Professor Dr. Stephan Held
Malte Schürks

Programming Exercise 1

Exercise P.1. *Task:* Implement the Cardinality Matching Algorithm as described in the lecture.

Usage: Your program should be called as follows:

```
program_name file1.dmx
```

Input: The argument, `file1.dmx`, is mandatory (i.e., your program should exit with an error message if it is not present), and it encodes the graph G for which your program should find a maximum cardinality matching.

File `file1.dmx` is expected to be in DIMACS format, which is used to encode undirected graphs as follows: All lines beginning with a `c` are comments. Now, ignoring any comment-lines, to encode a graph G, the first line has the format

`p edge` n m

where $n = |V(G)|$ and $m = |E(G)|$. From this, $V(G)$ is implicitly identified with $\{1, \dots, n\}$. Note that vertex indices start with 1 in the DIMACS format. The following m lines have the format

`e` i j

representing that $\{i, j\} \in E(G)$. If the given file does not match the described format your program should exit with an error message.

Output: Your program should return a maximum cardinality matching M in G by writing the complete DIMACS encoding of the subgraph $(V(G), M)$ to the standard output. Your implementation should achieve a runtime of $O(n^3)$.

Use of heuristics: As an extra speed-up, if you wish, you may use some heuristics. For example, you can start with a greedy routine to add edges to M_0 until it is maximal.

Programming language: Your program should be written in C or C++, although the use of C++ is strongly encouraged. By default, your program will be compiled using g++ 13.2.0 using C++20. Different compilers or compiler versions are

available upon request. Your program will be compiled using `-pedantic -Wall -Wextra -Werror`, i.e., all warnings are enabled and each remaining warning will lead to compilation failure. Program evaluation will be performed on Linux. The standard library can be used as you wish. No other libraries are allowed. Add a script containing your compile command.

Algorithm evaluation: You are expected to implement the algorithm as described in the lecture (slightly different from the presentation in Korte-Vygen, but it would certainly be helpful to take a look at their description as well). The runtime requirement will not be handled strictly, so complicated implementation details that make your program *slightly* slower may be overlooked, but the core of the algorithm must be as efficient as described in class.

Code evaluation: Running time in practice will also be evaluated, as well as the elegance, cleanness and organization of your code. Make sure to add good documentation and give the variables, functions and types meaningful names that make their role clear. Break your complicated functions into small simple ones, break your program into a few units etc. Of course, your program should not trigger undefined behavior. In particular, your program should be `valgrind`-clean, i.e. it should not leak memory and should not perform invalid operations on memory.

Help: On the website for the exercise classes, which you should visit regularly, you will find a C++ implementation of a simple graph class that you can use (if you wish). This implementation also includes a parsing routine for the DIMACS format. A few instances with their optimum values are also provided on the same website for testing purposes. You may use the Makefile provided with the graph class if you like.

Please submit your programs in groups of up to 3 students.

(64 points)

Deadline: November 16$^{\text{th}}$, before the lecture. The websites for lecture and exercises can be found at:

`http://www.or.uni-bonn.de/lectures/ws23/cows23.html`

In case of any questions feel free to contact me at schuerks@or.uni-bonn.de.