Classified Rank Maximal Matching Algorithm

Overview

The Classified Rank Maximal Matching Algorithm computes the maximal matching in an input bipartite graph while considering the rank of the edges and the classification of vertices.

Files

1. Vertex.h

- **Features**:
  - `ClassificationList classifications`
  - `Tree classification_tree`
- **Purpose**: Stores classifications and the classification tree.

2. GraphReader.h

- **Features**:
  - `read_classification`
  - `handle_classification`
- **Purpose**: Reads classifications from the input.

3. GraphDefs.h

- **Purpose**: Defines classes and typedefs for graph processing.
- **Includes**:
  - `TreeNode`
  - `ClassificationListElement`
  - `Edge`
  - `Graph`
  - Various typedefs for convenience.

4. ClassificationList.h

- **Purpose**: Defines the classification list interface using `vector`.

5. GraphReaderAlt.h

- **Purpose**: Transforms the input bipartite graph data into a ready-to-process format.

6. NetworkBuilder.h

- **Purpose**: Declares a class responsible for constructing the cumulative graph network used in the algorithm.

7. Algorithm.h

- **Purpose**: Declares a class responsible for executing the algorithm on the network to compute the rank maximal matching.

Key Components

TreeNode

- **Purpose**: Represents the information stored in the classification `TreeNode`.

Graph

- **Contains**:
  - `vector<Vertex> partA, partP`: Partitions of the graph.
  - `vector<vector<Edge>> edges`: Edges in the input graph (Part A to Part P).
  - `vector<vector<int>> network`: Adjacency matrix of the final cumulative graph.

ClassificationList

- **Purpose**: Acts as an interface and contains information about the classifications in the graph.

NetworkBuilder

- **Purpose**: Constructs the cumulative graph network from the input bipartite graph.

Algorithm

- **Purpose**: Executes the flow algorithm on the network to compute the maximal matching.

Process

1. **Read and Parse Graph**: Graph data is read and parsed, stored in the `BipartiteGraph` format. The `GraphReaderAlt` transforms the input format into an algorithm-acceptable format.
2. **Build Classification Tree**: Classification trees are constructed for each vertex based on their classifications.
3. **Construct Cumulative Graph Network**: The cumulative graph network is built by adding edges based on the rank and classification of vertices.
4. **Apply Flow Algorithm**: The flow algorithm is applied to the network to compute the classified rank maximal matching by augmenting the flow.

Input File Format

The input format is similar to Partition and PreferenceList Parser with an additional (optional) classification list.

Example

```
@PartitionA
a1;
@End

@PartitionB
b1, b2;
@End

@PreferenceListsA
a1 : (b1, b2);
@End

@PreferenceListsB
@End

@ClassificationListA
a1: {b1,b2} (1);
@End

@ClassificationListB
b1: {a1} (1);
@End
```

How to Run

1. Save the input as mentioned in the input file format into `input.txt`.
2. Run the algorithm with the following command to save the output to `output.txt`:

```
./graphmatching -q -i input.txt -o output.txt
```

3. To keep a log of the algorithm's workings, use the `-x` flag:

```
./graphmatching -q -i input.txt -o output.txt -x log_file.log
```

How to Test

1. Go to the `build` directory using `cd build`.
2. Run the bash script `run.sh` there

   ```
   ./run.sh
   ```

3. Hundred inputs are stored in `resources/input/` directory and their outputs are stored in the `resources/output` directory.

Details

- Outputs all violating pairs encountered for non-laminar flow (if any) and terminates execution.
- The log file contains all `Rank(i)` edges of the graph in the (i^{th}) iteration of the loop.
- The final output contains all edges in the matching of the graph.

Current Usage

- Works on a bipartite graph with Part A having a preference list (with ties).
- Supports classifications on both Part A and Part B.
- Includes a parser to read all the PreferenceLists and Classification Lists.
- A graph network interface has been written.
- All classification functions (Reader, Tree Builder, Network Builder) have been coded.
- `Algorithm.h` and `Algorithm.cc` contain the current implementation of the algorithm.

Future Work

- Make the network flow for general `s` and `t` (currently `s = 0`, `t = 1`).
- Given an input graph, provide maxflow and its S,T,U decomposition (functions are ready, just need module integration).
- Improve code modularity and naming conventions.
- Add a parser to read normal network and output the S,T,U decomposition.

Thank You