Politechnika Wrocławska
Wydział Elektroniki
Kierunek: AiR

Rok akademicki
2019/2020
semestr letni

Sprawozdanie z projektu 2

Kurs:	Projektowanie algorytmów i metody sztucznej inteligencji
Prowadzący:	Dr inż. Łukasz Jeleń
Wykonał:	Taras Radchenko (nr indeksu 248021)
Termin zajęć:	czwartek 9:15

Wrocław
30.04.2020

Wprowadzenie

Graf to struktura danych, która nie jest tak oczywista, jak np. tablica czy kolejka priorytetowa, aczkolwiek bardzo ważna. Grafy są nie do zastąpienia w projektowaniu i wykorzystaniu jakichkolwiek sieci: mogą to być sieci komputerowe, drogowe lub nawet społeczne. Jeżeli potrafimy przetłumaczyć problem na język grafów, mamy od razu wiele narzędzi do jego głębszej analizy i/lub rozwiązania.

Założenia

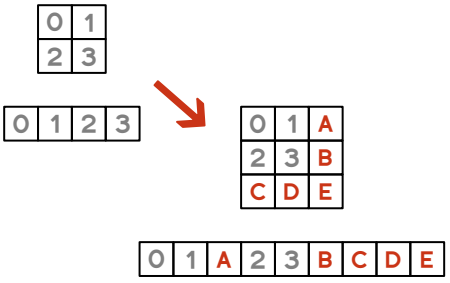
W tym projekcie omawiam grafy spójne, nieskierowane, bez cyklów, krawędzi ujemnych i równoległych.

Badane implementacje grafów

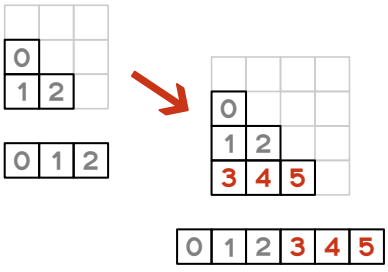
Graf w postaci macierzy sąsiedztwa

Wierzchołki i krawędzie są przechowywane w tablicach. Połączenia między nimi są przechowywane w macierzy, we której indeksom odpowiadają wierzchołki, a w komórce o indeksie i, j jest przechowywany wskaźnik na krawędź, która łączy wierzchołki v_i i v_j. Krawędź również zawiera wskaźniki na wierzchołki końcowe.

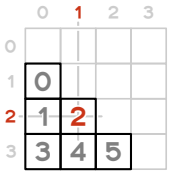
O ile założyłem, że grafy są nieskierowane i bez cyklów, w celu uniknięcia redundancji danych w mojej implementacji jest wykorzystana połowa macierzy. To również upraszcza dodanie wierzchołków, gdyż w przypadku stosowania macierzy w postaci tablicy potrzebny jest trochę bardziej zaawansowany algorytm, który by ją zarządzał, np. odpowiednio przesuwał komórki przy dodaniu nowego wierzchołka (rys. 1). Natomiast w mojej implementacji wystarczy tylko dodać nowe komórki na koniec tablicy (rys. 2). Operacje wyszukiwania krawędzi między dwoma wierzchołkami oraz krawędzi incydentnych są zilustrowane na rys. 3 i rys. 4.



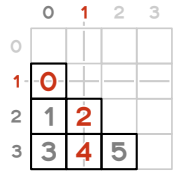
Rys. 1



Rys. 2



Rys. 3



Rys. 4

Graf w postaci listy sąsiedztwa

Wierzchołki i krawędzie są przechowywane w tablicach. W oddzielnej tablicy znajdują się listy krawędzi tak, że w i-tej komórce jest lista krawędzi incydentnych do i-tego wierzchołka. Krawędź również zawiera wskaźniki na wierzchołki końcowe.

Badane algorytmy

Algorytm Dijkstry

Algorytm Dijkstry konsekwentnie wybiera wierzchołek o najmniejszej wadze (na początku wierzchołek startowy ma wagę 0, pozostałe — ∞), łączy go z jego poprzednikiem i aktualizuje wagi wierzchołków sąsiednich.

Algorytm Bellmana-Forda

W odróżnieniu od algorytmu Dijkstry, algorytm Bellmana-Forda jednocześnie aktualizuje wagi wszystkich wierzchołków i robi to $|E| - 1$ razy, gdzie $|E|$ — liczba krawędzi. Wierzchołek początkowy ma wagę 0, pozostałe — ∞.

Przebieg eksperymentu

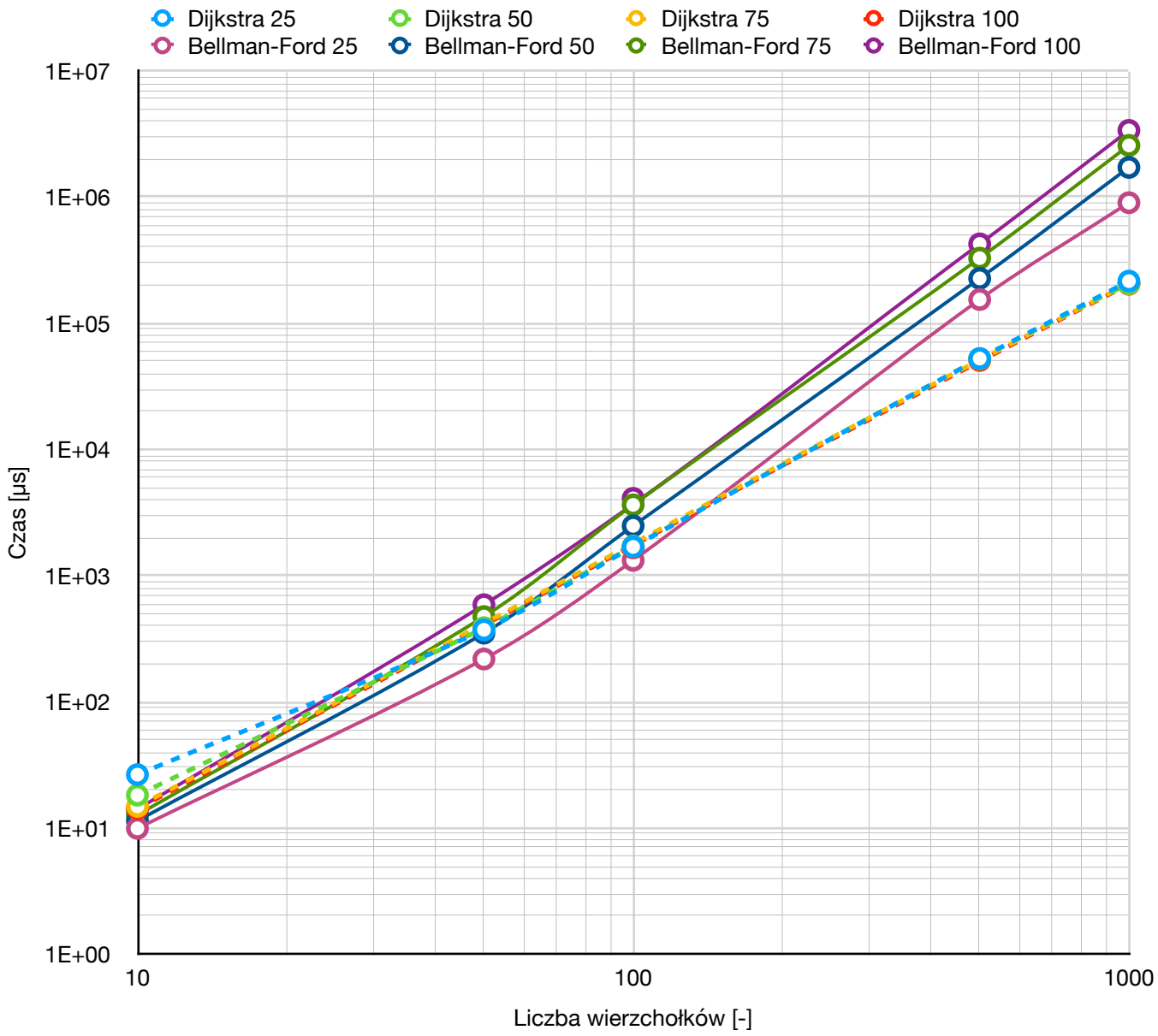
Z grubsza sposób prowadzenia testu jest przedstawiony w następującym pseudokodzie:

```
Algorytm testujAlgorytm(algorytm, graf, wierzchołekPoczątkowy)
    Opis: mierzy czas działania algorytmu algorytm na grafie graf
    Wyjście: zmierzony czas w nanosekundach

    początek ← zegar_systemowy.czas()
    algorytm(graf, wierzchołekPoczątkowy)
    koniec ← zegar_systemowy.czas()

    return (koniec - początek).nanosekundy()


Algorytm testuj(plikCSV, katalogDanych)
    Opis: mierzy czas działania każdego algorytmu na grafu każdej postaci
          na danych z katalogu katalogDanych i zapisuje wyniki do
          pustego pliku plikCSV
    Wyjście: nic

    plikCSV.zapiszLinię(
        ”Liczba wierzchołków, Gęstość, Czas M_D, Czas M_BF, Czas L_D, Czas L_BF”)

    for dana: katalogDanych do
        plikCSV.zapiszWPole(dana.liczbaWierzchołków())
        plikCSV.zapiszWPole(dana.gęstość())

        plikCSV.zapiszWPole(testujAlgorytm(Dijkstra,    dana.macierz(), dana.wp()))
        plikCSV.zapiszWPole(testujAlgorytm(BellmanFord, dana.macierz(), dana.wp()))
        plikCSV.zapiszWPole(testujAlgorytm(Dijkstra,    dana.lista(),   dana.wp()))
        plikCSV.zapiszWPole(testujAlgorytm(BellmanFord, dana.lista(),   dana.wp()))

        plikCSV.zapiszLinię()
```

Do pomiarów czasu jest wykorzystywany `std::chrono::steady_clock`, który zapewnia największą dokładność.

Wyniki

Graf w postaci macierzy sąsiedztwa

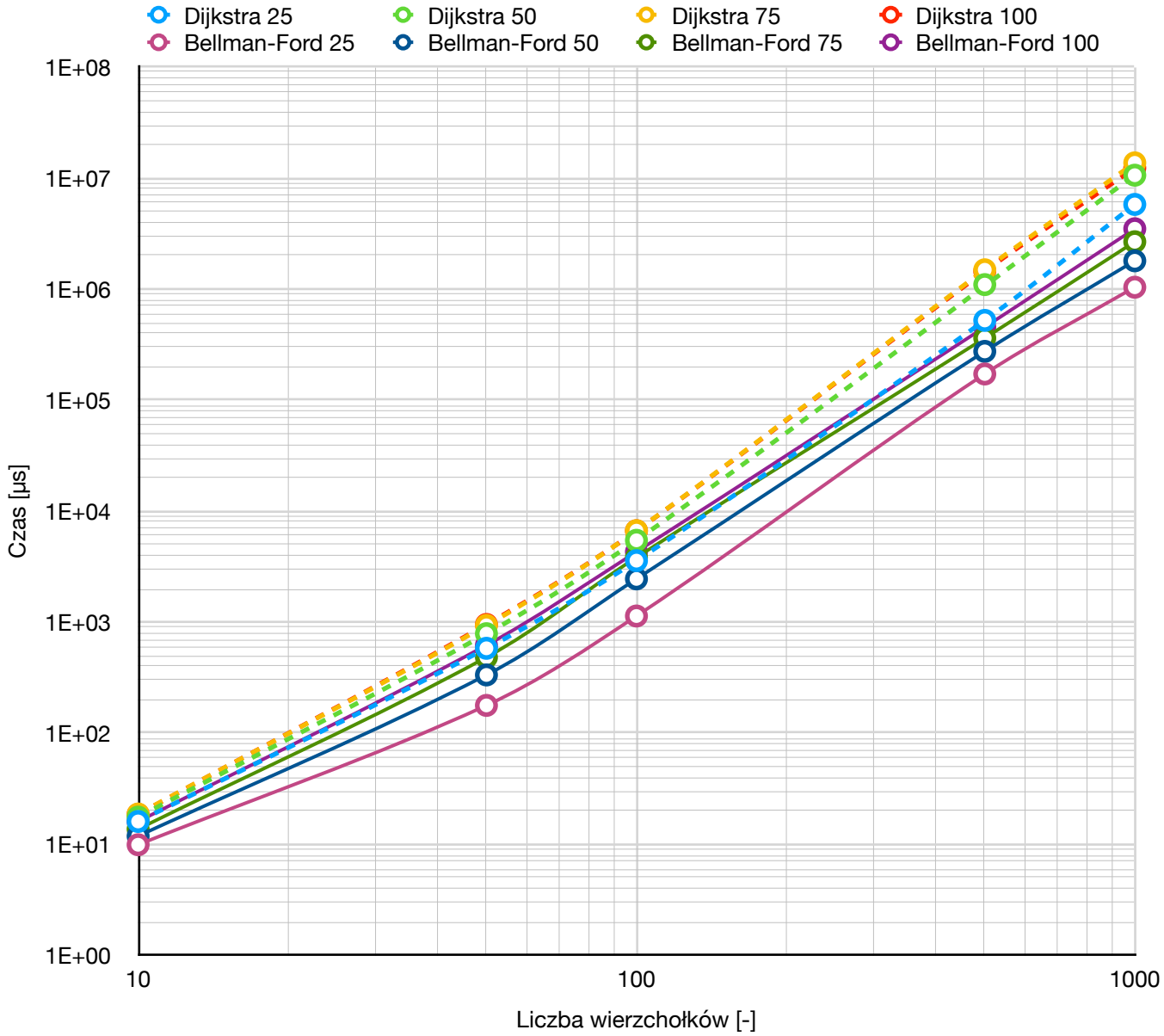


Jak widać z wykresów, algorytm Dijkstry (linia przerywana) działa lepiej w przypadku większych grafów i czas działania zależy od gęstości tylko dla małej liczby wierzchołków. Natomiast algorytm Bellmana-Forda działa tym szybciej, im mniejsza jest gęstość grafu. Jest tak dlatego, że w tym algorytmie iterowana jest tablica krawędzi.

Wyniki testu dla grafu w postaci macierzy sąsiedztwa

Algorytm	Dijkstra	Dijkstra	Dijkstra	Dijkstra	Bellman-Ford	Bellman-Ford	Bellman-Ford	Bellman-Ford
Gęstość	**25**	**50**	**75**	**100**	**25**	**50**	**75**	**100**
10	26,473	18,211	14,844	14,264	9,935	11,421	12,705	14,133
50	372,368	385,669	386,716	383,175	218,620	350,640	474,335	588,745
100	1692,229	1712,287	1710,126	1663,319	1323,827	2483,230	3643,344	4102,164
500	52626,627	52469,674	52334,888	50870,474	154646,884	227314,497	329015,060	426488,875
1000	216214,099	207238,063	206883,895	204398,860	904891,408	1723778,649	2575316,217	3387298,079

Graf w postaci listy sąsiedztwa



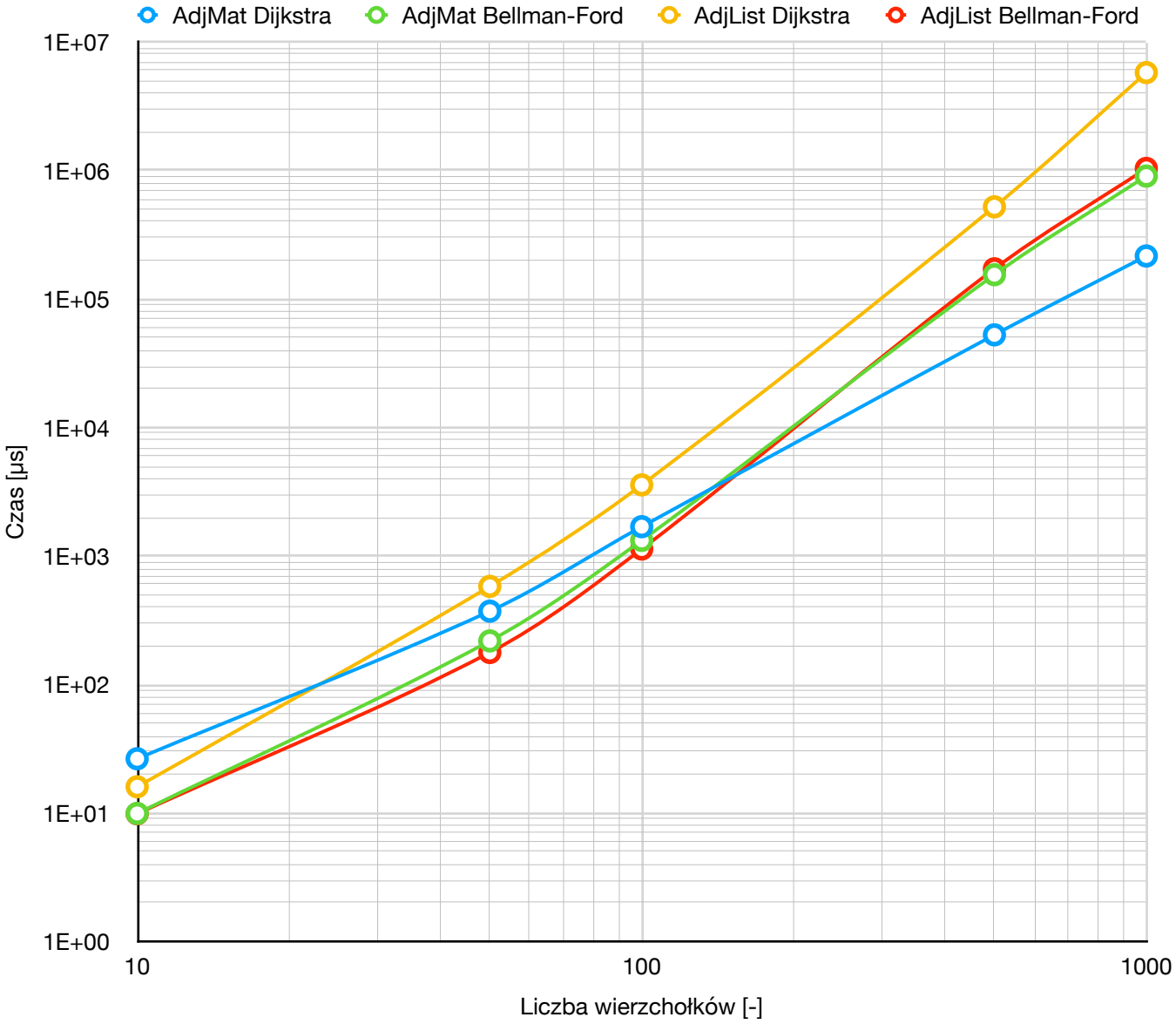
Na tej implementacji grafu algorytm Dijkstry działa gorzej w porównaniu do alg. Bellmana-Forda. Rownież ów algorytm już nie jest tak nieczuły na gęstość, ale wciąż rozrzut średnich wartości czasu działania jest mniejszy w porównaniu do BF.

Wyniki testu dla grafu w postaci listy sąsiedztwa [µs]

Algorytm	Dijkstra	Dijkstra	Dijkstra	Dijkstra	Bellman-Ford	Bellman-Ford	Bellman-Ford	Bellman-Ford
Gęstość	**25**	**50**	**75**	**100**	**25**	**50**	**75**	**100**
10	16,019	17,518	18,684	18,534	9,879	11,817	13,671	15,998
50	580,927	786,691	935,340	952,686	177,770	334,344	482,290	619,337
100	3578,444	5455,454	6662,621	6649,851	1134,943	2450,932	3824,157	4299,667
500	521706,864	1096055,514	1490379,658	1444562,016	171745,678	274445,815	361570,191	453732,625
1000	5782102,216	10598673,188	13687208,830	12320412,008	1038199,656	1795874,358	2666293,810	3489952,940

Grafy o gęstości 25%

Tu i dalej AdjMat oznacza wyniki dla grafu w postaci macierzy sąsiedztwa, AdjList — dla grafu w postaci listy sąsiedztwa.

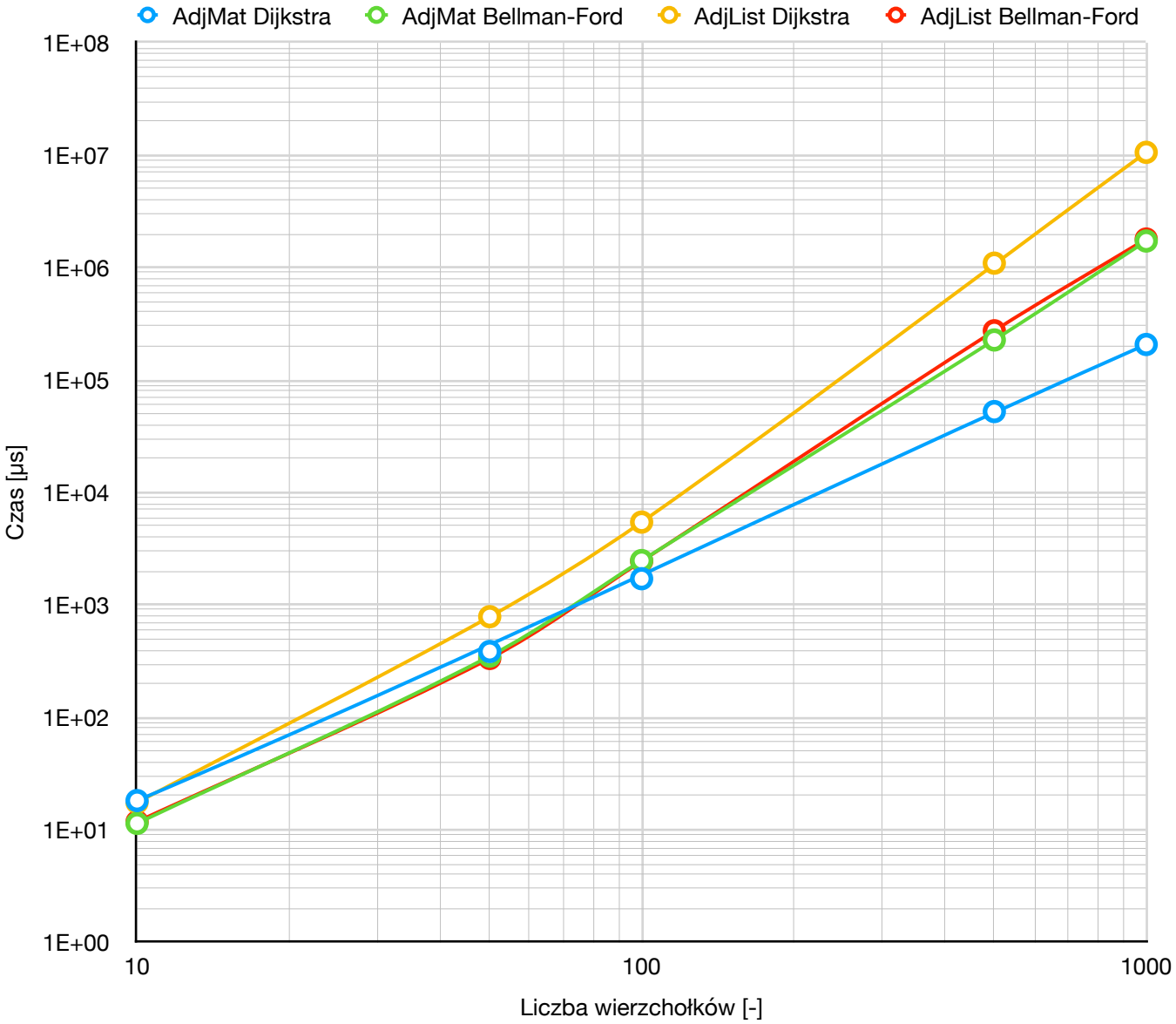


Z wykresu powyżej widać ewidentnie, że algorytm Bellmana-Forda nie jest zbyt czuły na postać grafu. Prawdopodobnie to dlatego, że w mojej implementacji używane są tylko metody `edges()` i `vertices()`, przy czym druga jest odziedziczona od `IVGraph`, który jest wspólnym rodzicem obu grafów. Natomiast w implementacji alg. Dijkstry jest często używana metoda `edgeBetween()`, która działa w czasie $O(1)$ w przypadku macierzy i w $O(min(deg(v), deg(w)))$ w przypadku listy.

Wyniki testu dla grafów o gęstości 25% [µs]

Reprezentacja	Macierz sąsiedztwa	Macierz sąsiedztwa	Lista sąsiedztwa	Lista sąsiedztwa
Algorytm	**Dijkstra**	**Bellman-Ford**	**Dijkstra**	**Bellman-Ford**
10	26,473	9,935	16,019	9,879
50	372,368	218,620	580,927	177,770
100	1692,229	1323,827	3578,444	1134,943
500	52626,627	154646,884	521706,864	171745,678
1000	216214,099	904891,408	5782102,216	1038199,656

Grafy o gęstości 50%

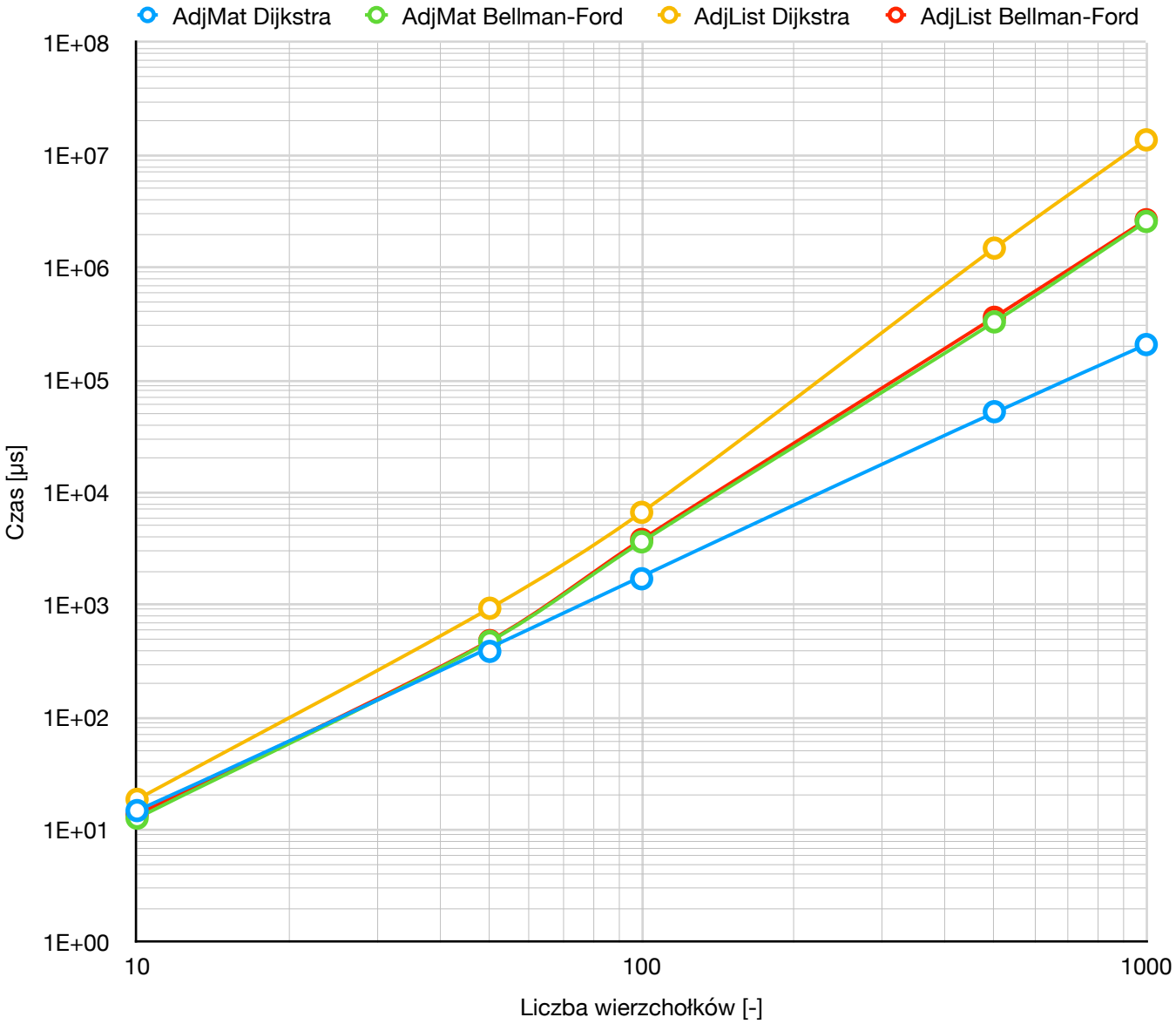


Te wykresy wyglądają podobnie do wykresów dla gęstości 25%: BF działa szybciej dla małych grafów niezależnie od ich implementacji, alg. Dijkstry działa szybciej dla grafów w postaci macierzy sąsiedztwa.

Wyniki testu dla grafów o gęstości 50% [µs]

Reprezentacja	**Macierz sąsiedztwa**	**Macierz sąsiedztwa**	**Lista sąsiedztwa**	**Lista sąsiedztwa**
Algorytm	**Dijkstra**	**Bellman-Ford**	**Dijkstra**	**Bellman-Ford**
10	18,211	11,421	17,518	11,817
50	385,669	350,640	786,691	334,344
100	1712,287	2483,230	5455,454	2450,932
500	52469,674	227314,497	1096055,514	274445,815
1000	207238,063	1723778,649	10598673,188	1795874,358

Grafy o gęstości 75%

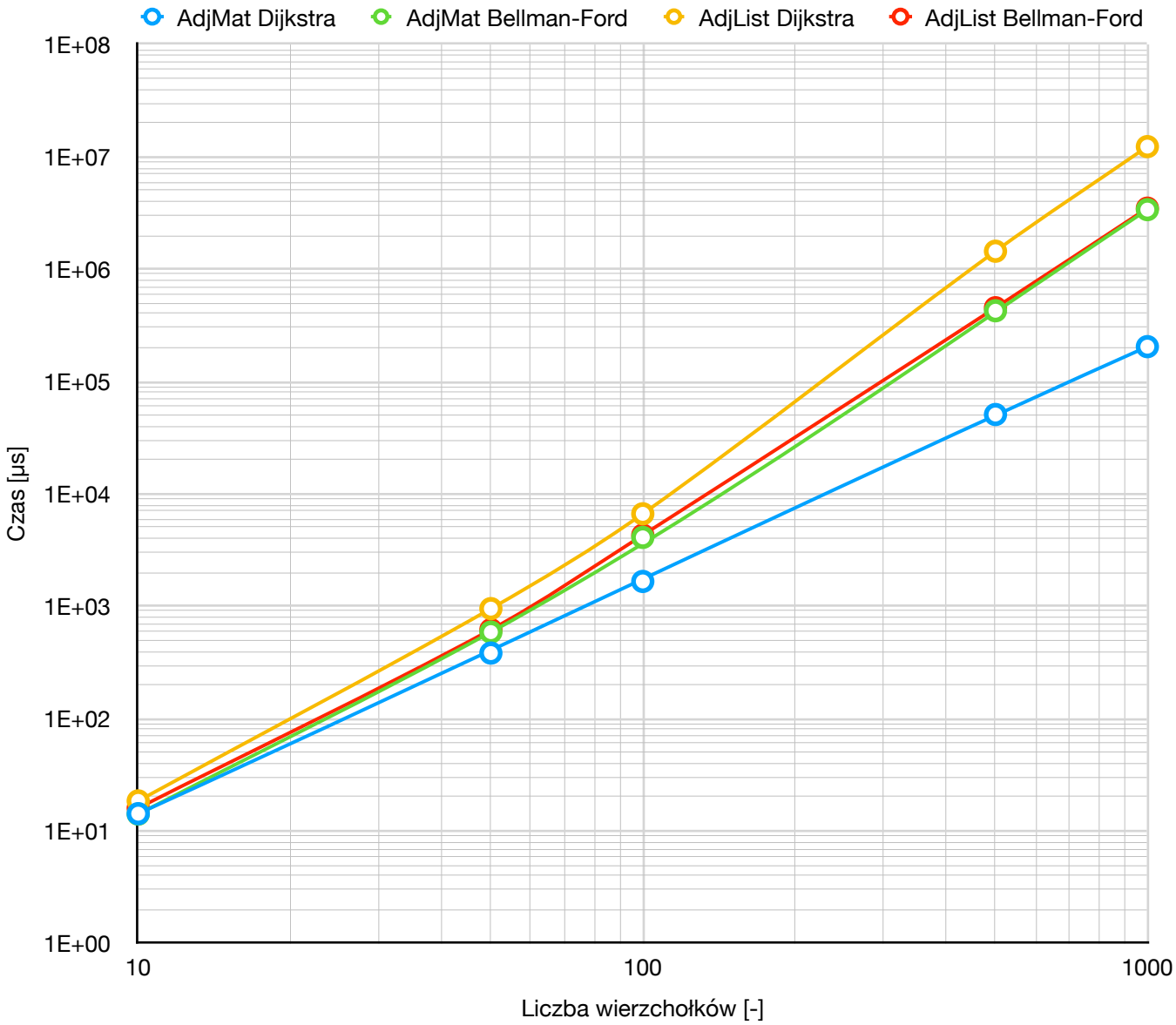


W porównaniu do poprzednich wykresów, wykresy dla algorytmu Bellmana-Forda już nie mają oczywistej przewagi dla małych grafów. Warto też zauważyć, że niebieski wykres prawie nie ruszył z miejsca, co zresztą też widać na pierwszych wykresach (testy grafu w postaci macierzy sąsiedztwa).

Wyniki testu dla grafów o gęstości 75% [µs]

Reprezentacja	**Macierz sąsiedztwa**	**Macierz sąsiedztwa**	**Lista sąsiedztwa**	**Lista sąsiedztwa**
Algorytm	**Dijkstra**	**Bellman-Ford**	**Dijkstra**	**Bellman-Ford**
10	14,844	12,705	18,684	13,671
50	386,716	474,335	935,340	482,290
100	1710,126	3643,344	6662,621	3824,157
500	52334,888	329015,060	1490379,658	361570,191
1000	206883,895	2575316,217	13687208,830	2666293,810

Grafy o gęstości 100%



Czas wykonania algorytmu Bellmana-Forda w końcu przewyższył czas wykonania alg. Dijkstry dla grafu w postaci macierzy sąsiedztwa. Poza tym, położenie wykresów względem siebie nie uległo zmianie.

Wyniki testu dla grafów o gęstości 100% [µs]

Reprezentacja	**Macierz sąsiedztwa**	**Macierz sąsiedztwa**	**Lista sąsiedztwa**	**Lista sąsiedztwa**
Algorytm	**Dijkstra**	**Bellman-Ford**	**Dijkstra**	**Bellman-Ford**
10	14,264	14,133	18,534	15,998
50	383,175	588,745	952,686	619,337
100	1663,319	4102,164	6649,851	4299,667
500	50870,474	426488,875	1444562,016	453732,625
1000	204398,860	3387298,079	12320412,008	3489952,940

Wnioski

Wykonując projekt, zapoznałem się z grafem, jego podstawowymi własnościami oraz implementowałem dwa algorytmy. Dla stosunkowo małych grafów najlepszym według mnie jest algorytm Dijkstry i graf w postaci macierzy sąsiedztwa: choć zużywa on dużo pamięci, ale działa szybko i mało zależy od gęstości.

Literatura

1. J. Wałaszek, Reprezentacja grafów w komputerze *eduinf.waw.pl/inf/alg/001_search/0124.php*, data dostępu 29.04.2020
2. Triangular number (Liczba trójkątna), Wikipedia *en.wikipedia.org/wiki/Triangular number*, data dostępu 29.04.2020
3. Алгоритм Дейкстры (Algorytm Dijkstry), Wikipedia *ru.wikipedia.org/wiki/Алгоритм Дейкстры*, data dostępu 29.04.2020
4. Bellman-Ford algorithm (Algorytm Bellmana-Forda), Wikipedia *en.wikipedia.org/wiki/Bellman–Ford algorithm*, data dostępu 29.04.2020