Implementation and parallelization of Dijkstra's algorithm

IN3200/IN4200 obligatory assignment 1, Spring 2024

Note: This is one of the two obligatory assignments that both need to be approved before a student is allowed to take the final exam. (The grade of the final exam is otherwise **not** related to these two assignments.)

Note: Discussions between the students are allowed, but each student should write her/his own implementations. The details about how to submit the assignment can be found at the end of this document.

1 Objectives

This obligatory assignment has the following objectives:

- The student is given a hands-on exercise of implementing a prescribed algorithm.
- The student is asked to implement the needed C functions by strictly following a pre-defined syntax of each function. (This is an important skill in writing modular scientific software.)
- The student is asked to learn about dividing an entire code into multiple program files and about compiling a multi-file code.
- The student is given a hands-on exercise of using a sparsity-oriented data structure instead of a dense regular data structure for performance benefits.
- The student is given a hands-on exercise of simple OpenMP parallelization.
- The student is given a hands-on exercise of measuring the time usage of some parts of a code.

2 Introduction

2.1 Directed graphs

Mathematically, a so-called directed graph consists of a set of vertices and a set of edges, where each edge symbolizes a *directional* connection between two vertices. (The overall vertex connectivity of a directed graph may not be symmetric. That is, if vertex i is directionally connected to vertex j, the opposite connection may not exist.)

Figure 1 shows a simple directed graph with 5 vertices and 10 edges, where

- vertex S is connected to vertices A & C;
- vertex A is connected to vertices B & C;
- vertex B is connected to vertex D;
- vertex C is connected to vertices A, B & D;
- vertex D is connected to vertices S & B.

Moreover, in Figure 1, the numerical value associated with each edge represents the direct distance from its "start" vertex to its "end" vertex.

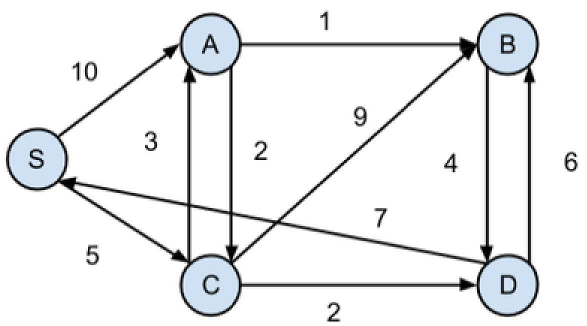

Figure 1: A very simple directed graph with 5 vertices and 10 directional edges. (Source: http://siddarthareddy.weebly.com/blog/dijkstras-algorithm-example.)

2.2 Dijkstra's algorithm

Dijkstra's algorithm (see e.g. [1, 2]) can be used to calculate the shortest distance from a prescribed "source" vertex S to all the other vertices. The shortest distance from vertex S to another vertex i is defined as the shortest path that connects S, through connected directional edges, to i. In case vertex S cannot reach vertex i through connected directional edges, then the distance from S to i is defined as ∞. Algorithm 1 shows Dijkstra's algorithm in the form of a pseudo code.

Algorithm 1 Dijkstra's algorithm for calculating the shortest distance from a source vertex to all the other vertices (with the vertex connectivity info represented by a 2D input array).

Input 1: `int n` contains the number of vertices in the graph.
Input 2: `int s` contains the index of the source vertex.
Input 3: `int** w` is a 2D array that has n rows and n columns. The `w` array contains the vertex adjacency information, where a positive value of `w[i][j]` stores the direct distance from vertex `i` to vertex `j`. If the value of `w[i][j]` is -1, it means there is no direct connection from vertex `i` to vertex `j`.
Output: `int* d` is a pre-allocated 1D array (of length n), which is to contain the shortest distance from the source vertex `s` to all the other $n-1$ vertices in the graph.

```
procedure DIJKSTRA(n, s, w, d)
begin
    Allocate a help array of length $n$ to later mark whether each vertex has been visited
    for $i = 0, 1, 2, \ldots, n-1$ do
    begin
        if w[s][i]>0 then set d[i] = w[s][i];
        else set d[i] = a large enough positive integer (to represent $\infty$);
    end
    Set d[s] = 0;
    Mark vertex s as visited, the other vertices as unvisited;

    for $i = 1, 2, \ldots, n-1$ do
    begin
        Find an unvisited vertex $u$ such that the value of d[u]
         is the minimum among all the so-far unvisited vertices;
        Mark vertex $u$ as visited;
        for each $v$ of the remaining unvisited vertices do
        begin
            if w[u][v]>0 then set d[v] = min(d[v], d[u]+w[u][v]);
        end
    end
end  DIJKSTRA
```

2.3 Example result of Dijkstra's algorithm

For the simple graph depicted in Figure 1, Dijkstra's algorithm will compute the shortest distance from source vertex S to the other four vertices as follows:

- The shortest distance from S to A is 8;
- The shortest distance from S to B is 9;
- The shortest distance from S to C is 5;
- The shortest distance from S to D is 7;

2.4 Representing the vertex connectivity info as a sparse matrix

It is very convenient to have the vertex connectivity info of a graph represented as a square dense matrix of dimension $n \times n$, as adopted in Algorithm 1, where n is the number of vertices in the graph. However, in realistic situations, most of the entries in the square matrix have value "-1", which indicates that there is no direct connection from vertex i to vertex j. A more memory-friendly storage format, which is also more performance-friendly, is to ignore all the values of "-1" and only store the non-negative values. More specifically, the non-negative values from the square dense matrix can be stored as a sparse matrix using the CRS format (see Section 3.6.1 of the textbook). **Note:** There is a typo on page 87 of the textbook. The length of array `row_ptr` should namely be $n+1$ (not n). The last value, `row_ptr[n]` (note that the index starts with 0), should store the total number of non-negative values in the square dense matrix.

3 The obligatory assignment

3.1 Four serial functions

The following four serial functions should be implemented:

1. `void read_graph_from_file_v1 (char *filename, int *n, int ***w)` This function reads a text file that contains a directed graph, so that its vertex connectivity info is returned as an $n \times n$ dense matrix named `w` (which is stored as a 2D array). If the value of `w[i][j]` is -1, then it means that vertex i has no direct connection to vertex j. **Note:** The reason for `w` using a triple pointer in the argument list is because this 2D array needs to be allocated **inside** the function, because the number of nodes, `n`, is NOT known before reading the file.

2. `void read_graph_from_file_v2 (char *filename, int *n, int **row_ptr, int **col_idx, int **val)` This function reads a text file that contains a directed graph, so that its vertex connectivity info is returned as a sparse matrix stored in the CRS format (see Section 3.6.1 of the textbook). **Note:** The reason for `row_ptr`, `col_idx` and `val` using double pointers in the argument list is because these 1D arrays need to be allocated **inside** the function, because the number of nodes and edges are NOT known before reading the file.

3. `void dijkstra_serial_v1 (int n, int s, int **w, int *d)` This function translates the pseudo code that is described in Algorithm 1. (Note: the value of `s` can be anything between 0 and `n-1`. The array `d` is supposed to be pre-allocated with length `n` before the function call.)

4. `void dijkstra_serial_v2 (int n, int s, int *row_ptr, int *col_idx, int *val, int *d)` This function also translates Dijkstra's algorithm, for which the calculated shortest distances will also be returned in the 1D array `int *d`. The difference is that the vertex connectivity info is now provided as a sparse matrix in the CRS format, that is, through `int *row_ptr, int *col_idx, int *val`. (**Note:** The array `d` is supposed to be pre-allocated with length `n` before the function call.)

3.2 OpenMP parallelization

The two serial functions `dijkstra_serial_v1` and `dijkstra_serial_v2` should be parallelized using OpenMP. That is, two OpenMP parallel functions are to be implemented:

- `void dijkstra_omp_v1 (int n, int s, int **w, int *d)`
- `void dijkstra_omp_v2 (int n, int s, int *row_ptr, int *col_idx, int *val, int *d)`

3.3 Two 'main' programs

To test the above functions, the student is required to write two 'main' programs, one (named `serial_main.c`) for testing the serial functions implemented, the other (named `omp_main.c`) for testing the OpenMP parallel counterpart.

3.4 File format of a directed graph

It can be assumed that any text file containing a directed graph has the following format:

- The first two lines both start with the `#` symbol and contain free text (for example, the name of the data file, authors etc.);
- Line 3 is of the form "`# Nodes: integer1 Edges: integer2`", where `integer1` is the total number of vertices, and `integer2` is the total number of edges;
- Line 4 is of the form "`# FromNodeId ToNodeId Direct-distance`";
- The remaining part of the file consists of a number of lines, one line per edge. Each line simply contains three integers: the index of the start-vertex, the index of the end-vertex and the direct distance between the two vertices;
- Some of the edges can be self-links (same start as end), these should be excluded when creating the vertex connectivity matrix;
- Note: all the indices start from 0 (C convention);
- It can be assumed that each edge is uniquely listed, that is, there will NOT be multiple text lines describing the same edge;
- You may however NOT assume that the edges are sorted with respect to `FromNodeId`;
- For each `FromNodeId`, you may NOT assume that the edges are sorted with respect to `ToNodeId`.

3.5 An example graph file

An example file that contains the vertex connectivity info of the directed graph that is shown in Figure 1 is as follows (where vertices S,A,B,C,D have indices 0,1,2,3,4):

```
# Directed graph: 5-nodes.txt
# Just an example
# Nodes: 5 Edges: 10
# FromNodeId    ToNodeId    Direct-distance
0    3    5
0    1    10
4    0    7
4    2    6
1    3    2
2    4    4
1    2    1
3    1    3
3    2    9
3    4    2
```

The corresponding sparse matrix stored in the CRS format will have the following three arrays:

```
values in array 'row_ptr': 0  2  4  5  8  10
values in array 'col_idx': 3  1  3  2  4  1  2  4  0  2
values in array 'val': 5  10  2  1  4  3  9  2  7  6
```

Note: In general graphs, there may be nodes that have no incoming edges, so the distance to these nodes will be infinity.

4 Submission

Each student is required to unload a zipped file folder (created by the `zip` or `gzip` tools) to Devilry within the deadline. Upon unpacking, the file folder should contain two sub-folders with at least the following content:

```
serial_part/:
dijkstra_serial_v1.c          dijkstra_serial_v2.c
read_graph_from_file_v1.c     read_graph_from_file_v2.c
README.txt                    serial_main.c                 serial_functions.h

omp_part/:
dijkstra_omp_v1.c             dijkstra_omp_v2.c
README.txt                    omp_main.c                    para_functions.h
```

The two files of `README.txt` should explain how to compile the code and run the 'main' programs (including examples of command-line input arguments). Where necessary, please insert comments in the implemented functions and 'main' programs, for the purpose of helping others to understand the code.

An example folder skeleton containing the needed files will be provided at the semester webpage, together with a few example graph files.

References

[1] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Addison Wesley, 2003.

[2] Wikipedia, *Dijkstra's algorithm*, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.