Chapter 4

GA/Heuristic Hybrid Network Design

*Although genetic algorithms using binary representation and single-point crossover and binary mutation are robust algorithms, they are almost never the **best** algorithms to use for any [specific] problem.*

Lawrence Davis, 1991

This chapter presents key details of NOMaD's object-oriented network toolset, briefly introduced in the context of the overall NOMaD architecture in §3.5 above. A description of the central features of NOMaD's network model is provided in §E.3, but here the author's genetic-algorithm (GA)/heuristic hybrid approach to multi-wavelength all-optical transport network design is described.

Given the large size of the NOMaD software, discussion of the design and implementation, both here and in Appendix E, is necessarily restricted to a somewhat high-level view. However, where it is deemed appropriate, more detailed description will be given. As an aid to the reader in assimilating this, §E.2 contains class cross-reference tables, which cite class descriptions and diagrams throughout the thesis. Further, although the classes are discussed both in terms of their functionality and their underlying data structure, it should be noted that the users of a class can only interact with it through its public interface [15, 16]. In almost all cases, this consists entirely of public member functions [16], to ensure rigid enforcement of data hiding; to a limited extent, however, some classes use protected data members and class 'friendship' to allow non-public access [16].

Throughout the chapter, and in Appendix E, when a particular class is discussed, its parameters may be briefly mentioned. However, the tables of class parameter names and their default values (where appropriate) are provided in §E.1.2. Classes obtain their parameters from a parameter database object, itself constructed from a NOMaD parameter file. NOMaD's use of execution-time class parameters affords considerable flexibility, allowing the user to make significant changes to class (and hence program) functionality without recompilation. Further details on parameter databases can be found in §E.6, and NOMaD file formats are discussed in §E.1.

The chapter is divided into three sections: §4.1 provides details of three metrics for network design assessment: one for minimum-cost topology design, another for network wavelength requirement, and the final one for minimum-cost wavelength allocation. Then §4.2 presents NOMaD's evolutionary algorithm framework: the overall structure of the GA/heuristic hybrid; operators for network topology design; operators for routing, fibre and wavelength allocation; and the operator-probability adaptation mechanism. Finally, the chapter is summarised in §4.3.

4.1 Metrics for Design Assessment

To enable the assessment of network designs, expressed using the classes of NOMaD's network model (§E.3), the object-oriented network toolset provides an abstract base class, `Metric`, from which users can derive concrete classes for their individual design-assessment metrics. As was mentioned in §3.2, the author has developed three such metrics, described in the following subsections, and implemented as the classes: `AlesCost`, `NetWaveReq` and `PhyNetCost` (see Fig. 4.1).

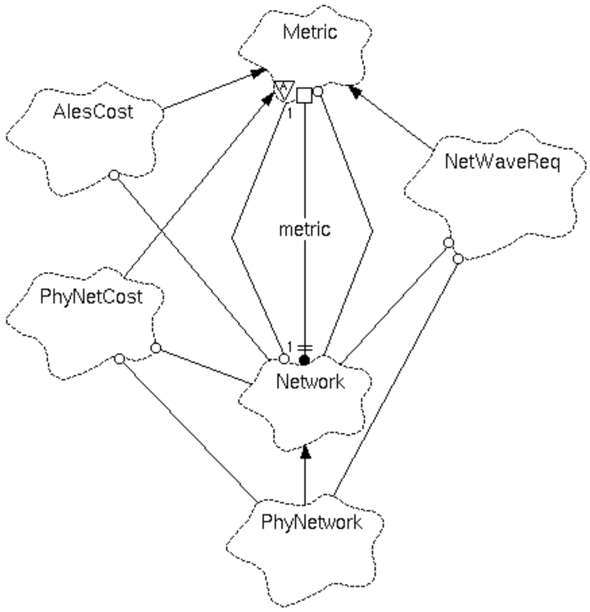


Figure 4.1: `Metric` class diagram

The `Metric` class constructor requires a single argument, the metric name, which can be up to four characters, and must be unique within the toolset. It is this name which is subsequently used to select the metric in a parameter file. For example, `AlesCost` uses the name `"ALCT"`, `NetWaveReq` uses `"NWR"`, and `PhyNetCost`, `"PNCT"`. The name is stored, and can be retrieved using a member function. The abstract virtual [16] member function `assess()`, which takes as a single argument the `Network` (or its derived class, `PhyNetwork`) to be assessed, is provided as a placeholder for concrete derived classes, which must implement it for themselves. In addition to being able to apply a `Metric` to a `Network` in this way, the `Network` class itself stores a pointer to a `Metric` object, which can thus be used for self-assessment: this ability is described (in the context of the `Network` class) in §E.3.1.

4.1.1 Network Cost Model A (`AlesCost`)

The original version of this cost model was developed by the author for minimum-cost topology design of the European Optical Network (EON) as part of COST 239 [291]. It was first presented at ICANNGA'95 in Alès [166], and as a result the corresponding NOMaD class was named `AlesCost`. However, the actual model presented here has been extended by the addition of the constant α in Eqn. 4.3 (below), which allows non-linear dependence of link cost on capacity, a possibility suggested as further work in the paper itself [166].

The model assumes that static two-shortest-node-disjoint-path routing [292] is used between node pairs, and that a reliability constraint is employed. The latter is to ensure that there are two, usually fully-resourced, node-disjoint routes between node pairs, thereby guaranteeing that the network will survive the failure of any single component (node or link). However, no attempt is made to do detailed modelling of the potential sharing of restoration capacity (if protection is not required for multiple simultaneous link or node failures), and this may lead to an overestimation of the required network capacity. Instead, the model simply provides a parameter (K_R) to set the desired level of capacity on the restoration routes; in the author's use of the model, this has usually been set to 1.0 (*i.e.* 100% restoration capacity) whereas a more restricted level of 0.4 (*i.e.* 40%) might well prove sufficient [293]. In addition, as the model was intended for network topology design, wavelength-allocation is not modelled, and as a result the model may underestimate the capacity required for wavelength-path (WP) routing [294], due to the presence of dark wavelengths.

To determine the cost of a given topology, separate models for both links and nodes are required. The intention is to approximate the relative contribution to purchase, installation and maintenance costs of the different network elements, while ensuring that the model is not too dependent on the details of the element designs, nor too complex for use in the 'inner loop' of a design procedure.

First, the two-shortest-node-disjoint routes are determined for all node pairs. In this, the 'length' of each path is taken to be the sum of the contributing link weights, with the weight of the bi-directional link between nodes i and j given by:

$$W_{ij} = 0.5N_i + L_{ij} + 0.5N_j \tag{4.1}$$

where N_i and N_j are the *node effective distances* of nodes i and j respectively (see below) and L_{ij} is the length of link (i,j) in km. Then, the link carried traffic is determined for each link by summing the contributions from all the primary and restoration routes that make use of it. As mentioned above, the restoration traffic is weighted by K_R, but restoration routes are usually taken to be fully-resourced, and thus are assumed to carry the same traffic as the corresponding primary routes *i.e.* the traffic they would be required to carry if their primary route failed. The capacity of link (i,j) is taken to be:

$$V_{ij} = \mathrm{ceil}_{K_G}(K_T T_{ij}) \tag{4.2}$$

where T_{ij} is the carried traffic in Gbit/s on the link, and $\mathrm{ceil}_x()$ rounds its argument up to the nearest x—here, the assumed granularity of the transmission links, K_G (say 2.5 Gbit/s). The factor of K_T is to allow for stochastic effects in the traffic[1]. The cost of link (i,j) is then taken to be:

$$C_{ij} = V_{ij}^{\alpha} L_{ij} \tag{4.3}$$

where α is a constant. Increasing capacity necessarily implies increased cost due, for example, to wider transmission bandwidth, narrower wavelength separation and/or increasing number and speed of transmitters and receivers. With $\alpha = 1$, a linear dependence of cost on capacity is assumed, but with $\alpha < 1$, the cost can be adjusted to rise more slowly with increases in the link capacity [295]. The linear link length dependency approximates the increasing costs of, for example, duct installation, fibre blowing and/or the larger number of optical amplifiers with increasing distance.

Node effective distance was used as a way of representing the cost of nodes in an optical network in equivalent distance terms. It can be regarded as the effective distance added to a path as a result of traversing a node. By including it in link weights (Eqn. 4.1) for the calculation of shortest paths, path weights reflect the cost of the nodes traversed (and half the costs of the end nodes). As a

[1] Following discussions in COST 239, K_G was taken to be 1.0 in the author's subsequent metrics, *i.e.* traffic taken to be identical to capacity requirement, and the factor quietly dropped.

result, a longer geographical path may have a lower weight if it traverses fewer nodes, thereby reflecting the relatively high costs of optical switching. The node effective distance of node i was taken to be:

$$N_i = K_0 + n_i K_n \tag{4.4}$$

where n_i is the degree of node i, *i.e.* the number of bi-directional links attached to it. The constants K_0 and K_n were taken to be say 200 km and 100 km, respectively. Node effective distance thus increases as the switch grows more complex. Node capacity is the sum of the capacities of all the attached links, *i.e.*

$$V_i = \sum_j V_{ij} \tag{4.5}$$

where V_i is the capacity of node i in Gbit/s. The node cost was taken to be:

$$C_i = 0.5 N_i V_i \tag{4.6}$$

The cost is thus derived as if the node were a star of links, each of half the node effective length, and each having the same capacity as the node itself. Further, if all the links attached to a node are of the same capacity, the node costs would grow approximately with the square of the node degree, corresponding, for example, to the growth in the number of crosspoints in a simple optical space switch. Overall, the relative costs of nodes and links can be adjusted by setting the values of K_0 and K_n appropriately.

The network cost is then taken to be the sum of the costs of all the individual links and nodes comprising the network. However, to ensure that the reliability constraint was met, the actual metric used also includes a penalty function [279, 280]. This adds P_R (say 250,000) to the network cost for every node pair that has no alternative path, and P_N (say 500,000) for every node pair with no routes at all between them. This avoids the false minimum of the topology with no links at all, whose cost, under the link and node cost models employed, would be zero. The penalties chosen must be large enough to ensure the final topology design meets the reliability constraint. However, they should not be much larger than this, or they will reduce the speed of (evolutionary) search, unnecessarily focusing the algorithm on meeting the constraint early in the run, rather than finding low-cost topologies with only minor violations.

The cost model and associated routing procedure were implemented in two NOMaD classes, `AlesCost` and `AlesRouter`, respectively. The parameters for these (obtained via a parameter database argument on each of their constructors) are given in Tables E.4 and E.5 on p. 248, and their relationships with other classes are illustrated in Fig. 4.1 on p. 67 and Fig. E.2(b) on p. 260. These two classes are also the default metric and router of class `Network` (see Table E.3 on p. 247).

4.1.2 Network Wavelength Requirement (`NetWaveReq`)

Many authors [29, 35, 296–300] have considered a low network wavelength requirement (*i.e.* the number of distinct wavelengths in the network) to be desirable as a design objective for wavelength-routed networks. This is due to the technological limitations, and resulting cost implications, of carrying more than a small number of wavelengths over national or even international distances [301].

However, the actual metric described here also includes a penalty [279, 280] added for each unsatisfied traffic requirement (*i.e.* wavelength channel that remains unrouted), to avoid the false minimum of carrying no traffic at all, which would otherwise have a network wavelength requirement of zero. A suitable penalty would be a number of wavelengths comparable with, but smaller than, the expected network wavelength requirement (*e.g.* 5 wavelengths for an expected network wavelength requirement in the range 6–10). This provides a balance between allocating all traffic requirements by the end of the (evolutionary) algorithm run, while avoiding undue focus on early satisfaction of the constraints.

This metric, including the node-to-node penalty, was first published by the author in *Electronic Letters* [35] (reprinted in Appendix A). Since then, an additional (optional) per-fibre penalty has been introduced, to penalise a network design with more than one fibre using the highest wavelength number. For every fibre using the current highest wavelength, a small penalty (say 0.01 wavelengths) is added to the overall assessment. For example, this enables two network designs to be distinguished, both with a wavelength requirement of say 8, but one with 5 fibres using the highest wavelength, the other only 2. The first would have an assessment of 8.05, the second, 8.02, indicating the latter's superior quality. The value of per-fibre penalty selected should be such as to avoid increasing the assessment by a full wavelength, even if many fibres use the highest wavelength number. A value comparable to the inverse of the total number of fibres in the network is recommended.

The `NetWaveReq` class was written to implement this metric, and its parameters (obtained via a parameter database argument to the constructor) are given in Table E.6 on p. 248; its relationships with other classes are illustrated in Fig. 4.1 on p. 67.

4.1.3 Network Cost Model B (PhyNetCost)

Network wavelength requirement was presented as a suitable design metric for wavelength-routed networks in §4.1.2 above. However, an objective that consists of a simplified assessment of the cost of the whole network including, at an appropriate level, the cost of exceeding the minimum network wavelength requirement, is arguably superior, and is the metric described here.

This second cost model is a further development of Network Cost Model A, described in §4.1.1 above (AlesCost). It was first published at the 6th IEE Conf. on Telecommunications [36] and subsequently in *IEE Proc. Communications* [37] (both reprinted in Appendix A), and makes Network Cost Model A specific to wavelength-routed all-optical transport networks [294]. It should be noted that although, in assessing similar designs, others have included failure restoration [297, 302] and restrictions on the number of wavelengths per fibre [302,303], for simplicity, these are not considered here.

Given the wavelength-routed network paths, the capacity of link (i, j) is taken to be:

$$V_{ij} = \lambda_{ij} V_\lambda \tag{4.7}$$

where λ_{ij} is the total number of wavelengths in use on the link (*i.e.* across all its fibres) and V_λ is the granularity of the wavelength channels (*e.g.* 10 Gbit/s). For example, in Fig. E.6 on p. 269, $\lambda_{ij} = 12$ and thus V_{ij} is 120 Gbit/s. The *wavelength-requirement* capacity of link (i, j) is taken to be:

$$V_{\lambda,ij} = \lambda_{\mathrm{req},ij} F_{ij} V_\lambda \tag{4.8}$$

where $\lambda_{\mathrm{req},ij}$ is the wavelength requirement of link (i, j) and F_{ij} is the number of fibres in use on link (i, j). The wavelength-requirement capacity is thus the capacity that would result if all the dark wavelengths on all the (partially-used) fibres of the link were filled, up to the current link wavelength requirement. For example, again in Fig. E.6, $\lambda_{\mathrm{req},ij} = 5$, $F_{ij} = 3$ and thus $V_{\lambda,ij}$ is 150 Gbit/s. Since capacity is one of the factors in the link cost calculation (see below), wavelength-requirement capacity thus allows the additional costs of uneven fibre loading and dark wavelengths to be assessed in equivalent capacity terms, and thus included at an appropriate level in the overall link cost.

The *capacity* cost[2] of link (i, j), *i.e.* the link cost omitting any contribution from wavelength

[2] Equivalent to link cost in Network Cost Model A, Eqn. 4.3.

requirement, is:

$$C_{V,ij} = V_{ij}^{\alpha} L_{ij} \tag{4.9}$$

where α is a constant and L_{ij} is the length of link (i, j) in km. Whereas, the *wavelength-requirement* cost of link (i, j) is:

$$C_{\lambda,ij} = V_{\lambda,ij}^{\beta} L_{ij} \tag{4.10}$$

where β is a constant. By setting $\beta \neq \alpha$, the link model can be adjusted to allow for a different dependency of cost on wavelength-requirement capacity. The overall cost of link (i, j) is then taken to be:

$$C_{ij} = \gamma C_{V,ij} + (1 - \gamma) C_{\lambda,ij} \tag{4.11}$$

where γ is a constant $(0 \leq \gamma \leq 1)$. Different values of γ allow the model to reflect the relative contributions to link cost of the capacity cost and the wavelength-requirement cost. With $\gamma = 1$, the link model only takes account of the capacity used, and ignores the link wavelength requirement; whereas, with $\gamma = 0$, the actual capacity used is ignored, and the focus is entirely on the wavelength requirement. An intermediate value of γ, such as 0.5, is regarded as being more representative of the actual costs involved.

For the nodes, node effective distance was again used as a way of representing the cost of nodes in an optical network in equivalent distance terms. However, while in Network Cost Model A it was used to influence the selection of paths in minimum-cost network topology design, here it is only used as part of the node cost model. Although node effective distance is defined as in Eqn. 4.4 above, node capacity here is taken to be the sum of the *effective* capacities of all the attached links, *i.e.*

$$V_i = \sum_j V_{e,ij} \tag{4.12}$$

where V_i is the capacity of node i in Gbit/s and the effective capacity of link (i, j) is given by:

$$V_{e,ij} = \gamma V_{ij} + (1 - \gamma) V_{\lambda,ij} \tag{4.13}$$

The node capacity is therefore dependent to some extent on the wavelength requirements of its attached links. The node cost was then assessed following Eqn. 4.6.

The network cost is once again taken to be the sum of the costs of all the individual links and nodes comprising the network. However, only one penalty (P_N, say 250,000) was needed, which was added for each individual traffic requirement (*i.e.* wavelength channel) unsatisfied, so as to avoid the false minimum of carrying no traffic at all.

The cost model was implemented in a NOMaD class, `PhyNetCost`, the parameters for which (obtained, as usual, via a parameter database argument to the constructor) are given in Table E.7 on p. 248. `PhyNetCost`'s relationships with other classes are illustrated in Fig. 4.1 on p. 67.

4.2 Genetic-algorithm Framework

In this section the focus is on solution approaches, primarily GA/heuristic hybrids, but also, to some extent, overall design heuristics. Both these approaches, supported by NOMaD's object-oriented network toolset, were discussed briefly in §3.3, and GA/heuristic hybrids in particular were introduced in §1.4 and the tutorial of §2.2.3.

In the following subsections, first the overall GA framework (§4.2.1), then the various operators designed by the author for network topology design (§4.2.2), as well as routing, fibre choice and wavelength allocation (§4.2.3) are described. Finally, in §4.2.4, the details of the operator-probability adaptation mechanism, which aims to improve operator productivity within the overall GA framework, are presented.

4.2.1 Overall Structure

The classes discussed in this subsection form the overall GA framework, a brief overview of which was given in §3.5 above. As well as the central framework classes (`GeneticAlgorithm`, `Population`, `Individual` and `OperatorPool`), the abstract classes from which all NOMaD's genetic operators are derived (`Operator`, `UnaryOp` and `BinaryOp`) are described, plus one very special operator, `Reproduction`. The relationships between most of these were illustrated in a class diagram, Fig. 3.5 on p. 59.

`GeneticAlgorithm` class

The `GeneticAlgorithm` class is central to NOMaD's GA/heuristic hybrid approach. It supports a generational, rather than steady-state, algorithm [9], and depends heavily on two other classes: `Population`, which holds the population of evolving network designs, and `OperatorPool`, which not only manages the pool of genetic operators used for population evolution, but also runs the operator-probability adaptation mechanism.

The main `GeneticAlgorithm` class constructor takes three arguments: parameter databases built from both the network specification and the NOMaD parameter files, plus a pseudo-random number generator object (of class `RandomGen`, §E.6). After extracting the GA's one parameter, the maximum number of trials[3] (Table E.8 on p. 248), the constructor uses the parameter databases and random number generator to build both population and operator-pool objects. Both of these, as well as the maximum number of trials, can subsequently be retrieved with the appropriate member functions. In addition, a GA object maintains one additional data member, the current number of trials, which is initialised to zero, and can also be accessed via a member function.

The `GeneticAlgorithm` class' principal member function is `run()`. This takes the desired number of generations as an integer argument, and runs the GA for that number of generations. However, it will terminate prematurely, although always at the end of a complete generation, if either the number of trials exceeds the maximum specified or the appropriate Unix interprocess signal has been received. On completion, the function returns a Boolean indicating whether it would be able to continue for at least one more generation ('run on'). However, if the desired number of generations is specified as zero (the default), the function continues without returning until the maximum number of trials is exceeded *i.e.* the run is completed, or the signal has been received. Nevertheless, although it allows for runs of multiple generations in this way, if recording of population and operator statistics is required every generation (as it is in the `ga` application program, see §3.9), the function is only invoked for one generation at a time.

A flowchart of the `run()` member function is given in Fig. 4.2 below. This shows the overall generational nature of the process, represented by the main loop, including breeding of a new population and its replacement of the old, as well as the use of operator-probability adaptation. However, there are a number of decision points that can end the iteration. As well as receipt of the appropriate interprocess signal, exceeding the maximum number of trials or completing the desired number of generations, there is one further possibility.

To appreciate this, first note that network objects both cache their last assessment result, as is mentioned in §E.3, and also count the number of trials they have used in self-assessment since their construction. This count is made available via a member function, and so the number of trials (*i.e.* assessments) used in constructing a population can be obtained by summing over all its constituent network objects. Consequently, if a new population is bred without consuming any trials, then not

[3] A trial is an assessment of a network design *i.e.* application of the appropriate design metric. The number of trials can be used as a platform-independent measure of algorithm effort, as assessment is often the most computationally-intensive part of algorithm execution.

a single network has been changed in the breeding process: the GA is 'spinning' (*i.e.* has converged to identical or almost identical population members), and the run is terminated.

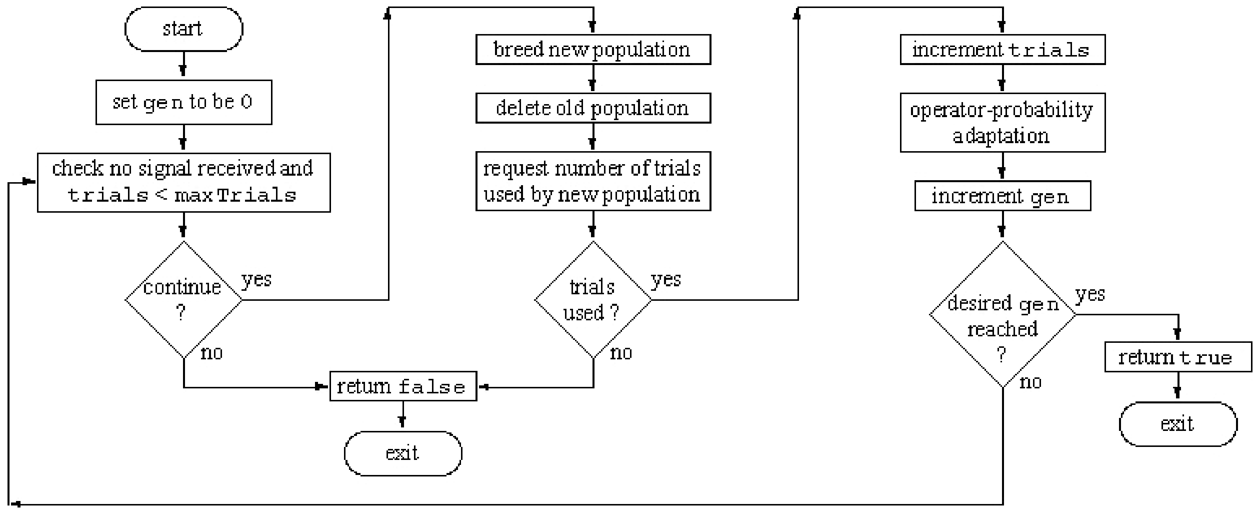


Figure 4.2: Flowchart of the `GeneticAlgorithm` `run()` member function

`Population` class

NOMaD's `Population` class is not only responsible for containing a population of individuals (each a network object plus additional accounting information), but also maintaining several important population statistics.

There are two main `Population` class constructors. One of these, taking both network specification and NOMaD parameter databases plus a random number generator, creates a full population. The other, omitting the network specification parameter database, creates an empty population, to which individuals can subsequently be added. The former constructor is used for initial population construction, whereas the latter is intended as part of breeding a new population from an old one. Both constructors extract and store several parameters (Table E.9 on p. 248): the population size, the window size (w) for best-in-window fitness, the selection function and, if the latter is tournament selection, the tournament size[4]. Member functions are provided to allow the population size, window size and selection function to be subsequently retrieved. In addition, for the former constructor, a choice of initialisation operator may also be given. This is applied to every network object in the population as it is constructed, and can be used to build a random initial population. For example, in network topology design, the default `RandomLinks` operator can be employed (see

[4]While NOMaD has been designed to allow for a choice of selection functions, only tournament selection has actually been implemented. However, there is strong support in the literature for choosing tournament selection [304, 305].

§4.2.2 below), and for routing, fibre choice and wavelength allocation, `WaveRandKSP` (§4.2.3).

Expanding a non-full population is achieved using a member function which enables an individual to be added to the population. At any point the current size of the population can be accessed via another member function, and then, when the population is full, a further member function is automatically invoked to refresh the population statistics. (The latter is also invoked at the end of initial population construction.)

The statistics stored include[5], at the end of the last generation, the highest fitness (*i.e.* assessment) of any individual in the population, the lowest fitness, the average fitness and the median fitness. Further, a record is maintained of the lowest fitness in each generation over a user-defined number of generations (see Table E.9). In addition, a copy of the best individual (*i.e.* of lowest fitness value) seen up to the end of the last generation is made. To access the statistics, member functions are provided for highest, lowest, average, median and best-in-window fitness, as well as one to access a `const` pointer to the best individual (and hence the best-so-far fitness).

Two further member functions are also available: an overloaded '`[]`' operator allows indexed access to the individuals in the population, and the total number of trials used by the population can be obtained, as was mentioned in the description of the `GeneticAlgorithm` class above.

`Individual` class

The `Individual` class is simply a `Network` (or `PhyNetwork`) with some additional accounting information to support operator credit assignment (see §3.5, and Fig. 3.5 on p. 59).

The main constructor takes two arguments: the parameter databases for network specification and NOMaD parameters. From the latter database, a single parameter is taken (Table E.10 on p. 248) which determines whether the individual constructs a `Network` or a `PhyNetwork` using the network database. However, construction of an individual from parameter databases in this way is computationally relatively expensive, and as a result, after the first, the remainder of the individuals in a population are actually created using the copy constructor (both in the initial population and when breeding).

An individual stores a pointer to its network object, as well as pointers to both its parent and grandparent operators (see §3.5, and §4.2.4 below). These latter objects are used by the credit-assignment mechanism to ensure that two- and three-operator sequences are properly rewarded.

[5] NOMaD assumes throughout that the *lowest* fitness value is best.

However, they are not actually maintained by the individual itself, but rather by its friend class, `OperatorPool` (see below).

For member functions, as well as allowing access to its network object as a `const` pointer, an individual also allows indirect retrieval of the underlying network fitness (*i.e.* self-assessment) using its own member function.

`OperatorPool` class

In addition to containing and owning the set of operators utilised in a particular GA/heuristic hybrid algorithm, the operator pool is responsible for both breeding the population and managing the credit-assignment and operator-probability adaptation mechanisms.

The main `OperatorPool` class constructor takes two arguments: a parameter database and a pseudo-random number generator. The random number generator is stored, and from the parameter database several parameters are retrieved (Table E.11 on p. 249). The number of operators (excluding simple reproduction) is stored, and then the names and initial probabilities of the expected number of operators is retrieved from the database. Each of these is constructed and stored, as is the reproduction operator object, whose initial probability is simply whatever remains to ensure the sum of the initial probabilities is 1.0. The remaining parameters, each of which is stored, influence the credit-assignment and operator-probability mechanisms, discussion of which is largely deferred until §4.2.4 below.

As well as a member function to obtain the number of operators and an overloaded '`[]`' operator to retrieve `const` pointers to the individual operators themselves, the `OperatorPool` class provides only two member functions: `breed()` and `adaptOpProb()`.

The first of these creates a new population from an old one. Until the new population is full, it first chooses an operator from the pool at random, according to their current probabilities. Next, for unary operators, it selects an individual from the old population using the designated selection function (*e.g.* tournament selection), creates a new individual from this, and then applies the chosen operator. Both the operator's children and trials counts are increased, by 1 and the number of trials actually used (which may be 0, 1 or more), respectively. Then, the operator itself, as well as the individual's parent and grandparent operators, are awarded any credit earned, according to the applicable credit type (§4.2.4). Next, the individual's grandparent and parent operator data members are updated (the parent operator is copied to the grandparent, and the current to the parent). Finally, the new individual, possibly modified, is added to the new population. A similar sequence, except that *two* parent individuals are used, is executed for binary operators.

The second member function periodically adjusts the individual operator probabilities according to one of the algorithms discussed in §4.2.4 below.

`Operator`, `UnaryOp`, `BinaryOp` and `Reproduction` classes

The abstract virtual base class `Operator` [16,17], and its two derived abstract classes, `UnaryOp` and `BinaryOp` have already been mentioned briefly in §3.5 (Fig. 3.5), and are discussed at more length in §E.3.1 on p. 259 (Fig. E.2(a)). Nevertheless, before reviewing their many concrete derived classes in the following subsections, it is appropriate to review their public interfaces and data structures (Fig. 4.3).

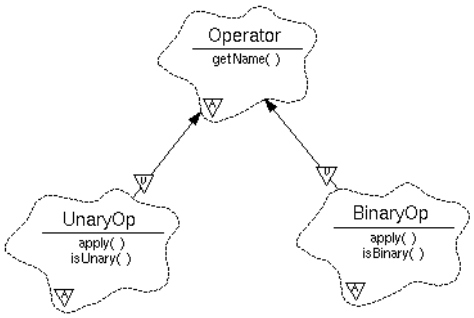


Figure 4.3: `Operator` class diagram

As with `Metric` (§4.1), the `Operator` class' main constructor requires a single argument, the operator name, which can be up to four characters, and must be unique within the toolset. For example, `DegreeTwo` uses `"DEG2"`, and `WaveKSPMutate`, `"WKM"`. In addition to its name, an operator has data members for: the number of children it has generated, the number of trials it has used, its probability of being applied, and the credit it has earned. All the latter four data members are maintained by the friend class `OperatorPool` and all five, including its name, can be accessed via member functions. In addition, the `Operator` class provides two virtual member functions, with the overloaded name `apply()`, that take either one or two network arguments, and return a Boolean to indicate success or failure. In the same way, it provides two virtual Boolean tests: `isUnary()` and `isBinary()`. All four virtual member functions are implemented in the base class to simply return `false`.

As the abstract base class of all unary operators, `UnaryOp` redefines as abstract virtual [16] the version of `apply()` with only one network argument, forcing all its derived classes to re-implement it for themselves. In addition, it also re-implements `isUnary()` to return `true`.

Class	Name	Description
`RandomLinks`	`RLS`	Random links
`MutateLink`	`ML`	Single-link mutation
`DegreeTwo`	`DEG2`	Degree-two operator
`DegreeDec`	`DEGD`	Degree-decrement operator
`RandomAddLink`	`RAL`	Random add link
`RandomRemLink`	`RRL`	Random remove link
`RandomChgLink`	`RCL`	Random change link
`RandomAddLinks`	`RALS`	Random add links
`RandomRemLinks`	`RRLS`	Random remove links
`RandomChgLinks`	`RCLS`	Random change links
`SPLinkCO`	`SPCO`	Single-point link crossover
`LinkGroupCO`	`LGCO`	Link-group crossover

Table 4.1: `Operators` for network topology design

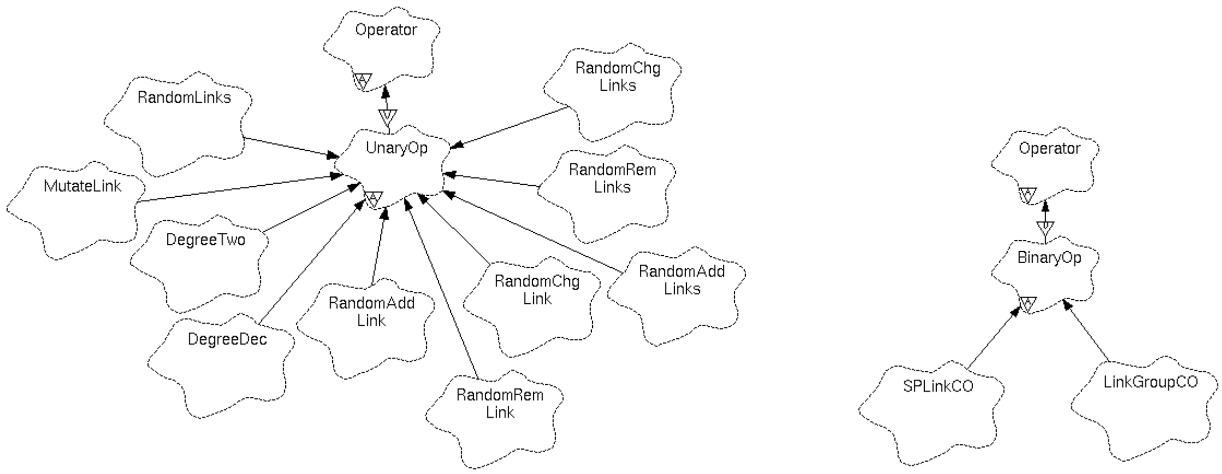
On the other hand, the `BinaryOp` class makes the complementary changes, redefining the other version of `apply()` as abstract virtual to force its own descendants to work with two networks at a time, and re-implementing `isBinary()` to return `true`.

`Reproduction`, derived from `UnaryOp`, is a very special concrete class: it does nothing, apart from return `true`, when applied! The reason for this is that in NOMaD's implementation of the overall GA framework, reproduction (*i.e.* cloning) of parent individuals is actually performed by the operator pool (see above). As a result, when simple reproduction is applied, as an operator, to the child individual, no further actions are required on its part. A reproduction operator object is still necessary, however, as it is capable of earning credit for its part in two- and three-operator sequences, just as any other operator, and is thus very much still part of the overall operator-probability adaptation mechanism (see §4.2.4 below).

4.2.2 Operators for Mesh Network Topology Design

The author has developed twelve operators for network topology design, ten of these unary and two binary. They are listed in Table 4.1, and the relationships between the classes illustrated in Fig. 4.4. Two of these (`RandomLinks` and `RandomAddLinks`) were mentioned in the author's paper at the 12th UK Performance Engineering Workshop [31], and more complete descriptions of another five (`MutateLink`, `DegreeTwo`, `DegreeDec`, `SPLinkCO` and `LinkGroupCO`) published by the author at the 3rd Intl. Conf. on Artificial Neural Networks and Genetic Algorithms (ICANNGA'97) [34] (both reprinted in Appendix A).

The *random links* operator is intended to establish a random network topology for use in the initial population of a GA/heuristic hybrid. For each node pair in the network, it flips an unbiased



(a) Unary (b) Binary

Figure 4.4: Topology-design operators class diagrams

coin to determine whether to directly connect the node pair with a link. It is implemented as NOMaD class `RandomLinks`.

Single-link mutation selects a random node pair and then, if they are already directly connected, removes their link, and if not, installs one. In NOMaD the operator is implemented as class `MutateLink`.

The *degree-two operator* alters a network by adding links to connect nodes to their nearest neighbours, where necessary, to make all nodes at least of degree two (*i.e.* connected to two other nodes). This ensures reasonable connectivity, perhaps avoiding a penalty, and hopefully thereby resulting in an improvement in the network assessment. It is implemented as NOMaD class `DegreeTwo`.

If there is at least one node of degree greater than two in the network, the *degree-decrement operator* removes a random link from the highest-degree node; if there is more than one such, then a random node is chosen from amongst these. Hopefully the removal of this potentially redundant link from such a highly-connected node will improve the overall network assessment. In NOMaD the operator is implemented as class `DegreeDec`.

The *random add link* operator randomly selects (*without* replacement) node pairs that are not directly connected until it finds the first pair between which adding a link improves the network assessment. If such a node pair is found, the link is added, and the operator has succeeded; if not, after all the pairs have been tried once, in random order, the operator has failed, and it leaves the network unchanged. The *random remove link* operator works in a complementary fashion, randomly selecting node pairs that *are* directly connected, until it finds the first pair between which removing their link improves the network assessment. A further variant is the *random change link* operator, which considers *all* node pairs, in random order, looking for the first change, either adding *or* removing a link, which results in an improvement. The three operators, which were inspired by branch exchange heuristics [11], are implemented as NOMaD classes `RandomAddLink`, `RandomRemLink` and `RandomChgLink`, respectively.

The next three operators augment the previous three by including iteration. *Random add links* initially acts identically to random add link, but after the first link is added, it 'forgets' which node pairs it has examined, and looks again for a pair between which adding a link results in an improvement. The operator is only complete when, having examined afresh all node pairs that are not directly connected, it cannot find a pair between which to successfully add a link. However, it is still regarded as having succeeded overall if, during the earlier iterations, it was able to add at least one link. Effectively, random add links is an iterative stochastic hill climber [29, 30]. In much the same way, *random remove links* extends random remove link, and *random change links*, extends the random change link operator. In NOMaD, the three operators are implemented as classes `RandomAddLinks`, `RandomRemLinks` and `RandomChgLinks`, respectively.

Link-group crossover selects a group of nodes in one network: those nodes contained within a circle, centred on a randomly-selected node, of radius defined by a second randomly-selected node on the circle's circumference. It then exchanges all the links used within that group with those used within the same group of nodes in the other network. It was believed that this would lead to better building blocks than a *single-point link crossover*, which assumes a linear encoding of the network topology, as has been employed earlier by some authors, including this one [150, 154, 166]. An example of link-group crossover in action, towards the end of a hybrid run, can be seen in Fig. 4.5. This shows two parent networks recombining by exchanging links between a (shaded) group of four nodes. One of the two resulting offspring (bottom right) is not only of lower network cost than either of its two parents, but was the best-of-run individual. The two operators are implemented as NOMaD classes `LinkGroupCO` and `SPLinkCO`.

4.2.3 Operators for Routing, Fibre & Wavelength Allocation

The author has also developed six routers (*i.e.* overall heuristics) for WP routing, fibre choice and wavelength allocation, plus one for virtual-wavelength-path (VWP). These are listed in Table 4.2, and the relationships between the classes illustrated in Fig. 4.6.

Wavelength-allocation random k-shortest paths is intended for the random, but reasonable ini-

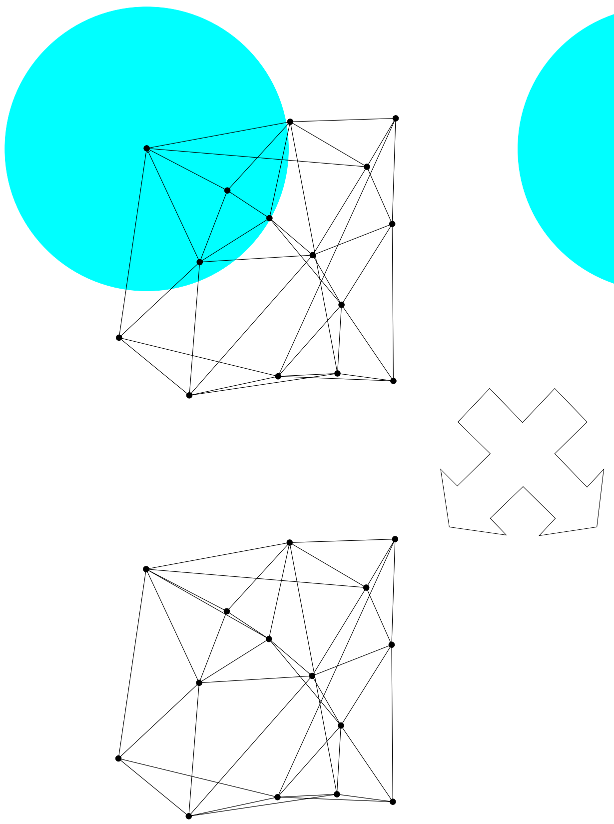


Figure 4.5: Example of link-group crossover

Class	Name	Description
`WaveRandKSP`	WRKS	Wavelength-allocation random k-shortest paths
`WaveNop`	WNOP	Wavelength-allocation no-op
`WaveWutti95`	WW95	Wuttisuttikulkij & O'Mahony (W&O) [298]
`WaveWauDijk`	WADI	Wauters & Demeester (W&D1) [306]
`WaveWauHRWA`	WAHR	Wauters & Demeester HRWA (W&D2) [306]
`WaveNagVWP`	WNVW	Nagatsu *et al.* VWP [296]
`WaveNagWP`	WNWP	Nagatsu *et al.* WP (NHS) [296]

Table 4.2: `Routers` for wavelength allocation

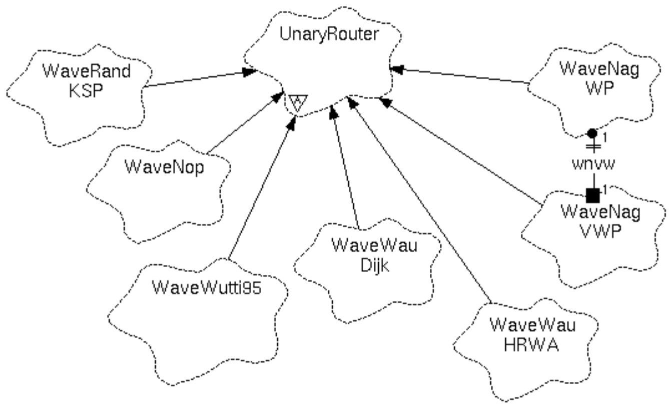


Figure 4.6: Wavelength-allocation routers class diagram

Class	Name	Description
`WaveKSPMutate`	`WKM`	Wavelength-allocation k-shortest path mutation
`WaveWauReroute`	`WWRR`	Wavelength-allocation path reroute operator
`WaveWauShiftOut`	`WWSO`	Wavelength-allocation path shift-out operator
`WaveKSPCO`	`WKCO`	Wavelength-allocation k-shortest path single-point crossover

Table 4.3: `Operators` for wavelength allocation

tialisation of WP routing, fibre choice and wavelength allocation for those networks in the initial population of a GA/heuristic hybrid. For each of the wavelength channels required between a node pair, a random path is selected from the k-shortest, and the wavelength channel routed along it on the lowest wavelength available end-to-end, irrespective of fibre choice (*i.e.* on each of the cables along the route, the lowest fibre id is used which has the required wavelength free). This router has been implemented in NOMaD as class `WaveRandKSP`, the parameters for which are given in Table E.12 on p. 249.

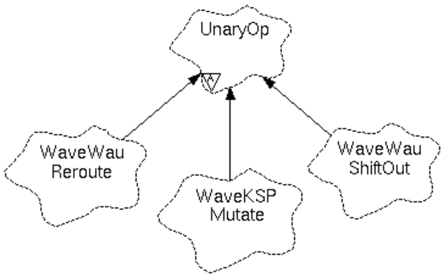
Wavelength-allocation no-op is the simplest of routers: it does nothing at all! It is implemented as class `WaveNop`, and is intended for use as the router data member of a wavelength-routed network object whose routing, fibre and wavelength allocation is to be determined by the external application of a router or GA/heuristic hybrid. If a wavelength-allocation no-op object is not installed in those circumstances, the network would reroute itself (possibly incorrectly) whenever it carried out a self-assessment (see §E.3, p. 253).

The remaining routers in Table 4.2 are all overall wavelength-allocation heuristics published in the literature. Details of these, and their implementation in NOMaD, can be found in Appendix C.

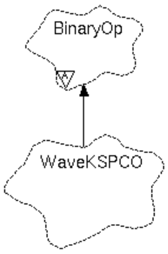
As well as the routers, described above, the author has developed four operators for WP routing, fibre choice and wavelength allocation, three unary and one binary. These are listed in Table 4.3, and the relationships between the classes illustrated in Fig. 4.7. All four were first published by the author in *Electronic Letters* [35] (reprinted in Appendix A).

For a random requirement (*i.e.* an individual wavelength channel of those required between one of the node pairs in the network), *wavelength-allocation k-shortest path mutation* reroutes the wavelength channel on a randomly chosen k-shortest path (where k is an operator parameter), using the lowest wavelength available end-to-end on that path (irrespective of fibre choice). The operator has been implemented as class `WaveKSPMutate`, the parameters of which are given in Table E.13 on p. 249.

The *wavelength-allocation path reroute operator* locates the first wavelength channel in the network which has the highest wavelength number (found by searching the cables in order of their



(a) Unary



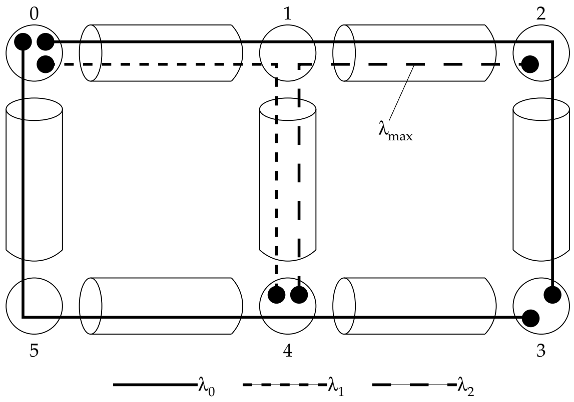
(b) Binary

Figure 4.7: Wavelength-allocation operators class diagrams

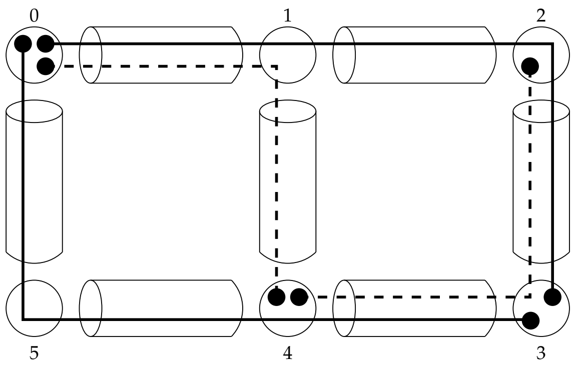
end-node ids), and attempts to reroute it on a lower wavelength number (and the lowest available end-to-end) using one of the k-shortest paths (where k is an operator parameter, and the shortest path is selected if there is more than one at the lowest available wavelength number). It is based on Step 2a in Wauters and Demeester's HRWA algorithm [306] (Appendix C). As an example, Fig. 4.8(a) shows a 6-node, 7-link network, with single-fibre cables, over which four WP wavelength channels have been routed. The wavelength channel using the highest wavelength number (λ_2) is labelled $\lambda_{\max}$. In Fig. 4.8(b), after applying the reroute operator, that channel has been successfully rerouted on a lower wavelength, λ_1. As a result, the network wavelength requirement has been reduced from three to two wavelengths. Within NOMaD, the operator has been implemented as class `WaveWauReroute`, the parameters of which are given in Table E.14 on p. 249.

The *wavelength-allocation path shift-out operator* locates the first wavelength channel in the network which has the highest wavelength number, and attempts to shift it to a lower wavelength number by shifting out (rerouting) all the paths blocking that particular wavelength number end-to-end. The wavelength chosen is the one that results in the smallest increase in overall path length, given the constraint that all rerouting must be to lower wavelength numbers than the original wavelength of the channel being shifted down. It is based on Step 2b in HRWA [306] (Appendix C). Illustrating this, Fig. 4.8(c) shows the original network of Fig. 4.8(a) after shift-out has been applied. By rerouting the blocking wavelength channel on λ_1, it has proved possible to reduce the wavelength number of channel $\lambda_{\max}$ itself to λ_1. This change has again lowered the network wavelength requirement from three to two. Within NOMaD, the operator has been implemented as class `WaveWauShiftOut`, the parameters of which are given in Table E.15 on p. 249.

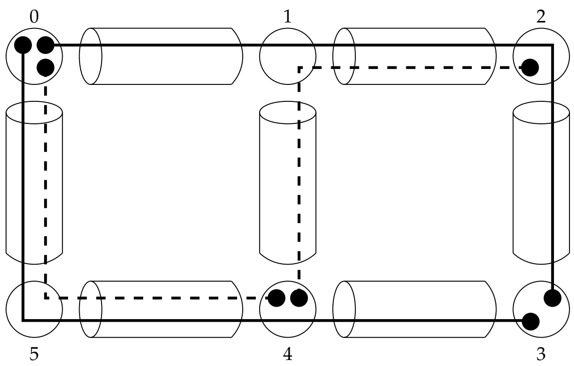
Wavelength-allocation k-shortest path single-point crossover operates on two networks. The first is modified such that up to a randomly selected individual requirement (considered in order



(a) Original network



(b) Reroute



(c) Shift-out

Figure 4.8: Example of reroute and shift-out operators

of their end-node ids), all the wavelength channels remain unchanged, but from then on the paths used are those in the second network, except that the lowest wavelength actually available end-to-end in the first network is used in each case. (However, if any requirement was unsatisfied in the second network, then a random k-shortest path is used instead.) The second network undergoes the complementary recombination. The operator has been implemented as class `WaveKSPCO`, the parameters of which are given in Table E.16 on p. 249.

4.2.4 Operator-probability Adaptation

As a variety of operators are incorporated in NOMaD's different GA/heuristic hybrid algorithms, it might seem necessary to determine appropriate operator probabilities for each of the distinct combinations of operators used. However, this implies a large computational effort. Instead, the probabilities of individual operators being applied to an individual adapt during the course of a run using a credit-assignment algorithm, rather than being fixed throughout, as was mentioned in §3.5 above.

As well as avoiding the need to determine appropriate fixed operator probabilities (except for the initial probabilities, which could be selected from a few trial runs), it was also hoped that there might be performance gains in allowing the probabilities to vary during the course of a run, such that the most appropriate blend of operators was used at each stage, as determined by the observed operator productivity.

The operator-probability adaptation algorithm adopted is based on that of Davis [9], and although mentioned in three papers by the author [34–36], was given its first detailed treatment at the IEEE Intl. Conf. on Evolutionary Computation (ICEC'98) [33] (all four reprinted in Appendix A). First, the basic description of the algorithm, as published in [33], is given, and then a number of further variants presented.

Basic Adaptation Algorithm

Whenever an individual is created whose fitness exceeds that of the best individual found up to the end of the previous generation, the improvement in fitness is credited to the operator responsible:

$$c'_u = c_u + (f_b - f_i), \quad f_i < f_b \tag{4.14}$$

where c_u' and c_u are the current and previous values of the accumulated credit of operator u, f_b is the best fitness found at the end of the previous generation, and f_i is the fitness of individual i. In addition, as has already been mentioned, a decreasing fraction of the credit (say, $\kappa = 0.5$ and $\kappa^2 = 0.25$) is awarded to the parent operator (*i.e.* the operator that created this individual's parent) and grandparent operator, to ensure that two- and three-operator sequences are rewarded appropriately. Thus:

$$c_p' = c_p + (f_b - f_i) \cdot \kappa, \quad f_i < f_b \tag{4.15}$$

$$c_g' = c_g + (f_b - f_i) \cdot \kappa^2, \quad f_i < f_b \tag{4.16}$$

where c_p and c_g are the accumulated credit of the parent (p) and grandparent (g) operators, respectively[6], and κ is the credit constant ($0 \leq \kappa \leq 1$).

It should be noted that, while Davis [9] maintained a complete tree (although of finite depth) of the operators responsible for the creation of an individual, here only a single (arbitrary) parent is recorded for binary operators. This greatly simplifies accounting, as instead of up to two parents, four grandparents, eight great-grandparents, *etc.*, only a single parent and grandparent operator have to be recorded. Indeed, the overhead of maintaining and crediting a complete operator tree has already been noted elsewhere [265]. Further, although the selection of a single arbitrary parent operator means that all the operators responsible for a superior individual are not rewarded, it would be expected that these unrewarded operators would have subsequent opportunities to acquire credit, and it is their relative accumulation of credit, rather than the absolute value, that is important.

After a few generations (say, $\epsilon = 4$), a small proportion (say, $\psi = 0.15$) of the operator probabilities is reassigned according to the average credit earned per child each operator has generated. In addition, operators are given a minimum probability (say, $p_{\min} = 0.05$ for less than 20 operators) to ensure they do not lose the opportunity to gain some credit if they, having decayed to the minimum level, are later found to be useful to further evolve the population. Thus, every ϵ generations, for all operators, first:

$$a_u = \begin{cases} c_u/o_u, & o_u > 0 \\ 0, & o_u = 0 \end{cases} \tag{4.17}$$

where, for operator u, a_u is the adaptation weight and o_u the number of children (*i.e.* offspring)

[6] The three operators, u, p and g are not necessarily different: the same operator may have been applied to a single individual two or three times in sequence.

generated. The total adaptation weight is then given by:

$$a = \sum_u a_u \tag{4.18}$$

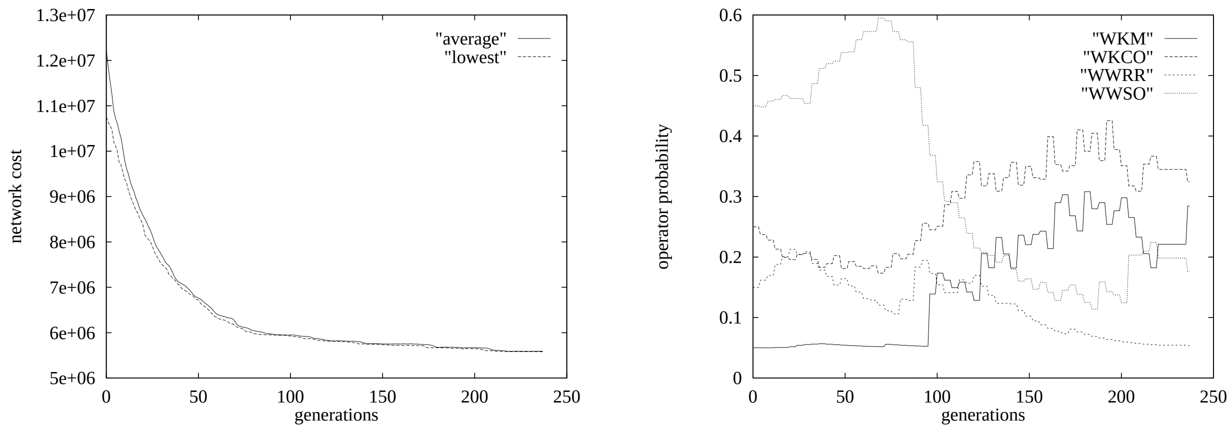
Now, provided some credit has been earned, and thus a is non-zero, probability reassignment can proceed. However, before this an adaptation scaling factor (K_a) must be calculated. This is the amount of the total probability that is to be reassigned, whilst still protecting the minimum probability level of all operators, rescaled using the total adaptation weight, thus:

$$K_a = \psi \cdot (1 - (s+1)p_{\min})/a, \quad a > 0, \quad (s+1)p_{\min} < 1 \tag{4.19}$$

where ψ is the adaptation proportion ($0 \leq \psi \leq 1$), s is the number of operators (excluding reproduction, hence the addition of 1) and $p_{\min}$ is the minimum operator probability level. All the operator probabilities can then be then reassigned as follows:

$$p'_u = (p_u - p_{\min}) \cdot (1 - \psi) + a_u K_a + p_{\min} \tag{4.20}$$

where p_u is the probability of operator u. The first term represents the unchanged part of the operator's probability, the second, that element reassigned according to adaptation weight, and the final one, the guaranteed minimum probability. It is straightforward to verify that, after the application of Eqn. 4.20, the $\sum_u p_u$ remains 1. Finally, before continuing, the accumulated credit (c_u), number of children (o_u) and number of trials (t_u, see below) of all operators are reset to zero.



(a) Progress over a single run

(b) Operator-probability adaptation

Figure 4.9: Example GA/heuristic hybrid run

As an example of the operator-probability adaptation mechanism, Fig. 4.9(a) illustrates the progress of a typical wavelength-allocation hybrid run, showing the lowest and average population cost against generation (Network Cost Model B). Fig. 4.9(b) gives the operator probabilities for the same run, where the effect of the adaptation algorithm can be clearly seen[7]. For example, wavelength-allocation k-shortest path mutation (WKM), which started with a low probability, became much more productive after generation 100; whereas the wavelength-allocation path shift-out operator (WWSO), which had the highest probability in early generations, was found to be much less useful towards the end of the run.

Variants on the Basic Algorithm

Variations to both the basic credit-assignment and operator-probability adaptation mechanisms have also been developed. First, in the basic credit-assignment algorithm, operators are rewarded as a result of improvement over the fitness of the best individual found up to the end of the previous generation. Instead, credit can be awarded for improvement over:

- best-in-window fitness (*i.e.* the best fitness achieved in the previous w generations), f_w [265, 271];
- the median fitness of the population at the end of the previous generation, f_m [270];
- or the fitness of the parent individual (or the fitter of the two parents, for binary operators), f_p [265, 271].

Thus, Eqn. 4.14 (and in a similar way, Eqn. 4.15 and 4.16) is altered to one of:

$$c'_u = c_u + (f_w - f_i), \quad f_i < f_w \tag{4.21}$$

$$c'_u = c_u + (f_m - f_i), \quad f_i < f_m \tag{4.22}$$

$$c'_u = c_u + (f_p - f_i), \quad f_i < f_p \tag{4.23}$$

where f_w is the best-in-window fitness at the end of the previous generation, likewise, f_m is the median fitness found at the end of the previous generation, and f_p is the (better) fitness of the parent(s) of individual i.

For the operator-probability algorithm, in the basic approach, operator credit was divided by the number of children each operator had generated before being used to reassign operator probabilities.

[7]The reproduction operator has been omitted for clarity.

However, if an operator uses more than one trial per child generated, this may unfairly favour operators that are expensive in their use of trials (*e.g.* `RandomChgLink`, `RandomChgLinks`, *etc.*). An alternative is to use the number of trials itself as the divisor before probability reassignment. Thus, Eqn. 4.17 is altered to:

$$a_u = \begin{cases} c_u/t_u, & t_u > 0 \\ 0, & t_u = 0 \end{cases} \tag{4.24}$$

where t_u is the number of trials used by operator u. However, this change also has an interesting secondary consequence, as not all operator applications require trial(s), due to assessment being cached by the network (§E.3.1 and §4.2.1): if the network is unchanged, the operator has been applied 'for free'.

4.3 Summary

This chapter has provided details of the design metrics used by the author for network assessment: one for minimum-cost mesh network topology design, another for network wavelength requirement, and the final one for minimum-cost wavelength allocation. Moreover, the evolutionary algorithm framework has been described, including the overall structure of the GA/heuristic hybrid; operators for network topology design; operators for routing, fibre and wavelength allocation; and the operator-probability adaptation mechanism. In the next chapter, experimental results will be reported exploring the suitability of the GA/heuristic hybrid approach for multi-wavelength all-optical transport network design.