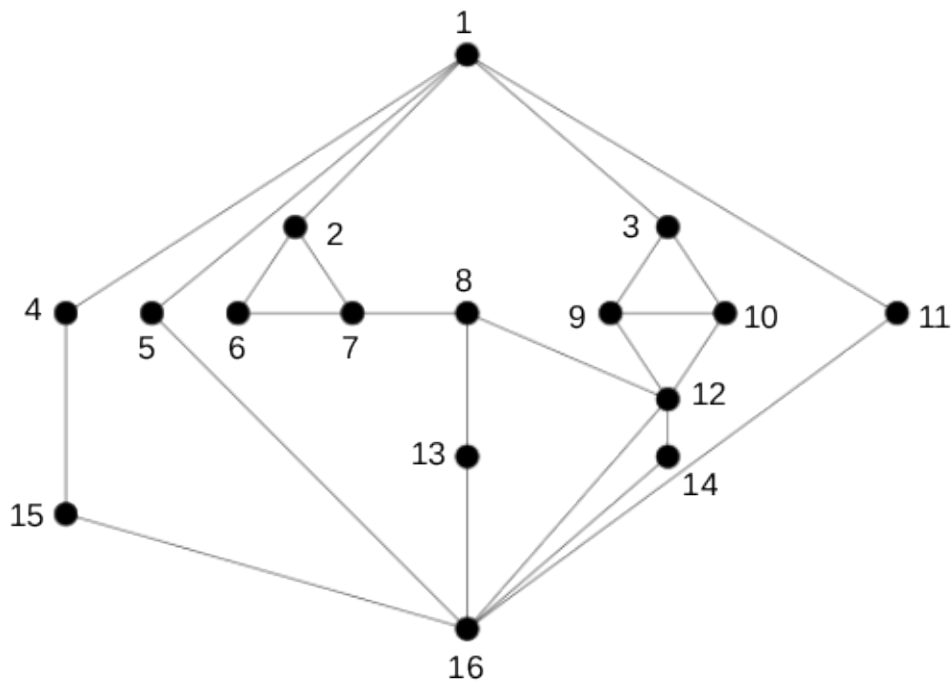# Handout for First Exercise Class

Exercise 0.1. Show that a graph G is factor-critical if and only if G is connected and for every vertex $v \in V(G)$ we have $\nu(G - v) = \nu(G)$.

Figure 0.1: Graph for exercise 0.2

Exercise 0.2. Find a maximum matching in the graph in Figure 0.1. (*Hint:* Consider the vertex set $X := \{1, 8, 12, 16\}$.)

Notes on theoretical exercises

- Submit solutions that are easily readable, i.e. either use a clean handwriting or use LaTeX.

- Use clean mathematical proofs.

- Be clear on what is a claim, and what a proof.

- Be clear on the structure of your proof, and where you use which result.

- If possible, structure longer proofs into subclaims that are proven separately.

- Don't use ambiguous notation, e.g. differentiate between *maximal matchings*, which are *inclusion-wise maximal*, and *maximum matchings*, which achieve the *maximum possible cardinality*.

- Submit at most one solution per exercise.

Notes on programming exercises

- There will be two challenging mandatory programming exercises. If you have no or very little programming experience, start preparing now.

- The programming exercise can be solved in C or C++. I strongly recommend using C++.

1 High-Level

- Avoid code duplication.

- Split your code into small, re-usable, independent units.

- Test your code. The better you separate your code into re-usable simple classes / functions, the better you can test these separately, e.g. using unit tests. Test corner cases (e.g. an empty graph).

- Document your code. State input/output of functions, and properties the input needs to satisfy, and properties the output is guaranteed to have.

- Validate your input. Do not rely on the program input to satisfy the requirements you need; check them.

- Avoid manual memory management.

2 Code Structure

- Distribute your code into separate files.

- Avoid long functions and huge classes. Split off code for independent subtasks into separate functions / classes providing a singe logical functionality.

- In particular, the `main` function should **only** consist of the very top-level logic that e.g. reads a graph and calls an algorithm on it.

3 Code Style

- Use `const` as often as possible. Mark member functions `const` that do not alter the object, mark passed functions argument `const` that are not to be modified, and even mark local variables in functions `const` if they do not need to change.

- Use a consistent naming scheme for types, members, functions, etc.

- Use meaningful variable names. Avoid generic names like `i,j,k,n,m,x,y,z`, in particular in longer functions. Of course, if used for the same value in a consistent way (e.g. `n` for $|V(G)|$), or used very locally (i.e. as a counter in a function of a few lines), exceptions may apply.

4 Memory Management

- Do not use `alloc, malloc, calloc, free`.

- If possible, do not use `new, delete`.

- Use built-in data structures (e.g. `std::vector` for an array of dynamic length).

- Use smart pointers for owning pointers (`std::unique_ptr` and `std::shared_ptr`) [requires C++11 or newer]. Construct these using `std::make_shared` and, from C++14 on, using `std::make_unique`.

5 Language Features

- Use standard library functionality where possible. For example, do not implement your own sorting algorithm, but use `std::sort`. Many common operations, in particular on iterators, often are implemented in a general way in the standard library. For example, instead of manually searching in an array for an element, use `std::find` and `std::binary_search`.

- Don't use macros if possible. In particular, do not use macros where functions suffice, e.g. to compute the maximum of two numbers.

- Don't use `goto`.

- Do **not** use `using namespace std;`. At the very least, restrict to functionality you actually use, for example by `using std::cout;`.

- Although supported by some compilers, variable-length arrays are not allowed in standard C++. Do not use them. Variable length arrays are arrays whose lengths is determined at run-time as in the following example:

```
void foo(int n)
{
    int arr[n];
    ...
}
```

6 General Notes

- C++ is not Java. For several reasons, do not write

```
void foo() {
    T* t = new T(args);
    ...
    delete t;
}
```

Instead, just write

```
void foo() {
   T t(args);
   ...
}
```

- Do not use non-`const` global variables.

- Always initialize your variables. Instead of `int i; i = 5;` write `int i = 5;`. Even better: `int const i = 5;`

- Be aware of whether you pass arguments by value or by reference, and the run time implications. For example, passing a `vector` of length n by value to a function results in a deep copy of the `vector` with runtime $\Omega(n)$. In most cases (unless the arguments need to be modified), passing as const reference is the correct choice.

- Always state all include dependencies. For example, if you use `std::max`, include `<algorithm>`, even if the program compiles without it. The set of includes inherited from included library headers may change from compiler to compiler.

- Fix **all** warnings.

- Never leak memory, even if the program input is invalid. You may want to use `valgrind` to check. If you do not manage memory manually, leaks are very unlikely.

- Strictly avoid undefined behavior (UB). Many operations in C/C++ immediately invalidate the whole program, for example:

  - signed integer overflow
  - reading uninitialized variables
  - integer division by zero
  - out-of-bounds array access

- Be aware of object lifetime, and avoid dangling references.

7   Useful Idioms

- Make use of the RAII (**R**esource **A**quisition **I**s **I**nitialization) idiom: When performing some task that requires cleanup later on (allocate

memory: needs to be deallocated; acquire a lock: needs to be released; make a temporary change: needs to be reverted), perform this operation in the constructor of a helper class, and perform the cleanup in its destructor. Then, use a local scope to define the lifetime of the helper object. This way, it is guaranteed that for each such task, the corresponding cleanup action is always executed. In particular, it cannot be forgotten (e.g. when using a `return` statement in the middle of a function), and is also executed in case of exceptions. Note that using standard library classes like `std::vector, std::unique_ptr, std::shared_ptr` already achieves this for the special case of memory allocations.

8 Debugging

- Use common debugging tools: Under Linux, `gdb` and `valgrind` are **very** helpful. In particular, `valgrind` is helpful to debug memory problems.

- Test small instances first. Try to reproduce bugs on instances as small as possible, possibly in some automated way.