Programming project

Algorithms and Data Structures (ADS).

January 24, 2022

1 Introduction

The vertices of a graph of n vertices can be arranged linearly, forming a sequence as in Fig. 1. The position of vertex v in the linear arrangement is $\pi(v)$ (the 1st vertex of the sequence is in position 0 and the last vertex of the sequence is in position $n-1$. In such a layout of a graph, the distance between two vertices u and v is defined as $|\pi(v) - \pi(u)|$. Thus consecutive vertices in the linear arrangement are at distance 1. The sum of distances between linked vertices is

$$D = \sum_{\{u,v\} \in E} |\pi(v) - \pi(u)|, \tag{1}$$

where E is the set of edges of the graph. Fig. 1 (a) and (b) show the same graph but with a different linear arrangement of the vertices. In Fig. 1 (a), $D = 13$, while $D = \binom{7}{2} = 21$ in Fig. 1 (b). In computer science, the maximum linear arrangement problems consists of finding D_{max}, the maximum value of D over the $n!$ possible linear arrangements of a graph. The problem is NP-complete. For the graphs in Fig. 1 (a) and (b), $D_{max} = \binom{7}{2} = 21$. For the graph in Fig. 1 (c), $D = D_{max} = 7$.

2 Work

You will have to create a series of programs in C++. These programs have to use a class *graph* that we have designed for this project. Have a look to `program0.cpp`, that shows how you can create a graph and write its edges to the standard output. In the programs to follow, graphs will be created by reading them from the standard input.

In this project, you will learn how to build programs with multiple parts and to compile with `Makefile`. You will practice on redirecting the standard input and the standard output. You will also learn how to test the correctness of programs and how to measure their time efficiency. This principles will be used for future exams. You will also learn on the branch & bound tecnique. We also expect that you learn to team up in a mutually enriching way.

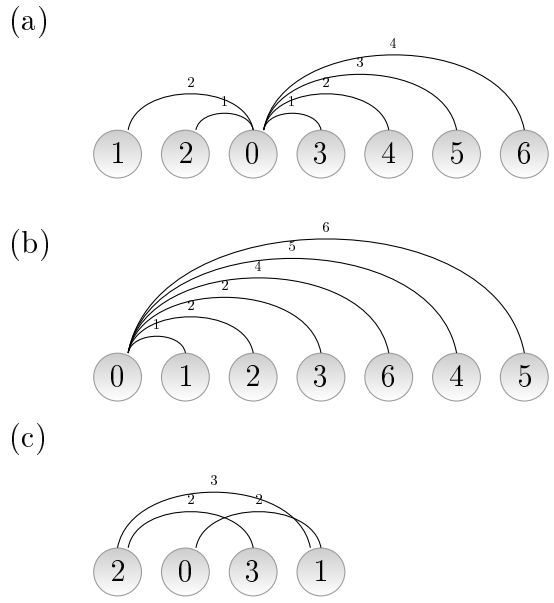

Figure 1: Linear arrangements of trees. The trees in (a) and (b) are the same but the vertices have been arranged differently. Edge labels indicate the distance between the linked vertices.

Central to testing the correctness of a program are test suits (*jocs de proves* in Catalan). In this project, a test suit consist of a series of text files with the input and the expected output for a specific program. When designing your test suit you have to assume, for simplicity, that there will be no error of format (namely, the input will follow always the format of the examples). Your test suits have to focus on the logics of the core algorithms following the black-box and the white-box approach.

2.1 Appetizer (`program1`)

Design a simple program that takes a single graph as input, and the position of the vertices, and writes to the standard output the value of D. The program has to call three functions: one to read the graph from the input; another to read a vector with the linear arrangement (corresponding to π) and finally, on that takes two parameters, i.e. the graph and that vector, producing D. Provide the code and the content of your test suit.

The input of this program is a text consisting of pairs of integers. The first pair indicates n, the number of vertices and m, the number of edges of the graph. After the first pair, there are there are m pairs defining the edges and, finally, a line with n integers indicating the position of every vertex, namely $\pi(0), ..., \pi(i), ..., \pi(n)$. For a tree of 3 vertices, the text format is

```
3 2 // number of vertices (n) and number of edges (m)
0 1 // vertices forming an edge
1 2 // vertices forming another edge

0 1 2 // n integers indicating the position of each vertex;
      // the i-th integer corresponds to $\pi(i)$.
```

Be aware that the // is just used for making comments. The true input does not have this comments as you can see in the folder `test/program1`.

For that input, $D = 2$ and then the output of the program is

```
2
```

If the input was

```
3 2
0 1
1 2

0 2 1
```

then $D = 3$ and the output would be

```
3
```

To run this program and the other to follow, you must redirect the standard input to a file, e.g.

```
./program1.x < test/program1/input0.txt
```

To check that this programs and others to follow give the expected output, you must redirect the standard output and then use `kompare` (or `diff`), e.g.

```
./program1.x < test/program1/input0.txt > test/program1/my_output0.txt
kompare test/program1/output0.txt test/program1/my_output0.txt
```

2.2 Calculation of D for an ensemble of graphs (`program2`)

Design a simple program that takes various graphs as input and produces as output the value of D for each of them when vertex labels are interpreted as vertex positions. Provide the code and the content of your test suit.

The input consists of a number of indicating the number of graphs in the input followed by the definition of each graph following the format described above. For instance, if the input is

```
2 // there are two graphs

3 2 // first graph
0 1
1 2

3 2 // second graph
0 2
1 2
```

then the output has to be

```
2
3
```

2.3 Calculation of D_{max} by exhaustive exploration, iterative solution (`program3`)

Design a simple program that takes various graphs as input and produces as output the value of D_{max} for each of them. The results has to be calculated using an exhaustive enumeration of the $n!$ linear arrangements. Use the method described here `http://www.cplusplus.com/reference/algorithm/next_permutation/` adapting it to STL vectors. Provide the code and the content of your test suit.

For instance, if the input is

```
3

3 2
0 1
1 2
```

```
3 2
0 2
1 2

4 3
0 1
0 2
0 3

4 3
0 1
1 2
2 3
```

then the output has to be

```
3
3
6
7
```

2.4 Calculation of D_{max} by exhaustive exploration, recursive solution (`program4`)

Same as the previous program but using a recursive algorithm for the exhaustive exploration. Provide the code and the content of your test suit, extending or refining the test suit of the previous program.

2.5 Calculation of D_{max} applying branch & bound, recursive solution (`program5`)

Produce a variant of the previous program where the results are obtained applying a branch & bound technique over the implicit tree. Provide the code and the content of your test suit, extending or refining the test suit of previous programs.

3 To deliver and general conditions

Aspects that will be evaluated:

1. The quality of the programs (their correctness, efficiency, clarity and simplicity).
2. The quality of the test suits.
3. The quality of the report (e.g., the adequacy, precision and clarity of the explanations, the quality of English).

Rules about programming

1. Use C++.
2. Use the class *graph*.
3. Do not modify files `graph.cpp`, `graph.hpp` or `Makefile`. If you wish to do that, please ask the professor first.
4. Use `graph_utilities.hpp` and `graph_utilities.cpp` for functions that you wish to use in different programs.
5. The core algorithm of `program3` must iterative but that of `program4` and `program5` must be recursive.
6. All data structures that you use must come from STL. For vectors (or matrices), use the class `vector` from STL instead of `array`.
7. Compile with *make*, using the Makefile that is provided. To compile `program0` type

   ```
   make program0
   ```

   Use the options `-D_GLIBCXX_DEBUG` (automatic error detection) and `-O3` (maximum speed).
8. Programs have to read the input from the standard input (using `cin`) and write the output to the standard output (using `cout`).
9. Keep the names of the programs that are indicated.

Rules about how the work is performed

1. Work is done in pairs. If there is an odd number of students a group of 1 or 3 people can be formed upon approval from the professor. Non-approved groups of 1 or groups of 3 people will receive a score of zero.
2. No plagiarism. Don not discuss your work with others, except your teammate; if in doubt about what is allowed, please ask the professor. Plagiarism implies a score of zero.
3. If you feel that you are spending much more time than the rest of the group, ask us for help.
4. Questions can be asked either in person or by email, and you will never be penalized for asking questions, no matter how stupid they look in retrospect.

Details that are heavily penalized:

1. Breaking any of the rules above.

Program	Compilation	Test suit	t_i	r_i
1			X	X
2			t_2	X
3			t_3	r_3
4			t_4	r_4
5			t_5	r_5

Table 1: The field *Compilation* takes *Yes* if the code that has been provided produces a .x file without compilation errors and *No* otherwise (if not code is provided the answer is also *No*). The field *Test suit* takes *Yes* if a test suit has been provided (in addition to the example files) and *No* otherwise. t_i is the execution time of the i-th program on the file `godzilla6.txt` and $r_i = t_i/t_3$ measure the speed of that program relative to the execution time of program 3. X indicates cells that have to be left empty.

2. Code without the corresponding test suit (in addition to the examples provided in the statement or supporting files) will receive a score of zero, even if it is correct.

To deliver in compressed file (`.tar.gz` or `.zip`):

1. A brief report (3-4 pages maximum in PDF format) with the the following content:

   (a) An explanation of the main choices or difficulties when designing and implementing the algorithms or the test suits. The rational behind each test case must also be explained.

   (b) Any ideas or tricks you have applied to improve the efficiency of the algorithms in general.

   (c) The rational behind your branch & bound technique.

   (d) A completed version of Table 1, that provides information about the execution time for the complex programs. To measure the running time, you can use the command `time` and you must redirect standard input and output to a file, e.g.

   ```
   time ./program4.x <test/godzilla6.txt >out.txt
   ```

   Redirecting the input only will give an unreliable time estimate. Please, specify the units of time and the characteristics of the computer with which you measured time: speed in GHz, number of cores, type of processor and manufacturer and the type of disk where the files are located: hard drive disk (HDD) or solid-state drive (SSD).

   (e) Include also a discussion of the results on the execution time in Table 1.

2. The source code of your implementations and the `Makefile` in the base directory. All the programs should be in the same folder and compile successfully using `make`.

3. The test suits for each program, in the corresponding folder of the form `test/program1`, `test/program2`,...

The compressed file must contain files and directories with the same organization as in the package with the statement of the project and supplementary files.

Procedure: Submit your work through Aula ESCI.

Deadline: Work must be delivered BEFORE February 10. Late submissions will not be accepted.