Algorytmy i struktury danych

Zadanie projektowe

P02 Jakub Goleń
Inżynieria i analiza danych

Temat zadania

Napisz program, który dla zadanego grafu skierowanego reprezentowanego przy pomocy macierzy incydencji wyznaczy i wypisze następujące informacje:

1) wszystkich sąsiadów dla każdego wierzchołka
2) wszystkie wierzchołki, które są sąsiadami każdego wierzchołka
3) stopnie wychodzące wszystkich wierzchołków
4) stopnie wchodzące wszystkich wierzchołków
5) wszystkie wierzchołki izolowane
6) wszystkie pętle
7) wszystkie krawędzie dwukierunkowe

Użyte biblioteki

W całym programie użyto:

- **vector** - wyposaża klasyczną tablicę w kilka mechanizmów.
- **chrono** - oblicza czas.
- **bits/stdc++.h** - wyposaża wszystkie standardowe biblioteki c++ (w programie wykorzystano fstream i sstream).
- **Graph.h** - autorski plik zawierający klasę do wykonywania funkcji dla podpunktów podanych w zadaniu.

Inne funkcje w programie

W Graph.h:

- matrixValidation(matrix)
  - Funkcja sprawdza czy podana macierz jest macierzą incydencji. Zwraca wartość typu boolean.
- edgeMapping()
  - Funkcja mapująca podaną macierz do macierzy, która zawiera w sobie wartości {source,destination}. Praktyczne zastosowanie w podpunkcie 7.

Podpunkt 1 (getVerticeNeighbors())

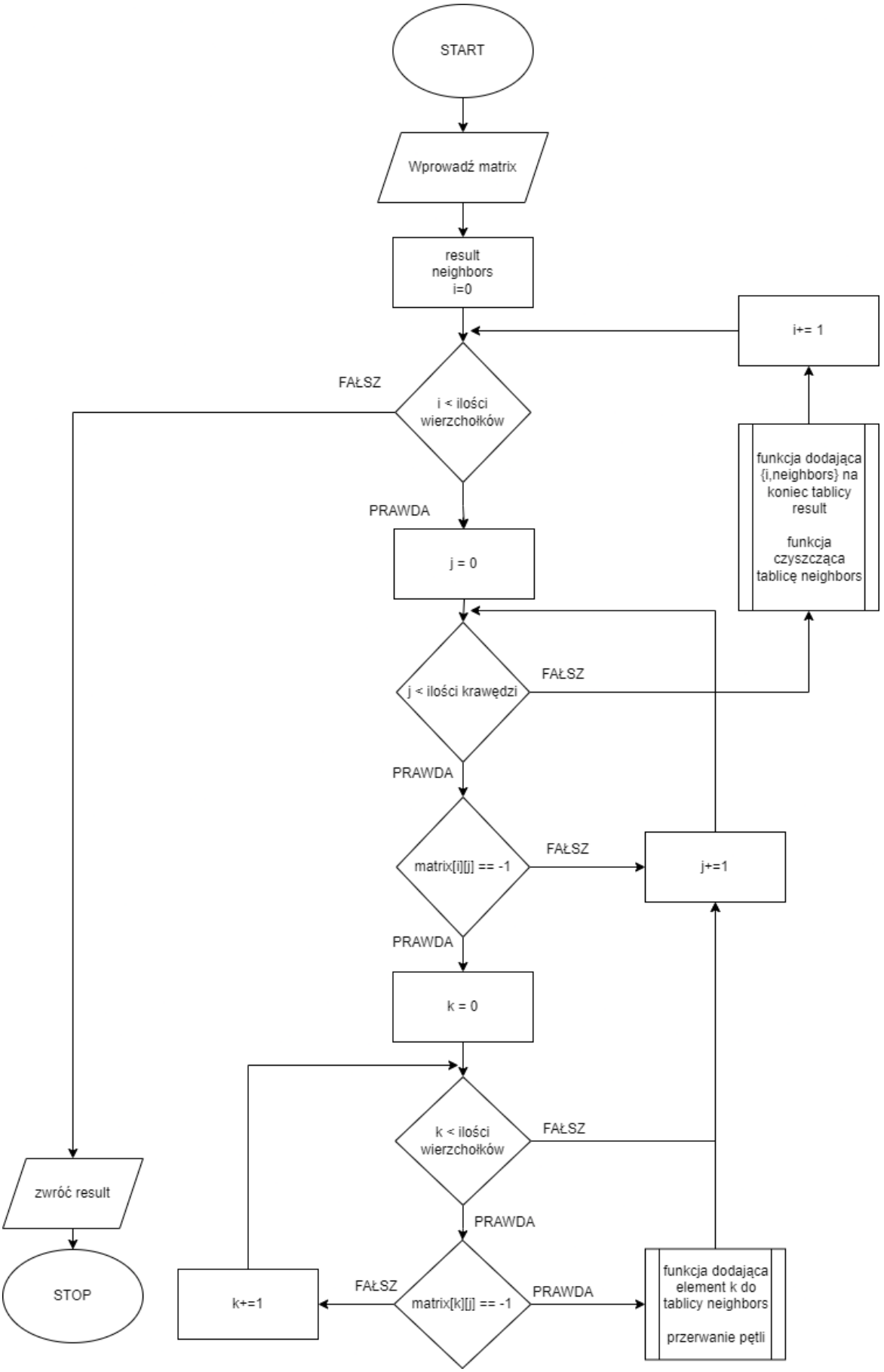
Opis działania

Wstępnie program przechodzi przez wszystkie elementy macierzy. Jeżeli natrafi na wartość 1, przechodzi przez wszystkie wartości w kolumnie od początku. Jeżeli natrafi na wartość -1 dodaje indeks wiersza (numer wierzchołka) do zmiennej typu vector<int> o nazwie neighbors. Po przejściu przez wiersz, funkcja dodaje do zmiennej result typu vector<int_vector> dane w formacie {indeks wiersza,{neighbors}}, a następnie czyści wektor neighbors. “int_vector” jest to typ danych zawierający zmienną typu int i zmienną typu vector<int> w formacie: {int num, vector<int> arr}.

Złożoność

Złożoność algorytmu to $O(n^3)$.

Schemat blokowy



Pseudokod

```
getVerticeNeighbors(matrix){
  result
  neighbors
     dla(i = 0 gdy i < ilość wierzchołków){
        dla(j = 0 gdy j < ilość krawędzi){
          jeżeli(matrix[i][j] == 1){
              dla(k = 0 gdy k < this->verticeAmount){
                 jeżeli(matrix[k][j] == -1){
                     funkcja dodająca wartość k na koniec tablicy neighbors
                    przerwanie pętli
                 }
              k+=1
              }
           }
        j+=1
        }
        i+=1
        funkcja dodająca wartość {i,neighbors} na koniec macierzy result
        funkcja usuwająca wszystkie elementy z tablicy neighbors
     }
  zwróć result
}
```

Kod

```
vector<int_vector> getVerticeNeighbors(){
  vector<int_vector> result
  vector<int> neighbors;

     for(int i = 0; i < this->verticeAmount; i++){
        for(int j = 0; j < this->edgeAmount; j++){
           if(this->matrix[i][j] == 1){
              for(int k = 0; k < this->verticeAmount; k++){
                 if(this->matrix[k][j] == -1){
                    neighbors.push_back(k);
                    break;
                 }
              }
           }
        }
        result.push_back({i,neighbors});
        neighbors.clear();
     }
  return result;//zwracamy wynik
}
```

Podpunkt 2 (getNeighborsOfAll())

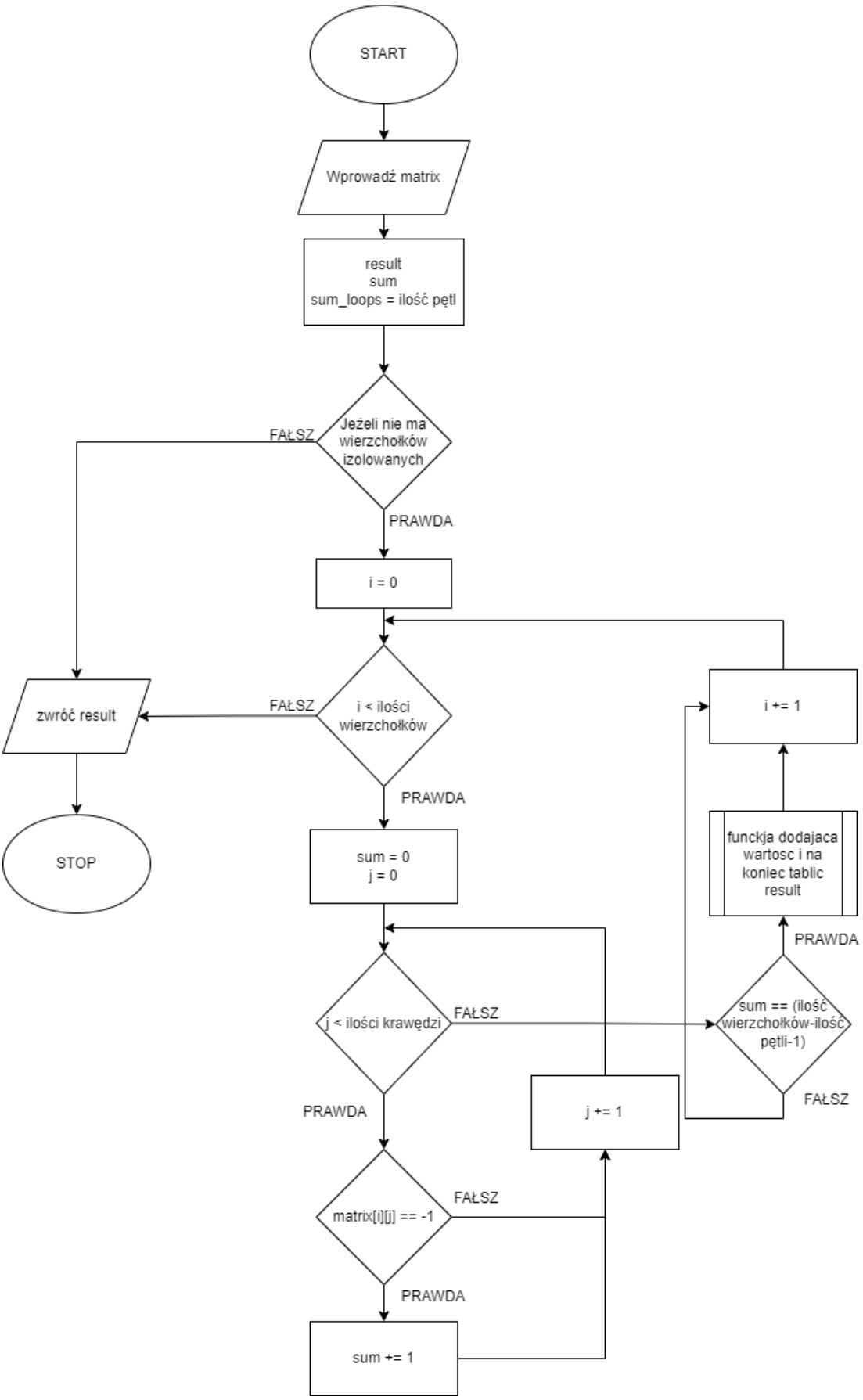
Opis działania

Metoda najpierw sprawdza czy w podanej macierzy incydencji występują wierzchołki izolowane. Z logicznego punktu widzenia jeżeli takowe występują, żaden z wierzchołków nie może być sąsiadem wszystkich wierzchołków i wtedy funkcja zwraca pustą tablicę. Jeżeli nie ma wierzchołków izolowanych, funkcja przechodzi po wszystkich elementach macierzy i w międzyczasie (po pierwszej pętli for) ustala wartość sumy na 0. Później sprawdza czy aktualna wartość jest równa 1. Jeżeli warunek jest spełniony, funkcja inkrementuje wartość sumy o 1. Po przejściu po wierszu, algorytm sprawdza czy suma jedynek jest równa wyrażeniu (ilość wierzchołków - ilość pętl -1). Jeżeli warunek został spełniony funkcja dodaje do zmiennej wyjściowej element o wartości indeksu wiersza (numer wierzchołka).

Złożoność

Złożoność algorytmu to $O(n^2)$.

Schemat blokowy



Pseudokod

```
getNeighborsOfAll(matrix){
  result
  sum
  sum_loops = ilość pętli

 jeżeli(tablica pętli jest pusta){
     dla(i = 0 gdy i < ilość wierzchołków){
        sum = 0
        dla(j = 0 gdy j < ilość krawędzi){
              jeżeli(matrix[i][j] == -1){
                 sum+=1;
              }
        j+=1
        }
        jeżeli(sum == ilość wierzchołków-sum_loops-1){
           funkcja dodająca wartość i na koniec tablicy result
        }
     i+=1
     }
  }
  zwróć result
}
```

Kod

```
vector<int> getNeighborsOfAll(){
  vector<int> result; //zmienna wyjsciowa
  int sum, sum_loops = this->loops.size();

  if(this->isolated.empty()){
     for(int i = 0; i < this->verticeAmount; i++){
        sum = 0;
        for(int j = 0; j < this->edgeAmount; j++){
              if( this->matrix[i][j] == -1){
                 sum++;
              }
        }
        if(sum == this->verticeAmount-sum_loops-1){
           result.push_back(i);
        }
     }
  }
  return result;
}
```

Podpunkt 3 i 4 (getVerticesOutdegrees()/getVerticesIndegrees())

Opis działania

Połączyłem dwa podpunkty w jeden, gdyż poza jednym warunkiem ich logika jak i kod są te same. Funkcja na początku przechodzi przez wszystkie elementy. Jeżeli natrafi na:

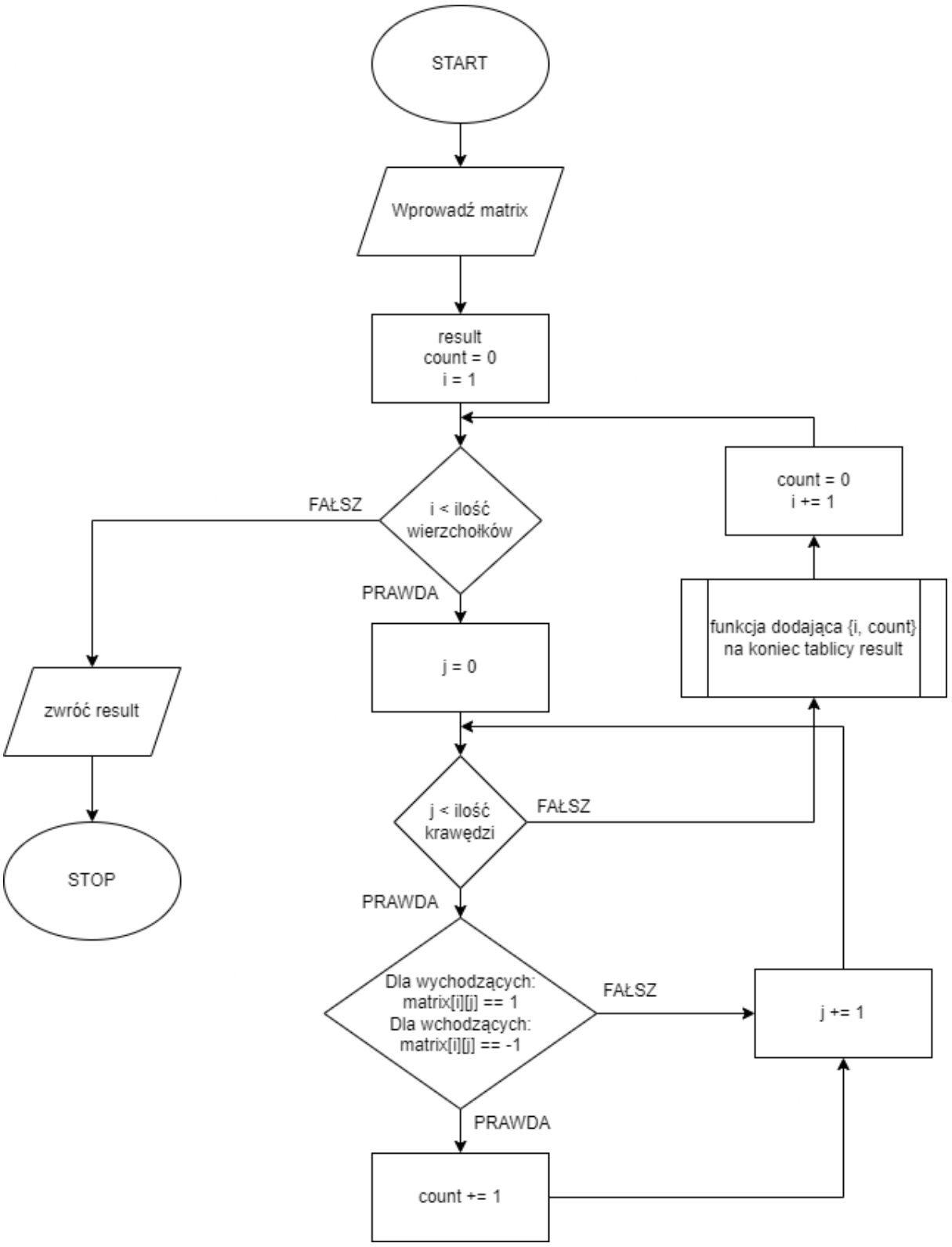
- ➔ -1 dla stopni wchodzących,
- ➔ 1 dla stopni wychodzących

inkrementuje się zmienna count. Po przejściu po jednym wierszu, metoda dodaje do zmiennej wyjściowej result dane w formie {indeks wiersza, count}. Algorytm na końcu zwraca tablicę result.

Złożoność

Złożoność algorytmu to $O(n^2)$.

Schemat blokowy



Pseudokod

```
getVerticesOutdegrees/getVerticesIndegrees(matrix){
  result
  count = 0
  index = 1

  dla( el w matrix){
    dla(cell w el){
      jeżeli((dla stopni wychodzących: cell == 1)(dla stopni wchodzących: cell == -1)){
        count+=1
      }
    }
    result.push_back({index, count});
    funkcja dodająca element {index,count} na koniec macierzy result
    index += 1
    count = 0
  }
  zwróć result
}
```

Kod

Stopień wychodzący

```
vector<vector<int>> getVerticesOutdegrees(){
  vector<vector<int>> result;
  int count = 0;
  int index = 1;

  for(vector<int> el: this->matrix){
    for(int cell: el){
      if(cell == 1){
        count++;
      }
    }
    result.push_back({index, count});
    index++;
    count = 0;
  }
  return result;
}
```

Stopień wchodzący

```
vector<vector<int>> getVerticesIndegrees(){
  vector<vector<int>> result;
  int count = 0;
  int index = 1;

  for(vector<int> el: this->matrix){
     for(int cell: el){
        if(cell == -1){
           count++;
        }
     }
     result.push_back({index, count});
     index++;
     count = 0;
  }
  return result;
}
```

Podpunkt 5 (getIsolatedVertices())

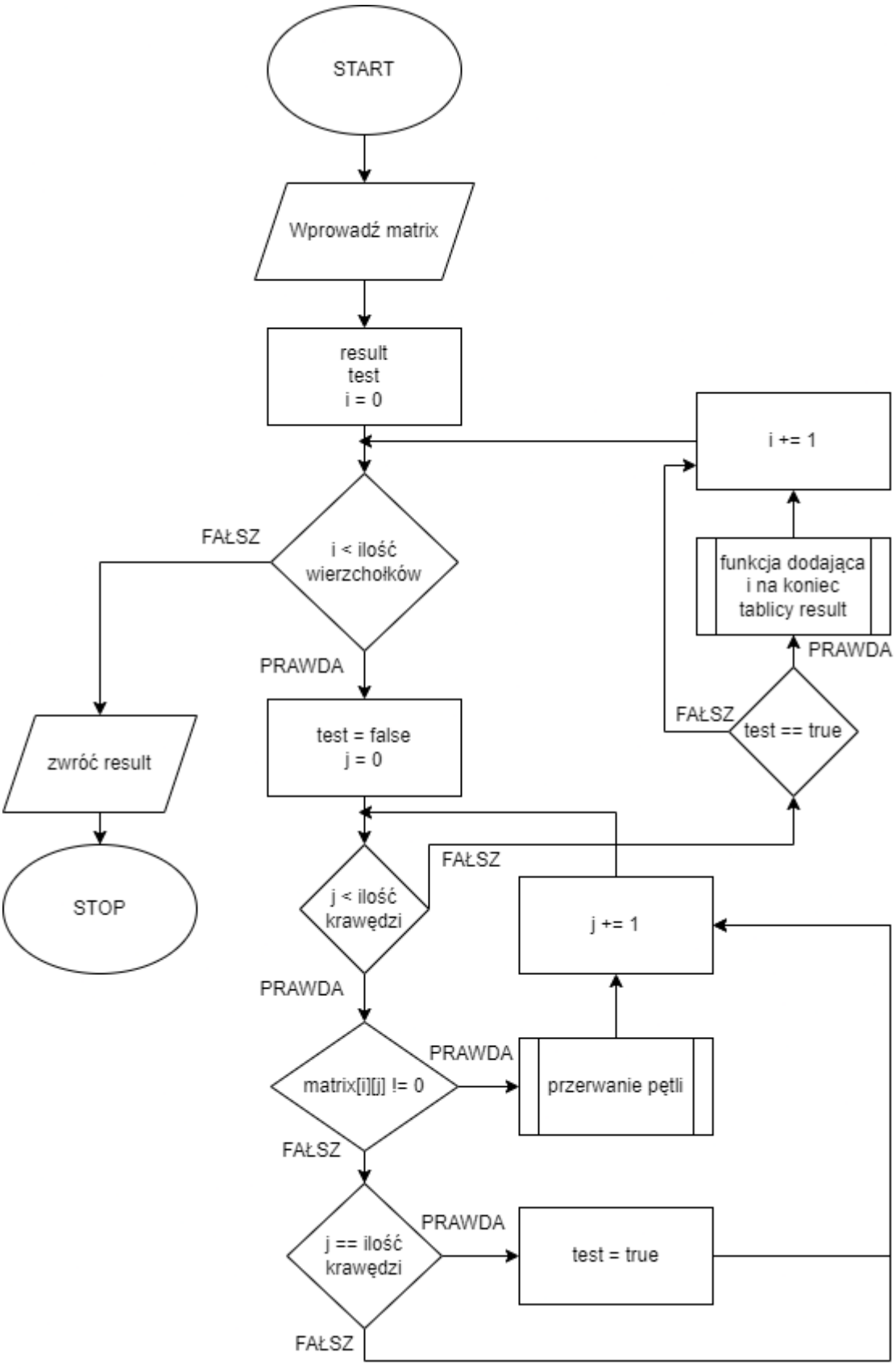
Opis działania

Funkcja na początku przechodzi po wszystkich elementach macierzy. Pomiędzy pętlami ustala wartość zmienny sprawdzającej test typu bool na false. Jeżeli algorytm nie natrafi na 0, druga pętla przerywa się. Jeżeli funkcja natrafi na 0, ustala wartość zmiennej test na true.
Po drugiej pętli funkcja sprawdza czy zmienna test ma wartość true. Jeżeli ma indeks wiersza zostaje dodany do tablicy wyjściowej. Funkcja na koniec zwraca tablicę wyjściową result.

Złożoność

Złożoność algorytmu to $O(n^2)$.

Schemat blokowy



Pseudokod

```
getIsolatedVertices(matrix){
   result
   index = 0
   test

   dla(el w matrix){
      test = false
      dla(i  = 0 gdy i < el.size()){
         jeżeli(el[i] != 0){
            przerwanie pętli
         }
         jeżeli(i == el.size()-1){
            test = true
         }
      i+=1
      }
      jeżeli(test == true){
          funkcja dodająca element index na koniec tablicy result
      }
      index += 1
   }

   zwróć result
}
```

Kod

```
vector<int> getIsolatedVertices(){
  vector<int> result;
  int index = 0;
  bool test;

  for(vector<int> el: this->matrix){
     test = false;
     for(int i  = 0; i < el.size(); i++){
        if(el[i] != 0){
           break;
        }
        if(i == el.size()-1){
           test = true;
        }
     }
     if(test){
        result.push_back(index);
     }
     index++;
  }

  return result;
}
```

Podpunkt 6 (getLoops())

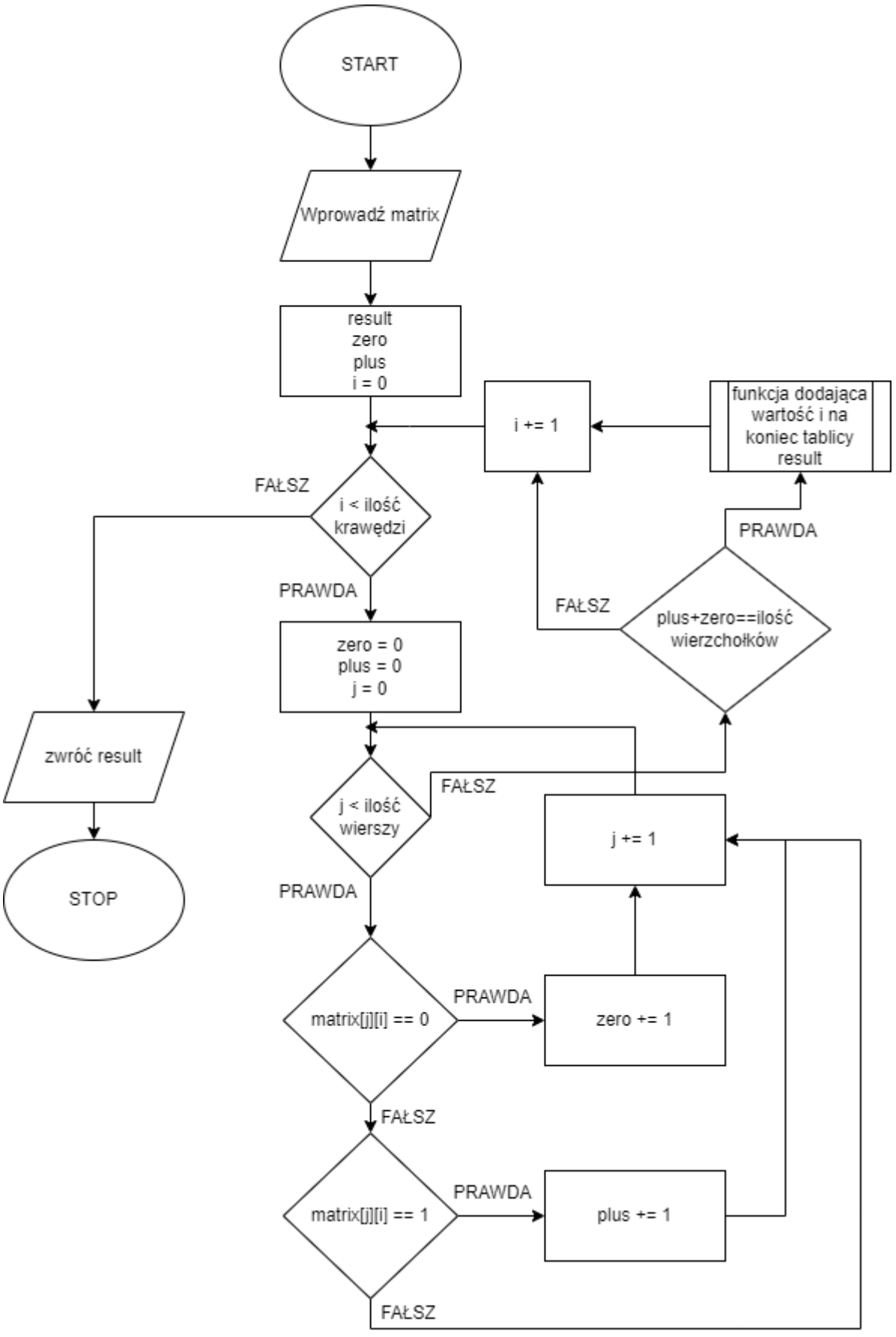
Opis działania

Funkcja przechodzi po wszystkich elementach po kolumnie, zliczając zera i jedynki. Jeżeli suma ilości zer i jedynek jest równa ilości wierzchołków, metoda dodaje indeks kolumny (numer krawędzi) do tablicy wyjściowej result. Na koniec algorytm zwraca tablicę wyjściową.

Złożoność

Złożoność algorytmu to $O(n^2)$.

Schemat blokowy



Pseudokod

```
getLoops(){
  result
  zero
  plus

  dla(i = 0 gdy i < ilość krawędzi){
     zero = 0
     plus = 0
     dla(el w matrix){
        jeżeli(el[i] == 0){
           zero+=1
        }
        w innym wypadku jeżeli(el[i] == 1){
           plus+=1
        }
     }
     jeżeli(plus+zero==ilość wierzchołków){
        funkcja dodająca wartość i na koniec tablicy result
     }
     i+=1
  }

  zwróć result
}
```

Kod

```
vector<int> getLoops(){
  vector<int> result;
  int zero, plus;

  for(int i = 0; i < edgeAmount;i++){
     zero = 0;
     plus = 0;
     for(auto & el : matrix){
        if(el[i] == 0){
           zero++;
        }
        else if(el[i] == 1){
           plus++;
        }
     }
     if(plus+zero==this->verticeAmount){
        result.push_back(i);
     }
  }

  return result;
}
```

Podpunkt 7 (getBidirectionalEdges())

Funkcja tworzy macierz połączeń ({source, destination}), czyli przechodząc kolumnowo przez elementy macierzy matrix natrafia na 1 lub -1 i dodaje je do zmiennej edgeMap w formie{wierzchołek z 1, wierzchołek z -1}. Następnie funkcja przechodzi po elementach tej macierzy. Po pierwszej pętli algorytm bierze dwa elementy (source,destination) i przeszukuje całą macierz tymi elementami w odwrotnej kolejności. Jeżeli natrafi na te same elementy w odwrotnej kolejności sprawdza czy już były wcześniej dodane. Jeżeli nie zostały wcześniej dodane są dodawane do zmiennej tablicy dwuwymiarowej result. Funkcja na koniec zwraca wartość result.

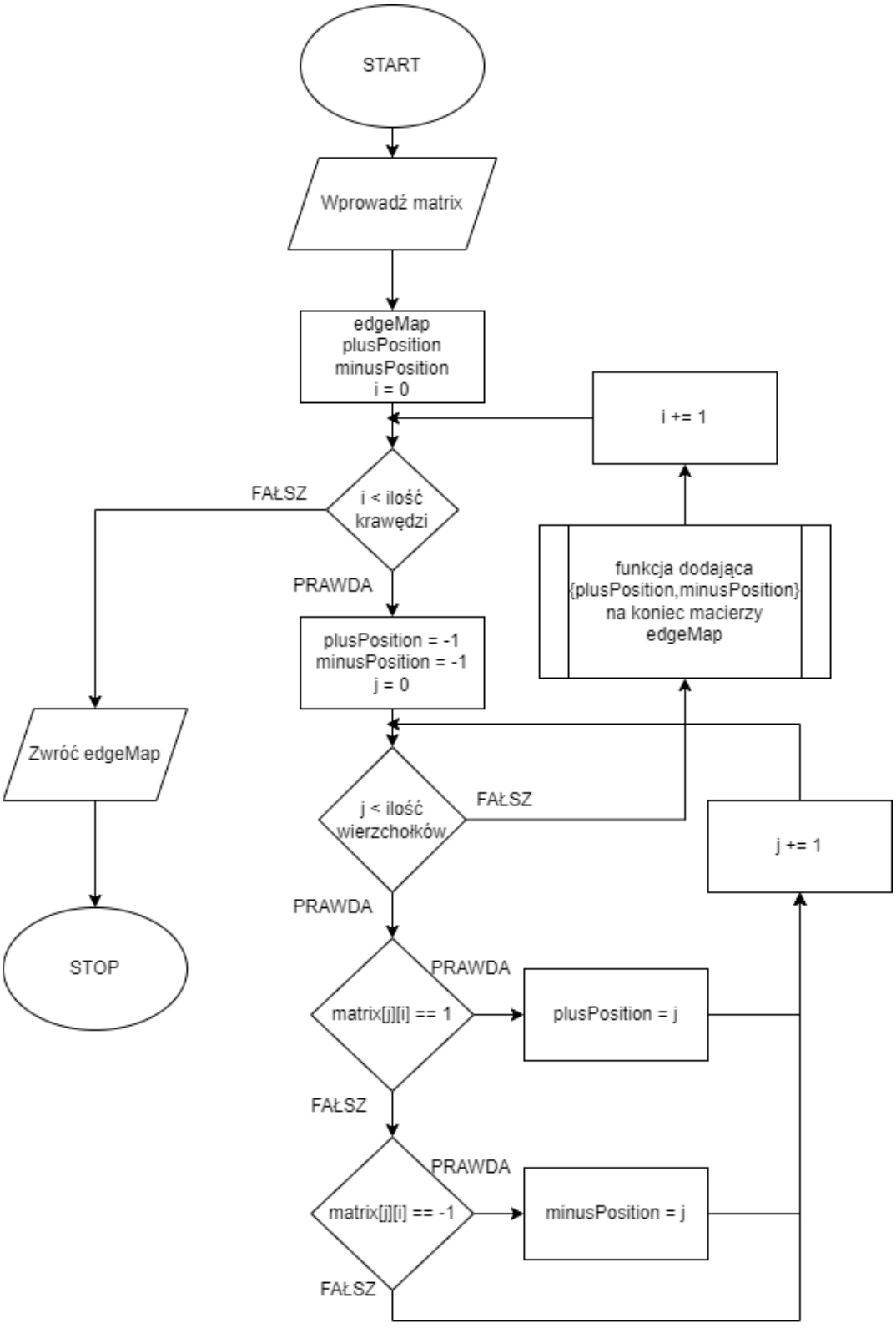
Złożoność

Złożoność algorytmu to $O(n^3)$.

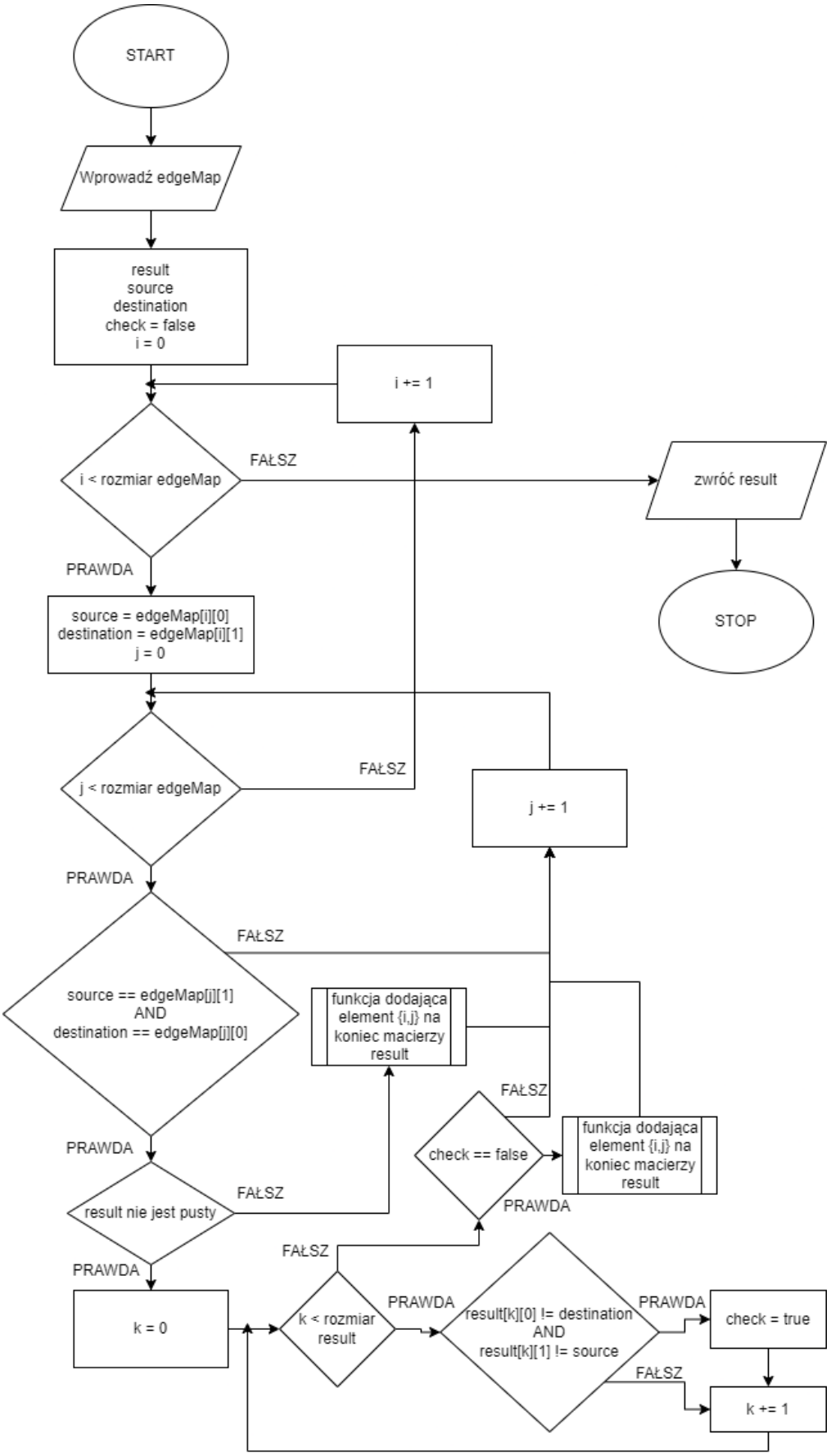
Złożoność funkcji edgeMapping to $O(n^2)$.

Schemat blokowy

EdgeMap



Algorytm


START
Wprowadź edgeMap
result
source
destination
check = false
i = 0
i += 1
i < rozmiar edgeMap
FAŁSZ
zwróć result
STOP
PRAWDA
source = edgeMap[i][0]
destination = edgeMap[i][1]
j = 0
j < rozmiar edgeMap
FAŁSZ
j += 1
PRAWDA
FAŁSZ
source == edgeMap[j][1]
AND
destination == edgeMap[j][0]
funkcja dodająca element {i,j} na koniec macierzy result
FAŁSZ
check == false
funkcja dodająca element {i,j} na koniec macierzy result
PRAWDA
PRAWDA
result nie jest pusty
FAŁSZ
FAŁSZ
PRAWDA
k = 0
k < rozmiar result
PRAWDA
result[k][0] != destination
AND
result[k][1] != source
PRAWDA
check = true
FAŁSZ
k += 1

Pseudokod

EdgeMap

```
edgeMapping(matrix){
  edgeMap
  plusPosition
  minusPosition

  dla(i = 0 gdy i < ilość krawędzi){
     plusPosition = -1
     minusPosition = -1

     dla(j = 0 gdy j < ilość wierzchołków){
        jeżeli(matrix[j][i] == 1){
           plusPosition = j
        }
        w innym wypadku jeżeli(matrix[j][i] == -1){
           minusPosition = j
        }
        j+=1

     }
     funkcja dodająca {plusPosition, minusPosition} na koniec macierzy edgeMap
     i += 1
  }
}
```

Algorytm

```
getBidirectionalEdges(edgeMap){
      result
      source
      destiantion
      bool check = false

      dla(i = 0 gdy i < rozmiar edgeMap){
         source = edgeMap[i][0]
         destiantion = edgeMap[i][1]
         dla(j = 0 gdy j < rozmiar edgeMap){
            jeżeli(source == edgeMap[j][1] AND destiantion == edgeMap[j][0]){
               jeżeli(macierz result nie jest pusta){
                  dla(el w result){
                     jeżeli(el[0] != destiantion AND el[1] != source){
                        check = true
                     }
                  }
                  if(check == false){
                     funkcja dodająca element {i,j} na koniec macierzy result
                  }
               }
               else{
                  funkcja dodająca element {i,j} na koniec macierzy result
               }
            }
            j+=1
         }
         i+=1
      }

      zwróć result
   }

};
```

Kod

EdgeMap

```
void edgeMapping(){
  int plusPosition, minusPosition;

  for(int i = 0; i < this->edgeAmount; i++){
     plusPosition = -1;
     minusPosition = -1;

     for(int j = 0; j < this->verticeAmount; j++){
        if(this->matrix[j][i] == 1){
           plusPosition = j;
        }
        else if(this->matrix[j][i] == -1){
           minusPosition = j;
        }

     }
     this->edgeMap.push_back({plusPosition,minusPosition});

  }
}
```

Algorytm

```
 vector<vector<int>> getBidirectionalEdges(){
     vector<vector<int>> result;
     int source, destiantion;
     bool check = false;

     for(int i = 0; i <this->edgeMap.size(); i++){
        source = this->edgeMap[i][0];
        destiantion = this->edgeMap[i][1];
        for(int j = 0; j < this->edgeMap.size(); j++){
           if(source == this->edgeMap[j][1] && destiantion == this->edgeMap[j][0]){
              if(!result.empty()){
                 for(auto el: result){
                    if(el[0] != destiantion && el[1] != source){
                       check = true;
                    }
                 }
                 if(!check){
                    result.push_back({i,j});
                 }
              }
              else{
                 result.push_back({i,j});
              }
           }
        }
     }

     return result;
  }

};
```

Podsumowanie

Program wykonuje wszystkie zadane podpunkty (zawartość pliku wyjściowego):

[1 1 0 0 0 1]
[-1 0 -1 0 0 0]
[0 -1 1 1 -1 0]
[0 0 0 -1 1 0]
[0 0 0 0 0 0]

Podpunkt 1: Wszyscy sasiedzi dla kazdego wierzcholka

0: 1, 2,
1:
2: 1, 3,
3: 2,
4:

Czas trwania algorytmu: 0.0052 ms

Podpunkt 2: Wszystkie wierzcholki, ktore sa sasiadami kazdego wierzcholka

Wierzcholki: Brak sasiadow kazdego wierzcholka

Czas trwania algorytmu: 0.0001 ms

Podpunkt 3: Stopnie wychodzace wszystkich wierzczolkow

0: 3 Stopien
1: 0 Stopien
2: 2 Stopien
3: 1 Stopien
4: 0 Stopien

Czas trwania algorytmu: 0.0065 ms

Podpunkt 4: Stopnie wchodzace wszystkich wierzczolkow

0: 0 Stopien
1: 2 Stopien
2: 2 Stopien
3: 1 Stopien
4: 0 Stopien

Czas trwania algorytmu: 0.0043 ms

Podpunkt 5: Wierzcholki izolowane

Wierzcholki: 4,

Czas trwania algorytmu: 0.0013 ms

Podpunkt 6: Petle

Krawedzie: 5,

Czas trwania algorytmu: 0.0007 ms
Podpunkt 7: Krawedzie dwukierunkowe
3 : 4
Czas trwania algorytmu: 0.0012 ms