As part of my independent work (described in my last post), I’ve been looking at the different types of pattern matching that’s permitted in flow tables for network switches. If you recall, the flow tables operate by matching packet fields on different patterns and performing the action associated with the matching pattern. The actions typically look like: DROP, FORWARD to port n, etc. Here’s a sample table:

[01\*\*] 🡪 DROP

[001\*] 🡪 FWD

[0\*\*\*] 🡪 DROP

[1\*\*\*] 🡪 FWD

In general, the patterns come in one of four varieties:

**Exact**: Self-explanatory. The packet header needs to match the predicate exactly.

**Range**: In this case, the packet header must lie in some range, e.g. [0,100] would match any packet header with a value between 0 and 100.

**Prefix-based**: These predicates are trailed by wildcard characters ‘\*’, which indicate that you need to match exactly up until the ‘\*’, after which you can have whatever you like. For example: if your pattern is ‘10\*\*’, then this would match ‘1000’, ‘1011’, ‘1010’, and ‘1001’.

**Ternary**: In this case, you can have ‘\*’ wildcards at *any* position. For example: if your pattern is ‘1\*01’, then this would match ‘1101’ and ‘1001’.

I’ve been focusing on the last two variants, which are closely related. In fact, I want to talk about two algorithms for compressing rules tables with these predicate formats, both of which must be credited to Alex Liu at the University of Michigan.

The Dynamic Approach

We’re going to focus on a single dimension (i.e., matching on a single field). This problem has been solved for the prefix-based case. The algorithm takes a dynamic programming approach, defining the concept of *consistency*: a rule table is *consistent* on a predicate *P* if, for every possible packet that matches *P*, the same rule is executed. For example, if your rule table is simply [\*] 🡪 DROP, then your table is completely consistent on ‘\*’.

From here, the algorithm starts with the predicate ‘\*’. If the rule table is consistent on this predicate with action *A*, it returns a rule table with the single rule [\*] 🡪 *A*. Else, it recurs on the predicates ‘1\*’ and ‘0\*’, optimizing the rule table on these sub-predicates and finally combining the solutions.

There is some care that’s needed for combining the two sub-solutions, but overall it isn’t too messy.

For the one-dimensional case, this dynamic programming algorithm manages to minimize the size (i.e., the cost) of the rule table.

Bit-Weaving

To solve the ternary case, Liu employs some ingenuity (the inspiration for this blog post). Instead of tackling it as an entirely new problem, he manages to **reduce the ternary case to the prefix-based case.**

The essence of this algorithm is that he cuts up the rule table carefully and then considers it to be a matrix in which each row is a predicate and each entry *(i, j)* is the *j*th character in the *i*th predicate. With that established, he then performs some swaps in the columns in order to convert from ternary to prefix-based format. From there, he can run the dynamic programming algorithm described above and convert back to ternary format by undoing his row swaps.

For example: if you have predicates [01\*00] and [00\*11], you would swap the third and last columns to get predicate [0100\*] and [0011\*], which are then in prefix-based format.

The algorithm is very clever—I’m a huge fan of reducing one problem to another.