SPRAWOZDANIE

PROJEKT 2

PROJEKTOWANIE ALGORYTMÓW I METOD SZTUCZNEJ INTELIGENCJI

AUTOR: EMILIA SZYMAŃSKA

PROWADZĄCY: DR INŻ. KRZYSZTOF HALAWA

GRUPA: WTOREK 13:15 – 15:00

1. Wstęp

Celem projektu jest implementacja algorytmu Dijkstry z grafami w postaci macierzy i listy sąsiedztwa, następnie przetestowanie algorytmu dla obu reprezentacji i zbadanie ich efektywności. Do przeprowadzenia testów efektywności wygenerowałam grafy, których wagi są w zakresie od 1 do 100. Tablic jest 100 każdej gęstości dla każdej liczby wierzchołków. Testowane liczby wierzchołków są następujące: 10, 50, 100, 500 i 1000. Gęstości grafów są równe 25%, 50%, 75% i 100% (graf pełny).

2. Omówienie algorytmu

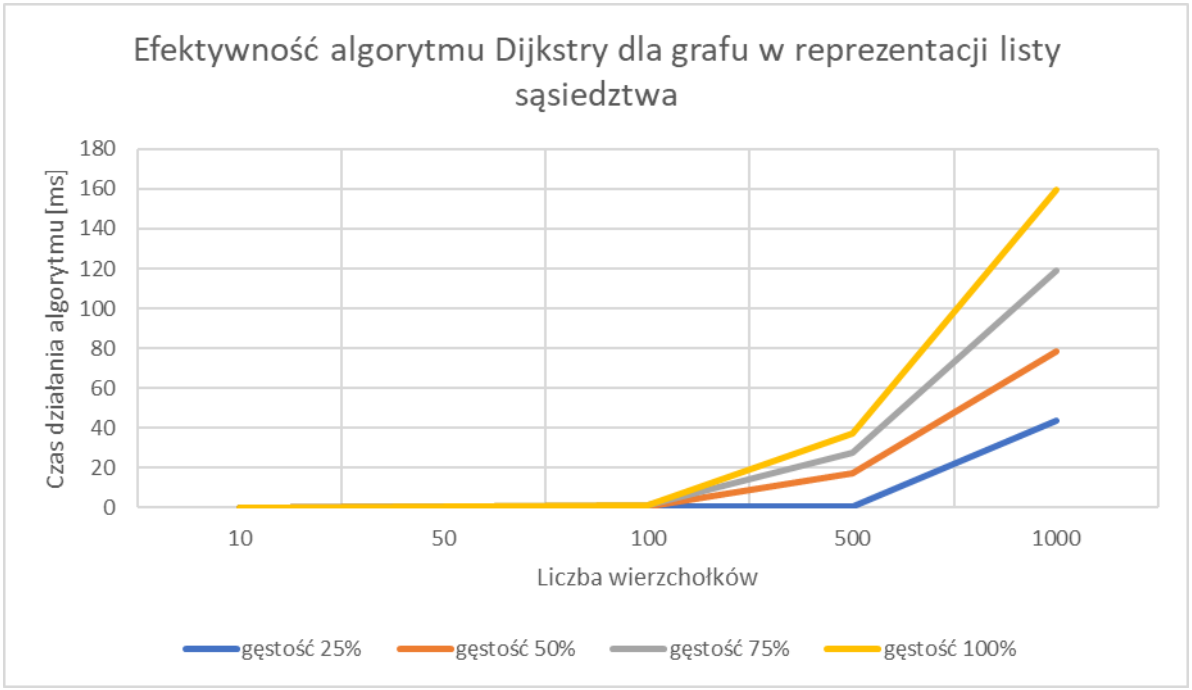
Algorytm Dijkstry zaczynam od utworzenia struktury, do której będę zapisywała wyniki (struktura ta ma pola previous oznaczające tablicę z poprzednim wierzchołkiem z ścieżki oraz distance będące tablicą kosztów ścieżek od wierzchołka startowego). Ustawiam wszystkie elementy obu tablic na wartość nieskończoność (u mnie w kodzie jest to wartość -1). Ustawiam koszt ścieżki z punktu startowego do punktu startowego na 0. Następnie tworzę kopiec par (dystans, wierzchołek) i wrzucam na niego parę z wierzchołkiem startowym. Dopóki kopiec nie jest pusty sprawdzam parę z korzenia kopca i ją zdejmuję z niego. Jeśli dystans ze zdjętego elementu jest większy niż ten w ostatecznej strukturze wynikowej, to przechodzę do następnego obiegu pętli (oznacza to, że lepszy koszt drogi jest w wyniku i nasza wartość jest gorszą alternatywą). W przeciwnym razie pobieramy listę przylegających do wierzchołka krawędzi i sąsiednich wierzchołków i dla każdego połączenia za pomocą swojej implementacji iteratora – jeśli dystans w strukturze wynikowej ma wartość nieskończoność lub jest większy niż znaleziona alternatywa – do struktury wynikowej dodajemy nowy koszt ścieżki, nowy wierzchołek poprzedni i wrzucamy nową parę na kopiec. Na kopiec wrzucam wartości z odwróconym znakiem, gdyż w korzeniu kopca znajduje się wartość największa, więc jeśli chcemy mieć na wierzchu najmniejszy koszt, to wystarczy dodać ,,-" w dystansie.

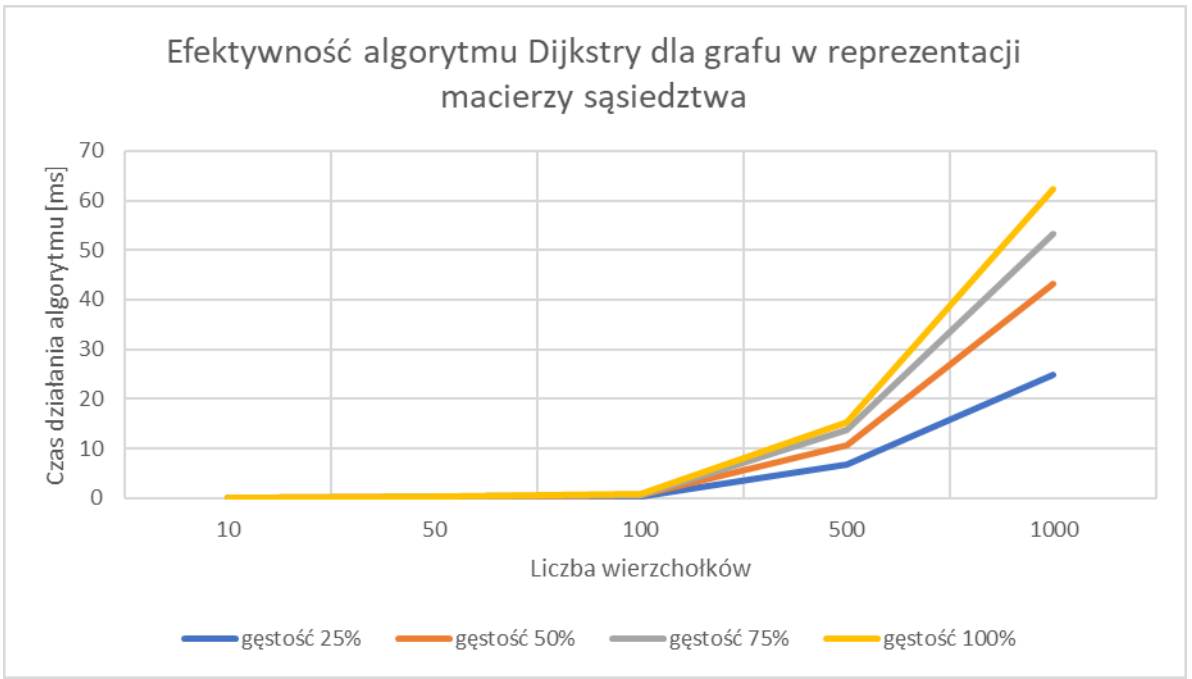
3. Wyniki eksperymentu

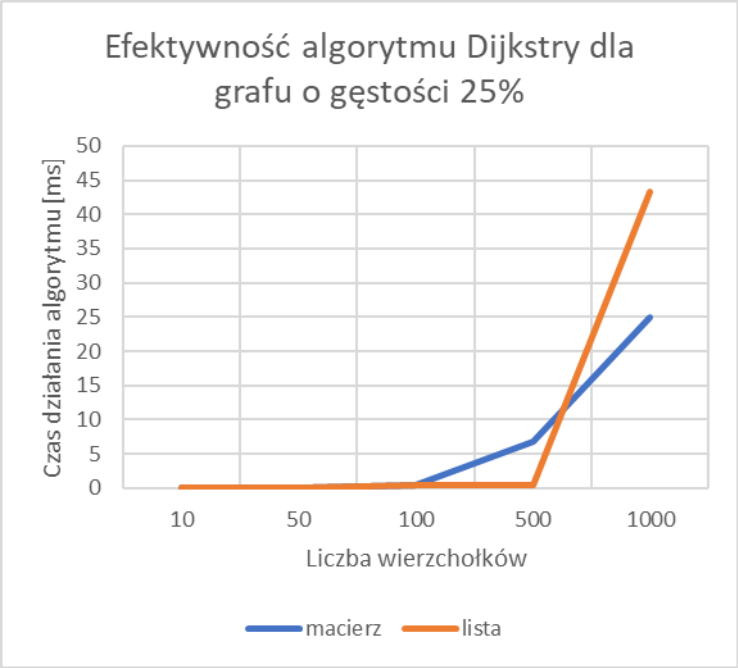
Wyniki wywołań algorytmu Dijkstry dla danych reprezentacji grafów znajdują się w tabelach oraz na wykresach. Tabele zawierają uśrednione wyniki czasu działania algorytmów w milisekundach dla konkretnych przypadków (kolumny odpowiadają liczbie wierzchołków grafu, a wiersze gęstościom). Wykresy natomiast przedstawiają porównania pomiędzy dwoma algorytmami przy tej samej gęstości, a także dla każdej gęstości porównania między czasem dla obu reprezentacji.

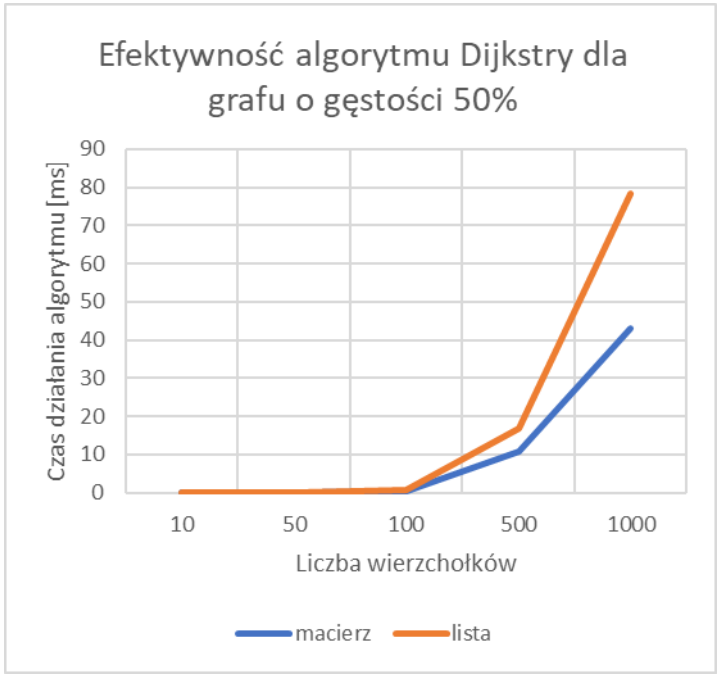
LISTA	10	50	100	500	1000
25%	0,00840	0,12262	0,36919	0,369190	43,34926
50%	0,01166	0,17362	0,59340	16,90508	78,49369
75%	0,01716	0,23549	0,94348	27,24198	119,1365
100%	0,02046	0,30697	1,24277	37,40068	159,7991

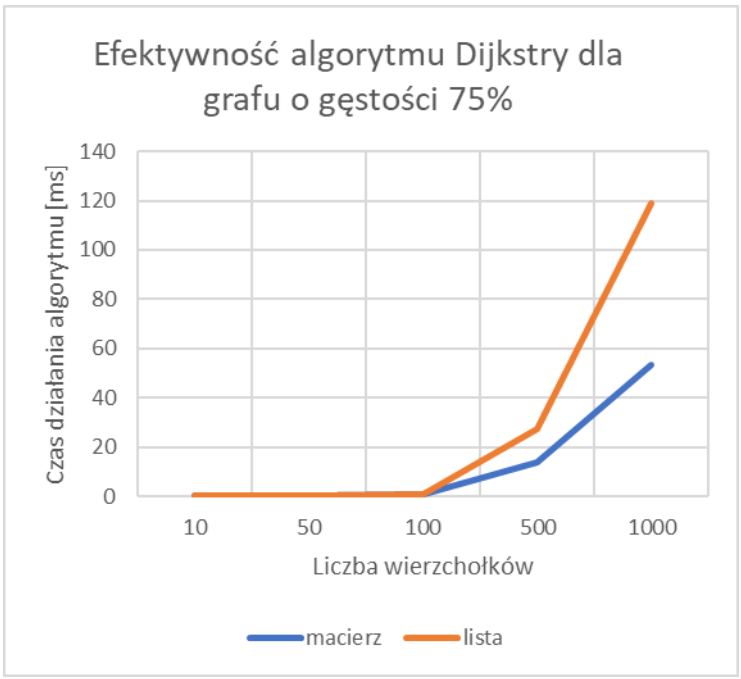
MACIERZ	10	50	100	500	1000
25%	0,00704	0,10200	0,32692	6,70353	24,94710
50%	0,01203	0,16207	0,51249	10,67722	43,21932
75%	0,01348	0,19449	0,69200	13,64639	53,19442
100%	0,01482	0,20118	0,83302	15,41193	62,28696

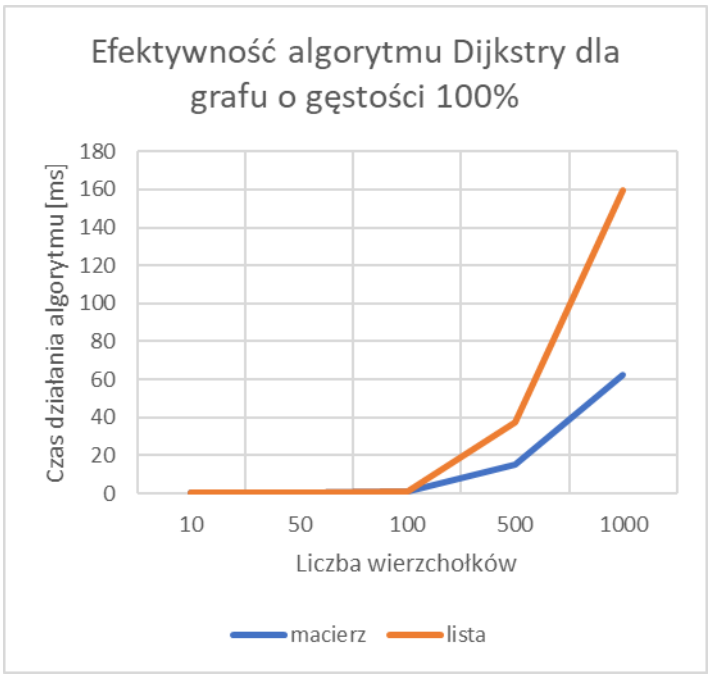

Efektywność algorytmu Dijkstry dla grafu w reprezentacji listy sąsiedztwa
Czas działania algorytmu [ms]
180
160
140
120
100
80
60
40
20
0
10
50
100
500
1000
Liczba wierzchołków
gęstość 25%
gęstość 50%
gęstość 75%
gęstość 100%


Efektywność algorytmu Dijkstry dla grafu w reprezentacji macierzy sąsiedztwa
Czas działania algorytmu [ms]
70
60
50
40
30
20
10
0
10
50
100
500
1000
Liczba wierzchołków
gęstość 25%
gęstość 50%
gęstość 75%
gęstość 100%









4. Wnioski

Wraz ze wzrostem gęstości i liczbą wierzchołków algorytm działa wolniej dla obu reprezentacji – szczególny przeskok widać pomiędzy liczbą wierzchołków 100 a 500. Jedynie dla gęstości 25% przy liście widać ten przeskok dopiero przy wartościach 500 i 1000. Dłuższy czas działania algorytmu dla listy sąsiedztwa może wynikać z własnej implementacji listy oraz grafu opartego na niej. Macierz może i jest tutaj wydajniejszą reprezentacją, lecz zajmuje więcej miejsca w pamięci (dla 1000 wierzchołków trzeba zaalokować dwie tablice 1000x1000). Lista ma to do siebie, że elementy są w różnych miejscach pamięci, gdzie w macierzy elementy są koło siebie. Z tego powodu graf z listą sąsiedztwa może dłużej działać, bo przez to porozrzucanie elementów po pamięci trudniej jest pamięć alokować i dealokować (trzeba elementy znajdować).

5. Opis kodu

5.1. Algorytm Dijkstry

Na implementację algorytmu składają się dwa pliki – Dijkstra.cpp i Dijkstra.hh. Działanie algorytmu zostało dokładnie omówione w punkcie 2., tutaj jest jego odzwierciedlenie.

5.2. Graf – macierz sąsiedztwa

Na kod składają się pliki GraphMatrix.cpp i GraphMatrix.hh. Klasa GraphMatrix zawiera w sobie dwie tablice (jedna informuje o sąsiedztwie, druga o wartości krawędzi łączących odpowiednie wierzchołki) oraz informację o ilości wierzchołków. Są zaimplementowane metody sprawdzania sąsiedztwa, wstawiania krawędzi, zwracania sąsiadów, zwracania ilości wierzchołków, konstruktor i dekonstruktor.

5.3. Graf – lista sąsiedztwa

Implementacja tego typu grafu znajduje się w plikach GraphList.cpp i GraphList.hh. Klasa GraphList zawiera w sobie listę par (wierzchołek, krawędź) oraz informację o ilości wierzchołków. Zaimplementowane są te same metody jak w przypadku macierzy sąsiedztwa (tylko przełożone na poruszanie się po liście, nie macierzy).

5.4. Funkcja main

Na początku wczytuję liczbę krawędzi, liczbę wierzchołków i wierzchołek startowy. Tworzę graf (zgodnie z wybraną reprezentacją), a następnie wczytuję kolejne krawędzie (wierzchołek1, wierzchołek2, waga krawędzi). Wywołuję algorytm Dijkstry dla grafu i wierzchołka startowego, przypisuję wyniki do zmiennej. W kolejnym kroku tworzę stos i przechodzę się po wynikach dla każdego wierzchołka, by odtworzyć ścieżkę, wrzucając poprzedniki dla pozostałych wierzchołków na stos, a następnie z niego zdejmując elementy. Ostatecznie program wypisuje dla każdego wierzchołka najlepszą (najkrótszą) ścieżkę od wierzchołka startowego oraz jej koszt.

5.5. Dodatkowe struktury i klasy

Z dodatkowych elementów mamy własną implementację listy (list.hh, list.cpp), w niej implementację własnego iteratora, const iteratora oraz strukturę elementu z listy, parę (Pair.hh, Pair.cpp), kopiec (heap.hh, heap.cpp) oraz stos (stack.hh, stack.cpp).

6. Literatura:

https://eduinf.waw.pl/inf/alg/001_search/0138.php
https://pl.wikipedia.org/wiki/Algorytm_Dijkstry
,,Wprowadzenie do algorytmów” - T. Cormen, C. Leiserson, R. Rivest, C. Stein