Politechnika Wrocławska

Projektowanie i analiza algorytmów

Projekt 3

Freddy-Ms

Poniedziałek 17:05 - 18:45

26 maja 2024

1 Wstęp

Algorytm Dijkstry to fundamentalne narzędzie w teorii grafów, używane do znajdowania najkrótszej ścieżki w grafie o nieujemnych wagach krawędzi.

W niniejszym sprawozdaniu przedstawię działanie algorytmu Dijkstry oraz jego implementację przy użyciu dwóch popularnych struktur danych: listy sąsiedztwa i macierzy sąsiedztwa.

Lista sąsiedztwa reprezentuje graf w postaci tablicy, gdzie każdy indeks odpowiada wierzchołkowi, a przypisane mu listy zawierają pary (wierzchołek, waga krawędzi). Jest to efektywne podejście w przypadku grafów rzadkich, gdzie liczba krawędzi jest znacznie mniejsza od kwadratu liczby wierzchołków.

Macierz sąsiedztwa, z kolei, to dwuwymiarowa tablica, gdzie elementy macierzy określają wagę krawędzi między parą wierzchołków. Jest to wygodne w przypadku grafów gęstych, gdzie większość możliwych krawędzi jest obecna.

2 Fragmenty kodu

W niniejszej pracy zaprezentowane zostaną kluczowe fragmenty kodu, obejmujące dodawanie krawędzi do grafu, generowanie struktury grafu oraz implementację algorytmu Dijkstry, wykorzystując zarówno listę sąsiedztwa, jak i macierz sąsiedztwa. Podczas tworzenia grafu i dodawania do niego krawędzi, jednocześnie uzupełniane są odpowiednie struktury danych: lista sąsiedztwa oraz macierz sąsiedztwa. W macierzy sąsiedztwa wartość 0 oznacza brak krawędzi między odpowiadającymi sobie wierzchołkami.

2.1 Konstruktor

```
Graph::Graph(int Vertex) {
    this->Vertex = Vertex;
    this->Edges = 0;
    InitializationMatrix();
    AdjList = new list<vPair>[Vertex];
}
```

Podczas tworzenia grafu, jako parametr podajemy liczbę wierzchołków, co pozwala na odpowiednie zainicjowanie struktur danych. Inicjalizacja listy sąsiedztwa oraz macierzy sąsiedztwa jest niezbędnym krokiem w przygotowaniu grafu, umożliwiającym dalsze operacje na grafie, takie jak dodawanie krawędzi czy przeszukiwanie grafu.

2.2 Dodawanie krawędzi

```
void Graph::addEdge(int V1, int V2, int weight) {
    AdjMatrix[V1][V2] = weight;
    AdjMatrix[V2][V1] = weight;
    AdjList[V1].push_back(make_pair(V2, weight));
    AdjList[V2].push_back(make_pair(V1, weight));
    this->Edges++;
}
```

Funkcja przyjmuje jako parametry dwa wierzchołki oraz wagę krawędzi łączącej te wierzchołki, a następnie dokonuje ich dodania do listy sąsiedztwa oraz odpowiedniego zapisu w macierzy sąsiedztwa.

2.3 Algorytm Dijkstry przy użyciu listy sąsiedztwa

```
    void Graph::DijkstraWithList(int StartVertex){
    priority_queue<vPair, vector<vPair>, greater<vPair>> pq;
    int *dist = new int[this->Vertex];
    bool *visited = new bool[this->Vertex];
    for(int i = 0; i<this->Vertex; i++){
        dist[i] = INT_MAX;
        visited[i] = false;
    }
    pq.push(make_pair(0, StartVertex));
    dist[StartVertex] = 0;
    while(!pq.empty()){
        int Vertex = pq.top().second;
        pq.pop();
        if(visited[Vertex] == true)
            continue;

        for(auto i : AdjList[Vertex]){
            int DestinationVertex = i.first;
            int weight = i.second;
            if(dist[DestinationVertex] > dist[Vertex] + weight){
                dist[DestinationVertex] = dist[Vertex] + weight;
                pq.push(make_pair(dist[DestinationVertex], DestinationVertex));
            }
        }
        visited[Vertex] = true;
    }
    printSolution(dist);
    delete[] dist;
    delete[] visited;
}
```

Algorytm przyjmuje jako parametr wejściowy wierzchołek początkowy, od którego rozpoczyna swą podróż. Inicjalizuje strukturę danych w postaci kolejki priorytetowej, upewniając się, że najmniejsza wartość znajduje się na jej początku, oraz pomocnicze tabele przechowujące informacje o odległościach do wierzchołków oraz ich odwiedzeniu. Następnie, początkowy wierzchołek zostaje dodany do kolejki priorytetowej, rozpoczynając tym samym algorytm. Dla każdego wierzchołka z kolejki priorytetowej, algorytm przeszukuje jego połączenia i wagi, weryfikując, czy nowa ścieżka do danego wierzchołka jest krótsza od dotychczasowej. W przypadku stwierdzenia krótszej drogi, aktualizuje on informacje o odległości oraz dodaje nowy wierzchołek do kolejki priorytetowej z uwzględnieniem nowego dystansu. Na zakończenie, wierzchołek zostaje oznaczony jako odwiedzony, uniemożliwiając powtórne przetwarzanie go w przyszłości.

2.4 Algorytm Dijkstry przy użyciu macierzy sąsiedztwa

```
void Graph::DijkstraWithMatrix(int StartVertex){
    priority_queue<vPair, vector<vPair>, greater<vPair>> pq;
    int *dist = new int[this->Vertex];
    bool *visited = new bool[this->Vertex];
    for(int i = 0; i<this->Vertex; i++){
        dist[i] = INT_MAX;
        visited[i] = false;
    }
    pq.push(make_pair(0, StartVertex));
    dist[StartVertex] = 0;
    while(!pq.empty()){
        int Vertex = pq.top().second;
        pq.pop();
        if(visited[Vertex] == true)
            continue;
        for(int i = 0; i<this->Vertex; i++){
            if(AdjMatrix[Vertex][i] != 0){
                int DestinationVertex = i;
                int weight = AdjMatrix[Vertex][i];
                if(dist[DestinationVertex] > dist[Vertex] + weight){
                    dist[DestinationVertex] = dist[Vertex] + weight;
                    pq.push(make_pair(dist[DestinationVertex], DestinationVertex));
                }
            }
        }
        visited[Vertex] = true;
    }
    printSolution(dist);
    delete[] dist;
    delete[] visited;
}
```

Algorytm ten operuje w sposób analogiczny do swojego poprzednika, z jednym istotnym rozróżnieniem. Podczas analizy macierzy sąsiedztwa, algorytm poszukuje wartości różnych od zera, co wskazuje na istnienie krawędzi między odpowiednimi wierzchołkami.

2.5 Funkcja generująca losowy graf

```
void Graph::generateGraph(int density){
    vector<pair<int, int>> allEdges;
    set<pair<int, int>> usedEdges;
    random_device rd;
    default_random_engine rng(rd());
    FullfilZerosMatrix();
    ClearAdjList();
    this->Edges = 0;
    for (int u = 0; u < this->Vertex; ++u) {
        for (int v = u + 1; v < this->Vertex; ++v) {
            allEdges.push_back(make_pair(u, v));
        }
    }

    shuffle(allEdges.begin(), allEdges.end(), rng);

    int totalPossibleEdges = allEdges.size();
    int edgesToAdd = (density * totalPossibleEdges) / 100;

    for (int i = 0; i < this->Vertex - 1; ++i) {
        int u = i;
        int v = i + 1;
        int weight = rand() % 100 + 1;
        addEdge(u, v, weight);
        usedEdges.insert(make_pair(min(u, v), max(u, v)));
    }

    edgesToAdd -= (this->Vertex - 1);

    int edgeIndex = 0;
    while (edgesToAdd > 0 && edgeIndex < totalPossibleEdges) {
        int u = allEdges[edgeIndex].first;
        int v = allEdges[edgeIndex].second;

        if (usedEdges.find(make_pair(min(u, v), max(u, v))) != usedEdges.end()) {
            edgeIndex++;
            continue;
        }

        addEdge(u, v, rand() % 100 + 1);
        usedEdges.insert(make_pair(min(u, v), max(u, v))); // Track the added edge
        edgesToAdd--;
        edgeIndex++;
    }
}
```

Funkcja przyjmuje jako parametr wejściowy gęstość grafu wyrażoną w procentach. W celu przygotowania struktury grafu, pierwotnie wypełnia macierz sąsiedztwa zerami oraz czyści listę sąsiedztwa, zapewniając pełną kontrolę nad stanem grafu. Następnie generuje wszystkie możliwe pary połączeń między wierzchołkami, które są następnie losowo wymieszane za pomocą odpowiednich funkcji, zapewniając losowy układ krawędzi w grafie. Kolejnym krokiem jest obliczenie liczby wszystkich potencjalnych krawędzi oraz na podstawie określonej gęstości grafu, obliczenie ilości krawędzi, które należy dodać. Aby zapobiec izolacji pewnych wierzchołków, każdy z wierzchołków musi być połączony co najmniej z jednym innym wierzchołkiem. W celu uniknięcia duplikacji krawędzi, funkcja przechowuje już użyte krawędzie, co pozwala uniknąć konfliktów w przyszłych operacjach. Wreszcie, funkcja dodaje określoną liczbę krawędzi, dbając o to, aby każda z nich była unikalna, co eliminuje możliwość nadpisania już istniejących połączeń.

3 Badania

Badania zostały przeprowadzone zgodnie z wytycznymi prowadzącego. Poniżej przedstawiono kod odpowiedzialny za przeprowadzenie badań.

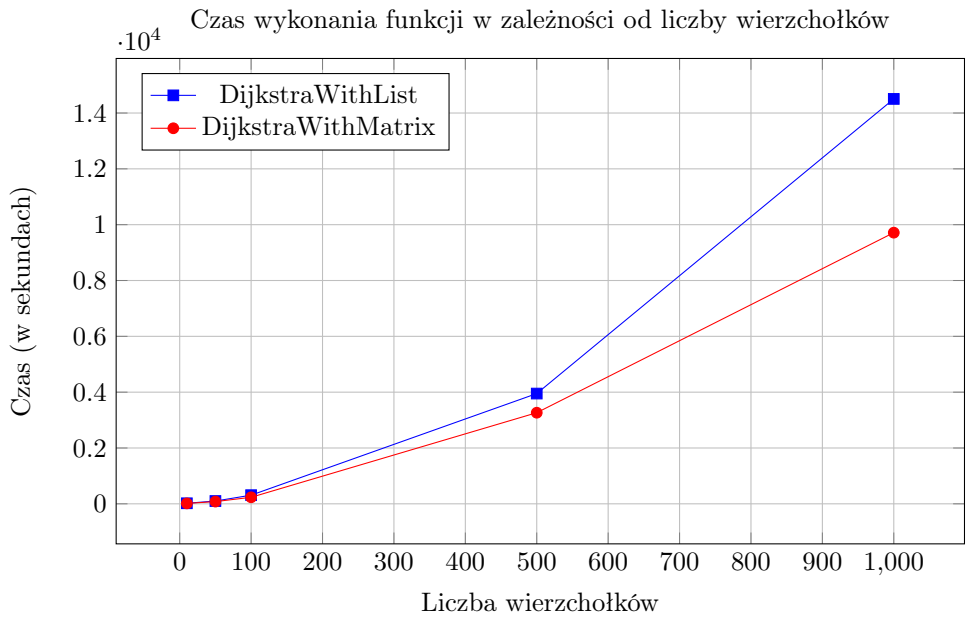
```
const int density[] = {25,50,75,100};
const int vertices[] = {10,50,100,500,1000};
void Tests()
{
    std::chrono::high_resolution_clock::time_point StartList, EndList,StartMatrix,
        EndMatrix;
    std::chrono::microseconds ElapsedList, ElapsedMatrix;
    for(int i = 0; i<4; i++){
        for(int j = 0; j<5;j++){
            ElapsedList = std::chrono::microseconds::zero();
            ElapsedMatrix = std::chrono::microseconds::zero();
            for(int k = 0; k<100;k++){
                Graph *g = new Graph(vertices[j]);
                g->generateGraph(density[i]);
                StartList = std::chrono::high_resolution_clock::now();
                g->DijkstraWithList(0);
                EndList = std::chrono::high_resolution_clock::now();
                ElapsedList += std::chrono::duration_cast<std::chrono::microseconds>(
                    EndList - StartList);
                StartMatrix = std::chrono::high_resolution_clock::now();
                g->DijkstraWithMatrix(0);
                EndMatrix = std::chrono::high_resolution_clock::now();
                ElapsedMatrix += std::chrono::duration_cast<std::chrono::microseconds>(
                    EndMatrix - StartMatrix);
                delete g;
            }
            cout << "Average time taken by function(DijkstraWithList) with density " <<
                 density[i] << " and vertices " << vertices[j] << " : " << ElapsedList.
                count()/100 << " microseconds" << endl;
            cout << "Average time taken by function(DijkstraWithMatrix) with density "
                << density[i] << " and vertices " << vertices[j] << " : " <<
                ElapsedMatrix.count()/100 << " microseconds" << endl;
        }

    }
}
```

3.1 Gęstość 25%

Tabela 1: Gęstość 25%

Liczba wierzchołków	DijkstraWithList (czas w mikrosekundach)	DijkstraWithMatrix (czas w mikrosekundach)
10	20	15
50	100	80
100	310	234
500	3950	3264
1000	14505	9714

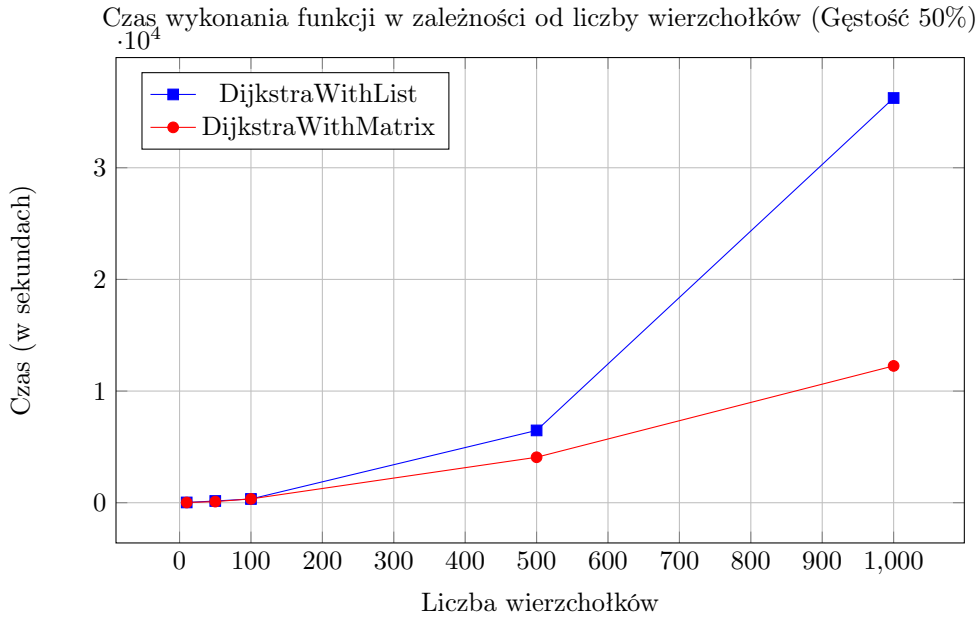


Rysunek 1: Czas wykonania funkcji DijkstraWithList i DijkstraWithMatrix dla gęstości 25%

3.2 Gęstość 50%

Tabela 2: Gęstość 50%

Liczba wierzchołków	DijkstraWithList (czas w mikrosekundach)	DijkstraWithMatrix (czas w mikrosekundach)
10	25	20
50	155	100
100	336	342
500	6476	4072
1000	36246	12254

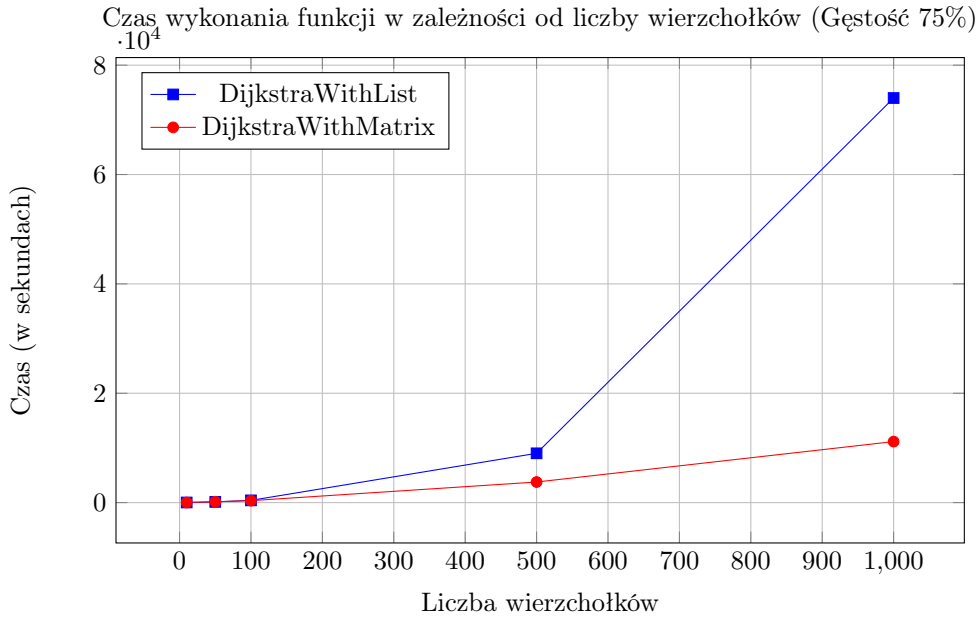


Rysunek 2: Czas wykonania funkcji DijkstraWithList i DijkstraWithMatrix dla gęstości 50%

3.3 Gęstość 75%

Tabela 3: Gęstość 75%

Liczba wierzchołków	DijkstraWithList (czas w mikrosekundach)	DijkstraWithMatrix (czas w mikrosekundach)
10	23	19
50	130	160
100	411	360
500	9017	3766
1000	73990	11159

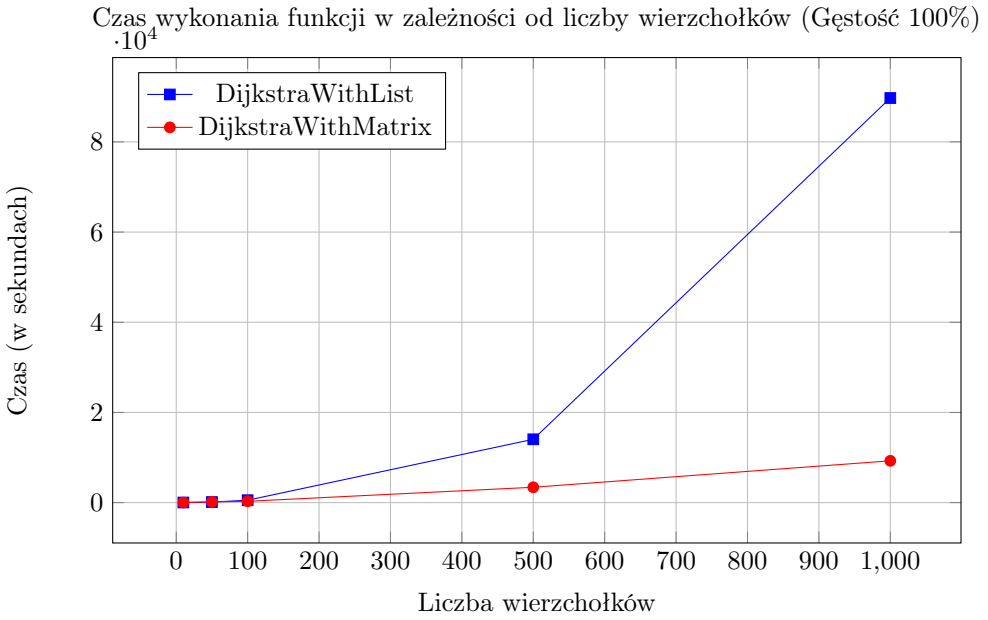


Rysunek 3: Czas wykonania funkcji DijkstraWithList i DijkstraWithMatrix dla gęstości 75%

3.4 Graf pełny

Tabela 4: Gęstość 100%

Liczba wierzchołków	DijkstraWithList (czas w mikrosekundach)	DijkstraWithMatrix (czas w mikrosekundach)
10	25	20
50	125	160
100	534	295
500	14044	3405
1000	89729	9269



Rysunek 4: Czas wykonania funkcji DijkstraWithList i DijkstraWithMatrix dla gęstości 100%

4 Wnioski

Zgodnie z teoretycznymi założeniami, złożoność obliczeniowa algorytmu Dijkstry wykorzystującego macierz sąsiedztwa wynosi $O(V^2)$, podczas gdy przy zastosowaniu listy sąsiedztwa wynosi $O(E * \log(V))$, gdzie V oznacza liczbę wierzchołków, a E liczbę krawędzi. Niemniej jednak, wyniki empiryczne z przeprowadzonych badań wykazały powtarzalne zjawisko, w którym algorytm działający na bazie listy sąsiedztwa wykazywał dłuższy czas wykonania w porównaniu do jego odpowiednika korzystającego z macierzy sąsiedztwa. Analiza wykresowa ukazała, że czas wykonania algorytmu opartego na liście sąsiedztwa wykazuje tendencję kwadratową, w przeciwieństwie do liniowej charakterystyki obserwowanej w przypadku algorytmu korzystającego z macierzy sąsiedztwa. Niezależnie od zastosowanej reprezentacji grafu, obydwa algorytmy skutecznie identyfikują najkrótsze ścieżki od wierzchołka źródłowego do pozostałych wierzchołków, działając przy tym w sposób szybki i wydajny.

5 Bibliografia

- https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7
- https://pl.wikipedia.org/wiki/Algorytm_Dijkstry

- Cormen T.; Leiserson C.E.; Rivest R.L.; Stein C.: Wprowadzenie do algorytmów, WNT