Sprawozdanie Struktury danych

Gabriela Czernecka

1. Wstęp

W sprawozdaniu przeprowadzono porównanie dwóch implementacji algorytmu Prima służącego do znajdowania minimalnego drzewa rozpinającego w grafach nieskierowanych. Porównane zostały dwie implementacje: z użyciem kolejki priorytetowej oraz z użyciem listy. Celem porównania było zbadanie efektywności i różnic w czasie wykonania obu implementacji.

Algorytm Prima opiera się na strategii zachłannej, polegającej na rozbudowywaniu minimalnego drzewa rozpinającego poprzez wybieranie krawędzi o najmniejszej wadze spośród dostępnych. Działa on na zasadzie stopniowego rozbudowywania drzewa, dodając do niego kolejne wierzchołki, aż do momentu, gdy zawiera ono wszystkie wierzchołki grafu.

W implementacji z użyciem kolejki priorytetowej, każdy wierzchołek jest reprezentowany jako element kolejki, a krawędzie są dodawane do kolejki w odpowiedniej kolejności, zgodnie z ich wagami. W każdym kroku algorytm pobiera z kolejki krawędź o najmniejszej wadze spośród dostępnych, sprawdza czy nie tworzy ona cyklu, a następnie dodaje odpowiednie krawędzie do kolejki.

W implementacji z użyciem listy, dla każdego wierzchołka utworzona jest lista jego sąsiadów, posortowana rosnąco według wag krawędzi. Algorytm iteruje po kolejnych wierzchołkach, dodając do drzewa rozpinającego krawędzie o najmniejszej wadze, które nie tworzą cykli.

Celem przeprowadzonych eksperymentów było zbadanie, które z tych dwóch implementacji jest bardziej efektywna pod względem czasu wykonania. Badania przeprowadzono dla różnych rozmiarów grafów, reprezentowanych przez liczbę wierzchołków, oraz dla różnych gęstości grafów

2. Fragmenty kodu

Lista

```
struct ListNode {
    Edge data;
    ListNode* next;

    ListNode(const Edge& e) : data(e), next(nullptr) {}
};

struct List {
    ListNode* head;
    ListNode* tail;

    List() : head(nullptr), tail(nullptr) {}

    bool isEmpty() {
        return head == nullptr;
    }

    void insert(const Edge& e) {
        ListNode* newNode = new ListNode(e);
        if (isEmpty()) {
            head = newNode;
            tail = newNode;
        }
        else if (e.weight < head->data.weight) {
            newNode->next = head;
```

```
            head = newNode;
        }
        else {
            ListNode* curr = head;
            while (curr->next != nullptr && curr->next->data.weight < e.weight) {
                curr = curr->next;
            }

            newNode->next = curr->next;
            curr->next = newNode;

            if (newNode->next == nullptr) {
                tail = newNode;
            }
        }
    }
```

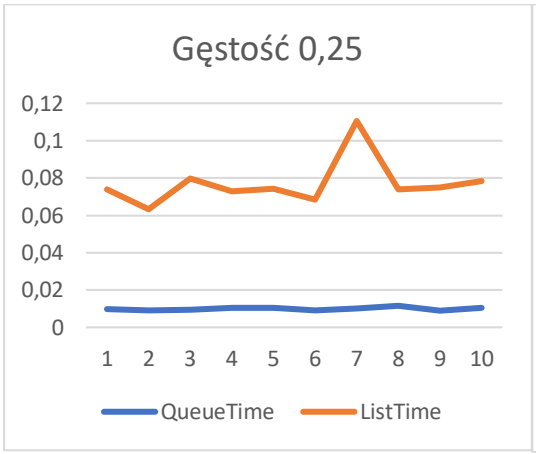
Struktura graf

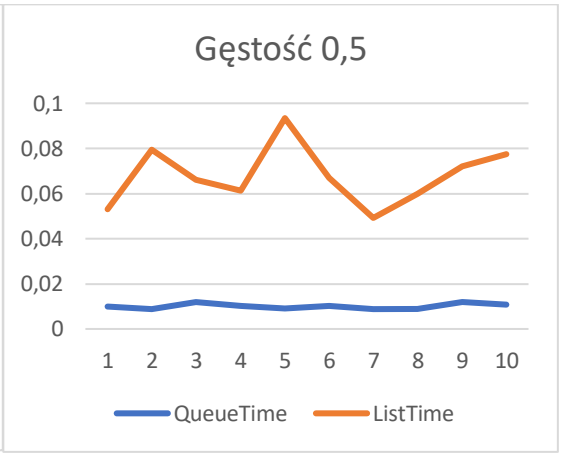
```
struct Graph {
    int numVertices;
    std::vector<List> adjList;

    Graph(int vertices) : numVertices(vertices), adjList(vertices) {}

    void addEdge(int src, int dest, int weight) {
        Edge newEdge(src, dest, weight);
        adjList[src].insert(newEdge);

        newEdge = Edge(dest, src, weight);
        adjList[dest].insert(newEdge);
    }

    std::vector<Edge> primMSTQueue() {
        std::vector<bool> visited(numVertices, false);
        std::vector<Edge> minimumSpanningTree;
        std::priority_queue<Edge*, std::vector<Edge*>, std::greater<Edge*>> pq;

        visited[0] = true;

        for (ListNode* node = adjList[0].head; node != nullptr; node = node->next) {
            pq.push(&node->data);
        }

        while (!pq.empty()) {
            Edge* current = pq.top();
            pq.pop();

            int nextWeight = current->weight;
            int nextVertex = current->dest;

            if (!visited[nextVertex]) {
                visited[nextVertex] = true;
                minimumSpanningTree.push_back(*current);

                for (ListNode* node = adjList[nextVertex].head; node != nullptr; node
= node->next) {
                    if (!visited[node->data.dest]) {
                        pq.push(&node->data);
                    }
                }
            }
        }
```

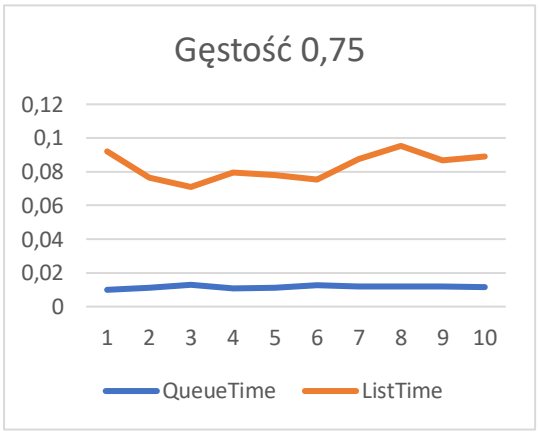
```
    return minimumSpanningTree;
}
```

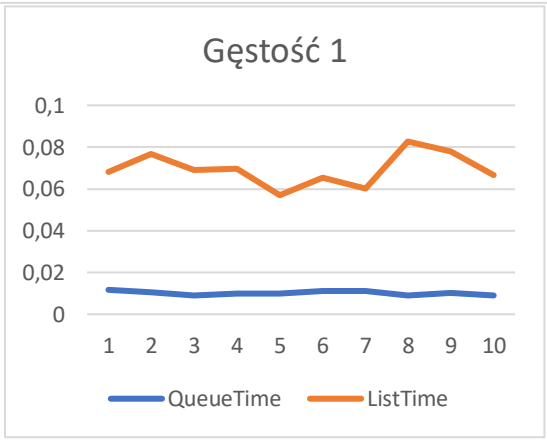
3. Wyniki

Dla 50 wierzchołków

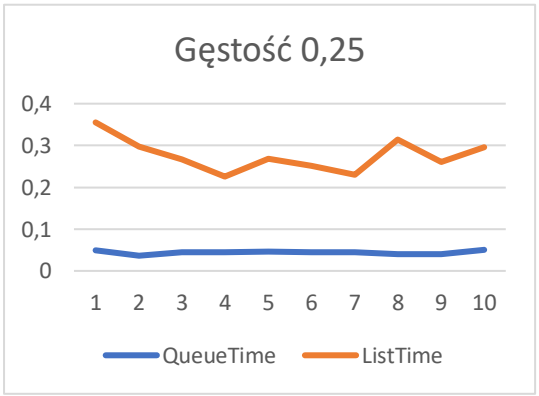


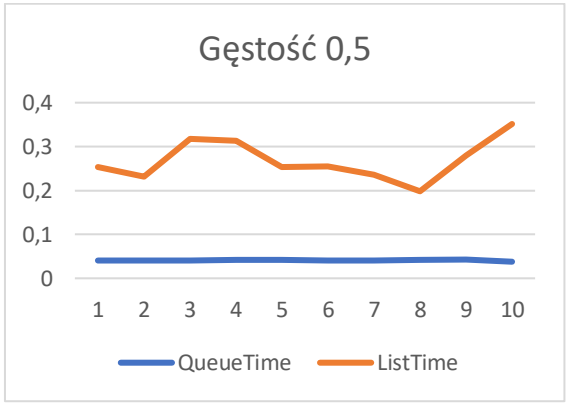


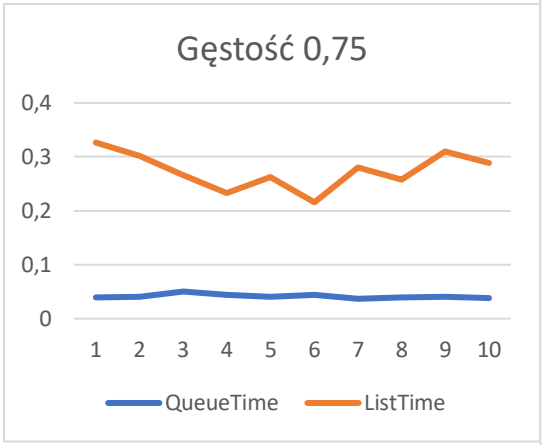


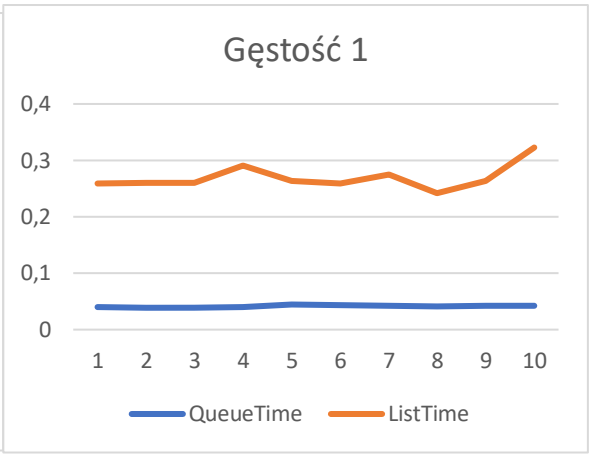


Dla 100 wierzchołków

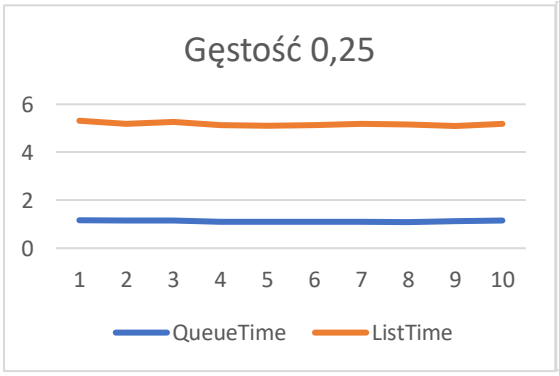


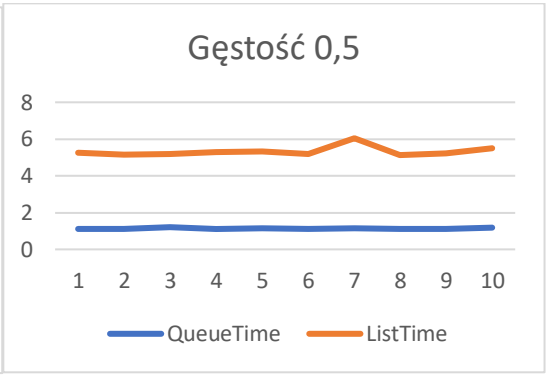


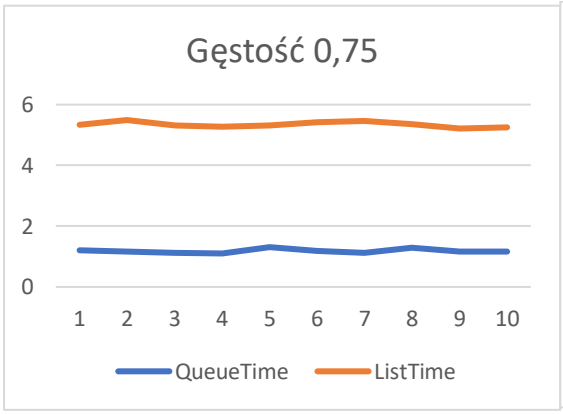


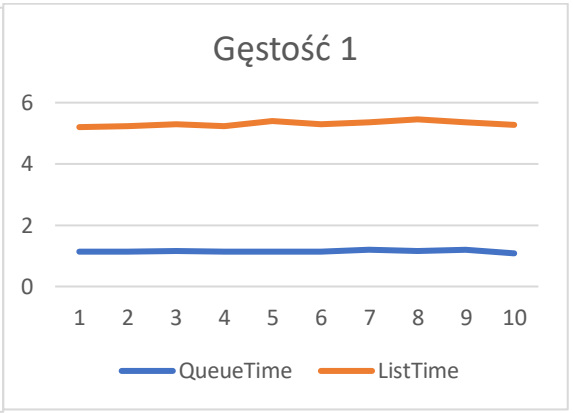


Dla 500 wierzchołków

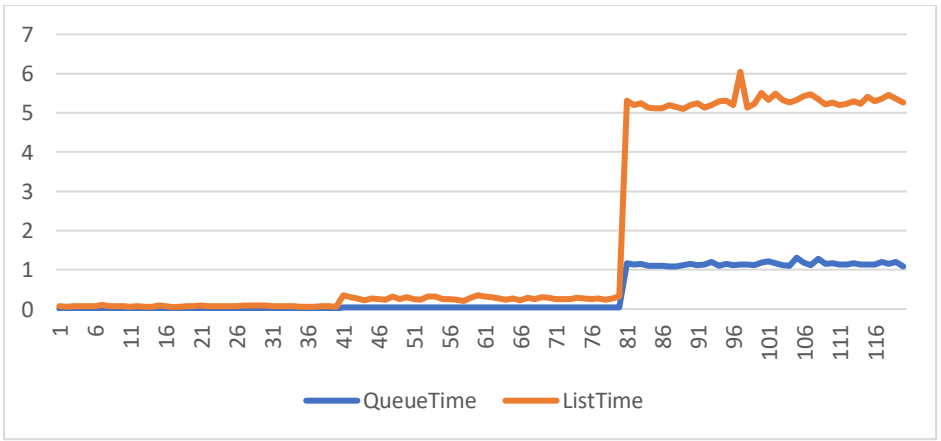








Porównanie wszystkich wyników



Obliczony średni czas w sekundach

		średni czas	
Ilość wierzchołków	Gęstość	Prim na liście	Prim na kolejce
50	0,25	0,009865278	0,07703612
	0,5	0,0100431	0,06791346
	0,75	0,01152823	0,08313787
	1	0,01046253	0,06937213
100	0,25	0,03999343	0,2765123
	0,5	0,04226215	0,2688001
	0,75	0,04111968	0,2744526
	1	0,04088401	0,269511
500	0,25	1,01141161	5,176854
	0,5	1,135688	5,329683
	0,75	1,185121	5,343334
	1	1,158494	5,308041

4. Wnioski

Na podstawie przeprowadzonych eksperymentów i analizy wyników można wyciągnąć następujące wnioski:

Efektywność czasowa: Implementacja algorytmu Prima z użyciem kolejki priorytetowej wykazuje znacznie lepszą efektywność czasową w porównaniu do implementacji z użyciem listy. Dzięki wykorzystaniu struktury danych, która automatycznie sortuje krawędzie według ich wag, czas działania algorytmu na kolejce priorytetowej jest znacznie krótszy. Wybór krawędzi o najmniejszej wadze jest bardziej efektywny, co przyspiesza proces budowy minimalnego drzewa rozpinającego.

Złożoność pamięciowa: Implementacja algorytmu Prima na kolejce priorytetowej wymaga większej ilości pamięci w porównaniu do implementacji na liście. Wynika to z konieczności przechowywania dodatkowych struktur danych, takich jak kolejka priorytetowa. Implementacja na liście zajmuje mniej pamięci, ponieważ wymaga tylko przechowywania list sąsiedztwa dla poszczególnych wierzchołków.

Wygoda implementacji: Implementacja algorytmu Prima na kolejce priorytetowej jest bardziej złożona pod względem implementacyjnym w porównaniu do implementacji na liście. Wymaga użycia dodatkowej struktury danych oraz dostosowania operacji dodawania i usuwania krawędzi do kolejki priorytetowej. Implementacja na liście jest prostsza, polega na utworzeniu list sąsiedztwa dla każdego wierzchołka i iteracji po nich.

Skalowalność: Implementacja algorytmu Prima na kolejce priorytetowej jest bardziej skalowalna i lepiej radzi sobie z większymi grafami. Dzięki optymalizacji wyboru krawędzi o najmniejszej wadze, czas działania algorytmu na kolejce priorytetowej rośnie wolniej wraz z rosnącą liczbą wierzchołków. Implementacja na liście może mieć większy wpływ na wydajność dla większych grafów ze względu na potrzebę iteracji po listach sąsiedztwa.

Podsumowując, implementacja algorytmu Prima z użyciem kolejki priorytetowej jest zdecydowanie bardziej efektywna pod względem czasu wykonania. Mimo większego zużycia pamięci i większej złożoności implementacyjnej, korzyści wynikające z optymalizacji wyboru krawędzi przeważają. Jednakże, dla mniejszych grafów i gdy złożoność pamięciowa ma większe znaczenie, implementacja

na liście może być wystarczająco efektywna. Ostateczny wybór zależy od specyfiki problemu i oczekiwanych wymagań dotyczących czasu wykonania i zasobów pamięciowych.

5. Bibliografia

https://en.wikipedia.org/wiki/Prim%27s_algorithm

https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

https://www.programiz.com/dsa/prim-algorithm

https://pl.wikipedia.org/wiki/Algorytm_Prima

https://eduinf.waw.pl/inf/alg/001_search/0141.php

http://www.algorytm.org/algorytmy-grafowe/algorytm-prima.html