wynik
- `Shared` – elementy współdzielone
  - `ServerInterface` – interfejs implementowany przez serwer, wykorzystywany przez klienta
- `testcases` – przykładowe przypadki testowe z różnymi macierzami sąsiedztwa
- `makefile` – zbudowanie i uruchomienie projektu
- `.gitignore` – lista wykluczająca pliki pośrednie i wynikowe z systemu kontroli wersji Git

2.2 Kompilacja i uruchomienie

Aby zbudować projekt, wystarczy uruchomić `make`.

Aby uruchomić procesy serwera: `make runserver REGISTRY_IP=127.0.0.1 PORTS="9991 9992 9993"`.

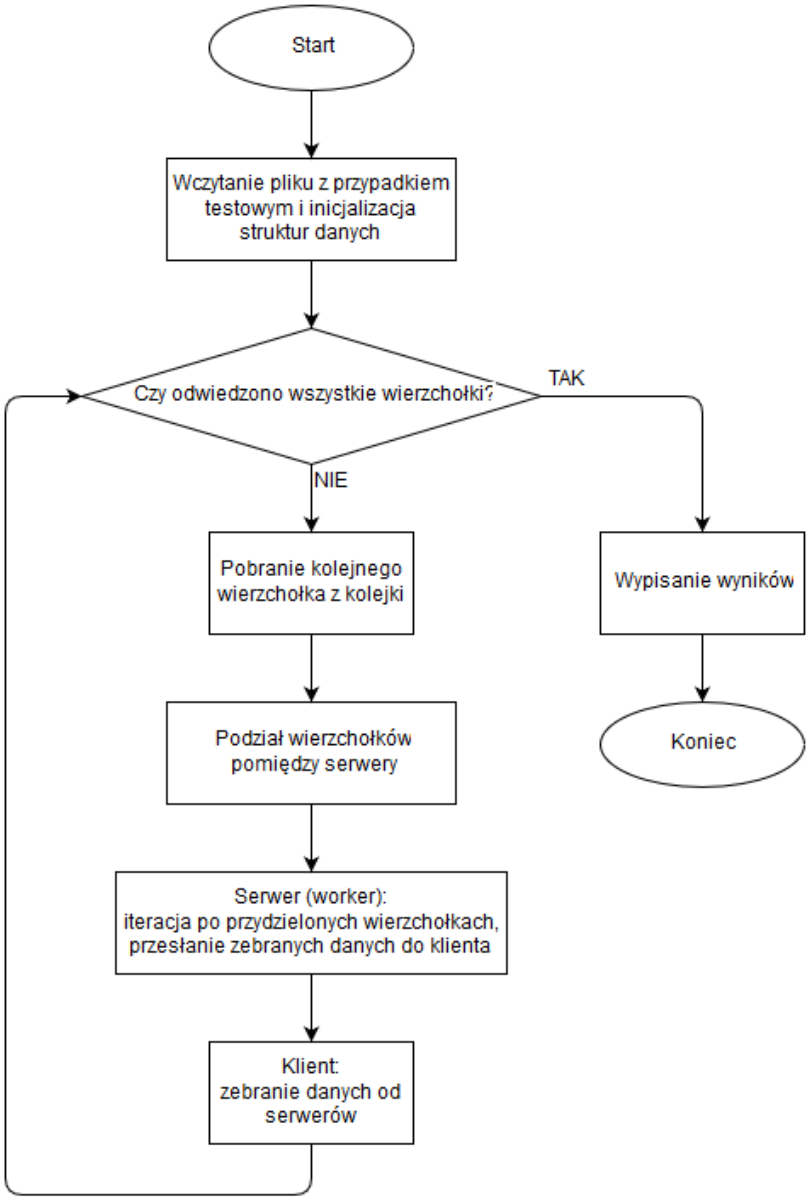
Aby uruchomić program kliencki: `make runclient REGISTRY_IP=127.0.0.1 PORTS="9991 9992 9993 TESTCASE=1"`.

Parametry:

- `REGISTY_IP` – adres IP, na jakim będą dostępne serwery
- `PORTS` – numery portów, na jakich zostaną uruchomione serwery
- `TESTCASE` – przypadek testowy.

2.3 Schemat działania aplikacji

Rysunek 1 prezentuje schemat przebiegu programu i przepływu informacji pomiędzy aplikacją kliencką, a procesami serwerów:



Rysunek 1: Schemat blokowy przebiegu programu

3 Przypadki testowe

Aplikację testowano na kilku przypadkach – paru dość prostych grafach o kilku wierzchołkach i jednym bardziej złożonym, o prawie 40 wierzchołkach.

3.1 Przypadek prosty

Jednym z prostych przypadków był następujący graf o 7 wierzchołkach:

```
A -- 1 --> B <-- 3 --> E -- 3 --.
|          |                    |
2          1                    v
|          |                    G
v          v                    ^
C -- 2 --> D -- 7 --> F -- 1 ---'
```

Odnalezione rozwiązanie:

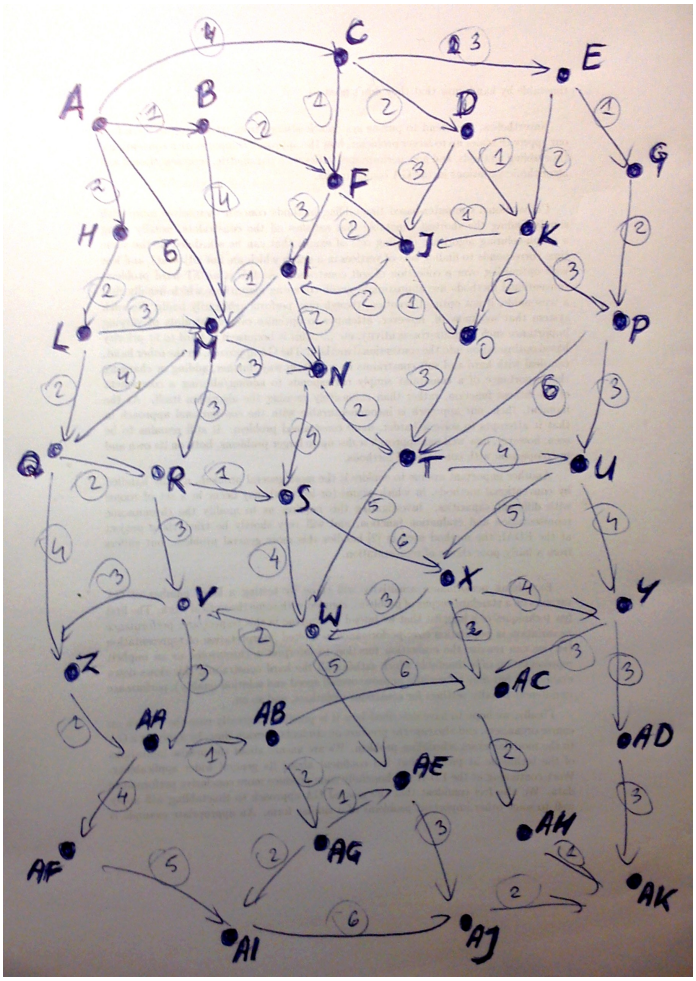
```
Distances (X means no path) = [0, 1, 2, 2, 4, 9, 7]
PrevNodes (X means initialNode) = [X, 0, 0, 1, 1, 3, 4]
```

Pierwsza tablica – `Distances` – ukazuje koszt najkrótszych ścieżek z pierwszego wierzchołka do wszystkich kolejnych.

Druga tablica – `PrevNodes` – to tablica odwrotnego przejścia: aby wyznaczyć najkrótszą trasę do wierzchołka o zadanym indeksie, należy znaleźć ten indeks. Wartość zapisana pod nim oznacza indeks poprzedniego elementu. W ten sposób można prześledzić trasę aż do wierzchołka początkowego.

3.2 Przypadek złożony

Tym razem algorytm uruchomiono na grafie o 37 wierzchołkach, którego schemat został przedstawiony na rysunku 2.



Rysunek 2: Graf przypadku testowego nr 3

Odnalezione rozwiązanie:

```
Distances (X means no path) = [0, 1, 4, 6, 7, 3, 8, 2, 6, 5, 6, 4, 5, 8, 6, 9, 6, 8, 9, 9,
    12, 11, 13, 15, 16, 10, 12, 13, 17, 19, 18, 16, 15, 19, 17, 21, 20]
PrevNodes (X means initialNode) = [X, 0, 0, 2, 2, 1, 4, 0, 5, 5, 9, 7, 1, 8, 9, 10, 11, 12,
    17, 14, 15, 17, 18, 18, 20, 16, 25, 26, 23, 24, 22, 26, 27, 28, 32, 30, 33]
```

Liczby w rozwiązaniu odpowiadają kolejnym oznaczeniom literowym na rysunku 2.

Literatura

[1] Grama, Gupta, „Introduction to parallel computing", p. 10.2-10.3.