V4C1 / MA-INF 1102 - Combinatorial Optimization

Programming Exercise 1
Cardinality Matching Algorithm

Ivana Koroman
Lucas Keiler
Noah Schaumburg

December 8, 2020

1 Compiling and Executing the Code

The compilation is performed using the (exactly identical to the original) *shell script* `compile.sh`. It creates an `matching.out` executable target file which can be executed as requested by the exercise statement:

```
>./compile.sh
>./matching.out ./ex1/simple_tests/g2.dmx
p 8 4
e 1 5
e 2 6
e 3 7
e 4 8
>
```

The only output given is a encoded in DIMACS subgraph $(V(G), M)$, where M is a maximum cardinality matching for G.

2 Implementation Structure

The implementation was performed entirely in the following files:

- `match_algo.cpp` and `match_algo.hpp`: These files contain the most important implementation sections. They contain the two main classes:

  - `Tree`: This class provides an abstraction for an alternating tree. All main steps from the algorithm (*augmenting, expanding, shrinking and unshrinking*) are performed with respect to an alternating tree, therefore they were implemented as methods of this class.

    The abstraction of the updated graph G' (graph after a set of shrinkings) is also contained in the alternating tree class. It was implemented through an adjacency list (which was updated with every shrinking step), a list of *cycles* (structure that contains all vertices of a shrunken cycle, and the "representative" vertex) and a label list (in which every vertex was originally its own label, that was updated after it became shrunken). The "representative" vertex in our implementation (i.e. the vertex that represents the cycle after its shrinking $\rightarrow$ pseudo-node) was chosen as the nearest cycle vertex from the root of the current tree.
  - `Matching`: This simple class is an abstraction of a matching. It provides an easy way of adding and removing edges (which were defined as an unordered pair in our code as a *struct*) to a matching and printing in DIMACS format.

- `graph.cpp` and `graph.hpp`: One method was added to introduce the new nodes in the graph for the main loop of the algorithm. Apart from this, the class was left identical to the original provided for this exercise.

3 Library Requirements

Our implementation requires only the standard library. It actually runs with only (aside the given graph class) the following headers:

- `vector`
- `set`
- `stack`
- `algorithm`
- `memory`
- `iostream`

4 Limitations

The implementation runs slowly with some input graphs (for instance, the provided `gr9882`). The bottleneck was identified as the "rerouting" of edges in the *shrinking* step. The way of keeping an original vertex as pseudo-node (no label changes for a representative of a cycle) was, however, a straight-forward and easy way to maintain the current state of the graph.

Neither undefined behaviors nor wrong outputs were identified in our tests.