

OPTIMISATION OF POLICY-BASED INTERNET ROUTING USING ACCESS CONTROL LISTS

Vic Grout and John McGinn

Centre for Applied Internet Research, University of Wales
NEWI Plas Coch Campus, Mold Road, Wrexham, LL11 2AW, UK
{v.grout|j.mcginns}@newi.ac.uk

ABSTRACT

This paper considers an optimisation problem encountered in the implementation of traffic policies on network routers. The problem is formulated and shown to be NP-complete. Exact and heuristic solution methods are introduced and compared and computational results given. Additional complications and extensions are considered in conclusion.

1. INTRODUCTION: ACCESS CONTROL LISTS

An *Internetwork (Internet)* is a network of networks. Key devices known as *routers* switch, or *route*, communications traffic, usually in the form of discrete *packets*, between networks. Routers are responsible for correct and appropriate delivery of packets from source to destination through the use of *routed* and *routing* protocols (or manually defined *static routes*) and the application of *policies*. The primary function of a router is to forward each packet to the most suitable device, often another router, at each step (*hop*) of the journey. However, a vital secondary role is to consider whether a given packet should be passed at all, according to a set of tests, or *rules*, against which it may be matched.

A typical rule, in the syntax of the Cisco *Internetwork Operating System (IOS)* (Colton, 2002), might be:

```
access-list 101 deny icmp any 10.0.0.0 0.255.255.255 echo-reply
```

This states that ICMP echo-reply packets from any source to the network 10.0.0.0 are to be blocked at this point. The first part of the rule assigns it to access list 101. An access list, or *Access Control List (ACL)*, is then a sequence of such rules designed to implement a given objective or set of objectives. ACLs can be used simply to pass or block packets or as filters for more sophisticated policies such as traffic shaping, queuing or encryption. An example of a complete ACL is given in Figure 1. Other than the ACL assignment, a rule may consist of up to five parts: the `permit` or `deny` type, the protocol, a source address, destination address and a flag function (as in the `echo-reply` parameter above) for fine tuning. Each parameter may be a single value or a range of allowable matches. For example, the `any` parameter above matches all source addresses whilst the `0.255.255.255` parameter matches destination addresses in the 10.0.0.0 network. (Although the simple examples given in this section may appear to imply *classful* routing, the techniques discussed in this paper are fully suited

to *Classless Inter-Domain Routing (CIDR)* applications. However, a discussion of *Variable Length Subnet Mask (VLSM)* principles would extend this paper unnecessarily and can be found elsewhere (Colton, 2002.)

The rules of an ACL are processed in order. That is, each incoming packet is tested against the first rule; if it matches, it is passed or blocked accordingly and no further rules are considered; otherwise it is tested against the second rule, and so on. There is an implicit `{deny all}` rule at the end of each ACL to block all packets not otherwise matched. Some rules are more likely to match packets than others and, depending on the method of implementation, some rules may take longer to process than others (for example if multiple parts of protocol units at different layers have to be examined). The time to process an ACL is then the total time taken to test a packet against each rule up to and including the one it matches.

```

access-list 101 permit tcp 192.168.212.0 0.0.0.255 10.0.0.0 0.255.255.255 eq telnet
access-list 101 permit tcp 192.168.212.0 0.0.0.255 10.0.0.0 0.255.255.255 eq ftp
access-list 101 permit tcp 192.168.212.0 0.0.0.255 10.0.0.0 0.255.255.255 eq http
access-list 101 deny ip 192.168.212.0 0.0.0.255 10.0.0.0 0.255.255.255
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 administratively-prohibited
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 echo-reply
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 packet-too-big
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 time-exceeded
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 unreachable
access-list 101 permit icmp 172.16.20.0 0.0.255.255
access-list 101 deny icmp any any
access-list 101 permit ip 202.33.42.0 0.0.0.255 any
access-list 101 permit ip 202.33.73.0 0.0.0.255 any
access-list 101 permit ip 202.33.48.0 0.0.0.255 any
access-list 101 permit ip 202.33.75.0 0.0.0.255 any
access-list 101 deny ip 202.33.0.0 0.0.255.255 any
access-list 101 deny tcp 210.120.122.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.183.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.114.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.175.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.136.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.177.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 permit tcp any 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp any any eq www
access-list 101 permit tcp any any
access-list 101 deny ip 195.10.45.0 0.0.0.255 any
access-list 101 permit ip any any
{access-list 101 deny all} {implicit}

```

Figure 1. An Access Control List (ACL)

Whatever the purpose of an ACL, it is clearly advantageous to have the rules ordered in such a way as to minimise, or at least reduce, processing time. However, the relationship between rules prohibits arbitrary reordering. For example, in Figure 2, an IP packet from network `192.168.16.0` to network `10.0.0.0` will match both rules shown. The packet will be passed in 2(a) but blocked in 2(b). Clearly then, rules may not be reordered if this changes the underlying intention of the policy.

We proceed now to a formal development of the problem, which is essentially to find the optimal ordering of the rules of an ACL that satisfies the original policy. There has been some work of limited scope in relation to these issues (Stoica, 2001 and Cisco, 2002). Bukhatwa and Patel (2003), for example report significant

improvements from manual reordering of rules. This paper, however, undertakes a deeper study, on a larger scale, suitable for implementation within the router IOS or embedded in hardware. We consider rule hit-rates, latencies and variable traffic flow in the optimisation of ACL order.

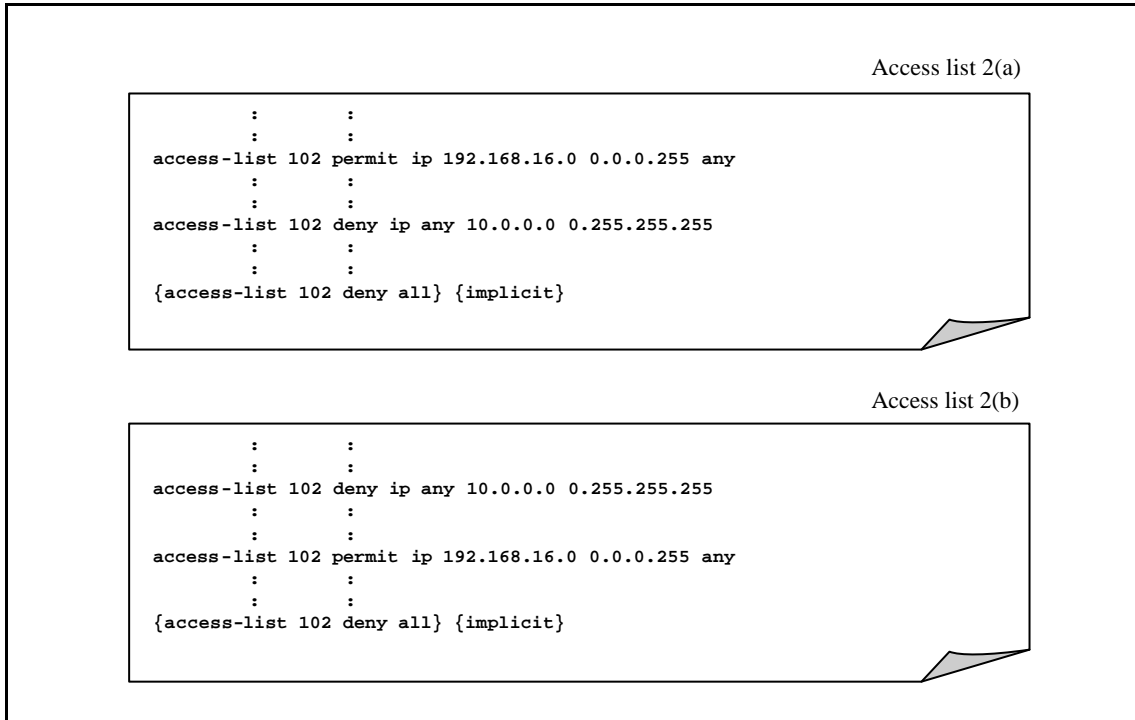


Figure 2. The importance of dependent rule order

2. DEFINITIONS AND NOTATION

Where appropriate in this paper, abbreviations are used as follows: \exists , ‘there is’ or ‘there exists’; \forall , ‘for all’ or ‘for every’; \wedge , ‘and’; $\hat{=}$, ‘if and only if’; and \textcircled{R} , ‘such that’.

Define A^* to be the set of all *addresses* available within a given system, define B^* to be the set of all *protocols* recognised by the system and define $\underline{F}^* = \{0, 1\}^w$ to be the set of w *flag vectors* ($\{0, 1\}$ w -tuples acting on B^*) valid for the system. For completeness, X^* represents the set of payloads.

2.1. Packets

A *packet*, $p_k = (s_k, d_k, b_k, f_k, X_k)$, is defined by its constituents: $s_k \hat{=} A^*$, the *source* address; $d_k \hat{=} A^*$, the *destination* address; $b_k \hat{=} B^*$, the *protocol*; $f_k \hat{=} \underline{F}^*$, the *flags* vector and $X_k \hat{=} X^*$, the *payload*.

A *traffic flow*, $T = [p_1, p_2, \dots, p_q]$, is a sequence of q packets. For sufficiently large q , this may be regarded as a distribution of packets and we simply refer to the *traffic*, T .

2.2. Rules

A rule, $r_i = (t_i, SA_i, DA_i, B_i, \mathbf{s}_i)$, consists of: a type, $t_i \in \{\text{permit}, \text{deny}\}$, $SA_i \in A^*$: the source range, $DA_i \in A^*$: the destination range, $B_i \in B^*$: the protocol range, and a flags predicate, $\mathbf{s}_i: \underline{F}^* \mapsto \{\text{true}, \text{false}\}$. Only t_i is a required component in all syntaxes. If any other components are absent then $SA_i = A^*$, $DA_i = A^*$, $B_i = B^*$ or $\mathbf{s}_i = \text{true}$ by default.

A packet, p_k , matches a rule, r_i (for which we write $p_k \tilde{N} r_i$), if its addresses and protocols are within the range of the rule and if its flags vector satisfies the rule's flags predicate. That is,

$$p_k \tilde{N} r_i \hat{U} (SA_k \hat{I} SA_i) \hat{U} (DA_k \hat{I} DA_i) \hat{U} (B_k \hat{I} B_i) \hat{U} \mathbf{s}_i(\underline{f}_k), \quad (1)$$

in which case the packet will be permitted or denied according to t_i .

2.3. Policies and Dependencies

A policy, $Z = [r_1, r_2, \dots, r_n]$ is an (ordered) sequence of n rules to achieve some purpose. It is assumed here that the rules of a policy are correctly ordered to achieve this purpose. Also, the last rule denies *all* traffic; that is, $t_n = \text{deny}$, $SA_n = A^*$, $DA_n = A^*$, $B_n = B^*$ and $\mathbf{s}_n = \text{true}$.

A dependency exists between two rules, r_i and r_j , if they are of opposite type and it is possible that there exists a packet, p_k , that matches both rules ($(p_k \tilde{N} r_i) \hat{U} (p_k \tilde{N} r_j)$); that is r_i and r_j are *dependent* if

$$\exists p_k \text{ such that } (t_i \neq t_j) \hat{U} (SA_k \hat{I} SA_i \hat{C} SA_j) \hat{U} (DA_k \hat{I} DA_i \hat{C} DA_j) \hat{U} (B_k \hat{I} B_i \hat{C} B_j) \hat{U} \mathbf{s}_i(\underline{f}_k) \hat{U} \mathbf{s}_j(\underline{f}_k). \quad (2)$$

Eliminating the packet, p_k , from this expression, allows a $\{0, 1\}$ dependency matrix, $D = (d_{ij}: 1 \leq i, j \leq n)$, to be defined:

$$d_{ij} \hat{U} (t_i \neq t_j) \hat{U} (SA_i \hat{C} SA_j \hat{I} \mathcal{A}) \hat{U} (DA_i \hat{C} DA_j \hat{I} \mathcal{A}) \hat{U} (B_i \hat{C} B_j \hat{I} \mathcal{A}) \hat{U} (\mathbf{s}_i \hat{C} \mathbf{s}_j \hat{I} \mathcal{A}), \quad (3)$$

where $\mathcal{S}_i \hat{I} \underline{F}^*$ is the subset of flag vectors satisfying \mathbf{s}_i .

2.4. Redundancies

A rule, r_j , in a policy, Z , is *redundant* (written $r_i \leftarrow r_j$) if there exists a rule, r_i ($i < j$), in Z , such that all packets matching r_j will be matched by r_i .

$$r_i \leftarrow r_j \Leftrightarrow (t_i = t_j) \hat{U} (SA_i \hat{E} SA_j) \hat{U} (DA_i \hat{E} DA_j) \hat{U} (B_i \hat{E} B_j) \hat{U} (\mathbf{s}_i \hat{E} \mathbf{s}_j). \quad (4)$$

A redundant rule may be removed from the policy without changing its purpose.

A rule, r_i , in a policy, Z , is *potentially redundant* if there exists a rule, r_j ($i < j$), in Z , such that all packets matching r_i will be matched by r_j . A redundant rule may be

removed from the policy without changing its purpose provided that no other rules between r_i and r_j are dependent upon r_j ; that is,

$$r_i \rightsquigarrow r_j \Leftrightarrow (t_i = t_j) \dot{U}(SA_i \dot{I} SA_j) \dot{U}(DA_i \dot{I} DA_j) \dot{U}(B_i \dot{I} B_j) \dot{U}(S_i \dot{I} S_j) \dot{U}'' \quad v (i < v < j), d_{vj} = 0. \quad (5)$$

Both forms of redundancy include the case, $r_i = r_j$.

Finally, and in brief, rules, r_a, r_b, \dots, r_w , are said to be *co-redundant* if there can be found a rule, r_i ($i < a, b, \dots, w$), such that r_i can replace r_a, r_b, \dots, r_w . Equivalent definitions may be derived for co-redundancy with respect to destination address and protocol, and for *potential co-redundancy*.

A detailed treatment of the detection of redundancies is given in Shih and Qian (2003). It is assumed, from this point, that the policy, $Z = [r_1, r_2, \dots, r_n]$, contains no redundancies.

2.5. Lists and Hit Rates

An *access list*, or simply *list*, L , implements a policy, $Z = [r_1, r_2, \dots, r_n]$, if it is a permutation of the rules of Z such that the order of dependencies is preserved. Let $r_i(L)$ be the rule at position i in L . A special case of a list implementing a policy, Z , is the *identity list*, $I_Z = [r_1, r_2, \dots, r_n]$, for which $r_i(I_Z) = r_i$ $\forall i (1 \leq i \leq n)$.

The *hit-rate*, $h(r_i(L), T)$, of rule r_i in a list L , is the probability that a packet from a traffic flow T will match r_i in L . It is initially assumed that static hit-rates are calculated dynamically using counters within the IOS or hardware (Cisco, 2002). However, this is discussed further in Section 7.

2.6. Latencies

The *latency*, $I(r_i)$, of a rule r_i is the time taken to (independently) process r_i . This may be calculated from the length of a rule, the nature of the protocols involved or taken from stored tables. In some systems, latencies may be constant for all rules but this is not assumed in this paper.

The *cumulative latency*, $k(r_i(L))$, of r_i in a list L , is the time taken to process r_i and all rules preceding it in L .

$$k(r_i(L)) = \sum_{j=1}^i I(r_j(L)). \quad (6)$$

The *expected latency*, $E(L, T)$, of a list L , in traffic T , is then given by

$$E(L, T) = \sum_{i=1}^n h(r_i(L), T) k(r_i(L)) = \sum_{i=1}^n h(r_i(L), T) \sum_{j=1}^i I(r_j(L)). \quad (7)$$

For a given traffic flow, T , we require to find (or approximate) the list, L , implementing a policy, Z , that minimises $E(L, T)$.

3. THE PROBLEM AND ITS COMPLEXITY

The problem, SEQUENCING TO MINIMISE EXPECTED LATENCY (SMEL), can be expressed, in standard terms (Garey and Johnson, 1979) as

INSTANCE: Traffic flow T , Policy Z of n rules, partial order on N given by dependency matrix D , for each rule $r \in Z$ a latency $l(r)$ and a hit-rate $h(r)$, and a target K .

QUESTION: Is there an ordering L of the rules of Z , obeying the dependency constraints D , such that the expected latency $E(L, T)$ as defined in (7) is K or less?

Initially, we assume the traffic flow T to have a constant packet distribution. The size of the solution space for (the unconstrained) SMEL is $(n-1)!$, taking the last deny all rule to be fixed. This is identical to that for the TRAVELING SALESMAN PROBLEM (TSP), the classic NP-complete combinatorial optimisation (CO) problem. The unconstrained problem (i.e. with no dependencies) has TSP complexity. The dependencies serve to reduce the size of the solution space by making certain orderings invalid but have no effect on the complexity as shown here.

THEOREM: SEQUENCING TO MINIMISE EXPECTED LATENCY (SMEL) is NP-complete

PROOF: Transformation to SEQUENCING TO MINIMIZE WEIGHTED COMPLETION TIME (SMWCT) (Lawler, 1978).

A direct mapping from SMEL to SMWCT is achieved by setting

<u>SMEL</u>		<u>SMWCT</u>
Z	to	N
r	to	t
D	to	? by taking $t_i < t_j \iff d_{ij} = 1$
$l(r)$	to	$l(t)$
$h(r)$	to	$w(t)$

(using the notation from Garey and Johnson, 1979) for any given flow, T .

It follows that (unless $P=NP$) guaranteed exact solutions are not reasonably to be expected for large values of n .

4. EXACT ALGORITHMS

A typical ACL may have less than 20 rules. Only in exceptional cases are instances of 100 or 1,000 rule ACLs to be found (Bukhatwa and Patel, 2003). There is some value then in considering exact approaches to optimising rule order (if only to have a benchmark against which to compare approximated solutions). Four standard methods are discussed briefly here. There is a close relationship between the order of the n rules in an ACL and the n arcs of the TSP, with the dependencies of the ACL denoting infeasible arcs of the TSP. Consequently, TSP notation and terminology may be used interchangeably with SMEL where appropriate.

4.1. Exhaustive Search

The simplest, but least efficient, approach to exact ACL optimisation will be to generate, by iteration or recursion, each ordering, L , of the rules in turn, test for validity against dependency constraints, D , and record the solution that minimises $E(L, T)$. The time complexity of such a process will be $O(n!)$ but with space complexity of $O(n)$. Although minimising space complexity may be of some value in environments with limited (storage) capacity, this time complexity is unacceptable in most practical circumstances.

4.2. Dynamic Programming

A more efficient dynamic programming technique is given by Held and Karp (1962) and adapted in various forms to the present day (Lawler et al., 1985 and Gutin and Punnen, 2002). The generic algorithm has time complexity $O(2^n)$, space complexity $O(2^n)$ and can be adapted for SMEL as follows.

$$Z = \{r_1, r_2, \dots, r_n\} \quad (\text{with } r_n \text{ fixed})$$

For $Y = \{r_1, r_2, \dots, r_{n-1}\}$ and $r \in Y$, let $|SMEL|(Y, r)$ be the minimum expected latency of the sublist $Y \setminus \{r\}$. Then

$$|SMEL|(Y, r) = \begin{cases} \mathbf{k}(r(Y)) & Y = \{r\} \\ \min_{s \in Y} |SMEL|(Y - \{r\}, s) + \mathbf{k}(s(Y)) & Y \neq \{r\} \end{cases} \quad (8)$$

$SMEL$ can then be calculated as $\min_r |SMEL|(Z, r) + \mathbf{k}(r(Z))$.

Although an improvement on exhaustive search, the time complexity is still exponential. The exponential space complexity may be a significant problem in restricted environments and, in practice, often translates to increased time complexity on implementation. However this method, on more powerful processors, provides good benchmarks for comparison with heuristics.

4.3. Linear Programming

Linear Programming (LP) techniques are well-established in solving large CO problems (Papadimitriou, 1994). The formulation of SMEL as an LP problem from an objective function (7) subject to the constraints of dependencies, D , is non-trivial

but achievable. On a stand-alone processor, this provides faster solutions than from Section 4.1 and 4.2. However, the implementation of LP solution software within the very tight constraints of router IOS and capacity, or in hardware, is unrealistic. For comparison purposes, such methods are only appropriate for small numbers of tests since each new instance has to be programmed into the system before solving. This is impractical for large repetitive test runs.

4.4. Branch and Bound

The most efficient known exact (or near-exact) solutions to large CO problems are the various branch-and-bound or branch-and-cut algorithms developed in relation to LP methods. Possibly the most efficient of these is the algorithm of Applegate et al. (2003). With these techniques, processing in parallel and often with human intervention, it is possible to derive exact (or near-exact) solutions to extremely large problems (Applegate et al., 2004). Such methods are clearly not suitable, nor required, for the moderately-sized SMEL problems in actual implementation. They are, however, useful for small numbers of larger comparisons.

5. APPROXIMATIONS AND HEURISTICS

Even for relatively small problems, heuristics will be necessary for implementation in real-time in operational networks. A typical distribution router may have a processor (clock) speed of about 80KHz and less than 50MB of dynamic memory. Nothing should be permitted to add to the inherent latency of the packet matching and routing process so any optimisation of ACL structure, implemented in IOS or hardware, must be both time- and space-efficient. Fortunately there are simple heuristics for the TSP, simultaneously fast and compact, which extend well to SMEL.

5.1. k-OPT

The simplest, and most easily implemented, heuristic algorithms for large CO problems are the *local search* methods known collectively as *k-OPT* (Rego and Glover, 2002). For the TSP, starting from some initial solution, arcs are swapped ($k=2$) or permuted ($k>2$) in a search to find superior solutions. For SMEL, these swaps/permutations correspond to k -wise reorderings of the rules of the list, L . An example of *2-OPT* applied to SMEL is given in Figure 3.

The initial solution, for a policy Z is the identity list, I_Z . $L_{\langle ij \rangle}$ is the list, derived from L , with rules i and j swapped. The algorithm works by applying a sequence of 2-swaps to the current list, L , and implementing the best while an improvement exists. The procedure $\text{swap}(r, s)$ reverses the places of r and s in L . The space complexity of this algorithm is $O(n)$. Its time complexity is $YO(n^2)$ where Y is the number of passes through the indefinite loop. The *2-OPT* algorithm is easily extended to the *3-OPT* of Figure 4, in which $L_{\langle ijk \rangle}$ and the procedure $\text{permute}(r, s, t)$ have the natural interpretation. *3-OPT* has space complexity $O(n)$ and time complexity $YO(n^3)$. If the algorithms are truncated in time to suit their environment (processor speed) by $Y \ll K$, where K is constant, then both time- and space-complexity are polynomial and the algorithms can be constrained to run within the tight restrictions of a router IOS or

even in hardware. The nature of the algorithms also aids easy implementation: swaps and permutations make for simple IOS code and/or logic design in hardware

```

L := IZ;
repeat
  Dmax := 0;
  for i := 1 to n-2 do
    for j := i+1 to n-1 do
      if dij = 0 then
        begin
          D := E(L,T) - E(L<ijmax then
            begin
              Dmax := D;
              i* := i;
              j* := j;
            end
          end;
        end;
      if Dmax > 0 then
        swap(ri*(L), rj*(L))
    until
      Dmax = 0

```

Figure 3. SMEL 2-OPT

```

L := IZ;
repeat
  Dmax := 0;
  for i := 1 to n-3 do
    for j := i+1 to n-2 do
      for k := j+1 to n-1 do
        if (dij = 0) and (djk = 0) and
           (dik = 0) then
          begin
            D := E(L,T) - E(L<ijkmax then
              begin
                Dmax := D;
                i* := i;
                j* := j;
                k* := k;
              end
            end;
          end;
        if Dmax > 0 then
          permute(ri*(L), rj*(L), rk*(L))
    until
      Dmax = 0

```

Figure 4. SMEL 3-OPT

5.2. Lin and Kernighan

The *Lin-Kernighan (LK)* approach to local search optimisation represents a family of heuristics concerned with varying the k of k -OPT. There have been a number of variations since the original algorithm (Lin and Kernighan, 1973) but all have the same essential premise: to extend the scope and resolution of a fixed search. Appropriate LK algorithms are known to generally produce the best results of all local search methods (Johnson and McGeoch, 2002 and Johnson et al., 2002).

```

L := IZi
repeat
  single_smel_2-opt           {2-OPT}
until
  Dmax = 0

L := IZi
repeat
  single_smel_3-opt           {3-OPT}
until
  Dmax = 0

```

Figure 5. SMEL 2-OPT and 3-OPT using procedures `single_smel_2-opt` and `single_smel_3-opt`

Let `single_smel_2-opt` and `single_smel_3-opt` be procedures that implement single iterations of the SMEL 2-OPT and 3-OPT processes (so that the algorithms of Figures 3 and 4 can be rewritten as in Figure 5, for example). Then the simplest, and fastest, version of an LK algorithm for SMEL will be the (2,3)LK-OPT algorithm as shown in Figure 6. This is the LK variant used in the computational results to follow.

```

L := IZi
repeat
  repeat
    single_smel_2-opt
  until
    Dmax = 0;
  single_smel_3-opt
until
  Dmax = 0

```

Figure 6. SMEL LK-OPT

6. COMPUTATIONAL RESULTS

For a given value of n , let m be the number of dependencies. That is

$$m = \sum_{i=1}^{n-1} \sum_{j=i+1}^n d_{ij} . \quad (8)$$

Results have been obtained, in the first instance, through simulation. Figures 7 and 8, for example, show a 25 rule/12 dependency ($n=25/m=12$) case before and after 2-*OPT* optimisation. (Access list numbers are omitted for brevity.) Test instances were generated as follows.

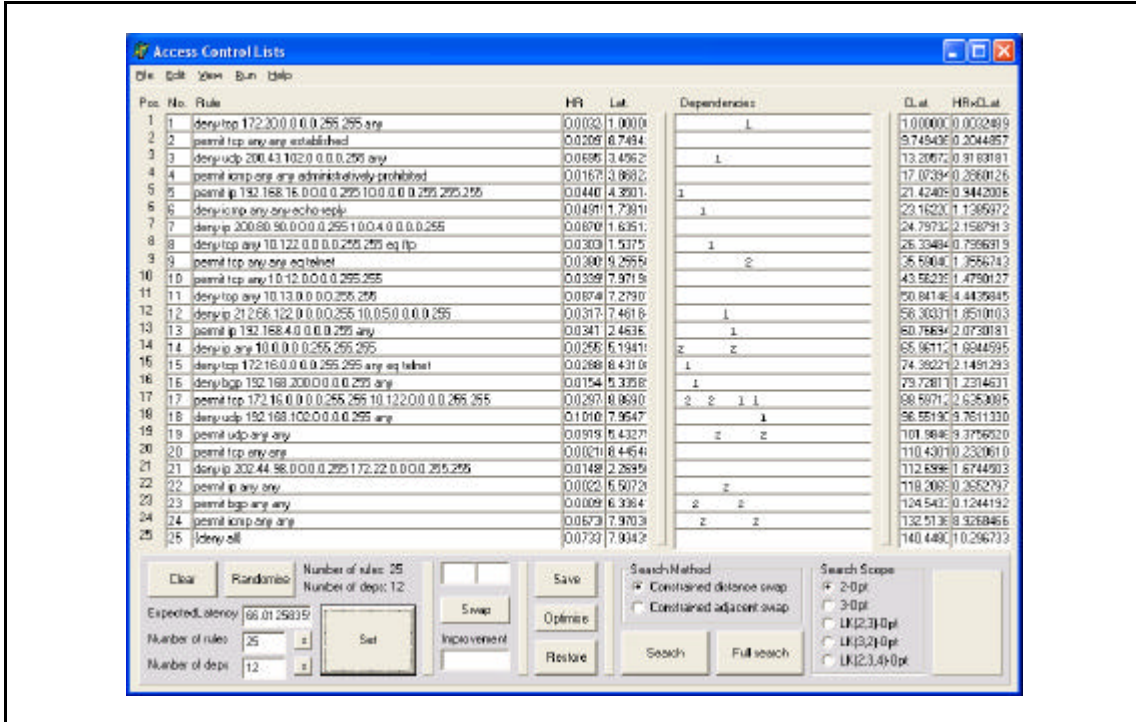


Figure 7. Simulated traffic policy

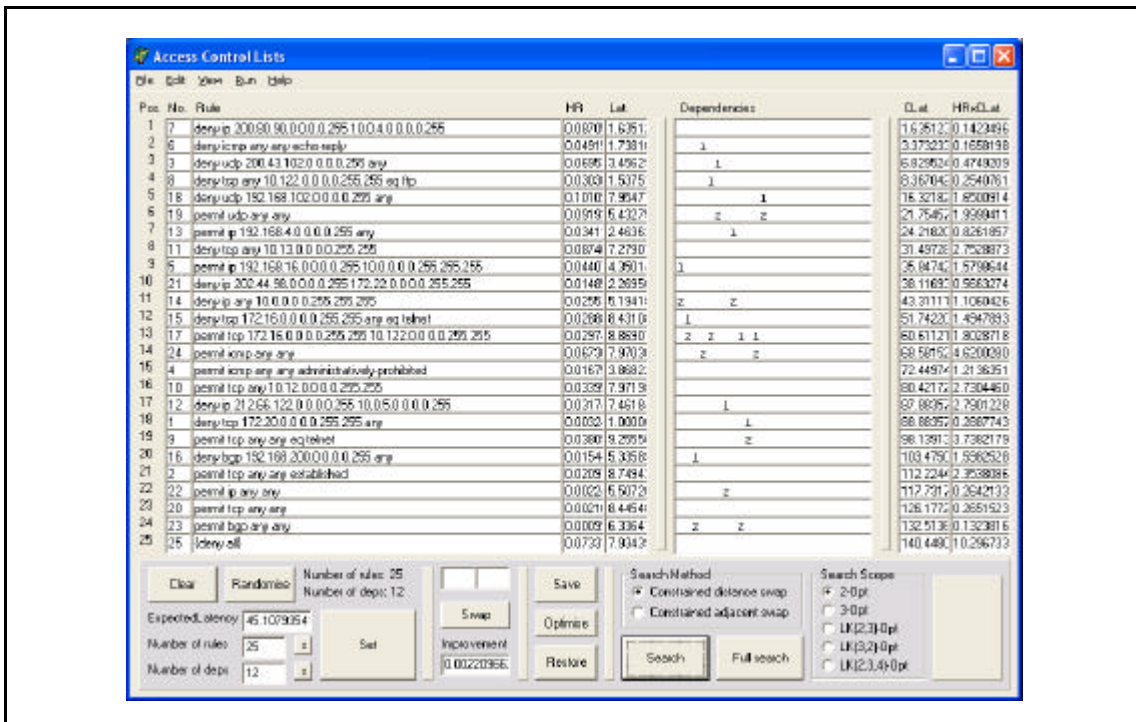


Figure 8. 2-*OPT* optimised ACL

$n=10/m=0,5,10,15,20,25$ - 100 instances of each and $n=25/m=0,10,20,30,40,50$ - 50 instances of each, solved to optimality by dynamic programming - Held-Karp variant (Section 4.2). $n=50/m=0,20,40,60,80,100$: 10 instances of each, solved to optimality by conventional LP (Section 4.3). $n=100/m=0,50,100,150,200,250$: 5 instances of each, solved to optimality by adaptive branch-and-bound methods (Section 4.4).

Table 1. Experimental comparisons

n	m	$c_{Z@O}$	$c_{O@2}$	\checkmark_2	S_2	$c_{O@LK}$	\checkmark_{LK}	S_{LK}
10	0	29.84	0.00	100	6.60	0.00	100	20.56
	5	21.11	2.36	36	5.84	2.11	45	18.02
	10	14.28	0.87	56	4.92	0.80	61	16.23
	15	12.13	3.40	44	4.16	2.93	54	13.09
	20	8.22	0.67	72	3.20	0.59	78	9.97
	25	6.87	0.79	68	2.68	0.69	76	8.41
25	0	36.02	0.00	100	31.50	0.00	100	76.48
	10	25.66	5.22	24	28.00	3.46	44	67.24
	20	17.68	4.86	8	23.24	3.80	14	59.92
	30	15.88	3.34	0	21.28	3.10	2	50.56
	40	10.04	3.78	0	19.90	3.24	0	59.22
	50	8.80	2.04	0	18.48	1.78	0	46.38
50	0	43.80	0.00	100	204.30	0.00	100	432.50
	20	30.00	7.80	0	187.90	6.90	20	346.20
	40	20.20	6.70	10	170.20	5.50	20	335.10
	60	18.60	5.90	0	164.30	4.80	0	326.60
	80	13.70	4.20	0	150.30	3.60	0	305.60
	100	12.30	3.40	0	136.50	3.00	0	283.70
100	0	50.80	0.00	100	1389.40	0.00	100	2095.60
	50	35.80	8.60	20	1320.40	5.60	20	1976.40
	100	23.20	8.40	0	1193.00	5.20	0	1739.20
	150	20.80	6.80	0	1150.20	4.80	0	1603.60
	200	15.60	7.80	0	1003.80	4.80	0	1447.40
	250	14.40	5.60	0	988.60	4.40	0	1404.20

n : number of rules. m : number of dependencies.

$c_{Z@O}$: mean (%) saving of optimal solution over original policy.

$c_{O@2}$: mean (%) increase of 2-*OPT* solution over optimum.

\checkmark_2 : mean (%) optimum found by 2-*OPT*. S_2 : mean number of iterations for 2-*OPT* to converge.

$c_{O@LK}$: mean (%) increase of *LK-OPT* solution over optimum.

\checkmark_{LK} : mean (%) optimum found by *LK-OPT*. S_{LK} : mean number of iterations for *LK-OPT* to converge.

A summary of results is given in Table 1. For each instance, 2-*OPT* and *LK-OPT* solutions were compared with the optimum obtained as described above. $c_{Z@O}$ gives the mean improvement (%) of the optimum EL over the EL of the original policy; $c_{O@2}$ gives the mean deterioration (%) of the 2-*OPT* solution from the optimum and $c_{O@LK}$ the equivalent figure for *LK-OPT*. \checkmark_2 , \checkmark_{LK} , S_2 and S_{LK} , give the percentage of

instances for which each method found the true optimum and the mean number of iterations (the number of passes through the central loop) for each method to converge. In each case, hit rates were generated randomly (uniform) and normalised so that

$$\sum_{i=1}^n h(r_i(Z), T) = 1 \quad (9)$$

and latencies generated randomly (uniform) in the interval $[1, 2]$. (1 to 2 **ms** are typical observed rule latencies across a range of routers.)

Although there is experimental variance in these figures, some patterns are clear. The heuristic *2-OPT* and *LK-OPT* methods are extremely accurate for smaller rule sets and, although they deviate more from optimality for larger sets, still give significant improvements over non-optimised policies. *LK-OPT*, as expected, gives generally better results than *2-OPT* but at a considerable expense in terms of run-time. Although the ratio of *LK-OPT* steps to *2-OPT* steps decreases as n increases, these larger ACLs are rare in practice (Bukhatwa and Patel, 2003). *2-OPT* gives excellent solutions for typical 10 or 20 rule ACLs and finds tolerable approximations quickly for larger exceptions.

7. EXTENSIONS AND VARIANTS

7.1. Variable Traffic Flows and Timing

Up to this point, the traffic/packet flow/distribution, T , has been taken as fixed. This is reasonable since, for each change in T , different hit-rates will apply and a new optimisation problem should be formulated. However, considering implementation, other factors arise. In reality, a router's only knowledge of traffic flow will come from logging packet types and this will probably take the form of incrementing counts and recalculating the hit-rates themselves. On this basis, the hit-rate, $h(r_i(L), T)$, of rule r_i in L under T changes constantly and two issues have to be addressed.

1. As the hit-rate of a rule is known only for its current position in the list and will be higher - the higher its position, how can the objective change, $E(L, T) - E(L_{\langle ij \rangle}, T)$, say, of a swap/permutation be calculated accurately?
2. How frequently should (re)optimisation be performed?

There is no practical method by which 'absolute' hit-rates can be calculated so there is no simple solution to the first question. Fortunately, the inequality is at least such that the process will be stable; that is, the hit-rate of a rule being considered for promotion up the list will always be under- rather than over-estimated so may not be swapped as far *up* the list as it should be but it will not be swapped *too* far. Constant re-swapping, then, will not occur unless the nature of the underlying traffic flow itself is oscillatory.

This suggests an answer to the second question. There is no practical value in (re)optimizing too frequently. Observed hit-rate probabilities will change with every packet processed, even if the packet distribution remains the same. There is no need

to recalculate expected latencies at this level, it being simpler to automatically promote (a fixed number of places or to the top of the list) the rule matched by the current packet. However, such an approach will be both unstable and resource-hungry.

A better solution will be to (re)optimize after a fixed period of time or number of packets or when the router is otherwise idle. Routing protocol packets between neighbouring routers, for example, are exchanged at intervals of between 5 and 120 seconds – depending on the protocol in use (Colton, 2002) – and an optimization period in this range may be intuitive. However, the formal optimization of this period/number, itself dependent upon the changing traffic flow, goes beyond the scope of this paper and is suggested as an avenue for future research.

7.2. Traffic Shaping, Queuing and Prioritisation

A final consideration comes from the application of packet prioritisation and other forms of traffic shaping. In a *weighted fair-queuing* (WFQ) system, for example, certain high-priority packets, such as voice or video, are processed ahead of low-priority traffic such as emails or file transfers. If ACLs are to be used to filter such traffic then it is essential that the rules identifying these packets are to be found, and remain, toward the top of the list (otherwise the delay in matching the packet against each ACL may increase the latency unacceptably).

In fact, the implementation of such *fixed rules* may be achieved through the existing system of dependencies. However, this is unlikely to be particularly efficient. On this basis, a fixed rule will have a dependency with *all* other rules in the policy. Testing such a large number of constraints through each *2-OPT* iteration will be complex and itself likely to increase latency. An alternative may be simply to implement a flag for each rule, identifying whether it can be moved. However, this in turn is not particularly flexible. If it is acceptable to move a rule only so far or to swap it with rules of particular types but not others, then the notion of dependencies is required once more. The ideal solution may be a compromise or hybrid method of marking the *freedom* of a rule, or a different method entirely? This problem also is suggested for future research.

8. CONCLUSIONS

This is a very real optimisation problem, albeit largely unaddressed until now. ACLs are used in many different ways in applying traffic policies on network routers and large, poorly-designed rule sets can add significantly to packet delay across internets. By comparison, any improvement that may be found through the valid reordering of these rules will be worthwhile, particularly applied across a sequence of routers. However, there is little value in the application of complex and time-consuming procedures in seeking optimum or improved orderings in such environments. Fast, simple heuristics, giving inexact but acceptable solutions, are to be preferred.

This paper has formulated and discussed the problem in its most general form and compared various exact and inexact methods of optimisation. The eventual

conclusion is that a simple, adapted and constrained, *2-OPT* process is preferable since

1. it has minimal space-complexity, making it ideal for implementation in router operating systems or even hardware,
2. it has small time-complexity, contributing as little as possible to processing delays in the router, and
3. although sub-optimal, it provides very good results in practice, at least sufficient to make optimisation worthwhile.

Some issues raised in the previous section (7) are left as open research questions.

REFERENCES

- Applegate, D., Bixby, R., Chvátal, V. and Cook, W. (2003) *CONCORDE TSP Solver*, Princetown University, <http://www.math.princeton.edu/tsp/concorde.html>.
- Applegate, D., Bixby, R., Chvátal, V. and Cook, W. (2004) *National Traveling Salesman Problems*, Princetown University, <http://www.math.princeton.edu/tsp/world/countries.html>.
- Bukhatwa, F. & Patel, A. (2003) Effects of Ordered Access Lists in Firewalls, *Proceedings of IADIS WWW/Internet 2003*, Algarve Portugal, 5th-8th November, pp257-264.
- Cisco (2002) *ACL Optimizer and Hits Optimizer*, Cisco Systems, www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/cw2000/fam_prod/acl_mgr/aclm_1_x/1_5/u_guide/ac1js.pdf
- Colton, A. (2002) *Cisco IOS for IP Routing*, Rocket Science Press Inc.
- Garey, M.R. and Johnson, D.S. (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York.
- Gutin, G. and Punnen, A.P. (2002) *The Traveling Salesman Problem and its Variations*, Kluwer Academic.
- Held, M. and Karp, R.M. (1962) A Dynamic Programming Approach to Sequencing Problems, *Journal of the Society of Industrial and Applied Mathematics (SIAM)*, Vol. 10, pp196-210.
- Johnson, D.S. and McGeoch, L.A. (2002) Experimental Analysis of Heuristics for the STSP, in *The Traveling Salesman Problem and its Variations* (eds. G. Gutin & A. Pullen), Kluwer Academic.
- Johnson, D.S., Gutin, G., McGeoch, L.A., Yeo, A., Zhang, W. and Zverovitch, A. (2002) Experimental Analysis of Heuristics for the ATSP, in *The Traveling Salesman Problem and its Variations* (eds. G. Gutin & A. Pullen), Kluwer Academic.
- Lawler, E.L. (1978) Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints, *Annals of Discrete Mathematics*, Vol. 2, pp75-90.
- E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. and Shmoys, D.B. (1985) *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimisation*, John Wiley & Sons.

Lin, S. and Kernighan, B.W. (1973) An Effective Heuristic Algorithm for the Traveling Salesman Problem, *Operations Research*, Vol. 21, pp972-989.

Papadimitriou, C.H. (1994) *Computational Complexity*, Addison Wesley Longman.

Rego, C. and Glover, F. (2002) Local Search and Metaheuristics, in *The 'Traveling Salesman Problem and its Variations'* (eds. G. Gutin & A. Punnen), Kluwer Academic.

Shih, C-S. and Qian, J. (2003) Security Policy Derivation, in *CS497: Cryptography and Computer Security*, University of Illinois at Urbana Champaign,
http://www-sal.cs.uiuc.edu/~steng/cs497_01/qian.ppt.

Stoica, I. (2001) Route Lookup and Packet Classification, CS 268, February 2001, Department of Electrical Engineering and Computer Science, University of California, Berkeley USA.