Algorytmy kombinatoryczne w Bioinformatyce – zadanie II

Natalia Michałkiewicz

147902

Cel projektu:

Należy zaimplementować algorytm dokładny o złożoności wielomianowej realizujący następujące czynności:

− wczytanie dowolnego grafu skierowanego z pliku tekstowego,

− sprawdzenie, czy wczytany graf jest grafem sprzężonym,

− jeśli graf jest grafem sprzężonym, sprawdzenie, czy jest grafem liniowym,

− wypisanie komunikatu o wyniku powyższego sprawdzenia,

− jeśli graf jest grafem sprzężonym, przekształcenie go w jego graf oryginalny (H),

− zapisanie grafu wynikowego H do pliku tekstowego innego niż wejściowy, lecz w tym samym formacie.

Opis zastosowanego formatu zapisu grafu:

Formą zapisu grafu jaką wybrałam jest lista sąsiedztwa.

W pierwszej kolumnie znajdują się wierzchołki, a w następnych – następniki danego wierzchołka:

```
                1 2 3
wierzchołki->   2 4 5
                3 3 6 7     <- następniki
                4 8 9 10
                5 11
```

Dzięki takiemu zapisowi uwzględniłam wierzchołki izolowane, pętle własne, czy krawędzie wielokrotne.

Opis algorytmu:

W mainie wywołuję funkcje typu void i bool zawarte wyżej w programie. Algorytm składa się z wektorów (int) i grafów które są zapisane za pomocą wektora wektorów. Zawiera:

1) Odczyt z pliku
2) Sprawdzenie czy graf jest sprzężony
3) Sprawdzenie czy graf jest liniowy
4) Transformacja
5) Konwersja i indeksowanie
6) Zapis do pliku

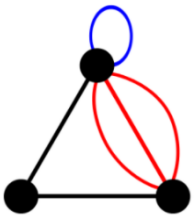
ad.1:

Odczyt z pliku – funkcja typu void odpowiada za załadowanie grafu z pliku tekstowego. Użyta została tutaj przeze mnie biblioteka fstream. Do otworzenia pliku użyłam funkcji open, getline do odczytu linii po linii z pliku txt, push_back w celu dodania następników i funkcji separatora, którą tylko tu wywołuję, a polega na tym, że dodaje do wektora wierzchołki, które u mnie są oddzielone spacją. Zawiera ona substr kopiujący string od pierwszej pozycji (0) do licznika linii, czyli do wartości zmiennej.

ad.2:

Sprawdzenie czy graf jest sprzężony jest już bardziej rozbudowane.

Najpierw musiałam sprawdzić czy plik, który załadowuję to 1-graf czy multigraf.



Multigraf to graf, w którym mogą występować krawędzie wielokrotne oraz pętle własne

1-graf jest grafem, w którym niedopuszczone są krawędzie wielokrotne.

Funkcja, która to u mnie sprawdza jest typu boolean. Porównuje ona czy dla jakiegoś wierzchołka występują dwa identyczne następniki, czyli np. jeśli z jednego wierzchołka 'wychodzą' dwa inne. Występują tu zagnieżdżone pętle for i if sprawdzające warunek czy oba następniki są sobie równe. Jeśli tak funkcja zwraca 'prawdę' = graf jest multigrafem czyli nie jest sprzężony, 'fałsz' jeśli nie znajdzie takiej krawędzi wielokrotnej.

Wtedy może przejść do następnego warunku, którym jest sprawdzenie czy następniki są takie same:

1-graf $G=(V,A)$ jest grafem sprzężonym wtedy i tylko wtedy, gdy następująca własność jest spełniona dla każdej pary $x,y \in V$:
$N^+(x) \cap N^+(y) \neq \emptyset \Rightarrow N^+(x) = N^+(y)$
gdzie $N^+(x)$ jest zbiorem następników x. Czyli, dla każdej pary wierzchołków grafu, ich zbiory (bezpośrednich) następników muszą być albo rozłączne, albo identyczne.

Jeśli są różne czyli zbiory są rozłączne to sprawdza pary wierzchołków. Następnie porównuje rozmiary obu tych zbiorów. 'False' – graf nie jest sprzężony ponieważ liczba następników nie jest taka sama. Gdy zwraca 'True' to może wejść do kolejnej funkcji: Pętla sprawdzająca czy owe następniki są takie same. Iteruje tak po wszystkich wierzchołkach grafu, gdy nigdzie nie pojawi się 'false' to zwróci się 'true' będące poza pętlami.

ad.3:

Dla grafów skierowanych: graf sprzężony jest grafem liniowym wtedy i tylko wtedy, gdy nie zawiera jako podgrafu żadnej z następujących struktur:

W celu sprawdzenia czy graf zawiera pierwszą strukturę, w pętli for szukam wierzchołka i ilość jego następników. Jeśli ich liczba jest większa lub równa dwa to porównuję następniki tych następników, czy są wspólne. ('true gdy jest obecna pierwsza struktura', 'false' gdy nie)

Druga struktura jest podobna. Również szukam forem następnika następników. Różnica jest taka, że tu porównuję wartość wierzchołka od którego zaczynam i kończę. Jeśli się zgadzają zwracane jest 'true' co oznacza, że dana struktura istnieje. ('false' gdy nie)

Trzecia struktura zawiera pętle własne, więc właśnie ich istnienie sprawdzam za pomocą pętli for i if. Jeśli jakas wartość pojawia się w pierwszej kolumnie i powtarza w innej w tym samym wierszu i dla innego wierzchołka występuje ta sama sytuacja, to funkcja porównuje czy istnieją pomiędzy nimi krawędzie.

ad 4.:

Transformacja – czyli główna część naszego zadania. Najpierw wykonałam wiele transformacji na kartce, bez tego nie da się iść dalej.

Funkcja polega na przekształceniu grafu sprzężonego na graf oryginalny. Kierowałam się Pani metodą:

Przekształcenie grafu *G* w graf *H* można zrealizować wg uznania, jednak moim zdaniem najprostszym sposobem jest podany poniżej. Każdy wierzchołek z *G* zamieniamy najpierw na osobny łuk w *H*. Następnie, dla każdego łuku w *G* realizujemy odpowiednie połączenie w *H* poprzez przeindeksowanie wierzchołków z *H*. Przykładowo, dla grafu *G* złożonego z łuków (a,b), (b,c), (b,d), (d,a), tworzymy graf *H* najpierw jako taki zbiór rozłącznych łuków: (1,2), (3,4), (5,6), (7,8), które reprezentują odpowiednio wierzchołki a, b, c, d. Następnie należy odtworzyć jeden do jednego połączenia obecne w *G* w następujący sposób. Mamy łuk (a,b) w *G*, czyli w *H* musi być przejście ze zgodnym zwrotem z łuku odpowiadającego a do łuku odpowiadającego b. Dla rozłącznych obecnie łuków (1,2) i (3,4) potrzebujemy więc skompresować wierzchołki 2 i 3 do jednego, co łatwo uczynić przez przeindeksowanie w zbiorze wszystkich wystąpień 3 na 2. Otrzymujemy łuki (1,2), (2,4), (5,6), (7,8). Tak samo należy postąpić dla wszystkich łuków z *G* po kolei. Po zastosowaniu tej procedury graf na wyjściu nie będzie miał wierzchołków poindeksowanych po kolei, można albo tak zostawić (tylko wtedy format zapisu grafu musi taką sytuację uwzględnić), albo wierzchołki przeindeksować, żeby były po kolei.

W programie rozwiązałam to za pomocą stworzenia dodatkowego grafu za pomocą wektora wektorów. Tworzy on łuki rozłączne, taką samą ich liczbę ile jest wierzchołków w grafie sprzężonym. Następnie numeruje je od 1. Potem pomiędzy tymi łukami tworzone są połączenia odwzorowane z grafu G. Iterując, zmieniane są w ten sposób numery wierzchołków w grafie pomocniczym.

ad 5.:

Jednak przejście nie jest wtedy zgodne więc musimy przeindeksować, posortować i skompresować wierzchołki.

Indeksowanie – ilość wierzchołków musi się zgadzać z ich faktycznymi wartościami. Funkcja, która to robi przechodzi przez wektor przepisując wierzchołkom wartości licząc od 1 i przypisuje do zmiennej maksymalną wartość wierzchołka co jest wykorzystane w pętli for.

Sortowanie – za pomocą sort(). Sortuję cały wektor wektorów w taki sposób, że wierzchołki są po kolei wyświetlane od 1.

Kompresowanie – kompresuję tu wektor przejściowy i przypisuję do wektora tworzącego graf H. Metoda polega na tym, że algorytm szuka wierzchołków i przypisuje je do oryginalnego wektora tak, że wierzchołek i następniki są w jednej linii.

ad 6.

Na końcu zostaje tylko zapisać mój graf oryginalny do pliku w formacie txt (tak jak graf wejściowy). Zasada podobna jak przy odczycie. Jednak tym razem przechodzę pętlami for w grafie oryginalnym.

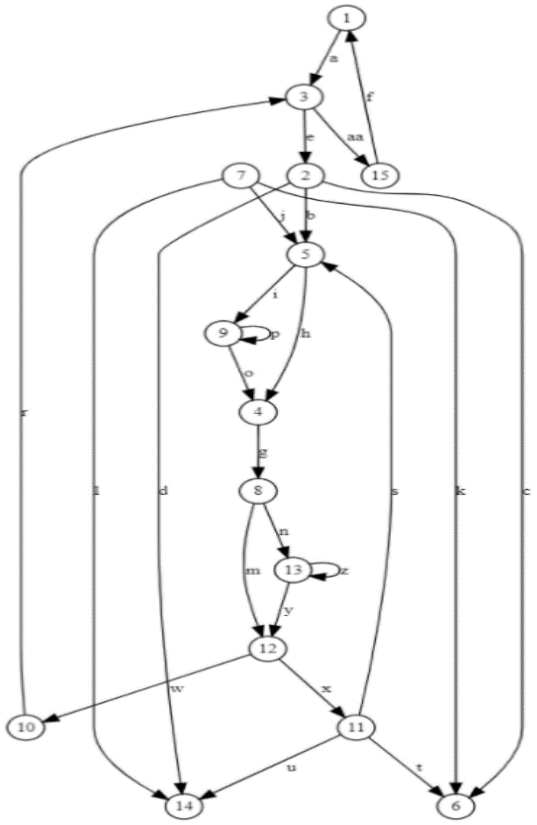
Oszacowanie złożoności algorytmu:

Jest ona szacowana licząc zagnieżdżone pętle przechodzące wierzchołki w grafie. U mnie jest ich 6 wiec złożoność powinna wynosić $O(n^6)$, gdzie n to liczba wierzchołków.

Przeprowadzone testy:

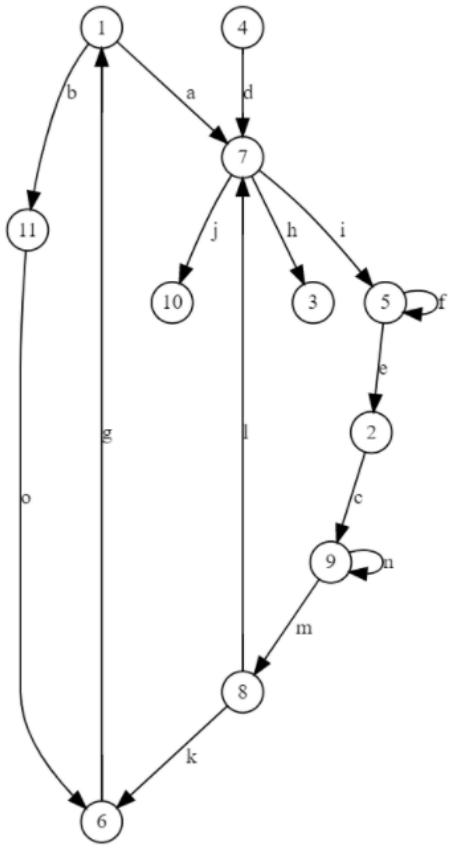
liniowe:

graf G1:



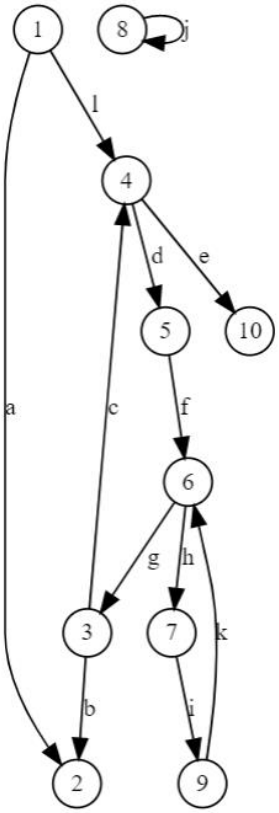

```
1 3
2 5 6 14
3 2 15
4 8
5 4 9
6
7 5 6 14
8 12 13
9 4 9
10 3
11 5 6 14
12 10 11
13 12 13
14
15 1
```

graf H1:



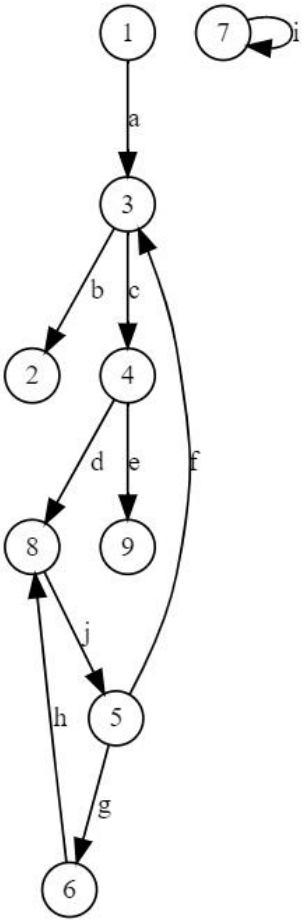

```
1 7 11
2 9
3
4 7
5 2 5
6 1
7 3 5 10
8 6 7
9 8 9
10
11 6
```

graf G2:




```
1 2 4
2
3 2 4
4 5 10
5 6
6 3 7
7 9
8 8
9 6
10
```

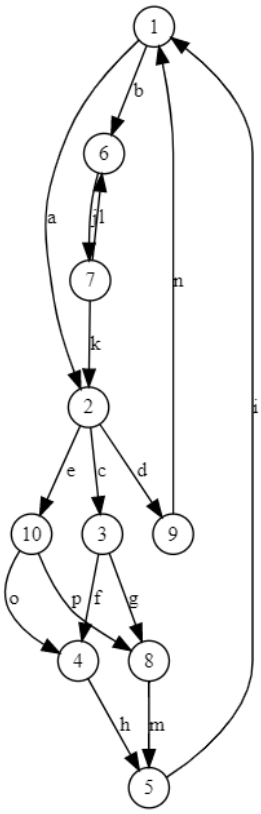
graf H2:




```
1 3
2
3 2 4
4 8 9
5 3 6
6 8
7 7
8 5
9
```

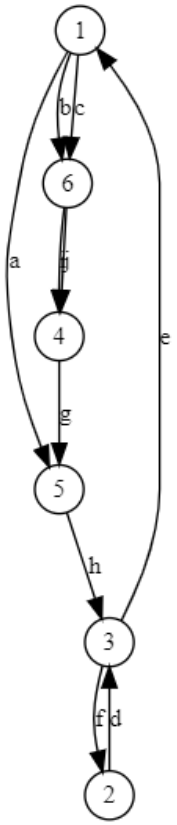
nieliniowe:

graf G3:



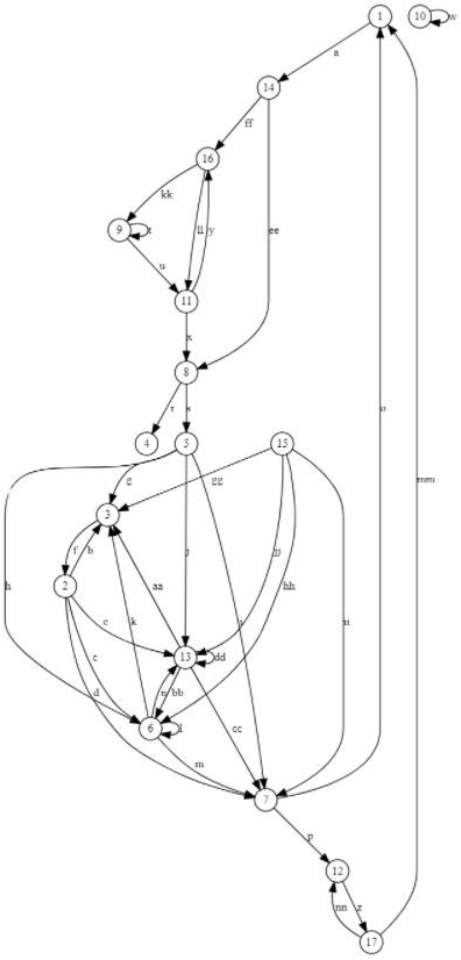

```
1 2 6
2 3 9 10
3 4 8
4 5
5 1
6 7
7 2 6
8 5
9 1
10 4 8
```

graf H3:



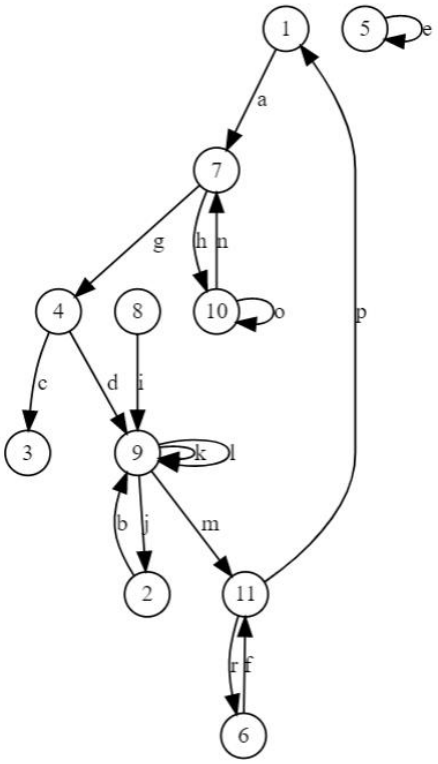

```
1 5 6 6
2 3
3 1 2
4 5
5 3
6 4 4
```

graf G4:



```
1 14
2 3 6 7 13
3 2
4
5 3 6 7 13
6 3 6 7 13
7 1 12
8 4 5
9 9 11
10 10
11 8 16
12 17
13 3 6 7 13
14 8 16
15 3 6 7 13
16 9 11
17 1 12
```

graf H4:



```
1 7
2 9
3
4 3 9
5 5
6 11
7 4 10
8 9
9 2 9 9 11
10 7 10
11 1 6
```

Wnioski:

Na początku próbowałam do rozwiązania używać macierzy, co tylko utrudniło mi pracę, z czym wiąże się tak późne oddanie pracy, ponieważ wszystko musiałam zacząć od nowa. Wektory są dużo prostszym rozwiązaniem. Rysowanie i transformowanie grafów ręcznie również okazało się niemałym wyzwaniem, jednak radość gdy zaczęły mi wychodzić jest nie do opisania.