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 $O(VE^2)$:

Wejście algorytmu:

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

Wyjście algorytmu:

- `F` – macierz `n` $\times$ `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 rozszerzającą: `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)`

20: **Dla** `(j = 0, 1 ... number_of_additional_connections - 1)`:
21: znajdź i usuń element `selected_additional_neighbors[j]` z `connections[i]`
22: `C[i][selected_additional_neighbors[j]] =` $\mathcal{U}_{[1,10]}(t)$
23: koniec **Dla**
24: koniec **Jeśli**
25: 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

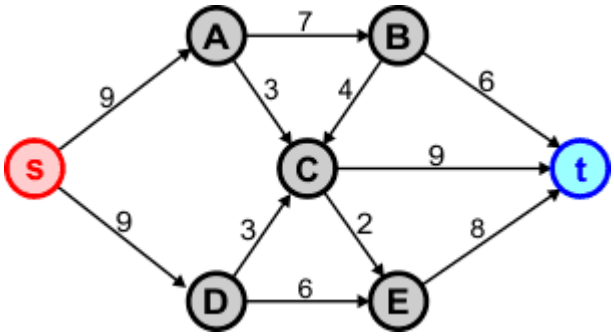
Przedmiotem badań będzie porównanie czasu wykonania algorytmu Edmondsa–Karpa zaimplementowanego w językach C++ oraz Java dla takich samych parametrów wejściowych, w zależności od liczby wierzchołków. Jako kolejne kroki planuje się:

- 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

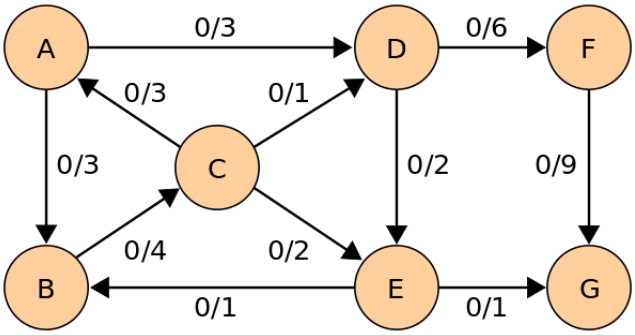
5 Badania

5.1 Testy dla znanych sieci

Do sprawdzenia poprawności zaimplementowanego algorytmu Edmondsa–Karpa posłużono się dwoma przykładami z wyliczonymi wcześniej parametrmi



Rys. 1: Plik *test_network_1* [1]



Rys. 2: Plik *test_network_2* [2]

W przypadku algorytmu generacji sieci, przejrzano kilka losowo utworzonych grafów dla różnej liczby wierzchołków w celu sprawdzenia czy spełniaja one wszystkie wymagane własności.

5.2 Określenie zakresu zmian liczby wierzchołków oraz ilości powtórzeń eksperymentu

Jako wartości liczby węzłów po wstępnych testach zostały wybrane liczby: 100, 250, 500, 750, 1000, 1500. Jest to kompromis pomiędzy czasem obliczeń, a stosunkiem czasów działań obu implementacj (wraz ze wzrostem liczby wierzchołków zauważono że coraz bardziej czasy działań się różnią, więc zdecydowano się dla większych wartości liczby wierzchołków na ich rzadsze

[1] `https://eduinf.waw.pl/inf/alg/001_search/0146.php`
[2] `https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm`

testowanie). Jako liczbę powtórzeń (taką samą dla każdej ilości wierzchołków w obu algorytmach) przyjęto początkowo wartość 1000. Jako metrykę przyjęto moduł względnej zmiany wartości średniej arytmetycznej pomiędzy kolejnymi koszykami o rozmiarach 200, 400, 600, 800 i 1000

$$diff(i,j) = \left| 100 \cdot \frac{j-i}{i} \right|$$

gdzie i to średnia arytmetyczna mniejszego koszyka, a j jego bezpośredniego, większego sąsiada.

liczba wierzchołków	**koszyki**	**różnice dla C++ [%]**	**różnice dla Java [%]**
100	200_400	0.68	5.59
100	400_600	2.46	3.02
100	600_800	2.42	3.48
100	800_1000	1.24	1.02
250	200_400	4.92	3.62
250	400_600	0.72	1.23
250	600_800	3.06	2.11
250	800_1000	0.63	0.94
500	200_400	3.91	3.81
500	400_600	1.40	1.16
500	600_800	0.27	0.48
500	800_1000	0.52	0.61
750	200_400	2.19	2.23
750	400_600	0.75	0.98
750	600_800	0.19	0.14
750	800_1000	0.69	0.44
1000	200_400	1.27	1.40
1000	400_600	5.34	4.79
1000	600_800	0.86	0.37
1000	800_1000	0.50	0.61
1500	200_400	5.64	6.12
1500	400_600	2.51	1.88
1500	600_800	1.58	1.61
1500	800_1000	0.11	0.47

Tab. 1: Wartości bezwzględne procentowych zmian między koszykami

Możemy zauważyć że choć nie zawsze zwiększając liczbę pomiarów otrzymujemy mniejszą różnicę między średnimi, to wahania między koszykiem z 800. elementami a z 1000. alementów nie przekraczając ok. 1.5 %, co możemy uznać za wartość stabilną. Dlatego ostatecznie jako liczbę powtórzeń działania algorytmu dla danej liczby wierzchołków przyjęto jako 1000.

5.3 Założenia

Poczyniono trzy założenia odnośnie przeprowadzanych eksperymentów

- badamy czas z rozdzielczością 1 ms
- ponieważ środowisko Java Virtual Machine potrafi optymalizować kod pod kątem wydajności (np: Just-In-Time compiler, optymalizacja pod kątem instrukcji procesora) zdecydowano się użyc dla języka C++ optymalizacji typu O3
- niewielka część wyników dla 100. wierzchołków w przypadku języka C++ wynosiła 0 (co oznacza że czas działania był mniejszy niż przyjęta rozdzielczość), przyjęto je zaokrąglić w analizie do 1 ms

5.4 Wyniki

5.4.1 Statystyki

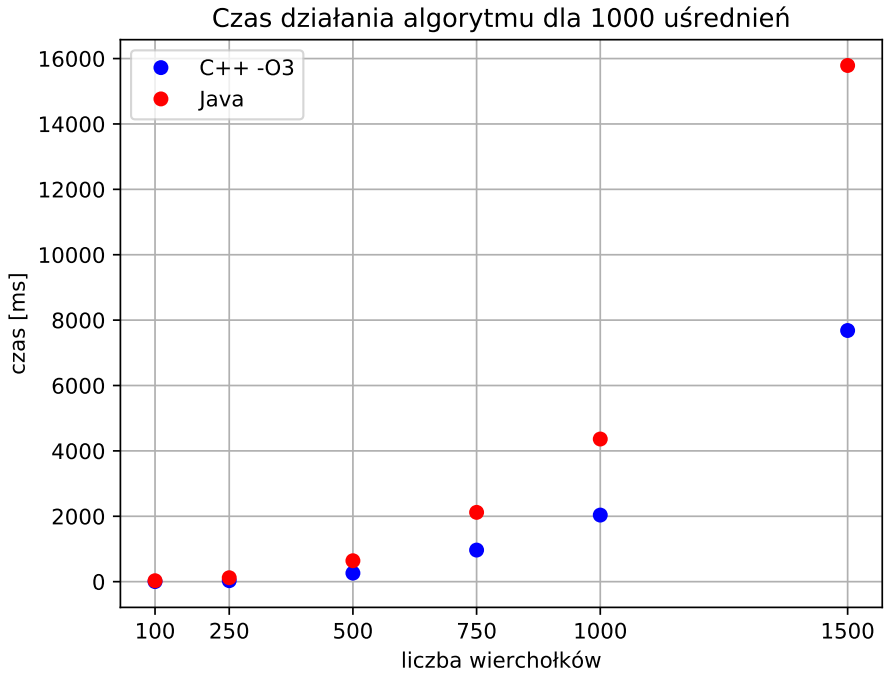
	minimum [ms]		**maximum [ms]**		**średnia [ms]**		**mediana [ms]**		**odchylenie standardowe [ms]**	
liczba wierzchołków	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**
100	1.00	1.00	18.00	92.00	2.38	28.89	2.00	32.00	1.88	12.00
250	1.00	2.00	82.00	316.00	30.19	119.86	28.00	143.50	24.36	81.94
500	1.00	2.00	550.00	1358.00	261.18	639.89	311.00	764.50	188.86	398.13
750	1.00	5.00	1855.00	3910.00	966.23	2119.89	1086.50	2604.50	596.53	1184.87
1000	1.00	3.00	4128.00	9485.00	2036.44	4362.86	2755.50	5947.00	1533.20	3238.99
1500	1.00	12.00	15286.00	33801.00	7682.16	15788.43	9667.00	20228.00	4941.64	9923.10

Tab. 2: Statystyki wyników

Wnioski

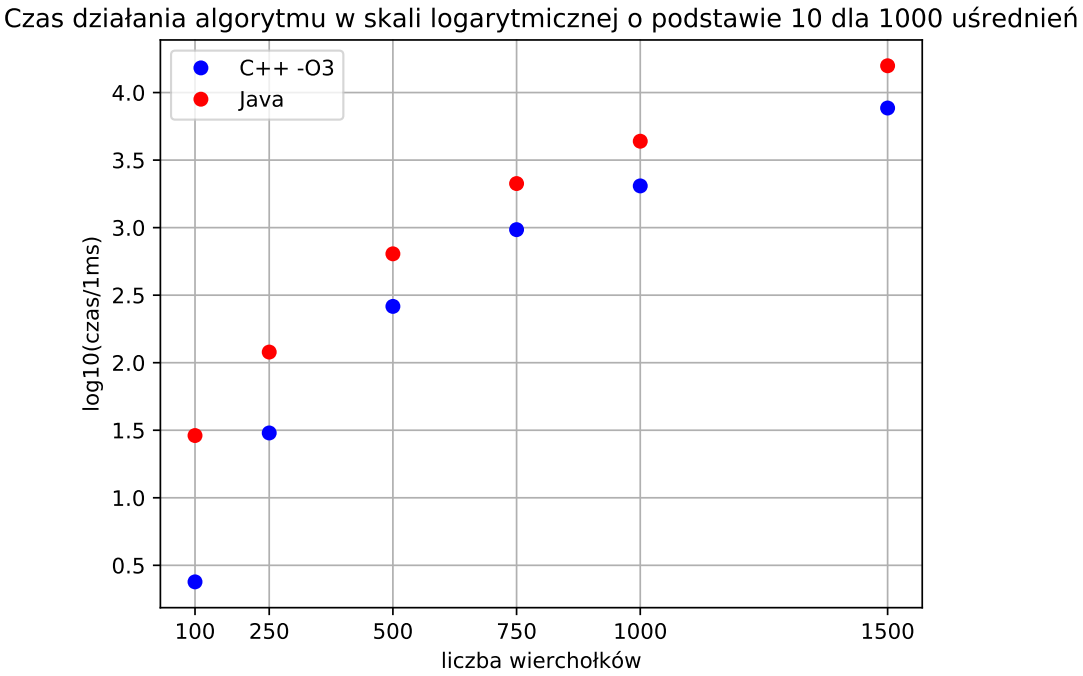
- Każde z minimów dla języka C++ jest mniejsze od odpowiadającemu mu minimum dla języka Java. Podobnie prezentuje się sytuacja z maksimami, średnimi, medianami oraz odchyleniami standardowymi.
- Wraz ze wzrostem liczby wierzchołków średnia arytmetyczna oraz mediana coraz bardziej się różnią, co sugeruje że rozkłady stają się coraz bardziej niesymetryczne. Dzieje się tak najprawdopodobniej ze względu na to że wraz ze wzrostem liczby wierzchołków rośnie dramatycznie liczba możliwych kombinacji postaci sieci, a my cały czas zachowujemy tę samą liczbę powtórzeń eksperymentu. Być może lepsze rezultaty zostałyby osiągnięte dla większej liczby powtowtórzeń, lub jej adaptacyjnym doborze.
- Jeśli przyjmiemy za estymator miary rozrzutu rozkładu odchylenie standardowe, to możemy zaobserwować że podane rozkłady są bardzo zróżnicowane.

5.4.2 Wykresy

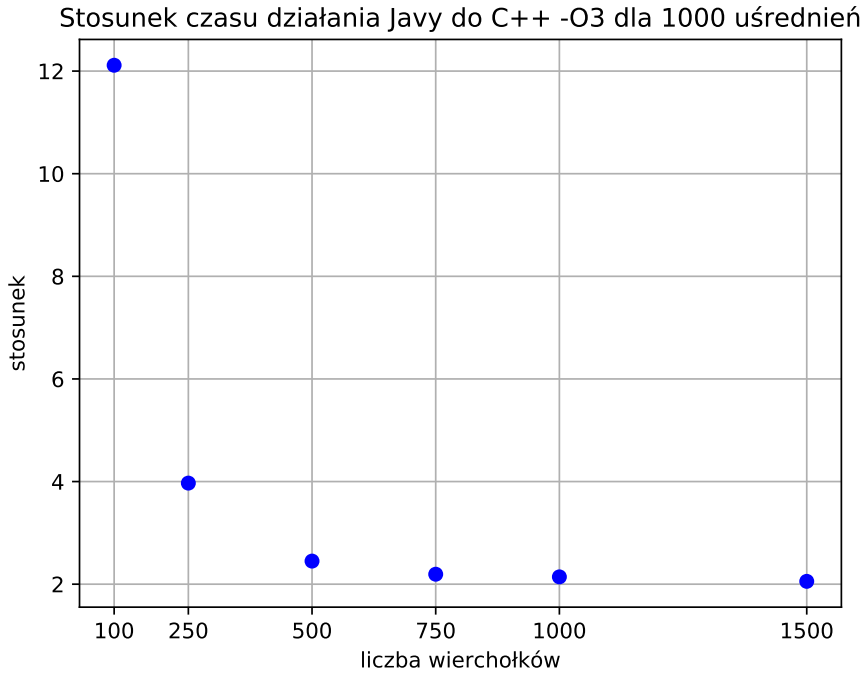


Rys. 3: Czas działania algorytmu

Możemy zaobserwować że zależność czasu działania obu implementacji od liczby wierzchołków jest nieliniowa oraz że różnice czasów między nimi rosną wraz ze wzrostem liczby wierzchołków. Dla mniejszych liczb wierzchołków na podanym wyżej wykresie granice się zacierają, dlatego postanowiono go narysować w skali logarytmicznej.



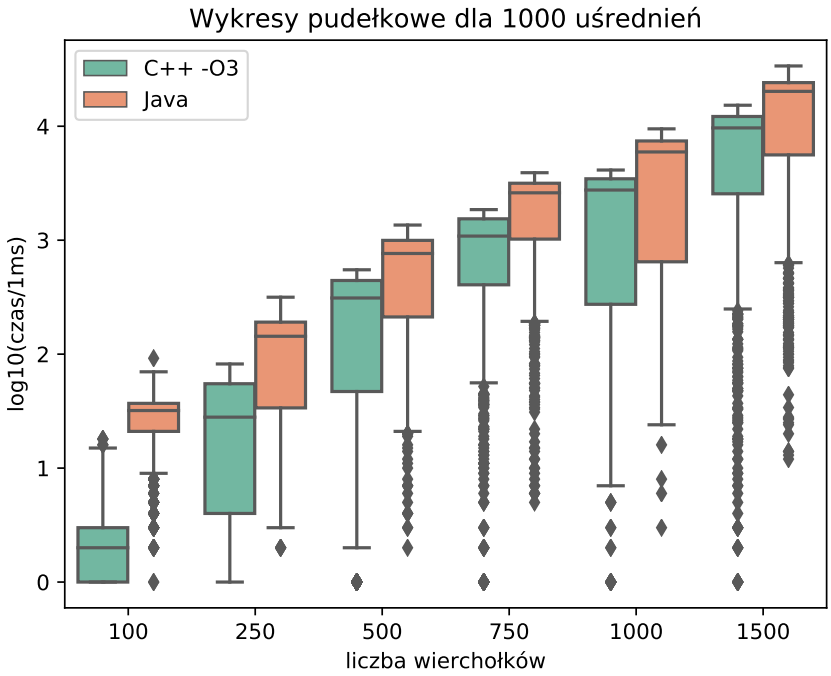
Rys. 4: Czas działania algorytmu w skali logarytmicznej o pdstawie 10



Rys. 5: Stosunek (nielogarytmiczny) czasów działania implementacji

Możemy zaobserwować że największy stosunek czasów działania osiągamy dla 100 wierzchołków,

a następnie maleje on asymptotycznie do wartości ok. 2.



Rys. 6: Wykres pudełkowy dla czasów w skali logarytmicznej

Możemy zaobserwować znaczące ilości wartości odstających poniżej dolnego progu, co wpływa na niesymetryczność rozkładów. Przyczynami mogły być m.in dwie sytuacje:

- specyficznie wylosowane architektury sieci
- organizacja zasobów przez system oparacyjny, który wykonywał również inne zadania

5.5 Podsumowanie

Pomimo drobnych wątpliwości związanych z zebranymi statystykami, możemy uznać że badany algorytm działa szybciej w implementacji w języku C++ z optymalizacją O3, niż w implementacji w języku Java. Największy stosunek szybkości działania pierwszej implementacji do drugiej obserwujemy dla mniejszych wartości liczb wierzchołków, po czym następuje jego asymptotyczna stabilizacjia w okolicach wartości 2.

6 Opis plików

- *networks* – folder do którego zapisywane są wygenerowane losowo sieci, sieci o takiej samej liczbie wierzchołków są nadpisywane przy kolejnych powtórzeniach działania algorytmów
- *report* – folder ze sprawozdaniem

- *results* – folder z wynikami testów oraz badań
  - *stats* – folder ze statystykami wyników
    - ☐ *diffs* – różnice średnich między koszykami z 200, 400, 600, 800 i 1000 pomiarów dla różniej liczby wierzchołków oraz implementacji
    - ☐ *sredni_czas.pdf* – czasy działania algorytmów uśrednione po 1000 prób
    - ☐ *log_sredni_czas.pdf* – czasy działania algorytmów uśrednione po 1000 prób w skali logarytmicznej o podstawie 10
    - ☐ *stosunek.pdf* – stosunek czasów działania algorytmów uśrednionych po 1000 prób
    - ☐ *boxplot.pdf* – wykresy pudełkowe czasów działania algorytmów uśrednionych po 1000 prób
  - *network_cpp_time* – czas działania głównej pętli badanego algorytmu dla 100, 250, 500, 750, 1000 i 1500 wierzchołków, 1000 powtórzeń zaimplementowanej w języku C++
  - *network_java_time* – czas działania głównej pętli badanego algorytmu dla 100, 250, 500, 750, 1000 i 1500 wierzchołków, 1000 powtórzeń zaimplementowanej w języku Java
  - *time_analysis.py* – skrypt pomocniczy w języku Python do analizy danych
- *tests* – testy algorytmu Edmondsa – Karpa dla obu implementacji na dwóch sieciach testowych o wyznaczonych wcześniej parametrach
- *edmonds_karp_algorithm.cpp* – implementacja algorytmu Edmondsa – Karpa w języku C++
- *EdmondsKarpAlgorithm.java* – implementacja algorytmu Edmondsa – Karpa w języku Java
- *flow_network_generation.cpp* – algorytm do generowania sieci zaimplementowany w języku C++
- *Makefile* – plik Makefile
- *run_scripts.sh* – główna pętla testowa w języku powłoki skryptowej Bash

7 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`