Suszec, 10.11.2020

Semester: 3

Group: 2

Section: 1

COMPUTER PROGRAMMING LABORATORY

Author: Anna Mrukwa

E-mail: annamru993@student.polsl.pl

Tutor: Krzysztof Pasterak

Task topic

The aim of the task was to implement the dijkstra algorithm to find the shortest path in the graph. In particular, the program should be able to read the data from the file and, with the specified start and the end in the last line of the file, find the shortest path between them.

Project analysis

The algorithm used in the program is the dijkstra algorithm. It is a simple shortest path search method. We start with setting all the nodes' distance (except the starting node's, which is set to 0), to infinity. All of them should be marked as infinity and have no previous node - as we did not visit them, there is no node we came through to reach it. Then, the starting point is inserted into the priority queue, sorted by the shortest distance from the starting point.Until the priority queue becomes empty or we reach the end point, we pop the consecutive elements from it, mark them as visited (this way even if they are inserted into the queue once again, they will not be processed again as this would not change anything) and check if the distance through them for the neighbours is smaller than the their current distance from the starting point. If yes, the distance is changed and the neighbours are inserted into priority queue. If the priority queue empties and the end was not processed (it is indicated by its member value visited = false and its member previous=-1), the path could not be found between the two specified points.

For the implementation, I need multiple nodes and edges between them. To store them, I will use two separate standard library vectors of specifically defined objects of classes Node and Edge. The "previous" members of nodes will hold the indices of the appropriate node in the vector. The same way the information about the end and start of the Edge will be stored. As for the priority queue, I decided to create the additional class Pair for storing the distance (at the moment of creation) from the start and the index of the node in the vector of Node objects. This is because one node could be inserted into the queue multiple times. As the priority queue makes a copy of the inserted object, the changes in the distance made on the original node would not be replicated. Therefore, the distance is stored for the sake of the sorting the queue and the index serves for checking if the node was already processed - if yes, there is no need to process it again and it may be skipped, as the distance in the Pair object is greater than the actual distance stored by the processed Node object. Then, after the end of the algorithm, if the path could be found, I extract the path by checking the "previous" of each point, starting with the destination and finishing with the start. As can be noticed, the path will be in a reversed manner. As the path should be further given to the user, I decided to store the consecutive points in the standard library vector and save them to the output file with the use of the reverse iterator.

I should also prepare the file input validation for reading the graph from the file. The user could specify twofold point names, forget to give the edge length or point name. The program should be able to cope with such unexpected behaviour.

External specification

1.1. Downloading

The easiest way to download the executable of the program is through one of the github repositories: university or private. Then, in the *Releases* tab, choose dijikstra.zip for the download. Unzip the folder. Now you can run the program easily. Remember that you should never remove the help.txt file - it holds all the clarifications for --help command. You may remove the data.txt file and, for example, replace it with the file of the same name of your own. The given file is supposed to be an exemplary file for better explanation of the proper file format.

1.2. Running

The program should be run from the command line. Otherwise, by default, the executable will try to find the file data.txt in the directory it is situated in. This means you will not be able to ask for help on the program work. Before the start of the program, you should change the command directory for the one holding the executable file (cd directory). Then, you can run the program in three possible combinations:

- dijikstra.exe - the program will expect to find the data.txt in the current directory. On the fail, you will be informed.
- dijikstra.exe --help - the program will show you all the other possible commands and messages. This will give you clearer insight into the program notices.
- dijikstra.exe --input=directory/filename.txt - instead of loading data from the data.txt, the program will try to open and read the specified file and create the graph from it. If it is impossible, you will be informed about it.

1.3. The input file format

The program needs the input file to run the dijkstra algorithm. The text file should be provided. By default, the data.txt in the executable program directory is looked for. However, as explained before, you can use another name and location if you specify it with the correct command. Yet, it still should follow some rules:

- if you wish to specify the path to be looked for, you should do so in the last line of the file. Otherwise, you will be asked to provide it through the command line. It should follow the format: start_point_name end_point_name. Both names should belong to the graph, otherwise you will have to specify both of them once again.
- do not use commas as the separators, the consecutive lines should look as follow: start_point name end_point_name distance
- as you may have noticed, the two-part names should be written with the underscore.
- the distance should be a positive number.
- end each line with enter.

1.4. Messages

You can encounter the following messages during the program execution:

- Help message: it will show you all the other messages that can be displayed, as well as the commands that can be used by you.
- "No such file in chosen directory." - it means exactly that. The directory and the file you provided could not be found or you did not specify any and the program has not found the default file.
- "Unknown command" - you provided the command that could not be processed.

- "The edge should have positive value. Specify the correct length." accompanied by the names of the points - you gave 0 for the edge length. Provide actual value for the edge between these points.
- "Invalid edge. Specify the correct length." - you gave a character for the edge length or gave a point with two part name and no edge. You can only specify the edge length here.
- "The edge length cannot be 0/character." - after the program request for the correct edge length you provided 0 or character instead of positive value.
- "... was provided. Specify the actual edge." - the input line was lacking/you provided too much. You should specify it again: start_point end_point distance
- "Choose the starting/ending point:" - the program could not find the start and end points in the last line of the input, so you should provide them manually.
- "No such node." - the point you provided is not in the graph.
- "There is no path." - the program did not find the connection between specified points.
- "There is a path!" - the program found the connection you asked for.
- "Failed to create file." - the file with the exact path could not be created.

1.5. The results

If the path could not have been found, you will be informed through the console message. Yet, if the path does exist, besides the appropriate notice, there also will be created the file with the detailed information about the path, point by point with corresponding distances.

1.6. Further inquiry

All the necessary information about the program work is also in the README Project files in both repositories. This should provide easy and fast information about the program outside the report as well.

Internal specification

In the project I declare the following classes:

- in the file_handling.h:

  class NoDirectory - exception class that should be thrown if the specified input file could not be found or the help.txt file is removed

  class Handle_Commands - class created to handle the commands declared along the program call. It is supposed to be constructed with command line arguments passed into main. Its method are:

  - void give_help() - reads help.txt (if possible) and prints the help message to the user.
  - void read_dir(char* adress, std::ifstream& ifile) - uses ifile to opens the file in directory specified by the address, previously checking, if it is possible)
  - void load_from_current(std::ifstream& ifile) - tries use ifile to open the default file data.txt in current directory
  - void use_command(char* p, std::ifstream& datafile) - checks the given non-default command and if the command is correct, reads the directory in the p or gives help, depending on p.

  - constructor and destructor
  - void finish() - simple replacement for system("pause")
  - void handle_commands(std::ifstream& datafile) - uses the stream to process possible commands, default or not.

- in the Edge.h:

  class NegativeEdge - another exception class

  class Edge - class with members specifying start and destination indices, as well as distance between them. (int start, int destination, double dist). Some of its method are:

  - void check_dist(double d) - checks whether specified given variable d is of correct format, i.e. positive and double. If not, the appropriate exceptions are raised.
  - void check_nodes(int v, int w) - checks if correct indices v and w were provided. They should not be the same (loops are not allowed) and should be nonnegative (as they are supposed to be indices of the vector).
  - Edge(int v, int w, double d) - parameterized constructor checking given nodes v, w and distance d before creating an object.
  - Edge(const Edge& e) - copy constructor.

- in the Node.h:

  class Node - class with members bool visited (false by default, as it is not visited yet), double dist (set to INF defined by std::numeric_limits<int>::max()), int previous (-1 by default, as there will never be -1 index in c++) and std::string name, holding the name of the object. Some of the class methods are:

  - Node(const Node& n) - copy constructor
  - bool is_name(std::string n) - checks if the n string and this->name are the same
  - void visit() - sets this->visited to true
  - bool check_prev(int prev, double d) - check if this-> dist is bigger than d. If so, this->dist is changed to d and this->previous to prev.

  class Pair - convenience class used for priority queue sorting needs. The members are: int index, which will hold the index of the object in the nodes vector and double and double cur_dist, which is the current distance of the said object from the start of the graph (dist member). The most important methods and operators are:

  - Pair(const Node& node, int idx) - parameterized constructor, setting this->index to idx and this->cur_dist to node.dist
  - bool operator> (const Pair& a, const Pair& b) {return a.cur_dist > b.cur_dist; } and < as well - check if cur_dist of the first Pair object is bigger/smaller. It will be useful for the priority queue implementation.

- in the Graph.h:

  class ZeroEdge - exception class, with double specify_edge(std::string st, std::string end) method. It is used to catch an error of specifying the edge of zero length. The user is shown the start and end (these are names of the nodes) and asked to give a correct value. This value is then returned by the method.

  class InvalidEdge - exception class with double specify_edge(std::string st, std::string end, std::istream& datafile) method. It should be used to resolve an error of loading characters into the edge length. The user is asked to specify the correct length, which will be returned. As the datafile status is set to fail, the method clears it, reads the problematic characters into the prepared variable inside the method to enable further file processing.

class InvalidNode - exception class that will be thrown in case of specifying invalid node format.

class CheckFile - checks the lines of the input file before processing it by Graph methods, its member is bool vector is_for_graph holding the upfront information about the lines of the file - if they consist of 3 variables or not, meaning correct format for Graph or not. The methods:

- void check_lines(std::ifstream& datafile) - checks the lines of the datafile stream and saves if they are correct or not to is_for_graph vector
- std::string get_line(std::ifstream& datafile) - reads the line from the datafile. It is adapted to read after both >> operator and std::getline.

class Graph - contains the information about the graph and the searched path. Its members are: vector of Node objects nodes, vector of Edge objects edges, int ending storing the index of the finishing Node, int start storing the index of the starting Node. The methods are as follow:

- int user_choose_node(std::string which_node) - returns the index of the Node in nodes (if it occurs) with the name that user gives after being asked to. which_node helps specifying if it should be a starting/ending node.
- int user_validate_node(std::string which_node) - catches exceptions that could be generated by the previous method by terminating the program. Otherwise returns the index returned by the previous method.
- void ask_about_path() - asks about both the starting and ending point of the graph, using the previous method.
- double from_three(std::string& st, std::string& end, std::istream& datafile) - tries to read the consecutive three values from the datafile to enable creating the edge and nodes (if they do not exist yet). If the dist that should be returned from the function is 0 or a character, an exception is raised.
- void ask_about_line(std::string line) - shows the user that line was provided and asks them to specify the expected data in a correct manner - start_point end_point distance. If the distance is incorrect again, the method asks for the last time to correct it. If everything is specified correctly, creates a new edge and possibly nodes and appends the elements to the appropriate vector.
- void read_typical_line(std::ifstream& datafile) - loads the current line in datafile, expecting it to be specified correctly (the program did not detect twofold names or missing distance or missing point). However, if now the specified distance proves to be incorrect anyway, it catches the raised exceptions and in order to fix them asks the user to specify it correctly.
- void read_aim_line(std::ifstream& datafile, std::vector<bool>& is_for_graph) - tries to read the last line as the specifying the start and the end of the graph (it is already checked if there is no distance at the line and the method assumes there is not). If at least one of the nodes at that line is not in the graph, the values are not loaded. Instead, the last element of the is_for_graph vector is changed to true, to signal the need to ask the user about the desired path.
- void load_data(std::ifstream& datafile, CheckFile& cf) - load the data after checking the datafile stream with cf methods (that is how it is supposed to be used at least). Using cf.is_for_graph vector, it calls the right methods for every checked line and loads the graph from it.

- void write_path(std::ofstream& out, std::vector<int> path) - writes the names corresponding to the contents of the path vector into out by iterating backwards through it, as well as the distance between consecutive points.
- Graph(std::ifstream& datafile) - creates the new objects using the datafile. If something is not properly prepared, the user will be asked to specify it again.
- void show_nodes() - shows all the nodes' names of the graph.
- void from_one_set(std::string start, std::string aim, double d) - creates new Edge and, if necessary, also new Nodes and then appends it to the Graph's vectors.
- int find_where_node(std::string name) - returns the index of the Node in the nodes vector which this->name is name, returns -1 if there is none.
- int find_and_fill_node(std::string name) - returns the index of the Node in the nodes vector which this->name is name. If there is none, creates new Node, appends it to the vector and returns the index.
- double find_edge_len(int v, int w) - returns the length of the edge between the Node objects at v and w indices in nodes. If there is no edge, method returns 0.
- void add_node(std::string name) - creates a new Node object with this->name equal to name and appends it to the nodes vector.
- std::vector<int> extract_path() - iterates over the nodes vector element, starting at the Node at index ending, then goes to its previous and so on until the previous of the current element is -1. Such constructed vector of indices is returned.
- void show_path() - firstly, tries to find the path between start and ending. If the path could not be found, informs the user about it. Otherwise, informs about success and tries to save the path to the output file "path.txt".
- void dijikstra() - runs the dijkstra algorithm.

Source code

All of my source code is in both university's and mine repositories, alongside the proper commentary. It may be directly cloned from the main branch, or downloaded from the *Releases* tab. I will not place it here in order to preserve the clarity of this report.

Testing

For the testing of the project I wrote unit tests that are placed on my repositories as well. To automate the process of testing and implement continuous integration, I introduced personally written github workflow files, enabling build and testing after each push on the remote repository. What is more, I manually performed end to end testing, checking the correctness of input validation, the displaying of the help information, as well as the actual running of the program and path calculation.

Conclusions

I strived to tag the consecutive major changes of the already working application, enabling easy access to both source code and executable version of the program, which both can be easily downloaded via the *Releases* tab. All the changes to the code are going through the process of the continuous integration and deployment.