Algorytmy i struktury danych. Sprawozdanie - algorytmy grafowe

Mariusz Hybiak, Informatyka, nr albumu 148117

Reprezentacje grafów

Celem laboratoriów była implementacja wybranych reprezentacji grafów. Przygotowanymi przez mnie są:

- lista krawędzi
- lista następników
- macierz sąsiedztwa
- lista poprzedników

W ramach implementacji należało utrzymać ustandaryzowany interfejs, który składał się z takich funkcji jak:

- dodanie łuku do grafu
- sprawdzenie czy wierzchołki *u*, *v* są połączone łukiem
- odczyt rzędu grafu
- iterowanne po wszystkich następnikach określonego węzła

Dodatkowo umieściłem funkcję usuwania wierzchołka, aby działanie jednym z algorytmów sortowania topologicznego opierało się na reprezentacjji grafu.

Lista krawędzi

Swoją listę krawędzi ostatecznie oparłem w wyniku eksperymentu z różnymi kontenerami danych na `std::set<edge>`. Początkowo wykorzystywałem `std::unordered_set`, ale w przypadku uporządkowanych rosnąco krawędzi wyszukiwanie następników danego wierzchołka odbywa się szybciej ($O(n)$), ponieważ jest przerywane kiedy zostanie przekroczony dany wierzchołek. W najgorszym przypadku, kiedy wierzchołek będzie miał etykietę z "końca grafu" algorytm będzie musiał przeszukać całą listę, podobnie w przypadku kiedy wszystkie krawędzie będą związane z jednym wierzchołkiem. Do opisu krawędzi wykorzystuję strukturę `struct edge`, która zawiera liczby `int From` oraz `int To`. Dodawanie, wyszukiwanie i usuwanie z mojej reprezentacji ma złożoność logarytmiczną $O(\log m)$. Zwracanie rzędu grafu odbywa się w czasie $O(n)$.

Lista następników

Moja lista następników jest w postaci `std::unordered_map<int, std::set<int>>`. Odwoływanie się do danego klucza to ma czas mniej więcej stały, a nowi następnicy są dodawani w czasie logarytmicznym. Więcej czasu zajmuje operacja usuwania danego wierzchołka, ponieważ trzeba przeiterować po wszystkich kluczach i z ich następników usunąć wskazaną wartość. Jest to więc złożoność $O(n \log n)$. W tym przypadku zwrócenie następników danego wierzchołka nie wymaga żadnych obliczeń. W przypadku pesymistycznym w tym grafie każdy wierzchołek ma jednego lub więcej następników, więc na przykład w czasie usuwania trzeba będzie przejrzeć wszystkie sety

Macierz sądziedztwa

Macierz sąsiedztwa zrealizowałem w postaci dwumiarowego wektora `std::vector<std::vector<int>>`, który jest powiększany za każdym razem, kiedy nowo dodawany wierzchołek jest większy od jego rozmiaru. Operacja ta odbywa sie w czasie $O(1)$. Tyle samo zajmuje dodawanie i usuwanie nowych wierzchołków, ponieważ zmieniają się jedynie cyfry na danej pozycji z 0 na 1 i odwrotnie. Tak samo w czasie $O(1)$ zwracany jest rząd grafu. Zwracanie listy następników ma złożoność $O(n \log n)$, ponieważ algorytm przegląda liniowo cały wektor, a następnie dodaje do setu wpółrzędne, gdzie znalazł jedynkę. W mojej macierzy sąsiedztwa nie zaznaczam łuków w obie strony, to znaczy nie używam nigdzie wartości -1, ponieważ nie jest to w żadnym z algorytmów potrzebne, a taka macierz byłaby zbędnie symetryczna. Ta reprezentacja będzie działać tak samo niezależnie od tego jakie połączenia znajdą się w grafie.

Lista poprzedników

Lista poprzedników jest zrealizowana na identycznej zasadzie jak lista następników. Inną złożoność ma jedynie zwracanie następników danego wierzchołka - $O(n \log^2 n)$ - dla każdej listy poprzedników algorytm sprawdza czy dany wierzchołek się w niej znajduje, jeśli tak, klucz dodawany jest do setu. W przypadku pesymistycznym w tym grafie każdy wierzchołek ma jednego lub więcej poprzedników, więc na przykład w czasie usuwania trzeba będzie przejrzeć wszystkie sety

Sortowanie topologiczne

Bazując na opisanym wyżej interfejsie należało zaimplementować dwie alternatywne metody sortowania topologicznego grafu: algorytm Kahna oraz algorytm oparty o metodę DFS. Zrealizowałem je w języku C++ i prezentują się one następująco:

Algorytm Kahna

```cpp
template<class GraphType>
int degree(int V, GraphType &G) {

    std::vector<int> temp(G.order());
    std::iota(temp.begin(), temp.end(), 1);

    std::unordered_set<int> verticles(temp.begin(), temp.end());

    for (auto &vertex : verticles) {
        if (G.check_exist(vertex, V)) {
            return 1;
        }
    }
    return 0;
}

template<class GraphType>
std::deque<int> Kahn_sorting(GraphType G) {
    std::deque<int> sorted;
    std::unordered_set<int> verticles;

    int N = G.order();

    for (int vertex = 1; vertex <= N; vertex++) {
        verticles.insert(vertex);
    }
    while (!verticles.empty()) {
        auto V = verticles.begin();

        while (V != verticles.end()) {
            if (degree<GraphType>(*V, G) == 0) {
                sorted.push_back(*V);
                G.delete_vertex(*V);
                V = verticles.erase(V);
            } else {
                ++V;
            }
        }
    }
    return sorted;
}
```

Realizując algorytm Kahna dokonałem usprawnienia, które znacząco przyspieszyło działanie sortowania. Stopień kolejnych wierzchołków badany jest na podstawie metody sprawdzającej połączenie łukiem dwóch wierzchołków, to znaczy jeśli połączenie zostanie odnalezione z jakimkolwiek innym wierzchołkiem, oznacza iż nie ma on stopnia zerowego i procedura jest przerywana. Zauważyłem, że zazwyczaj wierzchołek jest następnikiem wierzchołków z większą aniżeli mniejsza etykietą, dlatego w moim algorytmie nie przeszukuję ich w porządku rosnącym, a malejącym.

Algorytm oparty o DFS

```cpp
template<class GraphType>
void visit(int vertex, std::map<int, int> &visited, GraphType &G,
std::deque<int> &sorted) {
    if (visited[vertex] == 2) {
        return;
    }
    visited[vertex] = 1;
    for (auto &successor : G.successors(vertex)) {
        visit<GraphType>(successor, visited, G, sorted);
    }
    visited[vertex] = 2;
    sorted.push_front(vertex);

}

template<class GraphType>
std::deque<int> DFS_sorting(GraphType G) {
    std::deque<int> sorted;
    std::map<int, int> visited;

    int N = G.order();
    for (int vertex = 1; vertex <= N; vertex++) {
        visited[vertex] = 0;
    }

    while (std::count_if(visited.begin(), visited.end(), [](auto p) { return
p.second == 2; }) != N) {
        for (auto &vertex : visited) {
            if (vertex.second != 2) {
                visit<GraphType>(vertex.first, visited, G, sorted);
            }
        }
    }
    return sorted;
}
```

Oba algorytmy zostały napisane biorąc pod uwagę założenie, że sortofany graf jest grafem skierowanym acyklicznym (DAG).

Generowanie instancji wejściowych

Do przetestowania efektywności reprezentacji należało wygenerować losowe instancje wejściowe. Nie powinny one zawierać cyklu, a powinny być dopełnione łukami w 50%. Aby wygenerować takie połączenia tworzę tablicę wielkości $\frac{k(k-1)}{2}$ (ponieważ tyle jest krawędzi w grafie pełnym), a następnie wypełniam ją losowo w połowie jedynkami. Dzięki temu ustalam, że tyle krawędzi będzie obecne w grafie. Dalej, by nie generować cykli badam wszystkie pary wierzchołków (u, v) spełniające warunek $u < v$ i zapisuję je, jeśli we wspomnianej na początku tablicy licznik wskazuje na element o wartości 1.

Ilustruje to poniższy algorytm napisany w języku Julia. Wizualizuje on także powstały graf skierowany.

In [ ]:

```julia
using GraphPlot, LightGraphs, Random, Colors
L = []
k = 6
n = ceil(k*(k-1)/2)
for i in 1:n
    push!(L, 0)
end
for i in 1:ceil(Int, n/2)
    L[i] = 1
end
shuffle!(L)
G = DiGraph(k)
counter = 0
for i in 1:k-1
    for j in 0:i-1
        global counter += 1
        if L[counter] == 1
            add_edge!(G, i+1, j+1)
        end
    end
end
gplot(G, nodefillc=sequential_palette(8, k, c=0.7), nodelabel=vertices(G), layout=circ
```

Porównanie efektywności sortowania topologicznego

Algorytm Kahna

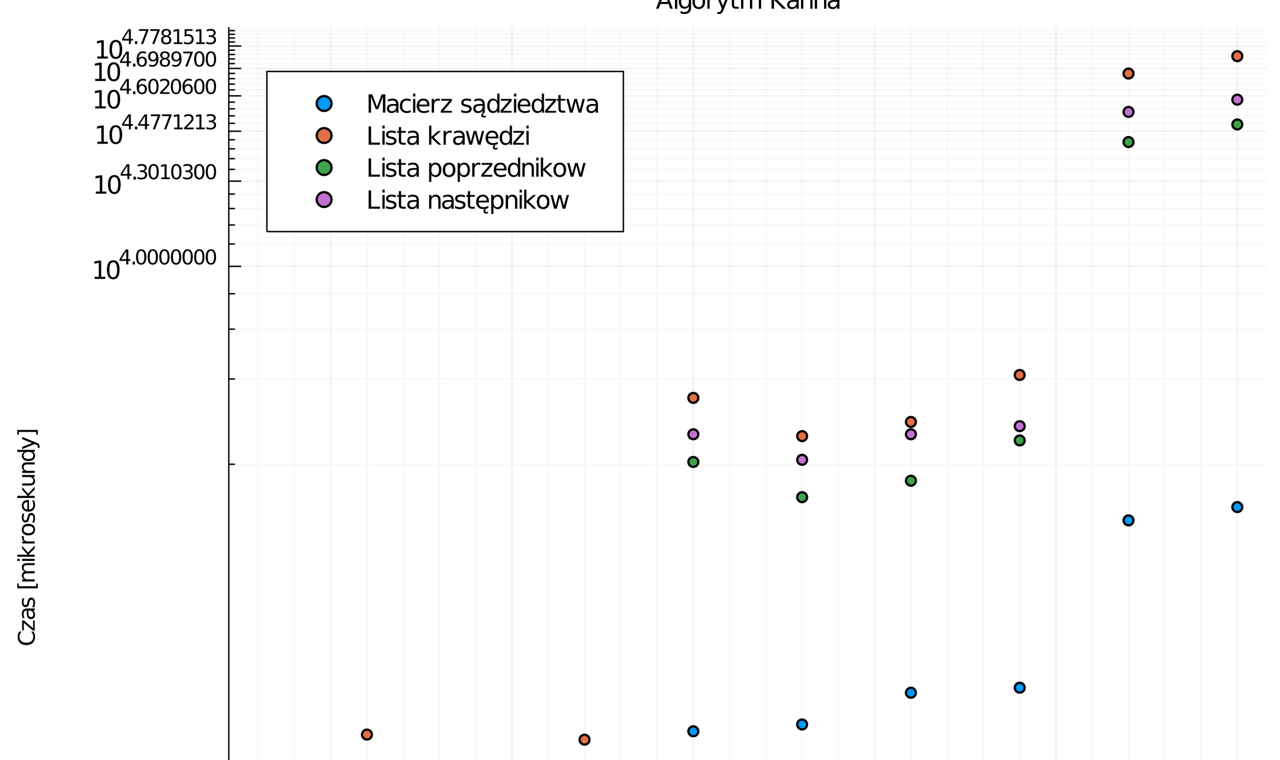
In [1]:

```julia
using CSV, Plots, DataFrames
d = CSV.read("Kahn.csv", DataFrame)
print(d)
labels = ["Macierz sądziedztwa" "Lista krawędzi" "Lista poprzednikow" "Lista następnil
data = [d.adjacency_matrix, d.edge_list, d.predecessor_list, d.successor_list]
fontsize = font(12)
scatter(d.rzad_grafu,
    data, xlabel="Liczba wierzchołkow", ylabel="Czas [mikrosekundy]",
    yaxis= :log, yticks = 0:10000:80000,
    label=labels, legend=:topleft,
    tickfont = fontsize, legendfont = fontsize, titlefont = fontsize,
    size = (900, 800),
    title = "Algorytm Kahna",
    minorgrid = true,
)
```

10×5 DataFrame

Row	rzad_grafu Int64	adjacency_matrix Int64	edge_list Int64	predecessor_list Int64	successor_list Int64
1	30	22	22	26	60
2	60	30	222	106	140
3	90	42	125	91	122
4	120	61	213	142	171
5	150	228	3431	2038	2552
6	180	241	2514	1530	2074
7	210	312	2821	1749	2552
8	240	325	4134	2427	2726
9	270	1267	47960	27474	35109
10	300	1412	55245	31701	38783

Out[1]:



Reprezentacja grafu, która w tym wypadku radzi sobie najlepiej to macierz sąsiedztwa. Jest to spowodowane faktem, że usuwanie wierzchołka z macierzy sąsiedztwa odbywa się przez wypełnienie wiersza i kolumny, których nie trzeba szukać, zerami. Tutaj również szybko odbywa się sprawdzenie czy dany stopień ma wierzchołek większy niż 0. Trochę więcej operacji wymagane jest przy usuwaniu wierzchołków z listy następników oraz listy poprzedników. Mają one podobne czasy wykonania. Najgorzej poradziła sobie lista krawędzi i wynika to z faktu, że jest ona największa strukturą danych (liczba krawędzi zwiększa się kwadratowo wraz ze wzrostem rozmiaru grafu), przez co zdecydowanie dłużej trwa usuwanie. Na wykresie można zauważyć, że dla części reprezentacji występują fluktuacje - dla większego grafu czas wykonania jest mniejszy. Jest to spowodowane operacjami na strukturach danych `std::unordered_map` oraz `std::set`.

Algorytm bazujący na DFS

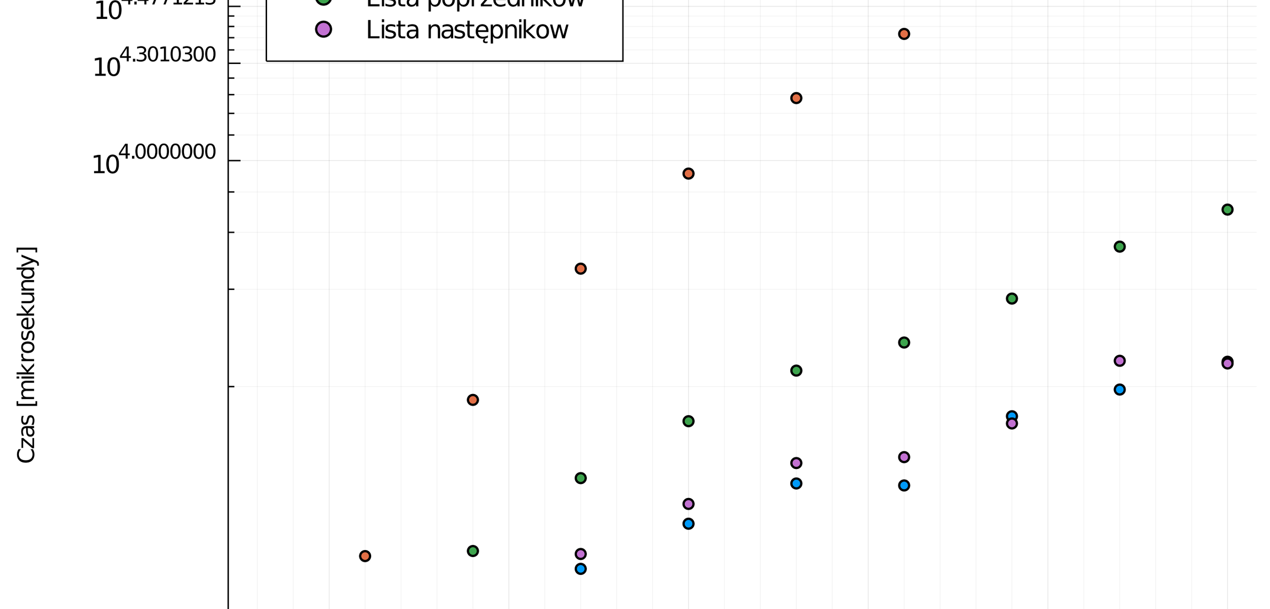
In [3]:

```julia
using CSV, Plots, DataFrames
d = CSV.read("DFS.csv", DataFrame)
print(d)
labels = ["Macierz sądziedztwa" "Lista krawędzi" "Lista poprzednikow" "Lista następnil
data = [d.adjacency_matrix, d.edge_list, d.predecessor_list, d.successor_list]
fontsize = font(12)
scatter(d.rzad_grafu,
    data, xlabel="Liczba wierzchołkow", ylabel="Czas [mikrosekundy]",
    yaxis= :log, yticks = 0:10000:80000,
    label=labels, legend=:topleft,
    tickfont = fontsize, legendfont = fontsize, titlefont = fontsize,
    size = (900, 800),
    title = "Algorytm bazujący na DFS",
    minorgrid = true,
)
```

10×5 DataFrame

Row	rzad_grafu Int64	adjacency_matrix Int64	edge_list Int64	predecessor_list Int64	successor_list Int64
1	300	88	140	110	91
2	600	210	598	325	224
3	900	359	1819	620	392
4	1200	546	4632	1042	607
5	1500	753	9117	1563	867
6	1800	1003	15612	2241	1160
7	2100	989	24654	2737	1210
8	2400	1619	36676	3743	1538
9	2700	1958	52555	5419	2403
10	3000	2387	71118	7054	2358

Out[3]:



W przypadku sortowania z wykorzystaniem algorytmu DFS również najszybsza okazała się macierz sąsiedztwa. Niskie czasy osiąga także lista następników, ponieważ DFS przegląda następników danego wierzchołka, a tu nie musimy ich generować. Wolniejsze okazały się lista poprzedników oraz lista krawędzi, którym zajmuje więcej czasu, aby wygenerować następników, szczególnie liście krawędzi, gdyż ma ona największą kolekcję danych do przeiterowania. Algorytm bazujący na DFS działa zdecydowanie szybciej od algorytmu Kahna, dlatego przetestowałem go na grafach z dziesięciokrotnie większą liczbą wierzchołków.

Podsumowanie

Jak pokazały testy, najefektywniejszą reprezentacją grafu okazała się być macierz sąsiedztwa. Można się było tego spodziewać, ponieważ tylko w tej strukturze dane o połączeniach między wierzchołkami są dostępne bez żadnego przeszukiwania, wystarczy odwołać się do indeksów. Z drugiej strony można postawić listę krawędzi, która będzie działać najwolniej, ponieważ przy n wierzchołkach w grafie krawędzi może być nawet $\frac{n(n-1)}{2}$ - o wiele więcej. Z tego też powodu zdecydowałem się używać przy wykresach skali logarytmicznej na osi Y, wtedy też bardziej widoczne stały się też różnice pomiędzy poszczególnymni reprezentacjamih.

Można dojśc do wniosku, że macierz sąsiedztwa działa tak sprawnie, ponieważ "zawiera" w prosto dostępnej formie dane, które, w pozostałych reprezentacjach musimy wyszukiwać. Z drugiej strony, jeśli potrzebne byłoby nam coś specyficznego, np. poprzednicy danego wierzchołka, ponieważ robilibyśmy algorytm sumujący stopnie wierzchołków, to lepiej sprawdzi się lista poprzedników. Pozostałe reprezentacje wydają się mieć węższe i bardziej specjalistyczne zastosowania.

Omawiane reprezentacje należałoby też rozpatrywać pod kątem tego jakie grafy chcemy w nich zapisywać. Jeśli są one z etykietami wierzchołków z nieciągłego zakresu, to nie jest to możliwe do zrealizowania przy pomocy macierzy grafu, ponieważ ta nie jest w stanie sygnalizować, że jakiegoś wierzchołka nie ma. Z kolei w liście krawędzi nie da się zapisać wierzchołka izolowanego. Lista następników/poprzedników pozwala w programie na przechowywanie wierzchołka z pustą listą, czyli informacji, że taki wierzchołek izolowany istnieje.