Using a Genetic Algorithm to Design Improved Storage Area Network Architectures

Elizabeth Dicke, Andrew Byde, Dave Cliff, Paul Layzell
Information Infrastructure Laboratory
HP Laboratories Bristol

HPL-2003-221
November 4th , 2003*

E-mail: {elizabeth.dicke | andrew.byde | dave.cliff | paul.layzell} @hp.com

Designing storage area networks is an NP-hard problem. Previous work has focused on traditional algorithmic techniques to automatically determine fabric requirements, network topology, and flow routes. This paper presents work performed with a genetic algorithm to both improve designs developed with heuristic techniques and to create new designs. For some small networks (10 hosts, 10 devices, and single-layered) we find that we can create networks which result in savings of several thousand dollars over previously established methods. This paper is the first publication, to our knowledge, to describe the successful application of this technique to storage area network design.

* Internal Accession Date Only Approved for External Publication

Using a Genetic Algorithm to Design Improved Storage Area Network Architectures

Elizabeth Dicke, Andrew Byde*, Dave Cliff, and Paul Layzell

Hewlett-Packard Labs Europe
Filton Road, Bristol, BS34 8QZ, UK
{elizabeth.dicke|andrew.byde|dave.cliff|paul.layzell}@hp.com

Abstract. Designing storage area networks is an NP-hard problem. Previous work has focused on traditional algorithmic techniques to automatically determine fabric requirements, network topology, and flow routes. This paper presents work performed with a genetic algorithm to both improve designs developed with heuristic techniques and to create new designs. For some small networks (10 hosts, 10 devices, and single-layered) we find that we can create networks which result in savings of several thousand dollars over previously established methods. This paper is the first publication, to our knowledge, to describe the successful application of this technique to storage area network design.

1 Introduction

As IT systems and employees become more geographically distributed and it becomes more and more important to access shared data, *Storage Area Networks* (SANs) will become the choice of companies looking for efficient, distributed storage solutions. A SAN is a set of *fabric elements* connecting a set of *hosts* – from which data is requested – to a set of storage *devices* – on which data is stored. The fabric elements are *fabric nodes*, which route data through the network, *ports* on the nodes and *links* physically connecting the ports. A link has a port at each end and a port is the terminal of at most one link. SANs allow for efficient use of storage related resources such as hardware and maintenance personnel, resulting in a storage solution that is more effective than local storage, in addition to being more scalable.

Once purchased, installed, and configured appropriately, a SAN can be a cost effective solution to the storage problem. Recent work has focused on automating this process, since solutions designed by hand to support specified data flow requirements tend to over provision resources by a considerable margin [7]. Efficiency is an important issue because the physical components of a storage area network can cost millions of dollars; An over-provisioned design can waste anywhere from thousands to millions of dollars, depending on the size of the network.

* to whom correspondence should be addressed.

A SAN problem is specified by providing a list of hosts, a list of devices, a list of possible types of fabric nodes, and a description of the network's data flow requirements. Each host, device, and fabric node has a cost, a maximum number of ports that are available to accept links, and a maximum amount of data that may pass through it, called its *bandwidth*. The network's data flow requirements are specified by a list of *flows*, each of which is defined by a source host, a destination device, and a bandwidth requirement. A flow may not be routed through a fabric element which does not have enough remaining bandwidth.

A SAN design specifies a list of each fabric element and its connectivity along with a path for each flow. The aim of an automated SAN designer is to find the cheapest SAN that supports the specified flows, while satisfying the port constraints, bandwidth constraints and non-splitting of flows.

The problem of SAN design can be compared to that of design of other types of networks, as well as the problem of routing data within those networks. However, SAN design is more difficult than other network design problems because in this case, there are also the additional limitations of not being able to split a data flow from host through to device, the limited number of ports available on the nodes, the limited amount of bandwidth associated with each node and port, and the fact that the network topology is not pre-determined. It is NP-hard to find the minimal cost network, and best-known algorithms on state-of-the-art machinery take days to complete for moderate sized problems.

Hewlett-Packard's Appia project [7] has shown that traditional algorithmic optimisation techniques can quickly specify a topology that both satisfies the design requirements and competes with designs created by human SAN experts. While able to quickly determine a possible SAN topology, the Appia algorithms are not guaranteed to find the optimal solution. As a result of the need to find a solution within minutes, the algorithms presented by the Appia group build usable networks following heuristic procedures that have previously shown to yield good networks.

This paper seeks to explore SAN design using genetic algorithms (GAs) to produce well designed SANs. We will discuss work using a GA to evolve SAN topologies which will result in both original buildable designs and improvements to previous designs.

We will show that the use of a genetic algorithm can result in SAN architectures which cost thousands of dollars less than designs created either by traditional heuristic methods, or by Appia. This paper is, to the best of our knowledge, the first publication to describe the successful application of these biologically inspired techniques to SAN design.

The remainder of this paper is structured as follows. In the next section we will discuss previous work relating to network design and routing of data through a network both in terms of other types of networks and in relation to SANs specifically. In Section 3, we will introduce the specific configuration and design of our genetic algorithm and discuss the results obtained both with creation of a storage area network from scratch and given an input of a previously designed network. We will then conclude with suggestions for further work.

2 Background

Automating SAN design is a relatively new research area and prior work specifically relating to Storage Area Networks is limited. In this section we will discuss research undertaken in network topology and network routing, both of which relate to SAN design. We will encompass work relating to both SANs and other types of networks which may be similar in structure and constraints.

2.1 Previous Automated SAN Design Work

Much work has been done by the Decision Technologies Department and the Storage Content and Distribution Department within HP Labs Palo Alto, in order to automate the process of SAN design [7]. They have concentrated on two different algorithms called *FlowMerge* and *QuickBuilder*, which each have different strengths and weaknesses in terms of finding efficient solutions to the SAN fabric design problem. Each will be described in brief, for more details, see [7].

The FlowMerge algorithm begins with a SAN connecting each host to its required device, given a set of flows. This configuration typically results in a large number of port violations (i.e. a node has more links than available ports). These are gradually reduced by considering individual *flowsets*. Each flow is initially considered to be in its own flowset. With each iteration of the algorithm, two flowsets are merged together, choosing an appropriate fabric node, and links to connect hosts and devices appropriately. Each iteration results in a reduction of the number of port violations, or, if that is not possible, a reduction in the cost of the design. The algorithm continues until there are no possible improvements on the design, or there are no other flowsets that may be merged.

Ward, et al. [7] show that FlowMerge is one of the faster performing algorithms for the smaller 10 host, 10 device networks, especially those which have 20 to 30 flows spread fairly evenly throughout the network.

The QuickBuilder algorithm also begins with a SAN connecting each host to an associated device as given by a set of flow requirements. However in this case, the initial SAN configuration includes assignment to a particular port on each device. The configuration is then arranged into port groups, which consist of all connected ports. Each port group is then analysed separately in order to determine fabric node requirements.

While FlowMerge tends to find solutions with many small port groups, the QuickBuilder algorithm tends to find SAN configurations with larger port groups when necessary. This algorithm tends to result in cost effective designs for large networks and those that are more densely populated. QuickBuilder is also faster than the FlowMerge algorithm for large problems (10 times as fast for the largest problems consisting of 50 hosts and 100 devices).

2.2 Automated Design of Other Types of Networks

There appears to be limited published work relating to automated design of Storage Area Networks specifically, aside from that referenced above. Much of the automated design work has been done for other types of networks, and will be discussed in the following sections.

Automated network design is not a new research field. Several researchers have attacked this problem with traditional techniques, for networks with varying constraints. However, there is no other network problem which also contains all of the constraints placed on network design for SANs [7]. Nodes within a SAN have a limited number of ports available for connections to other nodes (i.e. they are degree constrained) and have some cost associated with their use. Also, SAN elements have an associated maximum capacity. A flow must not be split between separate fabric elements (i.e. the network is non-bifurcated). And finally, the set of fabric elements in a SAN is not known in advance.

For example, Gavish ([4]) expresses the network design and routing problem as a combinatorial optimisation problem and uses Langrangean relaxation to obtain close to optimal networks. However, although Gavish's work includes restrictions on node cost and does not allow flows to be split, it does not take into account node capacity issues, as the SAN design problem must.

Network Design With a Genetic Algorithm Intending to improve on the work done with traditional techniques, several researchers have attempted network design with genetic algorithms. Chu et al. [2] describe work done using Genetic Algorithms to design a degree-constrained minimal spanning tree (DCMST). A minimal spanning tree is a collection of edges that joins together all vertexes in a set with a minimum sum of weighted edge values. The degree-constrained modifier implies that there is a maximum number of edges connected to a particular vertex. Like heuristic design of SANs, traditional programming approaches to DCMST design do not scale well. As network size increases, the number of constraints increases exponentially and realistic problems become difficult to solve with traditional mathematics. Encoding the connected components of the network within the genome, both valid and invalid solutions are evolved for a network of n nodes, where each node has varying degree constraints. Invalid solutions may be specified by the genome, in which case an attempt will be made to modify the network in order to make the solution viable. This process, which they call *chromosome repair*, may or may not be successful in producing a valid network. However, it acts as an effective local search mechanism for the genetic algorithm. The fitness of each specified network is measured as the cost of connecting the connected nodes together as specified. It was found that the GA could produce more optimal solutions than the traditional minimisation algorithms supplied, but at significantly higher computational cost. Knowles and Corne [5] also use a genetic algorithm to design DCMSTs, however in their case, they use a genome encoding that only permits generation of valid networks, effectively narrowing the search space to a much more manageable size. They find similar results in that the GA outperforms other compared design methods.

Raidl and Julstrom [6] also use a GA but for designing a bounded-diameter minimum spanning tree (BDMST). A bounded-diameter tree is one which has a maximum number of edges connecting any two vertexes in the graph. They also restrict their generated genomes to only specify valid networks. With this type of network, this GA implementation can outperform the other compared heuristic techniques.

Design of DCMSTs and BDMSTs is similar to SAN design in that both require connecting nodes when each node has a limited amount of connections available. Both problems also require the minimisation of some cost value. Additionally Chu, et al. allow the production of invalid networks. However the SAN design problem has the added issue of the data flow through the network. Each component in the network has its own limit on the amount of bandwidth it has available. Moreover a SAN does not need to be fully connected. Valid, cost-effective solutions will not have all nodes connected to each other. Furthermore, with SAN design, the set of nodes is not known in advance.

3 Methods

In this section we will describe the specific implementation of the genetic algorithm used. We will then present the results of experiments to both further optimise Appia designs and to create new networks. We will show that for small networks, optimisation of Appia designs is possible and we can save over 40% of the original FlowMerge cost. We will also show that it is possible to design networks with a direct-connection initialised GA, although the performance does not always equal that of the Appia improved designs. The outcome of these experiments leads us to investigate the nature of our fitness landscape and solution space which is explored later.

SAN design has two parts: the determination of the SAN topology, and the determination of the routing for the data flows. With the implementation described below, we will determine topology indirectly. The genome will specify only the route a particular flow should take when an arbitrarily large pool of fabric nodes is available for use. The topology is then inferred from the routes specified.

3.1 Genome Encoding

The genome encoding used here is limited to expression of single-layered networks only. For each flow, the genome specifies which one fabric node it should be routed through. Alternatively it may specify that the flow will not be routed through any switch, and therefore that there will be a direct connection between the source and destination of the flow. The fabric node pool is initialised at the start of each run. It contains a number of different types of fabric nodes[1] each with their own unique identification number, ranging for convenience from 1 to n

[1] In this paper we consider two types of fabric node: *switch* and *hub*. For our purposes, a hub differs from a switch in three ways: the cost of the fabric node itself, the cost of the ports on the fabric node, and the amount of incoming bandwidth that the fabric node can handle. The hub itself is cheaper than a switch node. Ports on a hub do not cost anything, but the ports on a switch do. However, a switch's bandwidth

where n is the number of fabric nodes available. A direct connection is specified with a fabric node number of 0. For example, if we have a fabric node pool of two switches, four flows (as defined in Table 1), three hosts, and two devices, we would represent a possible solution as in Figure 1(a). In this example, there is a direct connection between host0 and device0.

Table 1. Flow requirements for an arbitrary example SAN design problem. There are three hosts, and two devices

Name	Source	Destination	Bandwidth Required (MB)
flow0	host0	device0	$1.0e07$
flow1	host1	device1	$5.4e07$
flow2	host0	device1	$6.8e07$
flow3	host2	device1	$9.7e07$

0	1	2	1
host0 device0	host1 device1	host0 device1	host2 device1

(a)

0	1	1	1
host0 device0	host1 device1	host0 device1	host2 device1

(b)

0	1	0	1
host0 device0	host1 device1	host0 device1	host2 device1

(c)

Fig. 1. Possible genomes representing a solution for the SAN design problem whose flow requirements are specified in Table 1. The fabric node pool has two switches. Genomes (b) and (c) are mutations of (a) where the gene at locus 'host0-device1' is mutated

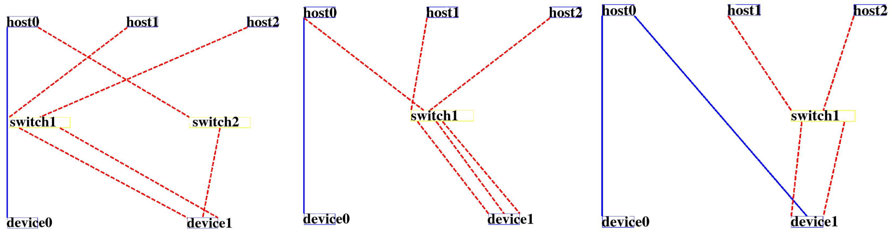

Fig. 2. SANs built from the specification given in Figure 1, from left to right (a), (b), and (c). The solid line represents a direct connection between a host and a device. The dashed lines are links that connect to a fabric node on one end or the other. Each link has a capacity of $10e07$MB. Each host and device has two available ports. The switch has 16 available ports. In each network device1 has a port violation of 1

is only limited to the sum of bandwidth supported by its ports, a hub has additional incoming bandwidth restrictions that are less than the bandwidth supported by the ports.

3.2 Solution Evaluation

A genome specifies a route for each of the flows. A model of the network this assignment represents is then constructed. We can consider the network to be an expression of the phenotype. A network is created from a genome representation in several steps. First, we determine which fabric nodes from the pool are being used (that is those that have flows routed through them). The network built from the genome in Figure 1(c) only uses one of the fabric nodes, while the network represented by the genome in Figure 1(a) uses two. Next, the number of links needed to support the flows is determined. For each flow, its path is determined from the genome specification. If the flow is routed directly between its host and device, a link is created between its source and destination. If the flow is routed through a fabric node, the algorithm first checks to see if there is already a link between the specified source and fabric node, that can support the bandwidth needed by the flow. If there is, then that link is used, otherwise a new link is created between the source and fabric node. For link allocation, we will only be constrained by available bandwidth; we will permit port violations at this stage. Following this method, the genomes in Figure 1 would be built as illustrated in Figure 2.

Once a network has been built, the topology and routing of the flows can then be evaluated. A formula for the *overall cost* C associated with the production of a particular design is given by Equation 1.

$$C = w_1 c_m + w_2 p_{hd} + w_3 p_f + w_4 b. \tag{1}$$

The terms c_m, p_{hd}, p_f and b are normalisations of, respectively, the monetary cost of each of the components necessary, the number of host/device port violations, the number of fabric node port violations, and the amount of bandwidth which is required but not available. The constants w_n, which are set at the start of each run, allow the relative importance of each term to be configured.

The terms c_m, p_{hd}, p_f and b are normalised to lie between 0 and 1 by dividing by an over-approximation of their worst-case values. The worst-case monetary cost c_{m_w} is approximated with the formula expressed in Equation 2, in which n_h, n_d and n_f are the number of hosts, devices, or flows in the problem and c_h, c_d, c_l, c_f and c_p is the monetary cost of a host, device, fibre cable, fabric node, or port.

$$\begin{aligned} c_{m_w} &= (n_h * c_h) + (n_d * c_d) \\ &+ (n_f * 2)(c_l + max(c_P)) \\ &+ (n_f)(max(c_f)) \end{aligned} \tag{2}$$

Each component element of the evaluation function corresponds to a constraint on the design of the network. The number of port violations and amount of over-allocated bandwidth are each a measurement of the 'badness' of un-buildable solutions. We separated the evaluation of host or device port violations from fabric node port violations because early versions of the direct connection initialised GA had difficulty creating designs which were buildable; in this case, it was found that weighting host/device port violations higher than fabric node port violations leads to better solutions.

3.3 Example Cost Calculation

As an example, consider the network pictured in Figure 2(b), as a solution to the SAN design problem specified in Table 1. Suppose that the weights in Equation 1 are $(w_1, \ldots, w_4) = (1, 10, 15, 1)$. Each element in the network has a monetary cost which is supplied as part of the SAN design problem specification. In this example, we will consider the cost of each host or device to be \$100, a switch \$25,000, a link \$50, a port on a host \$0, a port on a device \$500, and a port on a switch \$1,000. We calculate the total monetary cost of this network as $c_m = hostCost + deviceCost + fabricNodeCost + linkCost$ where each node's monetary cost is computed as the cost of its ports added to the cost of itself. The network in Figure 2(b) has three hosts, each with one port, two devices, one with one port, one with three ports, one switch, and seven links. Therefore $c_m = ((100+0)*3)+((100+500)+(100+500*3))+(25,000+1,000*6)+(50*7) = 33850$.

There is one device port violation on device1 ($p_{hd} = 1$) and there are no fabric node port violations ($p_f = 0$).

The amount of bandwidth exceeded should be a sum of the bandwidth exceed on each component. However, when building the network we have already ensured that the links and ports have not been allocated flows that would use more bandwidth than the links have available. Therefore, the amount of available bandwidth exceeded will only be a function of the amount of bandwidth exceeded within a fabric node. Since only hubs have input bandwidth limitations[2], and in this design we have only used switches, $b = 0$. For further explanation, see [3].

Normalisation Each value is now normalised to lie between 0 and 1 by dividing by an over-approximation of their worst-case values. For monetary cost, the worst network, in our example, will be one that requires a fabric node for each flow ($c_{m_w} = 109,300$). For port violations, the worst network will be one that routes each flow through the same fabric node, and yet uses a unique link for each. For p_{hd_w} we assume all flows originate/terminate in the same host/device and each node has an allowance of zero ports ($p_{hd_w} = 8$ and $p_{f_w} = 8$). Since we are not using hubs in this example, even in the worst case we will never use more bandwidth than is available due to the manner in which the links are allocated. Therefore, $b_w = 0$. Normalising all values and applying the weights we can then calculate $C = 0.309698 + 1.25 + 0 + 0 = 1.5597$.

[2] This is because a switch's bandwidth limitation is accounted for by its port bandwidth limitations which have already been taken care of by the network building algorithm.

3.4 Determining Size of the Fabric Node Pool

The size of the fabric node pool is established at the start of each run. The size of the pool contributes to the size of the search space. If the pool is initialised with many more fabric nodes than necessary, the search space becomes much bigger than necessary; if the pool is too small, then feasible solutions may not exist. Either way, the GA has difficulty finding feasible solutions. For the purposes of this paper, the size of the fabric node pool was determined from the number of fabric nodes present in an Appia QuickBuilder solution to the problem plus some constant (2 for the small 10 host, 10 device problems). So, for a SAN design problem with an Appia solution of 4 hubs and 6 switches the fabric node size passed to the GA will be 15. Which consists of $4 + 2$ hubs, $6 + 2$ switches, and 1 direct connection.

3.5 Creating a New Generation

Each generation is created through a series of several steps. First every member of the population has its overall cost evaluated as described in Section 3.2. Each solution is then ranked from best to worst (i.e. those genomes with the lowest overall cost are best). Next, two members of the population are chosen with preference given to better solutions. A single-point crossover operator is applied to the selected genomes with a probability of 0.05. One of the resulting genomes is then chosen at random as the 'child' and the mutation operator is applied to each gene with a probability of 0.01.

When a mutation occurs at a particular locus, a random number is chosen between 0 and n, to represent a new route for a flow. A mutation always results in a new value for a particular gene. Therefore, a mutation will be just as likely to result in a direct connection as it will be to result in the routing of the flow through any of the other fabric nodes. A mutation of the locus labeled 'host0-device1' in Figure 1(a) is equally likely to result in a gene with a value of '1' (Figure 1(b)) or '0' (Figure 1(c)).

Successive pairs are chosen as 'parents' and produce a 'child' until the new population size is one less than the size of the old population. The last member of the new population is an exact copy of the most fit member of the previous population. This ensures the fittest member is not lost through mutation or crossover (i.e. elitism is used).

4 Experimental Results

The GA was tested using two different initialisation methods. The first method is to initialise each member of the population so that each flow requirement is met by directly connecting its source host to its destination device. This method is called "direct connection initialisation". This is similar to the initial step in the FlowMerge algorithm. The second method is to initialise each genome with a buildable, though potentially sub-optimal solution from one of the Appia algorithms. This method is called "Appia initialisation". In each case, over successive generations, the GA will evolve new networks, routing the flows through available fabric nodes.

4.1 Test Data

The Appia project [7] has generated, a set of random test cases classified into nine distinct groups. Each test case has a possible solution, though the optimal solution is not necessarily known. Each group has two specific characteristics, one which represents the number of hosts and the number of devices, and the other which categorises the number of flows between host, device pairs. There were three possible categories of size: problems with 10 hosts and 10 devices, 20 hosts and 100 devices, and 50 hosts and 100 devices. The results presented in this paper are only for 10 by 10 problems. The flows were then characterised by three labels: sparse (a few number of flows generally uniformly distributed across possible host-device pairs), dense (a large number of flows generally uniformly distributed) or clustered (a small number of host-device pairs carry most of the flow requirements).

This same test set was applied to the GA described above, in order to measure its effectiveness against the more traditional algorithms developed and applied in the Appia project. Each grouping of sparse, clustered, and dense problems was numbered from 1 to 30. The first 10 represent problems whose hosts/devices have a higher maximum percentage of *port saturation* (i.e the the proportion of the maximum bandwidth that maybe used on a particular host or device). The last 10 have a higher number of maximum flows per individual host or device.

4.2 Results

Direct Connection Initialisation The GA was initialised with genomes representing a network with all direct connections. It was then run for 1000 generations with a population size of 100. The weights corresponding to Equation 1 were set to $(w_1, \ldots, w_4) = (1, 10000, 1000, 100)$. Since the weighting is applied after the normalisation, this tiered weighting ensures that a solution with host/device port violations is always worse than one without host/device port violations, even if the solution with no host/device port violations has the maximum possible number of fabric node violations. The idea is that as solutions are evolved the number of host or device port violations will be decreased first until there are none. Only then will a focus on decreasing the number of fabric node port violations occur, and so with bandwidth and then monetary cost. This ensures that the GA will find buildable solutions first, and only then will monetary cost become a consideration.

We ran the GA for each of 30 sparse problems, 30 clustered problems, and 30 dense problems for a 10 host, 10 device problem. The GA generated a buildable solution in 62% of the problems. There were 27 buildable sparse solutions, 29 buildable clustered solutions and only 10 buildable dense solutions.

Figure 3 shows the change in overall cost C over 300 generations for the entire population in the run that evolved the solution pictured on the left of

Figure 4. This corresponds to sparse problem 11. The light coloured bands are caused by jumps in the calculated overall cost due to a change in the number of port violations. This banding is discussed further in Section 4.2.

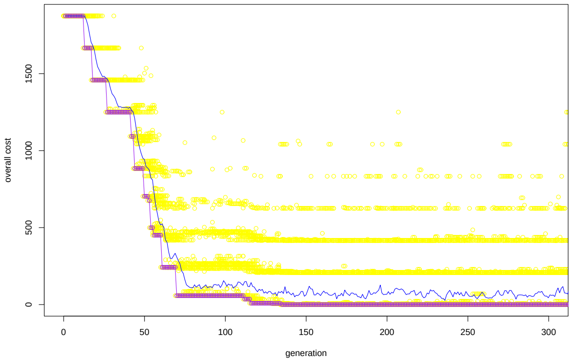

Fig. 3. Change in overall cost of all members of the population over 300 generations for sparse problem 11. The GA was initialised with direct connections. Each light coloured dot represents one member of the population. The thin, wavy, dark line is the average cost. The solid, thick, dark line across the bottom is the cost of the most elite member

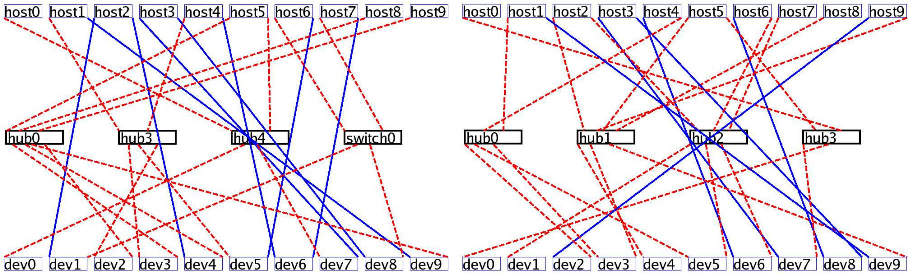

Fig. 4. On the left, an example of a network designed by the GA initialised with direct connections. After 1000 generations, monetary cost is $52,470. The network on the right is the corresponding Appia QuickBuilder solution, cost is $28,470. These solutions correspond to sparse problem 11

Table 2 compares the average solution monetary cost of each algorithm, for those cases where the GA was able to construct a solution.

Figure 5 shows the percent improvement of the GA's solutions over the Appia solutions for those cases in which the GA was able to find a solution within 1000

Table 2. Comparison between average solution monetary costs for direct-connection initialised GA, FlowMerge, and QuickBuilder Algorithms. Negative values indicate the GA did not perform as well as the specified Appia algorithm

		Average Cost			min(FM,QB)-GA	
Type	n	GA	FM	QB	Average	Standard Deviation
Sparse	27	$65,088	$46,629	$51,113	$-21,023	$17,584
Clustered	29	$50,512	$50,309	$54,912	$-2,004	$20,844
Dense	10	$88,470	$100,404	$147,700	$11,934	$27,308

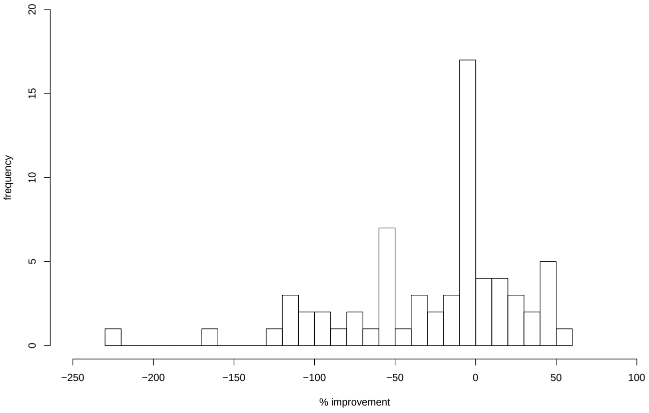

Fig. 5. Percent improvement of direct-connection initialised GA evolved networks over the lower of the Appia QuickBuilder or FlowMerge costs

generations. Although, the direct connection initialised GA does not generally out perform either of the Appia algorithms, it is able to create some better networks, especially for problems characterised as clustered or dense.

Appia Initialisation The GA approach to SAN design appears to be most effective when initialised with a design which has been provided by one of the Appia algorithms. To illustrate this, we initialised each member of the population with the cheaper solution produced by either the FlowMerge or QuickBuilder algorithm. The GA then evolved modified solutions, resulting in equal or lower cost designs. In many of these cases, the GA can find a solution that is less expensive than the cheapest Appia solution, which still satisfies the flow and feasibility requirements. An illustrative change in overall cost C for the most fit population member in one run of the GA initialised with the FlowMerge solution to clustered problem 1 is shown in Figure 6. The fittest member present in generation 1 is the representation of the original Appia solution fed to the GA. We see clearly here, that the fittest genome in each generation successively evaluates at lower and lower cost. The banding occurs because of the weighting factors applied.

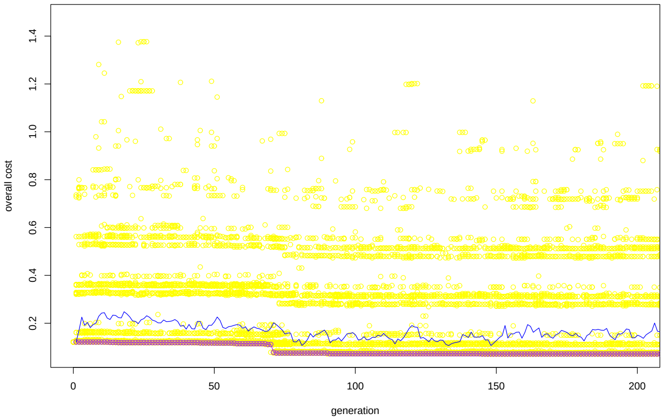

Fig. 6. Change in overall cost of all members of the population for clustered problem 1 over 200 generations after which no improvement occurs. Format as in Figure 3

For these experiments, we used the same 30 sparse, 30 clustered, and 30 dense sample problems as in the direct connection initialised experiments described in Section 4.2. The weights in Equation 1 were, however, changed to $(w_1, \ldots, w_4) = (1, 10, 15, 1)$. This puts slightly more emphasis on the fabric node port violations, over the host/device port violations. This GA was able to find better solutions 71% of the time. That is, in 23 of the 30 sparse problems, 18 of the 30 clustered problems, and 16 of the available 20 dense problems. The remaining 10 dense problems had Appia solutions which were for multi-layer networks. Since our genome representation only encompasses single layer networks there was no way to initialise the population with these solutions. Therefore, there was no data collected on improvement of Appia generated designs for these particular dense networks.

Table 3 shows the quantified ability of the GA to redesign the SAN topology and routing so that the monetary cost of the new SAN is cheaper.

These improvements over the Appia algorithms, summarised in Table 3 are shown graphically in Figure 7. The improvement in design over the Appia solutions results generally from a slight re-arrangement in flows in order to take advantage of already available components. For example, a GA solution to sparse problem 1 takes advantage of an available path through an existing switch, instead of creating an additional direct connection between a host and its corresponding device. The use of an already available route is cheaper than the use

Table 3. Comparison between average solution monetary costs for Appia initialised GA, FlowMerge, and QuickBuilder Algorithms

		Average Cost			min(FM,QB)-GA	
Type	n	GA	FM	QB	Average	Standard Deviation
Sparse	30	$45,142	$48,735	$53,923	$1,286	$1,423
Clustered	30	$46,328	$51,722	$57,206	$3,713	$7,550
Dense	20	$90,280	$94,766	$126,146	$4,062	$3,864

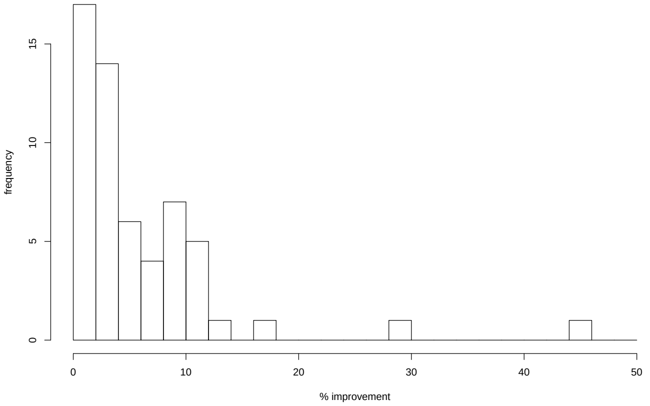

Fig. 7. Percent improvement of Appia initialised GA evolved networks over the lower of the Appia QuickBuilder or FlowMerge costs

of an unnecessary host or device port and additional link and in this case, gives savings of $620.

Other improved solutions will result in the use of a smaller number of fabric nodes, which leads to a more significant cost savings. For example, Figure 8 shows an Appia solution and a resulting improved solution found by the GA for clustered problem 1. The improved solution uses one less switch resulting in a monetary cost difference between the two designs of $33, 220.

The experiments described in this paper have concentrated solely on input designs which were single layered. More complex multi-layer designs were not considered. This has limited us to the exploration of relatively simple SANs. However, the above-described results have shown that in many cases, we can improve the design of a SAN generated by Appia to decrease the monetary cost needed to support the required flows. It is expected that this cost improvement will also be sustained when larger, more complex networks are explored.

Landscape Exploration The results obtained with both direct connection initialisation and Appia initialisation of the GA indicate that the fitness landscape

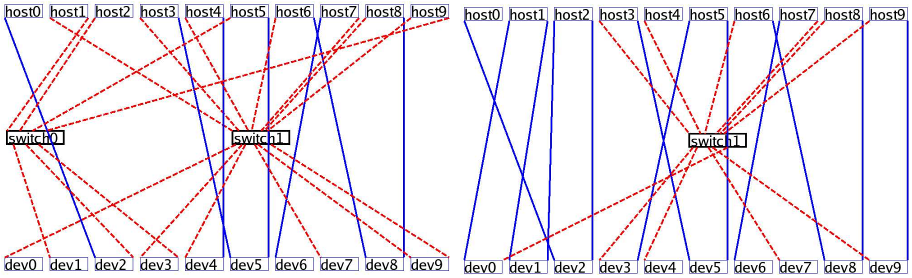

Fig. 8. On the left, SAN designed by the Appia FlowMerge algorithm at a cost of \$80,990. On the right, a lower monetary cost SAN designed by a GA given the design on the left as input. The improved solution is the one evolved by the GA whose solutions' overall costs are shown in Figure 6. The monetary cost savings is \$33,320

is probably very spiky. To show this, several brief experiments were undertaken. Each experiment used a simple Appia generated SAN design problem of ten hosts, ten devices, twenty flows and high port saturation. Ten hubs and ten switches were provided for use, resulting in a possible search space of 21^{20} possible genomes[3]

First, $100,000$ genomes were randomly created. For each genome an overall cost was calculated using the formula for C introduced in Section 3.2. The weights in Equation 1 were $(w_1, \ldots, w_4) = (1, 10, 15, 1)$. Each component (host/device port violations, fabric node violations, overused bandwidth, and monetary cost) was normalised against maximum values of $c_{m_w} = \$666,030$, $p_{hd_w} = 50$, $p_{f_w} = 50$, and $b_w = 1.037e09$. The calculated overall cost of each genome is displayed as a histogram in Figure 9. Given the above conditions, for this design problem, the overall cost of a buildable network will always fall below 0.2. None of the $100,000$ randomly generated networks were buildable. A majority of the values (81.3%) lie between 2.325 and 2.882. We can especially see the banding of solution costs, where 29.3%, 27.7%, and 13.3% of the genomes have costs in these bands from left to right respectively. This banding is due to the weighting of each component comprising the calculated overall cost. Each band consists of solutions with the same number of port violations. The variations within a band are due to differing bandwidth over-allocations and monetary cost fluctuations.

Several different genomes can specify equivalent networks and several non-equivalent networks can have the same evaluated overall cost C. This results in many non-equivalent genomes with the same overall cost. Certain costs (and therefore certain cost ranges) have more corresponding genomes than others due to the increased number of possible ways of comprising particular costs.

From the above, we can presume several properties of our landscape. First, there are several different genome representations which result in equivalently

[3] Twenty possible flows, each with twenty-one possible assignments. That is, a flow may be assigned to one of twenty fabric nodes or to a direct connection.

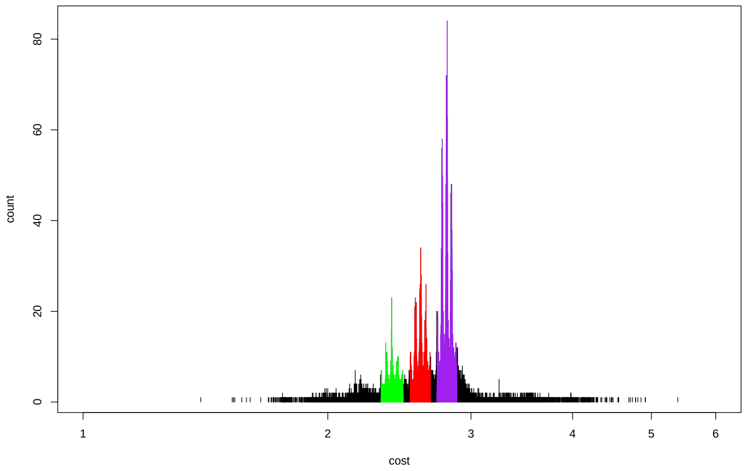

Fig. 9. Histogram of overall cost of 100, 000 genomes produced by random generation. Maximum possible value is 27. The cost axis is limited to [1,6]. Not shown is one cost value at 21. Colouring is used to highlight the individual banding patterns which are discussed further in the text

costed networks. Second, even for small problems, the search space can be quite large. Additionally the proportion of solutions which are buildable is quite small[4].

Secondly, we looked at how the mutation of a genome affects the resulting network's overall cost. Beginning with an Appia solution to the problem, represented as a genome, the mutation operator was applied 10, 000 times using a mutation rate of 0.01 (the same as that used in the genetic algorithm experiments shown previously). The overall cost of each genome relative to the worst possible overall cost for this storage system configuration problem was calculated and is displayed as a histogram in Figure 10. There were 10 hubs and 10 switches available for use. Due to the weighting of each component of the overall cost function (10 for host or device port violations, 15 for fabric node port violations, and 1 for monetary cost and bandwidth values), we see the characteristic banding. Even though there were over 2,000 mutated genomes produced, only four new genomes had an overall cost less than that of the Appia solution (0.179%).

A majority of the mutations take the genome into the second or third band (i.e. one or two port violations), while the remaining mutated genomes remain in the lower band. Those that remain, have variations in overall cost for other reasons such as over-used bandwidth or difference in monetary cost). A majority of the genomes in the first band that are not equivalent to the original genome

[4] Definitely less than 1/100, 000 as illustrated previously; other experiments conducted, but not reported here showed the proportion of buildable solutions was less than 1/350, 000.

(578 of the 584 new genomes in the first band) are buildable. These buildable genomes comprise 25% of the new genomes produced in all the bands.

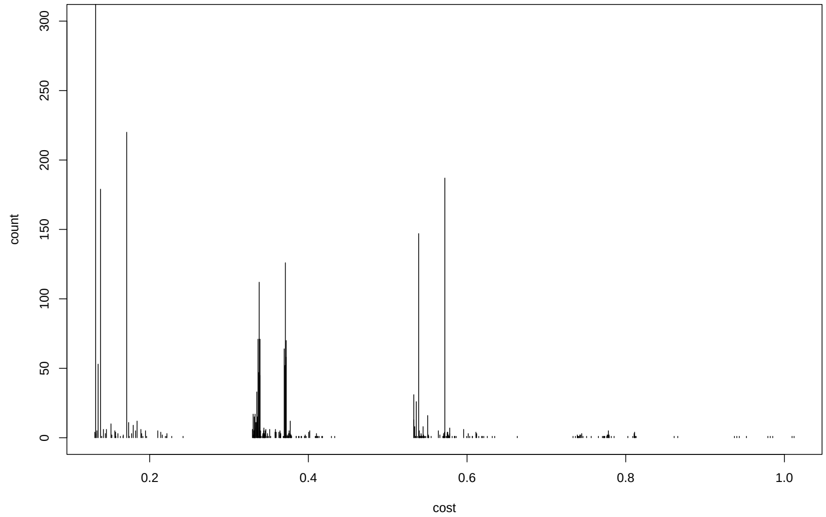

Fig. 10. A histogram of the overall costs of genomes produced by mutating an Appia solution with overall cost 0.1319. The mutation rate was 0.01, maximum value is 27. The count axis has been restricted to [0,300]

In order to ensure this was not a property of mutations of buildable solutions only, we generated a random, non-buildable genome, which was then mutated in the same manner as the Appia produced, buildable solution. The distribution of overall costs is displayed in Figure 11. Again we see the characteristic banding roughly at 0.2 intervals (which corresponds to an additional host/device port violation). As in Figure 10, the range of produced cost values, does not exceed 1.0. However, in this case 96.7% of the costs fall within the same band as the original genome's overall cost. Of these, 1,630 are newly generated genomes, and 8,038 are exact copies of the original. Leaving 16.9% of new genomes outside the main band of costs. This implies that the overall cost of most mutated genomes is close to the overall cost of the original genome.

These two experiments in mutation show that even a small change in a good genome can result in a drastically worst overall cost evaluation, although a small change in a bad genome tends to result in a genome with a similar overall cost. We can interpret this as meaning that the fitness landscape (inverse cost) looks something more like a very spread out Manhattan sky-line rather than rolling hills.

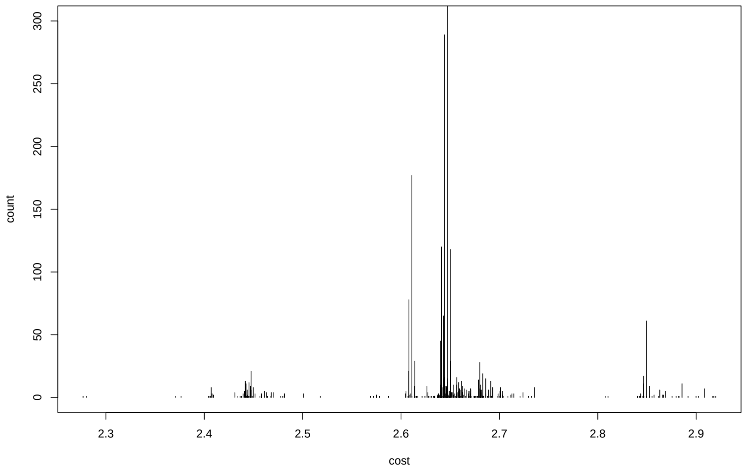

Fig. 11. Histogram of overall cost of genomes produced by random mutation of a non-buildable genome. Original genome had cost of 2.64708. The count axis has been restricted to [0,300]

5 Conclusions

In Section 3 we found that a GA which encoded single layered networks could evolve buildable networks for small, 10 host by 10 device problems. These solutions, when evolved from a directly connected network, do not generally outperform the Appia FlowMerge solutions, but can, about half of the time, outperform the QuickBuilder solutions. However, when the GA is initialised with an Appia solution, in most cases it can quickly evolve a better solution. The results of these two experiments are consistent with the data gathered concerning the shape of the solution space. We found previously that when presented with a non-buildable SAN design solution the mutation operator used in this GA implementation tends to produce designs of similar poor quality. This is reflected in the results obtained when the GA was run with direct connection initialisation. The GA in this case has difficulty finding good solutions, and when it does find one, it is still not one of the best.

However, when a buildable solution is mutated (as in the Appia initialised experiments), some solutions produced are close in fitness to the original, and these will be some of the few that are found by the GA as improvements. With Appia initialisation, the GA is effectively acting as a local search algorithm.

To summarise, we have shown that it is possible to design cost-effective storage area networks using a genetic algorithm. The use of either technique can result in solutions which provide significant monetary cost savings over Appia designed networks.

There are several different directions in which the work presented here could be taken in the future. First in terms of the problem representation and solution space, we made a specific decision that we limit possible solutions to include only single-layer networks. This may prevent the GA implementation from finding a more optimal solution. Further work will explore GAs for designing multi-layered networks.

We saw in Section 3 that the fitness landscape corresponding to the genome representation and overall cost function used for the genetic algorithm was not smooth. It may be possible to represent the problem in a manner that provides a smoother landscape. Further exploration of problem representation may prove to aid in the discovery of buildable networks for larger SAN design problems. Previous work [6], [5], [1] has shown that GAs where the genome and its mutation operators were limited to produce only valid networks can give good results. This may also prove to be useful effective design of large networks.

In this same section we also demonstrated that the GA was an effective tool for optimising already designed networks. One of the problems in SAN design, is that flow requirements have a tendency to change over the life-time of a SAN solution. Therefore, the SAN needs to be redesigned to accommodate a change in needs. The GA presented here should also be an effective tool for re-designing SANs when the requirements change, including an increase in the number of flows.

This work has addressed only some of the important issues of SAN design. There are also considerations of fault tolerance. Sometimes a client will request fault tolerance to be built into a SAN such that each flow has two possible routes in case of failure in a fabric element. To become more useful, the automated GA design approach will need to take into account these fault tolerant properties.

References

1. Hsinghua Chou, G. Premkumar, and Chao-Hsien Chu. Genetic algorithms for communications network design - an empirical study of the factors that influence performance. *IEEE Transactions on Evolutionary Computation*, 5(3):236–249, June 2001.
2. Chao-Hsien Chu, G. Premkumar, Carey Chou, and Jianzhong Sun. Dynamic degree constrained network design: A genetic algorithm approach. In *Proceedings of GECCO-99 (Genetic and Evolutionary Computation Conference 1999)*, pages 141–148, 1999.
3. Elizabeth K. Dicke. Designing storage area networks with biologically-inspired approaches. Master's thesis, University of Sussex, September 2003. To be published as an HP Labs Technical Report.
4. Bezalel Gavish. Topological design of computer communication networks - the overall design problem. *European Journal of Operational Research*, 58:149–172, 1992.
5. Joshua Knowles and David Corne. A new evolutionary approach to the degree-constrained minimum spanning tree problem. *IEEE Transactions on Evolutionary Computation*, 4(2):125–134, July 2000.

6. Gunther R. Raidl and Bryant A. Julstrom. Greedy heuristics and an evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In G. Lamont et al., editor, *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 747–752, 2203.
7. Julie Ward, Michael O'Sullivan, Troy Shahoumian, and John Wilkes. Appia: automatic storage area network fabric design. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, pages 203–217, January 2002.