DIY PATTERN MINING

WHAT'S UNDER THE HOOD

Mykola Rabchevskiy
Apr 2 1

This chapter is about implementing the Chronicle module, which provides the memorization of the sequence of events and discovering new temporal patterns on the fly as described in the previous chapter. Test/demo application demonstrates the principles and capabilities of the Chronicle module. Sources repository:

https://github.com/mrabchevskiy/chronicle

The application uses ASCII encoded text as a sequence of pseudo-events, where the role of an elementary event is played by a Latin alphabet character or a punctuation mark.

The test program is a console application. By running the executable included in the project, you will receive information on the progress of the process, statistical information, and a list of the longest detected patterns on the terminal window:

```
Sources from txt

    398643 bytes  txt/JulesVerne.txt
    594362 bytes  txt/SherlockHolmes.txt
    312656 bytes  txt/Kipling.txt

Process `txt/JulesVerne.txt`
Processed     385755 symbols

Process `txt/SherlockHolmes.txt`
Processed     964672 symbols

Process `txt/Kipling.txt`
Processed    1267936 symbols

Total 1267936 symbols processed in 75481195.98 msec ~ 59.53 microsec/symbol
Sequence length                    506451 elements ~ 96.60 % of capacity
Compacted                               0 times
Gap                                  7467
Total number of patterns             7562
Total number of views                8189
Distinct elements in the sequence    6843
Cases witch continuations             49.87 %
Sequence compression ratio             2.50

Max pattern` length: 30

Top 100 longest patterns:

  1 `and he thumped and he thumped `
  2 `and the next, and the next, `
  3 `of the forty-pounder train!`
  4 `perhaps--and perhaps--and `
  5 `the beaches of lukannon--`
```

Information about entities is stored in Semantic storage; in the test application, a minimalistic version of the semantic storage implementation is used. The entire semantic storage code is contained in the application code.

A detailed code revision is given below, but first, a few words about the implementation principles common to all modules.

Each entity, one way or another involved in the process, is typically represented internally by its numeric ID and some `external` representation. The unique entity ID is generated randomly the first time the entity is mentioned. Using these random IDs allows them to be used as a hash value of the correspondent entity; eliminating the need to compute a hash value speeds up data access to data stored in hash tables that uses ID as a key.

The stored sequence is formed of ID that refers to atomic entities or patterns.

Entities in our case are ***atomic*** pseudo-events (letters), ***patterns*** (sequences of letters), and

views. The ***view*** represents a ***pattern*** as a pair of ID that refers ***head*** part and ***tail*** part of the pattern; both head and tail can be an atomic entity (i.e., single letter) or pattern (sequence of letters).

Relations between atomic entities and their ID are stored in the array and hash table that provided two-way mapping.

In the same way, two other hash tables provide a two-way mapping between pattern ID and pair of IDs that form one of the pattern` view; so the view is not a full-fledged entity (no ID that represents a whole view) - it is a specific of simplified semantic storage.

Patterns discovered during processing text input are stored in the text output file.

As was described in the previous chapter, patterns are base on forecasting. In our case, this means that in the particular moment, if the current ***tail of the whole sequence*** matches the ***head of some view***, then the ***tail of this view*** is one of the possible text ***continuations***. Such data stored in the second output file: for each tail of the whole sequence with at least one continuation, provided a list of all possible continuations, ordered alphabetically. Such data can be used for predicting future text input.

Test/demo application implemented as a console application with no external dependencies, so it can be used without any changes under any operating system.

Application processes sequentially all ****.txt*** files from the subfolder `***txt***` of the folder where executable is located. The project doesn't contain such a folder, but archive ***texts.zip*** is included. When shell script ***run.sh*** executed, it makes `***txt***` folder and unpacks three text files with books, converted to ASCII; total size is around 1.25Gb. You can add/replace files in `***txt***`; converting text files from utf8 to ASCII can be done by utility `***uni2ascii***`. The archive can, of course, be manually unpacked. Output files ***pattern.txt*** and ***continuation.txt*** are overwritten each time.

TECHNICAL DETAILS

We use the latest C++ standard (C++20), Linux (Ubuntu), CodeBlocks IDE, and gcc++-10 compiler. But any C++20 - compliant compiler and any operating system can be used, as well as Windows System for Linux (WSL). The repository contains an executable for Linux (64 bit) that can also be run using WSL. In other cases, recompilation will be required. Script `***compile.sh***` can be used for compilation on Linux and WSL (in case of using WSL, you need to install compiler ***gcc++-10***). Note that compilation can be pretty time-consuming if the available RAM is less than 8GB.

Project files:

- ***chronicle*** - executable test application
- ***texts.zip*** - archived test input data
- ***run.sh*** - shell script that launches an application
- ***chronicle.cpp*** - test application code
- ***chronicle.h*** - implementation of the Chronicle module
- ***compile.sh*** - compilation script for GCC

Utility modules/classes:

- ***arena.h*** - arena memory allocator
- ***codec.h*** - converting ID into BASE64-like text and performs the inverse transformation
- ***def.h*** - types definition and some functions
- ***flat.h*** - the hash map that occupies a continuous piece of memory
- ***queue.h*** - fixed size, circular buffer-based queue
- ***random.h*** - adapter to standard C++ random number generator
- ***timer.h*** - time measuring utility

Chronicle class` constructor requires four parameters:

Chronicle(Lex lex, Sticky sticky, Act make, Act hunt)

All these parameters are fuctions:

using Lex = std::function< std::string(const Identity&) >;
using Act = std::function< Identity (const Identity&, const Identity&) >;
using Sticky = std::function< bool (const Identity&, const Identity&) >;

Identity type defined in def.h and represents entity's ID:

using Identity = uint32_t;

Function ***lex*** produces the name of the entity with the provided ID.
Function ***sticky*** accepts IDs of two consecutive entities and returns true if it is allowable to `glue` two entities into a new pattern.
Function ***make*** accepts two entities ID and returns the ID of the newly constructed

pattern.
Function ***hunt*** accepts two entities ID and returns true when exists pattern consisting of the two referred entities.

Methods:

bool incl(const Identity& id) - extends the sequence by adding one more element.
explicit operator bool() const - returns true is sequence is not empty.
bool contains(const Identity& id) const - check if sequence contains provided element at least once.
unsigned num(const Identity& id) const - returns a number of occurrences of the provided ID.
bool process(std::function< bool(const Elem& e, unsigned i) > f) - calls ***f(..)*** while it returns ***true*** for sequence elements starting from the oldest one.
void expo() const - prints sequence (should not be used in case of voluminous sequences).
void statistics() const - prints statistics.
bool consistent() const - check inner data consistency (for debugging).
bool save(const char* path) - saves state as a text file, returns ***false*** on failure.
int load(const char* path, std::function< bool(Identity) > exist) - loads state from file; function ***exist*** confirms that referred entity actually presented in the semantic storage; negative return mean error.
void reset() - re-initialize sequence.
unsigned compact() - removes `gaps` (empty elements) and returns number of eliminates gaps.

Our ***coding rules*** aim to reduce the number of logical connections between source code files and increase the part of the code that can be captured by the gaze on the screen, so the code's formatting may not look completely familiar (note also that we use ***tab-width*** = 2).

The ***commenting rules*** are also not usual: we assume that the main document is the code itself and not the documentation automatically generated from special comments in the code. Accordingly, the comments do not duplicate the code but only explain it.

Finally, our entire AGI library is implemented as a set of header files. Using the `***Chronicle***` in other projects requires you to include header files `***chronicle.h***`, `***codec.h***`, `***def.h***`, `***flat.h***` and `***queue.h***`.

Important note: we believe that removing ***assertions*** after debugging is similar to

removing lifejackets from a ship after testing is complete; our code ***requires assertions always activated to run correctly***. We use statements like this:
assert(**func**(**arg**));
So in the case of deactivated assertions, function `***func***` that returns `***true***` as confirmation of correctness is just not called. ***Visual Studio***, by default, turns assertion off for ***Release*** mode, so you need to remove the ***NDEBUG*** option from the compiler options.

Another aspect of ***Visual Studio***: C++ compiler/linker by default assigned too small stack size that is insufficient for our project; option ***/STACK:16000000*** should be added to linker options to avoid stack overflow. It is also helpful to turn off ***automatic code formatting*** and set ***tab-width=2***.

We do not recommend ***Visual Studio*** because executables produced by ***VS*** are ***twice slower*** than executable produced under Windows Subsystem for Linux on the same computer.

Good luck - and let us know in case of any problems (using comments to post).