PAMSI	
Kierunek *Automatyka i Robotyka*	Termin *poniedziałek 15:15*
Imię, nazwisko, numer albumu *Artur Ziółkowski 259276*	Data *6 maja 2022*
Temat ćwiczenia *Projekt 2*	

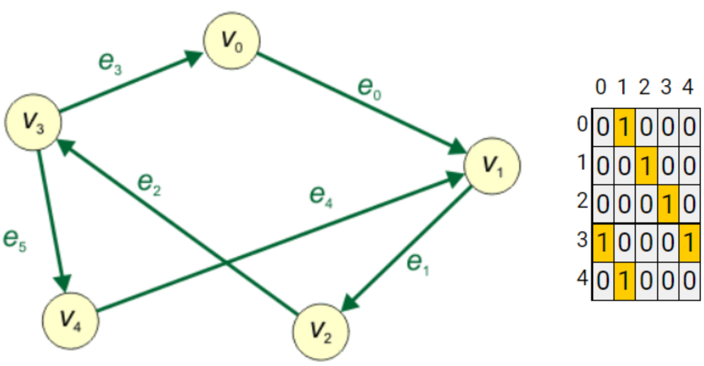
1 Wstęp

Celem ćwiczenia było na początku stworzenie grafu zareprezentowanego za pomocą macierzy i listy sąsiedztwa. Ponadto należało przygotować algorytm Dijkistry, który ma na celu znalezienie najkrótszej drogi od wybranego wierzchołka, do wszystkich pozostałych. Na końcu należało przeprowadzić testy efektywności zaimplementowanych algorytmów.

2 Reprezentacja grafu

2.1 Macierz sąsiedztwa

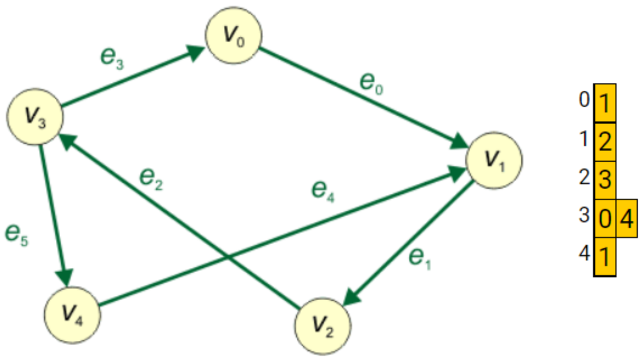
Najprostszym sposobem jest zaprezentowanie grafu pod postacią macierzy sąsiedztwa. Indeks wiersza w takiej macierzy symbolizuje wierchołek początkowy, indeks kolumny wierzchołek docelowy połączenia, a wartość w tym miejscu macierzy, wagę odpowiadającą temu połączeniu. Na zdjęciu poniżej pokazano przykład takiej macierzy.



Rysunek 1: Przykład macierzy sąsiedztwa.

2.2 Lista sąsiedztwa

Innym sposobem jest przedstawienie grafu za pomocą listy sąsiedztwa. Reprezentacja ta opiera się na tablicy, kórej indeks odpowiada kolejnym wierzchołkom. W tablizy znajdyją się listy, których poszczególnym elementem jest połączenie wychodzące z danego wierzchołka, niosące w sobie informację o wierzchołku docelowym i wadze połączenia. Poniżej przedstawiono przykład listy sąsiedztwa.



Rysunek 2: Przykład listy sąsiedztwa.

3 Algorytm Dijistry

Algorytm Dijistry jest przykładem algorytmu zachłannego. Oznaczo to, że nie dokonuje on oceny, czy dane działania lokalnie nie optymalne może skutkować decyzją globalnie optymalną. Otrzymane wynika zatem mogą nie być najszybszą uzyskaną drogą, a jedynie heurystyką. W trakcie wykonywania algorytmu dla każdego wierzchołka zostają wyznaczone dwie wartości: koszt dotarcia do tego wierzchołka oraz poprzedni wierzchołek na ścieżce. Na początku działania algorytmu dla wierzchołka źródłowego koszt dotarcia wynosi 0 (już tam jesteśmy), a dla każdego innego wierzchołka nieskończoność (w ogóle nie wiemy, jak się tam dostać). Wszystkie wierzchołki na początku znajdują się w zbiorze Q (są to wierzchołki nieprzejrzane). Następnie algorytm przebiega następująco:

Dopóki zbiór Q nie jest pusty:

1. Pobierz ze zbioru Q wierzchołek o najmniejszym koszcie dotarcia. Oznacz go jako v i usuń ze zbioru Q.
2. Dla każdej krawędzi wychodzącej z wierzchołka v (oznaczmy ją jako k) wykonaj następujące czynności:
3. Oznacz wierzchołek znajdujący się na drugim końcu krawędzi k jako u.
4. Jeśli koszt dotarcia do wierzchołka u z wierzchołka v poprzez krawędź k jest mniejszy od aktualnego kosztu dotarcia do wierzchołka u, to:
5. Przypisz kosztowi dotarcia do wierzchołka u koszt dotarcia do wierzchołka v powiększony o wagę krawędzi k.
6. Ustaw wierzchołek v jako poprzednik wierzchołka u.

4 Implementacja

Poniżej przedstawiono, w jaki sposób zaimplementowano obiekt grafu, jego reprezentacje macierzową i pod postacią listy, oraz algorytm Dijkistry osobny dla każdej reprezentacji.

4.1 Graf

W programie utworzono klasę grafu, z wygodną do użytkowania postacią. Poniżej pokazano plik nagłówkowy klasy grafu

```cpp
template <unsigned int vertices>
class Graph {
private:
    std::array<Linkedlist, vertices> adjacency_list_matrix;
    std::array<int, vertices * vertices> edge_weight_matrix;
    unsigned int density;
    void create_adjacency_list();
    unsigned int min_distance(std::array<unsigned int, vertices> &distance, std::array<bool, vertices> &Tset);
public:
    Graph(unsigned int density = 0);
    std::array<unsigned int, vertices> dijkstra_algorithm_matrix_repr(unsigned int vertex);
    std::array<unsigned int, vertices> dijkstra_algorithm_list_repr(unsigned int vertex);
    void reshuffle(unsigned int density = 0);

    void print_adjacency_list();
    template<unsigned int vertices1>
    friend std::ostream &operator<<(std::ostream &ost, Graph<vertices1> &graph);
};
```

Rysunek 3: Klasa grafu.

4.2 Algorytm Dijkistry dla macierzy sąsiedztwa

Poniżej przedstawiono implementacje algorytmu Dijkistry dla grafu reprezentowanego macierzą sąsiedztwa. Złożoność obliczeniowa algorytmu dla tej reprezentacji wynosi $O(V^2)$, gdzie V to liczba wierzchołków.

```cpp
template<unsigned int vertices>
std::array<unsigned int, vertices> Graph<vertices>::dijkstra_algorithm_matrix_repr(unsigned int vertex){
    std::array<unsigned int, vertices> distance;
    std::array<bool, vertices> Tset;

    for(int v = 0; v < vertices; ++v){
        distance[v] = INT_MAX;
        Tset[v] = false;
    }

    distance[vertex] = 0;

    for(int i = 0; i < vertices; ++i){
        unsigned int nearest = this->min_distance(distance, Tset);
        Tset[nearest] = true;
        for(int j = 0; j < vertices; ++j){
            if(!Tset[j] && this->edge_weight_matrix[vertices*nearest+j] && distance[nearest] != INT_MAX
                && distance[nearest] + this->edge_weight_matrix[vertices*nearest+j] < distance[j]){
                    distance[j] = distance[nearest] + this->edge_weight_matrix[vertices*nearest+j];
                }
        }
    }


    return distance;
}
```

Rysunek 4: Implementacja algorytmu Dijkistry dla grafu reprezentowanego macierzą sąsiedztwa.

4.3 Algorytm Dijkistry dla listy sąsiedztwa

Poniżej znajduje się implementacja algorytmu Dijkistry dla grafu reprezentowanego listą sąsiedztwa. Złożoność obliczeniowa algorytmu dla tej implementacji wynosi $O(Elog(V)$, gdzie V to liczba wierzchołków, a E to liczba połączeń. Wykorzystana została implementacja z wykorzystaniem kopca.

```cpp
template<unsigned int vertices>
std::array<unsigned int, vertices> Graph<vertices>::dijkstra_algorithm_list_repr(unsigned int vertex){

    unsigned int distance[vertices];

    PriorityQueue* min_heap = new PriorityQueue(vertices);

    for(int v = 0; v < vertices; ++v){
        distance[v] = INT_MAX;
        min_heap->array[v] = new PriorityQueueNode(v, distance[v]);

        min_heap->pos[v] = v;
    }

    min_heap->array[vertex] = new PriorityQueueNode(vertex, distance[vertex]);
    min_heap->pos[vertex] = vertex;
    distance[vertex] = 0;
    min_heap->decrease_key(vertex, distance[vertex]);

    min_heap->size = vertices;

    while(!min_heap->is_empty()){
        PriorityQueueNode* min_heap_node = min_heap->extract_min();

        unsigned int u = min_heap_node->vertex;

        Edge * pCrawl = this->adjacency_list_matrix[u].head;

        while(pCrawl != NULL){
            unsigned int v = pCrawl->vertex;

            if(min_heap->is_in_queue(v) && distance[u] != INT_MAX
                && pCrawl->weight + distance[u] < distance[v]){
                    distance[v] = distance[u] + pCrawl->weight;

                    min_heap->decrease_key(v, distance[v]);
                }
                pCrawl = pCrawl->next;
        }
    }

    std::array<unsigned int, vertices> result;
    for (int i = 0; i < vertices; ++i){
        result[i]= distance[i];
    }

    return result;
}
```

Rysunek 5: Implementacja algorytmu Dijkistry dla grafu reprezentowanego listą sąsiedztwa.

5 Wyniki testów wydajności

Na przygotowanych grafach dla obu reprezentacji przeprowadzono badania wydajności. Zbadano szybkość działania algorytmu dla gęstości grafu równej 25%, 50%, 75% i 100%, dla ilości wierzchołków 50, 100, 150, 200, 250. Badania przeprowadzono i usresniono dla 100 próbek. Tabele z wynikami przedstawiono poniżej.

Otrzymane dane przedstawiono na wykresach czasu wykonywania od ilości wierzchołków dla każdej gęstości i reprezentacji grafów (Typ 1 wykresu), oraz wykresach czasu wykonywania od gęstości grafów dla wszystkich ilości wierzchołków i reprezentacji grafów (Typ 2 wykresu).

Reprezentacja Macierzowa

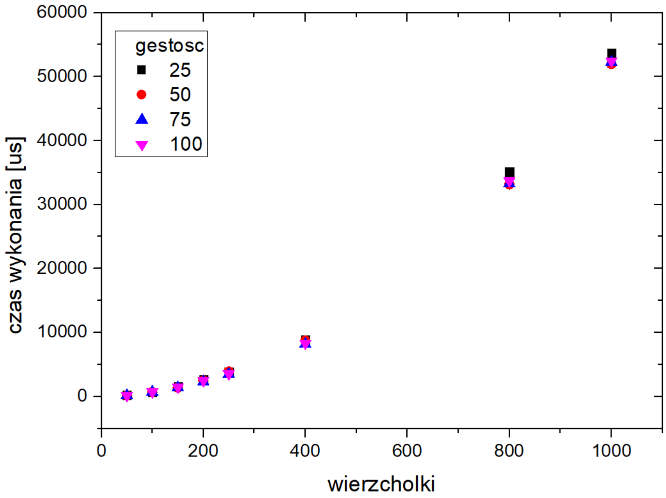
Gęstość \ Elementy	25% [us]	50% [us]	75% [us]	100% [us]
50	183.53	152.02	170.32	138.36
100	680.23	682.11	719.94	740.14
150	1532.66	1383.97	1392.06	1446.2
200	2600.11	2488.15	2347.48	2485.9
250	3826.16	3901.37	3511.71	3556.93
400	8838.88	8728.64	8220.41	8301.95
800	35119.9	33148.7	33295.6	33717.3
1000	53682.7	51960.5	52280.1	52454.7

Rysunek 6: Tabela wyników testów wydajności dla reprezentacji macierzy sąsiedztwa.

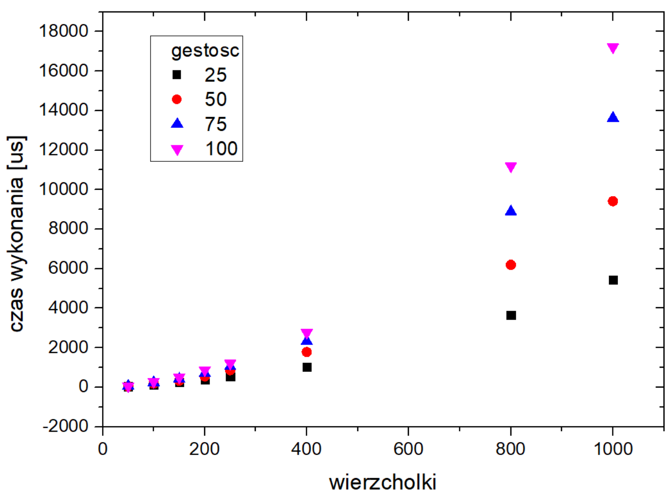
Reprezentacja za pomocą listy

Gęstość \ Elementy	25% [us]	50% [us]	75% [us]	100% [us]
50	33.72	41.7	59.39	56.17
100	122.41	183.11	241.01	282.15
150	250.85	328.35	422.94	498.64
200	369.31	540.64	706.24	868.43
250	529.64	857.97	1057.79	1208.67
400	1026.66	1777.86	2326.72	2757.11
800	3661.45	6194.86	8887.91	11197.8
1000	5443.88	9417.42	13627.9	17213.6

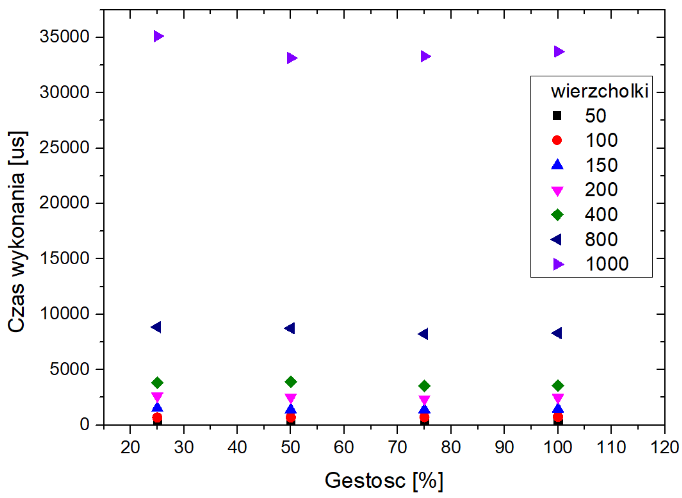
Rysunek 7: Tabela wyników testów wydajności dla reprezentacji listy sąsiedztwa.



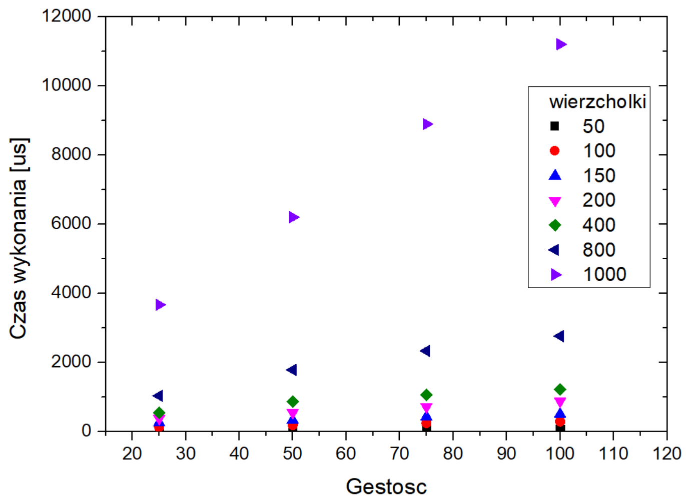
Rysunek 8: Czas wykonywania algorytmu w zależności od ilości wierzchołków dla każdej gęstości dla reprezentacji grafu pod postacią macierzy.



Rysunek 9: Czas wykonywania algorytmu w zależności od ilości wierzchołków dla każdej gęstości dla reprezentacji grafu pod postacią listy.



Rysunek 10: Czas wykonywania algorytmu w zależności od gęstości grafu dla każdej ilości wierczhołków dla reprezentacji grafu pod postacią macierzy.



Rysunek 11: Czas wykonywania algorytmu w zależności od gęstości grafu dla każdej ilości wierczhołków dla reprezentacji grafu pod postacią listy.

6 Wnioski

Na podstawie zebranych danych można zauważyć, że algorytm dla reprezentacji pod postacią listy, zgodnie z oczekiwaniami zależy nie tylko od ilości wierchołków, ale również od ilości brzegów. Świadczy o tym rosnący czas wykonywania algorytmu względem gęstości grafu. Nie jest to jednak obecne dla reprezentacji macierzowej. Tam Wykresy typu 2 są względnie stałe. Czas wyonywania algorytmu w przypadku listy, jest krótszy. Związane jest to między innymi z krótszym czasem wyszukiwania elementów w liście. Na podstawie otrzymanych wykresów można potwierdzić, że złożonośc obliczeniowa algorytmu dla reprezentacji macierzowej zależy od kwadratu wierzchołków, a dla reprezentacji listowej zależy od ilości wierzchołków w pierwszej potędze. Dla dużych grafów, reprezentacja pod postacią listy jest bardziej uzasadniona, pod warunkiem, że liczba brzegów jest znacznie ograniczona. W przypadku grafu pełnego, dla którego liczba brzegóciężko jednakw jest równa kwadratowi liczby wierzchołków, zastosowanie tej reprezentacji nie jest optymalne.