Politechnika Wrocławska

Projektowanie i analiza algorytmów

Projekt 3

ArtsyKimiko

6 czerwca 2024

1 Wstęp

Algorytm Dijkstry to jedno z kluczowych narzędzi w teorii grafów, służące do wyznaczania najkrótszej ścieżki w grafach z nieujemnymi wagami krawędzi. Jego główne zalety to deterministyczność oraz możliwość zastosowania w różnych typach grafów, co czyni go uniwersalnym narzędziem w analizie i rozwiązywaniu problemów związanych z najkrótszymi ścieżkami.

Lista sąsiedztwa reprezentuje graf jako strukturę danych, w której każdy wierzchołek jest przypisany do tablicy, a przylegające do niego wierzchołki są zapisane jako pary (wierzchołek, waga krawędzi). To podejście staje się korzystne szczególnie w przypadku grafów o niskiej gęstości, gdzie liczba krawędzi jest znacznie mniejsza od liczby wierzchołków podniesionej do kwadratu. Taka reprezentacja umożliwia oszczędność pamięci oraz przyspieszenie operacji związanych z analizą grafu, co jest istotne w przypadku aplikacji pracujących na dużych zbiorach danych, gdzie grafy mogą być rozległe, a połączenia występują rzadko.

Natomiast, macierz sąsiedztwa to dwuwymiarowa tablica, gdzie każdy element reprezentuje wagę krawędzi między dwoma wierzchołkami. Takie podejście jest szczególnie użyteczne w przypadku grafów o dużej gęstości, gdzie większość możliwych połączeń między wierzchołkami istnieje. Dzięki tej reprezentacji, łatwo jest uzyskać informacje o wszystkich krawędziach w grafie oraz ich wagach, co może być istotne w analizie struktury oraz zachowań w grafach o wysokiej liczbie krawędzi.

2 Opis algorytmów

Ten algorytm to implementacja Dijkstry dla grafu reprezentowanego listą sąsiedztwa. Inicjalizuje struktury danych: kolejkę priorytetową, tablice odległości oraz informacje o sprawdzonych wierzchołkach. Następnie, dla wierzchołka startowego, ustawia odległość na 0 i dodaje go do kolejki priorytetowej. W głównej pętli algorytmu, dopóki kolejka nie jest pusta, pobiera wierzchołek o najmniejszej odległości. Dla każdego sąsiada, sprawdza krawędzie i aktualizuje odległość, jeśli nowa droga jest krótsza. Po przetworzeniu wszystkich sąsiadów, oznacza wierzchołek jako sprawdzony. Na koniec usuwa zaalokowane tablice dynamiczne. Algorytm oblicza najkrótsze ścieżki od wierzchołka startowego do pozostałych wierzchołków w grafie.

2.1 Algorytm Dijkstry z wykorzystaniem listy

```
void Graph::DijkstraList(int start)
{
    priority_queue<vPair, vector<vPair>, greater<vPair>> queue;
    int *totalLength=new int[this->vertexCount];
    bool *checked=new bool[this->vertexCount];
    for(int i=0; i<this->vertexCount; i++)
    {
        totalLength[i]=INT_MAX;
        checked[i]=false;
    }
    queue.push(make_pair(0, start));
    totalLength[start]=0;
    while(!queue.empty())
    {
        int vertexCount=queue.top().second;
        queue.pop();
        if(checked[vertexCount]==true)
            continue;

        for(auto i:List[vertexCount])
        {
            int target=i.first;
            if(checked[target]==true)
                continue;
            int length=i.second;

            if(totalLength[target]>totalLength[vertexCount]+length)
            {
                totalLength[target]=totalLength[vertexCount]+length;
                queue.push(make_pair(totalLength[target], target));
            }
        }
        checked[vertexCount]=true;
    }
    delete[] totalLength;
    delete[] checked;
}
```

2.2 Algorytm Dijkstry z wykorzystaniem macierzy

Ten algorytm implementuje algorytm Dijkstry do znajdowania najkrótszych ścieżek w grafie ważonym z wykorzystaniem macierzy sąsiedztwa. Inicjalizuje struktury danych, takie jak kolejka priorytetowa i tablice do przechowywania odległości oraz informacji o sprawdzeniu wierzchołków. Następnie, dla wierzchołka startowego, ustawia odległość na 0 i dodaje go do kolejki priorytetowej. W głównej pętli, pobiera wierzchołek o najmniejszej odległości z kolejki i aktualizuje odległości do jego sąsiadów, jeśli nowa droga jest krótsza. Po przetworzeniu wszystkich sąsiadów, oznacza wierzchołek jako sprawdzony. Na koniec usuwa zaalokowane wcześniej tablice dynamiczne. Algorytm ten oblicza najkrótsze ścieżki od wierzchołka startowego do wszystkich pozostałych wierzchołków w grafie.

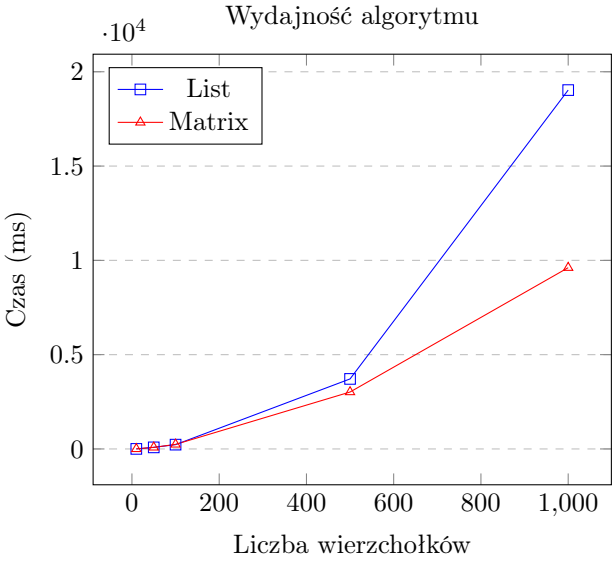
```
void Graph::DijkstraMatrix(int start)
{
    priority_queue<vPair, vector<vPair>, greater<vPair>> queue;
    int *totalLength=new int[this->vertexCount];
    bool *checked=new bool[this->vertexCount];
    for(int i=0; i<this->vertexCount; i++)
    {
        totalLength[i]=INT_MAX;
        checked[i]=false;
    }
    queue.push(make_pair(0, start));
    totalLength[start]=0;
    while(!queue.empty())
    {
        int vertexCount=queue.top().second;
        queue.pop();
        if(checked[vertexCount]==true)
            continue;
        for(int i=0; i<this->vertexCount; i++)
            if(matrix[vertexCount][i]!=0)
            {
                int target=i;
                if(checked[target]==true)
                    continue;
                int length=matrix[vertexCount][i];
                if(totalLength[target]>totalLength[vertexCount]+length)
                {
                    totalLength[target]=totalLength[vertexCount]+length;
                    queue.push(make_pair(totalLength[target], target));
                }
            }
        checked[vertexCount]=true;
    }
    delete[] totalLength;
    delete[] checked;
}
```

3 Badania

3.1 Gęstość grafu 25%

Liczba wierzchołków	Dijkstra List	Dijkstra Matrix
10	3	3
50	84	85
100	228	242
500	3713	3019
1000	19029	9612

Tabela 1: Gęstość: 25%



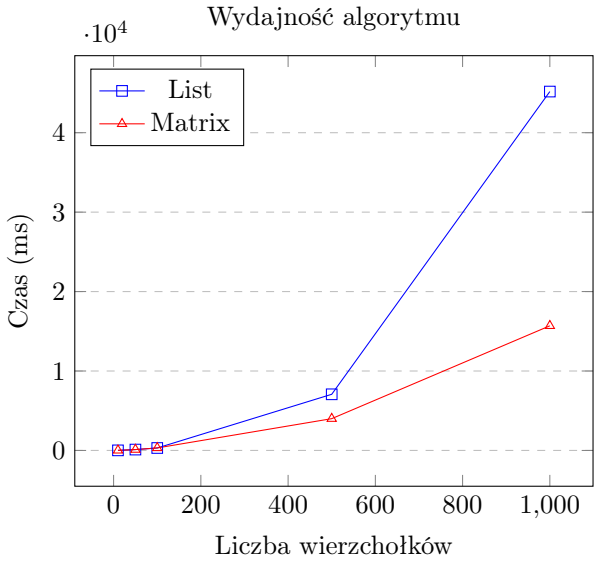
Rysunek 1: Porównanie algorytmu Dijkstry przy użyciu listy i macierzy

Dla grafów o gęstości 25%, algorytm Dijkstry wykazuje podobną wydajność zarówno przy użyciu listy sąsiedztwa, jak i macierzy sąsiedztwa. Wraz ze wzrostem liczby wierzchołków, czas wykonania algorytmu rośnie, jednak różnice pomiędzy wykorzystaniem obu struktur danych pozostają niewielkie.

3.2 Gęstość grafu 50%

Liczba wierzchołków	Dijkstra List	Dijkstra Matrix
10	10	10
50	109	109
100	307	314
500	7068	4000
1000	45180	15690

Tabela 2: Gęstość: 50%



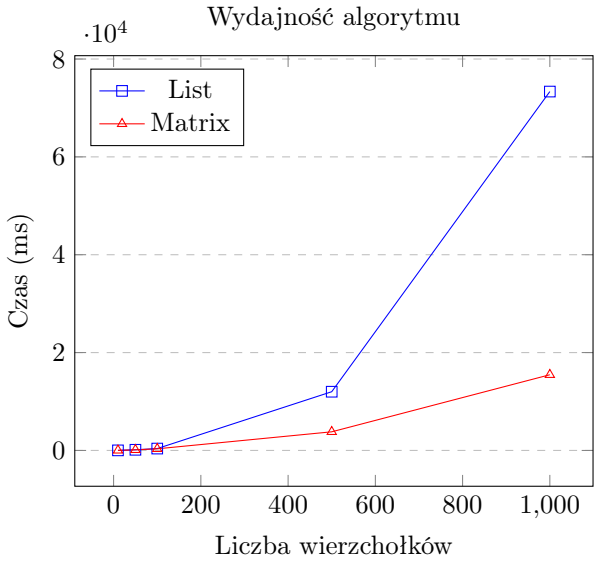
Rysunek 2: Porównanie algorytmu Dijkstry przy użyciu listy i macierzy

Dla grafów o gęstości 50%, zarówno lista sąsiedztwa, jak i macierz sąsiedztwa prezentują podobne czasy wykonania algorytmu Dijkstry. Wraz ze wzrostem liczby wierzchołków, czas potrzebny na przetworzenie grafu rośnie proporcjonalnie, co sugeruje, że obie struktury danych sprawdzają się podobnie w tego typu przypadkach.

3.3 Gęstość grafu 75%

Liczba wierzchołków	Dijkstra List	Dijkstra Matrix
10	14	13
50	131	126
100	382	333
500	12000	3807
1000	73349	15469

Tabela 3: Gęstość: 75%



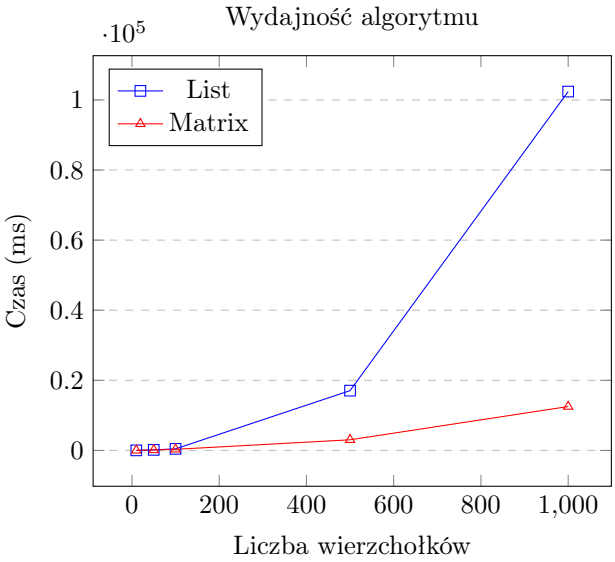
Rysunek 3: Porównanie algorytmu Dijkstry przy użyciu listy i macierzy

Dla grafów o gęstości 75%, wydajność algorytmu Dijkstry przy użyciu listy sąsiedztwa jest zwykle lepsza niż przy wykorzystaniu macierzy sąsiedztwa. Wraz ze wzrostem liczby wierzchołków, różnice w czasie wykonania między tymi dwoma podejściami stają się bardziej zauważalne.

3.4 Graf pełny

Liczba wierzchołków	Dijkstra List	Dijkstra Matrix
10	14	13
50	152	126
100	447	311
500	17055	3030
1000	102385	12516

Tabela 4: Graf pełny



Rysunek 4: Porównanie algorytmu Dijkstry przy użyciu listy i macierzy

Dla grafów pełnych, czyli o maksymalnej gęstości, algorytm Dijkstry zaimplementowany przy użyciu listy sąsiedztwa wykazuje tendencję do dłuższego czasu wykonania w porównaniu do macierzy sąsiedztwa. Wraz ze wzrostem liczby wierzchołków, różnice te stają się coraz bardziej wyraźne, co sugeruje, że dla grafów o tej gęstości, macierz sąsiedztwa może być preferowaną strukturą danych.

4 Wnioski

1. **Wybór struktury danych zależy od gęstości grafu**

   Dla grafów o niskiej gęstości (np. 25%), różnice w wydajności między listą sąsiedztwa a macierzą sąsiedztwa są zazwyczaj nieznaczne. Natomiast dla grafów o wyższej gęstości (np. 75% lub pełnych), wybór struktury danych może mieć istotny wpływ na czas wykonania algorytmu.

2. **Lista sąsiedztwa jest efektywna dla grafów o niskiej gęstości**

   W przypadku grafów o niskiej gęstości, lista sąsiedztwa może być preferowaną strukturą danych ze względu na oszczędność pamięci i możliwość szybkiego dostępu do sąsiadów. Dzięki temu można uzyskać podobne czasy wykonania algorytmu Dijkstry jak przy użyciu macierzy sąsiedztwa.

3. **Macierz sąsiedztwa sprawdza się dla grafów o dużej gęstości**

   Dla grafów o wyższej gęstości, zwłaszcza dla grafów pełnych, macierz sąsiedztwa może być bardziej efektywną strukturą danych. Pomimo większego zużycia pamięci, umożliwia ona szybkie sprawdzanie istnienia krawędzi między wierzchołkami oraz aktualizację odległości w algorytmie Dijkstry.

4. **Złożoność czasowa rośnie wraz z gęstością i liczbą wierzchołków**

   Zarówno dla listy sąsiedztwa, jak i macierzy sąsiedztwa, złożoność czasowa algorytmu Dijkstry rośnie wraz z zwiększającą się liczbą wierzchołków oraz gęstością grafu. W przypadku grafów pełnych, różnice w czasie wykonania między obiema strukturami danych mogą być znaczące.

5. **Analiza wydajnościowa jest istotna przy wyborze struktury danych**

   Przed zastosowaniem algorytmu Dijkstry w konkretnym problemie, warto przeprowadzić analizę wydajnościową dla różnych reprezentacji grafu, aby wybrać najbardziej odpowiednią strukturę danych z uwagi na specyfikę problemu i oczekiwane wymagania dotyczące wydajności. W tym kontekście, uwzględnienie zarówno teoretycznych założeń dotyczących złożoności obliczeniowej, jak i empirycznych wyników pomiarów czasu wykonania algorytmu, może pomóc w dokonaniu właściwego wyboru.

Literatura

[1] `https://pl.wikipedia.org/wiki/Algorytm_Dijkstry`

[2] `https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/`

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