Chapter 3

NOMaD

By the roadside you sat waiting for lovers,
sat like a nomad in the desert.
Jeremiah 3:2b, NIV

This chapter provides an overview of a software toolset for optical network optimisation, modelling and design (NOMaD) [31, 32] developed by the author. The toolset has been used by the author to support research into the application of genetic-algorithm (GA)/heuristic hybrid optimisation techniques [9] to optical network design, the topic of this thesis, as well as by others[1] for heuristic-based optical network design in three research projects. These included two under the European Commission funded research programme in Advanced Communications Technologies and Services (ACTS) [272, 273]: WOTAN (Wavelength-agile Optical Transport and Access Network) [125] and OPEN (Optical Pan-European Network) [18, 274]; and the Fujitsu Telecommunications Europe Ltd. (FTEL) "Future Broadband Networks" project [20–24].

Much of the material presented in this chapter was initially published by the author at the 12th UK Performance Engineering Workshop [31], when NOMaD was in the early stages of development, and then more recently at an IEE Colloquium on Multiwavelength Optical Networks: Devices, Systems and Network Implementations [32] (both reprinted in Appendix A).

The chapter is divided into ten sections: §3.1–3.3 describe in turn NOMaD's application areas, design metrics and solution approaches. Then §3.4 presents the overall architecture of the software. The next three sections, §3.5–3.7 describe different layers of that architecture: the object-oriented network toolset, the scripting layer and the graphical user interface (GUI). Next, in §3.8, the software testing & verification approach is reviewed. Then section (§3.9) gives a brief overview of

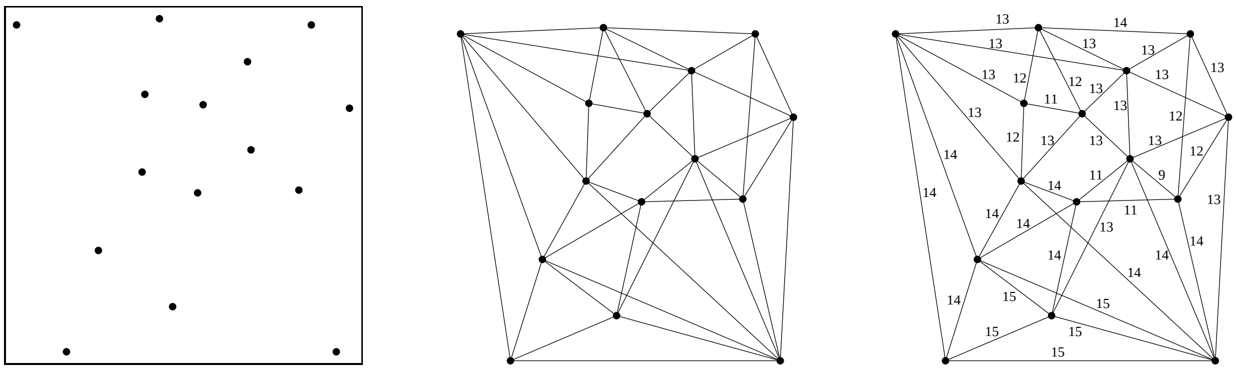
[1] Katerina Proestaki for the WOTAN and Fujitsu projects; Noel Parnis for OPEN

the main application program used to obtain the results reported in Ch. 5. Finally, the chapter is summarised in §3.10.

3.1 Application Areas

The design problems that NOMaD addresses are primarily those presented by optical transport networks over national or international distances. NOMaD's network model allows for a wide-range of network topologies, such as regular (*e.g.* torus, hyper-cube, *etc.*), arbitrary-mesh and (multiple) ring networks, as well as consideration of both 'flat' and hierarchical approaches. In addition, the network model can represent the cable, fibre and wavelength-routed structure of multi-wavelength all-optical transport networks. Two key application areas are directly supported: (i) network topology design, and (ii) routing, fibre choice and wavelength allocation.

Network topology design determines which node pairs are to be directly connected by a link (cable), given the node locations and static traffic requirements. However, while the network model supports both primary and restoration paths when used for topology design, fibre choice and wavelength allocation are not included. As an example, Fig. 3.1(a) shows a 15-node network design problem (this is actually Problem 1 from §F.2), for which an example flat mesh network topology design has been developed, given in Fig. 3.1(b).



(a) Network design problem (b) Mesh topology (c) Wavelength allocation

Figure 3.1: Example network design problem, mesh topology & wavelength allocation

Starting from a network topology, the second application area is routing, fibre choice and wavelength allocation. The static traffic requirement between each node pair in the network has to be partitioned into an appropriate number of separate wavelength channels. For each of these, a route through the network must be found, including which fibre and wavelength will be used on each link (cable) along the path. However, although restoration paths are supported during topology design, the network model provides no direct support for the fibre and wavelength allocation of restoration paths. An example wavelength allocation, for the network topology already developed, is illustrated in Fig. 3.1(c), where (assuming just a single fibre for each cable) the number of wavelengths used has been drawn next to each network link.

A more complete review of NOMaD's network model can be found in §E.3, and detailed descriptions of the specific versions of these design problems addressed by experimental work are given in §5.1 and §5.2 below.

3.2 Design Metrics

As a vital part of any network design process, it is necessary to define one or more objectives, to allow prospective designs to be compared, and the best selected. While it is possible to carry out a design with multiple objectives simultaneously [275–277], in NOMaD a single overall design metric is assumed. This approach is common to much evolutionary-algorithm (EA) research, as it allows a single 'fitness' value to be determined for each individual in the population. As a result, however, if multiple objectives must be considered, they will have to be combined as a weighted sum. In addition, any design constraints may be enforced either by repairing individual network designs that violate them [278], or adding a penalty [279,280] to the design metric for each violation (usually also dependent on the *extent* of the violation).

Within NOMaD, the author has developed three metrics, one for topology design and two for routing, fibre and wavelength allocation, and these are presented in §4.1 below. In addition, it is straightforward for users to incorporate their own individual metrics for network design assessment into the software.

3.3 Solution Approaches

Two main solution approaches to network design problems have been anticipated within NOMaD: overall design heuristics, and EAs.

Problem-specific, constructional heuristic algorithms employing the network model can be readily built by users. Moreover, while the author did not identify any suitable overall heuristics for minimum-cost topology design of optical transport networks, several routing, fibre and wavelength allocation heuristics from the literature were implemented, and these are described in Appendix C (additional algorithms have also been identified in §6.4).

Extensive support for EAs in the form of GA/heuristic hybrids is provided, with both an overall adaptive GA framework, and also the straightforward ability to add heuristic operators. This applies whether the latter are inspired by traditional genetic operators, based on elements from overall design heuristics, or developed specifically for a particular hybrid. Several heuristic operators for both network topology design, and routing, fibre and wavelength allocation have been devised by the author, and these are described in §4.2, as are the overall GA framework and NOMaD's operator-probability adaptation mechanism.

In addition to these two approaches, however, only a modest amount of effort would be required to further develop NOMaD to support other modern heuristic search methods, such as tabu search [6, 7] or simulated annealing [5] (*cf.* also Bhandarkar & Zhang [262]). However, as was explained in Ch. 1, these fall outside the scope of the thesis. More distant methods, such as (integer) linear programming [25], would not however be suited to NOMaD's object-oriented network representation.

3.4 Overall Architecture

NOMaD has been developed as an overall architecture to be extended by several people in a variety of directions to serve the needs of their individual projects [31, 32]. Consequently, its users have not just been given a single unchangeable piece of software, but rather a framework which they can actively add to for themselves. As a result, NOMaD has had to be easy to understand, flexible and extensible. The overall layered architecture is shown in Fig. 3.2.

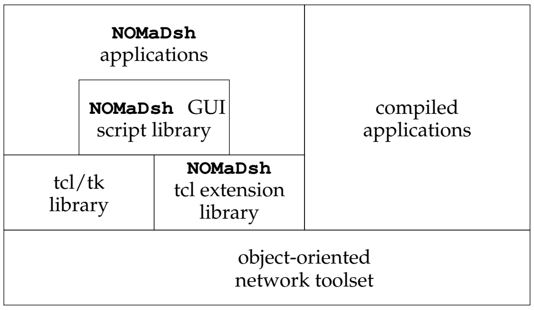


Figure 3.2: Overall NOMaD architecture

To aid both understanding and extensibility, the toolset has been developed using object technology. The analysis and design were documented using Booch Notation [15], and the implementation done in the C++ programming language [16, 17, 281, 282]. This process has been aided by the use of a CASE tool, Rational Rose/C++ [283], which generates skeletal C++ code directly from the diagrams and specifications created by the author.

Although the majority of the experimental work undertaken by the author and others using NOMaD has employed executables compiled directly from the object-oriented network toolset, for flexibility, rather than only providing a C++ interface, a NOMaD-specific tcl extension has also been developed (termed `NOMaDsh`). Tcl [284, 285] is a scripting language designed to be extended by the incorporation of application-specific commands, coded in C or C++. Consequently, `NOMaDsh` can be used either interactively to access the underlying network toolset—creating networks, modifying them, assessing their cost/performance, *etc.*—or via scripts written using a combination of tcl and the NOMaD-specific commands. It was hoped that this latter option would enable new high-level algorithms to be developed with much faster edit-compile-debug cycle times, because no compile time would be needed—the scripts could be immediately run by the `NOMaDsh` interpreter. Only if the user needed to extend the underlying C++ network toolset would the `NOMaDsh` itself need to be recompiled. It was envisaged that these benefits would be particularly important in a multi-person research environment, where the rapid prototyping of ideas is of paramount importance. In addition, a library of graphical user interface (GUI) scripts was developed using the tk toolkit (tcl/tk) to allow `NOMaDsh` scripts to display networks.

3.5 Object-oriented Network Toolset

The top-level structure of NOMaD's object-oriented network toolset is partitioned into five class categories, as shown in Fig. 3.3. Although included from an early design stage, the `NOMaD` class category contains no classes, but instead represents a 'place holder' for executables compiled directly from the network toolset, such as `ga` (see §3.9) and `NOMaDsh` itself. The remaining four categories are `NetworkModel`, `GA`, `Operators` and `Utilities`.

The `NetworkModel` class category contains classes used to represent networks for topology design as well as routing, fibre and wavelength allocation, plus metrics to allow the assessment of designs and several minor classes. For example, the main `Network` class diagram, given in Fig. 3.4(a), shows the `Network` class, composed of `Node`s, `Link`s, and `PathLossSeq`s (path-loss sequences, *i.e.* ordered sequences of `Path`s [286]), plus several implementation classes used to represent the network's adjacency matrix [25, 287], connection matrix (which records link levels in hierarchical networks) and the static traffic requirements. Further, Fig. 3.4(b) illustrates the wavelength-routed network

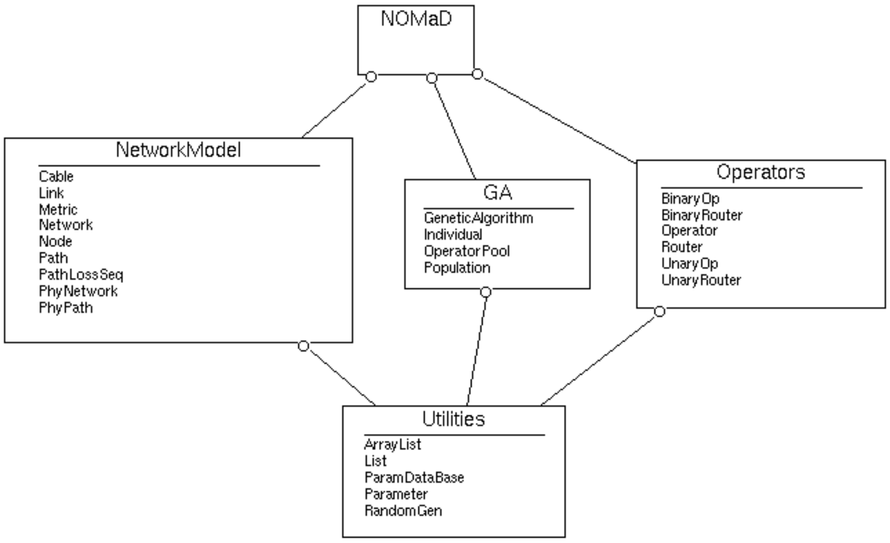
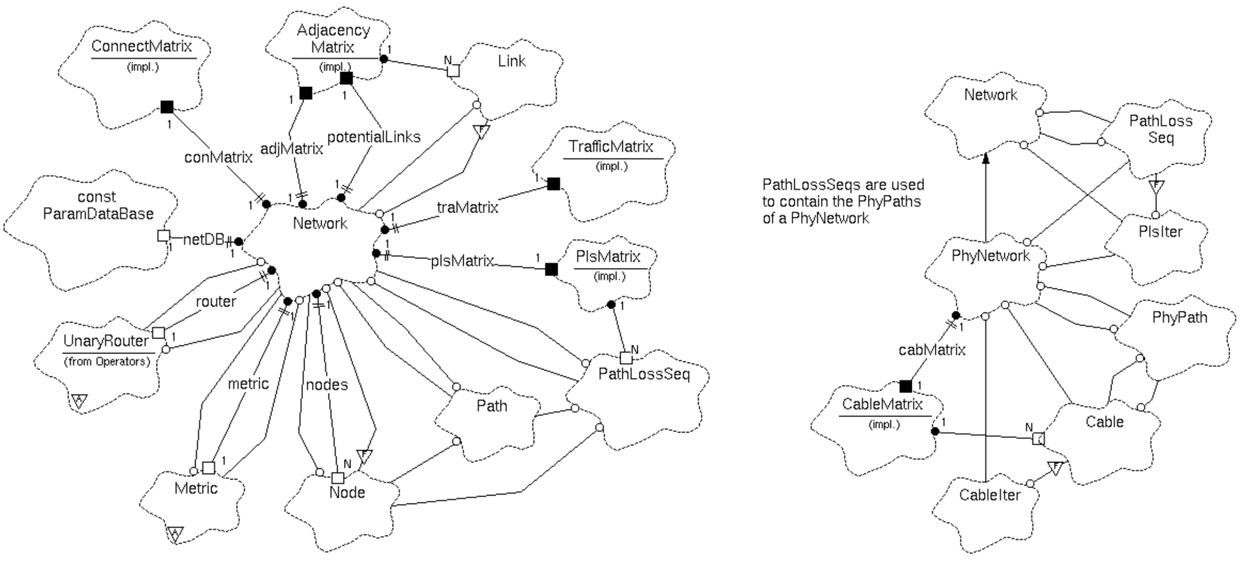


Figure 3.3: NOMaD class categories

class (`PhyNetwork`, derived from `Network`), which uses wavelength-routed paths (`PhyPath`s, derived from `Path`) and a matrix of `Cable`s to represent the cable, fibre and wavelength-routed structure of a multi-wavelength optical network.

The classes that implement NOMaD's GA framework are grouped into the `GA` class category, and illustrated in the main `GA` class diagram in Fig. 3.5—the `GeneticAlgorithm` class, with its `Population` of `Individual`s and its `OperatorPool` of `Operator`s. An `Individual` simply consists of a single `Network` (or `PhyNetwork`) object and some additional accounting information. Operators themselves are created by deriving classes from either unary operator (`UnaryOp`) or binary operator (`BinaryOp`) classes (not shown). In general, those derived from `UnaryOp` are analogous to mutation [2] in a traditional GA, but can be powerful heuristics that utilise arbitrary amounts of problem-specific information. Likewise, `BinaryOp` subclasses are analogous to traditional crossover [2], although once again, there is considerable potential for problem-specific heuristics.

As in Davis [9], the probabilities of individual operators being applied to an `Individual` adapt through the course of a run using a credit-assignment algorithm, rather than being chosen with a fixed probability throughout [33]. Whenever an individual, whose fitness exceeds that of the best individual up to the end of the previous generation (`bestIdv`), is created, the improvement in fitness is credited to the operator responsible. In addition, a decreasing fraction of the credit is awarded to the parent operator (*i.e.* the operator that created this individual's parent, `parentOp`) and grandparent operator (`grandParOp`), to ensure that two- and three-operator sequences are rewarded appropriately. After a few generations, a small proportion of the operator probabilities is



(a) `Network` (b) `PhyNetwork`

Figure 3.4: Network class diagrams

reassigned according to the credit earned.

As well as including the abstract classes representing operators (`Operator`, `UnaryOp` and `BinaryOp`) and routers (`Router`, `UnaryRouter` and `BinaryRouter`), the `Operators` class category includes a large number of concrete classes derived from these. The operators range from those of modest effect, analogous to traditional mutation or crossover, intended for use within a GA framework, through more powerful problem-specific operators, up to complete design heuristics, intended for use in isolation. Routers are those operators capable of completely routing a network, rather than merely modifying an existing routing (*i.e.* either simple routing, for topology design, or complete routing, fibre and wavelength allocation as appropriate).

The final class category, `Utilities`, contains a number of comparatively unrelated classes used by the other categories. These include dynamic arrays, linked lists, file handling and pseudo-random number generation.

3.6 `NOMaDsh`: The Tcl Scripting Layer

Tcl [284, 285] is a scripting language that allows for application-specific extension in a variety of ways. The approach adopted in creating `NOMaDsh` is an object-oriented rather than action-oriented one [284,285], with the user able to create objects that directly correspond to the underlying network toolset implemented in C++. Once an object is created, a new command is added to the interpreter

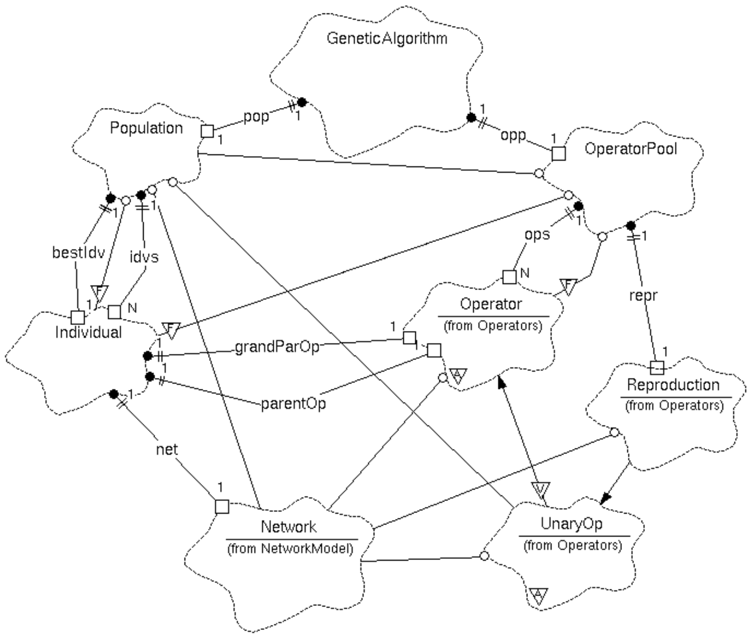


Figure 3.5: `GA` class diagram

that allows the user to send commands to that particular object, equivalent to invoking its member functions in C++. For example, a `Network` object, created in C++ by:

```
ifstream netifs("central.net");
ifstream nmdifs("central.nmd");
ParamDataBase* ndb = new ParamDataBase(netifs);
ParamDataBase* pdb = new ParamDataBase(nmdifs);
Network*       net = new Network(ndb, pdb);
```

could have been created in `NOMaDsh` with the following commands, where the NOMaD-specific extensions can be identified by the `nomad_` prefix:

```
set ndb [nomad_paramdatabase "central.net"]
set pdb [nomad_paramdatabase "central.nmd"]
set net [nomad_network $ndb $pdb]
```

In both the C++ and `NOMaDsh` commands, the two input files, one describing the network itself (`"central.net"`) and the other certain necessary parameters (`"central.nmd"`), are used to construct [16] two parameter databases (`ParamDataBase` objects `ndb` and `pdb`, respectively). It is from these that the `Network` itself is then constructed. Parameter databases are simply random access storage objects, which free other NOMaD classes from the complexities of direct file handling. Once it has been constructed, the `Network` object can then be queried for the number of nodes it contains, either in C++ as below:

```
int numNodes = net->getNumNodes();
```

or in `NOMaDsh`, by using the new command for the underlying `Network` object, whose name was stored in the `net` variable:

```
set numnodes [$net numnodes]
```

It was hoped that this semantic similarity (though partially masked by syntax!), would result in a short learning curve for users familiar with both C++ and tcl/tk. Consequently, the full benefits of the interactive and interpretive style of `NOMaDsh` scripts would be swiftly realised.

As an aid to the reader in understanding the relationship between a `NOMaDsh` script and the underlying C++, the code in `NOMaDsh` that allows access to a compiled `Network` object and the invocation of its `getNumNodes()` member function is shown below. For clarity, both error-handling code and the large number of alternatives to the `numnodes` command argument have been removed.

```
int Nomad_NetworkCmd(ClientData clientData,
          Tcl_Interp *interp, int argc, char *argv[]) {
  ostrstream rst(interp->result, TCL_RESULT_SIZE);
  Network* net = (Network*) clientData;
  if (strcmp(argv[1], "numnodes") == 0) {
    rst << net->getNumNodes() << '\0';
  }
  return TCL_OK;
}
```

First, an output string stream (`ostrstream`) is constructed around the interpreter's return buffer (`interp->result`). Then, a pointer to the underlying `Network` object is retrieved from the `clientData` parameter. If the first argument passed to the `$net` command was `numnodes`, then the `getNumNodes()` member function is invoked and the result placed in the return buffer. Finally, the function signals success by returning `TCL_OK`.

3.7 Graphical User Interface

The final element of the NOMaD architecture is the GUI. Although it was originally envisaged that a large library of GUI scripts would be built [31], including widgets for the display and editing of networks, the actual achievement in this area was relatively modest. Scripts were written for the display of network topologies (`netdisplay.tcl`), graph drawing (`graphdraw.tcl`) and the output of network topologies in `xfig` v2.1 format [288] (`netxfig.tcl`). From these, a genetic-algorithm application and a network-display application were developed, although both are suitable only for network topology design (as opposed to routing, fibre and wavelength allocation). In addition, a prototype network editor was implemented, but this was simply written in tcl/tk as a feasibility study, without employing the NOMaD-specific extensions to access the underlying object-oriented network toolset.

Screen shots of the genetic-algorithm (`ga.tcl`), network-display (`netdisp.tcl`) and prototype network-editor (`network.tcl`) `NOMaDsh` applications are given in Figures 3.6–3.8.

The first of these (Fig. 3.6) shows the genetic-algorithm application's control panel, where file names are entered and the GA run started, two graph windows, and a series of best individuals found at the end of successive generations. The network design problem tackled here was the minimum-cost topology design of the nine central nodes of the COST 239 European Optical Network (EON) [166], with the result found at the end of generation 14 matching the best known. The two graph windows have recorded the lowest & average network costs (`Network Assess` window) and the operator probabilities (`Operator Prob` window) changing as execution proceeded. In the latter window, the varying probabilities of the three operators actually used in this run *i.e.* link-group crossover (LGCO), single-link mutation (ML) and reproduction (REPR) can be seen; for a full description of these, see §4.2 below.

The second screen shot (Fig. 3.7) has captured two windows of the network display application: the control panel, where a succession of file names may be entered for display, and a network topology window, once again showing the nine central nodes of the EON. As with `ga.tcl`, the network topology window allows the network design to be saved to disk (using a member function of the underlying `Network` class) or an `xfig` format diagram generated and saved.

Finally, the NOMaD prototype network editor is shown in Fig. 3.8, with an 8-node network displayed. The vertical row of icons on the left-hand side switch the editor into the following modes (starting from the top): query, add-node, add-link, move-node (and attached links), traffic-display and delete. The status line along the bottom of the window prompts the user for the next expected action, and provides appropriate feedback during operations; in Fig. 3.8, it can be seen displaying the initial add-link prompt.

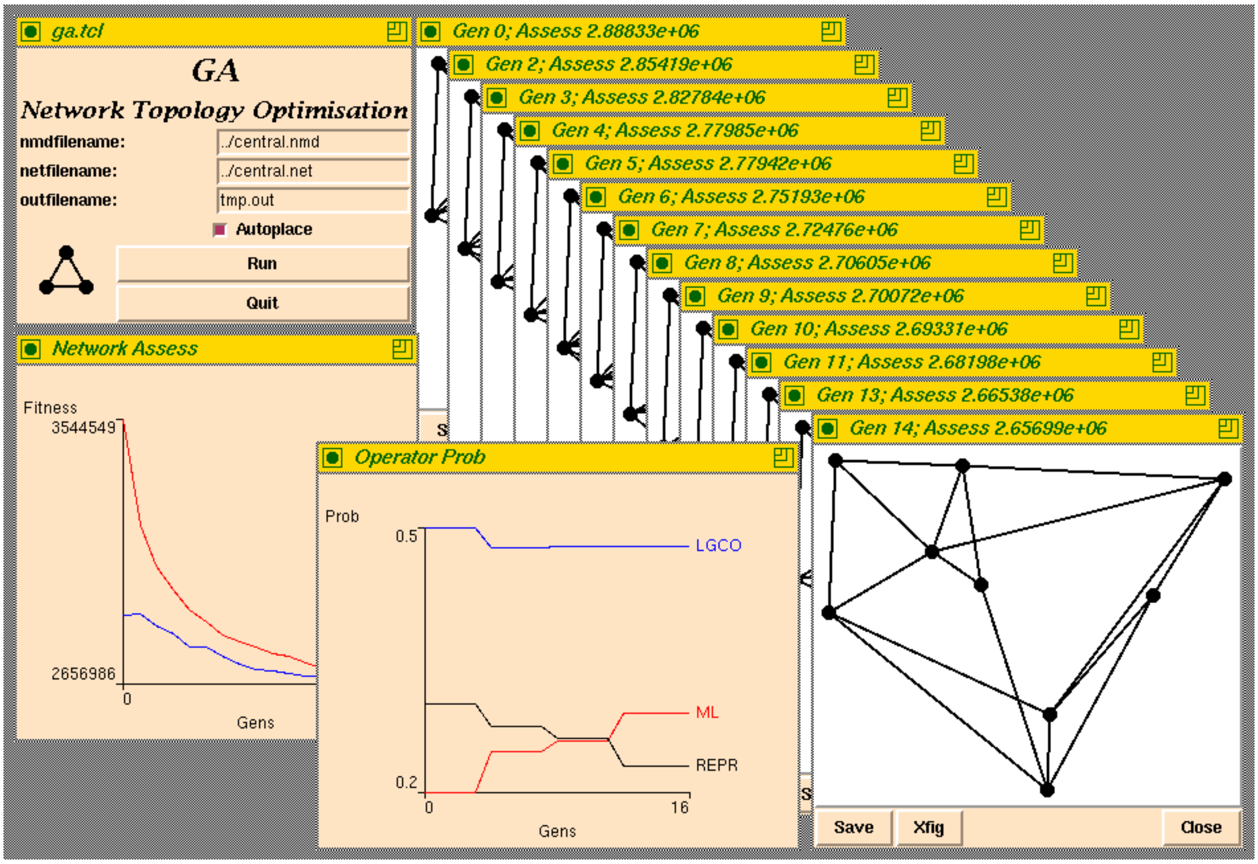


Figure 3.6: Genetic-algorithm (`ga.tcl`) application

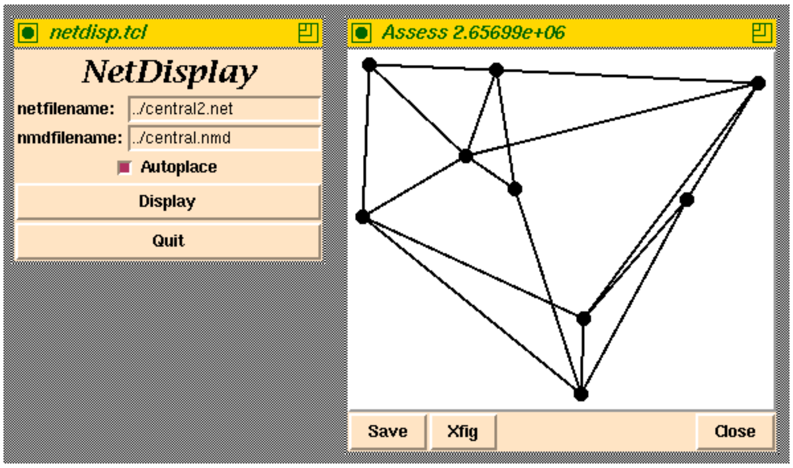


Figure 3.7: Network-display (`netdisp.tcl`) application

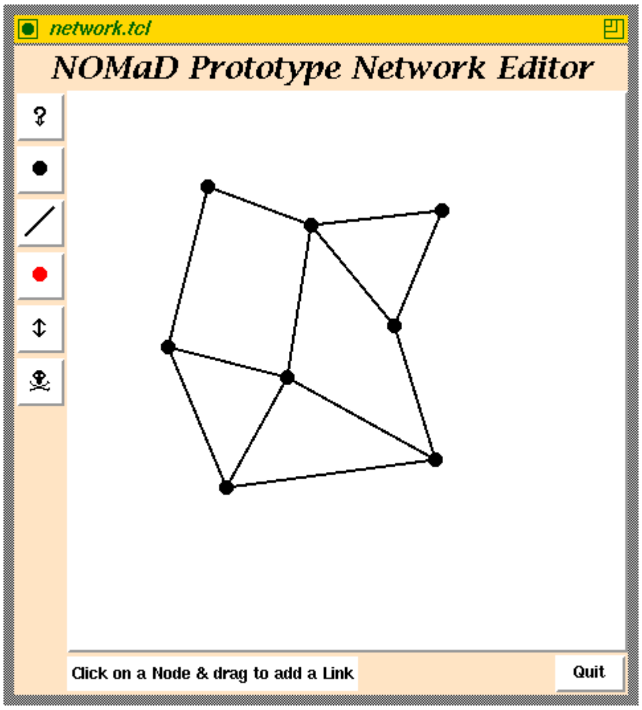


Figure 3.8: Network-editor (`network.tcl`) application

3.8 Testing & Verification

During the design and implementation of NOMaD's object-oriented network toolset, three different approaches were used to try and minimise errors in the software: Programming by Contract, dynamic memory tracking and regression testing.

Meyer's concept of Programming by Contract [289] introduces pre- and postconditions on class member functions: preconditions define what a class requires to do its job; postconditions, what it will provide if its preconditions have been met. In the object-oriented network toolset, explicitly stating these conditions in this way led to clarity in class design, a benefit which was further enhanced by run-time checking using appropriate assertions.

Now, as C++ requires the programmer to directly manage the allocation and release of all dynamic memory objects [16,282], this can be a significant source of error. In addition, the dynamic allocation of objects that are not subsequently released when no longer required ('memory bleed'), can lead to program failure due to exhaustion of machine memory. To combat problems in this area, dynamic memory tracking code (called `xmem`) for C [290] was rewritten by the author for C++. This allowed the detection of a variety of different problems during NOMaD's development.

The final approach to minimising software errors adopted was regression testing. A large test program (`testNOMaD`) was written, with a long series of assertions for many of the classes in the object-oriented network toolset. These checked the expected outcome of invoking the class member functions under a variety of different conditions. By running, and extending, the tests periodically throughout the development of NOMaD, several subtle errors were detected and eliminated.

Further details of all three elements of NOMaD's testing and verification strategy can be found in §E.7.

3.9 The `ga` Application Program

Although several application programs were developed by the author using NOMaD's object-oriented network toolset, the most important of these is `ga`, which has therefore been selected for description here. Others include, for example, `wavealloc`, used to apply complete design heuristics for routing, fibre and wavelength allocation (such as those described in Appendix C) to individual network topologies. The `ga` application program is partially equivalent to the `ga.tcl` `NOMaDsh` application mentioned in §3.7 above, but allows access to the full range of GA/heuristic hybrid techniques in NOMaD's object-oriented network toolset. Consequently, the majority of the experimental results reported in Ch. 5 were obtained using `ga`.

The `ga` program is invoked with four arguments: the name of the parameter file (`.nmd`), the name of a file containing a description of the network design problem (or network topology, as appropriate; `.net`), a filename to be used to output the results of the run (`.out`), and an initial value for the pseudo-random number generator.

A high-level flowchart for the `ga` application program is shown in Fig. 3.9. After checking and storing the command-line arguments, the major program objects are constructed. These consist of three parameter databases, two built from the input files and one to record data (in variable-value pairs *i.e.* parameter form) that will be output to file at the end of program; a pseudo-random number generator; and a genetic-algorithm object. Initial statistics are then recorded in the output parameter database, including the average and lowest fitness of the initial population, as well as the number and names of the operators, and their initial probabilities. The program then enters its main loop, which is controlled by a Boolean variable which, after initially being set `true`, is assigned the value returned by the genetic-algorithm object's `run()` member function. The latter is invoked to evolve the population for one generation, with the return value indicating whether further evolution is required. Similar statistics to those recorded for the initial population are then added to the output parameter database for this generation. Finally, at the end of program execution, both the best network design found and the contents of the output parameter database are written to file.

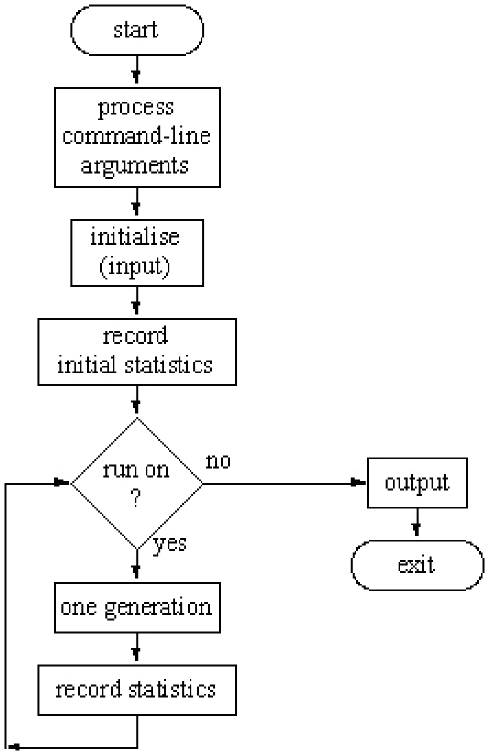


Figure 3.9: Flowchart of the `ga` application program

3.10 Summary

This chapter has provided an overview of the NOMaD software toolset, including its application areas, design metrics and supported solution approaches. The overall NOMaD architecture, its object-oriented network toolset, interactive scripting capabilities, graphical user interface, testing & verification strategy and an example compiled application have all been discussed. Building on this overview, the next chapter will provide greater detail of NOMaD's design metrics and EA framework. It is these elements, implemented in the NOMaD software toolset, that encapsulate the author's GA/heuristic hybrid approach to multi-wavelength all-optical transport network design.