

Netbench – Design Document

Authors: Viktor Puš (ipus@fit.vutbr.cz), Vlastimil Košar (ikosar@fit.vutbr.cz),
Jiří Tobola (itobola@fit.vutbr.cz)



INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Introduction

This document describes the design of the Netbench experimental framework from the programmer's point of view. The aim of the framework is to serve as independent platform for researchers seeking the easiest way to implement their algorithms, as well as a comparison of their algorithms with reference implementations of the other approaches.

Netbench focus is in the field of longest prefix matching, packet classification and regular expression matching. There are many approaches to IP lookup, packet classification and RE matching. They vary in the technology used. The list of common technologies includes software implementation for general-purpose processors, network processors, implementations for graphics processing units (GPUs), FPGA and ASIC designs, TCAMs. While all of these technologies are rapidly innovated, **algorithms** for IP lookup, packet classification and RE matching are often independent on the technology, and may only benefit from technology improvements.

The following chapter briefly introduces the Netbench design, while three chapters then describe separate parts in more detail.

Content

Introduction.....	1
Netbench Design.....	1
Common Classes.....	2
LPM.....	2
Algorithms.....	4
Data Sets.....	4
Example Use.....	4
Classification.....	4
Packet Classification Algorithms.....	4
Data sets.....	5
Example Use.....	5
Pattern Matching.....	6
Algorithms.....	7
Data Sets.....	8
Example Use.....	8
Further Reading.....	8

Netbench Design

The Netbench framework is built around a set of Python 2.6 classes. There are common classes for the representation of frequent objects (packet, packet header, prefix, symbol, ...). Based on these classes, models of various algorithms are built. Algorithms also often use the helper classes to load inputs from files (PCAP parser, classification rules parser, ...). There are also packet traces and example rule sets for instant algorithm testing. Fig. 2 Shows the basic idea behind Netbench.

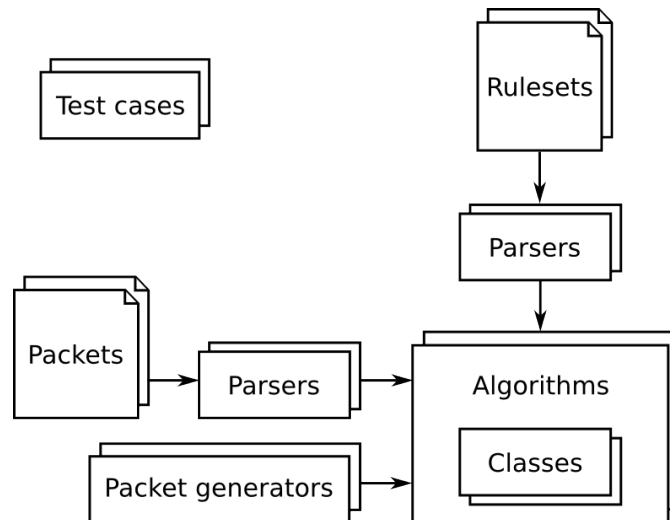


Figure 1: Netbench design

Common Classes

Common classes are stored in the common directory. The structure of these classes is shown in Fig. 2.

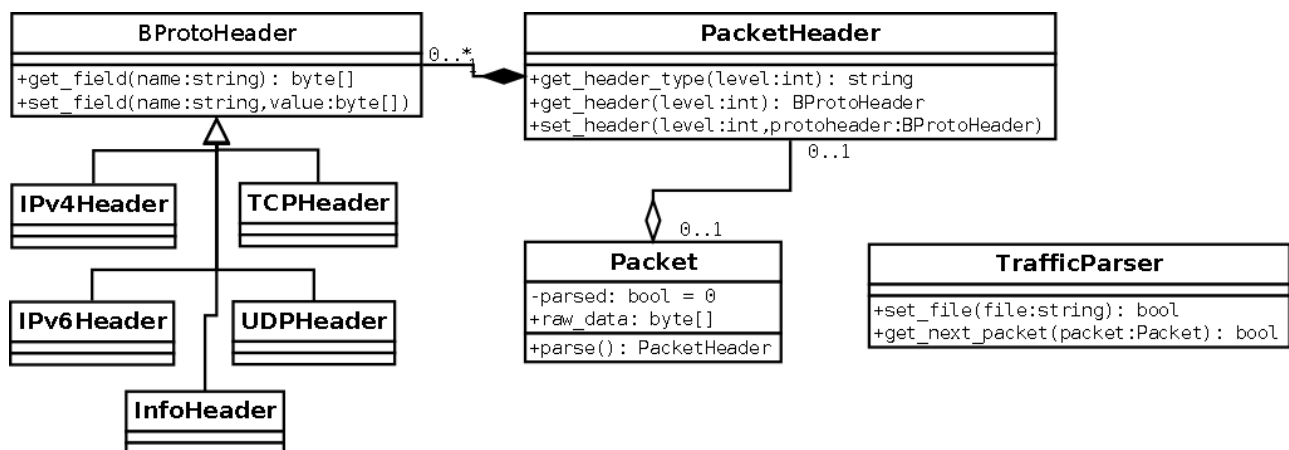


Figure 2: Diagram of common classes.

LPM

Basic classes used in Longest Prefix Match / IP lookup are stored in the *lpm/* directory. The *lpm/algorithms* directory contains several LPM algorithm models together with the test script. The basic structure of LPM classes is shown in Figure 3.

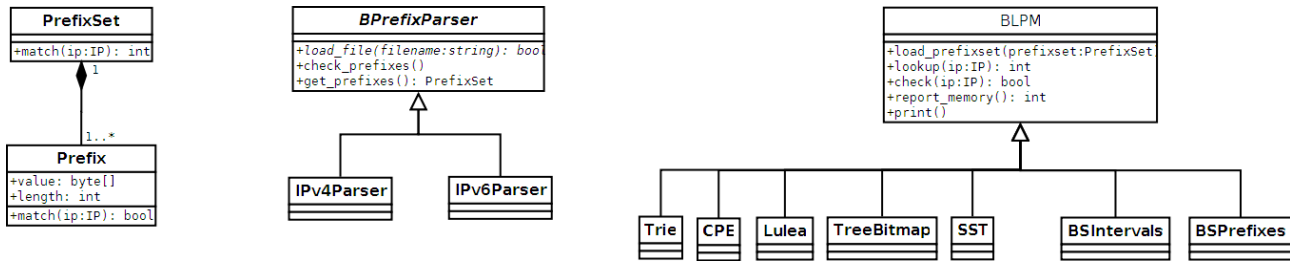


Figure 3: Diagram of LPM classes.

The most important class for the longest prefix match is the BLPM, which is used as a base class for all LPM algorithms. All derived classes must implement three basic methods:

- *load_prefixset()* is used to load one prefix set into the algorithm and to generate all necessary data structures for the LPM operation.
- *lookup()* uses the generated data structures to perform IP lookup operation on given IP address. It returns the list of matched prefixes starting with the longest one.
- *report_memory()* prints text information, which depends on the nature of the algorithm. The information may contain numbers and sizes of data structures, depth of tree etc.

BLPM moreover implements one final (leaf) method *check()* which compares result of *lookup()* method and standard linear search (validation of new methods).

Algorithms

Algorithms for IP lookup are usually tree-based, because prefix directly corresponds to the descent path in binary tree. However, Netbench implements also approaches based on range searching or hashing.

Unibit Trie is a basic memory structure which includes all prefixes directly in its construction. Each node may contain prefix and up to two pointers to the child nodes. One input value bit is processed in each cycle during the Trie algorithm run. According to this bit the next node is the left (with bit = 0) or right (with bit = 1) successor until there is no continuation possible. The longest prefix is the last matched node in the Trie. The main disadvantage of this algorithm is low matching speed given by the high number of processing steps. Therefore many following algorithms focus on speed (and memory) optimizations.

Controlled Prefix Expansion (CPE) algorithm introduces multiple bits processing per clock cycle and therefore higher algorithm speed. Processing of more input bits is enabled by prefixes expansion to requested processing width, which leads to higher memory requirements. The leaf pushing optimization decreases the memory requirements typically to one half of the original memory structure.

Lulea starts with a conceptual leaf pushed expanded trie and replaces all consecutive elements in a

trie node that have the same value with a single value. This can greatly reduce the number of elements in a trie node. To allow trie indexing to take place even with the compressed nodes, a bitmap with 0's corresponding to the removed positions is associated with each compressed node.

LC-Tries (LCT) are another form of compressed multibit tries. The main idea behind LC-Tries is to recursively pick strides (and therefore multibit tries) that result in having real prefixes at all of the leaves. Having fully populated leaves means that none of the internal prefixes need to be pushed to the leaves of the multibit tree or to lower trees (no leaf pushing). In addition to this strategy, LC-Tries use path compression to save space on non-branching paths.

Tree Bitmap (TBM) structure is focused on hardware suitable implementation. Therefore it doesn't use LC-Tries scheme with variable processing steps, but it further optimizes memory usage of Lulea structure. Each compressed node of the structure represents 2^n unibit trie elements and stores only pointers to the first successor node and the first prefix. Next successors and prefixes follow in the memory, so their positions are given by the bitmaps.

Hash-Tree Bitmap algorithm (HTBM) is focused on longer IPv6 addresses processing and reduction of number of Tree Bitmap steps. This is done by hash functions which serve as shortcuts in the trie memory structure. The resulting algorithm run consists of three stages: parallel multiple hash functions, longest matched prefix selection and consequent matching using the Tree Bitmap algorithm.

Shape Shifting Trie (SST) starts with the Tree Bitmap structure and introduces memory optimization for long non-branched segments. Each node can represent the standard balanced tree or any other "shape" which is more suitable for given prefix set and allows to move faster and with better memory utilization through the prefix space.

Binary Search on Ranges (BSR) treats a database of prefix addresses as a set of intervals. It expands all prefixes to full length and builds a sorted table of intervals limits where more specific intervals have higher priority than less specific ones. Binary search or B-Tree search is subsequently used to find the match for specific input IP address.

Binary Search on Prefix Lengths (BSP) is based on the idea of modifying a prefix table to make it possible to use hashing. The hashing is used to search among all entries of a given prefix length. Instead of searching every possible prefix length and picking the longest prefix length with a match, binary search is used to reduce the number of searches.

MultiMatch is trivial hardware algorithm which splits input prefixes to n subsets with the same length. Shorter prefixes are expanded to get requested length. After that, parallel hash searches are used to find the LPM in the group and the longest positive search is the operation result.

Data Sets

We gather and add to Netbench several data sets. For IP lookup, we focus on three target data sets -- IPv4 routing tables, IPv4 firewall tables and IPv6 routing tables. After further deployment of IPv6 to end user networks the IPv6 firewall tables will be added.

Most of the new LPM algorithm's authors use BGP tables from Potaroo, but unfortunately the AS numbers, table release dates and numbers of currently available prefixes change in time. Therefore we download available IPv4 and IPv6 BGP tables and store them on regular time basis. As a result, any author can use the same data sets for comparison of introduced approach to other algorithms. Some more BGP table sources are also included.

As IP lookup is the important step of decomposition based classification methods, we use the same data sets for firewall tables as in the packet classification section. Both source and destination IP addresses are extracted from the rules and obtained sets form a benchmark for firewall IPv4 tables.

Please follow README in *lpm* directory for more information.

Example Use

Please follow README in *lpm* directory.

Classification

Basic classes used in packet classification are stored in the *classification* directory. The *classification/algorithm* directory contains several models of packet classification algorithms together with the test script. The basic structure of classification classes is shown in Figure 5.

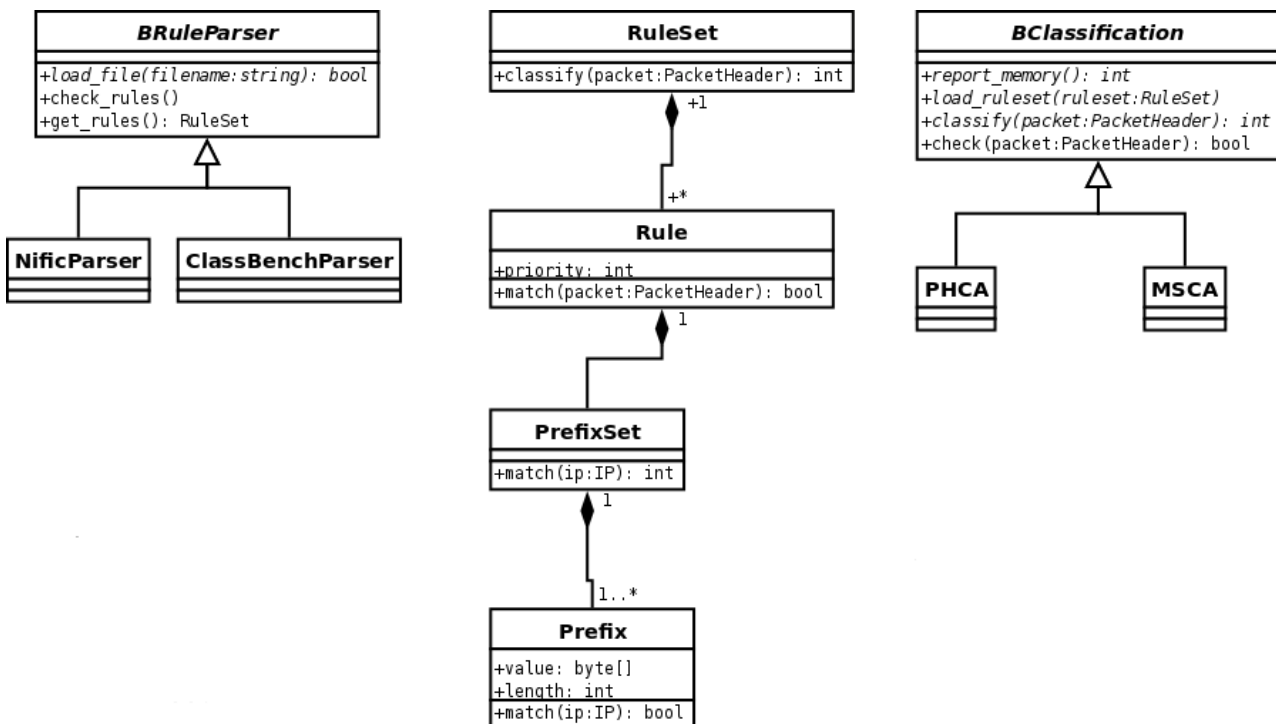


Figure 4: Diagram of classification classes.

The most important class for the packet classification is the *Bclassification*, which is used as a base class for all packet classification algorithms. All derived classes must implement three basic methods:

- `load_ruleset()` is used to load one rule set into the algorithm and to generate all necessary data structures for the classification.
- `classify()` uses the generated data structures to classify one packet. It returns the

selected rule.

- `report_memory()` writes text information, which depends on the nature of the algorithm. The information may contain numbers and sizes of data structures, depth of tree etc.

Packet Classification Algorithms

Distributed Crossproducing of Field Labels (DCFL) by Taylor and Turner modifies the LPM to return all valid prefixes (not only the longest one) for the given field value. What follows is the hierarchical structure of small crossproduct engines. Inputs of each engine are two sets of prefixes (or Labels, in general). Engine then performs set membership query for each possible pair of Labels. Result of the engine is another set of Labels. The result of the last engine is in fact a set of rules, from which the one with the highest priority is selected.

Multi Subset Crossproduct Algorithm (MSCA) by Dharmapurikar et al. provides heuristics on how to break rule set into several subsets, eliminating the memory requirements significantly. The paper also identifies rules that generate excessive amount of memory. These rules are called spoilers and are treated in a separate algorithm branch to further reduce the memory.

Perfect Hashing Crossproduct Algorithm (PHCA) improves MSCA by using specifically constructed hash function to map all possible results of the LPM stage directly onto the correct rule. Considerable amount of memory is saved this way. Perfect (collision-free) hash construction algorithm is used, but in this case, many collisions are intentionally introduced to create many-to-one mapping of LPM results to rules.

Prefix Filtering Classification Algorithm (PFCA) improves PHCA in terms of memory. It is based on the observation that many rules does not specify condition for all dimensions. PFCA finds generalization rules to avoid meaningless combinations of the LPM stage results.

Prefix Coloring Classification Algorithm (PCCA) further improves PHCA. The LPM stage is extended by adding abstract color property to prefixes. Prefixes now contain bitmaps of allowed and suppressed colors of prefixes from other dimensions. Simple logic is then used to filter out most of unwanted combinations of LPM results. This lowers the perfect hash table size significantly, while the throughput is not affected.

Multi Subset Prefix Coloring Classification Algorithm (MSPCCA) combines MSCA and PCCA to obtain even lower memory requirements while keeping the same throughput as in PCCA.

Hi-Cuts algorithm takes completely different approach. It constructs a decision tree which divides the state space into several smaller ones in each node. Packet classification then equals to the tree descent. Leaf nodes contain several remaining rules which are processed sequentially.

Data sets

The `classification/rulesets` directory contains several rule sets generated by ClassBench and also some very simple rule sets for early experiments and debugging. All rule sets are stored in the NIFIC format, which is used by the parser in `classification/parsers`. The rule format is also described in that directory. For testing purposes, packets from `common/test.pcap` are used.

Example Use

In the `classification/algorithms` directory type

```
python2.6 test.py dcfl ../rulesets/simple2.rul
```

to run a test script. This script creates an instance of the selected algorithm, loads it with the selected rules and then runs classification of packets from the `common/test.pcap` file. The outputs of the algorithm are compared to the outputs of linear search in rules. In case of mismatch, an error is reported.

Pattern Matching

Basic classes used in pattern matching are stored in the `pattern_match` directory. The `pattern_match/algorithms` directory contains several implementations of pattern match algorithms together with the test script. The simplified basic structure of pattern match classes is shown in Figure 5.

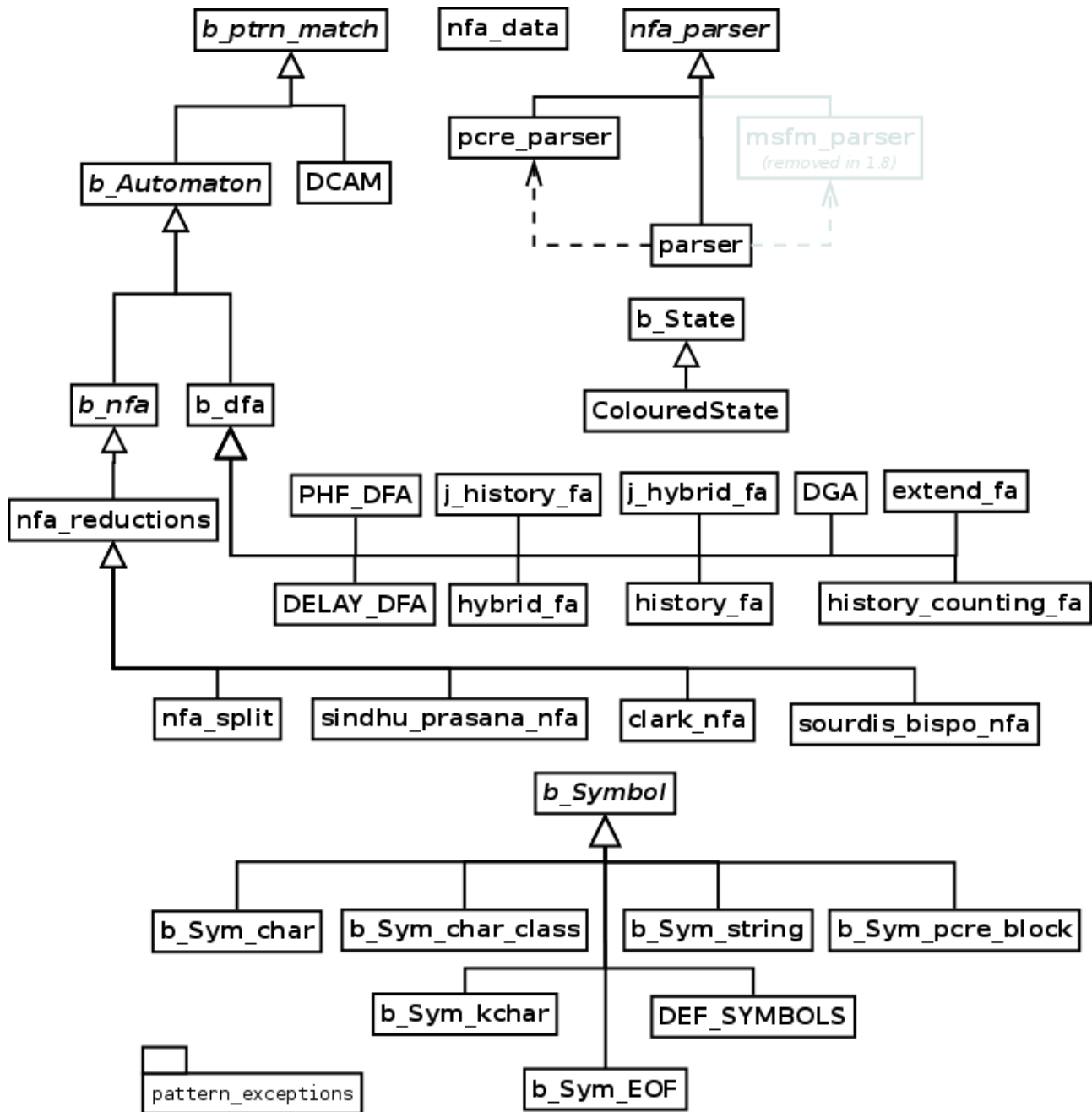


Figure 5: Simplified diagram of pattern match classes.

Pattern matching in Netbench is mainly focused on algorithms for regular expression (RE) matching based on nondeterministic finite automata (NFA) and on deterministic finite automata (DFA), although some basic support for string matching algorithms exist.

Netbench provides several sets of classes to support those algorithms. Parser classes (`pcre_parser` and `msfm_parser` [removed in Netbench 1.8 as deprecated]) based on abstract class `nfa_parser` provides access from python to RE parsers. Class `parser` provides cover over various parser classes and independent interface to various parser classes. Class `nfa_data` store an automaton and provides

basic methods for automata construction. This class is used internally and also for inter algorithm exchange of automata. States are represented by *b_State* class or by *ColouredStates* class, when state coloring is needed. Symbol classes are based on abstract class *b_Symbol*, which provides common interface for all symbols. There are classes for various symbols. Class *b_Sym_char* represents single ASCII char or epsilon, class *b_Sym_char_class* represents character class, class *b_Sym_kchar* implements strided symbol which can have any stride and each subsymbol can be either ASCII char or char class, *b_Sym_EOF* represents end of input symbol, class *DEF_SYMBOLS* implements default symbol, class *b_Sym_string* represents string of ASCII characters and finally *b_Sym_cnt_constr* represents PCRE counting constraint. Abstract class *b_ptrn_match* provides common interface for any pattern matching algorithm. Abstract class *b_Automaton* implements many common algorithms for finite automata and provides means for automatic automata construction via parser objects. Any implemented algorithm based on NFA should be based on *b_nfa* class and any algorithm implementation based on DFA should be based on *b_dfa* class. NFA based algorithms can also be based on class *nfa_reductions*. The class *nfa_reductions* provides various algorithms for reduction of NFA.

New algorithm should be based on correct class and must implement those methods:

- `compute()` - implementation of algorithm. Should expect that parsed REs were already loaded.
- `report_memory*()` - all DFA based algorithms must implement these methods – minimal and maximal limits of utilised memory for this algorithm and /or real number of utilised memory if exact mapping of transition table to memory is specified.
- `report_logic()` - all NFA based algorithms must implement this method. Reports amount of utilised logic for FPGA,
- `get_state_num()` and `get_trans_num()` - should be overloaded if necessary. These methods reports number of states and number of transitions.
- `search()` - should be overloaded if necessary. `Search()` method performs pattern matching on input string.

Algorithms

Dedicated decoders for each transition (*sindhu_prasana_nfa*) were introduced by Sindhu and Prasana. Each transition has dedicated character decoder and each state is implemented by FlipFlop (FF). Our implementation of the algorithm also supports strided Nondeterministic finite automata (NFA).

Algorithm by Clark et al. (*clark_nfa*) improves previous algorithm by introduction of shared character decoders, prefix sharing and NFA striding.

Architecture by Sourdis and Bispo (*sourdis_bispo_nfa*) improves previous algorithm by introduction of character class sharing, separate matching of string subpatterns of RE by DCAM string matching algorithm and special units for implementation of PCRE counting constraints.

The Nfa Split architecture (*nfa_split*) compute sets of states in collision and then splits NFA according to those sets into deterministic and nondeterministic parts of NFA. Two states are in collision if they can be active at once. The deterministic parts are implemented as DFA and the nondeterministic part are implemented as NFA.

Delay DFA (*DELAY_DFA*) by Kumar et al. introduce default transitions, that do not accept any character and connect states with similar outgoing transitions. Delay transitions exploits fact, that many states in DFA have very similar sets of outgoing transitions. Since the default transition does not accept any symbol, the throughput of the DDFA may be slower than DFA but the time complexity of accepting one symbol remains constant.

History DFA (*history_fa*) by Kumar et al address one of the DFA problems – forgetting history of matching. To solve this problem, Kumar introduces additional memory to store history of the matching. Content of this memory can be used as a condition on the transition of the automaton.

History DFA with counting (*history_counting_fa*) by Kumar et al. address another DFA problem – counting constraints in PCRE. Additional counters are introduced by Kumar to store the number of repetitions. Those counters can be used as condition on the transition of the automaton.

Hybrid finite automaton (*hybrid_fa*) by Becchi et al. solve problem of exponentially larger memory requirements of DFA than NFA. DFA construction algorithm stops as soon as state blow-up is recognized. The remaining parts of the automaton are left as NFA or transformed into new DFAs. This method provides speed and memory consumption trade-off.

Sparse transition table of DFA can be effectively implemented using Perfect hash function (*phf_dfa*). PHF can be created as collision-free for given set of keys (transitions of DFA in this case). However, collisions occur for unknown keys – the non-existent transitions. Therefore the value of each key must be stored in the PHF table for the validation of the transition.

Classes *j_hybrid_fa* and *j_history_fa* implement experimental variants of *hybrid_fa* and *history_fa* algorithms. Those variants use both RE and FA to construct the pattern matching algorithm.

Class *DGA* implements experimental pattern matching algorithm based on deterministic generalized automaton..

Experimental class *extend_fa* currently implements only NFA based variant of this pattern matching algorithm.

Data Sets

The `pattern_match/rulesets/*` directories contain sets of regular expressions (RE) from various real systems such as Snort and Bro IDS and L7 project. Directory `L7` contains set of RE from the L7 project. Directory `Snort` contains sets of RE from the Snort IDS. Set of RE from the Bro IDS is in directory `Bro`.

The `pattern_match` part of Netbench provides parser of RE. The PCRE parser is stored in directory `pattern_match/pcre_parser` and supports wide range of PCRE features. This parser can be used from python code via the *pcre_parser* class. The deprecated old parser and python *msfm_parser* class were removed from Netbench 1.8.

Example Use

In the `pattern_match/algorithms/<algorithm>` directory type

```
python2.6 example_of_use.py
```

to run an example script. This script creates an instance of the selected algorithm, loads it with set of regular expressions and then creates a representation of the algorithm.

Makefile System

Pattern match part of Netbench uses Makefiles for easy usage of the library. Makefile target *doc* opens documentation, *run* executes the above mentioned example of use, *test* runs unit tests and *clean* removes generated files. Makefiles are located in algorithms directories and in parent directories of algorithms: `pattern_match`, `algorithms` and `experimental`. Makefiles in parent directories executes the targets for all child directories.

Further Reading

- Most directories in Netbench contain `README` file with detailed description of that directory. This is especially true for the netbench root directory, where the instructions about the environment settings are.
- All Netbench classes are documented by the Doxygen-like Sphinx tool. The generated documentation in HTML format can be found in the `doc` directory.
- There is motivation document `netbench.pdf` stored also in the `doc` directory.