10-1-2006

Real-time optimisation of access control lists for efficient internet packet filtering

Vic Grout
Glyndwr University, v.grout@glyndwr.ac.uk

John Davies
Glyndwr University, j.n.davies@glyndwr.ac.uk

John McGinn
Glyndwr University, j.mcginn@glyndwr.ac.uk

Recommended Citation

Grout, V., McGinn, J., & Davies, J. (2007) 'Real-time optimisation of access control lists for efficient Internet packet filtering'. Journal of Heuristics,13(5), 435-454

Real-time optimisation of access control lists for efficient internet packet filtering

Abstract
This paper considers an optimisation problem encountered in the implementation of traffic policies on network routers, namely the ordering of rules in an access control list to minimise or reduce processing time and hence packet latency. The problem is formulated as an objective function with constraints and shown to be NP-complete by translation to a known problem. Exact and heuristic solution methods are introduced, discussed and compared and computational results given. The emphasis throughout is on practical implementation of the optimisation process, that is within the tight constraints of a production network router seeking to reduce latency, on-line, in real-time but without the overhead of significant extra computation.

Keywords
Access control lists, Internet packet filters, Real-time, optimisation, traffic policies

Disciplines
Computer Engineering

Comments

Journal of Heuristics, Vol. 13, No. 5, October 2006, pp435-454

REAL-TIME OPTIMISATION OF ACCESS CONTROL LISTS FOR EFFICIENT INTERNET PACKET FILTERING

Vic Grout, John McGinn and John Davies
Centre for Applied Internet Research (CAIR), University of Wales
NEWI Plas Coch Campus, Mold Road, Wrexham, LL11 2AW, UK
{v.grout|j.mcginn|j.n.davies}@newi.ac.uk

ABSTRACT

This paper considers an optimisation problem encountered in the implementation of traffic policies on network routers, namely the ordering of rules in an access control list to minimise or reduce processing time and hence packet latency. The problem is formulated as an objective function with constraints and shown to be NP-complete by translation to a known problem. Exact and heuristic solution methods are introduced, discussed and compared and computational results given. The emphasis throughout is on practical implementation of the optimisation process, that is within the tight constraints of a production network router seeking to reduce latency. on-line, in real-time but without the overhead of significant extra computation.

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, address translation, queuing or encryption. A packet may be matched against several ACLs on a single router and many on its complete journey from source to destination.

Inefficient ACLs then may add significantly to packet delay and even small ACLs will contribute to this latency simply by their aggregation across several routers.

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. The absence of any term, such as a protocol or flag, indicates the rule will match a packet with any such values – provided the specified fields are matched.

```
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)

Although the simple examples given in this section may appear to imply *classful* routing, the rules can use *wildcard masks* to match any required subnet so that the techniques discussed in this paper are fully suited to *Classless Inter-Domain Routing* (*CIDR*) applications. However, a discussion of such *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.

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.

Access list 2(a)

```
         :       :
         :       :
access-list 102 permit ip 192.168.16.0 0.0.0.255 any
         :       :
         :       :
access-list 102 deny ip any 10.0.0.0 0.255.255.255
         :       :
         :       :
{access-list 102 deny all} {implicit}
```

Access list 2(b)

```
         :       :
         :       :
access-list 102 deny ip any 10.0.0.0 0.255.255.255
         :       :
         :       :
access-list 102 permit ip 192.168.16.0 0.0.0.255 any
         :       :
         :       :
{access-list 102 deny all} {implicit}
```

Figure 2. The importance of dependent rule order

The short history of the study of ACL design is as follows. The issue of efficiency in packet filters was first addressed in this context by Stoica (2001) but largely as an aside and without significant outcome. Shih & Qian (2002) discuss the crucial question of how to identify rule dependencies in ACLs although the subject is first considered in any form in Hari et al. (2000), again as an aside. The first attempt at optimisation comes from Cisco (2003) but this work ignores individual rule latencies: that is, all rules are assumed to take the same time to process. Bukhatwa & Patel (2003) show the value of ACL optimisation but ignore both differences in rule latencies and, more crucially, rule dependencies. Bukhatwa (2004) gives a simplified method for ordering a list efficiently, based on the classification of rules by latency, but still fails to consider rule dependencies. In these approaches, rules are permitted to migrate freely within the list. Al-Shaer & Hamed (2004) give a much-improved treatment of the problem – with an awareness of rule dependency, but only for the purpose of discovering rule anomalies. All the above methods are *off-line*, that is, although rule hit-rates may be recorded automatically, any optimisation of rule order takes place as a separate, semi-manual process and the revised ACL loaded back on to the router. With the introduction of Turbo Access Lists (Cisco, 2004), the searching of ACLs is made more efficient. The list is pre-compiled into tables for which the packet header can then be used as a search key. Whilst this may be seen as the first semi-automatic implementation of ACL optimisation, it is actually a batch process - there is no attempt to change rule order in response to traffic flow. Also different rule latencies are still not considered.

This paper undertakes an entirely deeper study of the optimal ACL problem, suitable for implementation, on a larger scale, within the router IOS or embedded in hardware. We consider rule hit-rates, latencies and variable traffic flow in the optimisation of ACL order. It is proposed that the optimisation of rules within ACLs should take place in real-time (*on-line*) and automatically. Such processes must be efficient and worthwhile – reducing packet latency without adding significant computational overhead. They must also be practical and not conflict with the requirements and expectations of the *Network Administrator* (*NA*) configuring and maintaining the ACLs.

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.

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'; $\Leftrightarrow$, 'if and only if'; and $\rightarrow$, '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*, $p_k = (sa_k, Da_k, b_k, \underline{f}_k, X_k)$, is defined by its constituents: $sa_k \in A^*$, the *source* address; $Da_k \in A^*$, the *destination* address; $b_k \in B^*$, the protocol; $\underline{f}_k \in \underline{F}^*$, the flags vector and $X_k \in X^*$, the payload.

A *traffic flow*, $T = [p_1, p_2, ..., p_q]$, 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*, $r_i = (t_i, SA_i, DA_i, B_i, \sigma_i)$, consists of: a *type*, $t_i \in \{permit, deny\}$, $SA_i \subseteq A^*$: the source *range*, $DA_i \subseteq A^*$: the destination *range*, $B_i \subseteq B^*$: the protocol *range*, and a *flags predicate*, σ_i: $\underline{F}^* \mapsto \{true, false\}$. 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 $\sigma_i \equiv$ *true* by default.

A packet, p_k, *matches* a rule, r_i (for which we write $p_k \nabla r_i$), 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,

$$p_k \nabla r_i \Leftrightarrow (Sa_k \in SA_i) \wedge (Da_k \in DA_i) \wedge (b_k \in B_i) \wedge \sigma_i(\underline{f}_k), \tag{1}$$

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

2.3. Policies and Dependencies

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

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 ($(p_k \nabla r_i) \wedge (p_k \nabla r_j)$); that is r_i and r_j are *dependent* if

$$(t_i \neq t_j) \wedge \exists p_k \rightarrow (Sa_k \in SA_i \cap SA_j) \wedge (Da_k \in DA_i \cap DA_j) \wedge (b_k \in B_i \cap B_j) \wedge \sigma_i(\underline{f}_k) \wedge \sigma_j(\underline{f}_k). \tag{2}$$

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:

$$d_{ij} \Leftrightarrow (t_i \neq t_j) \wedge (SA_i \cap SA_j \neq \varnothing) \wedge (DA_i \cap DA_j \neq \varnothing) \wedge (B_i \cap B_j \neq \varnothing) \wedge (\Sigma_i \cap \Sigma_j \neq \varnothing), \tag{3}$$

where $\Sigma_i \subseteq \underline{F}^*$ is the subset of flag vectors satisfying σ_i.

If $d_{ij} = 1$ then the order of rules i and j must be preserved if the behaviour of the policy is to be maintained.

2.4. Redundancies

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

$$r_i \hookleftarrow r_j \Leftrightarrow (t_i = t_j) \wedge (SA_i \supseteq SA_j) \wedge (DA_i \supseteq DA_j) \wedge (B_i \supseteq B_j) \wedge (\Sigma_i \supseteq \Sigma_j). \tag{4}$$

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,

$$r_i \hookrightarrow r_j \Leftrightarrow (t_i = t_j) \wedge (SA_i \subseteq SA_j) \wedge (DA_i \subseteq DA_j) \wedge (B_i \subseteq B_j) \wedge (\Sigma_i \subseteq \Sigma_j) \wedge \forall v \rightarrow (i < v < j), d_{vj} = 0. \tag{5}$$

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

Finally, and in brief, rules, $r_\alpha, r_\beta, .., r_\omega$, are said to be *co-redundant* if there can be found a rule, r_i ($i < \alpha, \beta, .., \omega$), such that r_i can replace $r_\alpha, r_\beta, .., r_\omega$. Equivalent definitions may be derived for co-redundancy with respect to source/destination address and protocol/flags, and for *potential co-redundancy*.

A useful tutorial approach to the management of redundancies is given in Shih and Qian (2003). Al-Shaer & Hamed (2004) give an updated treatment. Although interesting, these concepts are not central to this work. The techniques discussed in this paper will work whether or not the policy, $Z = [r_1, r_2, ..., r_n]$, contains redundancies. Techniques for removal and detection of redundancies may be applied independently if required.

2.5. Lists and Hit Rates

An *access list*, or simply *list*, L, *implements* a policy, $Z = [r_1, r_2, ..., r_n]$, 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*, $I_Z = [r_1, r_2, ..., r_n]$, 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. Hit-rates can be calculated dynamically using counters within the IOS or hardware (Cisco, 2002, 2003).

2.6. Latencies

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

$$\kappa(r_i(L)) = \sum_{\varphi=1}^{i} \lambda(r_\varphi(L)) \,. \tag{6}$$

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

$$E(L,T) = \sum_{i=1}^{n} h(r_i(L),T)\kappa(r_i(L)) = \sum_{i=1}^{n} h(r_i(L),T) \sum_{\varphi=1}^{i} \lambda(r_i(L)) \,. \tag{7}$$

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 $\lambda(r)$ 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?

For the purposes of this section only, 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

SMEL		SMWCT	
Z	to	N	
r	to	t	
D	to	$\lessdot$	[by taking $t_i \lessdot t_j \Leftrightarrow (i < j) \wedge d_{ij} = 1$]
$\lambda(r)$	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

ACLs vary considerably in size. An ACL to select addresses for translation, for example, may have only two or three rules. A typical filter may have between *10* and *100* rules. Large enterprise and service providers may have ACLs with anything from several hundred rules to tens of thousands. However, smaller ACLs are more common so there is some value 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.

$Z = \{r_1, r_2, \ldots, r_n\}$ (with r_n fixed)

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

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

$SMEL$ can then be calculated as $\min_r |SMEL|(Z, r) + \kappa(r(Z))$.

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, may be a reasonable option for smaller lists and provides good benchmarks for comparison with heuristics for smaller values of n.

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 for on-line implementation in routers although 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 access router may have a processor (clock) speed of about 80KHz and less than 50MB of dynamic memory whereas large distribution or core routers use GHz processors and multi-GB memory. The *relative* significance of reducing packet latency, however, remains acute in all cases, as does the requirement that any attempt to optimise packet processing be worthwhile. 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 the IOS or hardware, must be both time- and space-efficient. Some of the more well-known and recent search techniques such as tabu search, simulated annealing and genetic algorithms (Aarts and Lenstra, 2003) produce very good results but are either too complex or difficult to implement within the strict constraints of the router IOS or hardware. Fortunately there are simple heuristics for the TSP and other problems, 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 re-orderings 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. $L_{<ij>}$ 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 `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 $\Psi O(n^2)$ where Ψ 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 $L_{<ijk>}$ and the procedure `permute(r,s,t)` have the natural interpretation. *3-OPT* has space complexity $O(n)$ and time complexity $\Psi O(n^3)$. If the algorithms are truncated in time to suit their environment (processor speed) by $\Psi \leq K$, 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 := I_Z;
repeat
   Δ_max := 0;
   for i := 1 to n-2 do
      for j := i+1 to n-1 do
         if d_ij = 0 then
            begin
               Δ := E(L,T) - E(L_<ij>,T);
               if Δ > Δ_max then
                  begin
                     Δ_max := Δ;
                     i* := i;
                     j* := j
                  end
            end;
   if Δ_max > 0 then
      swap(r_i*(L), r_j*(L))
until
   Δ_max = 0
```

Figure 3. SMEL *2-OPT*

```
L := I_Z;
repeat
   Δ_max := 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 (d_ij = 0) and (d_jk = 0) and
               (d_ik = 0) then
               begin
                  Δ := E(L,T) - E(L_<ijk>,T);
                  if Δ > Δ_max then
                     begin
                        Δ_max := Δ;
                        i* := i;
                        j* := j;
                        k* := k
                     end
               end;
   if Δ_max > 0 then
      permute(r_i*(L), r_j*(L), r_k*(L))
until
   Δ_max = 0
```

Figure 4. SMEL *3-OPT*

5.2. Lin and Kernighan

The *Lin-Kernigham* (*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 := I_Z;
repeat
   single_smel_2-opt            {2-OPT}
until
   Δmax = 0

L := I_Z;
repeat
   single_smel_3-opt            {3-OPT}
until
   Δmax = 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. As with most, local search processes, it has space complexity, *O(n)*. It's time complexity, however, is less predictable. Empirical results are given in Section 6.

```
L := I_Z;
repeat
   repeat
      single_smel_2-opt
   until
      Δmax = 0;
   single_smel_3-opt
until
   Δmax = 0
```

Figure 6. SMEL LK-OPT

5.3. Constrained Sort

A final heuristic considered for SMEL is a form of constrained sort process. It may be seen as a restricted version of *2-OPT* in which only adjacent rules are considered for swapping. Searching from the top of the ACL, each rule is compared with the one following it to see if swapping them would improve the expected latency of the list. The process continues through the list and repeats until there are no further improvements to be found. This *C-SORT* approach is detailed in Figure 7.

```
L := I_Z;
repeat
   Δ := 0;
   for i := 1 to n-2 do
        if d_i i+1 = 0 then
             if E(L,T) - E(L<i i+1>,T) > 0 then
                  begin
                     Δ := 1;
                     swap(r_i(L), r_i+1(L))
                  end
until
   Δ = 0
```

Figure 7. SMEL *C-SORT*

An essential difference between *C-SORT* and *2-OPT* is that *all* swaps giving an improvement in expected latency are implemented immediately. (*Δ* is only maintained to flag when no further reduction is possible). *C_SORT* is considerably quicker than *2-OPT* (*ΨO(n)*) at the expense of being more inherently greedy, and hence (potentially) less accurate. The next section discusses results.

6. COMPUTATIONAL RESULTS

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

$$m = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} d_{ij} \text{ ,} \qquad (8)$$

not including rule n, which is dependent with all other rules.

Results have been obtained through simulation in two ways. Firstly, a number of moderately sized ($n \leq 100$) test instances were generated randomly and the *2-OPT* and *LK-OPT* processes compared with the optimal solution as described below. Without explicitly taking traffic into account, the only pertinent parts of a rule are its hit-rate and latency and these can be generated through stand-alone simulation. Figures 8 and 9, 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.) Secondly, a number of larger test instances ($500 \leq n \leq 10{,}000$) were produced to compare *2-OPT* with *C-SORT*.

These random test instances were generated as follows.

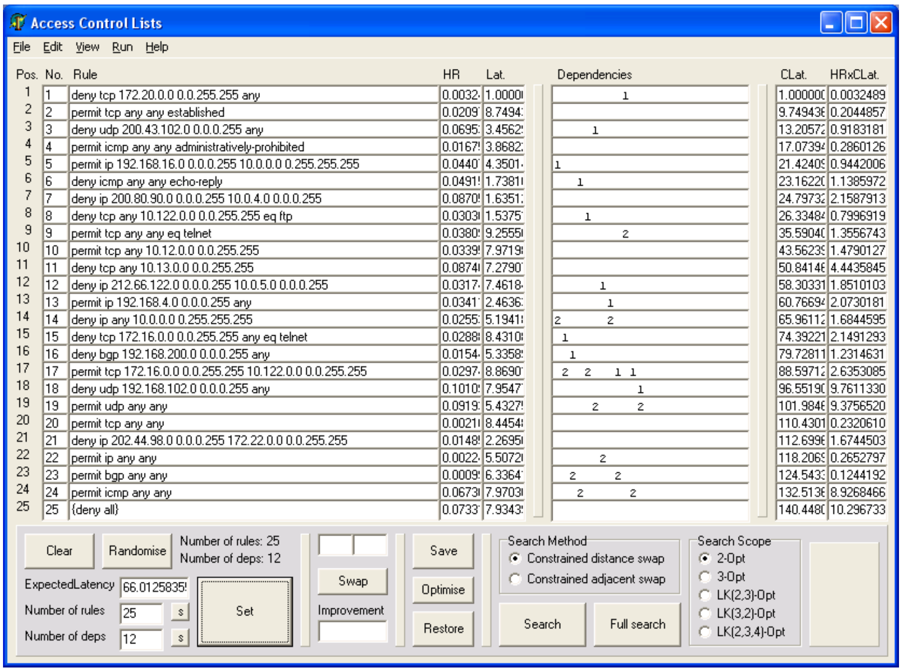

Figure 8. Simulated traffic policy

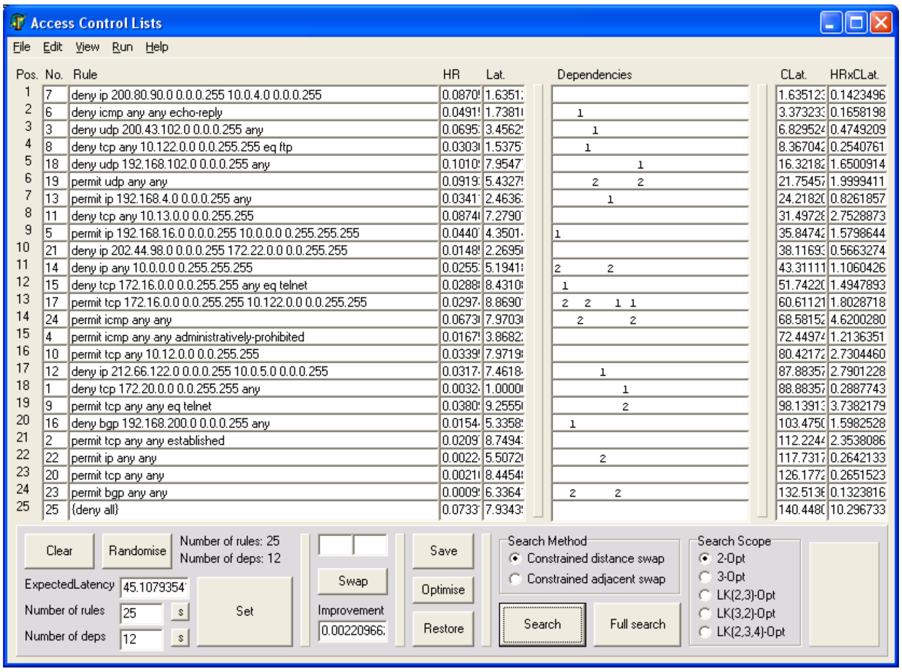

Figure 9. *2-OPT* optimised ACL

- *n=10/m=0,10,20*: *100* instances of each and *n=25/m=0,20,40*: *50* instances of each, solved to optimality by dynamic programming - Held-Karp variant (Section 4.2) – and compared with *2-OPT* and *LK-OPT*.

- *n=50/m=0,40,80*: *10* instances of each, solved to optimality by conventional LP (Section 4.3) – and compared with *2-OPT* and *LK-OPT*.
- *n=100/m=0,100,200*: *5* instances of each, solved to optimality by adaptive branch-and-bound methods (Section 4.4) – and compared with *2-OPT* and *LK-OPT*.
- *n=500/m=500,1000,5000,10000*. *100* instances of each comparing *2-OPT* with *C-SORT*.
- *n=1000/m=1000,5000,10000,50000*. *100* instances of each comparing *2-OPT* with *C-SORT*.
- *n=5000/m=5000,10000,50000,100000*. *100* instances of each comparing *2-OPT* with *C-SORT*.
- *n=10000/m=10000,50000,100000,500000*. *100* instances of each comparing *2-OPT* with *C-SORT*.

Table 1. Comparing *2-OPT* and *LK-OPT* with optimal solution.

n	m	$c_{Z\to O}$	$c_{O\to 2}$	✓$_2$	S_2	$c_{O\to LK}$	✓$_{LK}$	S_{LK}
10	0	29.84	0.00	100	6.60	0.00	100	20.56
	10	14.28	0.87	56	4.92	0.80	61	16.23
	20	8.22	0.67	72	3.20	0.59	78	9.97
25	0	36.02	0.00	100	31.50	0.00	100	76.48
	20	17.68	4.86	8	23.24	3.80	14	59.92
	40	10.04	3.78	0	19.90	3.24	0	59.22
50	0	43.80	0.00	100	204.30	0.00	100	432.50
	40	20.20	6.70	10	170.20	5.50	20	335.10
	80	13.70	4.20	0	150.30	3.60	0	305.60
100	0	50.80	0.00	100	1389.40	0.00	100	2095.60
	100	23.20	8.40	0	1193.00	5.20	0	1739.20
	200	15.60	7.80	0	1003.80	4.80	0	1647.40

n: number of rules. m: number of dependencies.
$c_{Z\to O}$: mean (%) saving of optimal solution over original policy.
$c_{O\to 2}$: mean (%) increase of *2-OPT* solution over optimum.
✓$_2$: mean (%) optimum found by *2-OPT*. S_2: mean number of iterations for *2-OPT* to converge.
$c_{O\to LK}$: mean (%) increase of *LK-OPT* solution over optimum.
✓$_{LK}$: mean (%) optimum found by *LK-OPT*. S_{LK}: mean number of iterations for *LK-OPT* to converge.

In each case, hit rates were generated randomly (uniformly) and normalised so that

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

and latencies generated randomly (uniformly) in the intervals *[1,2]* for $10 \le n \le 100$ and *[0.1,0.3]* for $500 \le n \le 10000$. *1* to $2\mu s$ are typical observed rule latencies across a range of (slower) access routers of the kind likely to be implementing smaller ACLs with reduced times for more powerful processors (Davies and Grout, 2005). In each case, the required number of dependent rule pairs (m) is also generated/assigned randomly (uniformly).

The first 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. $c_{Z \to O}$ gives the mean improvement (%) of the optimum EL over the EL of the original policy; $c_{O \to 2}$ gives the mean deterioration (%) of the *2-OPT* solution from the optimum and $c_{O \to LK}$ 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. Timings in seconds are not given as this will depend entirely on the processor within the router – see the next section for a full discussion.

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, *2-OPT* gives excellent solutions for *10* or *20* rule ACLs and finds tolerable approximations reasonably quickly for ACLs of *50* or *100* rules. It appears that the computationally intensive *LK-OPT* is unnecessary for these values.

Table 2. Comparing *2-OPT* with *C-SORT*

n	m	$c_{2 \to CS}$	S_2	S_{CS}
100	100	6.44	2,525	764
	500	6.05	2,203	675
	1,000	5.91	1,996	599
	5,000	5.83	1,789	552
500	500	5.29	44,932	2,225
	1,000	5.09	39,102	1,803
	5,000	5.02	35,079	1,654
	10,000	4.95	29,441	1,559
1,000	1,000	4.30	344,183	3,145
	5,000	4.22	308.949	2,672
	10,000	4.17	270,056	2,209
	50,000	4.11	224,721	1,992
5,000	5,000	3.61	12,101,000*	9,032
	10,000	3.56	10,884,000*	7.962
	50,000	3.52	10,069,000*	7,110
	100,000	3.48	9,170,000*	6,282
10,000	10,000	3.11	128,336,000*	17,219
	50,000	3.07	115,508,000*	14,843
	100,000	3.02	109,910,000*	12,055
	500,000	2.98	93,991,000*	10,276

n: number of rules. m: number of dependencies.
$c_{2 \to CS}$: mean (%) increase of *C-SORT* solution over *2-OPT* solution .
S_2: mean number of iterations for *2-OPT* to converge. (* rounded)
S_{CS}: mean number of iterations for *C-SORT* to converge.

However the *2-OPT* heuristic, with its $\Psi O(n^2)$ complexity will not be suitable for large n within the tight constraints of a production router. (In fact, as n increases the gap in convergence steps between *2-OPT* and *LK-OPT* is decreasing.) In such cases, *C-SORT* ($\Psi O(n)$) is the last resort. Table 2 compares *2-OPT* with *C-SORT*. The exact solutions are no longer available for larger problem instances but the deterioration of *C-SORT* compared to *2-OPT* is given instead. Using the notation from Table 1, in Table 2, $c_{2 \to CS}$ gives the mean percentage increase in expected latency from the *C-SORT* method over *2-OPT*, S_2 and S_{CS} give the mean number of iterations (the number of passes through the central loop) for each method to converge, the larger values for S_2 being rounded to the nearest thousand.

It can bee seen from Table 2 that *C-SORT* performs well. Although less accurate than *2-OPT*, the difference decreases as n increases. It's convergence time, however, is much better. *C-SORT* appears a better choice for larger ACLs. All the lessons from this section are discussed in the next.

7. PRACTICAL IMPLEMENTATION

In principle, we now have the necessary techniques to allow ACL rule order optimisation to be carried out on-line (in real time) on a router. This section discusses their practical implementation.

7.1. Choosing Processes

We have discussed, in varying depth, the following ACL optimisation algorithms:

- Exhaustive search (*ES*)
- Held and Karp dynamic programming (*H-K*)
- Linear programming (*LP*)
- Branch and bound (*BB*)
- *2-OPT*
- *3-OPT*
- *LK-OPT*
- *C-SORT*

Our discussions lead us to reject *ES* (too complex), *LP* & *BB* (difficult to automate) and *3-OPT* & *LK-OPT* (poor return for increased complexity) in favour of *H-K*, *2-OPT* and *C-SORT*.

ACLs vary considerably in size. Many are extremely small and, for these cases, *H-K* may be entirely viable. Some are larger (requiring *2-OPT*) and some are very large (where only *C-SORT* can be expected to run). Depending on the processing power of the router, it is proposed that two limit values be set. (This will be an inbuilt feature of the processor/IOS.)

max_{small}: the maximum number of rules in a 'small' ACL
max_{medium}: the maximum number of rules in a 'medium' ACL

Thus, 'small' and 'medium' are terms specific to each router with the ACL optimisation process being controlled at the top level as follows.

```
if n ≤ max_small then
      H-K
else if n ≤ max_medium then
      2-OPT
else
      C-SORT
```

7.2. Measuring Hit-Rates and Timing

Traffic characteristics may change over time and with them packet hit-rates. This is not a problem in itself but, 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, $E(L,T) – E(L_{<ij>},T)$, 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.

7.3. Traffic Shaping, Queuing and Prioritisation

Another 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, video or other multimedia traffic, 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, for example, 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.

7.4. The Network Administrator (NA)

There is a final practical consideration concerning the NA maintaining the ACL. ACLs often (in fact, usually) evolve over time. The NA will add new rules from time to time to add new policies, etc. They need to be fully aware of the current structure of the ACL. Whilst, they will want the ACL to be as efficient, as possible, they will not tolerate an actual re-ordering of the list with which they work on a day-to-day basis. As the purpose of this paper is to explicitly *re-order* the ACL for greater efficiency, how is this conflict to be resolved?

We suggest a simple system of pointers. The NA maintains the primary ACL. An array of pointers is used, however, to indicate the working order of the list. Optimisation and packet-matching itself takes place with regard to the indexed list. The pointer array will be of inconsequential size compared to the original ACL and the extra processing minimal

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, while exact optimisation may be a possibility for small ACLs, a simple, adapted and constrained, *2-OPT* process is preferable for medium ACLs (the limits depend on the router in question) since

1. it has minimal space-complexity, making it ideal for implementation in router operating systems or even hardware,
2. it has moderate time-complexity, contributing little to processing delays in the router, and
3. although sub-optimal, it provides very good results in practice.

However, for larger ACLs, the *C-SORT* process is to be preferred since

1. it has the same minimal space-complexity as *2-OPT*,
2. it has minimal time-complexity, contributing as little as possible to processing delays in the router,
3. it can be implemented more frequently than *2-OPT*, and
4. although less accurate than *2-OPT*, 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.

ACKNOWLEDGEMENTS

The comments and suggestions from the referees were invaluable in improving this paper and preparing it for publication and were gratefully received.

REFERENCES

Al-Shaer, E. & Hamed, H., (2004) Modeling and Management of Firewall Policies, *IEEE Transactions on Network and Service Management*, Vol. 1-1, April 2004.

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.

Aarts, E. and Lenstra, J.K. (2003) *Local Search in Combinatorial Optimisation*, Princetown University Press.

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.

Bukhatwa, F., (2004) High Cost Elimination Method for Best Class Permutation in Access Lists, *Proceedings of IADIS WWW/Internet International Conference (W3I 2003)*, Madrid, Spain, 6th-9th October 2004, pp287-294.

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

Cisco (2003) *ACL Manager*, Cisco Systems, http://www.cisco.com/en/US/partner/products/sw/cscowork/ps402/products_user_guide_book09186a00801f42b9.html

Cisco (2004) *Turbo Access Control Lists*, Cisco Systems, http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120s/120s6/turboacl.htm

Colton, A. (2002) *Cisco IOS for IP Routing*, Rocket Science Press Inc.

Davies, J.N. and Grout, V. (2005) Network Monitoring and Measurement, *First International Conference on Internet Technologies and Applications* (*ITA 05*), Wrexham, North Wales, UK, 7th – 9th September 2005.

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.

Hari, B., Suri, S. & Parulkar, G., (2000) Detecting and Resolving Packet Filter Conflicts, *Proceedings of the 19th Joint Conference of the IEEE Computer and Communications Societies (INFOCOM00)*, pp1203-1212.

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.