Technical Manual

Carbon Semantics
"https://github.com/CarboSem"
Department of Computer Science, University of L'Aquila

June 9, 2017

Abstract

CarboSem is a webapp interface for the DIANA project (Data science analysis to determine the Influence of multiple conjoint mirnAs on caNcer diseAse). developed with jquery, d3.js, python and the bottle micro-framework.

0.1 File Roles

The webapp comprises of 2 main files, one **"carbosem.py"** acts as a minimal server, the other **"carbosem.js"** acts as a data visualizer.

```
.
└── carbosem.py
    └── js
        └── carbosem.js
```

0.1.1 carbosem.py

The main server-side script that handles client requests, fetches files from the server to be loaded to the client, and relays submitted data from the client to the 3rd party server-side scripts. It is mainly a skeleton script with routing subroutines.
One certain routing subroutine that the 3rd party developers should be made aware of is the one that receives data from the client to be queried:

```
@route('/graph')
def graph():
  # TODO
  return True
```

In the TODO section should go the interface to the scripts that handle queries.

0.1.2 carbosem.js

The main client-side script that handles data manipulation on the returned JSON file and visualization of said data, it is responsible for the drawing and handling of the graph canvas, the checkbox area and the ledger area.
We will go in detail over some of the code segments which those who wish to use this script should pay attention to.

We start off with client events, which include calling the submission routine and the drawing routine on client submission of a string to be queried:

```
/*
* calling functions according to DOM bindings
*/
$("#search").submit(submitQuery);
$("#search").submit(drawGraph);
$(window).resize(drawGraph);
return;
```

From here we examine the submission routine, as every submission brings a new set of relations, we tend to reset the checkbox area (which when an element of which is active it denotes an active relationship on the screen):

```
function submitQuery() {
  /*
  * resetting the checkbox area
  */
  checkbox.states = [];
  checkbox.vals = [];
  checkbox.colors = [];
  /*
  * TODO
  * var query = $("#search").find("input[name=search]").val();
  * $.get("/graph?mir=" + encodeURIComponent(query));
  */
}
```

It is also important to note that 3rd party developers take good care in articulating the routing request from the submission routine to the **"graph()"** routine on the server side.

Moving on to the drawing routine, we notice that every time we call this routine the ledger is reset, because this routine is called regardless of any server submissions sometimes, depending on client change of the checkbox area, which in turn has the ledger require adjustment according to the new node types on display and according to the checkbox states.

```
/*
* resetting the ledger area
*/
ledger.elements = [];
ledger.colors = [];

d3.json("/getJSON", function (error, graph) {
  if (error) {
    alert("Error, no JSON file found!");
    return;
  }

  /*
  * cleaning up previously rendered checkboxes and ledger elements
  */
  d3.selectAll("div #addedCheckbox").remove();
  d3.selectAll("div #addedLedger").remove();
```

Also at this stage, the JSON file is reloaded and drawn from according to the checkbox states, and both the ledger and the checkbox areas are removed from screen for re-rendering to reflect new or less elements in the ledger area and the new chosen states of the checkbox area.

As for choosing the colors, we opted in for using the schemeCategory functions already packaged with d3 to construct our source color arrays, and we chose different schemes for the checkbox area and the ledger area:

```
/*
* defining color arrays
*/
var linkColor = d3.scaleOrdinal(d3.schemeCategory10);
var nodeColor = d3.scaleOrdinal(d3.schemeCategory20);
```

Whenever we get the result of a query, we construct our target color arrays depending on node/link types and the source color arrays.

For example, when we submit a query for the first time, as soon as we get the JSON file we scan through it looking for different types of links to populate our checkbox area accordingly, we store information about the link type, its state: whether it's active or inactive, and we derive its targeted color according to its index from the source color array:

```
var pushState = checkbox.vals.indexOf(graph.nodes[i].targets[j].type);
if (pushState == -1) {
  checkbox.vals.push(graph.nodes[i].targets[j].type);
  checkbox.states.push(true);
  checkbox.colors.push(linkColor(checkbox.states.length - 1));
}
```

If one wants fixed colors for certain nodes or links, one can choose a function with *"if"* statements to choose certain colors for certain node/link types. One other way is getting rid of the target color arrays and depending solely on one set of color arrays, but that requires using another method to choose the proper color index for a type, which is embedding this information inside the JSON file for each link/node type.

One final thing we ought to mention is the recalling of the drawing routine on checkbox changes by the client, that is done using the following:

```
addedCheckbox.on("change", function () {
     for (var i = 0; i < checkbox.states.length; i++) {
             checkbox.states[i] = addedBoxes[i].property("checked");
             checkbox.colors[i] = checkbox.states[i] ? linkColor(i) : "#
    FFFFFF";
             }
      drawGraph();
      return;
});
```

We notice that before the recall of the drawing routine, the checkboxes are checked for changes and if so, color and state arrays are changed accordingly to be re-rendered on routine recall.