Minimalne drzewo rozpinające Algorytm Prim'a z użyciem MPI.

Ryszard Borzych, Wiktor Urban

May 3, 2023

Abstract

Program służy do znalezienia minimalnego drzewa rozpinające i wyliczania jego wagi z użyciem algorytmu Prima w sposób rozproszony dzięki bibliotece MPI.

1 Obsługa programu

Definiowanie wierzchołków odbywa się poprzez plik `Matrix.txt`. Zapisana jest w nim macierz sąsiedztwa. Kolejne liczby to wagi połączeń między wierzchołkami. Przykładowo w komórce macierzy `i,j` znajduje się waga połączenia pomiędzy węzłami `i` oraz `j`. W przypadku braku połączenia program oczekuje wartości -1.

Plik `Matrix.txt` można wygenerować przy pomocy dołączonego generatora `generate_matrix.py` napisanego w języku Python. Uruchomienie go wymaga zainstalowania modułu networkx poprzez `pip install networkx`. W pliku `generate_matrix.py` zdefiniowane `min_weight` i `max_weight`, określają zakres z którego będą losowane wagi. Liczba `n` określa liczbę wierzchołków w grafie. Ten skrypt pythonowy generuje losowe grafy dopóki nie wylosuje grafu połączonego (czyli nie las). Następnie nadaje krawędziom losowe wagi ze zdefiniowanego na górze pliku zakresu.

Generator grafu był testowany przy pomocy Python 3.10.6, a pakiet networkx w wersji 3.1 .

Po uruchomieniu generatora poprzez `python ./generate_matrix.py` zostaje utworzony plik `Matrix.txt`.

Żeby główny program działał w pracowni nr 204 konieczne dodanie zmiennych środowiskowych. Można to zrobić poprzez następującą komendę: `source /opt/nfs/config/source_mpich401.sh`

Następnie by uruchomić program należy wykonać polecenie `make` w folderze projektu.

2 Opis kodu

2.1 Inicjacja

W tej części inicjowane jest środowisko MPI. Procesy dowiadują się jaka jest liczność procesów oraz który mają rank. Tworzony jest ponadto typ danych `EdgeAndNodes_MPI` który będzie użyty później.

```
int main(int argc, char *argv[]) {

    int rank, process_count;

    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);

    char process_name[MPI_MAX_PROCESSOR_NAME]; // MPI_MAX_PROCESSOR_NAME = 128
    int name_len;
    MPI_Get_processor_name(process_name, &name_len);
    if(name_len >= MPI_MAX_PROCESSOR_NAME){
        printf("Process name has overflown the reserved space for process name string.
     Reserved space: %d, process name length: %d\n", MPI_MAX_PROCESSOR_NAME, name_len);
    }
    printf("hello from process %s, rank %d out of %d processes\n", process_name, rank,
     process_count);
```

```
    MPI_Barrier(MPI_COMM_WORLD);

    // this is three int continuous array, 0-element is edge weight, 1-element is
    start_node id, 2-element is end_node-id
    MPI_Datatype EdgeAndNodes_MPI;
    MPI_Type_contiguous(3, MPI_INT, &EdgeAndNodes_MPI);
    MPI_Type_commit(&EdgeAndNodes_MPI);
```

Następnie proces o rank równym 0 odczytuje macierz sąsiedztwa z pliku oraz synchronizuje pomiędzy procesy liczbę węzłów grafu przechowywaną w zmiennej `global_row_count`. Wczytaną macierz wpisuje do pamięci w zmiennej `full_adjacency_matrix`. Pamięć ta jest ciągła, ponieważ później użyjemy `MPI_Scatterv`, który wymaga ciągłego obszaru pamięci.

```
    int *full_adjacency_matrix = NULL;

    int global_row_count = 0;
    if(rank == 0){
        printf("process 0 reads the adjacency_matrix_from_file\n");

        global_row_count = read_adjacency_matrix_from_input_file("Matrix.txt", &
    full_adjacency_matrix);

        print_adjacency_matrix(full_adjacency_matrix, global_row_count);
    }
    MPI_Bcast(&global_row_count, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

W kolejnym kroku obliczana jest ilość wierszy macierzy jaka ma zostać przydzielona do każdego z procesów. Ponieważ ilość wierszy macierzy może być niepodzielna przez liczbę procesów, jest to nietrywialne. Macierz przeczytana z pliku zostaje rozproszona na procesy poprzez `MPI_Scatterv`.

```
    int local_row_count = get_local_row_count(global_row_count, process_count, rank);
    printf("local_row_count array in process %d: %d\n", rank, local_row_count);

    // Allocate memory for the local matrix
    int * local_matrix = (int*) calloc(local_row_count * global_row_count, sizeof(int)
    );
    int * send_counts = (int *)calloc(process_count, sizeof(int));
    int * send_displacements = (int *)calloc(process_count, sizeof(int));
    calculate_send_counts_and_send_displacements(global_row_count, process_count,
    send_counts, send_displacements);

    MPI_Scatterv(full_adjacency_matrix, send_counts, send_displacements, MPI_INT,
    local_matrix, local_row_count * global_row_count, MPI_INT, 0, MPI_COMM_WORLD);
```

Tworzona jest tablica `are_nodes_included_in_mst` która zawiera informację o tym czy dany węzeł został już dodany do drzewa rozpinającego czy nie. Dodatkowo tworzona jest tablica struktur do której proces o rank $= 0$ będzie zapisywał kolejne krawędzie oraz wagi drzewa rozpinającego. Ponieważ drzewo musi się od czegoś zacząć wybieramy węzeł numer 0 jako początkowy węzeł.

```
  bool *are_nodes_included_in_mst = calloc(global_row_count, sizeof(bool));

  // this struct will be used by process with rank 0 to keep history about edges and
    nodes that were connected.
  // This will be later used for printing the result.
  // minimal spanning tree has n -1 edges where n = amount of graph nodes
  struct EdgeAndNodes *edge_and_nodes = NULL;
  if(rank == 0){
    edge_and_nodes = malloc( (global_row_count -1) * sizeof(struct EdgeAndNodes));
  }
  // choose first node as starting node
  are_nodes_included_in_mst[0] = true;
```

2.2 Pętla w której znajdowana jest nowa krawędź drzewa MST

Poniższy kod wykonuje się wewnątrz pętli `for(int k = 0; k < global_row_count -1; k++)`

Na początku pętli Na początku każdej z iteracji `k`, tworzymy zmienne określające wagę oraz id węzłów krawędzi.

```
        int local_minimal_edge_weight = MY_INFINITY;
        // nodes connected by the edge with local_minimal_edge_weight
        int global_id_of_start_node = 0;
        int global_id_of_end_node = 0;
```

Każdy z procesów szuka krawędzi o minimalnej wadze wśród przydzielonych do niego wierszy macierzy. W pętli zewnętrznej z iteratorem `i` wybierany jest węzeł początkowy od którego będzie wychodzić krawędź. Ten węzeł wybierany jest spośród węzłów już będących częścią MST. W pętli wewnętrznej wybieramy węzeł końcowy który nie może być częścią MST. Każdą krawędź łączącą pasujące pary węzłów sprawdzamy pod kątem tego czy ma mniejszą wagę od najlepszej wcześniej znalezionej krawędzi.

```
        for(int i = 0; i < local_row_count; i++){ // for every local row

          int global_row_id = i + send_displacements[rank] / global_row_count;
          global_row_id, i, k);
          bool is_node_included_in_mst = are_nodes_included_in_mst[global_row_id];
          if(is_node_included_in_mst == false) continue;

          // we check each edge to find minimal weight
          for(int j = 0; j < global_row_count; j++){ // for each column
            if(i == j) continue;

            // we ignore the edges to nodes which are already included in minimal
    spanning tree
            bool is_compared_node_included_in_mst = are_nodes_included_in_mst[j];
            is_compared_node_included_in_mst);
            if(is_compared_node_included_in_mst) continue;

            int current_edge_weight = local_matrix[i * global_row_count + j];
            local_minimum_edge_weight = %d \n", rank, i, j, k, current_edge_weight,
    local_minimal_edge_weight);
            local_minimal_edge_weight is %d\t", rank, local_minimal_edge_weight);

            if(current_edge_weight < local_minimal_edge_weight){
              current_edge_weight, i, j, k);
              local_minimal_edge_weight = current_edge_weight;
              global_id_of_start_node = global_row_id;
              global_id_of_end_node = j;
            }
          }
        }
```

Wybieranie globalnie optymalnej krawędzi. Każdy z procesów przesyła swoją lokalnie optymalną krawędź do procesu o rank $= 0$. Proces ten wybiera globalnie optymalną krawędź, a następnie aktualizuje tablicę struktur `edge_and_nodes` oraz `are_nodes_included_in_mst`, a następnie synchronizuje `are_nodes_included_in_mst` z pozostałymi procesami.

```
    if(rank != 0){
      int edge_and_nodes_to_send[3] = {local_minimal_edge_weight,
    global_id_of_start_node, global_id_of_end_node};
      MPI_Send(edge_and_nodes_to_send, 3, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    if(rank == 0){
      int globally_minimal_weight = local_minimal_edge_weight;
      int start_node = global_id_of_start_node;
      int end_node = global_id_of_end_node;

      int received_counter = 0;
      while(received_counter < process_count -1){
        int received_edge_and_nodes[3] = {0, 0, 0};

        MPI_Status mpi_status;
        MPI_Recv(received_edge_and_nodes, 3, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
    MPI_COMM_WORLD, &mpi_status);
        if(received_edge_and_nodes[0] < globally_minimal_weight){
          globally_minimal_weight = received_edge_and_nodes[0];
```

```
          start_node = received_edge_and_nodes[1];
          end_node = received_edge_and_nodes[2];
        }
        received_counter += 1;
      }

      printf("globally minimal edge weight: %d of edge connecting nodes: %d and %d\n",
     globally_minimal_weight, start_node, end_node);
      are_nodes_included_in_mst[end_node] = true;
      edge_and_nodes[k].edge_weight = globally_minimal_weight;
      edge_and_nodes[k].start_node_id = start_node;
      edge_and_nodes[k].end_node_id = end_node;

    }
    MPI_Barrier(MPI_COMM_WORLD);

    // synchronization of mst
    MPI_Bcast(are_nodes_included_in_mst, global_row_count, MPI_C_BOOL, 0,
    MPI_COMM_WORLD);
```

2.3 Finalizacja

Po znalezieniu minimalnego drzewa rozpinającego program wypisuje łączną wagę drzewa oraz krawędzie tworzące drzewo. Ponadto należy zwolnić zaalokowaną pamięć oraz zakończyć połączenia MPI.

```
if(rank == 0){

  int total_weight = 0;
  for(int i = 0; i < global_row_count -1; i++){
    printf("edge: %d  -  %d with weight: %d\n", edge_and_nodes[i].start_node_id,
  edge_and_nodes[i].end_node_id, edge_and_nodes[i].edge_weight);
    total_weight += edge_and_nodes[i].edge_weight;
  }
  printf("cumulative weight of the minimal spanning tree: %d\n", total_weight);

  printf("\n");
}
MPI_Barrier(MPI_COMM_WORLD);

printf("cleaning up allocated memory\n");
free(are_nodes_included_in_mst);
free(send_counts);
free(send_displacements);
free(local_matrix);
if(rank == 0){
  free(edge_and_nodes);
  free(full_adjacency_matrix);
}
printf("finished cleaning up allocated memory\n");

MPI_Type_free(&EdgeAndNodes_MPI);

// printf("before MPI_Finalize\n");
MPI_Finalize();
```

2.4 Wyniki

Oto wyniki dla przykładowego grafu o 6 węzłach.

```
 6
-1 -1 12 10 45 11
-1 -1 47 18 15 36
12 47 -1 34 27 -1
10 18 34 -1 16 16
45 15 27 16 -1 25
11 36 -1 16 25 -1

edge: 0  -  3 with weight: 10
edge: 0  -  5 with weight: 11
edge: 0  -  2 with weight: 12
edge: 3  -  4 with weight: 16
edge: 4  -  1 with weight: 15
cumulative weight of the minimal spanning tree: 64
```

W celach weryfikacji można użyć strony https://visualgo.net/en/mst

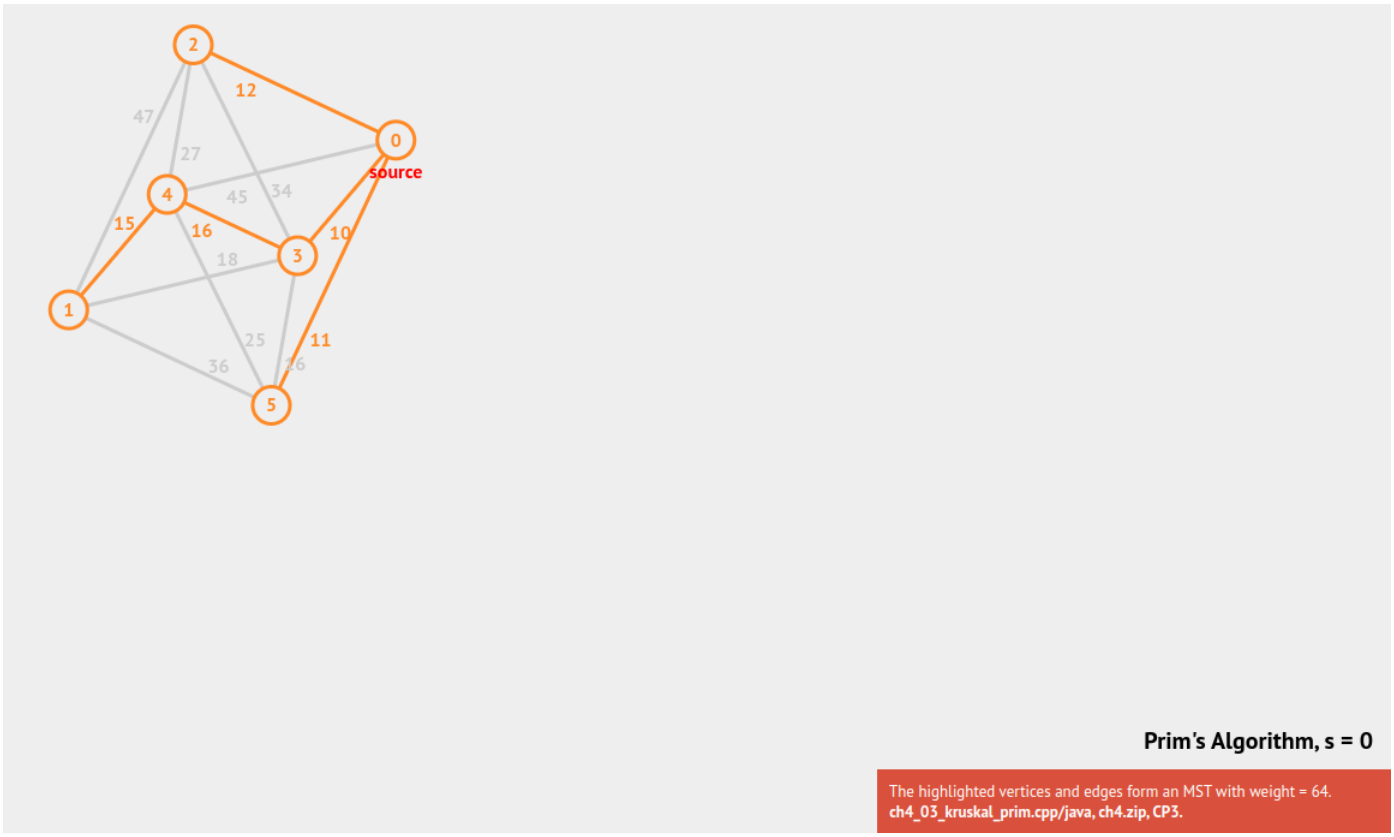

Figure 1: Wynik otrzymany ze strony https://visualgo.net/en/mst zgadza się z wynikiem naszego programu.