Jakub Tokarzewski 283778
sem. 5 18Z Informatyka ISI wydział EiTI

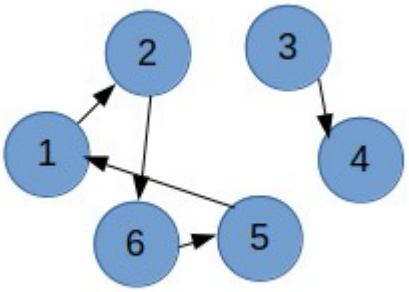
AAL

Zadanie 2.

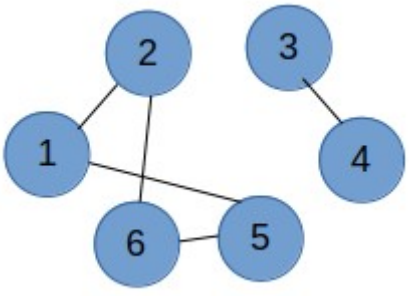
Król próbuje zebrać informacje o bieżącej sytuacji na rozlicznych frontach walk, które prowadzi jego królestwo. Wie, że jego zwiadowcy rozmawiali ze sobą i każdy z nich mógł dla bezpieczeństwa przekazać informacje o realizacji swojego zadania maksymalnie jednemu innemu zwiadowcy. Każdy zwiadowca, który przekazał informacje, przekazał także to co usłyszał od innych zwiadowców. Zwiadowcy wysłali gołębie pocztowe z informacją dla króla o tym kto i komu przekazał informacje. Król chciałby wezwać jak najmniejszą liczbę zwiadowców tak, aby nie odciągać ich od pracy ale jednocześnie pozyskać maksymalną wiedzę.
Zaproponuj algorytm, który dla zadanego wejścia (które będzie zawierało informacje o tym, który zwiadowca raportował do którego zwiadowcy) znajdzie minimalną liczbę zwiadowców, których król musi wezwać, aby uzyskać wszystkie informacje. Oceń jego złożoność czasową oraz pamięciową.

Analiza problemu

Problem ten polega na znalezieniu ilości spójnych składowych grafu, w którym zbiorem wierzchołków są zwiadowcy, a zbiorem krawędzi jest wymiana informacji między zwiadowcami. Skoro jeden zwiadowca może przekazać tylko jednemu innemu zwiadowcy swoje informacje, tzn. że z jego wierzchołka może wychodzić tylko jedna krawędź. To założenie nie mówi ile krawędzi może dochodzić do danego wierzchołka, więc w przypadku n zwiadowców do jednego zwiadowcy może wchodzić nawet n-1 krawędzi. Założyłem także, że chronologia rozmów między zwiadowcami nie ma znaczenia.
Przykładowa wymiana informacji mogłaby wyglądać w ten sposób.



Po szybkiej analizie grafu widać, że potrzeba zapytać dwóch zwiadowców, aby mieć informacje na temat wszystkich zwiadowców. Liczba zwiadowców zgadza się w tym przypadku z liczbą składowych tego grafu. Można zauważyć, że nieważne jaki będzie przepływ informacji między wierzchołkami to zawsze w składowej znajdzie się jeden taki wierzchołek, który ma informacje na temat wszystkich pozostałych wierzchołków ze składowej, z czego wynika, że aby mieć informacje o wszystkich zwiadowcach z danej składowej grafu wystarczy zapytać jednego konkretnego zwiadowcę. Ten wniosek nasuwa nam rozwiązanie problemu, które sprowadza się do wyliczenia ilości spójnych składowych grafu. Skoro mamy znaleźć tylko ilość spójnych składowych grafu, to możemy założyć, że graf ten jest nieskierowany i może zostać przedstawiony w postaci .



Rozwiązanie problemu

Algorytm rozwiązujący problem polegał będzie na przeszukiwaniu grafu w głąb do momentu odwiedzenia wszystkich wierzchołków. Odwiedzone wierzchołki będą oznaczane za pomocą tablicy *visited*. Liczba uruchomionych przeszukiwań w głąb będzie mówiła o tym ile składowych jest w grafie. Zaimplementuję dwie różne funkcje przeszukiwania w głąb, aby móc porównać ich wydajność. Pierwsza implementacja będzie rekurencyjna a druga iteracyjna.

Pseudokod

```
int function(Graph) {

        boolean visited[Graph.size]; // tablica odwiedzonych wierzchołków, o wielkości
// równej liczbie wierzchołków w grafie. Początkowe wszystkie wartości ustawione na false.
        Integer scouts = 0; // liczba zwiadowców (ang. scouts), czyli liczba spójnych
                            // składowych w naszym grafie.
        for ( int i = 0 ; i < Graph.size ; ++i ) {
                if ( visited[i] == false ) { // jeżeli wierzchołka i-tego nie odwiedziliśmy jeszcze
                                             //to zaczynamy od niego nowe przeszukiwanie w głąb.
                        ++scouts;            // zwiększamy ilość składowych
                        DFS(Graph, visited, i); // funkcja przeszukiwania w głąb
                }
        }

        return scouts; // ta zwrócona liczba jest poszukiwanym wynikiem, czyli liczbą
//składowych tego grafu, co za tym idzie liczbą zwiadowców, jakich król musi zapytać, aby
//mieć wszystkie informacje na temat toczących się bitew.
}

//////////////////////////////REKURSYWNY WARIANT//////////////////////////////////////////////////
void DFS(Graph, visited, current_node) {
        visited[current_node] = true; // odznacza aktualny wierzchołek jako odwiedzony.
        For( int i = 0 ; i < Graph.size ; ++i ) {
                if ( visited[i] == false && nodes_are_connected(current_node, i )){
                        DFS(Graph, visited, i); // jeśli danego wierzchołka jeszcze nie
//odwiedziliśmy i sąsiaduje on z wierzchołkiem, który wywołał przeszukiwanie w głąb. To
//przeszukujemy dalej w głąb od nowego wierzchołka, czyli idziemyr rekurencyjnie po
//wszystkich krawędziach danej składowej grafu.
                }
        }
}
```

////////////////////////////////////**ITERACYJNY WARIANT**//

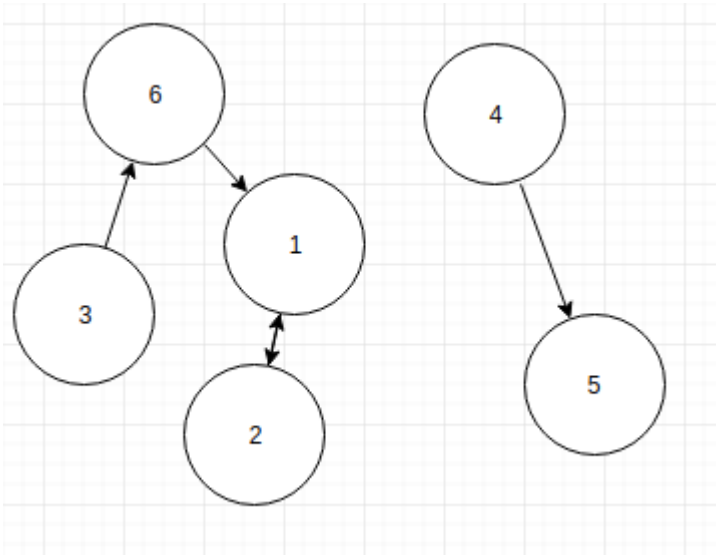
```
void DFS(Graph, visited, starting_node){
        Stos stos = new Stos(); // na stosie będą składowane wierzchołki do odwiedzenia
        stos.push(starting_node);
        visited[starting_node] = true; // oznaczamy startowy wierzchołek jako odwiedzony

        //pętla kończy się gdy stos będzie pusty
        while( stos.pusty() == false ) {
                node = stos.pop(); //zdejmij ze stosu wierzchołek do odwiedzenia

                for( neighbor : neighbors of node ) { //przeglądamy wszystkich sąsiadów
                                                      //wierzchołka
                        if( visited[neighbor – 1 ] == false) {
                                visited[neighbor – 1] = true;
                                stos.push(neighbor);        //wrzucamy sąsiadów na stos, aby
        //móc potem odwiedzić sąsiadów tego wierzchołka, czyli przeszukiwać dalej w głąb.
                        }
                }
        }
}
```

Czas wykonania obu algorytmów porównam dla różnych wielkości grafów i przedstawię za pomocą wykresów.

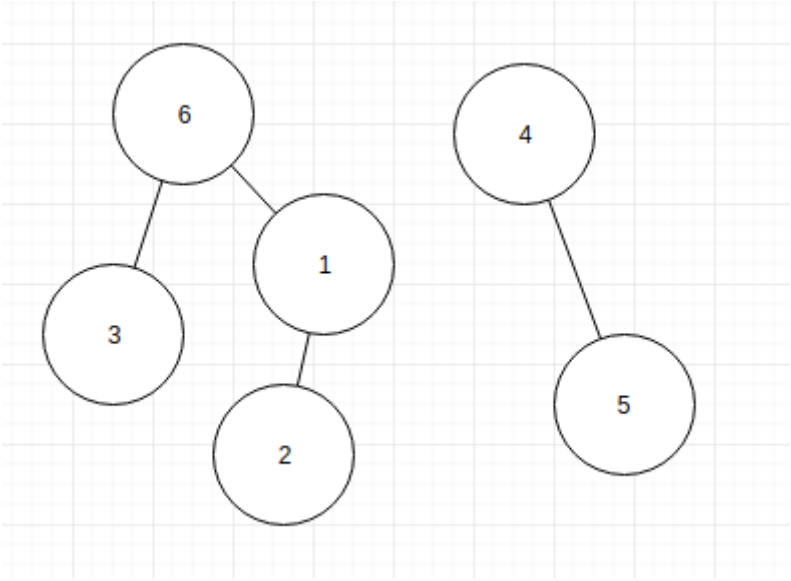
Do wizualnej reprezentacji grafu użyję języka **Python** i biblioteki *graphviz*.

Przykładowe działanie programu

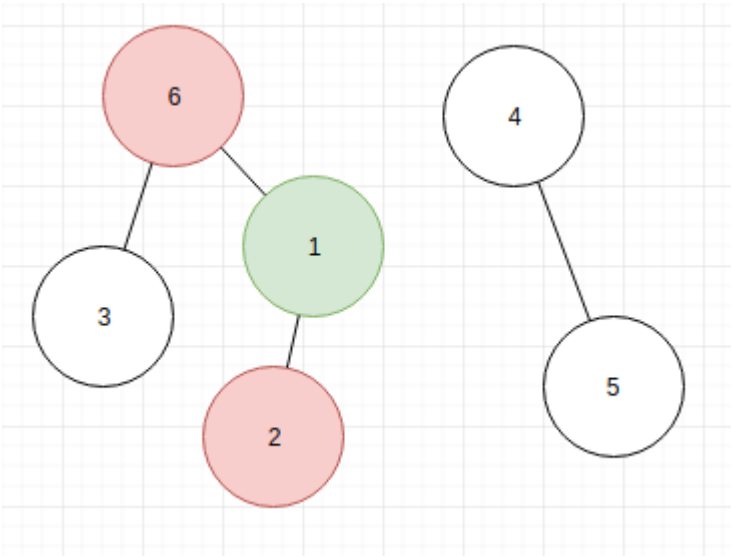
Nasz graf początkowy wygląda w ten sposób. Na wejściu jest on grafem skierowanym, ale będzie przechowywany w pamięci jako graf nieskierowany, gdyż chcemy policzyć tylko składowe tego grafu.



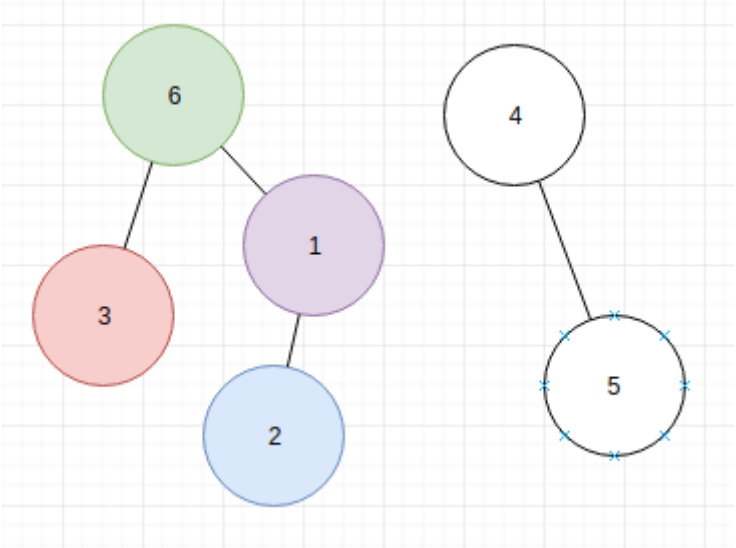
Graf startowy w postaci nieskierowanej będzie miał postać przedstawioną poniżej.

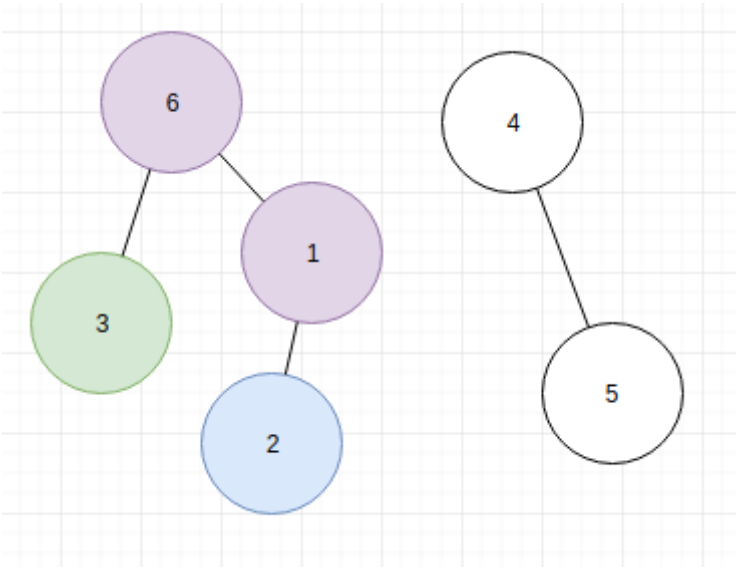


Nasz algorytm zaczynamy od wierzchołka oznaczonego nr 1.

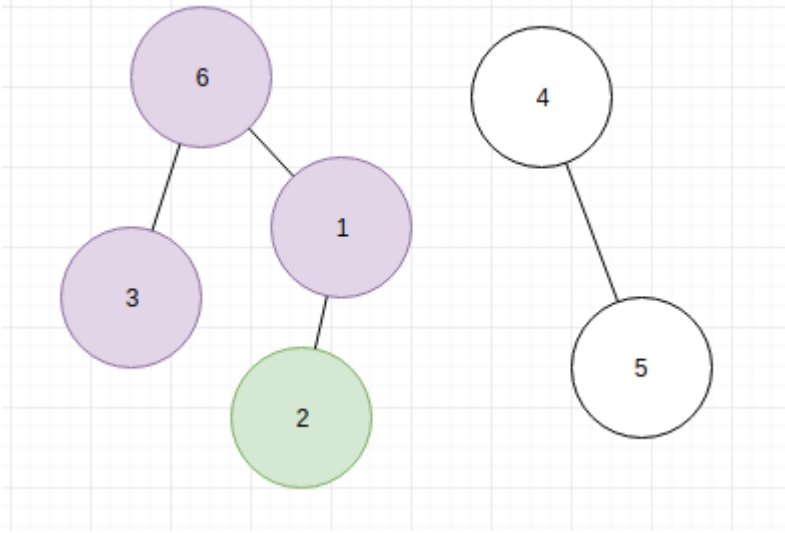


Zielonym kolorem oznaczony wierzchołek, w którym aktualnie jesteśmy (ten wierzchołek znajduje się na stosie). Czerwonym kolorem oznaczeni są sąsiedzi aktualnego wierzchołka, którzy trafią na górę stosu. Fioletowym kolorem będą oznaczone już odwiedzone wierzchołki i są już zdjęte ze stosu. Niebieskim kolorem będą oznaczone wierzchołki czekające na stosie.

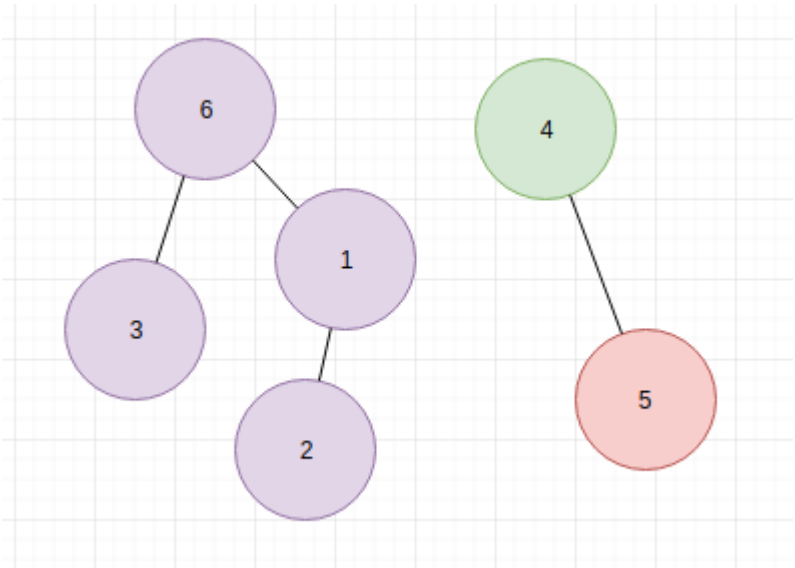


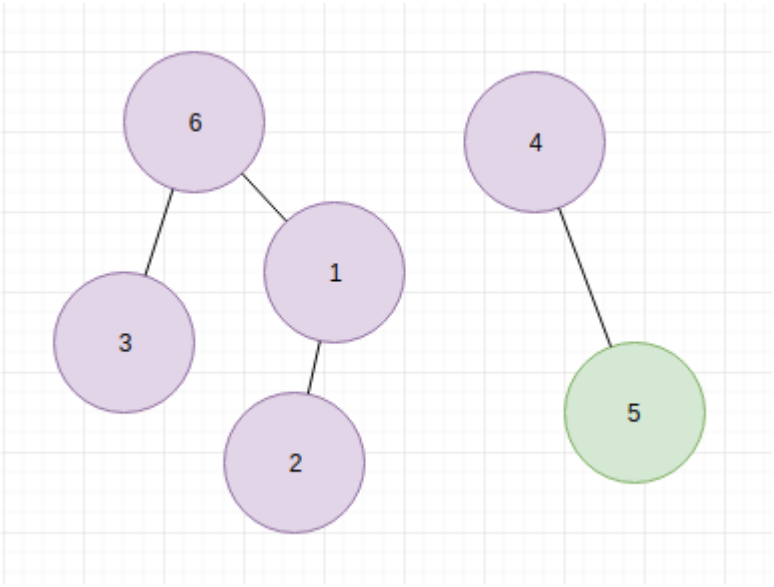


W kroku powyżej widać, że wierzchołka o numerze 3 nie mamy już jak rozwijać, więc oznaczamy go jako odwiedzony i zdejmujemy ze stosu. W ten sposób ostatnim wierzchołkiem, który pozostał na stosie jest wierzchołek oznaczony numerem 2.

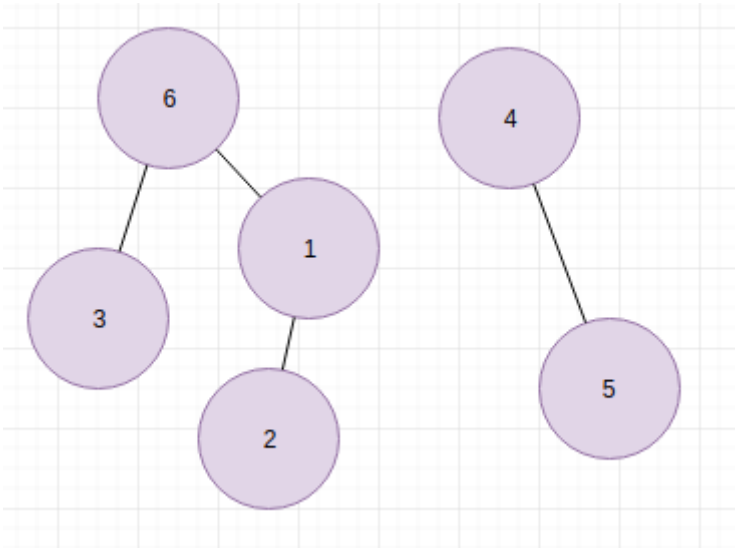


Na tym etapie nie da się przejść do sąsiadów wierzchołka drugiego, bo wierzchołek nr 1 jest odwiedzony, więc zdejmujemy wierzchołek nr 2 ze stosu. Jak widać na obrazku ten wierzchołek był ostatnim elemencie na stosie, czyli po usunięciu go nasz stos jest pusty. Inkrementujemy więc liczbę spójnych składowych i przeszukujemy listę wierzchołków w poszukiwaniu nieodwiedzonych wierzchołków. Pierwszym znalezionym, nieodwiedzonym wierzchołkiem, więc od niego zaczynamy kolejne wywołanie przeszukiwania w głąb.





W tej sytuacji nie ma już jak rozwinąć wierzchołka o numerze 5, więc stos zostaje pusty oraz inkrementujemy liczbę spójnych składowych. Przeszukiwanie listy wierzchołków wykaże, że nie ma już żadnych nieodwiedzonych wierzchołków w grafie i nasz algorytm skończył pracę i jako wynik zwróci zliczoną liczbę spójnych składowych.



Złożoność obliczeniowa

Złożoność obliczeniowa całego algorytmu sprowadza się do przeanalizowania złożoności algorytmu przeszukiwania w głąb (dalej zwany DFS). Optymistyczna złożoność algorytmu DFS to **O(V)**, gdzie V to liczba wierzchołków grafu. Jest to przypadek, gdy każdy wierzchołek jest izolowany, więc jedyne co trzeba zrobić to przejść po każdym wierzchołku, dowiedzieć się że żaden nie ma sąsiadów i zakończyć algorytm.

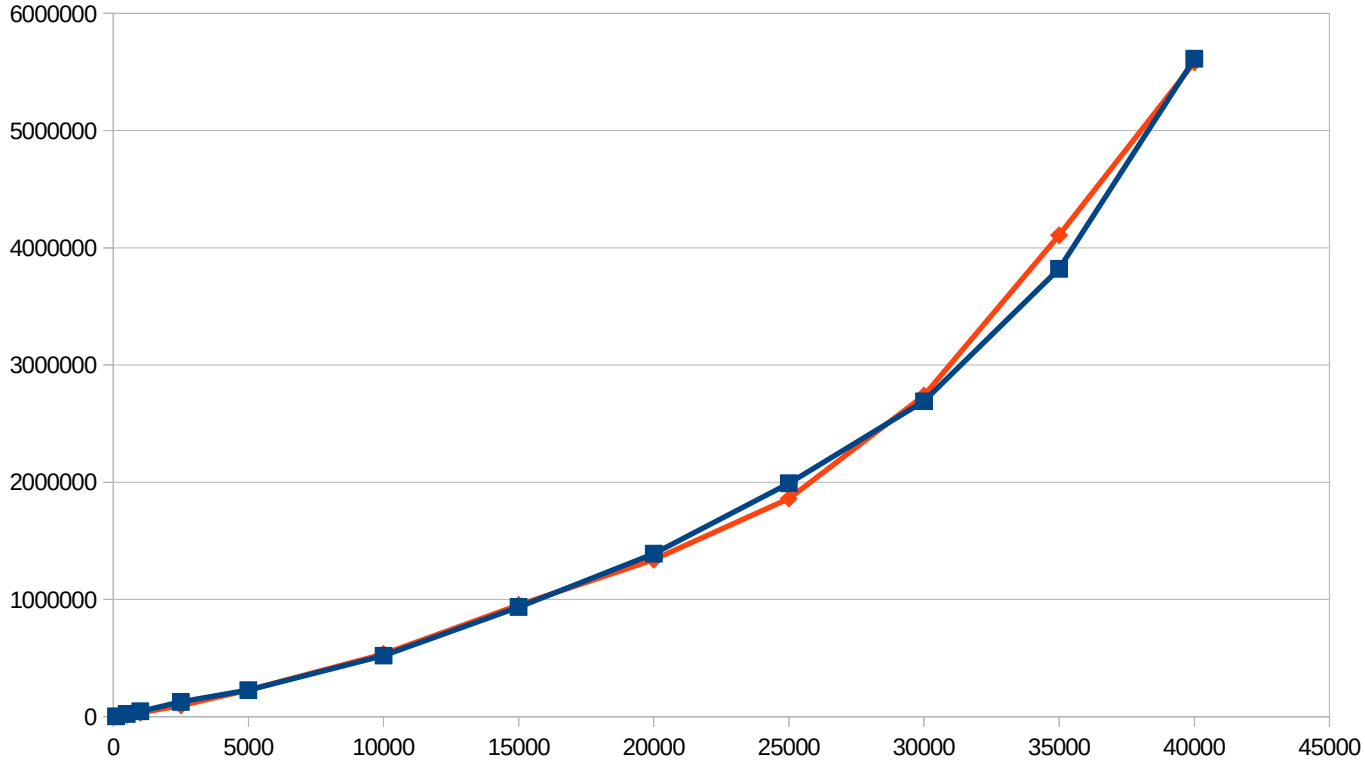
Złożoność algorytmu DFS zależy od sposobu zaimplementowania grafu. Jeżeli graf jest zaimplementowany w postaci macierzy sąsiedztwa (tablica V x V), wtedy dla każdego wierzchołka musimy przejść całą tablicę o długości V, aby poznać jego sąsiadów. Złożoność takiego rozwiązania wynosi zatem **O(V*V) = O(V^2)**.

Innym podejściem jest przedstawienie grafu jako listy sąsiedztwa, czyli tablicy o wielkości V, w której każdy element jest listą swoich sąsiadów. W takim przypadku,dla każdego wierzchołka można odkryć wszystkich sąsiadów przechodząc przez listę jego sąsiadów w czasie liniowym. Dla grafu nieskierowanego każda krawędź pojawia się dwukrotnie, więc złożoność obliczeniowa to **O(V) + O(2E) ~= O(V + E).** Po przeprowadzonej analizie można dojść do wniosku, że bardziej optymalnie jest przedstawiać graf jako lista sąsiedztwa. Sprawdzanie, czy wierzchołek został już odwiedzony to operacja liniowa, więc nie wliczam jej w analizie złożoności.

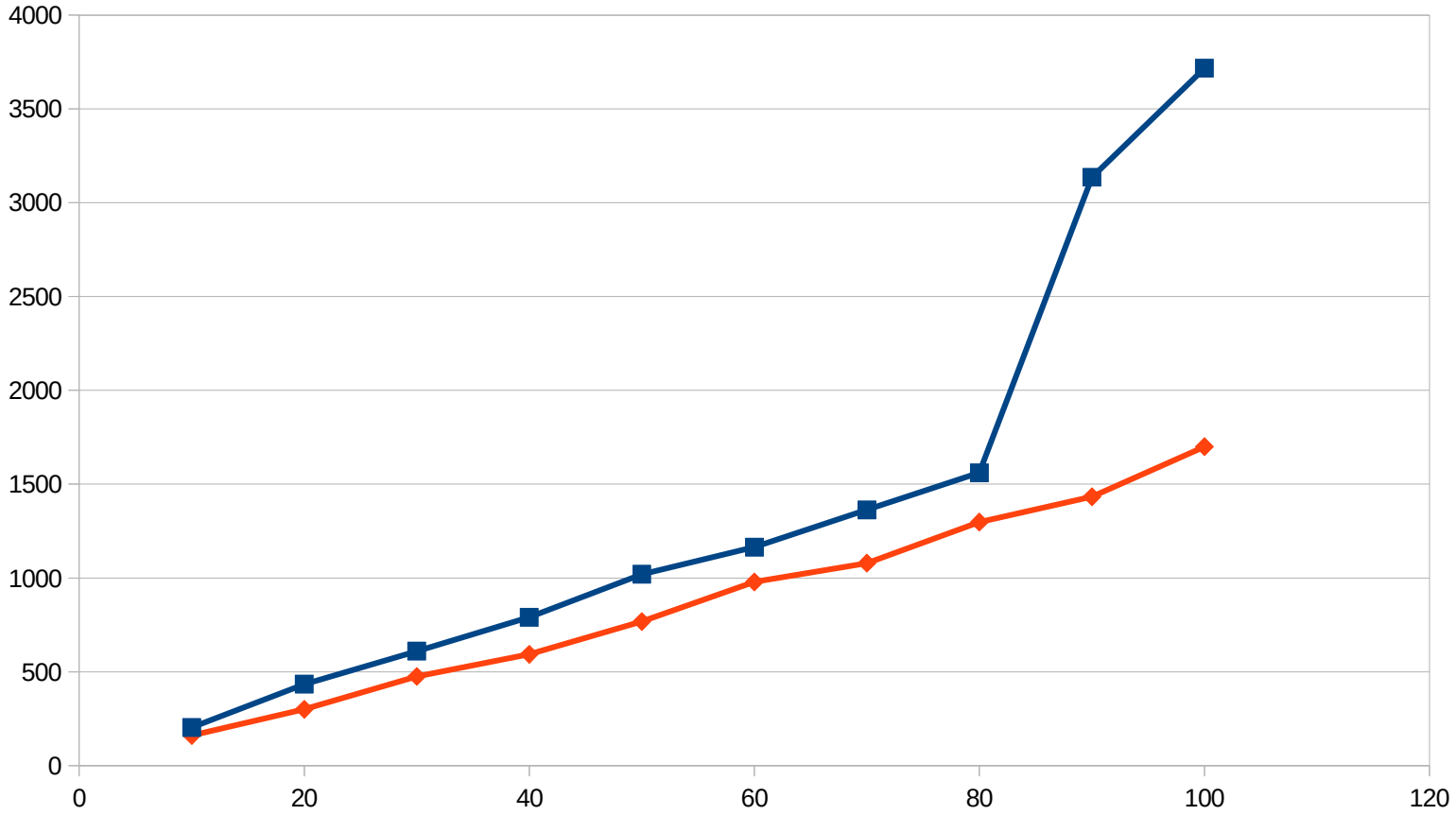
Poniżej załączam wykres zależności liczby wierzchołków od czasu wykonywania algorytmu przeze mnie zaimplementowanego.

Rzeczywiste pomiary czasu przy użyciu biblioteki JMH

Poniżej przedstawiam dwa wykresy obrazujące czas działania programu w zależności od ilości wierzchołków. Niebieska linia przedstawia rozwiązanie iteracyjne, a pomarańczowa rekurencyjne.



Jak widać na wykresie rozbieżności dla małej ilości wierzchołków są nieduże, jednak nieco lepiej sobie radzi algorytm rekurencyjny, do momentu 30000 wierzchołków gdzie zaczyna zwalniać i zostaje wyprzedzany przez algorytm iteracyjny. Nie są to jednak charakterystyki liniowe, której się spodziewałem podczas teoretycznej analizy problemu. Nieliniowość praktycznych wyników może wynikać z losowego czynnika podczas generowania grafów, co sprawia, że jedne są szybciej, a inne dłużej rozwiązywane przez program.



Tutaj sprawa wygląda nieco lepiej. Algorytm rekurencyjny dla małej liczby wierzchołków utrzymuje złożoność liniową, natomiast algorytm iteracyjny dla 90 wierzchołków głatownie traci liniową charakterystykę.

Problemy implementacyjne i wnioski

Zdecydowałem się na wybór języka Java do implementacji algorytmu, ponieważ ten sam kod napisany w tym języku działa zarówno w systemie Windows, Linux i MacOS. Podczas implementacji algorytmu nie było żadnych problemów związanych z językiem. Pierwsza przeszkoda pojawiła się podczas wykonywania testów czasowych programu. W przypadku robienia benchmarków w języku Java trzeba sprostać problemowi rozgrzewania maszyny wirtualnej. W tym celu użyłem biblioteki „jmh”, w której możemy specyfikować jak długo ma być wykonywane rozgrzewanie maszyny wirtualnej przed wykonaniem testów benchmarkowych, które ta biblioteka również udostępnia. Drugi problem w implementacji pojawił się podczas prób wygenerowania w postaci graficznej grafu. Biblioteki do języka Java okazały się mało intuicyjne i nie spełniały moich oczekiwań (próbowane biblioteki : JUNG, JGraph), więc skorzystałem z kolejnego cross-platformowego języka, którym jest Python oraz biblioteki „graphviz”, która bez problemu generuje duże grafy w postaci graficznej i zapisuje je czytelnie w pliku „pdf”.
Aby ułatwić innym pracę z projektem wszystkie zależności można łatwo zainstalować, dzięki narzędziu Maven.