Grafy i Sieci

Projekt 2018 Z

Porównanie implementacji wybranego algorytmu grafowego w różnych językach programowania

Igor Markiewicz
Wojciech Węgierek

Prowadzący – dr inż. Sebastian Kozłowski

Spis treści

1 Opis tematu

Głównym zadaniem jest zaimplementowanie tej samej wersji algorytmu Forda–Fulkersona (znajdywanie maksymalnego przepływu w sieci przepływowej) używając dwóch języków programowania: Java, Python, a następnie porównanie wydajności obliczeniowej obu rozwiązań dla takich samych parametrów wejściowych w zależności od rozmiaru sieci. Oprócz samych implementacji algorytmu oraz ich oceny, zostanie zrealizowany również program pozwalający na półlosowe generowanie danych wejściowych.

Errata – po wstępnych testach postanowiono zmienić język Python na język C++.

2 Opis algorytmów

Poniżej przedstawiono opisy dwóch głównych algorytmów wykorzystywanych w niniejszym projekcie.

2.1 Algorytm Edmondsa–Karpa

Algorytm Edmondsa–Karpa [1, 2] jest algorytmem Forda–Fulkersona w którym zamiast arbitralnego przeszukiwania ścieżek, zastosowano do tego celu przeszukiwanie wszerz (*breadth–first search*). Zmniejsza to złożoność do $O\left(VE^2\right)$:

Wejście algorytmu:

- `n` – liczba wierzchołków w grafie
- `C` – macierz `n` × `n` przepustowości kanałów o wartościach rzeczywistych nieujemnych
- `s` – numer wierzchołka będącego źródłem sieci
- `t` – numer wierzchołka będącego ujściem sieci

Wyjście algorytmu:

- `F` – macierz `n` × `n` przepływów netto
- `f_max` – wartość maksymalnego przepływu sieciowego

Elementy pomocnicze:

- `Q` – kolejka FIFO przechowująca wierzchołki dla metody BFS
- `P` – tablica n elementowa przechowująca ścieżki (poprzedniki) wyszukiwane przez BFS
- CFP – tablica `n` elementowa przechowująca wartości `c_f(p)` (przepustowości rezydualne) dla ścieżki kończącej się w danym węźle sieci `p`

- `cp` – przechowuje wartość przepustowości rezydualnej
- `x`, `y` – przechowują numery wierzchołków połączonych krawędzią
- `esc` – zmienna służąca do wychodzenia z zagnieżdżonej pętli

1: inicjalizujemy i zerujemy `f_max`, `F`, `Q`, `P`, `CFP`, `cp`, `x`, `y`
2: inicjalizujemy i ustawiamy `esc := false`
3: **Dopóki** `(true)`:
    4: ustaw każdy element tablicy `P` jako: `-1`
    5: zapobiegnij wybieraniu źródła przez BFS, ustaw `P(s) := -2`
    6: dla źródła ustawiamy `CFP[s] :=` ∞
    7: zerujemy kolejkę `Q`
    8: wkładamy do kolejki `Q` źródło `s`: `Q.put(s)`
    9: ustawiamy `esc := false`
    10: **Dopóki** `(kolejka Q nie jest pusta)`:
        11: pobieramy i zdejmujemy element z kolejki `Q`: `x = Q.get()`
        12: **Dla** `(y = 0, 1 ... n - 1)` wykonujemy BFS:
            13: wyznaczamy przepustowość rezydualną kanału `(x, y)`:
                `cp := C[x][y] - F[x][y]`
            14: **Jeśli** `((!is_close(cp, 0))` $\wedge$ `(P[y] == -1))`:
                # jeśli `cp` jest wystarczająco różne od zera oraz nie odwiedziliśmy jeszcze wierzchołka `y`
                15: zapamiętujemy poprzednika na ścieżce: `P[y] = x`
                16: obliczamy przepustowość rezydualną do wierzchołka `y`:
                    `CFP[y] := min(CFP[x], cp)`
                17: **Jeśli** `(y == t)`: # znaleźliśmy ścieżkę rozszerzającą
                    18: zwiększamy przepływ sieciowy: `f_max := f_max + CFP[t]`
                    19: ustawiamy `y_tmp := y`
                    20: **Dopóki** `(y_tmp != s)`:
                        # cofamy się po ścieżce rozszerzającej od ujścia `t` do źródła `s`
                        21: `(x, y_tmp)` jest krawędzią roszerzającą: `x := P[y_tmp]`
                        22: `F[x][y_tmp] += CFP[t]`
                            # w kierunku zgodnym ze ścieżką zwiększamy przepływ
                        23: `F[y_tmp][x] -= CFP[t]`
                            # w kierunku przeciwnym zmniejszamy przepływ
                        24: przechodzimy do następnej krawędzi ścieżki: `y_tmp := x`
                    25: koniec **Dopóki**
                    26: znaleźliśmy ścieżkę rozszerzającą więc wychodzimy do głównej pętli:
                        `esc := true`, **`break`**
                27: koniec **Jeśli**
                28: włóż wierzchołek `y` do kolejki `Q` jeśli nie jest ujściem `t` i kontynuuj BFS:
                    `Q.put(t)`
            29: koniec **Jeśli**

30: koniec **Dla**
31: **Jeśli** `(esc)`: **break** # wychodzimy z **Dopóki**, jeśli została znaleziona
ścieżka rozszerzająca
32: koniec **Jeśli**
33: koniec **Dopóki**
34: **Jeśli** `(!esc)`: **break** # Jeśli nie znaleziono ścieżki rozszerzającej, to `esc := false` i w
tym miejscu nastąpi wyjście z głównej pętli **Dopóki**
35: koniec **Jeśli**
36: koniec **Dopóki**

W związku z tym że wartości przepływów mogą być liczbami niecałkowitymi, postanowio w linii **14** zastąpić ostre porównywanie z zerem wartości `cp`, przez przybliżone (`is_close`):

$$|a - b| \leq \max(rel\_tol \cdot \max(|a|, |b|), abs\_tol)$$

Gdzie domyślnie:

- $rel\_tol = 10^{-9}$
- $abs\_tol = 0$

2.2 Algorytm generacji sieci przepływowych

W trakcie generacji przykładowych sieci przyjęto następujące założenia:

- do źródła `s` nie wchodzą żadne kanały
- z ujścia `t` nie wychodzą żadne kanały
- nie ma pętli własnych
- każdy graf ma zapewnioną słabą spójność – na początku łączymy w drzewo wierzchołki od źródła, przez kolejne elementy do ujścia: $0 \to 1 \ldots n-1$ (węzeł 0 stanowi źródło, a $n-1$ ujście)
- wybór dodatkowej liczby sąsiadów danego wierzchołka odbywa się zgodnie z dyskretnym rozkładem jednostajnym, podobnie jak losowanie (bez zwracania) identyfikatorów tychże sąsiadów
- wagi połączeń są losowane zgodnie z rozkładem jednostajnym $\mathcal{U}_{[1,10]}$

Wejście algorytmu:

- `n` – liczba wierzchołków w grafie

Wyjście algorytmu:

- `C` – macierz `n` × `n` przepustowości kanałów o wartościach rzeczywistych nieujemnych

Główne elementy pomocnicze:

- `connections` – wektor o rozmiarze `n`, który dla każdego wierzchołka przetrzymuje wszytkich możliwych jego sąsiadów, uwzględniając połączenie w drzewo, to że do źródła nic nie wpływa a z ujścia nie wypływa oraz brak pętli własnych

- `number_of_additional_connections` – wylosowana liczba dodatkowych połączeń dla danego wierzchołka

- `selected_additional_neighbors` – dla danego wierzchołka, wektor o rozmiarze `number_of_additional_connections` wylosowanych sąsiadów z dostępnych możliwych

1: inicjalizujemy i zerujemy `connections`, `number_of_additional_connections`, `selected_additional_neighbors`, C

2: **Dla** `(i = 0, 1 ... n - 2)`:

3: `C[i][i + 1] =` $\mathcal{U}_{[1,10]}(t)$ # tworzymy drzewo

4: **Dla** `(i = 0, 1 ... n - 1)`:

5: **Jeśli** `(i != n - 1)`: # jeśli nie jesteśmy w ujściu

6: zainicjalizuj wektor `connections_tmp`

7: **Dla** `(j = 1, 2 ... n - 1)`: # pomiń źródło

8: **Jeśli** `((i != j) && (i != j - 1))`: # jeśli nie odnosimy się sami do siebie oraz do # swojego bezpośredniego następnika (wykorzystaliśmy go na drzewo)

9: `connections_tmp.push_back(j)` # dodaj na koniec do `connections_tmp` # węzeł `j`

10: koniec **Jeśli**

11: koniec **Dla**

12: `connections.push_back(connections_tmp)` # dodaj na koniec `connections` wektor # możliwych sąsiadów dla wierzchołka `i` `connections_tmp`

13: **W przeciwnym razie jeśli** jesteśmy w ujściu, dodaj pusty wektor sąsiadów: `connections.push_back(empty_vector)`

14: koniec **Jeśli**

15: koniec **Dla**

16: **Dla** `(i = 0, 1 ... n - 2)`: # dla każdego wierzchołka poza ujściem

17: `number_of_additional_connections = random_int(0, connections[i].size())` # wylosuj liczbę dodatkowych połączeń dla wierzchołka `i` na podstawie długości # dostępnego wektora możliwych, dodatkowych połączeń tego wierzchołka # (minimalnie `0`, maksymalnie rozmiar `connections[i]`)

18: **Jeśli** `(number_of_additional_connections > 0)`:

19: wyznacz wektor wylosowanych dodatkowych sąsiadów dla wierzchołka `i`: `selected_additional_neighbors = random_choice_without_replacement(connections[i], number_of_additional_connections)`

Dla `(j = 0, 1 ... number_of_additional_connections - 1)`:
  znajdź i usuń element `selected_additional_neighbors[j]` z `connections[i]`
  `C[i][selected_additional_neighbors[j]]` = $\mathcal{U}_{[1,10]}(t)$
koniec **Dla**
koniec **Jeśli**
koniec **Dla**

3 Implementacje

Algorytm Edmondsa–Karpa zostanie zaimplementowany w językach C++ oraz Java, zaś program do generacji sieci w języku C++. Cały system będzie spięty językiem skryptowym powłoki Bash, zarządzającym testami. Dokładne wersje języków, jak również ewentualne bardziej zaawansowanych bibliotek zostaną podane w ostatniej części projektu.

4 Plan badań

Przedmiotem badań będzie porównanie czasu wykonania algorytmu Edmondsa–Karpa zaimplementowanego w językach C++ oraz Java dla takich samych parametrów wejściowych, w zależności od liczby wierzchołków. Jako kolejne kroki planuje się:

- po implementacji algorytmu generowania sieci oraz Edmondsa–Karpa, przetestowanie ich dla małych zbiorów danych w celu sprawdzenia poprawności ich działania
- określenie zakresu zmian liczby wierzchołków generowanych grafów
- określenie wystarczającej ilości powtórzeń eksperymentu dla danej liczby wierzchołków, w celu uzyskania miarodajnych wyników (najprawdopodobniej kryterium będzie stanowić względna zmiana średniej arytmetycznej wyników – jeśli będzie ona poniżej pewnego progu, uznajemy że dana liczba powtórzeń jest wystarczająca)
- zebranie wyników
- analiza wyników – porównanie statystyk, wykresy i wykresy pudełkowe
- wnioski

5 Bibliografia

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Wprowadzenie do algorytmów*, wyd. VII – 2 dodruk, Warszawa 2013

[2] `https://eduinf.waw.pl/inf/alg/001_search/0146.php`