PROJECT TRIPLESTORE

Documentation

PROJECT DESCRIPTION

Project TripleStore is a C program that implements a triple data structure in memory. Triplets always have exactly three values S, P and O. The structure is optimized for fast access with arbitrary matching patterns ({S}, {P}, {O}, {SP}, {PO}, {SO}, {SPO}). The data structure runs as WebAssemly Code on any modern web browser and render the content as a Graph Drawing.

GETTING STARTED

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes.

```
1    // clone git directory on your local machine
2    $ git clone https://diuf-gitlab.unifr.ch/SOP_2020_P01_cappuccino/sop2020_p01.git
3
4    // go the project directory
5    $ cd sop2020_p01
```

Prerequisites

In order to compile the C code to WebAssembly, you need to install Emscripten. We highly recommend using emsdk at version 1.38.48 (emsdk-1.38.48). The archive is included in the *utils* folder of the project.

You will also need Node.Js in order to run the project in web browsers because unfortunately, several browsers do not support file:// XHR requests and can't load extra files needed by the HTML (like a .wasm file, or packaged file data). For these browsers you'll need to serve the files using a webserver.

All the details regarding the installation of these prerequisites, then compilation and execution on the different targets are described very precisely in the *Readme.md* file which is at the source of the project.

IMPLEMENTATION DECISIONS

Architecture

The central core of the project, the C code, is at the root of the folder. The "build" folder gathers all the objects of the compiled C classes. In addition, there's the *Makefile* and a *Readme.md* file that provides all the commands needed to compile the project.

The folder named *benchmark* is the program that measures the performance of the software.

The folder named *utils* contains various and secondary elements related to the project, but which are still important. It contains the recommended Emsripten archive as well as a C class coded by us which allows to export the content of the triplets into a JSON file in order to render them graphically with D3js.

Finally, the file named *WebAssembly* is a *npm package* that contains all the files needed to run the project on web browsers. By starting a server by using the command *npm start run* in it, you'll be able the visualize the content of the folder *WebAssembly/public* in your web browser at *localhost:9090*. Details related only to D3js can be found in the folder of the same name in *WebAssembly/public/D3js*.

Additional features

We have chosen as additional features to optimize for size in memory and also allow dynamic size of values. The initial situation was such that we allocate 3 * 1024 bytes for each new triplet, even if there are duplicates among the elements. For example, among the data that are parsed in the benchmark, there's 117'731 unique words out of the 473'016 inputs. In other words, it exists 355'285 duplicates in the data set.

This is the reason why we choose to optimize for size in the memory by creating on one side a collection of words without duplicates and on the other side a structure *s, *p, *o which only takes pointers to the corresponding words in the word collection. To go further in this approach, we also used and integrated a dynamic allocable size to each element where instead of allocating 1024 *char* to each word we could allocate the length of the word using the "strlen()" function multiplied by the size of a *char* which is by definition 1 byte.

Advantages and disadvantages of the implementation decisions

Our implementation decisions were to optimize memory space to its maximum. Of course, this was done at the price of writing performance. However, this had no impact on reading performance.

Indeed, from now on when inserting new elements, we iterate on each one of the words of the word collection in order to check if the element already exists, or if it's necessary to allocate additional memory to the word collection in the program and then save the new element to it.

While inserting data to the program became a bit long, we've got the intuition that the huge memory optimization makes it worth because on a real project memory usage could be a lot more important than inserting speed.

The results obtained are set out with more detail in the following and last chapter.

BENCHMARK SCORES

The table below compares program performance before and after memory optimization. The trade-off between speed or memory size is well highlighted.

Standard Version (without any optimization)	**Memory Optimization Version**
Outputs • insert() took 1'419 milliseconds **~ 1,5 seconds** • match(s, _, _, _) took 63 milliseconds • match(_, p, o, 2) took 11 milliseconds • match(s, p, o, 0) took 6 milliseconds	***Outputs*** • insert() took 479'696 milliseconds **~ 7,9 minutes** • match(s,,,_) took 16 milliseconds • match(s,p,o,2) took 7 milliseconds • match(s,p,o,0) took 5 milliseconds
Memory usage estimation: **~ 486.8 mb**	Memory usage estimation: **~ 24.14 mb**

All the details of the calculation are developed more precisely in the Readme.md file at the root of the project.