

WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ



SYSTEMY RÓWNOLEGŁE I ROZPROSZONE
**Drzewa wszystkich najkrótszych
ścieżek - algorytm Dijkstry**
Aplikacja równoległa MPI

Arkadiusz Kasprzak
Aleksandra Poręba

21 kwietnia 2020

Spis treści

1	Wstęp	3
1.1	Algorytm Dijkstry	3
1.2	Aplikacje równoległe	3
2	Budowa projektu	4
2.1	Podział na komponenty	4
2.2	Najważniejsze klasy w projekcie	5
3	Działanie projektu	6
3.1	Inicjalizacja	6
3.2	Implementacja i zakończenie algorytmu	11
3.3	Zastosowane funkcjonalności MPI	15
4	Testy projektu	15
5	Obsługa programu	16
5.1	Kompilacja	16
5.2	Uruchomienie	16
5.3	Dodatkowe opcje	17
5.4	Dane wejściowe	18
5.5	Generowanie grafów	18

1 Wstęp

Niniejszy dokument stanowi dokumentację projektu wykonanego w ramach przedmiotu *Systemy równoległe i rozproszone*. Tematem projektu było stworzenie, z użyciem protokołu MPI, aplikacji równoległej implementującej algorytm Dijkstry.

1.1 Algorytm Dijkstry

Zaimplementowany w ramach projektu algorytm Dijkstry jest algorytmem poszukiwania najkrótszych ścieżek z wybranego wierzchołka do wszystkich pozostałych w grafie (skierowanym lub nieskierowanym) o nieujemnych wagach krawędzi. Oparty jest on na metodzie zachłannej: w każdym kroku wybierany jest wierzchołek, do którego koszt dojścia jest najmniejszy.

```
V – zbior wierzchołkow
E – zbior krawędzi z wagami
v – wierzchołek startowy

Dijkstra(V, E, v)
  Q := ∅;
  p := -1;
  d := ∞
  d(v) := 0
  dopoki Q != V wykonaj
    wybierz spoza zbioru Q wierzchołek u o najmniejszym koszcie d(u)
    Q := Q ∪ {u}
    dla każdego sasiada w wierzchołka u spoza zbioru Q wykonaj
      jeśli d(w) > d(u) + E(w, u) to
        d(w) := d(u) + E(w, u)
        p(w) := u
```

Listing 1: Pseudokod algorytmu Dijkstry.

W czasie wykonania algorytmu program operuje na dwóch tablicach - jedna z nich, zwykle oznaczana literą d , przechowuje koszty dotarcia z wierzchołka źródłowego do każdego z wierzchołków w badanym grafie. Druga, oznaczana literą p , przechowuje informację o poprzedniku każdego z wierzchołków - taka informacja pozwala po zakończeniu działania algorytmu odtworzyć ścieżkę z wierzchołka źródłowego do każdego z wierzchołków w grafie.

1.2 Aplikacje równoległe

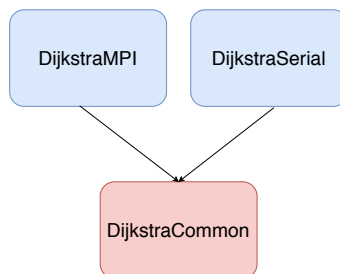
2 Budowa projektu

W tej części dokumentacji opisana została budowa przygotowanego projektu - zarówno jeśli chodzi o podział projektu na komponenty (biblioteki), jak i na poszczególne klasy. Nie jest to natomiast dokumentacja kodu - ta została przygotowana odrębnie, w postaci zbioru dokumentów HTML, przy pomocy narzędzia *Doxygen*.

2.1 Podział na komponenty

Projekt podzielony został na trzy główne komponenty:

- **DijkstraCommon** - biblioteka statyczna zawierająca klasy oraz funkcje współdzielone przez pozostałe dwa komponenty. W jej skład wchodzi klasy odpowiedzialne za: zapis wyników obliczeń, walidację danych podanych przez użytkownika, przechowywanie danych na temat przetwarzanego grafu i jego poszczególnych wierzchołków czy przetwarzanie argumentów linii poleceń. Zawiera również klasę *DijkstraAlgorithmBackend* stanowiącą szkielet implementowanego algorytmu.
- **DijkstraSerial** - implementacja algorytmu bez użycia protokołu MPI, czyli w formie klasycznego algorytmu szeregowego. Komponent ten został dołączony do projektu jako punkt odniesienia w procesie testowania wydajności implementacji równoległej. Zawiera funkcję *main*.
- **DijkstraMPI** - implementacja algorytmu używająca protokołu MPI. Jest to jedyny komponent projektu używający tej biblioteki. Zawiera funkcję *main*.



Rysunek 1: Komponenty wchodzące w skład projektu wraz z wzajemnymi zależnościami.

Zależności między poszczególnymi komponentami przedstawia rysunek 1. Na niebiesko oznaczone zostały komponenty produkujące plik wykonywalny, na czerwono natomiast biblioteki. Każdy z komponentów znajduje się w osobnym katalogu o nazwie odpowiadającej nazwie komponentu.

2.2 Najważniejsze klasy w projekcie

Poniżej przedstawiono listę najważniejszych klas wchodzących w skład projektu (nie jest to pełna lista wszystkich klas, taką listę znaleźć można w dokumentacji wygenerowanej za pomocą narzędzia *Doxygen*):

- klasa `DijkstraAlgorithmBackend` z komponentu `DijkstraCommon` - stanowi podstawę implementacji algorytmu zarówno w przypadku równoległym, jak i sekwencyjnym. Zarządza stanem wykonania algorytmu i wykonuje na tym stanie podstawowe operacje przewidziane przez algorytm, np. dodanie wierzchołka do zbioru wierzchołków przetworzonych czy wybór wierzchołka o najniższym koszcie.
- klasa `DijkstraMPI` z komponentu `DijkstraMPI` - zawiera równoległą implementację algorytmu Dijkstry. Korzysta z API udostępnionego przez klasę `DijkstraAlgorithmBackend`. Dostarcza pojedynczą metodę `run` wykonującą algorytm i zwracającą jego wynik.
- klasa `DijkstraSerial` z komponentu `DijkstraSerial` - zawiera sekwencyjną implementację algorytmu Dijkstry. Korzysta z API udostępnionego przez klasę `DijkstraAlgorithmBackend`. Dostarcza pojedynczą metodę `run` wykonującą algorytm i zwracającą jego wynik.
- klasa `AdjacencyMatrix` z komponentu `DijkstraCommon` - pozwala na wczytanie i przechowywanie danych wejściowych
- klasa szablonowa `Log` - udostępnia opcję wypisywania na standardowe wyjście informacji na temat działania algorytmu. Posiada specjalizację `Log<false>`, które wyłącza mechanizm wypisywania informacji. Implementacja wypisywania oparta jest na tzw. *variadic templates* z C++11.

3 Działanie projektu

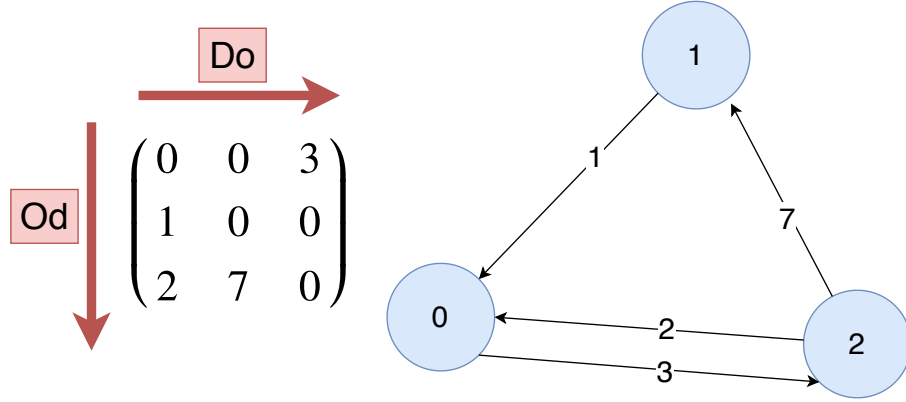
W tej części dokumentacji omówiony zostanie sposób działania implementacji algorytmu Dijkstry wykorzystującej protokół MPI. Algorytm składa się z trzech głównych części:

- inicjalizacja - podział danych wejściowych na części, walidacja, rozesłanie danych do poszczególnych procesów
- obliczenie odległości i ścieżek (algorytm Dijkstry)
- zapisanie wyników do pliku i zakończenie działania programu

3.1 Inicjalizacja

Pierwsza część programu składa się z kilku mniejszych etapów. Na początku ma miejsce standardowa inicjalizacja protokołu MPI oraz przypisanie każdemu z procesów rangi. Kolejnym etapem jest utworzenie klasy odpowiedzialnej za wyświetlanie informacji o przebiegu programu na standardowe wyjście. W dalszej kolejności przetwarzane i walidowane są dane wejściowe - numer wierzchołka stanowiącego źródło oraz ścieżka do pliku z danymi wejściowymi. W przypadku, gdy dane są niepoprawne, program kończy działanie wyświetlając odpowiedni komunikat.

Następnie odbywa się przede wszystkim podział danych wejściowych na części i rozesłanie ich do poszczególnych procesów. Program przyjmuje informacje na temat przetwarzanego grafu w postaci tzw. **macierzy sąsiedztwa**. W i -tym rzędzie i j -tej kolumnie takiej macierzy znajduje się waga przypisana do krawędzi łączącej wierzchołek i z wierzchołkiem j . W przypadku braku krawędzi waga wynosi 0. Rysunek 2 przedstawia przykład tego typu reprezentacji wraz z odpowiadającym jej grafem.



Rysunek 2: Przykładowa macierz sąsiedztwa wraz z odpowiadającym jej grafem skierowanym.

Macierz sąsiedztwa wczytywana jest z pliku z danymi wejściowymi przez proces główny (proces `root` o identyfikatorze 0). Proces ten dokonuje podziału macierzy na części zgodnie z ilością procesów biorących udział w wykonaniu algorytmu. Każdy z procesów otrzymuje k kolumn oryginalnej macierzy w formie ciągłego obszaru pamięci - przykładowy podział dla grafu o 5 wierzchołkach i 3 procesów przedstawia rysunek 3.

Algorytm podziału jest prosty. Dana jest macierz o n kolumnach i m procesów. Najpierw każdemu z procesów przypisywane jest:

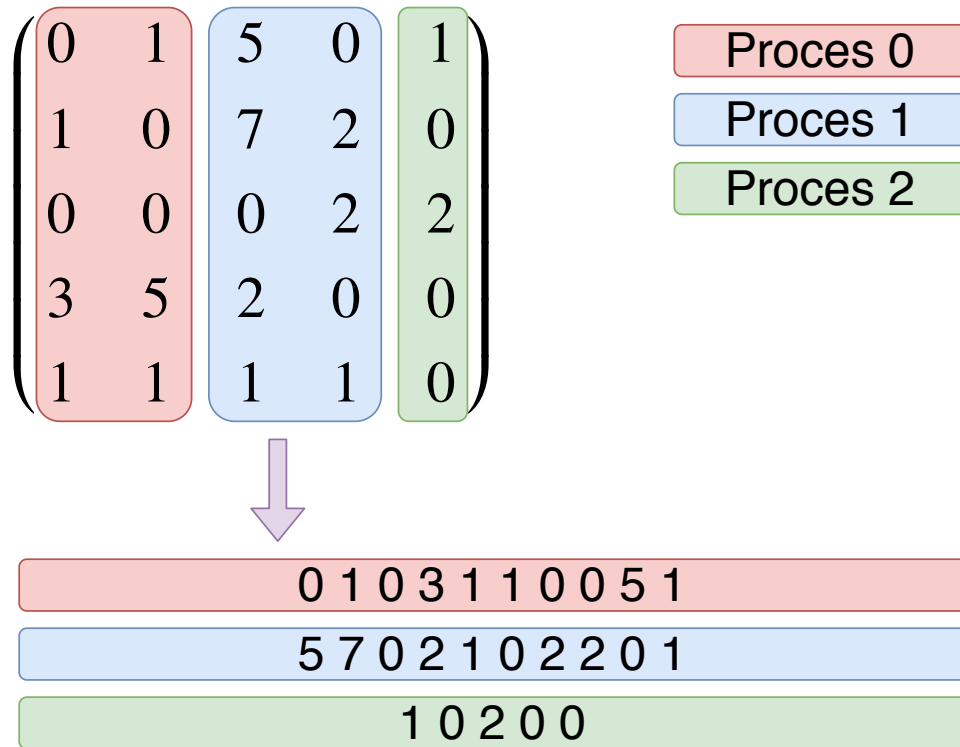
$$a = \left\lfloor \frac{n}{m} \right\rfloor \quad (1)$$

kolumn - w przypadku $n = 5$ i $m = 3$ każdy proces otrzymuje domyślnie jedną kolumnę. Ilość pozostałych kolumn dana jest zależnością:

$$k = n \bmod m \quad (2)$$

W analizowanym przykładzie jest to $k = 2$. Kolumny te rozdzielane są po jednej pomiędzy k pierwszych (zgodnie z numeracją nadaną przez MPI)

procesów. Ostatecznie w omawianym przykładzie proces zerowy i pierwszy otrzymują po dwie kolumny, proces drugi otrzymuje natomiast tylko jedną kolumnę.



Rysunek 3: Przykładowy podział macierzy sąsiedztwa reprezentującej graf o 5 wierzchołkach. W wykonaniu algorytmu biorą udział 3 procesy.

Kolumny przypisywane są każdemu procesowi po kolei, tzn. proces 0 otrzymuje x_0 pierwszych kolumn macierzy, proces 1 kolejne x_1 itd. Zbiór kolumn reprezentowany jest za pomocą jednowymiarowej tablicy - ułożone są w niej kolejno dane z pierwszej, drugiej itd. kolumny.

Po dokonaniu przez proces `root` podziału, do każdego z procesów wysyłane są trzy informacje:

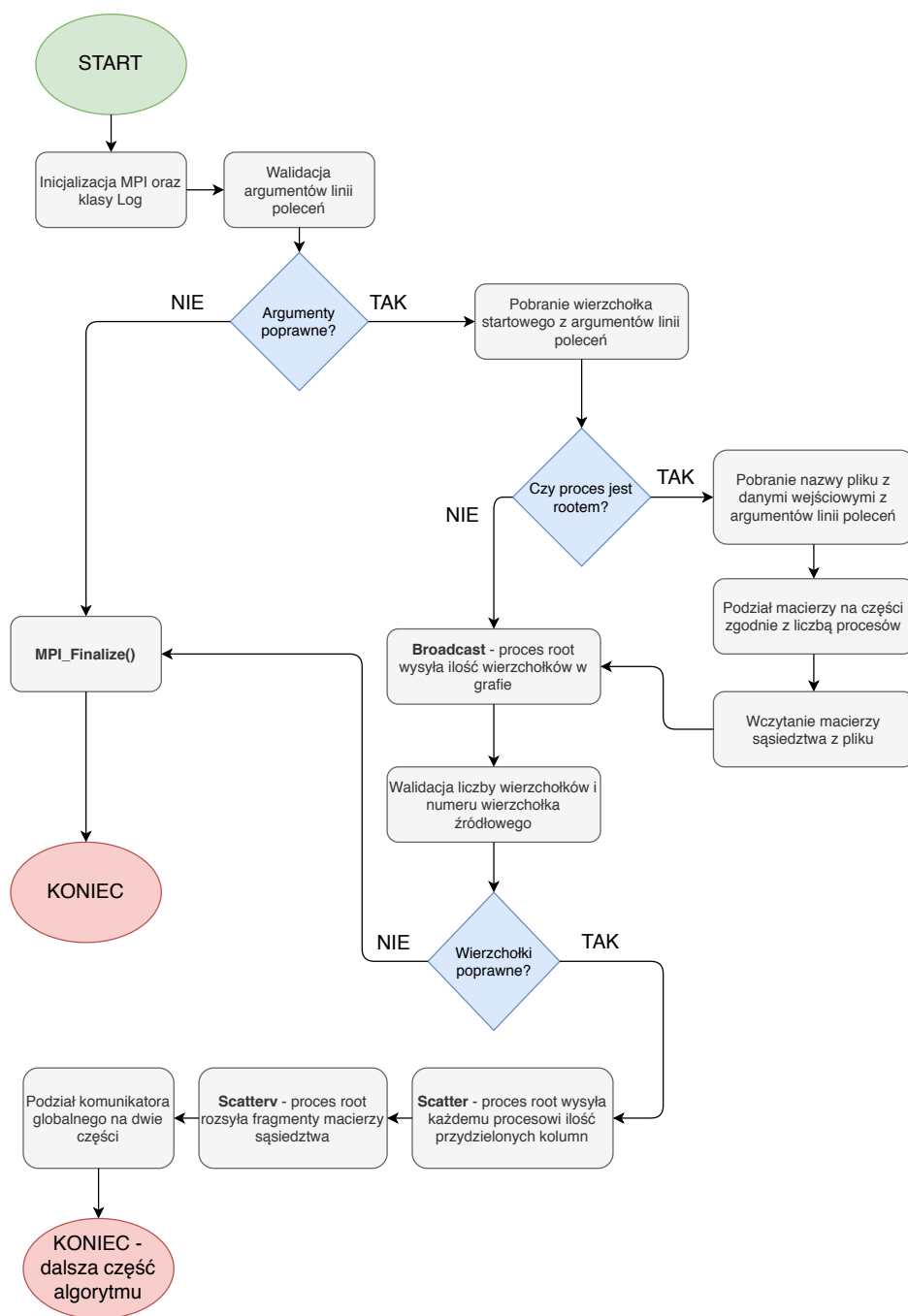
- całkowita liczba wierzchołków w przetwarzanym grafie - wysyłanie odbywa się za pomocą operacji kolektywnej `MPI_Bcast`
- ilość kolumn macierzy sąsiedztwa, jaka każdy z procesów będzie musiał obsłużyć. Informacja ta jest w tym momencie potrzebna, ponieważ każdy z procesów musi przygotować sobie odpowiednio duży bufor na dane z samej macierzy. Wysyłanie tej informacji odbywa się za pomocą operacji `MPI_Scatter`, ponieważ każdy z procesów otrzymuje swoją indywidualną informację.
- zawartość kolumn macierzy sąsiedztwa, które będą przetwarzane przez dany proces. Wysyłanie tej informacji odbywa się za pomocą funkcji `MPI_Scatterv`, ponieważ jest ona w stanie obsłużyć wysyłanie innej ilości danych do każdego z procesów.

Po zakończeniu wysyłania danych następuje podział procesów na dwie grupy: procesy które otrzymały przynajmniej jedną kolumnę oraz takie, które nie otrzymały żadnej (ma to miejsce, gdy do wykonania algorytmu przydzielone zostaje więcej procesów, niż przetwarzany graf posiada wierzchołków). W tym celu za pomocą operacji `MPI_Comm_split` komunikator globalny dzielony jest na dwa komunikatory. Kryterium podziału stanowi tutaj właśnie sprawdzenie, czy procesowi zostały przydzielone kolumny macierzy do obsługi (`numberOfColumnsToHandle > 0`). Pozwala to wykluczyć nadmiarowe procesy z dalszego przebiegu algorytmu.

Przebieg procesu inicjalizacji jest komunikowany użytkownikowi - na standardowym wyjściu pojawiają się odpowiednie informacje. Poniżej zamieszczono przykład takich informacji wygenerowany dla przykładu widocznego na rysunku 3:

```
[Process 0] This process will handle 2 vertices in range [0 , 1]
[Process 1] This process will handle 2 vertices in range [2 , 3]
[Process 2] This process will handle 1 vertices in range [4 , 4]
...
```

Rysunek 4 przedstawia **schemat blokowy** ilustrujący przebieg procesu inicjalizacji.



Rysunek 4: Schemat blokowy ilustrujący przebieg procesu inicjalizacji.

3.2 Implementacja i zakończenie algorytmu

Początek właściwej części programu stanowi sprawdzenie, czy danemu procesowi przydzielone zostały wierzchołki grafu (czyli kolumny macierzy sąsiedztwa) do przetworzenia. Jeśli nie, to taki proces nie bierze udziału w dalszym wykonaniu algorytmu - następuje dealokacja utworzonego komunikatora i zakończenie działania. W przeciwnym wypadku proces przystępuje do realizacji algorytmu.

Implementacja algorytmu Dijkstry za pomocą protokołu MPI nie różni się w sposób znaczący od klasycznej implementacji szeregowej. Każdy z procesów obsługuje dwie tablice:

- tablica odległości (kosztów) od wierzchołka źródłowego do każdego z wierzchołków w grafie
- tablica poprzedników każdego z wierzchołków - służy do odtworzenia najkrótszych ścieżek między wierzchołkami

oraz zbiór obsługiwanych wierzchołków (często nazywany klastrem). Początkowo zbiór ten jest pusty, tablica kosztów zainicjalizowana jest nieskończonościami, a tablica poprzedników wartościami -1. Różnica w stosunku to szeregowej wersji algorytmu jest taka, że w przypadku implementacji równoległej każdy z procesów przechowuje **fragmenty tych tablic** odpowiadające przydzielonym im wierzchołkom grafu.

W procesie obsługującym wierzchołek źródłowy jego wpis w tablicy kosztów ustawiany jest na wartość 0. Następnie, dopóki wszystkie wierzchołki nie zostały przetworzone, wykonywany jest w pętli algorytm:

- każdy proces wybiera spośród przydzielonych mu wierzchołków taki, który nie został jeszcze dodany do zbioru wierzchołków obsługiwanych i którego wartość w tablicy kosztów jest najmniejsza
- za pomocą operacji `MPI_Allreduce` wybierany jest wierzchołek o globalnie najniższej wartości kosztu (czyli najlepszy spośród najlepszych z każdego procesu)
- jeśli nie został wyznaczony taki wierzchołek, to następuje wyjście z pętli i zakończenie algorytmu

- w przeciwnym razie wierzchołek zostaje dodany do zbioru wierzchołków przetworzonych (w każdym procesie zbiór ten jest przechowywany osobno, w całości)
- dla każdego nieprzetworzonego sąsiada wyznaczonego wierzchołka dokonywana jest aktualizacja w tablicy kosztów i poprzedników (wybrany wierzchołek staje się poprzednikiem swoich sąsiadów), jeśli nowy koszt okazuje się niższy od dotychczasowego

Po zakończeniu działania algorytmu fragmenty tablic kosztów i poprzedników są łączone w całość za pomocą operacji `MPI_Gatherv`. Proces `root` wyznacza ścieżki z wierzchołka źródłowego do każdego wierzchołka w grafie i zapisuje wyniki do pliku. Następnie wyświetlany jest czas działania algorytmu. Ostatnim krokiem jest zwolnienie utworzonego wcześniej komunikatora, wywołanie operacji `MPI_Finalize` i zakończenie działania programu.

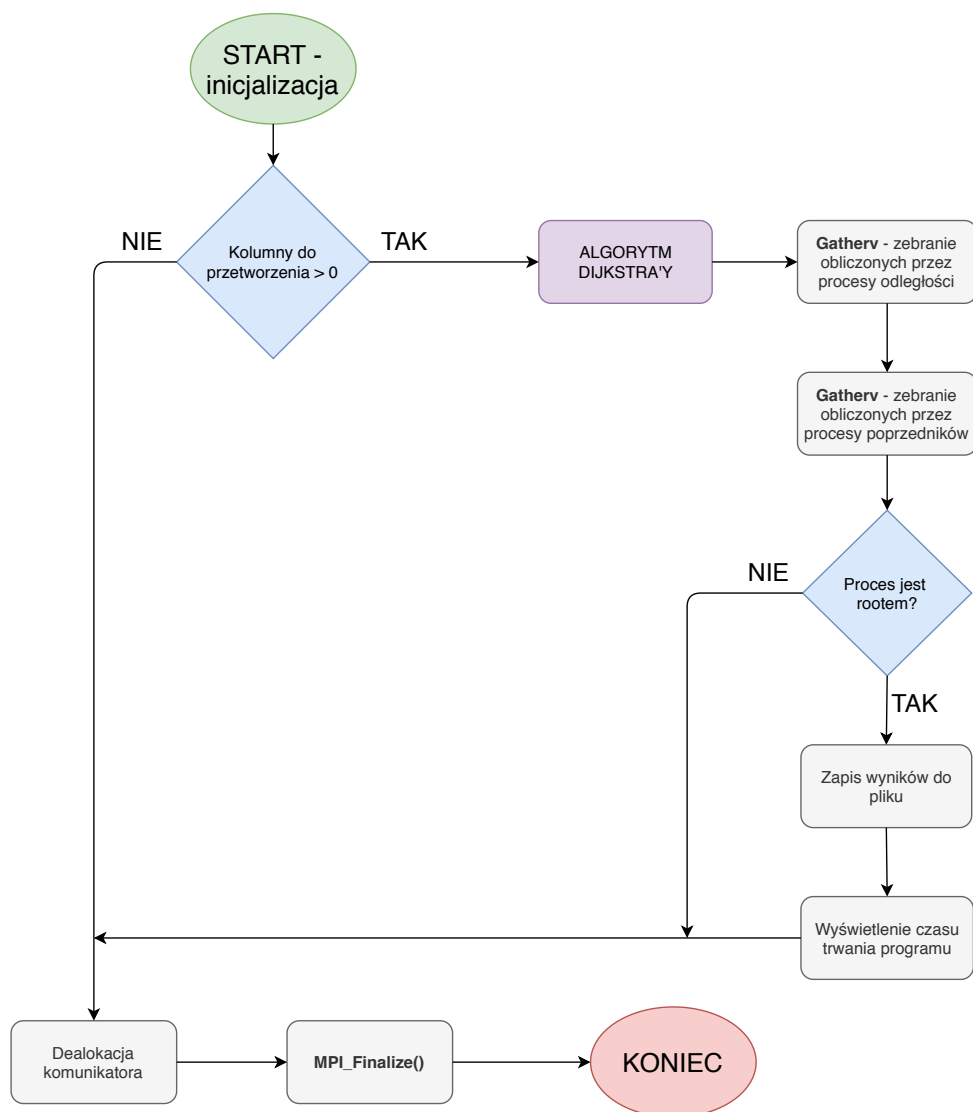
W kilku miejscach w programie tworzona jest zmienna zawierająca aktualny czas. Pod koniec działania, proces `root` zbiera te informacje i wyświetla czas wykonywania się każdej z trzech części programu:

```
[Process 0] Total elapsed time: 0.0298846s
[Process 0] Setup took: 0.0260188s
[Process 0] Algorithm took: 0.0020106s
[Process 0] Printing solution took: 0.0018552s
```

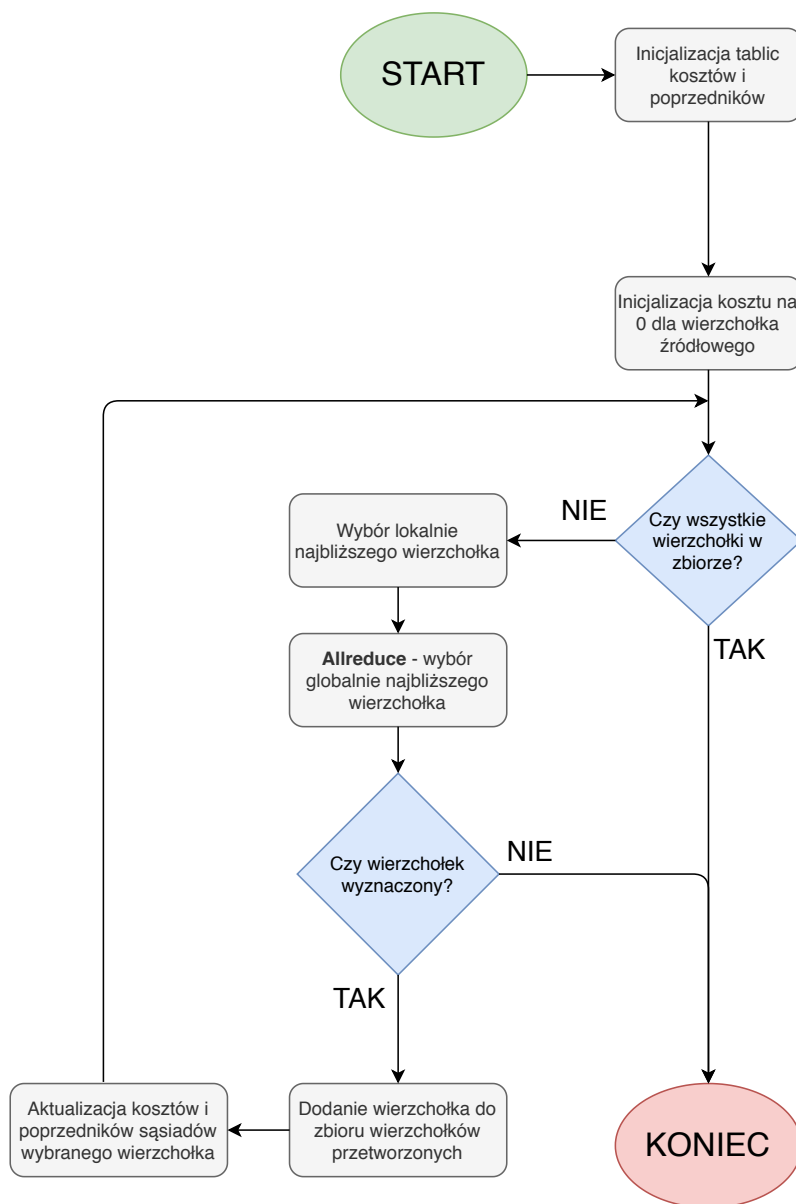
Wyświetlane są wyniki pomiarów dokonanych jedynie w procesie głównym - wyniki z pozostałych procesów są pomijane. Pomiar wykonywany jest z użyciem obiektu klasy `high_resolution_clock` z biblioteki `chrono` (standard C++11). Użycie takiego zegara gwarantuje, że mierzony czas jest czasem rzeczywistym, a nie czasem procesora.

Rysunek 5 przedstawia schemat blokowy ilustrujący przebieg działania algorytmu po inicjalizacji. Kolorem fioletowym oznaczony został blok związany z główną pętlą algorytmu Dijkstry. Schemat ten nie zawiera szczegółów jej działania.

Rysunek 6 przedstawia schemat blokowy ilustrujący działania głównej pętli algorytmu Dijkstry. Schemat ten stanowi rozwinięcie fioletowego bloku z rysunku 5.



Rysunek 5: Schemat blokowy ilustrujący przebieg głównej części programu wraz z zapisaniem wyników i zakończeniem. Schemat nie zawiera implementacji algorytmu Dijkstry.



Rysunek 6: Schemat blokowy ilustrujący przebieg głównej pętli algorytmu Dijkstry.

3.3 Zastosowane funkcjonalności MPI

W ramach projektu zastosowane zostały następujące funkcjonalności MPI:

- operacja `MPI_Bcast` - w celu rozesłania wszystkim procesom informacji o całkowitej ilości wierzchołków w przetwarzanym grafie
- operacja `MPI_Scatter` - w celu wysłania każdemu z procesów liczby przypisanych mu do obsłużenia wierzchołków grafu
- operacja `MPI_Scatterv` - celu wysłania każdemu z procesów danych z macierzy sąsiedztwa. Wymagana jest w tym przypadku możliwość wysłania każdemu z procesów danych o różnej długości.
- operacja `MPI_Allreduce` - w celu wyznaczenia wierzchołka o najmniejszej wartości kosztu
- operacja `MPI_Gatherv` - w celu zebrania wyników wyprodukowanych przez każdy z procesów. W tym przypadku wymagana jest możliwość odbioru od każdego z procesów danych i innej długości
- operacje `MPI_Comm_split` oraz `MPI_Comm_free` - w celu utworzenia i zwolnienia nowego komunikatora. Pozwala to wykluczyć bierne (nadmiarowe) procesy z udziału w wykonywaniu algorytmu.

4 Testy projektu

TODO:Wtorek

5 Obsługa programu

Niniejsza część dokumentacji przedstawia informacje związane z kompilacją i uruchomieniem projektu.

5.1 Kompilacja

Do kompilacji projektu został przygotowany plik `CMakeLists.txt`. Za jego pomocą możliwe jest wygenerowanie pliku `Makefile` służącego do kompilacji. Preferowanym sposobem użycia tego pliku jest tzw. użycie *out-of-source* z katalogu `build`. Znajdując się w katalogu głównym projektu należy wykonać następującą operację:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Aby kompilacja zakończyła się powodzeniem wymagane jest posiadanie zainstalowanej biblioteki `MPI`. Na pracowni `WFiS` konieczne jest ponadto załadowanie odpowiednich zmiennych środowiskowych przed kompilacją za pomocą polecenia:

```
$ source /opt/nfs/config/source_mpich32.sh
```

W dalszej części przydatny może się również okazać plik zawierający informacje o dostępnych węzłach. Na pracowni generuje się go za pomocą poniższej komendy:

```
$ /opt/nfs/config/station_name_list.sh 201 216 > nodes
```

5.2 Uruchomienie

Aby uruchomić program użytkownik może skorzystać z opcji udostępnianych przez wygenerowany plik `Makefile`:

- `make runMPI` - uruchomiona zostaje równoległa wersja programu,
- `make runSerial` - uruchomiona zostaje sekwencyjna wersja programu.

Można skorzystać z domyślnych parametrów uruchomienia, lub podać własne:

- VERTEX=V - jako wierzchołek startowy zostanie wybrany wierzchołek V, domyślną wartością jest 0,
- FILE=F - jako plik wejściowy zostanie użyty plik F, plik domyślny to `../data/graph.dat`
- N=N - parametr `mpirun` ustalający ilość procesów, wartość domyślna jest równa 1,
- HOSTS=H - parametr `mpirun`, plik wejściowy z węzłami.

Przykładowe uruchomienie programu może wyglądać następująco:

```
$ make runMPI VERTEX=1 FILE="../data/graph.dat" N=5
```

Jeśli użytkownik nie chce skorzystać z udostępnionych opcji można również uruchomić pliki wykonywalne samodzielnie. Po wykonaniu komendy `make install-all` zostaną one udostępnione w katalogu z którego prowadzona była kompilacja. Jako argument należy podać kolejno numer wierzchołka startowego oraz plik wsadowy. Pierwszy argument jest obligatoryjny. Może to wyglądać np. tak jak poniżej:

```
$ make install-all  
$ mpirun -n 5 ./DijkstraMPI 1 "../data/graph.dat"
```

Po uruchomieniu programu, jeśli zostało włączone logowanie, zostaną wyświetlone informacje na temat czasu wykonania. Znalezione ścieżki są zapisywane do plików: odpowiednio `resultMPI.txt` oraz `resultSerial.txt`.

5.3 Dodatkowe opcje

Korzystając z udostępnionego `CMakeLists.txt` można również włączyć dodatkowe opcje, takie jak:

- `-DBUILD_DOC=ON|OFF` generowanie dokumentacji `Doxygen`, domyślnie ta opcja jest wyłączona,
- `-DSHOULD_LOG=TRUE|FALSE` włączenie/wyłączenie generowania logów.

Dostępne są również dodatkowe opcje programu `make`, takie jak:

- `make clean-all` - przywrócenie zawartości podkatalogu do stanu wyjściowego,

- `make install-all` - udostępnienie plików wykonywalnych w podkatalogu,
- `make doc` - wygenerowanie dokumentacji Doxygen,
- `make runMPI` - uruchomienie programu w wersji równoległej, opisane w punkcie 5.2,
- `make runSerial` - uruchomienie programu w wersji sekwencyjnej, opisane w punkcie 5.2,

5.4 Dane wejściowe

Program jako jedną z danych wejściowych przyjmuje plik z grafem zapisanym w postaci macierzy sąsiedztwa. W pierwszej linii pliku powinna znajdować się ilość wierzchołków.

```
5
0.0000  0.0000  0.0000  8.2700  0.0000
4.2000  0.0000  8.5200  0.0000  0.0000
3.0700  0.0000  0.0000  0.0000  7.7700
0.0000  0.0000  7.4900  0.0000  0.0000
9.5700  6.9400  7.7900  6.6900  0.0000
```

Listing 2: Przykładowy graf zapisany w odpowiednim formacie.

Do wygenerowania pliku w takim formacie można użyć skryptu pomocniczego `generateGraphs.py` opisanego w sekcji 5.5.

5.5 Generowanie grafów

W ramach projektu został udostępniony dodatkowy skrypt `generateGraphs.py`. Służy on do generowania grafów skierowanych o wybranej przez użytkownika liczbie wierzchołków i krawędzi. Znajduje się on w katalogu `\data`. Dostępne są opcje:

- `-h` - pomoc skryptu,
- `-e` - liczba krawędzi w grafie, jeśli wartość nie została podana, ustawiana jest wartość domyślna = 10,
- `-v` - liczba wierzchołków w grafie, wartość domyślna = 10,

- `-f` - nazwa pliku wyjściowego, domyślna nazwa to `graph.dat`.

Jeśli wartość krawędzi jest zbyt duża dla danej liczby wierzchołków, użytkownik zostaje poinformowany i zostaje ustawiona nowa, losowa wartość. Wagi krawędzi są nieujemnymi liczbami rzeczywistymi, z zakresu $[0, 10]$.

```
generateGraphs.py [-h] [-e E] [-v V] [-f FILE]

optional arguments:
  -h, --help            show this help message and exit
  -e E, --edges E       number of edges
  -v V, --vertices V    number of vertices
  -f FILE, --file FILE  name of output file
```

Listing 3: Pomoc skryptu `generateGraphs.py`.

Grafy wygenerowane przy pomocy tego skryptu mogą zostać użyte jako dane wejściowe do programu.

Literatura

- [1] Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar *Introduction to Parallel Computing, Second Edition*. Addison-Wesley, 2003.
- [2] Aditya Pore, Russ Miller *Parallel implementation of Dijkstra's algorithm using MPI library on a cluster*. Link: <https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Pore-Spring-2014-CSE633.pdf> (dostęp: 21.04.2020)
- [3] Zilong Ye *An Implementation of Parallelizing Dijkstra's Algorithm*. Link: <https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ye-Fall-2012-CSE633.pdf> (dostęp: 21.04.2020)
- [4] Wikipedia, The Free Encyclopedia *Dijkstra's algorithm* Link: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm (dostęp: 21.04.2020)
- [5] Dokumentacja MPICH: <https://www.mpich.org/documentation/guides/> (dostęp: 21.04.2020)
- [6] Dokumentacja Microsoft MPI: <https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi> (dostęp: 21.04.2020)
- [7] Wes Kendall *MPI Tutorial: MPI Scatter, Gather, and Allgather*. Link: <https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/> (dostęp: 21.04.2020)
- [8] Dokumentacja języka C++ dostępna na stronie <https://en.cppreference.com/w/> (dostęp: 21.04.2020)