Optymalizacja wydajności wybranych algorytmów grafowych.

Łukasz Marcinkowski Jacek Sosnowski

13 maja 2015

1 Testowanie

1.1 Wprowadzenie

Przedmiotem projektu był algorytm odnajdowania cyklu Eulera. Jako cel została przyjęta jego implementacja w językach C++ i Java. Podwójna implementacja miała umożliwić eksperyment polegający na dokładnym zbadaniu wydajności obu podejść. Ostatecznie w niniejszym sprawozdaniu zostanie uzasadniona odpowiedź na pytania:

> *Czy możliwa jest optymalizacja algorytmu poprzez zmianę języka (środowiska) jego implementacji? A jeśli tak, to jak bardzo znaczącą poprawę można uzyskać?*

1.2 Warunki eksperymentów

Wstępnie założono, że żadna z implementacji nie zostanie poddana optymalizacji kodu. Cenniejsze jest uzyskanie czytelnego rozwiązania, aniżeli skomplikowanego i niezrozumiałego kodu. Jest to sytuacja zbliżona do rzeczywistości – gdzie zazwyczaj nie ma miejsca na wyścig optymalizacji.

Schemat działania jest wspólny dla obu podejść. Po pierwsze należy wczytać dane wejściowe (graf) z pliku oraz zbudować odpowiednią strukturę danych w pamięci programu. Po drugie należy uruchomić implementację algorytmu Fleury'ego (opisanego dokładnie w poprzedniej dokumentacji).

1.3 Środowisko testowe

Wszystkie testy były uruchamiane w systemie Ubuntu w wersji 14.04. Pomiary wydajności zostały wykonane z wykorzystaniem środowiska języka R

(za pomocą narzędzia R-Studio). Znaczenie ma również konfiguracja sprzętowa, która w tym przypadku była oparta na dwurdzeniowym procesorze taktowanym zegarem 2,2 GHz. Kod źródłowy został skompilowany przez kompilator GCC 4.8 w przypadku języka C++. Natomiast aplikacja stworzona w Javie podczas testów była uruchamiana z wykorzystaniem platformy JRE w wersji 1.8.

Pomiar czasu był wykonywany z dokładnością do jednej mikrosekundy. Lecz dla zachowania czytelności niniejszej dokumentacji oraz z uwagi na charakter prowadzonych testów pomiar (czas trwania dłuższy niż sekunda), wyniki są zapisywane z mniejszą precyzją.

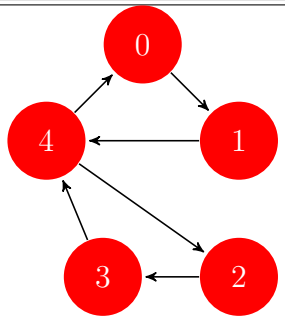
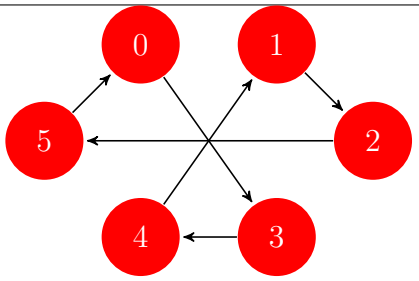
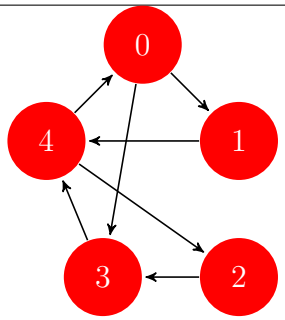
1.4 Testy akceptacyjne

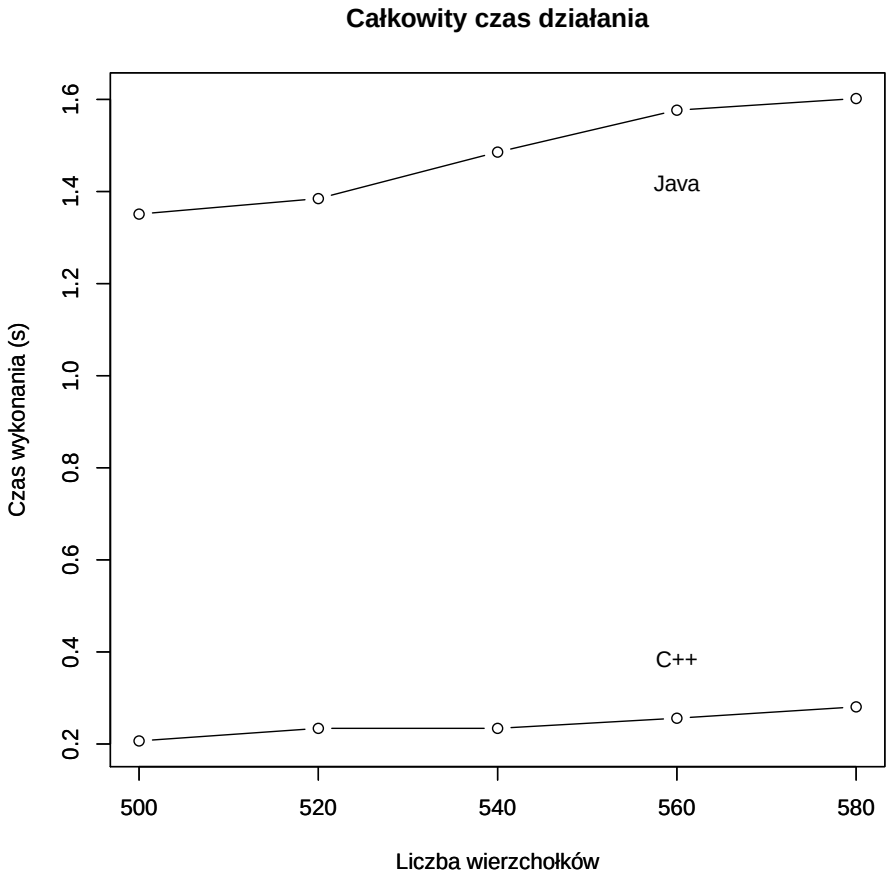
Zanim oba rozwiązania zostały poddane ocenie wydajności, musiały przejść zestaw testów akceptacyjnych. Mowa o poszukiwaniu cykli Eulera w prostych grafach wejściowych. Kilka przykładów przedstawia tabela 1. Więcej grafów znajduje się w katalogach z testami projektu oraz w poprzedniej dokumentacji.

Dla wyjaśnienia, należy zaznaczyć, że ostatni przykład z tabeli 1 jest tym samym grafem co pierwszy, lecz z dodaną dodatkową krawędzią $(0, 3)$. Taka modyfikacja, sprawiła, że nie można znaleźć cyklu Eulera. Istotnie, idąc tą samą drogą, co dla pierwszego grafu, odwiedzone zostaną wszystkie krawędzie oprócz $(0, 3)$.

Wynikowy cykl Eulera może być zwracany jako tekst przekazany na standardowe wyjście znakowe systemu operacyjnego lub zapisany do pliku.

Tablica 1: Przykłady grafów stanowiących część testów akceptacyjnych.

Plik wejściowy	Wygląd grafu	Wynik algorytmu
directed 5 0 : 1 1 : 4 2 : 3 3 : 4 4 : 0 2		0, 1, 4, 2, 3, 4, 0
directed 6 0 : 3 1 : 2 2 : 5 3 : 4 4 : 1 5 : 0		3, 4, 1, 2, 5, 0, 3
directed 5 **0** : 1 **3** 1 : 4 2 : 3 3 : 4 4 : 0 2		Euler Path: (not found)



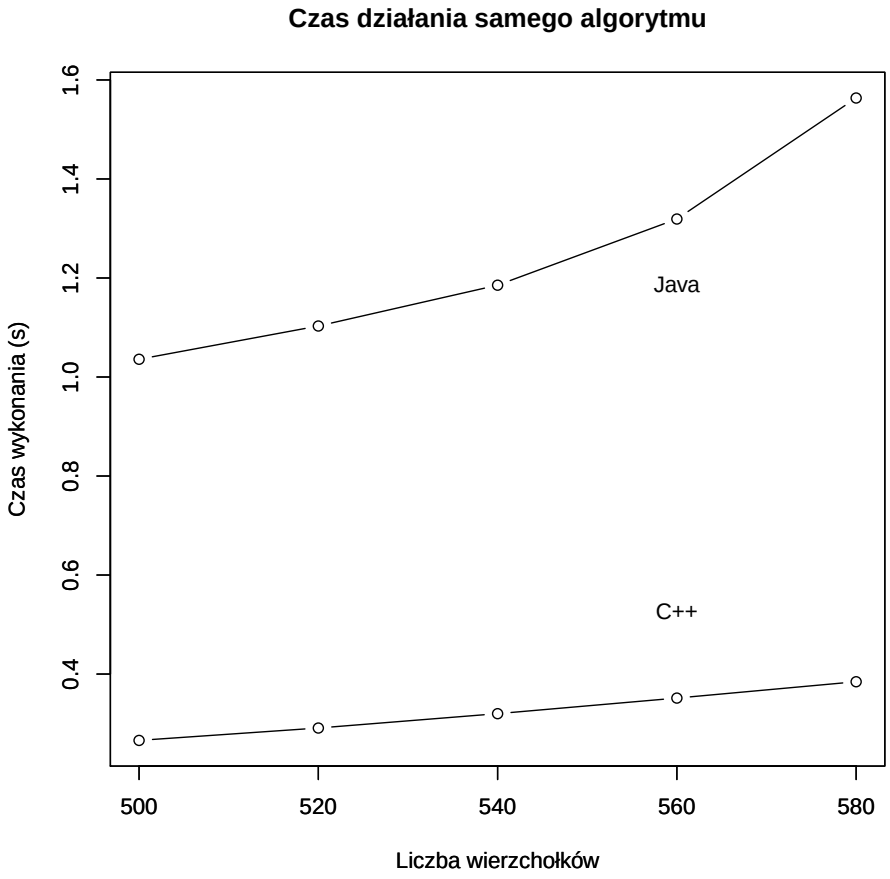
Rysunek 1: Całkowity czas wykonywania zadania.

2 Porównanie wydajności

Pierwszy eksperyment reprezentuje wykres 1. Nie jest to zbyt sprawiedliwe porównanie, ponieważ pokazuje czas całkowitego uruchomienia (od startu do zwrócenia wyniku). Z tego powodu, w przypadku implementacji Javy jest również uwzględniany narzut czasowy startu interpretera – maszyny wirtualnej Javy.

Z tego powodu powstaje pytanie: Czy takie porównanie ma sens? Odpowiedź brzmi: tak, ale wyłącznie z punktu widzenia użytkownika. Dla niego nie jest istotne w jakiej technologii jest stworzona aplikacja, lecz jak działa. W tym przypadku widać, że aplikacja Javy zmusza do kilkukrotnie dłuższego oczekiwania na wynik.

Dla usunięcia obciążenia interpretera Javy czas został zmierzony również



Rysunek 2: Czas działania algorytmu.

wewnątrz aplikacji. Zastosowano rozdzielczość jednej mikrosekundy. Wynik przedstawia wykres 2. Jednak jak widać, narzut ten był spory wyłącznie, dla mniejszych danych wejściowych. Dalej odgrywał już mniejszą rolę.

2.1 Schemat eksperymentu

Oba eksperymenty przedstawione na wykresach 1 i 2 są wykonane według podobnego schematu:

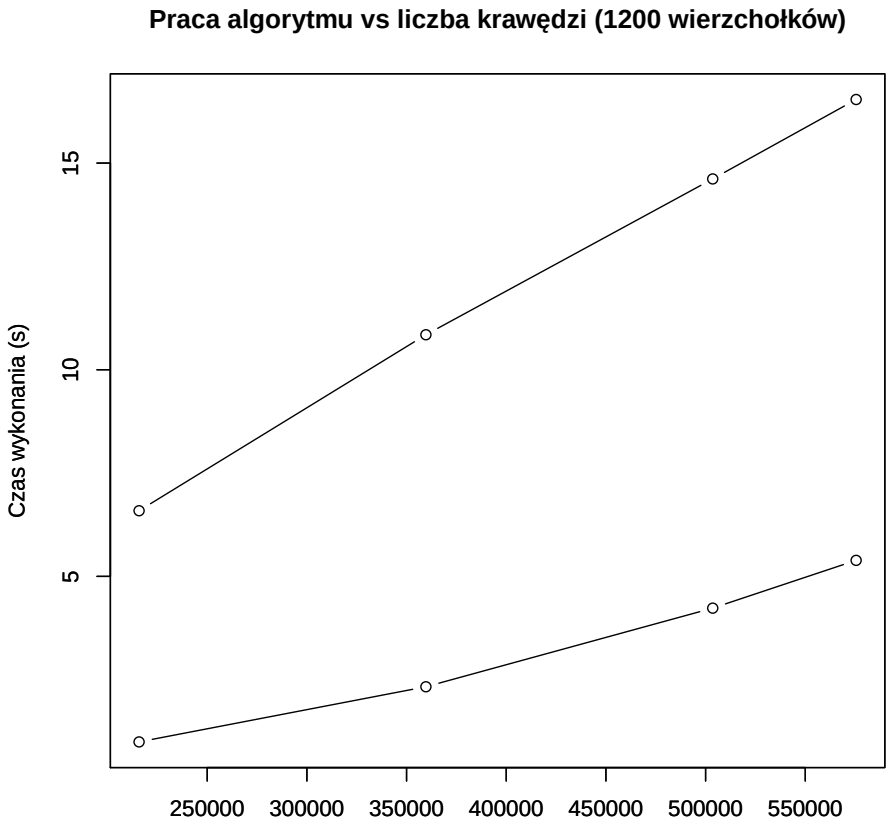
1. Generacja (losowanie) grafów o wymaganych własnościach (np. liczbie wierzchołków, krawędzi, itp.)
2. Dla każdego grafu:
   (a) Kilkukrotne uruchomienie każdej implementacji – pomiar czasu każdego uruchomienia.
   (b) Średnia z powyższych pomiarów jest traktowana jako czas pracy dla określonych danych wejściowych (grafu).
3. Tworzenie wykresu.

2.2 Złożoność

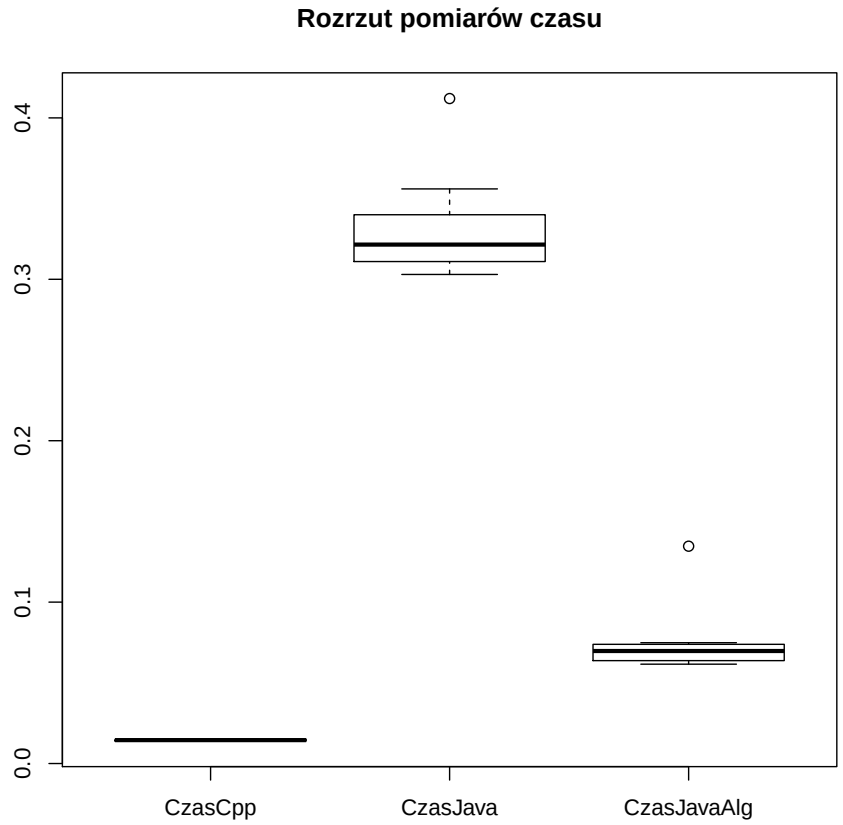
W paragrafie 1.4 zostało pokazane, iż aplikacje działają poprawnie. Jednak należy również zweryfikować złożoność czasową obu rozwiązań.

Jak wskazano w poprzedniej dokumentacji, algorytm Fleury'ego ma złożoność liniową względem liczby krawędzi w grafie. W celu weryfikacji tego stwierdzenia, wygenerowano zbiór grafów, każdy o 1200 wierzchołkach ale różnych gęstościach. Przeliczając gęstość na liczbę krawędzi otrzymano zbiór grafów o licznościach krawędzi pomiędzy 100 tys., a 600 tys. Wykres 3 pokazuje opisywaną tutaj zależność.

Jednym z podstawowych wniosków jest fakt, że obie implementacje bezsprzecznie spełniają założenia algorytmu i posiadają złożoność liniową względem liczby krawędzi (O(q), gdzie q to liczba krawędzi). Ale dodatkowo można zwrócić uwagę na nachylenia obu krzywych. Okazuje się, że zapotrzebowanie na czas rośnie szybciej w przypadku implementacji w Javie. Nie jest to wzrost zatrważający, ale może być znaczącym argumentem do zastosowania optymalizacji w postaci zmiany języka z Javy na C++. Szczególnie wtedy, kiedy wiadomo, że często rozpatrywane będą grafy zbliżone do pełnych.



Rysunek 3: Praca algorytmu w zależności do liczby krawędzi. Górny wykres – Java, dolny – C++.



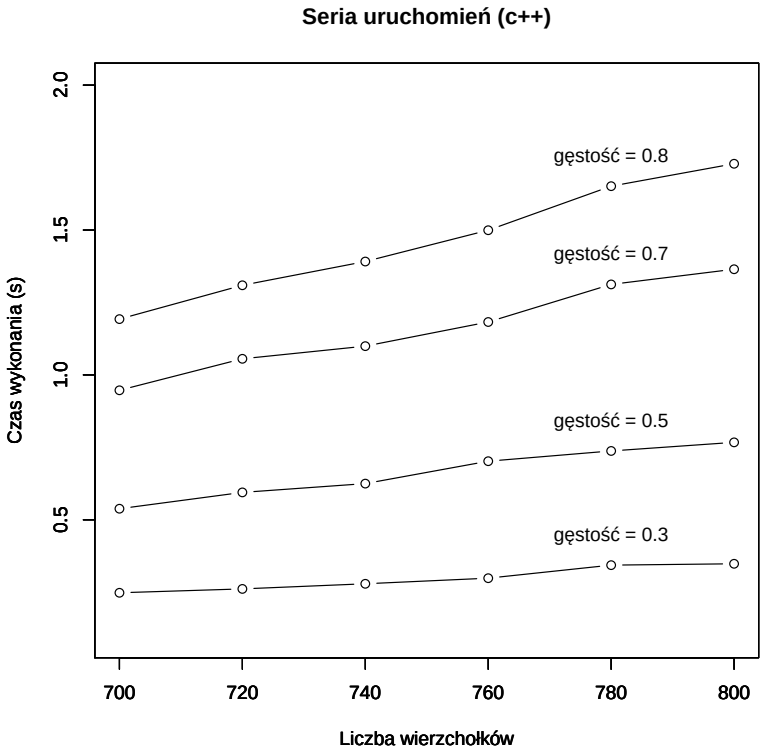
Rysunek 4: Wykresy pudełkowe: czasu wykonania dla C++, pełnego Javy oraz samego algorytmu w Javie.

2.3 Rozrzut wyników

Realizując serię pomiarów dla pojedynczej instancji problemu (w tym przypadku – pojedynczego grafu), należy zauważyć, że każdy pomiar jest realizacją pewnej zmiennej losowej. Przy porównywaniu obu aplikacji interesujące jest również to jak stabilne są prezentowane wyniki.

Do interpretacji cech statystycznych wykonywanych pomiarów zastosowano wykres pudełkowy (ang. *Box plot*) przedstawiony na Wykresie 4.

Największy rozrzut wyników można zaobserwować w przypadku badania pełnego uruchomienia aplikacji w Javie, co może być istotne dla użytkownika końcowego. Z drugiej strony jest to fakt w pełni uzasadniony działaniem maszyny wirtualnej Javy i systemu operacyjnego.



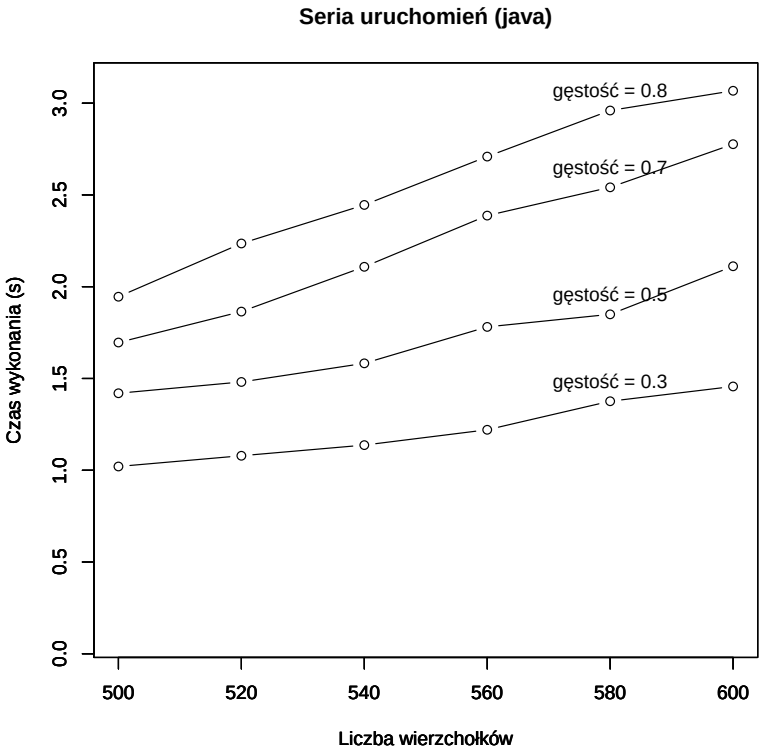
Rysunek 5: Seria uruchomień aplikacji napisanej w C++.

Dlatego do porównania lepiej zastosować pomiar pracy samego algorytmu, który w tym przypadku pokazuje stabilniejsze działanie. Jednak również w tej kategorii niepokonana została implementacja z wykorzystaniem C++. Dla której wahania były tak niewielkie, że z racji skali wykresu 4, są one niewidoczne.

2.4 Serie uruchomień

Na koniec warto przyjrzeć się jeszcze zmienności wyników w perspektywie serii uruchomień każdej z implementacji – rysunki 5 oraz 6.

Pokazują one cechę odporności obu implementacji na zmienną gęstość tych samych grafów. Wyniki są zgodne z oczekiwaniami – uzyskano zależność liniową – ale nie jest to cecha oczywista i automatyczna. Gdyby w którymś rozwiązaniu zastosowano niewłaściwą strukturę danych (a co za tym idzie nieefektywną), można by zaobserwować znaczące fluktuacje na poszczególnych krzywych.



Rysunek 6: Seria uruchomień aplikacji napisanej w Javie.

3 Podsumowanie

W ramach tego projektu algorytm Fleury'ego został poprawnie zaimplementowany w dwóch najbardziej popularnych językach obecnego świata IT. Przeprowadzono szereg różnorodnych wyszukiwań cykli Eulera na losowych grafach o różnych liczbach wierzchołków i krawędzi. Testowano grafy od najmniejszego trój-wierzchołkowego, po takie o liczbie wierzchołków rzędu tysięcy. Ponadto dokonano wielu badań, których wyniki zostały zaprezentowane w niniejszym sprawozdaniu.

Na podstawie zdobytych doświadczeń, a także w ramach podsumowania można wysnuć wniosek o istnieniu **optymalizacji** w kontekście zmiany języka implementacji dla algorytmów grafowych. Innymi słowy, pokazano tutaj, że ten sam algorytm osiąga znacząco różne rezultaty w różnych językach. Nie zakłada się, że cechą obu języków jest fakt, że jeden jest efektywniejszy od drugiego. Ale posiadając kod Javy implementujący pewien algorytm grafowy (np. poszukujący cykli Eulera) jest wielce prawdopodobne, że jego „przepisanie" w języku C++ przyniesie korzyść w postaci zmniejszonej konsumpcji czasu. Jest to w rzeczywistości pewien rodzaj optymalizacji, który można osiągnąć stosunkowo niskim kosztem oraz bez ryzyka straty czytelności samego kodu źródłowego.

Literatura

[1] J. Wojciechowski, K. Pieńkosz, *Grafy i sieci*, Wydawnictwo Naukowe PWN, 2013

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowo-Techniczne, Warszawa 2001.