Appendix A

Supporting Publications

At submission, the author had published eight papers that directly support the thesis. These are listed below, and copies of the papers themselves have been included in the following pages.

1. Sinclair, M.C., "NOMaD: Initial architecture of an optical network optimisation, modelling and design tool", Proc. 12th UK Performance Engineering Workshop, Edinburgh, UK, September 1996, pp. 157–167

2. Sinclair, M.C., "NOMaD: Applying a genetic-algorithm/heuristic hybrid approach to optical network topology design", Proc. Intl. Conf. on Artificial Neural Networks and Genetic Algorithms (ICANNGA'97), Norwich, UK, April 1997, pp. 299–303

3. Sinclair, M.C., "Minimum network wavelength requirement design using a genetic-algorithm/heuristic hybrid", *Electronics Letters* v34 n4 February 1998 pp. 388–389

4. Sinclair, M.C., "Minimum cost routing and wavelength allocation using a genetic-algorithm/heuristic hybrid approach", Proc. Sixth IEE Conf. on Telecommunications, Edinburgh, UK, March/April 1998, pp. 67–71

5. Sinclair, M.C., "Operator-probability adaptation in a genetic-algorithm/heuristic hybrid for optical network wavelength allocation", Proc. IEEE Intl. Conf. on Evolutionary Computation (ICEC'98), Anchorage, Alaska, USA, May 1998, pp. 840–845

6. Sinclair, M.C., "NOMaD: An optical network optimisation, modelling and design toolset", Proc. IEE Colloquium on Multiwavelength Optical Networks: Devices, Systems and Network Implementations, Digest No. 1998/296, London, UK, June 1998, pp. 6/1–6/6

7. Sinclair, M.C., "Minimum cost wavelength-path routing and wavelength allocation using a genetic-algorithm/heuristic hybrid approach", *IEE Proc. Communications* v146 n1 February 1999 pp. 1–7

8. Sinclair, M.C., "Evolutionary telecommunications: A summary", Proc. GECCO'99 Workshop on Evolutionary Telecommunications: Past, Present and Future, Orlando, Florida, USA, July 1999, pp. 209–212

Appendix B

Personal Publications

A complete list of the author's publications, at submission, is given below (including those directly supporting the thesis, already listed in Appendix A).

1. Sinclair, M.C., "Single-moment analysis of unreliable trunk networks: Two methods for the network grade-of-service", Proc. 8th UK Teletraffic Symposium, Beeston, April 1991, pp. 8/1–8/11

2. Sinclair, M.C., "Trunk network fault-tolerance through routing table updating in the event of trunk failure", Proc. 9th UK Teletraffic Symposium, Guildford, April 1992, pp. 17/1–17/9

3. Sinclair, M.C., "Trunk network fault-tolerance through network augmentation", Proc. 4th Bangor Symposium on Communications, University of Wales — Bangor, May 1992, pp. 252–256

4. Sinclair, M.C., "Single-moment analysis of unreliable trunk networks employing k-shortest-path routing", Proc. IEE Colloquium on Resilience in Optical Networks, Digest No. 1992/189, London, October 1992, pp. 3/1–3/6

5. Sinclair, M.C., "The application of a genetic algorithm to trunk network routing table optimisation", Proc. 10th UK Teletraffic Symposium, Martlesham Heath, April 1993, pp. 2/1–2/6

6. Chng, R.S.K., Sinclair, M.C., Donachie, S.J. & O'Mahony, M.J., "Distributed restoration algorithm for multiple failures in a reconfigurable network", Proc. 5th Bangor Symposium on Communications, University of Wales — Bangor, June 1993, pp. 203–206

7. Sinclair, M.C. & O'Mahony, M.J., "COST239: Initial network design and analysis", Proc. 36th RACE Concertation, Brussels, July 1993, pp. 89–94

8. O'Mahony, M., Sinclair, M.C. & Mikac, B., "Ultra-high capacity optical transmission network: European research project COST 239", *Informacija Telekomunikacije Automati* v12 n1-3 1993 pp. 33–45

9. Chng, R.S.K., Sinclair, M.C., Botham, C.P. & O'Mahony, M.J., "Multi-layer real-time restoration for survivable networks", Proc. IEE Colloquium on Safety Critical Distributed Systems, Digest No. 1993/189, London, October 1993, pp. 8/1–8/5

10. Stevens, R.D., Sinclair, M.C. & Dilworth, I.J., "Finite source modelling of private mobile radio systems", Proc. 11th UK Teletraffic Symposium, Cambridge, March 1994, pp. 12B/1–12B/7

11. Johnson, D., Brown, G.N., Beggs, S.L., Botham, C.P., Hawker, I., Chng, R.S.K., Sinclair, M.C. & O'Mahony, M.J., "Distributed restoration strategies in telecommunications networks", Proc. IEEE Intl. Conf. Commun. (ICC'94), New Orleans, USA, May 1994, pp. 483–488

12. Stevens, R.D., Dilworth, I.J. & Sinclair, M.C., "Fading and its effects on the planning of private mobile radio systems", Proc. 11th National Radio Science Colloquium, Liverpool, July 1994

13. Stevens, R., Dilworth, I. & Sinclair, M., "Fading and its effects on private mobile radio systems", *Philips Telecom Review* v52 n2 August 1994 pp. 24–27

14. Sinclair, M.C., "Improved model for European international telephony traffic", *Electronics Letters* v30 n18 September 1994 pp. 1468–1470

15. Stevens, R.D., Sinclair, M.C. & Dilworth, I.J., "Finite source analysis and simulation of private mobile radio systems", Proc. 10th UK Performance Engineering Workshop, Edinburgh, September 1994, pp. 39–45

16. Chng, R.S.K., Botham, C.P., Johnson, D., Brown, G.N., Sinclair, M.C., O'Mahony, M.J., & Hawker, I., "A multi-layer restoration strategy for reconfigurable networks", Proc. IEEE Global Telecom. Conf. (GLOBECOM'94), San Francisco, USA, November 1994, pp. 1872–1878

17. Robson, S.R. & Sinclair, M.C., "An improved channel resource queueing algorithm for trunked mobile radio", Proc. 12th UK Teletraffic Symposium, Old Windsor, March 1995, pp. 17/1–17/10

18. Segkhoonthod, S. & Sinclair, M.C., "Analysis of a fault-tolerant ATM switch based on a parallel architecture", Proc. 12th UK Teletraffic Symposium, Old Windsor, March 1995, pp. 22/1–22/9

19. Sinclair, M.C., "Minimum cost topology optimisation of the COST 239 European optical network", Proc. Intl. Conf. on Artificial Neural Networks and Genetic Algorithms (ICANNGA'95), Alès, France, April 1995, pp. 26–29

20. Segkhoonthod, S. & Sinclair, M.C., "Re-routing analysis of a fault-tolerant ATM switch based on a parallel architecture", Proc. 3rd Workshop on Performance Modelling and Evaluation of ATM Networks, Ilkley, July 1995, pp. 73/1–73/12

21. Robson, S.R. & Sinclair, M.C., "Reduction of channel idle time in a trunked mobile radio network", Proc. 11th UK Performance Engineering Workshop, Liverpool, September 1995, pp. 355–369

22. Brander, A.W. & Sinclair, M.C., "A comparative study of k-shortest path algorithms", Proc. 11th UK Performance Engineering Workshop, Liverpool, September 1995, pp. 370–379

23. Tan, L.G. & Sinclair, M.C., "Wavelength assignment between the central nodes of the COST 239 European optical network", Proc. 11th UK Performance Engineering Workshop, Liverpool, September 1995, pp. 235–247

24. Segkhoonthod, S. & Sinclair, M.C., "Performance analysis of a fault-tolerant ATM switch in single-fault conditions", Proc. 11th UK Performance Engineering Workshop, Liverpool, September 1995, pp. 48–63

25. Proestaki, A. & Sinclair, M.C., "Initial survey of heuristics for optical core network design in ACTS WOTAN", Proc. 13th UK Teletraffic Symposium, Glasgow, March 1996, pp. 20/1–20/8

26. Griffith, P.S., Proestaki, A. & Sinclair, M.C. , "Heuristic topological design of low-cost optical telecommunication networks", Proc. 12th UK Performance Engineering Workshop, Edinburgh, September 1996, pp. 129–140

27. Parnis, N., Jones, E.V., Sinclair, M.C. & O'Mahony, M.J., "Hierarchical network design for ACTS OPEN: Initial considerations", Proc. 12th UK Performance Engineering Workshop, Edinburgh, September 1996, pp. 141–155

28. Sinclair, M.C., "NOMaD: Initial architecture of an optical network optimisation, modelling and design tool", Proc. 12th UK Performance Engineering Workshop, Edinburgh, UK, September 1996, pp. 157–167

29. Kirkwood, I.M.A., Shami, S.H. & Sinclair, M.C., "Discovering simple fault-tolerant routing rules using genetic programming", Proc. Intl. Conf. on Artificial Neural Networks and Genetic Algorithms (ICANNGA'97), Norwich, UK, April 1997, pp. 285–288

30. Sinclair, M.C., "NOMaD: Applying a genetic-algorithm/heuristic hybrid approach to optical network topology design", Proc. Intl. Conf. on Artificial Neural Networks and Genetic Algorithms (ICANNGA'97), Norwich, UK, April 1997, pp. 299–303

31. Ghanbari, M., Hughes, C.J., Sinclair, M.C. & Eade, J.P., *Principles of Performance Engineering for Telecommunication and Information Systems*, IEE, 1997

32. Jackson, E., Spindley, R., Sinclair, M. & Reader, S., "SS7 network management using link monitors", Proc. IEEE Intl. Conf. Commun. (ICC'97), Montréal, Canada, June 1997, pp. 883–888

33. Segkhoonthod, S. & Sinclair, M.C., "Design of fault-tolerant ATM switch based on parallel architecture", *Electronics Letters* v33 n15 July 1997 pp. 1289–1290

34. Stevens, R.D. & Sinclair, M.C., "Finite-source analysis of traffic on private mobile radio systems", *Electronics Letters* v33 n15 July 1997 pp. 1292–1293

35. Shami, S.H., Kirkwood, I.M.A., & Sinclair, M.C., "Evolving simple fault-tolerant routing rules using genetic programming", *Electronics Letters* v33 n17 August 1997 pp. 1440–1441

36. Aiyarak, P., Saket, A.S. & Sinclair, M.C., "Genetic programming approaches for minimum cost topology optimisation of optical telecommunication networks", Proc. IEE/IEEE Intl. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'97), Glasgow, September 1997, pp. 415–420

37. Sinclair, M.C. & Shami, S.H., "Evolving simple software agents: Comparing genetic algorithm and genetic programming performance", Proc. IEE/IEEE Intl. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'97), Glasgow, September 1997, pp. 421–426

38. Sinclair, M.C., "Minimum network wavelength requirement design using a genetic-algorithm/heuristic hybrid", *Electronics Letters* v34 n4 February 1998 pp. 388–389

39. Sinclair, M.C., "Minimum cost routing and wavelength allocation using a genetic-algorithm/heuristic hybrid approach", Proc. Sixth IEE Conf. on Telecommunications, Edinburgh, UK, March/April 1998, pp. 67–71

40. Sinclair, M.C., "Operator-probability adaptation in a genetic-algorithm/heuristic hybrid for optical network wavelength allocation", Proc. IEEE Intl. Conf. on Evolutionary Computation (ICEC'98), Anchorage, Alaska, USA, May 1998, pp. 840–845

41. Sinclair, M.C., "NOMaD: An optical network optimisation, modelling and design toolset", Proc. IEE Colloquium on Multiwavelength Optical Networks: Devices, Systems and Network Implementations, Digest No. 1998/296, London, UK, June 1998, pp. 6/1–6/6

42. Sinclair, M.C., "Minimum cost wavelength-path routing and wavelength allocation using a genetic-algorithm/heuristic hybrid approach", *IEE Proc. Communications* v146 n1 February 1999 pp. 1–7

43. Proestaki, A. & Sinclair, M.C., "Interconnection strategies for dual-homing multi-ring networks", Proc. 16th International Teletraffic Conference (ITC-16), Edinburgh, UK, June 1999, pp. 169–181

44. Proestaki, A. & Sinclair, M.C., "Wavelength routing in all-optical dual-homing hierarchical multi-ring networks", Proc. European Conference on Networks and Optical Communications — Core Networks and Network Management (NOC'99), Delft, The Netherlands, June 1999, pp. 52–59

45. Shami, S.H. & Sinclair, M.C., "Co-evolutionary agents for telecommunication network restoration", Proc. Recent Advances in Soft Computing'99, Leicester, UK, July 1999, pp. 146–151

46. Navarro Varela, G. & Sinclair, M.C., "Ant colony optimisation for virtual-wavelength-path routing and wavelength allocation", Proc. Congress on Evolutionary Computation (CEC'99), Washington DC, USA, July 1999, pp. 1809–1816

47. Moore, S.J. & Sinclair, M.C., "Design of routing tables for a survivable military communications network using genetic algorithms", Proc. Congress on Evolutionary Computation (CEC'99), Washington DC, USA, July 1999, pp. 1788–1795

48. Sinclair, M.C., Corne, D. & Smith, G.D., "Evolutionary telecommunications: Past, present and future", Proc. GECCO'99 Workshop on Evolutionary Telecommunications: Past, Present and Future, Orlando, Florida, USA, July 1999, p. 208

49. Sinclair, M.C., "Evolutionary telecommunications: A summary", Proc. GECCO'99 Workshop on Evolutionary Telecommunications: Past, Present and Future, Orlando, Florida, USA, July 1999, pp. 209–212

50. Sinclair, M.C. & Clark, A.F., "Evolving an artificial vision system: Initial considerations", Proc. GECCO'99 Workshop on Evolution of Sensors in Nature, Hardware and Simulation, Orlando, Florida, USA, July 1999, pp. 191–195

51. Sinclair, M.C., "Optical mesh topology design using node-pair encoding genetic programming", Proc. Genetic and Evolutionary Computation Conference (GECCO-99), Orlando, Florida, USA, July 1999, pp. 1192–1197

52. Pagonis, J. & Sinclair, M.C., "Evolving personal agent environments to reduce internet information overload: Initial considerations", Proc. IEE Colloquium on Lost in the Web: Navigation on the Internet, Digest No. 1999/169, London, UK, November 1999, pp. 2/1–2/10

53. Proestaki, A. & Sinclair, M.C., "Impact of topology on wavelength and switch-port requirements in all-optical hierarchical multi-ring networks", Proc. IEEE Global Telecom. Conf. (GLOBECOM'99), Rio de Janeiro, Brazil, December 1999, pp. 1035–1041

54. Proestaki, A. & Sinclair, M.C., "Dimensioning all-optical dual-homing hierarchical multi-ring networks with various connectivities", Proc. Second Intl. Workshop on the Design of Reliable Communication Networks (DRCN2000), Munich, Germany, April 2000

55. Proestaki, A. & Sinclair, M.C., "Design and dimensioning of dual-homing hierarchical multi-ring networks", *IEE Proc. Communications* v147 n2 April 2000 pp. 96–104

56. Pagonis, J. & Sinclair, M.C., "Evolving user profiles to reduce internet information overload", Proc. Intl. Conf. on Recent Advances in Soft Computing 2000, Leicester, UK, June 2000

57. Sinclair, M.C., "Node-pair encoding genetic programming for optical mesh network topology design" in Corne, D.W., Oates, M.J. & Smith, G.D. (Ed.), *Telecommunications Optimization: Heuristic and Adaptive Techniques*, Wiley, 2000, Ch.6, pp. 99–114

Appendix C

Wavelength Allocation Heuristics

This appendix provides details of four overall heuristics for wavelength-path (WP) routing, fibre choice and wavelength allocation, plus one for virtual-wavelength-path (VWP). They have all been implemented in NOMaD, and the corresponding classes are listed in Table 4.2 and illustrated in Fig. 4.6 on p. 83.

The appendix is divided into three sections: §C.1 describes a WP allocation heuristic by Wuttisittikulkij & O'Mahony [298]; §C.2 details two WP heuristics by Wauters & Demeester [306]; and finally, §C.3 presents a VWP allocation heuristic by Nagatsu *et al.*, which forms the basis for a further WP one by the same authors [296].

C.1 Wuttisittikulkij & O'Mahony

Wuttisittikulkij and O'Mahony [298] have developed a simple, fast and non-iterative WP allocation heuristic, which works with k-lowest hop count paths[1] (and multiple fibres) to obtain a low network wavelength requirement (NWR).

First, for all source-destination node pairs, the k-lowest hop count paths between them are determined. Then, in the basic (unsorted) variant of the heuristic, the node pairs are considered in order of their end-node ids. For each wavelength channel required, the path currently with the lowest free wavelength end-to-end is selected from the choice of k paths. If there is more than one path with the same lowest wavelength free then, of these, the one with the lowest hop count is chosen. However, if there is still not a unique choice, then a random selection is made between all those with the lowest wavelength and hop count.

[1] In [298], the expression 'k-shortest path' is used rather than 'k-lowest hop count path'. However, this and other potential ambiguities in the paper were clarified in discussion between the author and Wuttisittikulkij.

Parameter	Default	Description
`ww95.theK`	`4`	Number of alternative paths, k
`ww95.sortPaths`	`false`	Allocate the paths in hop-count order?
`ww95.firstFibreOnly`	`true`	Use only first fibre on each cable?

Table C.1: `WaveWutti95` parameters

For the sorted variant, rather than simply allocating in order of their end-node ids, the node pairs are instead pre-sorted in decreasing order of the minimum number of hops between them. It was hoped by the paper's authors that this might improve the resulting NWR, as paths with a greater number of hops are often found to be more difficult to allocate.

Their heuristic has been implemented in NOMaD as class `WaveWutti95`, the parameters for which are given in Table C.1. The first of these (`ww95.theK`) is simply the constant k. The third, `ww95.firstFibreOnly`, occasionally avoids the need to edit the NOMaD network file from which the wavelength-routed network is built: if `true`, the router uses only the *first* fibre on each cable in the network. This option can be useful if a single-fibre network design is under consideration. The remaining parameter, `ww95.sortPaths`, determines whether the paths are allocated in hop-count order or not.

C.2 Wauters & Demeester

In [306], Wauters and Demeester presented two WP allocation heuristics. The first of these is a non-iterative algorithm based on a version of Dijkstra's algorithm modified such that in "extend[ing] an already found part of a route, only those [cables] are considered which have a wavelength channel free which is available on all [cables] of the already found part of the route". Although not stated in [306], the allocation presumably proceeds in arbitrary order of source-destination node pairs, such as according to end-node ids, and one wavelength channel at a time. This heuristic results in a low-cost allocation, whilst also keeping NWR low.

Their second algorithm (HRWA), is based on iterative application of a modified version of Yen's k-shortest path algorithm [320], and aims to produce the lowest cost allocation that still achieves minimum NWR. The modification to Yen's algorithm required can be achieved by simply incorporating as a sub-routine the modified version of Dijkstra's algorithm already referred to above. An overall flowchart for the HRWA heuristic[2] is given in Fig. C.1.

[2]The flowchart and description differ slightly from that given in [306] due to some ambiguities/inconsistencies in the original paper. Although Wauters was contacted to obtain clarification of these, no response was received.

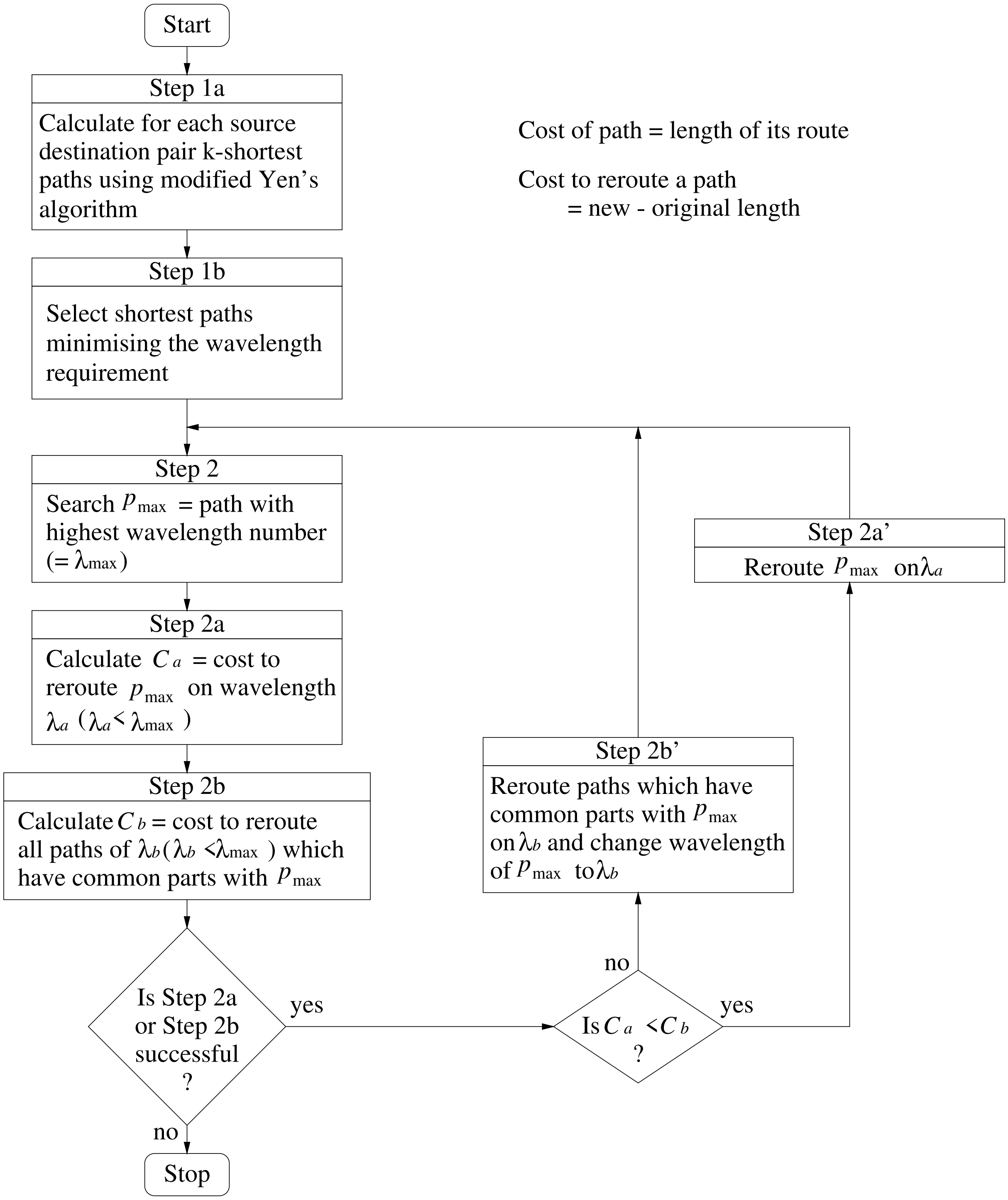


Figure C.1: Flowchart of HRWA (after [306])

Parameter	Default	Description
`wadi.firstFibreOnly`	`true`	Use only first fibre on each cable?

Table C.2: `WaveWauDijk` parameter

Parameter	Default	Description
`wahr.theK`	4	Number of alternative paths, k
`wahr.firstFibreOnly`	`true`	Use only first fibre on each cable?

Table C.3: `WaveWauHRWA` parameters

The HRWA heuristic starts (Step 1a) by determining the k-shortest paths for each node pair using the modified version of Yen's algorithm. For each wavelength channel, in arbitrary node-pair order, that path is selected with the lowest free wavelength end-to-end; if there is more than one, the shortest amongst these is chosen (Step 1b). Then, the first wavelength channel (*i.e.* wavelength-routed path; $p_{\max}$) in the network that uses the highest wavelength number ($\lambda_{\max}$), found by searching the cables in order of their end-node ids, is located (Step 2). Two different attempts are then made to move this wavelength channel to a lower wavelength number. The first (Step 2a) attempts to reroute it on a lower wavelength number (and the lowest available end-to-end) using one of the k-shortest paths (and the shortest path is selected if there is more than one at the lowest available wavelength number). The second (Step 2b) 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. Both these steps have been selected as wavelength allocation operators for NOMaD, and they are described on p. 84 and illustrated by examples in Fig. 4.8. If both Step 2a and 2b are feasible, then the one that causes the least growth in overall network path length is selected. Having changed the network allocation, the heuristic then proceeds to another iteration in the hope of further improvements. It should be noted however that the modified k-shortest paths have to be recalculated, when required, each iteration due to the effect of the reallocations. If at any point no improvements can be made, the algorithm terminates.

Both Wauters & Demeester's heuristics have been implemented in NOMaD as classes `WaveWauDijk` and `WaveWauHRWA` respectively, the parameters for which are given in Tables C.2 and C.3. These parameters are directly equivalent to those used by `WaveWutti95`, and similar considerations apply.

C.3 Nagatsu *et al.*

In contrast, an overall allocation heuristic for VWP networks is described by Nagatsu *et al.* [296]. This is a two-stage algorithm, which first establishes reasonable initial routes, and then improves these in a second iterative rerouting stage, aiming for minimum NWR[3].

The initial routing (Step 1) is achieved by first determining both the minimum number of hops and the number of wavelength channels required between each source-destination node pair. Individual channels are then routed in priority order, with the highest priority given to the node pair with the greatest product of minimum hop count and channels remaining to be routed (*cf.* Wuttisittikulkij & O'Mahony's sorted variant). The routing itself is achieved by maintaining a weight associated with each cable; this is simply the number of channels (paths) using it. A channel is actually routed by assigning it to the path with the lowest sum of link weights using Dijkstra's algorithm[4]; any affected weights are then updated.

Once all the channels have been given an initial route, rerouting can begin (Step 2). For each iteration, two different attempts are made to improve the eventual NWR. Both involve identifying those cables (called *maximum-accommodating cables*) with a fibre carrying the highest number of channels, and which will therefore have to use the highest wavelength number after allocation. First, those paths which use the largest number of maximum-accommodating cables are identified as candidates, and an attempt is made to reroute them using a minimum-link-weight path. If the rerouted path would be on a lower wavelength than the original, then it is adopted, otherwise the original path is restored. If, after trying all candidates paths, no improvement is achieved, the second attempt is made. This time, any path which uses one or more maximum-accommodating cables is a candidate, and each one is rerouted onto a new path that contains as few as possible of the maximum-accommodating cables it was originally using (and not including, of course, any maximum-accommodating cables that were not part of the original path). Throughout, Nagatsu *et al.* impose an upper limit on path length: no path is ever adopted which is more than, say, 2 hops longer than the minimum-hop-count path between that node pair. In addition, they impose a limit, say 50, on the number of iterations of the second stage.

Finally, wavelength assignment is simply achieved by a greedy method, which assigns channels to the lowest available wavelength on any fibre of a cable.

[3] As with previous algorithms, some reasonable assumptions were taken to overcome ambiguities/inconsistencies in the original paper.

[4] The initial weights, which would otherwise be zero, are instead initialised to a small constant causing lowest-hop-count routing.

Parameter	Default	Description
`wnvw.firstFibreOnly`	`true`	Use only first fibre on each cable?
`wnvw.step2Count`	50	Maximum iterations of Step 2
`wnvw.hopIncrMax`	2	Maximum allowed increase in path hop count over minimum achievable

Table C.4: `WaveNagVWP` parameters

In the same paper [296], Nagatsu *et al.* also described a WP allocation heuristic derived straightforwardly from their earlier VWP one. They argue that the routes used by the VWP paths they obtain would also make excellent choices for the equivalent WP-routed network. Consequently, after their VWP algorithm has terminated, all the routes used by the network paths are adopted unchanged, but their associated fibre and wavelength allocations discarded. Then, starting from the lowest wavelength number, each path in turn is checked to see whether it can be allocated end-to-end on that wavelength. After each iteration, the wavelength number is incremented, and all the remaining paths checked again. The algorithm terminates when the last path has been successfully re-allocated.

Both of Nagatsu *et al.*'s heuristics have been implemented in NOMaD as classes `WaveNagVWP` and `WaveNagWP` respectively. The latter has no parameters of its own, being based on the VWP heuristic, so both share the parameters given in Table C.4. The first of these, as before, is used to limit the heuristic to just the first fibre of each cable. The other two correspond to the two parameters of the VWP heuristic discussed above.

Appendix D

Evolutionary Telecommunications Tables

This appendix provides detailed tables that were judged inappropriate for direct inclusion in Ch. 2. Those papers included within the scope of the evolutionary telecommunications survey of §2.3 are listed (in alphabetical order of first author surname) in Table D.1. For each paper, as well as its own reference number in the thesis bibliography, those papers it cites and which in turn cite it, from within the survey, are also recorded. In addition, classification of each paper in terms of application area (and problem element), solution representation and genetic operators (including population dynamics) are provided[1]. Next, Table D.2 shows the chronological distribution of the survey papers. Finally, Table D.3 records their classification into application groups. While the papers have usually only been included in one column (application area), they have been counted in as many rows (problem elements) as necessary.

[1] It has usually been assumed in determining classifications that authors have followed their earlier work in subsequent papers, unless they indicate otherwise.

Table D.1: Evolutionary telecommunications papers – alphabetical order

Author(s) (Year)	Ref.	Appl.	Repr.	Ops.	Cites	Cited by
Abed & Ghanta (1996)	[230]	O, w	b	E, f, x, m	—	—
Abuali *et al.* (1993)	[203]	G, x	i, p	G, S, u, x, p	—	[193], [204], [205], [206], [207]
Abuali *et al.* (1994)	[193]	G, t	p	G, f, p, m, a	[203]	[204], [205], [206], [207], [192]
Abuali *et al.* (1994)	[204]	G, x	i	x, u, h, m	[203], [193]	[205], [206], [207], [192]
Abuali *et al.* (1995)	[205]	G, x	i	u, m	[203], [204], [193]	[206]
Abuali *et al.* (1995)	[206]	G, x	i, p	G, f, h, p, m, a	[203], [204], [193], [205]	—
Abuali *et al.* (1995)	[207]	G, x	i	u, h, m, a, r	[203], [204], [193]	—
AbuAli *et al.* (1996)	[194]	G, t	b	G, f, x, u, m, a, r	[172]	—
Aiyarak *et al.* (1997)	[147]	O, t, d	g	G, t, g	[166]	—
Bentall *et al.* (1997)	[218]	G, p	h:p	G, f, p, m	[178]	—
Berry *et al.* (1995)	[209]	G, x	i	E, h, m	—	—
Brittain *et al.* (1997)	[174]	O, n, t	h:b,p	S, x, p, m	[161]	—
Calégari *et al.* (1997)	[186]	R, n	b	G, M, m	—	—
Carse *et al.* (1995)	[221]	C, r	h:r	S, s, h, m	—	[222], [223]
Celli *et al.* (1995)	[179]	G, n	b	G, f, u, m	[138]	—
Chang *et al.* (1993)	[169]	G, R, d, n	i, b	E, f, u, x, m, r	[138], [137], [142]	[171], [170]
Chang & Potter (1994)	[171]	G, R, d, n	i, b	E, f, u, x, m, r	[169], [138], [137], [142], [170]	—
Chardaire *et al.* (1995)	[178]	G, R, C, x, f, n	b	G, E, f, s, x, u, h, m, r	[177], [176]	[218]
Christensen *et al.* (1997)	[197]	G, r, x	p	G, f, p	[140], [202]	[198]

Applications: general networks (G); optical networks (O); radio networks (R); computer networks (C); dimensioning (d); node location (n); topology (t); trees (x); routing (r); restoration (p); call admission (a); wavelength allocation (w); frequency assignment (f)

Representation: binary (b); integer (i); real (r); permutation (p); tree (g); hybrid (h)

Operators: generational (G); elitist (E); steady-state (S); multi-population (M); breeder (B); fit. prop. selection (f); tournament selection (t); ranked selection (s); single- or multi-point crossover (x); uniform crossover (u); permutation crossover (p); heuristic recombination (h); tree crossover (g); mutation (m); inversion (i); adaptation (a); repair or local search (r)

continued on next page

continued from previous page

Author(s) (Year)	Ref.	Appl.	Repr.	Ops.	Cites	Cited by
Coombs & Davis (1987)	[139]	C, d	i	E, s, x, m, a, r	[137], [138]	[138], [213], [214], [163], [164]
Cox *et al.* (1991)	[140]	G, r	p	p	—	[197], [172], [185], [214], [223], [164], [198]
Crisan & Mühlenbein (1998)	[240]	R, f	i	B, h, m, r	[232]	—
Crompton *et al.* (1993)	[231]	R, f	i	E, M, f, x, u, m, r	—	[234], [235], [177], [239]
Crompton *et al.* (1994)	[232]	R, f	i, h:i	E, M, f, x, u, h, m	—	[240], [234], [235], [236], [237]
Crompton *et al.* (1994)	[233]	R, f	i, h:i	E, M, f, x, u, h, m, r	—	—
Cuppini (1994)	[238]	R, f	b	E, f, m	—	[241]
Davis & Coombs (1987)	[138]	C, d	i	E, s, x, m, a	[139], [137]	[179], [169], [171], [139], [137], [213], [214]
Davis & Coombs (1989)	[137]	C, d	i	E, s, x, m, a	[138]	[169], [171], [139], [138], [163], [164]
Davis *et al.* (1993)	[172]	G, d, r	h:i,p	S, s, h, m, r	[140]	[194], [191], [192], [151], [152], [153], [214]
Deeter & Smith (1997)	[148]	G, t, d	i	E, s, u, m	[188], [181], [168], [180]	[190], [191]
Dengiz *et al.* (1995)	[188]	G, t	b	E, f, x, m	—	[148]
Dengiz *et al.* (1997)	[189]	G, t	b	E, f, x, m	[181], [180]	—
Dengiz *et al.* (1997)	[190]	G, t	b	E, f, x, m	[181], [180], [148]	[191], [192]
Dengiz *et al.* (1997)	[191]	G, t	i	E, s, h, m, r	[172], [148], [190], [181], [180]	[192]

Applications: general networks (G); optical networks (O); radio networks (R); computer networks (C); dimensioning (d); node location (n); topology (t); trees (x); routing (r); restoration (p); call admission (a); wavelength allocation (w); frequency assignment (f)

Representation: binary (b); integer (i); real (r); permutation (p); tree (g); hybrid (h)

Operators: generational (G); elitist (E); steady-state (S); multi-population (M); breeder (B); fit. prop. selection (f); tournament selection (t); ranked selection (s); single- or multi-point crossover (x); uniform crossover (u); permutation crossover (p); heuristic recombination (h); tree crossover (g); mutation (m); inversion (i); adaptation (a); repair or local search (r)

continued on next page

continued from previous page

Author(s) (Year)	Ref.	Appl.	Repr.	Ops.	Cites	Cited by
Dengiz *et al.* (1997)	[192]	G, t	i	E, s, h, m, r	[181], [180], [172], [193], [204], [168], [190], [191]	—
Elbaum & Sidi (1995)	[210]	C, x	h:i,p,i	G, f, x, m, r	[175]	[154]
Elbaum & Sidi (1996)	[211]	C, x	h:i,p,i	G, f, x, m, r	[175], [27], [142]	—
Esbensen & Mazumder (1994)	[200]	G, x	h:b	E, s, x, m, i, r	[176]	[201]
Esbensen (1995)	[201]	G, x	h:b	E, s, x, m, i, r	[200], [144], [199], [176]	[202]
Esbensen (1995)	[202]	G, x	b	S, s, u, m, a, r	[201], [176]	[197], [198]
Fogarty *et al.* (1995)	[222]	C, r	h:r	S, s, h, m	[221]	—
Garcia *et al.* (1998)	[173]	G, d	h:i,r	S, x, m, r	—	—
Gondim (1996)	[185]	R, n	i	E, h	[140], [227]	—
He (1997)	[149]	C, d, r	i	E, f, h, m	—	—
Hesser *et al.* (1989)	[144]	G, x	b	G, f, x, m	—	[201], [199]
Hewitt *et al.* (1995)	[150]	C, t, d	h	G, s, h, m	[143], [175], [27]	[34]
Hoelting *et al.* (1996)	[215]	G, r	p	G, p, m	—	—
Hoelting *et al.* (1996)	[216]	G, r	p	p, m	—	—
Huang *et al.* (1996)	[151]	G, r, t, d	i	S, f, h, m	[172], [160]	[153]
Huang & Ma (1996)	[152]	G, r, t, d	i	M, f, h, m	[172], [160]	[153]
Huang *et al.* (1997)	[153]	G, r, t, d	i	M, f, h, m	[172], [151], [152], [160]	—
Hurley & Smith (1995)	[234]	R, f	h:i	E, M, f, h, m	[231], [232]	[236]
Hurley *et al.* (1996)	[235]	R, f	h:i	E, M, f, h, m	[231], [232]	—
Julstrom (1993)	[199]	G, x	h:i,b	S, s, x, h, m	[144]	[201], [160]
Kaminsky (1997)	[242]	R, f	h:i	S, s, u, m	—	—

Applications: general networks (G); optical networks (O); radio networks (R); computer networks (C); dimensioning (d); node location (n); topology (t); trees (x); routing (r); restoration (p); call admission (a); wavelength allocation (w); frequency assignment (f)

Representation: binary (b); integer (i); real (r); permutation (p); tree (g); hybrid (h)

Operators: generational (G); elitist (E); steady-state (S); multi-population (M); breeder (B); fit. prop. selection (f); tournament selection (t); ranked selection (s); single- or multi-point crossover (x); uniform crossover (u); permutation crossover (p); heuristic recombination (h); tree crossover (g); mutation (m); inversion (i); adaptation (a); repair or local search (r)

continued on next page

continued from previous page

Author(s) (Year)	Ref.	Appl.	Repr.	Ops.	Cites	Cited by
Kapsalis *et al.* (1993)	[176]	G, x	b	E, f, s, x, m, r	[144]	[178], [200], [201], [202], [177], [239], [213], [214]
Kapsalis *et al.* (1995)	[177]	R, f	b	G, x, u, h, m, a	[231], [176]	[178], [213]
Kapsalis & Smith (1996)	[239]	R, f	b	G, x, u, h, m, a, r	[176], [231]	—
Kirkwood *et al.* (1997)	[219]	G, r	g	M, f, g	—	[220]
Ko *et al.* (1997)	[154]	C, t, r, d	b, i, h:i	S, f, x, h, m, r	[210], [160], [163]	—
Kozdrowski *et al.* (1997)	[155]	G, r, d	h:i	S, s, u, m	—	—
Kumar *et al.* (1992)	[145]	C, t	b	E, t, h, m, r	—	[182]
Kumar *et al.* (1995)	[181]	C, t	b	E, t, h, m, r	—	[148], [189], [190], [191], [192]
Kumar *et al.* (1995)	[180]	C, n, t	b	E, t, h, m, r	—	[148], [189], [190], [191], [192], [182]
Mann *et al.* (1995)	[213]	G, r	b	—	[139], [138], [176], [177], [212]	[156]
Mann & Smith (1996)	[214]	G, r	b	S, f, t, x, u, m	[139], [140], [138], [172], [176], [141], [212]	[156]
Mann & Smith (1997)	[156]	O, r, d	b	G, E, S, f, t, x, u, m	[213], [214]	—
Markaki *et al.* (1997)	[226]	C, r	h:b	S, f, h, m, r	—	—
Michalewicz (1991)	[143]	C, t	h:b	G, h, m	—	[150], [158], [208]
Mikac & Inkret (1997)	[157]	O, t, d	b	E, f, x, m	[166]	—
Munakata & Hashier (1993)	[158]	G, r, d	i	G, f, h, m, a	[143]	[223]
Munetomo *et al.* (1997)	[223]	C, r	h:i	S, s, h, m	[140], [158], [221]	—
Nakamura & Oda (1997)	[196]	G, t	i	S, x, m	—	—

Applications: general networks (G); optical networks (O); radio networks (R); computer networks (C); dimensioning (d); node location (n); topology (t); trees (x); routing (r); restoration (p); call admission (a); wavelength allocation (w); frequency assignment (f)

Representation: binary (b); integer (i); real (r); permutation (p); tree (g); hybrid (h)

Operators: generational (G); elitist (E); steady-state (S); multi-population (M); breeder (B); fit. prop. selection (f); tournament selection (t); ranked selection (s); single- or multi-point crossover (x); uniform crossover (u); permutation crossover (p); heuristic recombination (h); tree crossover (g); mutation (m); inversion (i); adaptation (a); repair or local search (r)

continued on next page

continued from previous page

Author(s) (Year)	Ref.	Appl.	Repr.	Ops.	Cites	Cited by
Ngo & Li (1998)	[241]	R, f	b	G, f, h, m, r	[238]	—
Palmer & Kershenbaum (1994)	[159]	G, x, d	b, i	E, f, x, m	—	—
Palmer & Kershenbaum (1995)	[160]	G, x, d	b, i	E, f, x, m	[199]	[151], [152], [153], [154]
Pan & Wang (1991)	[141]	C, r, d	i	G, f, h, m	—	[214], [212], [224], [225]
Paul & Tindle (1996)	[161]	O, n, x, d	h	h	—	[174], [162]
Paul *et al.* (1996)	[162]	O, n, x, d	h	h	[161]	—
Pierre & Legault (1996)	[163]	C, t, d	b	G, f, x, m	[139], [137]	[154]
Pierre & Legault (1998)	[164]	C, t, d	b	G, f, x, m	[140], [139], [137]	—
Potter *et al.* (1992)	[142]	G, R, d, n	i, b	E, f, u, h, x, m	—	[169], [171], [211], [170]
Potter *et al.* (1993)	[170]	G, R, d, n	i, b	E, f, u, x, m, r	[169], [142]	[171]
Qin *et al.* (1997)	[165]	C, t, d	b	E, f, x, m, r	—	—
Routen (1994)	[175]	G, t, x, n	i	G, f, x, m	—	[210], [211], [150], [208]
Saha & Chakraborty (1997)	[195]	G, t	b	G, f, x, m	—	—
Sandalidis *et al.* (1998)	[243]	R, f	b	G, s, m	—	—
Servet *et al.* (1997)	[217]	G, r	b, r	E, f, x, h, m	—	—
Shami *et al.* (1997)	[220]	G, r	g	M, f, g	[219]	—
Shimamoto *et al.* (1993)	[212]	G, r	i	S, f, x, m	[141]	[213], [214], [224], [225]
Sinclair (1993)	[27]	G, r	b	G, f, x, m	—	[211], [150], [224], [225]
Sinclair (1995)	[166]	O, t, d	b	E, f, x, m	—	[147], [157], [34], [36], [29], [184]
Sinclair (1997)	[34]	O, t, d	h	G, t, x, h, m, a, r	[166], [150]	[35], [36]
Sinclair (1998)	[35]	O, r, w	h	G, t, h, m, a, r	[34], [29]	—

Applications: general networks (G); optical networks (O); radio networks (R); computer networks (C); dimensioning (d); node location (n); topology (t); trees (x); routing (r); restoration (p); call admission (a); wavelength allocation (w); frequency assignment (f)

Representation: binary (b); integer (i); real (r); permutation (p); tree (g); hybrid (h)

Operators: generational (G); elitist (E); steady-state (S); multi-population (M); breeder (B); fit. prop. selection (f); tournament selection (t); ranked selection (s); single- or multi-point crossover (x); uniform crossover (u); permutation crossover (p); heuristic recombination (h); tree crossover (g); mutation (m); inversion (i); adaptation (a); repair or local search (r)

continued on next page

continued from previous page						
Author(s) (Year)	Ref.	Appl.	Repr.	Ops.	Cites	Cited by
Sinclair (1998)	[36]	O, r, w	h	G, t, h, m, a, r	[34], [166]	—
Sobotka (1992)	[146]	G, t	?	E, h	—	—
Srivastava *et al.* (1997)	[182]	G, t, n	b	E, t, h, m, r	[145], [180]	—
Tan & Sinclair (1995)	[29]	O, r, w	b	G, f, x, m	[166]	[35]
Tanaka & Berlage (1996)	[167]	G, t, d	h:b	G, f, h, r	—	—
Tang *et al.* (1997)	[187]	R, n	h:b,r	S, s, x, h, m	—	—
Tanterdtid *et al.* (1997)	[224]	C, r	i	S, f, x, m	[141], [27], [212]	—
Tanterdtid *et al.* (1997)	[225]	C, r	i	S, f, x, m	[141], [27], [212]	—
Valenzuela *et al.* (1997)	[236]	R, f	p	S, s, p, m	[232], [234]	—
Valenzuela *et al.* (1998)	[237]	R, f	p	S, s, p, m	[232]	—
Walters & Smith (1995)	[168]	G, x, d	i	E, s, h, m	—	[148], [192]
Webb *et al.* (1997)	[183]	G, n, t	b	G, f, x, m, r	—	—
Webb *et al.* (1998)	[184]	G, n, t	b	G, f, x, m, r	[166]	—
Yener & Rose (1994)	[227]	R, a	b	G, f, x, m	—	[185], [228], [229]
Yener & Rose (1995)	[228]	R, a	b	G, f, x, m	[227]	[229]
Yener & Rose (1997)	[229]	R, a	b	G, f, x, m	[227], [228]	—
Zhao *et al.* (1995)	[208]	G, x	h	E, s, h, m, r	[143], [175]	—
Zhu *et al.* (1998)	[198]	G, r, x	p	G, f, p	[197], [140], [202]	—

Applications: general networks (G); optical networks (O); radio networks (R); computer networks (C); dimensioning (d); node location (n); topology (t); trees (x); routing (r); restoration (p); call admission (a); wavelength allocation (w); frequency assignment (f)

Representation: binary (b); integer (i); real (r); permutation (p); tree (g); hybrid (h)

Operators: generational (G); elitist (E); steady-state (S); multi-population (M); breeder (B); fit. prop. selection (f); tournament selection (t); ranked selection (s); single- or multi-point crossover (x); uniform crossover (u); permutation crossover (p); heuristic recombination (h); tree crossover (g); mutation (m); inversion (i); adaptation (a); repair or local search (r)

Year	Count	Papers
1987	2	[139], [138]
1988	0	
1989	2	[137], [144]
1990	0	
1991	3	[140], [143], [141]
1992	3	[145], [142], [146]
1993	10	[203], [169], [231], [172], [199], [176], [158], [170], [212], [27]
1994	10	[193], [204], [171], [232], [233], [238], [200], [159], [175], [227]
1995	25	[205], [206], [207], [209], [221], [179], [178], [188], [210], [201], [202], [222], [150], [234], [235], [177], [181], [180], [213], [160], [166], [29], [168], [228], [208]
1996	14	[230], [194], [211], [185], [215], [216], [151], [152], [239], [214], [161], [162], [163], [167]
1997	33	[147], [218], [174], [186], [197], [148], [189], [190], [191], [192], [149], [153], [242], [219], [154], [155], [156], [226], [157], [223], [196], [165], [195], [217], [220], [34], [182], [187], [224], [225], [236], [183], [229]
1998	10	[240], [173], [241], [164], [243], [35], [36], [237], [184], [198]
Total	112	

Table D.2: Evolutionary telecommunications papers – chronological order

Problem element	Application area			
	General networks (G)	Optical networks (O)	Radio networks (R)	Computer networks (C)
Dimensioning (d)	[169], [171], [172], [148], [173], [151], [152], [153], [155], [158], [159], [160], [142], [170], [167], [168]	[147], [156], [157], [161], [162], [166], [34],	[169], [171], [142], [170]	[139], [138], [137], [149], [150], [154], [141], [163], [164], [165]
Node location (n)	[179], [169], [171], [142], [170], [175], [182], [183], [184]	[174], [161], [162]	[186], [169], [171], [185], [142], [170], [187]	[178], [180]
Topology (t)	[193], [194], [148], [188], [189], [190], [191], [192], [151], [152], [153], [196], [175], [195], [146], [182], [167], [183], [184]	[147], [174], [157], [166], [34]	—	[150], [154], [145], [181], [180], [143], [163], [164], [165]
Trees (x)	[203], [204], [205], [206], [207], [209], [178], [197], [200], [201], [202], [144], [199], [176], [159], [160], [175], [168], [208], [198]	[161], [162]	—	[210], [211]
Routing, restoration & call admission (r/p/a)	[218], [197], [140], [172], [215], [216], [151], [152], [153], [219], [155], [213], [214], [158], [217], [220], [212], [27], [198]	[156], [35], [36], [29]	[227], [228], [229]	[221], [222], [149], [154], [226], [223], [141], [224], [225]
Wavelength allocation & frequency assignment (w/f)	—	[230], [35], [36]. [29]	[178], [240], [231], [232], [233], [238], [234], [235], [242], [177], [239], [241], [243], [236], [237]	—

Table D.3: Evolutionary telecommunications papers – application groups

Appendix E

Detail of NOMaD

This appendix provides details of NOMaD that were judged inappropriate for Chapters 3, 4 and 6. It is divided into eight sections: §E.1 gives details of the NOMaD file formats, and §E.2 provides cross reference tables for NOMaD's classes. The central features of NOMaD's network model are presented in §E.3. NOMaD's support for hierarchical networks and node clustering is discussed in §E.4, and §E.5 presents details of various iterator classes. Then in §E.6, class diagrams and descriptions of classes from the `Utilities` class category are provided. Greater detail on NOMaD's testing and verification strategy, initially reviewed in §3.8, is given in §E.7. The final section (§E.8) considers the design and implementation of the NOMaD software and makes suggestions for further work.

E.1 NOMaD File Formats

To simplify the input of both network specifications (`.net`) and algorithm parameters (`.nmd`), as well as the output of results (`.out`), a standard file format was adopted (after [321]). Each line in a NOMaD file is blank or a comment (and hence ignored) or a variable-value pair (*i.e.* a *parameter*). Comment lines simply begin with a `#`. Variable-value pairs have the following form:

```
variable = value
```

where `variable` is a string of non-white characters, and `value` is an arbitrary string. The variable-value pairs in a file can be included in any order, although clearly they are more readable to a user if some structure is imposed. However, if a variable appears more than once in a file, the last value is the one retained.

E.1.1 NOMaD Network (`.net`) Files

Network specification files are usually designated by a `.net` suffix, and provide either the node locations (plus other details) and traffic distribution required for network topology design, or the more complete network descriptions necessary for routing, fibre choice and wavelength allocation.

The variable-value pairs used for network topology design are listed in Table E.1, where the 'M/O' column indicates whether a pair is mandatory (M) or optional (O). The additional pairs required to specify a topology are given in the first part of Table E.2; the second part lists those necessary for routing, fibre and wavelength allocation. The designation of M or O is fairly arbitrary for some pairs in the latter table as, for example[1], `links[?].nodeIdS` is only mandatory if the optional `numLinks` has been specified with a value greater than zero!

E.1.2 NOMaD Parameter (`.nmd`) Files

NOMaD parameter files, designated by a `.nmd` suffix, provide all those parameters (*i.e.* variable-value pairs) that do not directly specify either a network design problem or a network description. The parameters are tabulated here in the context of the classes which make use of them. However, the class descriptions themselves can be found in Ch. 4, for the majority of class categories, Appendix C for the wavelength-allocation heuristics, and later in this Appendix for the `Utilities` class category (§E.6).

The parameters for class `Network` are given in Table E.3, and discussed on p. 256 in the main body of the thesis.

Next, the parameters for the design assessment metrics of §4.1 are tabulated. Those for classes `AlesCost` and `AlesRouter` are given in Tables E.4 and E.5, respectively. Together they implement Network Cost Model A, discussed in §4.1.1 on p. 68. Next, those for class `NetWaveReq` are detailed in Table E.6. This class implements the network wavelength requirement metric, described in §4.1.2 on p. 71. Finally, the parameters for class `PhyNetCost` are listed in Table E.7. This class implements Network Cost Model B, discussed in §4.1.3 on p. 72.

The next four tables give the parameters for the `GeneticAlgorithm`, `Population`, `Individual` and `OperatorPool` classes *i.e.* Tables E.8, E.9, E.10 and E.11, respectively. They are all part of the implementation of the author's genetic-algorithm (GA)/heuristic hybrid approach, discussed in §4.2 on p. 74.

[1] In this appendix, '`?`' has been used as a wildcard.

Variable	Example Value	M/O	Description
`numnodes`	`15`	M	Number of nodes; must be ≥ 2
`kmPerUnit`	`1`	O	Multiplier for node `x` and `y` locations; defaults to 200.0, must be > 0.0
`nodes[0].name`	`Paris`	O	Node 0 name; defaults to `Node0`, `Node1`, ...
`nodes[0].x`	`184.9`	M	Node 0 `x` location; must be ≥ 0.0
`nodes[0].y`	`970.1`	M	Node 0 `y` location; must be ≥ 0.0
`nodes[0].levels`	`{0,1}`	O	Node 0 levels; defaults to `{0}`, each entry in the list of levels must be in the range `0`–`31`
`nodes[0].clusters`	`{0,1}`	O	Node 0 clusters; defaults to `{0}`, each entry in the list of clusters must be in the range `0`–`31`
`traMatrix[0][1]`	`3.480`	M	Traffic from node 0 to node 1; must be ≥ 0.0, except self traffic (*e.g.* node 0 to node 0) which must be 0.0
`lenMatrix[0][1]`	`742.5`	O	Length of (potential) link between node 0 and node 1; defaults to the Euclidean distance between end nodes, must be > 0.0, and only lengths between node `x` and node `y` where `x` < `y` are recognised

Table E.1: Network file variable-value pairs

Variable	Example Value	M/O	Description
`numLinks`	`42`	O	Number of links; must be ≥ 0
`links[0].nodeIdS`	`1`	M	End node of link 0; must be included for all links in the range `0`–`(numLinks-1)`, value must be in the range `0`–`(numNodes-1)`
`links[0].nodeIdT`	`2`	M	Other end node of link 0; must be included for all links in the range `0`–`(numLinks-1)`, value must be in the range `0`–`(numNodes-1)`, no self loops allowed, and all links must be between different pairs of end nodes
`links[0].levelS`	`1`	O	Connect level of link 0 at end node `S`; defaults to `0`, must be in the range `0`–`31`, and node `S` must be on this level
`links[0].levelT`	`1`	O	Connect level of link 0 at end node `T`; defaults to `0`, must be in the range `0`–`31`, and node `T` must be on this level
`cabMatrix[0][1].numFibres`	`2`	M	Number of fibres in cable between node 0 and node 1; required for all links; must be > 0
`net.defaultNumFibres`	`2`	O	Default number of fibres; equivalent to specifying `cabMatrix[?][?].numFibres` for all possible links, although overridden by any specific `cabMatrix[?][?].numFibres` pairs included; must be > 0
`net.lambdaCap`	`10.0`	O	Capacity of a single wavelength channel; defaults to `2.5`; must be ≥ 0.0

Table E.2: Additional network file variable-value pairs

Parameter	Default	Description
`net.metric`	`ALCT`	Metric used for network self-assessment; default is `AlesCost` (§4.1.1)
`net.router`	`ALES`	Router used to route the network prior to self-assessment; default is `AlesRouter` (§4.1.1)

Table E.3: `Network` parameters

The parameters for the wavelength-allocation random k-shortest paths router class, `WaveRandKSP` (see p. 82), are given in Table E.12. The first of these (`wrks.theK`) is simply the constant k. The second, `wrks.firstFibreOnly`, occasionally avoids the need to edit the `.net` file from which the `PhyNetwork` was built: if `true`, the router uses only the *first* fibre on each cable in the network. This option can be useful if a single-fibre network design is under consideration. The final parameter (`wrks.cacheKSP`) is only appropriate if the `WaveRandKSP` object is to be applied to multiple `PhyNetwork`s all with identical topologies (as happens in a GA/heuristic hybrid!). The k-shortest paths used are generated by `WaveRandKSP`'s `Router` base class (using Yen's algorithm [320]): for identical topologies, an identical set of paths is always generated. Consequently, if the k-shortest paths are obtained the first time the `WaveRandKSP` router is applied, and then cached, they can be re-used on subsequent applications with a considerable saving in execution time.

Those for the wavelength-allocation k-shortest path mutation operator class, `WaveKSPMutate` (see p. 84), are detailed in Table E.13. They are directly equivalent to those used by `WaveRandKSP` above, and similar considerations apply.

Next, the parameters for the wavelength-allocation path reroute operator class, `WaveWauReroute` (see p. 84), are tabulated in Table E.14. Both of these are equivalent to the similarly named parameters of `WaveRandKSP` above, but unfortunately path cacheing cannot be applied, as the results of Wauters and Demeester's modified Yen's algorithm depend not only on the network topology, but also on the wavelength occupancies of the cables. Consequently, it must be run afresh whenever a k-shortest path decision is made.

Those for the wavelength-allocation path shift-out operator class, `WaveWauShiftOut` (see p. 85), are listed in Table E.15. These parameters are directly equivalent to those used by `WaveWauReroute` above, and again similar considerations apply.

Finally, the parameters for the wavelength-allocation k-shortest path single-point crossover operator class, `WaveKSPCO` (see p. 85), are given in Table E.16. They are directly equivalent to those used by `WaveRandKSP` above (*i.e.* path cacheing can be used here), and once again, similar considerations apply.

Parameter	Default	Description
`ales.eMin`	200.0	Constant K_0, shared with `AlesRouter`
`ales.eLink`	100.0	Constant K_n, shared with `AlesRouter`
`ales.traToCapK`	1.4	Constant K_T
`ales.capUnit`	2.5	Constant K_G
`ales.nnPenalty`	500,000.0	Penalty P_N
`ales.apPenalty`	250,000.0	Penalty P_R
`ales.alpha`	1.0	Constant α

Table E.4: `AlesCost` parameters

Parameter	Default	Description
`ales.eMin`	200.0	Constant K_0, shared with `AlesCost`
`ales.eLink`	100.0	Constant K_n, shared with `AlesCost`
`ales.altPathWgtK`	1.0	Constant K_R

Table E.5: `AlesRouter` parameters

Parameter	Default	Description
`nwr.nnPenalty`	5.0	Node-to-node penalty
`nwr.pfPenalty`	0.0	Per-fibre penalty

Table E.6: `NetWaveReq` parameters

Parameter	Default	Description
`pnct.eMin`	200.0	Constant K_0
`pnct.eLink`	100.0	Constant K_n
`pnct.nnPenalty`	250,000.0	Penalty P_N
`pnct.alpha`	1.0	Constant α
`pnct.beta`	1.0	Constant β
`pnct.gamma`	0.5	Constant γ

Table E.7: `PhyNetCost` parameters

Parameter	Default	Description
`maxTrials`	—	Maximum number of trials; must be ≥ 1

Table E.8: `GeneticAlgorithm` parameter

Parameter	Default	Description
`pop.size`	—	Population size
`pop.winSize`	1	Window size for best-in-window credit assignment, w
`pop.select`	—	Selection operator
`pop.tournament.size`	4	Tournament size for tournament selection
`pop.initOp`	`RLS`	Population initialisation operator; default is `RandomLinks` (§4.2.2, p. 80)

Table E.9: `Population` parameters

Parameter	Default	Description
`pop.phyNetwork`	`false`	Construct the individual using a `PhyNetwork` (`true`) or a `Network` (`false`)

Table E.10: `Individual` parameter

Parameter	Default	Description
`opp.numOps`	—	Number of operators (excluding `Reproduction`), s; must be ≥ 1
`opp.ops[0].name`	—	Name of operator 0
`opp.ops[0].prob`	—	Initial probability of operator 0
`opp.adaptProp`	`0.15`	Adaptation proportion, ψ; $0 \leq \psi \leq 1$
`opp.adaptNumGens`	`4`	Adaptation generation gap, ϵ; must be ≥ 1
`opp.creditK`	`0.5`	Credit constant, κ; must be ≥ 0
`opp.fitCredType`	`best`	Credit type: `best`, `winBest`, `median` or `parent`
`opp.fitCredDiv`	`children`	Credit divisor: `children` or `trials`

Table E.11: `OperatorPool` parameters

Parameter	Default	Description
`wrks.theK`	`4`	Number of alternative paths, k
`wrks.firstFibreOnly`	`true`	Use only first fibre on each cable?
`wrks.cacheKSP`	`true`	Cache k-shortest paths?

Table E.12: `WaveRandKSP` parameters

Parameter	Default	Description
`wkm.theK`	`4`	Number of alternative paths, k
`wkm.firstFibreOnly`	`true`	Use only first fibre on each cable?
`wkm.cacheKSP`	`true`	Cache k-shortest paths?

Table E.13: `WaveKSPMutate` parameters

Parameter	Default	Description
`wwrr.theK`	`4`	Number of alternative paths, k
`wwrr.firstFibreOnly`	`true`	Use only first fibre on each cable?

Table E.14: `WaveWauReroute` parameters

Parameter	Default	Description
`wwso.theK`	`4`	Number of alternative paths, k
`wwso.firstFibreOnly`	`true`	Use only first fibre on each cable?

Table E.15: `WaveWauShiftOut` parameters

Parameter	Default	Description
`wkco.theK`	`4`	Number of alternative paths, k
`wkco.firstFibreOnly`	`true`	Use only first fibre on each cable?
`wkco.cacheKSP`	`true`	Cache k-shortest paths?

Table E.16: `WaveKSPCO` parameters

Variable	Example Value	Description
`plsMatrix[0][1]`	`{P0(0[0:2],8[0:2],12[0:2],1)}`	Path-loss sequence for source node 0, destination node 1
`cabMatrix[0][2].occup[1]`	`077777`	Occupancy of fibre 1 on the cable between node 0 and node 2

Table E.17: Output file variable-value pairs

E.1.3 NOMaD Output (`.out`) Files

The format of NOMaD output files, usually designated by a `.out` suffix, varies widely between applications. However, it is usual practice to at least output the network design obtained, via the `Network` or `PhyNetwork` `print()` member function, as appropriate.

For network topology design (`Network`), this will result in variable-value pairs for all the entries in Table E.1, plus all the entries in the first part of Table E.2. In addition, the path-loss sequence matrix will be output as a series of variable-value pairs (Table E.17, `plsMatrix`).

For routing, fibre and wavelength allocation (`PhyNetwork`), as well as all the data for a `Network`, the path-loss sequences will now contain wavelength-routed paths (not simple paths). In addition, `net.lambdaCap` and `cabMatrix[?][?].numFibres` (Table E.2) entries will be provided, plus an extra (redundant) set of variable-value pairs (Table E.17, `cabMatrix[?][?].occup[?]`) recording individual fibre occupancies; these are intended for display and consistency use, as extracting the information from the `plsMatrix` entries would require considerable processing, especially at script level.

Beyond the (`Phy`)`Network` related variable-value pairs, `ga` for example (§3.9), outputs data including: (for each generation) average fitness, lowest fitness and operator probabilities; as well as number of operators, operator names, total number of generations, best fitness value for the run, and the generation in which it was achieved.

E.2 Class Cross-references

Tables E.18–E.26 list the main classes used in NOMaD and, for each class, record references to the appropriate descriptions and diagrams from Ch. 3–4, Appendix C and later in this Appendix. Where one figure is the most important for a particular class, its figure number has been shown in bold.

Class	Text (pp.)	Figure(s)
`Network`	56, 253	**3.4(a)**, 3.4(b), 3.5, E.4, E.5, E.8, 4.1, E.13
`Node`	56, 257	**3.4(a)**, E.4, E.13
`Link`	56, 257	**3.4(a)**
`PathLossSeq`	56, 263	3.4(a), 3.4(b), **E.5**
`Path`	56, 260	3.4(a), **E.4**, E.5, E.8

Table E.18: `NetworkModel` category — `Network` classes

Class	Text (pp.)	Figure(s)
`PhyNetwork`	56, 267	**3.4(b)**, E.8, 4.1
`Cable`	56, 268	**3.4(b)**, E.8
`PhyPath`	56, 272	3.4(b), **E.8**
`Join`	275	**E.8**

Table E.19: `NetworkModel` category — `PhyNetwork` classes

Class	Text (pp.)	Figure(s)
`Metric`	67	3.4(a), **4.1**
`AlesCost`	68	**4.1**
`AlesRouter`	68	**E.2(b)**
`PhyNetCost`	72	**4.1**
`NetWaveReq`	71	**4.1**

Table E.20: `NetworkModel` category — `Metric` classes (and `AlesRouter`)

Class	Text (pp.)	Figure(s)
`PlsIter`	280	3.4(b), **E.5**
`PathIter`	278	**E.4**
`PathLinkIter`	279	**E.4**
`CableIter`	280	**3.4(b)**
`PathJoinIter`	280	**E.8**
`PathHeap`	259	

Table E.21: `NetworkModel` category — minor classes

Class	Text (pp.)	Figure(s)
`GeneticAlgorithm`	56, 74	**3.5**
`Population`	56, 76	**3.5**
`Individual`	56, 77	**3.5**
`OperatorPool`	56, 78	**3.5**
`Reproduction`	79	**3.5**

Table E.22: `GA` category classes (and `Reproduction`)

Class	Text (pp.)	Figure(s)
`Operator`	56, 259, 79	3.5, E.2(a), **4.3**, 4.4(a), 4.4(b)
`UnaryOp`	56, 259, 79	3.5, E.2(a), **4.3**, 4.4(a), 4.7(a)
`BinaryOp`	56, 259, 79	E.2(a), **4.3**, 4.4(b), 4.7(b)
`Router`	56, 259	**E.2(a)**
`UnaryRouter`	259	3.4(a), **E.2(a),** E.2(b), 4.6
`BinaryRouter`	259	**E.2(a)**

Table E.23: `Operators` category — abstract classes

Class	Text (pp.)	Figure(s)
`RandomLinks`	80	**4.4(a)**
`MutateLink`	81	**4.4(a)**
`DegreeTwo`	81	**4.4(a)**
`DegreeDec`	81	**4.4(a)**
`RandomAddLink`	81	**4.4(a)**
`RandomRemLink`	81	**4.4(a)**
`RandomChgLink`	82	**4.4(a)**
`RandomAddLinks`	82	**4.4(a)**
`RandomRemLinks`	82	**4.4(a)**
`RandomChgLinks`	82	**4.4(a)**
`SPLinkCO`	82	**4.4(b)**
`LinkGroupCO`	82	**4.4(b)**

Table E.24: `Operators` category — topology classes

Class	Text (pp.)	Figure(s)
`WaveRandKSP`	82	**4.6**
`WaveNop`	84	**4.6**
`WaveWutti95`	229	**4.6**
`WaveWauDijk`	230	**4.6**
`WaveWauHRWA`	230	**4.6**
`WaveNagVWP`	233	**4.6**
`WaveNagWP`	233	**4.6**
`WaveKSPMutate`	84	**4.7(a)**
`WaveWauReroute`	84	**4.7(a)**
`WaveWauShiftOut`	85	**4.7(a)**
`WaveKSPCO`	85	**4.7(b)**

Table E.25: `Operators` category — wavelength-allocation classes

Class	Text (pp.)	Figure(s)
`ArrayList`	281	E.4, E.8
`AlIter`	281	E.4, E.8
`List`	282	E.5, **E.12(a)**
`ListNode`	282	**E.12(a)**
`ListIter`	282	E.5, **E.12(a)**
`ParamDataBase`	284	3.4(a), **E.13**
`Parameter`	284	**E.13**
`RandomGen`	286	

Table E.26: `Utilities` category classes

E.3 Network Modelling

The primary aim of NOMaD's network model was to provide a set of classes capable of directly supporting network design in the application areas outlined in §3.1 above: both network topology design, and routing, fibre choice and wavelength allocation of multi-wavelength all-optical transport networks. In addition, it was hoped that the set of classes provided might be able to form the basis of design work in other areas, such as optical access networks, through derivation [16] and (if necessary) minor modifications to the existing classes. Further, based on the lessons learned in developing the model, it was envisaged that the author, and others, would be able to design even more robust and reusable network modelling class libraries in the future.

E.3.1 Overall Structure

The network modelling classes discussed in this subsection are those of more general applicability, including `Network`, `Node`, `Link`, `Router` and `UnaryRouter`. Those classes which are more specific to particular design problems are described in §E.3.2 and §E.3.3 below.

`Network` class

Arguably the most important class in NOMaD's network model is the `Network`, illustrated in Fig. 3.4(a) on p. 58. Both its data structure and functionality are complex as, in association with other classes, `Network` (or its derived class for wavelength-routed networks, `PhyNetwork`) must be able to represent everything from, for example, a simple 4-node network design problem (simply a traffic matrix and a set of node locations) up to, say, a 20-node or larger multi-wavelength all-optical transport network. Including, in the latter case, the details of all its routing, down to individual fibre and wavelength assignments on every link of a large number of wavelength channels.

The main `Network` constructor [16] takes two parameter database objects: one built from the network specification file, and the other from the NOMaD parameter file (see §3.6 for example code). Large numbers of variable-value pairs (*i.e.* parameters) in the network parameter database are then retrieved as necessary during construction, although a `const` pointer [16] to the network parameter database (`netDB`) is retained for possible further reference (it can also be obtained externally using a member function). However, in contrast, only two parameters are actually taken by the constructor from the NOMaD parameter database (Table E.3); their application is discussed below. (For NOMaD file formats, see §E.1.)

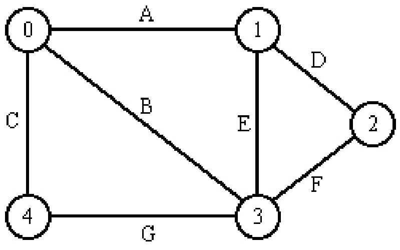
To represent the nodes of the network, such as optical cross connects, the `Network` class contains a count of the number of nodes (`numNodes`), and an array of pointers to the node objects (class `Node`) themselves (`nodes`). These are made available through respective member functions, the latter returning a `const` pointer to a node object (selected by node identification number; id), thus preventing modification by the user of the `Network`'s private node objects [16].

The network traffic matrix (`traMatrix`) is represented by a two-dimensional array of size `numNodes` $\times$ `numNodes` of double-precision floating-point numbers[2]. These represent the traffic originating in each direction between node pairs. Using a source-destination node id pair, individual traffic values can be retrieved with a member function. Example traffic matrices used in some of the network design problems studied in Ch. 5 are given in §F.2.

The network topology (if any) is represented by a count of the number of links (`numLinks`), and an adjacency matrix (`adjMatrix`) [25, 287]. This consists of a two-dimensional array of size `numNodes` $\times$ `numNodes` of pointers to link objects (class `Link`): if a link exists between two nodes, then the corresponding entry in the array will contain a valid link pointer; otherwise it is void. Fig. E.1 illustrates the concept for a small 5-node network with 7 links where, in the adjacency matrix, link letters are shown (rather than pointers!). Note that, as the links are bidirectional, they each appear in two places in the matrix (*e.g.* link D connects node 2 to 1, as well as node 1 to 2). In addition, as nodes are not permitted to link to themselves ('self loops'), the diagonal is empty. Although in the network model only one link is allowed between node pairs (*i.e.* no parallel links), support for multiple-fibre connections is provided by the `Cable` class for wavelength-routed networks: see §E.3.3 below. Within NOMaD, two adjacency matrices are actually used: `adjMatrix` is the 'true' adjacency matrix, whereas the *potential* adjacency matrix (`potentialLinks`) stores pointers to all possible links (*i.e.* all those that would be required in a full-mesh superset of the current network). By simply copying pointer values from the potential to the actual adjacency matrix when a link is added to the network topology, the time overhead of `Link` construction (and when a link is removed, destruction [16]) is eliminated. This particular use of the adjacency matrix, combined with the need to access link objects efficiently via their end-node ids, was the primary reason for the selection of a (node-node) adjacency matrix representation for the network topology, rather than the equivalent node adjacency list [25, 287] or node-link incidence matrix [25, 292].

Users can obtain the number of links using a member function. A `const` pointer to individual

[2] Since the traffic matrix is dynamically allocated, as are most objects in NOMaD, its structure is not simply that of a two-dimensional array, but a more complex pointer based one; however, the exact details need not concern us here.



(a) Network

From/To	0	1	2	3	4
0		A		B	C
1	A		D	E	
2		D		F	
3	B	E	F		G
4	C			G	

(b) Matrix

Figure E.1: Example adjacency matrix

links can be retrieved either using the link id, or the node ids of the two end nodes; the latter option also permits the adjacency matrix to be queried, as a void pointer is returned if two nodes are not connected. To allow modification of the adjacency matrix, member functions to add and remove links are provided, with links specified by their end-node ids; the former permits link connect levels to be supplied as optional parameters if an hierarchical network is being constructed (see below). Further, although a node's degree (*i.e.* number of attached links) could be determined by querying the adjacency matrix, a member function is provided to automate this for a specified node id.

NOMaD's network model directly supports the concept of network hierarchy, by enabling nodes to be on one or more levels, and links to be connected to a particular level at each end. In addition, as an aid to the design of cluster-based architectures, such as multi-ring networks, the network model provides direct support for node clusters; these are also intended to be useful in the development of any cluster-based operators or overall design heuristics. However, neither of these capabilities have been used in the experimental work reported in Ch. 5, although they have been utilised elsewhere[3]. Further details on NOMaD's support for network hierarchies and node clustering is given in §E.4.

The static routing of traffic through the network is represented using either simple paths (`Path`) or wavelength-routed paths *i.e.* paths that also record fibre choice and wavelength allocation (`PhyPath`). The former are used in `Network`s for network topology design; the latter in wavelength-routed networks (`PhyNetwork`s) for the routing, fibre and wavelength allocation problem. For each source-destination node pair in the network, the associated path(s), of the appropriate type, are listed in a path-loss sequence [286] (`PathLossSeq`): an ordered sequence of simple or wavelength-routed paths. These in turn are recorded in the network's path-loss sequence matrix (`plsMatrix`), a two-dimensional array of size `numNodes` $\times$ `numNodes` of pointers to path-loss sequence objects. Al-

[3]Specifically, by Noel Parnis for OPEN.

though not enforced by NOMaD, the author has adopted the convention that for network topology design (class `Network`), each of the path-loss sequences records just two simple paths, representing the primary and restoration routes, whereas for wavelength allocation (class `PhyNetwork`), each wavelength-routed path in a sequence represents a single wavelength channel of the primary route (and restoration routes are not included). The details of the `Path`, `PhyPath` and `PathLossSeq` classes are given in the appropriate sections below (§E.3.2 and §E.3.3), where `Network`'s member functions for the manipulations of paths, path-loss sequences and the path-loss sequence matrix will also be discussed.

Although it is possible to assess a network object against a particular metric (*e.g.* cost or network wavelength requirement) by applying the appropriate metric object (derived from the abstract `Metric` class, see §4.1), network objects also store a pointer to a metric object to allow self-assessment (`metric`). Further, as (re-)routing may need to be carried out prior to the assessment of a design (although see `WaveNop` in §4.2.3), the network object also stores a pointer to a unary router object (`router`). Thus, when the `Network`'s assessment member function is invoked, first the router's routing member function is used to reroute the network, and then the assessment itself is carried out by using the metric's assessment member function. The user's choices of which metric and router to use are made via the NOMaD parameter file: see Table E.3 on p. 247. To avoid unnecessary re-assessment however, the network object monitors internally whether any changes have been made to its topology or routing; if not, then the last assessment result, which will have been cached within the network object, is returned instead; no actual rerouting or assessment is carried out. Given that assessing a network design is often computationally intensive, this approach can therefore lead to considerable savings in processing time. Nevertheless, cacheing of network assessments in this way has implications for performance monitoring of GAs in terms of trials (*i.e.* number of assessments), and is thus also discussed in §4.2.

The output of networks can be carried out with either of two member functions: one of these writes network data in variable-value pairs on an output stream (`ostream` [16]) *i.e.* a file, whereas the other returns the data in a dynamically-allocated parameter database.

In addition to those aspects of `Network`'s data structure and functionality already presented in the preceding paragraphs, where it has been judged appropriate, description of several further elements has been included in the discussion of other classes within the network model. For example, in support of routing, the `Network` class provides several member functions to allow 'link marking'; however, rather than here, this concept will be discussed below with reference to the `Link` class.

`Node` class

The `Node` class is not a 'true' class, in the sense that it cannot exist independently of a particular network object. However, in this, NOMaD's network model reflects reality, because nodes do not exist in isolation, but only as part of a network. Although usually used to represent an optical cross connect (OXC) by the author, the `Node` class itself is not specific to any particular technology—it does not model any internal structure.

The main `Node` constructor is private, and so can only be invoked by the `Node`'s one friend class: `Network`. The constructor is supplied with a network parameter database, a pointer to the invoking network object, and the node's id: an integer unique to this node within the network. Both a `const` pointer to the node object's parent network and its own id are stored internally, and can be accessed through the appropriate member functions.

From the supplied parameter database, the `Node` extracts its (`x`, `y`) location on a Cartesian plane, its name, levels and clusters; all of which are stored internally and can be retrieved via member functions. (A description of levels and clusters is given in §E.4.) The (`x`, `y`) location is in arbitrary units and is stored as two double-precision floating-point numbers. However, the parent network object stores a variable (`kmPerUnit`) from the network parameter database, which scales node coordinates to km, and can be obtained using the appropriate `Network` member function (if not provided, it defaults to 1 *i.e.* units are simply km). In addition, given two node ids, `Network` provides a member function that will calculate the Euclidean distance in km between them. The node's name is an arbitrary string; if one is not provided in the parameter database, it defaults to `"Node<id>"`, where `<id>` is the node's id.

As with `Network`, the `Node` class provides member functions for output either to file (output stream), or to a specified parameter database; both are invoked, as appropriate, by the `Network` class to output its own nodes.

`Link` class

As was the case for nodes, link objects are entirely dependent on their parent network, and never exist in isolation. Once again, and perhaps even more than with nodes, this reflects reality!

The main `Link` constructor is private; however, it can still be invoked by the parent network, as `Network` is a friend of `Link`. The constructor takes only two arguments: a `const` pointer to the parent network object, and the length of the link in km. Both are stored by the link object, and can be accessed via member functions. However, a parameter database argument is not required, as all the detail on link connect levels this would have supplied is instead retained by the parent network. The link length, provided by the parent network, is either retrieved by it from the network parameter database, or if not present, calculated (as the Euclidean distance between the end nodes) using the `Network` class' own inter-nodal distance member function.

In addition to data supplied and stored at its construction, the `Link` class also supports link capacity, link weight and link marking. Link capacity, stored as a double-precision floating-point number, is set directly by the network object as part of its handling of network routing: the link capacity is simply the sum of the capacity of all the paths (either simple or wavelength-routed, as appropriate) that make use of it; the current value can be retrieved via a member function. Further, as the network model takes node capacity to be the sum of the capacity of all a node's attached links, `Network` provides a node capacity member function which will calculate this value for a given node id.

Link weight supports the network model's concept that routing need not necessarily be done only with respect to link length. Instead, although at construction its weight and length are equal, a link's weight can be reset to an arbitrary value using either of two `Network` member functions: one selecting the link by its id[4], the other using a node id pair. All length-based routing in NOMaD actually uses these link weights, retrieved via a member function, rather than the actual link length[5].

To support network routing, the ability to 'mark' links is provided. Each link object stores a Boolean value (`marked`), initially `false`, which can be queried by a member function. When a link is marked, NOMaD's routing functions ignore the existence of the link: it is as if it is no longer part of the network topology. This provides an invaluable aid to finding *e.g.* link-disjoint [292] or k-shortest paths through the network [322]. Link marking employs several `Network` member functions: links can be marked by choosing them using either their own id or the ids of their end nodes; nodes can be marked, selected by their id (node marking is equivalent to marking all the node's attached links); and both simple and wavelength-routed paths can be marked, equivalent to marking all but their first and last nodes (or the single link on a 2-node path). Finally, the marking of all links can be reset by calling the appropriate `Network` member function.

[4]Although link objects do not store an id, a network can determine a unique id for each one on the basis of their end-node ids; however, their ids are necessarily changed if the network topology alters.

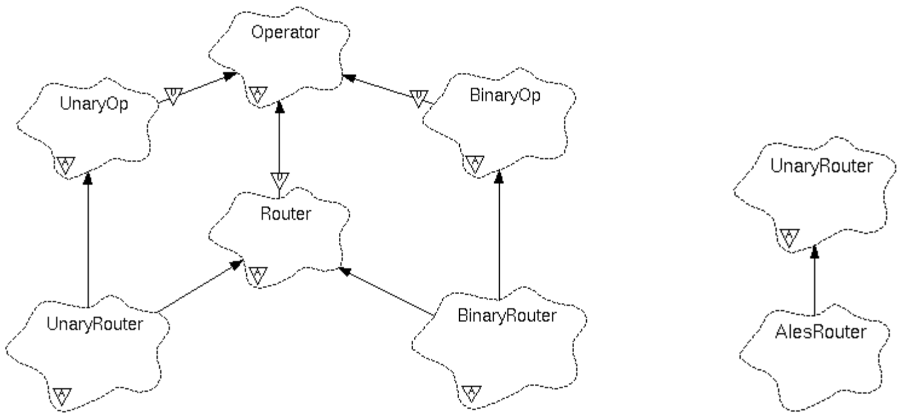
[5]Link weights are used, for example, in the `AlesRouter` class, discussed in §4.1.1.

`Router` and `UnaryRouter` classes

As was mentioned in §3.5 above, (genetic) operators in NOMaD are all derived from the abstract virtual base class `Operator` [16, 17]. To support both unary and binary[6] operators, *i.e.* operators that can be applied to either one or two network objects respectively, two further abstract classes were derived from `Operator`: `UnaryOp` and `BinaryOp` (see Fig. E.2(a) and §4.2). Although it was envisaged that operators would include both those for routing, fibre and wavelength allocation, as well as network topology design, it was felt important to distinguish the former operators from classes able to completely route a network (rather than just modifying an existing routing). These overall routing heuristics would therefore be derived from another abstract class, `Router`, but because its own derived classes were still to be considered operators, `Router` was itself derived from `Operator`. An additional complication arose because, by analogy with `BinaryOp`, it was anticipated that some routers might be developed that would route two networks, rather than just one, in a single operation. Consequently, abstract classes `UnaryRouter` and `BinaryRouter` were developed, derived from `UnaryOp` and `Router`, and from `BinaryOp` and `Router`, respectively. However, no concrete classes have been developed that derive from `BinaryRouter`, and it therefore seems likely that the development of `BinaryRouter` was an unnecessary complication. The design of `UnaryRouter` would appear to be more useful though, as classes derived from it are both unary operators (*i.e.* able to be applied to one network) and routers. Concrete classes derived from `UnaryRouter` include `AlesRouter` (see Fig. E.2(b) and §4.1.1), used in network topology design, and several wavelength-allocation routers discussed briefly in §4.2.3 and in more detail in Appendix C.

In addition to its role in the class hierarchy, the `Router` class was also used to make public several routing functions required by other classes. These include implementations of: Dijkstra's shortest-path algorithm [322] (between a pair of nodes, or between one node and all others); Yen's k-shortest-path algorithm [320] (for a pair of nodes, and which uses the `PathHeap` class); modified versions of Dijkstra and Yen that support Wauters & Demeester's HRWA algorithm [306] (§C.2); and the author's own simplistic algorithms for the k-lowest-hop-count paths and the minimum-number-of-hops paths (both for pairs of nodes).

[6]Binary operators are an important component of GAs.



(a) `Router` (b) `AlesRouter`

Figure E.2: Router class diagrams

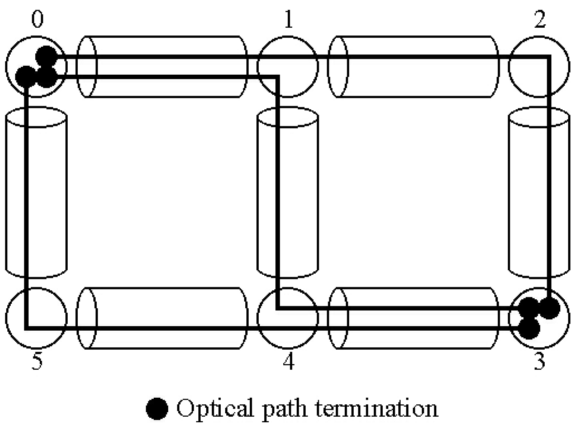
E.3.2 Mesh Network Topology Design

None of the classes in NOMaD's network model are entirely specific to network topology design. Nevertheless, two classes will be discussed in this context: `Path` and `PathLossSeq`. While `Path` is of most use in network topology design, it is also the base class of `PhyPath`, which is used for wavelength allocation. Further, given this relationship between the two path classes, `PathLossSeq` objects can be used to maintain ordered lists of either type. For descriptions of the related iterator classes, `PathIter`, `PathLinkIter` and `PlsIter`, see §E.5.

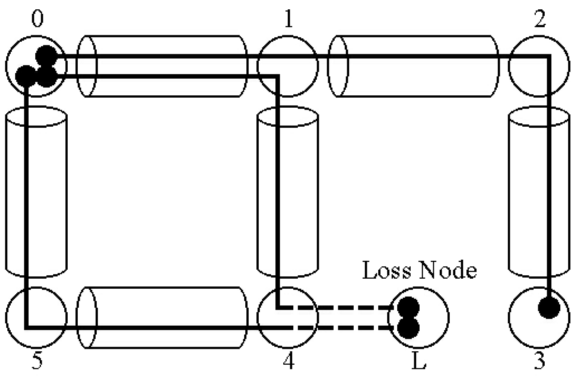
The path model used here (including the *loss node* and *path-loss sequences*) was first developed by Lin *et al.* [286], and subsequently used by the author, both in an M.Sc. project [323] and several research papers [28, 324, 325].

`Path` class

Paths are modelled in NOMaD as an ordered series of nodes, which is exactly equivalent to an ordered series of links, given the exclusion of parallel links from the network model. In addition, paths are not allowed to contain cycles *i.e.* a node may only appear *once* in a path. Fig. E.3(a) of a 6-node, 7-link network (with ids given adjacent to the nodes) shows three paths, all running between nodes 0 and 3. Taking node 0 as the source node, these can be represented as strings: `"(0,1,2,3)"`, `"(0,1,4,3)"` and `"(0,5,4,3)"`. For those cases where a path is unable to reach its intended destination (*i.e.* it is blocked), Lin *et al.* [286] introduced a fictitious loss node. This is shown, for example, in Fig. E.3(b), where link (4,3) has been removed so that two of the paths can no longer reach node 3. The three paths are now: `"(0,1,2,3)"`, `"(0,1,4,L)"` and `"(0,5,4,L)"`, where `L` is used to represent the loss node. It is thus possible to distinguish `"(0,5,4)"`, a *success path* from node 0 to node 4, from `"(0,5,4,L)"`, a *loss path*.



(a) Paths



(b) Success/loss paths

Figure E.3: Example paths and path-loss sequences

The main `Path` constructor takes only two arguments: a `const` pointer to the parent network object and the path capacity, which is passed as an optional [16] argument that defaults to zero. These are stored, although the capacity can be set to another value later; both can be accessed by member functions. Associating paths with a parent network enables considerable integrity checking, although at a cost in flexibility. For example, (apart from topology changes), NOMaD paths only consist of series of nodes that are actually connected by links in the parent network. However, a path constructed as above is *empty*, since an actual node series has not been specified. Thus, a variety of member functions are provided to extend a path, empty or otherwise (see below), but an additional constructor is also available. This takes as arguments the parent network, a string specifying the path, and (optionally) the path capacity. In the event that the string provided to this `Path` constructor also includes fibre choice and wavelength allocation (§E.3.3), a simple path will still be constructed; this also applies when a simple path object is either set from a string or input from a file.

To implement paths, the `Path` class (Fig. E.4) uses a dynamic array (an array-list, class `Array-List`, §E.6) to store an ordered set of pointers to node objects. This representation, rather than simply storing node ids, was chosen to allow more rapid access to the node objects themselves, but requires more storage space. In addition, the path also stores both its length and weight, equal to the sums of the lengths and weights, respectively, of all the links along the path. The lengths and weights are kept up-to-date by the class as the actual path changes, and can be retrieved via member functions. To enable straightforward derivation, all of the `Path` class data members are merely protected, rather than private [16]; `PhyPath`, derived from `Path`, thus has direct access to them (see §E.3.3 below).

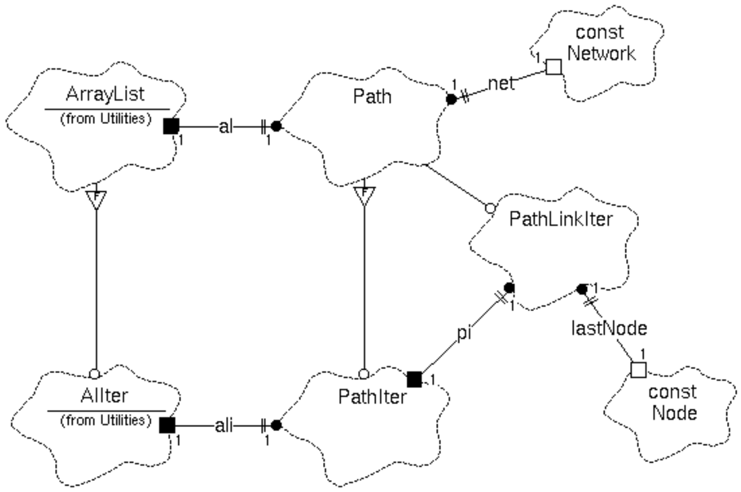


Figure E.4: `Path` class diagram

Many of the `Path` class member functions are made available in two forms: as an overloaded operator [16] or as an ordinary member function. For example, a `const` pointer to one of the node objects on a path (identified by its index number along the path) can be retrieved using either the `getNode()` member function or the overloaded '`[]`' operator (*e.g.* `pp[2]` would retrieve the third `Node`—counting from 0—along `Path pp`, by invoking the `operator[]()` member function with argument `2` [16, 17]).

To simplify both the interface and implementation of paths, they can only be set (from a string), cleared (*i.e.* emptied) or extended—there is no provision for removing nodes. Hence, as long as a path is not a loss path (which can itself be tested with a member function), it can be extended by appending a single node. However, if this is not to be the first node on the path, there must be a link in the network adjacency matrix between the two nodes concerned! Examples of the addition of nodes to paths using both the `append()` member function and the overloaded '`+=`' operator are shown in §E.7. Note that to convert a success path to a loss path, the loss node itself must be appended: this is a special node object stored by each network, that can be retrieved with the appropriate member function. Further, not only paths as a whole, but also individual node objects can be queried to see if they are the loss node for their network.

As well as extending by just one node, a copy of an entire path can be appended to another, using either the `append()` member function[7] or overloaded '`+`' . Amongst the requirements imposed, the path appended to must be both a success path and must end with the same node the other path begins with. Thus, `(0,5,4) + (4,3)` would be valid in the network of Fig. E.3(a), with the result `(0,5,4,3)`, whereas `(0,5,4) + (2,3)` would not. Nevertheless, if the two paths come from different, but compatible (*i.e.* having the same number of nodes) networks, it is still possible to append one to the other: the copy would be mapped to the other network, although it (and hence the resultant path) would be truncated to a loss path if the topology could not support it[8]. Hence, for path `(1,2)`, valid in the network of Fig. E.3(b), adding `(2,3,4,5)`, although valid in network (a), would result in `(1,2,3,L)`, as link (3,4) is missing in network (b). Note that, when two path objects are combined with `append()`, the first is modified by the addition of a copy of the second; however, when '`+`' is used, neither path is actually altered, but instead a third, new path object is returned, so as to preserve the expected semantics of the '`+`' operator [16].

Several functions are provided for accessing the individual nodes along a path. As well as `getNode()` and '`[]`', mentioned above, both the first and the last nodes can be retrieved with particular member functions. A path can be queried as to whether it contains a given node object; whether it is empty or not; and what size it is (*i.e.* how many nodes it contains). In addition, two paths can be tested for equality (using either `sameAs()` member function or overloaded '`==`' operator), for both identical and compatible parent networks. A further member function tests, in a similar way, for equality up to a given number of nodes along the paths.

For file input/output (`iostream` [16]), friend functions are provided overloading '`>>`' and '`<<`', respectively. These in turn invoke virtual private member functions [16], ensuring that classes derived from `Path` (and `PhyPath` in particular) will handle input/output correctly.

`PathLossSeq` class

A path-loss sequence [286] is an ordered series of paths; however, all its paths must share the same source, and all the success paths the same destination node. Thus `"{P0(0,1,2,3), P1(0,1,4,3), P2(0,5,4,3)}"` is a valid path-loss sequence (Fig. E.3(a)), as is `"{P0(0,1,2,3), L0(0,1,4,L), L1(0,5,4,L)}"` (Fig. E.3(b)), but `"{P0(0,1,2,3), P1(3,4,1,0), P2(0,5,4,3)}"` is not. Within the string representation of a path-loss sequence, the success and loss paths are indexed separately (from `0`), and their index, along with a `P` or `L` respectively, is prepended to the path string itself.

[7] In C++, which function to invoke is determined by its arguments as well as its name; as a result, it is possible for `Path` to have two different `append()` member functions.

[8] This possibility, like most actions involving two networks, has been included to support GAs

This is mainly an aid to the user, however, and may be omitted, *e.g.*"{`(0,1,2,3), (0,1,4,3), (0,5,4,3)`}" is also valid.

Given that a path is an ordered series of nodes and a path-loss sequence is an ordered series of simple (or wavelength-routed) paths, a good measure of similarity was maintained between the two class interfaces; nevertheless, some differences between the designs did arise. As with paths, both overloaded operator and ordinary member functions were provided wherever appropriate.

One difference in particular between paths and path-loss sequences is of note. NOMaD's network model regards node objects as 'owned' by their parent network; each node object has a unique id, and nodes are never copied (they have no copy constructor[9] [16]). Consequently, a path object is seen primarily as representing a sequence of nodes: it owns the *sequence*, but not the node objects themselves. On the other hand, it was anticipated that paths and path-loss sequences would need to be able to exist 'outside' a network, although their validity is checked against a particular network. This allows the user to manipulate paths and path-loss sequences before 'giving' them to a network (see below). As a result, a path-loss sequence owns the path objects in its sequence: they are constructed by the path-loss sequence directly from a string, or as copies of 'external' path objects. In addition, the path-loss sequence's own path objects are only accessible to users via `const` pointers. These measures preserve the integrity of the path-loss sequence, as otherwise paths in its sequence might be altered without its knowledge!

As has been mentioned, path-loss sequences can contain both simple and wavelength-routed paths. However, the design decision not to introduce a separate path-loss sequence class for wavelength-routed paths has introduced some inelegance and lack of type safety. Although simple and wavelength-routed path objects should never be used together in the same sequence, the `PathLossSeq` class does not enforce this. In addition, at certain points, only the user can determine that a sequence contains objects of class `PhyPath`, and is required to cast them back from `Path`. However, the decision was taken to avoid both introducing an additional (almost) identical class to `PathLossSeq`, as well as the resulting need to develop the `Network` class' `plsMatrix` from an implementation class [283] to two full classes, so as to contain the matrix of existing and new path-loss sequence objects for networks and wavelength-routed networks, respectively.

`PathLossSeq`'s main constructor takes just a single argument: a `const` pointer to the parent network; this is stored, and can be retrieved with a member function. As with paths, such a path-loss sequence is empty, however, and so another constructor is provided which also takes a string

[9] Actually, they *do* have a copy constructor, but it is private, and fails an assertion if invoked!

argument, specifying the pass-loss sequence.

Rather than using a dynamic array, as do paths, path-loss sequences use a circularly-linked list (a `List`, §E.6) to store their sequence of pointers to path objects (Fig. E.5). Although a linked-list requires more processing than an array-list to retrieve an individual object, it is far more efficient when objects are removed: an operation which path-loss sequences, unlike paths, would need to support.

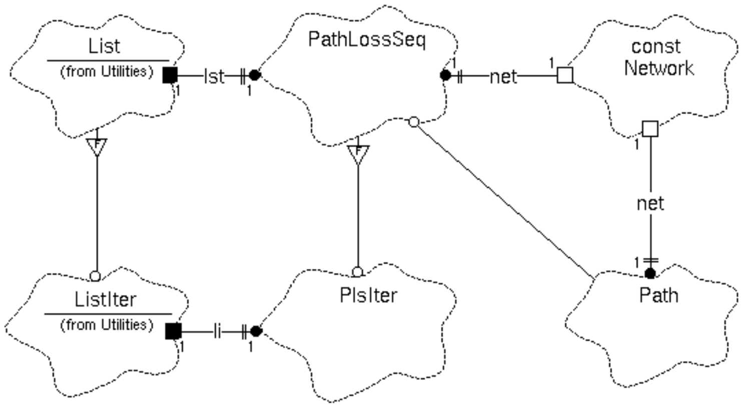


Figure E.5: `PathLossSeq` class diagram

As well as setting a path-loss sequence from a string and clearing (*i.e.* emptying) it, the sequence can be extended, by appending either a path or a whole path-loss sequence, and individual path objects can also be removed. Single paths are appended using either `append()` or '`+=`'. Each path appended must be non-empty. After the first path has been appended, all subsequent paths must be for the same source, and if not a loss path, share the same destination. They can either be from the same network as the path-loss sequence, or a compatible one *i.e.* one having the same number of nodes. (Although, in the latter case, any copy of a path that was unsupportable by its parent network topology would be truncated to a loss path.) The same set of restrictions apply when a whole path-loss sequence is appended using either the overloaded `append()` member function or '`+`' operator.

The paths in a sequence can be accessed as `const` pointers using a member function. As this is an expensive operation for the underlying linked-list, no equivalent overloaded operator function is provided (*cf.* `Path`'s '`[]`' operator). However, the first and the last paths can be retrieved using efficient special-case functions. The source and destination nodes can also be obtained (as `const` pointers) using member functions: the latter simply returns the loss node if all the paths in the sequence are loss paths. A path-loss sequence can be queried to see if it contains a given path object; whether it is empty or not; what size it is (*i.e.* how many paths it contains); and whether all its paths are loss paths or not. In addition, as with individual paths, two sequences can be tested for equality, either for identical or compatible parent networks, and equality can also be tested up to a given number of paths along the sequences.

For input/output, friend functions are provided overloading '`>>`' and '`<<`', respectively. By invoking the equivalent overloaded operators in class `Path`, which in turn invoke virtual private member functions, input/output is handled correctly whether the sequence contains either simple or wavelength-routed paths.

At this point, having reviewed some of the details of the `Path` and PathLossSeq classes, it is appropriate to examine the member functions provided by the `Network` class in support of network routing. The `Network` class' `plsMatrix` of path-loss sequences between source-destination pairs was described above, but how are these sequences set and manipulated?

One member function enables the path-loss sequence between a given source-destination node id pair to be set. Any existing sequence between the pair is deleted, and a copy of the new sequence is installed in the path-loss sequence matrix. The new sequence must be for this network, be non-empty, and apply to the correct source-destination pair. For a given pair, other member functions allow a `const` pointer to a path-loss sequence to be retrieved from the network, or the sequence explicitly removed (*i.e.* deleted). As an aid to efficiency, a member function is further provided to completely clear a network's path-loss sequence matrix (clearly of particular use in routers). In addition to whole sequences, two member functions allow individual paths to be appended to or removed from the sequence for a given source-destination pair. Where necessary, in all these functions and as was mentioned above, the capacity of the network's links is maintained to always reflect the sum of the capacities of the paths carried. However, where a simple path has a capacity of zero, its actual capacity is simply taken to be that of the traffic between the source-destination pair that form its end nodes[10]. Further, to ensure the cable, fibre and wavelength structure of wavelength-routed networks can also be appropriately adjusted during routing modifications, these member functions are both declared virtual [16] and re-implemented in the `PhyNetwork` class (see below).

[10] For a wavelength-routed path, however, its capacity is always uniquely determined by the number of wavelengths and their granularity.

E.3.3 Routing, Fibre & Wavelength Allocation

This subsection presents those classes specific to the routing, fibre choice and wavelength allocation problem: `PhyNetwork`, `Cable`, `PhyPath` and `Join`. (For descriptions of the related iterator classes, `CableIter` and `PathJoinIter`, see §E.5.) Through these classes, the NOMaD network model supports multiple-fibre cables, and both virtual-wavelength-path (VWP) and wavelength-path (WP) routing [294].

`PhyNetwork` class

The `PhyNetwork` class[11] is derived from `Network`, adding (and redefining) those elements necessary to allow the representation of the cable, fibre and wavelength-routed structure of a multi-wavelength optical network (see Fig. 3.4(b) on p. 58, and Fig. E.8 on p. 273 below).

As in `Network`, the main `PhyNetwork` class constructor takes two parameter database arguments, one built from the network specification and the other from the NOMaD parameter file. After constructing the base `Network` class, two optional parameters (the default number of fibres per cable and the wavelength channel capacity *i.e.* granularity) plus the cable matrix are retrieved from the network parameter database. The wavelength channel capacity (or its default value) is stored as a double-precision floating-point number (`lambdaCap`), and may then be accessed via a member function. It should be noted that, since there is only a single granularity value for the whole network, NOMaD's network model does not support the mixing of wavelength channels of different capacity.

The cable matrix (`cabMatrix`) consists of a two-dimensional array of size `numNodes` × `numNodes` of pointers to cable objects: if a cable exists between two nodes, then the corresponding entry in the array will be a valid cable pointer; otherwise it is void. As with links (§E.3.1 and Fig. E.1 on p. 255), cables are bidirectional, and so any cable object pointed to by the matrix appears in two entries: (i, j) and (j, i). In addition, as 'self loops' are not allowed, the diagonal is empty. The network parameter database determines whether a cable will actually be constructed for any particular node pair. If an overall default number of fibres per cable has been specified, then a cable is constructed for *every* node pair. In this case, the number of fibres used for each cable will be the default given, unless the database contains a different value for that node pair. However, if no overall default is given, then a cable will only be constructed for those node pairs where the number

[11] Originally described as a *physical* network by the author, hence `PhyNetwork`, the name wavelength-routed network is now preferred; the class name persists, however.

of fibres is specified in the database. In addition, the constructor requires that for every node pair connected in the network topology (*i.e.* for which there is a link in the adjacency matrix), there must also be a supporting cable in the cable matrix. Consequently, if a cable has been constructed for every node pair, then arbitrary topology changes can be made by adding and removing links. However, if not, then links can only be included between node pairs where the potential has been indicated by providing a cable. An entry in the cable matrix can be retrieved, as a `const` pointer to a cable object, using either a link id[12] or the end-node ids; clearly, in the former case not all cables may be able to be retrieved, but those not corresponding to links are not part of the 'real' network topology anyway.

Many of the functions in the `PhyNetwork` class' public interface are simply redefinitions of the `Network` class' virtual member functions. Thus, for example, link addition is redefined to allow for the further requirement that any new link be supported by a cable. Those functions that support routing are necessarily altered: setting or removing path-loss sequence matrix entries, plus adding or removing paths (actually wavelength-routed paths) from entries. Clearly, output to both parameter database and file also had to be changed.

The final group of member functions relates to fibres and wavelengths over the whole wavelength-routed network; these are best dealt with below, in the context of the `Cable` class.

`Cable` class

In addition to `PhyNetwork`, several classes could have been designed to represent a multi-wavelength network's cable, fibre and wavelength-routed structure: individual classes for cables, fibres and even wavelengths. However, any classes designed for fibres or wavelengths would have required only limited functionality, and for efficiency reasons, would have had to allow access to their data members by the cable class. Consequently, it was decided that a single class, `Cable`, would provide the data structure and functionality to support all three concepts.

In a wavelength-routed network, an individual cable has a number of fibres, each of which will support several wavelengths. Each potential wavelength is either occupied by a wavelength-routed path or unoccupied. For example, Fig. E.6 shows a 3-fibre cable, with three occupied wavelengths (*i.e. light* wavelengths) on fibre 0 (*i.e.* wavelengths 0, 1 and 3), five occupied on fibre 1, and four on fibre 2. If we assume each fibre can support up to 32 wavelengths, say, then there are 29 unoccupied (*i.e. dark* wavelengths) on fibre 0 (*i.e.* wavelengths 2 and 4–31), 27 on fibre 1, and 28 on fibre 2.

[12]See footnote 4 on p. 258

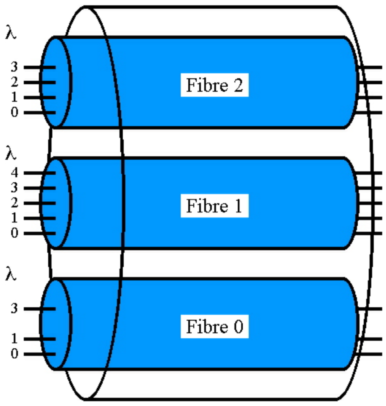


Figure E.6: Example cable of three fibres

As with nodes and links, cables are not intended to exist in isolation, but to be constructed and used by a wavelength-routed network (Fig. E.8 on p. 273). In addition, while cables necessarily store pointers to wavelength-routed paths, these are not owned by the cables themselves, but by their parent wavelength-routed network.

The main `Cable` constructor takes one argument, the number of fibres on the cable; this can subsequently be retrieved using a member function. After construction, although the number of fibres on a cable cannot be varied, this does not exclude the use of the network model to determine the number of fibres actually required by a cable. As long as a generous number of fibres has been preset, after routing, fibre and wavelength allocation is completed, dark fibres (*i.e.* those that are completely unoccupied) on each cable may be excluded from any assessment of the design.

As well as storing the number of fibres (`numFibres`), the data structure of a cable consists of an array of bit vectors (32 bits each) to store the occupancy of each fibre[13], as well as a separate occupancy matrix. The latter is a `numFibres` $\times$ 32 array of pointers to wavelength-routed paths: each row corresponding to a fibre, and each column a wavelength, allowing the individual wavelength-routed path using each occupied wavelength to be recorded. For example, the occupancy vectors (in octal) for the cable in Fig. E.6 would be `013`, `037` and `017` for fibres 0–2, respectively. The use of both these data structures is, of course, redundant: the occupancy vector for an individual fibre could be constructed simply by noting which wavelengths are occupied in a given row in the matrix (unoccupied wavelengths are indicated by a void pointer). However, reconstructing the occupancy vectors when required was judged to be more costly in processing time than maintaining them as a separate data structure (and incurs only a modest additional space requirement). The decision

[13] As with levels and clusters (§E.4), restriction to this range allows an unsigned integer [16] to be used as a bit vector (32 bits); a greater range would require the use of a true bit-vector class.

to limit the number of wavelengths per fibre to 32 was taken on the grounds of both efficiency and memory size[14]. In the event that a user wishes to enforce a lower limit, this can be achieved with appropriate constraints in the routers (or individual operators) used.

An additional restriction placed on the design of the `PhyPath` class, and which thus impacts all the other classes in this section, was that while a wavelength-routed path could make use of one or more wavelengths on each cable, these must all be on a single fibre. This decision allows the occupancy of a wavelength-routed path on a cable to be expressed as a fibre id plus an occupancy vector (32 bits) indicating the wavelengths used: a concept captured in the `Join` class (see below). Whereas, if the occupancy of a wavelength-routed path (and hence the underlying structure of the `Join` class) was to have been expressed as an arbitrary list of fibre id and wavelength number pairs, this would have had serious implications on both processing time and storage size. Limiting each wavelength-routed path to use only a single fibre on each cable was not seen as a serious limitation, in any case, as a sequence of such paths would allow any arbitrary routing to be expressed between a source-destination node pair. Indeed, in the author's own work, wavelength-routed paths were entirely restricted to just a single wavelength over a single fibre on each cable (and were thus equivalent to wavelength channels); where multiple channels were required, a sequence of separate wavelength-routed paths was used.

A wavelength-routed path can be placed on a cable using a member function; as well as the wavelength-routed path object itself, a join object must be provided to indicate the fibre id and wavelength number(s). Clearly, the new wavelength-routed path must only be added to a set of unoccupied wavelengths on the given fibre. To remove a wavelength-routed path, and free the associated wavelengths, there are two member functions. One of these, taking a pointer to a wavelength-routed path, removes it and returns a join object recording the fibre id and wavelength(s) the wavelength-routed path used to occupy. The other, given a join object, removes and returns the corresponding wavelength-routed path; however, it requires that the join object provided specify exactly the fibre and wavelength(s) occupied by a particular path. As well as the removal of individual wavelength-routed paths, the `Cable` class also has a member function to remove them all.

Several member functions allow a cable object to be queried. One checks if a given join object (*i.e.* fibre id and wavelength number(s)) is unoccupied. With another two, either the individual

[14] This is of particular importance for wavelength-routed paths, which represent a high proportion of the total space requirements for a wavelength-routed network object.

wavelength-routed path (if any) corresponding to a particular join object can be retrieved, or even a particular entry in the occupancy matrix. Further, whether a particular wavelength-routed path is carried on the cable can be checked, or the details retrieved as a join object. As well as queries relating to individual wavelength-routed paths, the occupancy of a given fibre, as a whole, can be accessed as a bit vector, or even as an array of wavelength-routed path pointers. In addition, the occupancy of the entire cable can be obtained as a bit vector, where each bit is only set if the corresponding wavelength is occupied on *all* its fibres.

A function is provided to count the (partially) occupied, completely unoccupied, or all fibres on the cable. In the same way, the wavelengths on an individual fibre can be counted; or the wavelengths on the whole cable, either counting the distinct wavelengths used between all fibres or the wavelength channels used in total. As an example, in Fig. E.6 above, for fibre 0 there are 3, 29 and 32 wavelengths, for light, dark, or in total, respectively. Now, for the cable as a whole, considering just the distinct wavelengths used, gives 5, 27 and 32; whereas we get 12, 84 and 96 by simply counting across all fibres, for light, dark, or total channels, respectively.

As an aid to assessing wavelength requirements (*i.e.* the total number of distinct wavelengths required, including dark ones), member functions are available to determine the highest wavelength number used on either a given fibre, or the cable as a whole[15]. Further, to support VWP routing [294], the `Cable` class supplies a packing member function. This re-packs the wavelength-routed paths on a cable to use an unbroken sequence of wavelength numbers, thus minimising their wavelength requirement. The actual modification to any wavelength-routed path involved is actually achieved by invoking its own packing member function, which should only be used in this context. As an example, the cable in Fig. E.6 could be re-packed so that the wavelength requirement on fibre 0 was only 3 (it is currently 4), by moving the path on wavelength 3 down to wavelength 2.

For the `PhyNetwork` class, as was mentioned above, a number of member functions are provided summarising information that could be obtained by iterating over all cables in the network topology. One function obtains, as required, a count of (partially) occupied, completely unoccupied, or all fibres in the wavelength-routed network; two more, similar counts for the network wavelengths (counting each wavelength used on more than one fibre/cable either once, or as appropriate); and yet another returns the highest occupied wavelength number in the whole network. Further functions are even defined to return the first, or all the wavelength-routed paths actually using the highest

[15]In NOMaD, the highest wavelength number is always exactly one less than the corresponding wavelength requirement, as wavelengths are numbered from 0, whereas the wavelength requirement is a *count* of the total number of distinct wavelengths needed.

wavelength number.

`PhyPath` class

In the NOMaD network model, the wavelength-routed path[16] is a straightforward extension of the path concept described in §E.3.2 above. Where a path is an ordered sequence of nodes, with the links implicit, a wavelength-routed path also specifies the fibre and wavelength(s) used on each cable (link) along the path. However, as has already been discussed, while several wavelengths may be used on each cable, they must all be on the same fibre. Also, though a different *set* of wavelengths may apply on each cable (in VWP routing), the actual *number* of wavelengths used on each cable must remain constant throughout the path. Nevertheless, if required, the user can obviously constrain the path to use the same set end-to-end *i.e.* in WP routing.

As an example, Fig. E.7 is of a 6-node, 7-cable network, with each cable consisting of two fibres, and each fibre able to carry up to two wavelengths. This network can be viewed as equivalent to that in Fig. E.3(a) on p. 261, but with the wavelength-routing structure made visible. The network carries three wavelength-routed paths, which are again equivalent to the paths in the earlier figure. Their string representation, taking node 0 as the source, is: `"(0[1:1],1[1:1],2[0:1],3)"`, `"(0[0:1],1[0:1],4[1:1],3)"` and `"(0[0:0],5[0:0],4[1:0],3)"`. Here, a *join* between nodes is represented by `"[a:b]"`, where `a` is the fibre id, and `b` a set of wavelength numbers (see class `Join`, below). Further, if the three wavelength-routed paths were placed in a path-loss sequence, then this would be represented as:

```
"{P0(0[1:1],1[1:1],2[0:1],3), P1(0[0:1],1[0:1],4[1:1],3), P2(0[0:0],5[0:0],4[1:0],3)}"
```

Reflecting the relationship between the simple and wavelength-routed path concepts, the `PhyPath` class is derived from `Path` (Fig. E.8). The main constructor takes two arguments, a `const` pointer to the parent wavelength-routed network and the (optional) capacity; these are simply passed to the `Path` base class constructor. The resulting wavelength-routed path is empty and so, as with simple paths, a second constructor is also provided that takes an additional string argument specifying the wavelength-routed path itself. Although, for compatibility with simple paths, a capacity argument is provided for both constructors, the capacity of a wavelength-routed path can always be determined by the product of the number of wavelengths used and their granularity. Consequently, a wavelength-routed path maintains its own capacity, and these constructor argu-

[16] Originally called a *physical* path by the author, hence `PhyPath`, the name wavelength-routed path is now used; however, the class name remains unchanged.

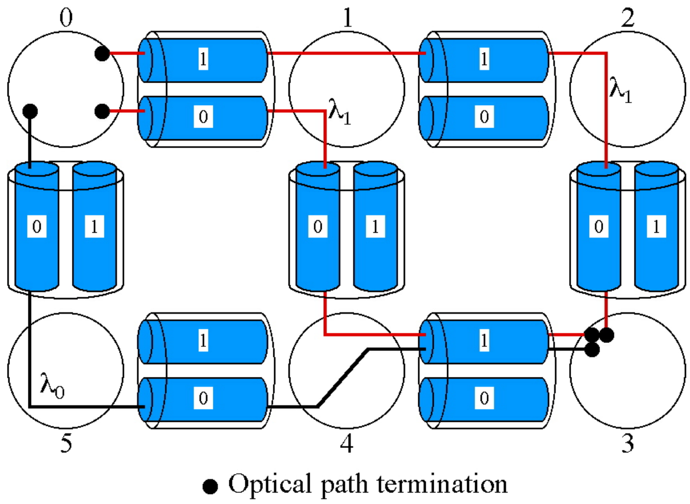


Figure E.7: Example wavelength-routed paths

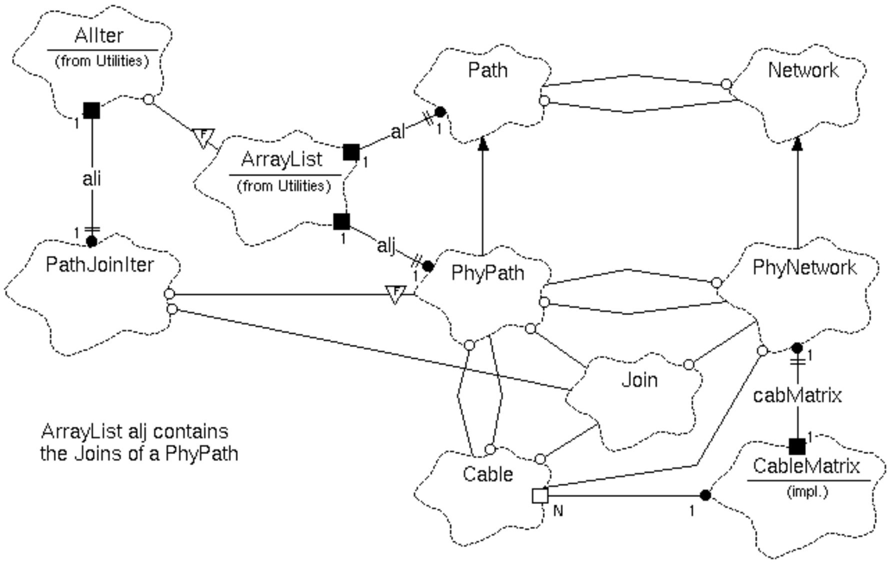


Figure E.8: `PhyPath` class diagram

ments are ignored; in addition, the member function used to set path capacity is re-implemented to fail an assertion if invoked.

To record the additional join information between each node pair along the wavelength-routed path, a dynamic array (array-list `alj`, see §E.6) of pointers to join objects is used. This structure shadows the array-list (`al`) that holds the node pointers for the underlying path, but necessarily consists of a shorter sequence of joins. For a wavelength-routed success path, exactly one less join than node will be held; for a wavelength-routed loss path, two less, as no join is required to connect the wavelength-routed path to the fictitious loss node. Unlike node objects, the joins are actually owned by the wavelength-routed path: it is responsible for their creation and destruction.

Many of the `PhyPath` class' member functions are simply necessary re-implementations of the

`Path` class'; others provide equivalent operations on the sequence of joins to those the `Path` class provides for its sequence of nodes; whereas some functions are completely new.

As well as re-implemented member functions to set (from a string) and clear a wavelength-routed path, several functions are needed to extend a wavelength-routed path by a single node or many. For extension by one node, the `append()` member function is re-implemented, but it can only be applied to an empty wavelength-routed path: in all other cases, a join must be provided as well[17]. Hence a version of `append()` is included that takes two arguments: the node to add, and the join to make the connection. The latter is passed as a `const` pointer, as only a copy will be stored by the wavelength-routed path. However, as well as the limitations imposed to extend the underlying path, several additional requirements must be met. Unless the node is the loss node, the join must be non-empty. If it is the first join on the wavelength-routed path, it sets the wavelength count; if not, it must use the same number of wavelengths as all the prior joins. Further, the join must not be blocked by the cable it will use (*i.e.* all the wavelengths and fibres needed must be free).

Directly appending an actual path object to a wavelength-routed path is impossible, due to the complete absence of wavelength-routing data in the former, so the appropriate `append()` function is re-implemented in the `PhyPath` class to fail an assertion. However, the '`+`' operator, that appends a simple path to a wavelength-routed path, *can* be re-implemented successfully: it returns the new path (*not* a wavelength-routed path) formed by first copying its wavelength-routed path argument into a new path object, and then adding a copy of the second path. Nevertheless, appending two wavelength-routed paths is supported by a more specific `append()` member function and '`+`' operator that each take two wavelength-routed path parameters. For both these functions, as well as satisfying the path requirements, the two wavelength-routed paths appended must also use the same number of wavelengths.

Access to individual joins is provided, returning a `const` pointer to maintain data integrity, by four member functions. These retrieve the first join, the last join, a join by index along the wavelength-routed path, and a join by specifying its end-node ids. The three path test functions are re-implemented to work correctly for wavelength-routed paths, as are the private member functions underlying file input/output. In addition, a member function is provided to count the total number of distinct wavelengths used on the wavelength-routed path (which may be more than the number per join if VWP routing [294] is used[18]) and another returns its highest wavelength number.

[17] This also acts as a restriction on '`+=`', which is implemented by `Path` using this version of `append()`.
[18] If the number of wavelength channels is required, then the wavelengths used on the first (or any) join can be counted.

`Join` class

As has been explained in the context of the `Cable` and `PhyPath` classes, in the network model a join is simply a fibre id and a set of wavelengths: it captures the occupancy of a wavelength-routed path on a cable. In string form, it is represented by `"[a:b]"`, where `a` is the fibre id, and `b` is a comma-separated list of either individual wavelength numbers or hyphen-separated wavelength ranges. For example, `"[2:4,6-9]"` means wavelengths 4, 6, 7, 8 and 9 on fibre 2. As elsewhere in NOMaD, wavelength numbers are restricted to the range 0–31.

Three main constructors are provided for `Join`: one takes no arguments, and constructs an empty join; another needs a string representation; the third requires both a fibre id and a bit vector (32 bits) of the wavelength numbers. The underlying data structure of a join consists of the fibre id and a bit vector (32 bits) of wavelength numbers.

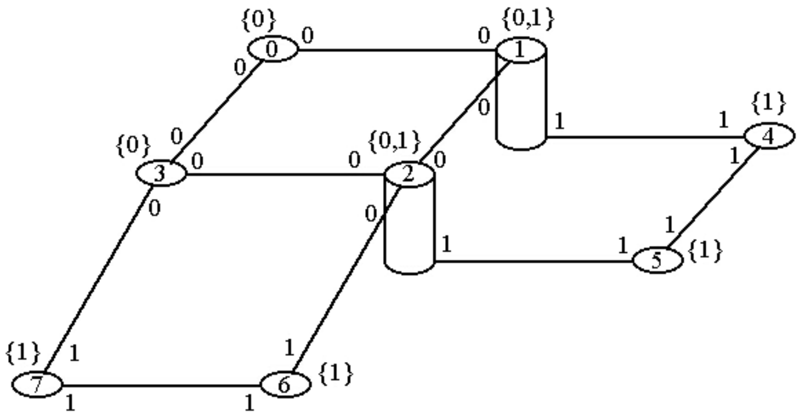
As well as member functions to clear (*i.e.* empty) the join and set it from a string, the fibre id can be set on its own, or individual wavelength numbers added or removed. Query functions allow the fibre id to be retrieved, and the wavelength numbers to be accessed as a bit vector (32 bits), or even as an individual wavelength number (if the join is on only one wavelength). The wavelengths can be counted, and the highest wavelength number obtained. The join can be tested to see if it is empty (*i.e.* on no wavelengths), or whether it is on a particular wavelength. Two joins can be tested for equality, either by an overloaded '`==`' operator, or an ordinary member function. In addition, file/input output is provided by the usual overloaded `iostream` operators.

E.4 Network Hierarchy and Node Clustering

NOMaD's network model directly supports the concept of network hierarchy, by enabling nodes to be on one or more levels, and links to be connected to a particular level at each end[19]. Fig. E.9(a) illustrates this with an 8-node, 10-link, 2-level hierarchical network. In the figure, node numbers are drawn on the nodes themselves; node levels are shown in braces adjacent to the nodes (*e.g.* the label {0,1} by node 1 records its membership of both levels 0 and 1); and link connect levels are indicated by numbers next to each end of the link. For clarity, the links themselves have not been labelled, but will be referred to by their end node ids. Nodes 0 and 3 are on level 0 (the 'top' level); nodes 4, 5, 6 and 7 on level 1; and nodes 1 and 2, drawn as cylinders, are on both levels.

[19]This section assumes the reader is familiar with §E.3.1 on p. 253, and in particular the discussion of the `Network` and `Node` classes.

Most of the links clearly belong to one level or another, but links (2,6) and (3,7) are 'sloped'—they are connected at two different levels. While NOMaD supports networks that have both nodes on multiple levels, and links that interconnect different levels, the author would expect that any given network design would use only one or other of these; this restriction is not, however, enforced by the network model.



(a) Network

From/To	0	1	2	3	4	5	6	7
0		0		0				
1	0		0		1			
2		0		0		1	0	
3	0		0					0
4		1				1		
5			1		1			
6			1					1
7				1			1	

(b) Matrix

Figure E.9: Example connect levels matrix

To represent the network hierarchy, node objects store the levels they are on (restricted by implementation to the range 0–31, with 0 the default[20]), although any change to their levels is made through the public interface of the `Network` class, to allow the overall consistency of the network hierarchy to be maintained (*e.g.* it is not permitted to remove a node from a level on which it has an attached link connected; the connection must be (re)moved first). The link connect levels are stored in the network's connect levels matrix (`conMatrix`), a two-dimensional array of size `numNodes` × `numNodes` of integers (see Fig. 3.4(a) on p. 58). It was decided to make the matrix an attribute of the `Network` class, rather than storing the individual connect levels in the links, as it was felt that these are more correctly a property of the network as a whole, as they represent connections *between* links and nodes. The connect levels matrix corresponding to the network in Fig. E.9(a) is provided in in Fig. E.9(b), where the reader should note that the matrix is not symmetrical, due to sloped links (2,6) and (3,7).

The `Network` class provides a comprehensive set of member functions for the manipulation of both levels and connect levels. For node levels, there are functions to add a level to a node[21], remove

[20]Restriction to this range allows an unsigned integer [16] to be used as a bit vector (32 bits); a greater range would require the use of a true bit-vector class.

[21]Or equivalently, to add a node to a level, depending on your point of view!

a level from a node, or change a node to a new level, along with all its connected links (provided it is only on *one* level!). Queries on the levels of an individual node are handled directly through its own member functions, described below. However, the levels can be counted or retrieved for the network as a whole, either as a string (*e.g.* `"{0,1}"`) or a bit vector (32 bits). For link connect levels, individual entries in the connect levels matrix can be set and retrieved; in addition, the connect levels for an individual node id can be counted, queried with respect to a particular level, or retrieved as a whole, either as a string or a bit vector.

As an aid to the design of cluster-based architectures, such as multi-ring networks, the network model also provides direct support for node clusters; these are also intended to be useful in the development of any cluster-based operators or overall design heuristics. Individual nodes can be included in one or more clusters, although cluster numbers are restricted to the range 0–31 for implementation reasons, with 0 the default[22]. Fig. E.10 illustrates the concept with a 9-node, 12-link, 4-ring (or 'cluster') network. Nodes 3 and 4 are in cluster 1; nodes 5 and 6 in cluster 2; and nodes 7 and 8 in cluster 3. The central three nodes, while all being in cluster 0, are each also included in two other clusters: hence node 0 is in clusters {0,1,3}, node 1 in {0,1,2} and node 2 in {0,2,3}.

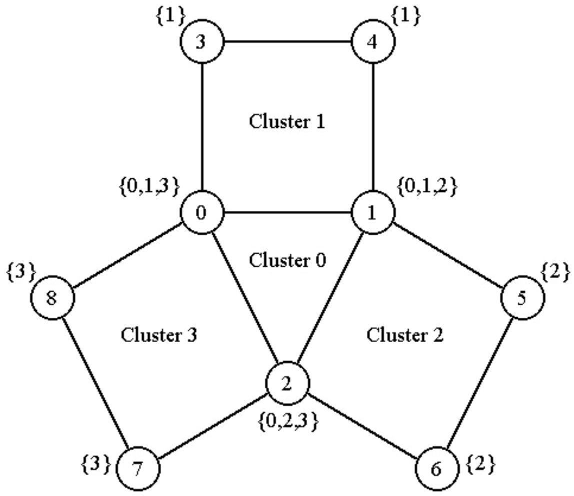


Figure E.10: Example network with node clusters

While individual nodes store the clusters they are in, any change to their clusters is made through the public interface of the `Network` class, both for consistency with the handling of network hierarchy (see above), and also to allow for some interaction between the network hierarchy and node clusters. There are `Network` class member functions to add a cluster to a node[23], remove a cluster from a

[22] As with levels, this range allows the use of an unsigned integer as a bit vector (32 bits).
[23] Or equivalently, to add a node to a cluster, depending on your point of view!

node, or change a node to a new cluster (provided it is only in *one* cluster!). Cluster queries for an individual node object are handled directly through its own member functions, described below. However, the clusters can be counted or their ids retrieved for the network as a whole, either as a string (*e.g.* `"{0,1}"`) or a bit vector (32 bits). In addition, to support the use of rings/clustering in hierarchical networks, the clusters can be counted or their ids retrieved for all nodes (as a whole) connected at a given level.

As was explained in §E.3.1 on p. 257, the main `Node` constructor can only be invoked by the `Network` class. It is supplied with a network parameter database, from which the node extracts, amongst other items, its levels and clusters. The node's levels are supplied as a string, *e.g.* `"{0,1}"`, which is mapped into a bit vector (32 bits) for storage. If absent from the parameter database, `"{0}"` is assumed. Via member functions, the levels can be counted, and retrieved either as a string or a bit vector; in addition, the node object can be queried as to whether it is on a particular level, and whether it is on only one level (if `true`, the actual level is returned as a reference parameter [16]). Clusters, apart from their meaning, are supported by the `Node` class in exactly the same way as levels.

E.5 Iterators

The descriptions of several iterator classes (useful, but less important than more central classes) are provided here, rather than in Ch. 4: `PathIter`, `PathLinkIter`, `PlsIter`, `CableIter` and `PathJoinIter`.

`PathIter` class

The *path iterator* class (`PathIter`) is provided to enable a user to access efficiently, conveniently and in series the nodes on a path object. It conforms to Stroustrup's design for a non-intrusive iterator [16].

The main constructor takes a reference to a path, and only one member function is provided, the overloaded '`()`' operator (*i.e.* function call [16]). Each time it is invoked, this returns a `const` pointer to the next node on the path, starting with the first; after these have all been returned, it simply returns a void pointer. This is perhaps better understood by examining the example code below, which implements the `Path` class' `print()` private member function (for clarity, error-handling code has been removed):

```
void Path::print(ostream&os) const {
  os << '(';
  PathIter pi(*this);
  const Node* n;
  if (n = pi())
    os << n->getId();
  while (n = pi()) {
    os << ',';
    if (n->isLoss()) {
      os << 'L';
      break;
    } else
      os << n->getId();
  }
  os << ')';
}
```

First, a bracket is output, then a path iterator (`pi`) is constructed for the current path object (`this`) and a `const` node pointer declared (`n`). The first node is assigned to `n`, and if not void, its id printed. The code then enters a loop (until the iterator is exhausted), alternatively printing commas and node ids, while ensuring the appropriate response is made if the loss node is encountered. Finally, the closing bracket is output.

Implementation of the `PathIter` class is achieved simply by using the appropriate iterator (an array-list iterator, see Fig. E.4 on p. 262) on the path's underlying array-list (see §E.6).

`PathLinkIter` class

The *path-link iterator* class [16] (`PathLinkIter`) complements the path iterator class by allowing the user sequential access to the links on a path. The main constructor has one argument, a reference to a path, and the usual overloaded function-call operator is provided. Implementation (Fig. E.4 on p. 262) is achieved using a path iterator object (`pi`) to iterate over the nodes, a data member to store a pointer to the previous node (`lastNode`), and access to the parent network (via any of the nodes) to retrieve each link in turn[24].

`PlsIter` class

The *path-loss-sequence iterator* class [16] (`PlsIter`) allows the user sequential access to the paths in a path-loss sequence. The main constructor has one argument, a reference to a path-loss sequence, and the usual function-call operator is provided. Implementation (Fig. E.5 on p. 265) is achieved using a linked-list iterator object (`li`) to iterate over the underlying linked-list object. Unfortunately, as the function-call operator returns a `const` pointer to a simple path object (class `Path`), if the sequence actually contains wavelength-routed paths (class `PhyPath`), then the appropriate cast [16] will have to be used.

`CableIter` class

The *cable iterator* class [16] (`CableIter`) allows the user sequential access to both the wavelength-routed paths on a cable and their corresponding joins. It is the iterated equivalent of the `Cable` class' `getJoin()` member function, which retrieves the join corresponding to a particular wavelength-routed path on the cable, but here the user does not need to provide the individual paths: the iterator does it all.

The main constructor has one argument, a reference to a cable, and the usual function-call operator is provided. However, this can only return one argument, a `const` pointer to the next wavelength-routed path. Consequently, a `getJoin()` function is provided which, invoked after a particular path has been retrieved, will return the corresponding (dynamically-allocated) join object.

Implementation is achieved by stepping through the fibres one at a time, retrieving each wavelength-routed path in turn. The cable iterator keeps track of which fibre it is working on and the wavelengths on that fibre for which it has already returned a path. In addition, to allow a copy to be obtained whenever requested, it also maintains a join object for the last wavelength-routed path returned.

`PathJoinIter` class

The *path-join iterator* class [16] (`PathJoinIter`) allows the user sequential access to the joins along a wavelength-routed path. The main constructor has one argument, a reference to a wavelength-

[24]For a loss path, the fictitious link to the loss node has no 'real' existence; the iterator simply returns a void pointer.

routed path, and the usual function-call operator is provided. Implementation is achieved simply by using the appropriate iterator (an array-list iterator, see Fig. E.8 on p. 273) on the wavelength-routed path's underlying array-list of joins (see §E.6).

E.6 `Utilities` Class Category

The `Utilities` class category contains a few, relatively unrelated, classes that were developed to support those in other categories. These include dynamic arrays, circularly-linked lists, file handling and pseudo-random number generation.

`ArrayList` and `AlIter` classes

The *array-list* class (`ArrayList`) is a type of dynamic array that blends some of the aspects of a linked list with the inherent capabilities of a basic C/C++ array. It maintains an indexed array of `const void*` pointers [16] *i.e.* untyped pointers that can store the address of any object, which the array-list 'grows' as and when required. Object pointers contained in the array are always stored in an unbroken range, starting from index 0. Its most important use is as a base class for the `Path` and `PhyPath` classes (see §E.3.2 and §E.3.3).

Two main constructors are provided, one without arguments which creates an empty array-list, the other with a single argument: a pointer to the first object to be contained.

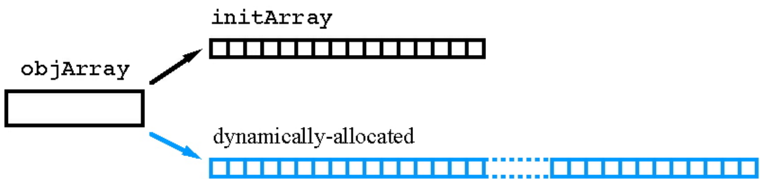


Figure E.11: `ArrayList` array implementation

An array-list is implemented as a pointer to an array (`objArray`) of `const void*` pointers. Initially, `objArray` points to another data member (`initArray`), which is simply an array of 16 `const void*` pointers (Fig. E.11). However, if the size of the array needs to be increased beyond 16, the array-list switches to using dynamically-allocated arrays, each twice the size of the one before. This approach was adopted to increase the speed of array-list allocation, because `initArray`, as a data member, is available immediately after the construction of the object. Using a dynamically-allocated array from the start would have required an additional allocation request. As the `Path` and `PhyPath` classes, both derived from `ArrayList`, use rapid class-based memory allocation, this modification resulted in considerable increase in the speed of programs that make heavy use of the two path classes (the majority of NOMaD applications), particularly where the number of nodes in the paths is 16 or less. To monitor the need to increase the size of the array, two integer data members are used: one records the number of objects contained, the other the maximum allowed in the current array. Then, whenever needed, a private member function is automatically invoked to increase (double) the array size.

The contents of the array-list can be modified in several different ways. One member function places a single object at the end of the array, whereas another places one at the start (*i.e.* index 0), moving the prior contents of the array up by one location. A complementary member function removes, and returns, the object at index 0, moving all the remaining elements down. Further, the array can be cleared (*i.e.* emptied) as a whole, or an object at an existing index replaced by a new one.

As well as modifications to the array, a number of queries have also been made available. Individual objects can be retrieved from a given location, either using a member function or an overloaded operator; special member functions also allow retrieval of the objects at either the first or the highest current index locations. The size of the array (*i.e.* a count of the number of objects contained) can be obtained, or the array checked to determine if it is empty. In addition, whether the array-list contains a particular object can be queried.

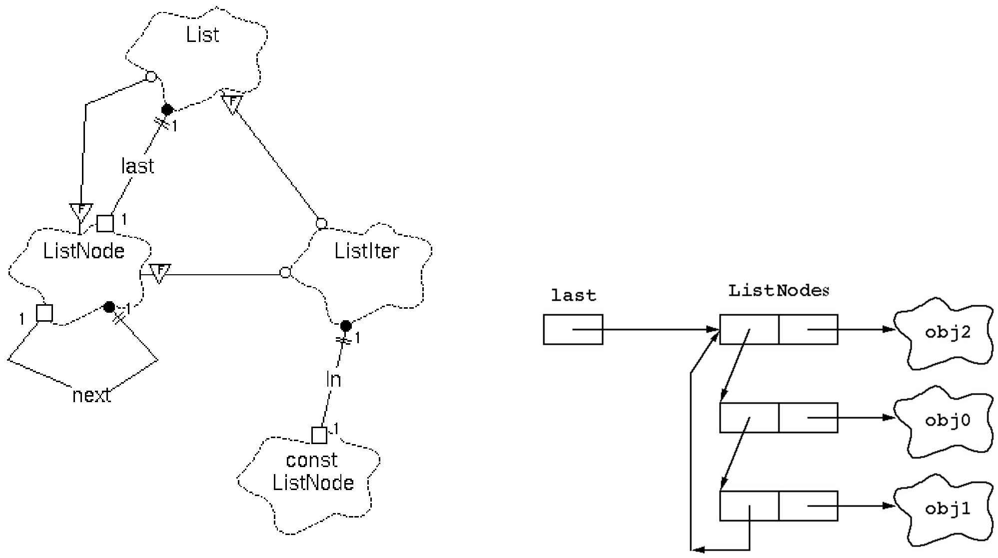
The array-list iterator class (`AlIter`) [16] allows the user sequential access to all the objects in the array[25]. Its main constructor has one argument, a reference to an array-list, and the usual overloaded function-call operator is provided. Implementation is achieved by simply maintaining an index into the array as iteration proceeds.

`List`, `ListNode` and `ListIter` classes

The *linked-list* class (`List`) is a non-intrusive circularly-linked list of objects, each stored as a `const void*` pointer [16] (see Fig. E.12(a)). While implemented very differently, its public interface has much in common with that of an array-list.

The *linked-list-nodes* themselves are objects of class `ListNode`, each of which simply consists of a `const void*` pointer to the linked object, plus a pointer to the next node in the list. For example, Fig. E.12(b) shows a list of three objects (`obj0`, `obj1` and `obj2`). The linked-list contains a single private data member, `last`, which points to the *last* node on the list: this allows efficient

[25] For a more complete description of an iterator, see the description of the path iterator on p. 278.



(a) Class diagram (b) `List` of three objects

Figure E.12: `List` class diagram and example

access to both the beginning and end of the list, without the overheads necessary to maintain a doubly-linked list [287,326]. Thus `obj0` is the first object on the example list, `obj1` the second, and `obj2` the third and final.

The linked-list-node class (`ListNode`) is a simple one, with only one member function, its private constructor (used only by friend class `List`) which takes two arguments, a pointer to the object to be linked into the list, and a pointer to the next linked-list-node object. The two pointers are then simply copied into the appropriate data members in the newly constructed linked-list-node object.

On the other hand, the `List` class presents a more complex interface. Two main constructors are provided: one, with no arguments, creates an empty linked-list, the other constructs a linked-list containing one object, passed in as an argument.

For linked-list modification, member functions are provided to insert an object at the start, or append an object to the end of a list. Either the object at the start can be removed, or any of the objects on the linked-list, specified by pointer. Further, the list as a whole can be cleared (*i.e.* emptied) of all its objects.

Queries are supported by member functions to determine the size of the linked-list (*i.e.* how many objects it contains), test whether it is empty, or ascertain whether it contains a particular object. In addition, individual objects can be retrieved by their index in the linked-list, using either a member function or an overloaded '`[]`' operator; member functions to access the first and last

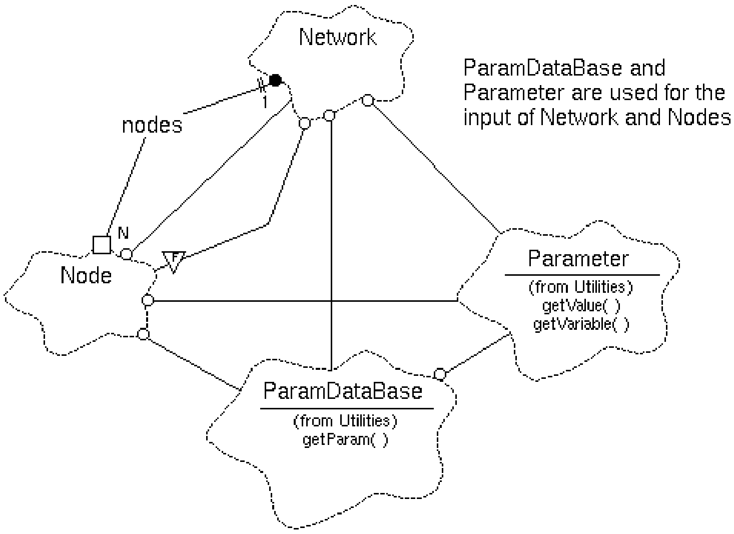


Figure E.13: `Network/Node` input/output class diagram

objects in particular are also provided.

The *linked-list iterator* class (`ListIter`) [16] allows the user sequential access to all the objects on the linked-list. Its main constructor has one argument, a reference to a linked-list, and the usual function-call operator is provided. Implementation is achieved by maintaining a pointer to the appropriate linked-list-node as iteration proceeds.

`Parameter` and `ParamDataBase` classes

The *parameter* (`Parameter`) and *parameter database* (`ParamDataBase`) classes together provide a convenient abstraction of file input/output (itself accessed indirectly through the `iostream` classes [16]). They operate on files whose formats were detailed earlier in this Appendix (§E.1), and their use by NOMaD's classes is described throughout Ch. 3 and 4, in Appendix C and elsewhere in this Appendix. For example, Fig. E.13 illustrates the relationships between the classes involved in network input/output. During network construction (§E.3.1), first a parameter database is itself constructed from an input file (both network specification and NOMaD parameter files will be required), and then the individual parameters (*i.e.* variable-value pairs) can be retrieved as parameter objects, using the `getParam()` member function (selected using the variable string argument as key). The variables or values of these can then be accessed via the `getVariable()` and `getValue()` member functions, respectively. In the same way, nodes use a parameter database during their own construction.

The `Parameter` class has two main constructors: one, without arguments, constructs an empty object; whereas the other stores the variable and value string arguments provided. Input/output is supported by overloaded `iostream` operators, with the overloaded input operator the only way to change the variable-value pair once the parameter object is constructed. As mentioned above, the stored variable and value strings can be retrieved using member functions, and in addition, whether a parameter object is empty can be tested using an overloaded `int` operator. Use of this aspect of the `Parameter` class is perhaps best illustrated by example code. The following was taken from one of the parameter database constructors; for clarity, both error checking code and some of the implementation have been deleted.

```
ParamDataBase::ParamDataBase(istream&is) {
  // ...
  is.clear();
  is.seekg(0); //Rewind input stream
  while(is) {
    //Get next Parameter
    Parameter param;
    is >> param;
    if (!param || strlen(param.getVariable()) == 0 ||
        strlen(param.getValue()) == 0)
      continue;
    //Create new entry, or modify existing one
    // ...
  }
}
```

First, the input stream (`is`) has its error status cleared, and is rewound to the beginning of the file. Then, while the input stream remains `good` (itself tested via an overloaded `int` operator thus: `while(is)`), successive lines from the file are read into parameter `param`. If it remains empty (tested via the overloaded `int` operator thus: `!param`), or either the variable or value strings are of zero length, then that line is ignored; otherwise, the parameter is stored as an entry in the parameter database (code omitted).

For efficiency, the parameter database class (`ParamDataBase`) is implemented using a tcl hash table [284, 285]. Actual parameter objects are not stored, but instead only value strings, accessed via the hash value generated from the corresponding variable string. However, this implementation detail is concealed by the public interface, which uses parameter objects.

Two main constructors are provided: one without arguments creates an empty parameter database; the other, taking an input stream argument, uses this to populate the new database with parameters (see example code above). Individual parameters can then be set, whether they are already in the database or not, using a member function that takes a parameter argument. In addition, parameters can be deleted using a function that takes a variable string argument; this returns a Boolean indicating whether the parameter was actually already in the database to be deleted. Further, given a variable string, individual parameters can be retrieved, as well as the database as a whole printed to an output stream.

`RandomGen` class

The *pseudo-random number generator* class (`RandomGen`) provides uniform, negative-exponential, geometric, Bernoulli and discrete-uniform probability distributions. It is based on the `urand5()` generator first published by the author in *Principles of Performance Engineering for Telecommunication and Information Systems* [327], which was itself an implementation of a generator described by Fishman [328]. The generator is linear congruential, of prime modulus multiplicative type, and is coded for portability to either 32- or 64-bit machine architectures.

Two main constructors are provided: one takes an `unsigned long` as the generator seed, the other assumes a seed of 1. The current number in the pseudo-random sequence can be retrieved at any point using the `getZ()` member function, but the series itself is only advanced when a number from one of the distributions is requested. These include `uniform()`, which returns a double-precision floating-point value uniformly distributed in the range 0+ to 1; and `exponential()`, which provides a double-precision floating-point value from a negative-exponential distribution, mean `1/lambda`, where `lambda` is supplied as an argument. The `geometric()` member function returns integer values, geometrically-distributed, with the probability of success given as an argument. Bernoulli-distributed numbers can be obtained via a member function `flip()`, which acts as a biased coin, returning a Boolean, where the probability of `true` is passed in as an argument (default 0.5). Finally, `intRange()` gives random integers in a range inclusive of its two arguments.

E.7 Testing & Verification

As was explained in §3.8, 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. Although each of these was mentioned briefly in the main text, more detailed explanation is provided here.

Meyer's concept of Programming by Contract [289] requires that the preconditions necessary to the correct operation of a class member function be made explicit. In return, the member function will either meet certain explicit postconditions on completion, or signal failure. This contract, between the invoker and implementer of a class member function, helps to ensure software reliability. The member function should only be invoked when all the preconditions to its correct operation are present; it should then either completely satisfy all its promised postconditions, benefiting the caller, or obviously fail. In addition to member function pre- and postconditions, Meyer also introduced the idea of class invariants: these are conditions which should be met both before and after the invocation of any member function of a class (although clearly only *after* the constructor and *before* the destructor!). Effectively, the class invariants can simply be thought of as additional pre- and postconditions that are shared by all member functions.

Although pre- and postconditions, building on the concept of abstract data types, are an obvious aid to clarity in class design, they can also be used as an execution-time aid to software reliability. Meyer's Eiffel language [289] included keywords `require`, `ensure` and `invariant` to designate Boolean clauses expressing preconditions, postconditions and class invariants, respectively. If any of these conditions was violated during execution, the run-time environment would throw an exception. Various mechanisms would then permit the exception to be caught and handled, but ultimately execution would fail if a correct sequence of instructions could not be found. More details on both Programming by Contract and the Eiffel language can be found in [329, 330]. The concept of Programming by Contract has also been incorporated into various object-oriented analysis and design approaches, including BON (Business Object Notation) [331], which mainly supports Eiffel implementation, and the more general Booch Method [15].

Although C++ does not directly support pre- and postconditions on class member functions, a set of macros was written by the author, and used extensively in the implemented code. For simplicity, C++'s exception mechanism was not used, but rather a simple error message was output on condition failure, indicating source file, line number and a short textual error message (the *tag*).

For example, the source code of the `insert()` member function of class `List` is given below. This places an object, pointed to by `obj`, at the head of the circularly-linked list. For clarity, the actual implementation code has been omitted.

```
void List::insert(const void*obj) {
  require("this not void", this);
  require("obj not void", obj);
  // implementation
  ensure("not empty", !isEmpty());
  ensure("contains obj", contains(obj));
  ensure("is first", getFirst() == obj);
}
```

As can be seen above, the `require` and `ensure` macros take two arguments: the tag, *i.e.* a string containing a brief textual description of the condition, and the Boolean to be tested. The two `require` statements check that both the linked-list object (`this`) and the object to be inserted (`obj`) are non-void *i.e.* point to (hopefully) valid addresses. The three `ensure` statements test that after the insertion, the linked-list is non-empty, it contains the inserted object, and that `obj` has been placed at the front of the list.

While the use of Programming by Contract cannot eliminate all errors in a piece of software, the author found that, as well as encouraging clear thinking in both design and implementation, some subtle errors, that might otherwise have gone undetected, would often be caught by one of the conditions. However, while Programming by Contract was used throughout the development of NOMaD, it should be noted that during long program executions, the pre- and postconditions were omitted (using conditional compilation), as they would otherwise have significantly increased program execution time.

The second approach used was dynamic memory tracking: as C++ requires the programmer to directly manage the allocation and release of all dynamic memory objects [16, 282][26], 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. This was particularly important for some NOMaD runs, where a large fraction of machine memory was required to hold the evolving population of networks.

[26] "It is said that Lisp programmers know that memory management is so important that it cannot be left to the users and C programmers know that memory management is so important that it cannot be left to the system." [16]

To combat problems in this area, dynamic memory tracking code (called `xmem`) for C [290] was rewritten by the author for C++. The tracker operates by replacing the implementation of the global `new` and `delete` operators. The replacement `new` operator stores the size and location of each dynamic memory allocation in a hash table, as well as recording the current total amount of memory allocated, the largest total amount allocated so far, and the current overhead of additional allocation caused by `xmem` itself. These three values are made available to the user though global variables. In addition, every allocation request is increased by two bytes into which a known data pattern is written, in order to detect array over-run and other possible memory violations on deallocation. Clearly, the `new` operator can also detect when dynamic memory itself is exhausted. Using the hash table built by `new`, the replacement `delete` operator is able to detect deallocations that do not correspond to a currently allocated area of dynamic memory: this will catch both the use of invalid addresses, and also accidental 'double' deallocation of objects. In addition, any change to the two additional bytes after each allocated area will be reported, indicating invalid access to those memory locations.

The `xmem` dynamic memory tracker was used throughout the development of NOMaD, and several subtle memory problems were identified by its use. However, it should, once again, be noted that `xmem` was not linked into the executables used for long experimental runs, due to the increased run times produced by dynamic memory tracking.

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. During the course of NOMaD's development, as new classes were added or existing classes extended, the regression tests would be run again. Any unexpected change in the operation of existing classes and their member functions would (hopefully) result in a failed assertion. Subsequent investigation would enable the fault to be traced to a design error, an implementation error, or a failure on the part of the author to anticipate valid changes to NOMaD's operation! Periodically, the regression test suite would itself be extended to incorporate NOMaD's growing functionality.

As an illustration of the regression tests, the code below represents a small portion of the tests applied to the `Path` class (see §E.3).

```
Path* p1 = new Path(net1);
p1->append(net1->getNode(0));
```

```
p1->append(net1->getNode(1));
p1->append(net1->getNode(2));
p1->append(net1->getNode(7));

char buf[256];
ostrstream osb(buf, 256);
osb << *p1 << '\0';
assert(strcmp("(0,1,2,7)", buf) == 0);

Path* p2 = new Path(net1);
(*p2) += (*net1->getNode(0));
(*p2) += (*net1->getNode(4));
(*p2) += (*net1->getNode(3));
(*p2) += (*net1->getNode(6));

osb.seekp(0);
osb << *p2 << '\0';
assert(strcmp("(0,4,3,6)", buf) == 0);
```

The code shows an empty path, pointed to by `p1`, being constructed for use in network `net1`. The path is then grown by the addition of nodes 0, 1, 2 and 7 using the `append()` member function. The path is then output (using its overloaded '`<<`' operator friend function [16]) into an `ostrstream` buffer, and the results compared with the expected string, `"(0,1,2,7)"`. The next few lines of code show a similar sequence for a second path, but this time the '`+=`' operator is the one under test.

E.8 NOMaD Software: Discussion & Further Work

The design and implementation of NOMaD has been considered at length in the thesis, with Ch. 3 providing an overview, Ch. 4 and §E.3 exploring much of the object-oriented toolset, and the remainder of Appendix E providing additional details.

E.8.1 Overall Architecture

The overall architecture of NOMaD is illustrated in Fig. 3.2 on p. 55, consisting on one hand of compiled applications built using the object-oriented network toolset and, on the other, of scripted applications with optional graphical user interfaces (GUIs) built using the `NOMaDsh` tcl extension library. When NOMaD was originally conceived, it was anticipated, as was explained in §3.4, that `NOMaDsh` would play an important role. However, with two research assistants[27] and the author all using the NOMaD toolset, it soon became apparent that the clear preference was for compiled applications in C++. When the object-oriented toolkit was extended by the author, the new features were available to compiled applications immediately, but there was an inevitable implementation delay in also making them available in `NOMaDsh`. Consequently, whilst working with interpreted scripts might have been preferable had NOMaD not been continually developed, in practice `NOMaDsh` fell further and further behind the object-oriented network toolset. As a result, active development of both `NOMaDsh` and its GUI script library effectively ceased after the work on mesh network topology design was completed.

An unfortunate casualty of discontinuing `NOMaDsh` development was not developing the ability to visualise the results of routing, fibre and wavelength allocations (unlike mesh network topology, which was included). This could be an important area of future work, as while it is relatively straightforward to display topology designs, routing, fibre and wavelength allocations are much more difficult to present concisely in a two-dimensional diagram, even with the use of colour. This can be eased in a design tool, to some degree, as different aspects of the allocation can be displayed in response to interactive prompts from the user (*e.g.* basic routes could be selectively elaborated to show wavelength allocations). An intriguing alternative would be to try three-dimensional representation of allocations using virtual reality modelling (*e.g.* VRML [332]). This might well provide the designer with far greater insight into the allocations than can be gleaned from, say, examining summary statistics (*e.g.* average path hop count, range of link wavelength requirements).

E.8.2 Reimplementation in Java

Given the limitations that have become apparent in combining an object-oriented network toolset in C++ with an interpreted GUI in tcl/tk, a promising future possibility would be to re-implement NOMaD in Java [333,334]. By using the AWT or Swing libraries, this would allow the development

[27] Katerina Proestaki and Noel Parnis.

of both the object-oriented network toolset and the GUI library in one programming language. As a result, GUIs could be seamlessly incorporated into applications. In addition, the multi-threading capabilities of Java would allow a greater interaction with the applications whilst background processing proceeded. Moreover, the portable nature of Java would allow applications to be run on a variety of platforms.

Indeed, partial re-implementation of NOMaD in Java was undertaken in early 2000, with many of the GA classes being ported from C++ to Java to create a simple framework for GA/heuristic hybrids (referred to as sjhga). The sjhga framework has been used successfully in two M.Sc. projects to date [335, 336], and is currently being used on a Ph.D. project by one of the author's students (John Pagonis).

One aspect of the design of NOMaD which adds notable flexibility is the use of execution-time class parameters (parameter databases), as discussed in Appendix E. However, to allow users the full choice of, say, which genetic operators to use, all the C++ class code has to be compiled into the corresponding executable. Consequently, these are often of considerable size (*e.g.* `ga` is currently 1.1 Mbytes), and are built using large and complex make files. With Java, however, only those classes actually used have to be loaded, no make files are required and they do not need to be combined into several large executables.

E.8.3 Network Modelling

NOMaD's network model has been used for optical network design in several research projects, with a variety of heuristic approaches successfully employing the model (*e.g.* [20–24]), in addition to the author's own GA/heuristic hybrid work. Nevertheless, there are a number of areas in which the object-oriented design could be improved.

The classes that support mesh network topology design, including `Network`, `Link`, `Path` and `PathLossSeq` (see Fig. 3.4(a) on p. 58) were developed before those for routing, fibre and wavelength allocation. However, given the amount of code that had already been developed, it was decided to derive `PhyNetwork` from `Network`, and `PhyPath` from `Path` (Fig. 3.4(b)). While these are valid design choices, it may be that a better choice would have been to have had abstract network and path classes from which ones specific to topology and wavelength allocation were derived. In a similar way, there are subtle problems with both path-loss sequences and the relationship between link and cable classes that might benefit from re-design.

For future work, there are a whole range of possible extensions to the network model that could be considered. At present, nodes in the NOMaD are represented by a single class. However, to support more detailed cost modelling and to allow the inclusion of appropriate design constraints, the current `Node` class could be replaced by an abstract class. From this, classes for OXCs, add-drop-multiplexers (ADMs), *etc.* could then be derived. This would allow, for example, network assessments to enumerate the ports required in a network, which could in turn be a significant factor in network cost [23].

In all the metrics for design assessment used in the thesis (§4.1) only a single design period has been used, and accurate forecasting of both costs and traffic requirements has been assumed. However, in practice networks are often deployed over several months or years, and there is often forecasting uncertainty [337]. These factors could be addressed in future work, allowing for both network roll-out and robustness to minor variations in cost or demand [338]. Indeed, the author has already supervised two M.Sc. projects in this area, one which of which studied traffic estimation for future broadband networks [339], whilst the other explored the use of the GA/heuristic hybrid approach to cable-distribution network design [336].

An interesting and difficult network design problem, not considered in this thesis, is determining the location of network nodes, perhaps choosing these from a pre-determined list of sites [10, 11]. NOMaD's network model could also be extended appropriately to include this. Again, an M.Sc. project applying a GA/heuristic hybrid approach to this difficult area has already been supervised by the author [340].

Currently, the network model allows users to set an arbitrary granularity for the capacity of a wavelength channel (`net.lambdaCap` in Table E.2 on p. 246). However, all wavelength channels must have this same granularity. The model could instead be expanded to allow a mixture of channel granularities (*e.g.* both 2.5 Gbit/s and 10 Gbit/s) in the same network.

While NOMaD supports both primary and restoration path routing when used for topology design, only primary routes can be represented in fibre and wavelength allocation. The network model could, however, be extended to include the allocation of restoration paths by allowing wavelength-routed paths to be flagged as 'reserved' for restoration purposes. An alternative, which would better allow for sharing of restoration capacity (typically employed in practical algorithms), is to only reserve the capacity, without including actual paths in the path-loss sequence matrix.

Although static traffic matrices are appropriate for the design of multi-wavelength all-optical transport networks at present, in time 'point-and-click' (dynamic) provisioning of wavelength channels may well be required. NOMaD's network model could be modified to allow for dynamic traffic, although this would not be a straightforward change, as the assumption of static traffic pervades the present design.

In this thesis, three design metrics have been used, one based on network wavelength requirement (NWR) and the other two on simplified network cost models (§4.1). However, more detailed cost models could be developed, particularly with the co-operation of network vendors or service providers. These could be based, for example, on more explicit modelling of the structure of nodes and links, including enumerating ports, amplifiers, *etc.* [23]. In addition, more physical constraints could be included either in the network model or as penalties in the design assessment. These might include restrictions on *e.g.* the number of wavelengths on the fibres [302, 303], the number of ports on the OXCs [302, 303], or the maximum transmission distance allowed to a wavelength channel [301]. In addition, the use of adaptive rather than arbitrary fixed penalties could also be investigated.

E.8.4 Memory and Execution Time Limitations

One of the potential problems with the GA/heuristic hybrid approach that has already been identified (§1.4) is its large memory requirement. The GA framework used by NOMaD is a generational one, so two complete populations—old and new—are required in memory at once. Each individual contains a complete network model, comprising a large number of separate objects. For example, GA2 for mesh network topology design (§5.1.1) requires 27.6 Mbytes of memory. This is used for two populations of 200 individuals, each of 64 kbytes, plus an overhead for the GA framework and executable code of 2.7 Mbytes. GA18 for minimum cost routing, fibre and wavelength allocation (§5.2.2), uses 91.6 Mbytes for two populations of 500 individuals, each of 90 kbytes, as well as an overhead of 3.3 Mbytes. More memory is needed in the latter case not only to represent the wavelength-routed structure of the networks, but because of the larger population size required.

While the amount of memory required may seem large, a number of design decisions were taken to minimise the memory requirements. In particular, the number of wavelengths on a fibre (`Cable` class on p. 268), and the total number of clusters and levels in a network (§E.4), are limited to 32. This allows their compact representation using unsigned integers [16] as vectors of 32 bits; a greater range would have required the use of a true bit-vector class, with a corresponding growth in memory footprint. In a similar way, the `Cable` class alone is used to represent cables, fibres and wavelengths, rather than individual separate classes.

Nevertheless, there are additional areas where further savings in memory size could be made.

At present, for example, paths are implemented using array-lists of pointers to node objects. Each network design requires many path objects, and any reduction in their size would have a large impact on the overall memory requirement for an individual. A more compact representation would be to use arrays of node ids stored as unsigned characters. This would limit the number of nodes to 256, and require additional processing to retrieve the node objects, but would nevertheless reduce the memory requirement for simple paths by a factor of four (assuming 32-bit pointers). Additional savings could be made by other similar representation changes.

Clearly, however, there is a limit in additional memory savings that can be made, and as a result both the size of networks that can be examined and the number of individuals in the population are limited by available memory. The experiments described in Ch. 5 were carried out on an UltraSPARC 5 with 128 Mbytes of memory, which limited the size of the GA to less than, say, 100 Mbytes. Machine memory might be upgraded to 256 or 512 Mbytes, but this would allow only a modest increase in the network size, as only a few additional nodes causes a marked increase in the memory needed to represent a network design. Moreover, larger problems typically require even larger populations for 'good enough' results, further magnifying the problem.

One possible direction for further work in this area would be to investigate compact representations for the network designs, such as a binary string! However, it should be noted that considerable difficulty has been previously experienced by the author in designing a successful binary representation for routing, fibre and wavelength allocation [29]. The resulting GA could still use problem-specific genetic operators, either by designing these to work directly with the binary string representation, or by expanding the encoding to an object model representation before the operators are applied. Either approach would result in a significant memory saving, as even in the latter case only one or two networks would be 'expanded' at a time. However, they would probably both involve a marked increase in processing time over the current approach, as in the former case the operators would involve partial encoding and decoding of the binary representation, and in the latter, considerable time would be lost in memory allocation.

A complementary limitation of the GA/heuristic hybrid approach, although not one that any approach to these difficult network design problems can avoid, is the amount of computer time required. For example, while a single run of GA2 takes about 99 seconds, running GA18 once takes some 5 hours 50 minutes. Consequently, for example, the work reported on operator-probability adaptation in §5.3 (390 runs) is estimated to have taken more than 95 days non-stop computer time on an UltraSPARC 5.

Nevertheless, considerable efforts were taken during NOMaD's implementation to keep the processing time down. As well as the beneficial effects of some of the design decisions taken, such as using unsigned integers as bit vectors, employing class-specific memory management for key classes also provided a considerable improvement in execution time [16].

Overall then, taking both memory and time limitations into account, it must be concluded that there is only limited scalability in NOMaD's current implementation of the GA/heuristic hybrid approach to multi-wavelength all-optical network design. However, the design problems that have been addressed are complex and difficult ones, and a similar absence of scalability is to be expected from other approaches. However, one promising direction for future work would be to exploit the obvious parallelism of the GA framework, and develop a distributed version of NOMaD. Possibilities would include either distributed design assessment, controlled by a master GA on a single processor, or a distributed multi-population approach, with distinct small populations occasionally exchanging promising individuals [341]. In this way, both the memory and processing limitations of an individual machine might be overcome, and networks of larger size designed.

E.8.5 Additional Software

A piece of software that could be developed to complement the existing NOMaD toolset would be an interactive network editor. Although a prototype editor was produced by the author (see §3.7 and Fig. 3.8), the finished tool was never developed. However, a more complete implementation was made by two of the author's B.Eng. students working in Java [342, 343], and their efforts could in future form the basis of a fully-developed network editor.

Also, to allow for the optimal design of small networks[28], software employing NOMaD's network model could be written to generate linear programming (LP) formulations of topology and wavelength-allocation problems. These could then be tackled with an LP solver, *e.g.* `lp_solve` [344] or an equivalent commercial product. Indeed, an M.Sc. project addressing 'linearised' versions of the design problems considered in this thesis has already been supervised by the author [345].

[28]Larger problems are difficult to tackle with the exact methods of mathematical programming due to their large time and memory requirements [20, 23].

Appendix F

Detail of Experimental Results

This appendix provides detailed experimental results that were judged inappropriate for Ch. 5. It is divided into five sections: §F.1 describes how the network design problems used in the experimental work were generated. Then §F.2 tabulates the five network design problems actually used in the experimental work. The remaining three sections record detailed results on mesh network topology design (§F.3); routing, fibre choice and wavelength allocation (§F.4); and operator-probability adaptation (§F.5).

F.1 Network Design Problem Generation

In the telecommunications domain, one difficulty faced by researchers when developing new network design heuristics, is to find suitable network design problems to compare their work with that of others. Such an example problem would consist of a set of node locations, between which the network is to be formed, and a traffic (or capacity) matrix, representing the demands that the network is required to meet. The heuristics being developed (or evaluated) might, for example, include those for mesh network topology design with subsequent routing and wavelength allocation of a multi-wavelength all-optical transport network. For a network planner working in a telecommunications service provider, real node locations and measured traffic statistics for the company's own network would be available. For new networks, or extensions to existing networks, proposed locations and forecast statistics would be required. Such data are of great commercial value, and telecommunications service providers are consequently reluctant to make them available to independent researchers. In some cases proposed or hypothetical networks are available, such as the European Optical Network studied by COST 239 [291]. This used twenty European cities as node locations and appropriately scaled international telephony traffic obtained from the ITU, although the latter was still incomplete and had to be augmented by an empirical model [346]. It is the situation where appropriate example problems are unavailable, or a researcher requires a greater number or range of representative examples, that is addressed here. While there is an extensive literature on traffic estimation and forecasting (*e.g.* Hopkins *et al.* [347]), little has been published on telecommunications problem generation, with only Klingman *et al.* [348], Kershenbaum [12] and Cahn's work [10] known to the author.

F.1.1 Node Locations

There are two main possibilities for node locations in example problems: either existing geographical sites can be used, as in COST 239 [291], or locations can be selected at random within a total service area, *e.g.* Griffith *et al.* [313]. However, in the latter case, with repeated sampling from a uniform random distribution, there exists the danger of sites being selected that are unrealistically close together. This limitation can be addressed by a more controlled method of space filling. One potential approach, from the literature on statistical design of experiments, is *latin hypercubes*, first proposed in [349] and now widely used (see *e.g.* [350,351]). However, this was only recently brought to the author's attention[1], and the approach adopted here, based on the concept of a *minimum clear radius*, was developed independently.

Consider a network problem of N nodes, with a total service area A, and an expected service area per node, $a = A/N$. If, for the moment, we assume circular service areas, then the *expected service radius* is given by:

$$r = \sqrt{a/\pi} = \sqrt{A/(N\pi)} \tag{F.1}$$

This is thus slightly more than half the expected distance between each node and its nearest neighbour (ignoring the 'edge effect' caused by the finite total service area). The minimum clear radius is then taken to be:

$$r_{\min} = k_r \times r \tag{F.2}$$

where k_r is a constant, and usually $0 \leq k_r \leq 0.8$, as explained below.

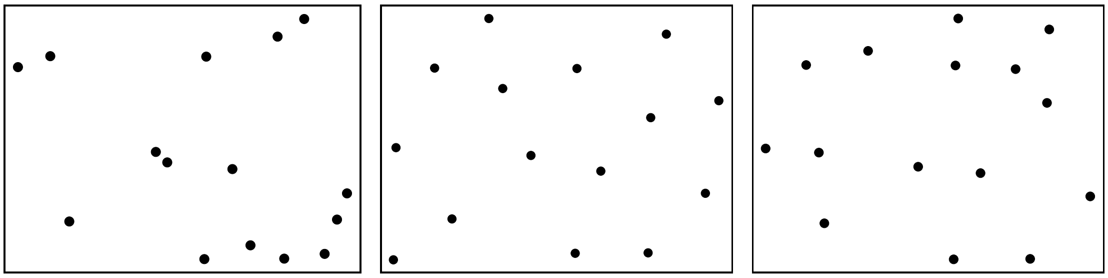
Node locations can now be be generated as follows. The location of node 0 is generated by sampling from a uniform distribution over area A. The subsequent locations ($1 \leq i \leq N-1$) are still generated using a uniform distribution, but each node location in turn is rejected and

[1]By Andy Keane, one of the examiners.

regenerated until $r_i \geq r_{\min}$, where r_i, the *clear radius* of node i, is:

$$r_i = \min_{\forall j \neq i}(d_{ij}/2) \tag{F.3}$$

and where d_{ij} is the (Euclidean) distance between nodes i and j. With the *minimum clear radius constant* $k_r = 0$, this is simply random selection of node locations (Fig. F.1(a)), but with $k_r = 0.8$, a near grid-like arrangement of nodes is obtained (Fig. F.1(b)). A value of $k_r = 0.5$ appears to produce a more realistic distribution, and is the default for the author's own work (Fig. F.1(c)). Values of $k_r > 0.8$ are not recommended using the simple resampling method described, as they will require a large number of iterations to generate the locations; in addition, the resulting node distributions are somewhat unrealistic, being constrained to a grid.



(a) $k_r = 0$ (b) $k_r = 0.8$ (c) $k_r = 0.5$

Figure F.1: Three fifteen-node problems generated with different k_r

F.1.2 Traffic Distribution

Given a figure for the total network traffic (or capacity), a method is needed to distribute this between all node pairs (or a limited subset thereof), such that the resulting traffic matrix is a good approximation to real-world situations. A simple uniform random (or Gaussian) distribution of traffic, without regard to the location of nodes, will not create the 'communities of interest' typical of real networks. Instead, the traffic distribution approach described here is based on that developed by the author for estimating European international telephony traffic [346], and was first presented in [313].

A *weighting factor* for each node pair in the network is calculated as follows:

$$W_{ij} = (r_i^2 \times r_j^2)/d_{ij}, \qquad i \neq j \tag{F.4}$$

The amount of traffic T_{ij} assigned to a node pair $(i,\, j)$ is then:

$$T_{ij} = (W_{ij} / \sum_{\forall i \neq j} W_{ij}) \times T \tag{F.5}$$

where T is the total traffic. Hence, the amount of traffic between each node pair is approximately proportional to the number of subscribers within a service radius (represented by the clear radius) around each of the end nodes, and inversely proportional to the distance between them (Fig. F.2).

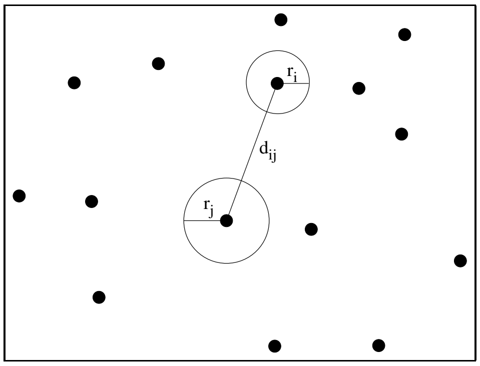
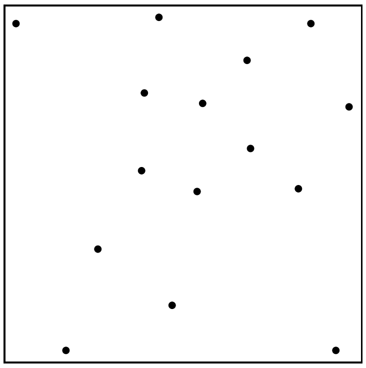


Figure F.2: Node-pair weighting factor calculation

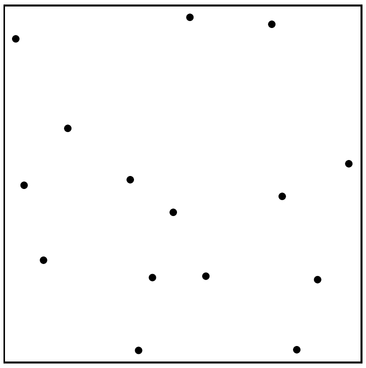
This approach assumes that the inverse distance relationship, found to be a reasonable approximation for international telephony traffic, is suitable for the traffic mix expected on the network being designed; if not, Eqn. F.4 can easily be modified to incorporate the anticipated distance dependency. In addition, another possible modification is the introduction of a small amount of (Gaussian) noise in generating either the weighting factors or the traffics themselves.

F.2 Network Design Problems

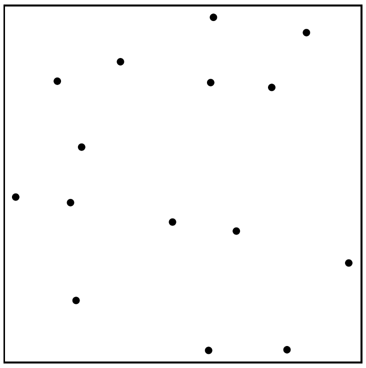
The node locations of the five fifteen-node network design problems, generated using the above method and used in §5.1, are both illustrated in Fig. F.3 and detailed in Tables F.1–F.5, with the coordinates in km from an arbitrary origin. The corresponding traffic matrices are given in Tables F.6–F.10.



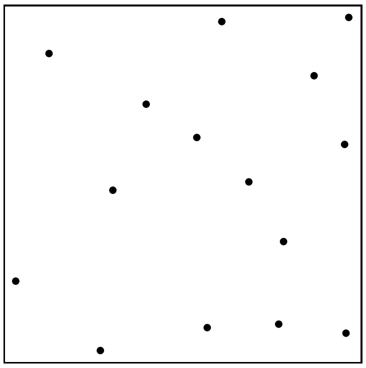
(a) Problem 1



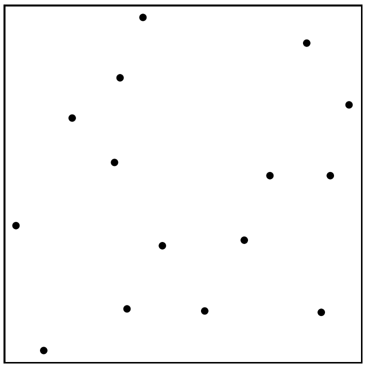
(b) Problem 2



(c) Problem 3



(d) Problem 4



(e) Problem 5

Figure F.3: Network design problems – node distributions

Node	x	y	Node	x	y
0	184.963	970.089	8	272.612	689.93
1	399.824	259.399	9	688.237	412.764
2	921.603	969.277	10	558.554	287.226
3	542.979	531.692	11	475.789	844.987
4	49.794	66.5666	12	391.042	472.429
5	819.319	523.871	13	679.526	168.088
6	853.394	67.1846	14	439.812	49.367
7	957.024	297.194			

Table F.1: Network design problem 1 – node locations

Node	x	y	Node	x	y
0	369.925	940.177	8	522.72	29.2276
1	799.649	518.797	9	571.719	736.444
2	843.205	938.555	10	998.591	429.283
3	85.9583	693.458	11	28.0581	488.135
4	159.411	333.484	12	345.844	473.75
5	768.234	48.4108	13	474.812	562.889
6	412.09	741.222	14	3.13337	87.1987
7	905.789	746.226			

Table F.2: Network design problem 2 – node locations

Node	x	y	Node	x	y
0	554.888	910.266	8	930.295	679.523
1	199.473	778.196	9	185.085	519.811
2	764.808	907.832	10	38.5772	504.265
3	628.938	595.075	11	214.456	373.646
4	149.382	199.7	12	724.512	215.422
5	457.956	571.612	13	319.437	148.101
6	560.183	201.554	14	567.147	30.0959
7	817.835	69.7889			

Table F.3: Network design problem 3 – node locations

Node	x	y	Node	x	y
0	739.85	880.355	8	903.157	379.948
1	599.297	37.5946	9	538.096	361.459
2	171.917	126.767	10	330.326	508.053
3	413.577	268.738	11	913.446	26.4141
4	828.095	188.776	12	564.17	889.717
5	90.4472	759.719	13	666.104	484.42
6	907.059	906.03	14	299.349	953.896
7	752.946	651.055			

Table F.4: Network design problem 4 – node locations

Node	x	y	Node	x	y
0	924.813	850.444	8	883.824	132.538
1	999.122	296.993	9	441.183	63.8193
2	608.013	846.387	10	378.947	224.935
3	714.896	658.459	11	172.621	951.823
4	248.97	332.833	12	948.302	486.114
5	96.5929	619.353	13	397.629	840.442
6	785.119	485.97	14	493.948	673.209
7	363.059	449.648			

Table F.5: Network design problem 5 – node locations

	To							
From	0	1	2	3	4	5	6	7
0	0	3.48041	28.2109	4.67568	16.6	3.78823	3.57941	6.18347
1	3.48041	0	7.10871	2.59262	11.4365	1.78551	2.46002	3.41731
2	28.2109	7.10871	0	11.0846	29.288	15.5817	10.7728	22.8053
3	4.67568	2.59262	11.0846	0	6.90465	3.28028	2.22152	4.1077
4	16.6	11.4365	29.288	6.90465	0	5.80517	8.85029	11.9654
5	3.78823	1.78551	15.5817	3.28028	5.80517	0	3.00873	8.18153
6	3.57941	2.46002	10.7728	2.22152	8.85029	3.00873	0	11.7723
7	6.18347	3.41731	22.8053	4.1077	11.9654	8.18153	11.7723	0
8	20.7896	4.09683	20.9339	6.01279	16.3052	3.6599	3.361	5.71205
9	3.91257	2.71065	11.8	4.83045	7.1551	5.8588	3.59739	7.41663
10	3.3198	4.83249	8.1056	3.2559	8.24158	2.5145	3.29398	4.78808
11	20.531	3.31738	34.0387	6.26122	12.9557	4.73639	3.5256	6.58389
12	4.91326	3.74075	8.82422	5.00864	8.82735	2.10229	2.0187	3.29897
13	3.04435	2.9381	8.30644	2.27951	7.95378	2.57128	6.69178	6.92201
14	4.76117	6.41144	10.616	2.84633	20.607	2.56515	5.15292	5.85751

	To							
From	8	9	10	11	12	13	14	
0	20.7896	3.91257	3.3198	20.531	4.91326	3.04435	4.76117	
1	4.09683	2.71065	20.531	3.31738	34.0387	6.26122	12.9557	
2	20.9339	11.8	8.1056	34.0387	8.82422	8.30644	10.616	
3	6.01279	4.83045	3.2559	6.26122	5.00864	2.27951	2.84633	
4	16.3052	7.1551	8.24158	12.9557	8.82735	7.95378	20.607	
5	3.6599	5.8588	2.5145	4.73639	2.10229	2.57128	2.56515	
6	3.361	3.59739	3.29398	3.5256	2.0187	6.69178	5.15292	
7	5.71205	7.41663	4.78808	6.58389	3.29897	6.92201	5.85751	
8	0	4.18596	3.72385	18.1009	7.60589	3.08525	4.89017	
9	4.18596	0	4.90572	4.62457	2.99162	4.01461	3.54067	
10	3.72385	4.90572	0	3.47403	3.19385	5.09149	5.15626	
11	18.1009	4.62457	3.47403	0	5.25084	3.07613	4.32944	
12	7.60589	2.99162	3.19385	5.25084	0	2.1113	3.29661	
13	3.08525	4.01461	5.09149	3.07613	2.1113	0	5.68844	
14	4.89017	3.54067	5.15626	4.32944	3.29661	5.68844	0	

Table F.6: Network design problem 1 – traffic (Gbit/s)

	To							
From	0	1	2	3	4	5	6	7
0	0	6.3571	6.9486	9.72611	5.15408	4.9919	10.0818	5.77075
1	6.3571	0	8.96659	5.72956	5.71384	11.8995	5.27966	15.0771
2	6.9486	8.96659	0	4.54633	3.58516	5.3979	4.277	16.0814
3	9.72611	5.72956	4.54633	0	9.9127	5.71345	6.84457	4.40464
4	5.15408	5.71384	3.58516	9.9127	0	7.21891	4.25471	3.83807
5	4.9919	11.8995	5.3979	5.71345	7.21891	0	3.85926	6.77956
6	10.0818	5.27966	4.277	6.84457	4.25471	3.85926	0	4.10727
7	5.77075	15.0771	16.0814	4.40464	3.83807	6.77956	4.10727	0
8	5.27834	9.97364	5.00121	6.74815	10.2409	29.0283	4.17233	5.93165
9	7.15029	7.4859	5.99148	4.62628	3.54007	4.20143	7.91675	6.06757
10	4.723	20.1798	7.10677	4.43091	4.50892	12.6031	3.55144	11.458
11	5.83994	4.93185	3.51491	17.0713	16.237	5.63684	4.43747	3.57801
12	4.23073	4.98555	2.87077	6.38872	8.4303	4.83321	4.42171	3.13825
13	5.0459	6.93556	3.71423	5.30038	5.04311	4.8917	6.44525	4.17262
14	7.36656	8.68726	5.6559	12.2995	23.3413	13.091	5.46776	6.05271

	To							
From	8	9	10	11	12	13	14	
0	5.27834	7.15029	4.723	5.83994	4.23073	5.0459	7.36656	
1	9.97364	7.4859	5.83994	4.93185	3.51491	17.0713	16.237	
2	5.00121	5.99148	7.10677	3.51491	2.87077	3.71423	5.6559	
3	6.74815	4.62628	4.43091	17.0713	6.38872	5.30038	12.2995	
4	10.2409	3.54007	4.50892	16.237	8.4303	5.04311	23.3413	
5	29.0283	4.20143	12.6031	5.63684	4.83321	4.8917	13.091	
6	4.17233	7.91675	3.55144	4.43747	4.42171	6.44525	5.46776	
7	5.93165	6.06757	11.458	3.57801	3.13825	4.17262	6.05271	
8	0	4.24076	9.02353	7.19227	6.0558	5.40718	19.1824	
9	4.24076	0	4.48607	3.41472	3.51687	6.12958	4.887	
10	9.02353	4.48607	0	3.91681	3.47505	4.20605	7.47683	
11	7.19227	3.41472	3.91681	0	6.18287	4.34215	16.9482	
12	6.0558	3.51687	3.47505	6.18287	0	7.48968	7.86787	
13	5.40718	6.12958	4.20605	4.34215	7.48968	0	6.06738	
14	19.1824	4.887	7.47683	16.9482	7.86787	6.06738	0	

Table F.7: Network design problem 2 – traffic (Gbit/s)

	To							
From	0	1	2	3	4	5	6	7
0	0	20.0743	23.8597	10.4555	4.38716	9.61011	4.36122	3.86114
1	20.0743	0	13.1229	11.0178	9.39288	15.5457	6.90539	5.49483
2	23.8597	13.1229	0	9.92741	3.82575	7.43681	4.20351	4.04934
3	10.4555	11.0178	9.92741	0	3.90286	13.2562	5.22912	4.11671
4	4.38716	9.39288	3.82575	3.90286	0	5.01956	5.39155	3.57813
5	9.61011	15.5457	7.43681	13.2562	5.01956	0	5.44111	3.72134
6	4.36122	6.90539	4.20351	5.22912	5.39155	5.44111	0	7.25065
7	3.86114	5.49483	4.04934	4.11671	3.57813	3.72134	7.25065	0
8	20.5076	18.6204	32.0476	19.5141	7.06505	12.605	9.22456	9.8941
9	4.5873	14.4856	3.53639	3.70343	5.48829	6.0028	3.09464	2.15682
10	3.75592	11.7999	2.96928	2.79129	5.45443	3.92522	2.52424	1.87704
11	3.97506	9.48202	3.29366	3.63302	9.74666	5.44018	4.03643	2.53838
12	4.32148	6.10245	4.45648	5.33576	3.84965	4.69546	11.5659	12.1309
13	4.49951	8.50322	4.07571	4.46178	14.4728	5.44389	8.98132	4.82953
14	3.80192	6.10077	3.71968	3.97955	5.31876	4.09433	12.0348	8.95099

	To							
From	8	9	10	11	12	13	14	
0	20.5076	4.5873	3.75592	3.97506	4.32148	4.49951	3.80192	
1	18.6204	14.4856	3.97506	9.48202	3.29366	3.63302	9.74666	
2	32.0476	3.53639	2.96928	3.29366	4.45648	4.07571	3.71968	
3	19.5141	3.70343	2.79129	3.63302	5.33576	4.46178	3.97955	
4	7.06505	5.48829	5.45443	9.74666	3.84965	14.4728	5.31876	
5	12.605	6.0028	3.92522	5.44018	4.69546	5.44389	4.09433	
6	9.22456	3.09464	2.52424	4.03643	11.5659	8.98132	12.0348	
7	9.8941	2.15682	1.87704	2.53838	12.1309	4.82953	8.95099	
8	0	5.83976	4.89743	5.85446	10.9841	7.99767	8.11453	
9	5.83976	0	8.24685	8.34516	2.4578	4.47256	2.6537	
10	4.89743	8.24685	0	5.67905	2.04539	3.89734	2.32122	
11	5.85446	8.34516	5.67905	0	2.91897	7.27611	3.42798	
12	10.9841	2.4578	2.04539	2.91897	0	5.39383	8.49426	
13	7.99767	4.47256	3.89734	7.27611	5.39383	0	8.74008	
14	8.11453	2.6537	2.32122	3.42798	8.49426	8.74008	0	

Table F.8: Network design problem 3 – traffic (Gbit/s)

	To							
From	0	1	2	3	4	5	6	7
0	0	4.27472	4.04297	1.68858	2.34375	5.981	8.21574	7.46624
1	4.27472	0	22.9635	10.3739	15.6582	11.7517	3.96408	7.1255
2	4.04297	22.9635	0	11.4638	6.80526	16.9924	3.56117	6.01457
3	1.68858	10.3739	11.4638	0	3.25985	5.66058	1.45223	2.82504
4	2.34375	15.6582	6.80526	3.25985	0	4.97925	2.2645	4.30464
5	5.981	11.7517	16.9924	5.66058	4.97925	0	4.76188	7.26023
6	8.21574	3.96408	3.56117	1.45223	2.2645	4.76188	0	5.75567
7	7.46624	7.1255	6.01457	2.82504	4.30464	7.26023	5.75567	0
8	3.89174	11.7602	7.26665	3.43644	11.7268	6.49095	3.89384	8.15475
9	2.10247	9.33258	7.38744	6.34991	4.07734	5.55292	1.77948	4.00507
10	5.63368	15.1198	20.7294	10.3637	6.19892	25.4914	4.44974	8.62251
11	1.87517	13.6601	5.99416	2.47733	10.4734	4.21356	1.85762	3.12602
12	8.54409	4.63166	4.80968	1.98122	2.35954	8.69768	4.37885	6.09473
13	3.79911	8.8997	6.88488	3.88026	5.33614	6.81571	3.15084	10.0465
14	8.07423	9.82827	11.8273	4.37208	4.55835	35.937	5.91532	8.15729

	To							
From	8	9	10	11	12	13	14	
0	3.89174	2.10247	5.63368	1.87517	8.54409	3.79911	8.07423	
1	11.7602	9.33258	1.87517	13.6601	5.99416	2.47733	10.4734	
2	7.26665	7.38744	20.7294	5.99416	4.80968	6.88488	11.8273	
3	3.43644	6.34991	10.3637	2.47733	1.98122	3.88026	4.37208	
4	11.7268	4.07734	6.19892	10.4734	2.35954	5.33614	4.55835	
5	6.49095	5.55292	25.4914	4.21356	8.69768	6.81571	35.937	
6	3.89384	1.77948	4.44974	1.85762	4.37885	3.15084	5.91532	
7	8.15475	4.00507	8.62251	3.12602	6.09473	10.0465	8.15729	
8	0	4.71993	7.82943	6.80961	3.61905	8.70561	6.3799	
9	4.71993	0	10.3271	2.73522	2.39356	7.25994	4.75457	
10	7.82943	10.3271	0	4.84699	7.53393	10.1976	18.101	
11	6.80961	2.73522	4.84699	0	1.89766	3.45591	3.81125	
12	3.61905	2.39356	7.53393	1.89766	0	3.95965	14.3123	
13	8.70561	7.25994	10.1976	3.45591	3.95965	0	6.66343	
14	6.3799	4.75457	18.101	3.81125	14.3123	6.66343	0	

Table F.9: Network design problem 4 – traffic (Gbit/s)

	To							
From	0	1	2	3	4	5	6	7
0	0	10.3285	20.4127	18.337	4.71041	17.4566	10.2602	5.81081
1	10.3285	0	4.54462	5.37594	2.53031	7.42228	6.64764	2.90508
2	20.4127	4.54462	0	12.8211	3.40048	14.2547	5.29935	4.56992
3	18.337	5.37594	12.8211	0	3.02346	10.3841	9.21669	4.20067
4	4.71041	2.53031	3.40048	3.02346	0	15.2393	2.36642	8.09108
5	17.4566	7.42228	14.2547	10.3841	15.2393	0	7.04269	15.6543
6	10.2602	6.64764	5.29935	9.21669	2.36642	7.04269	0	3.11479
7	5.81081	2.90508	4.56992	4.20067	8.09108	15.6543	3.11479	0
8	8.43706	14.3149	4.21264	4.70733	3.00266	8.08385	5.44031	3.27836
9	4.85873	3.51607	2.97956	2.93754	4.47093	8.46431	2.71127	3.75505
10	5.16948	3.25753	3.44324	3.35381	8.37056	10.9119	2.92494	6.2768
11	12.4905	4.26057	11.2453	6.59031	5.00812	34.2822	4.05383	5.81574
12	10.9696	9.69151	4.29425	5.91604	1.84304	5.72951	8.0758	2.25024
13	10.6232	3.27459	14.1423	6.56386	3.48915	18.496	3.50973	4.70411
14	12.0229	4.21434	14.3537	10.8416	4.4007	17.2282	5.3244	7.12394

	To							
From	8	9	10	11	12	13	14	
0	8.43706	4.85873	5.16948	12.4905	10.9696	10.6232	12.0229	
1	14.3149	3.51607	12.4905	4.26057	11.2453	6.59031	5.00812	
2	4.21264	2.97956	3.44324	11.2453	4.29425	14.1423	14.3537	
3	4.70733	2.93754	3.35381	6.59031	5.91604	6.56386	10.8416	
4	3.00266	4.47093	8.37056	5.00812	1.84304	3.48915	4.4007	
5	8.08385	8.46431	10.9119	34.2822	5.72951	18.496	17.2282	
6	5.44031	2.71127	2.92494	4.05383	8.0758	3.50973	5.3244	
7	3.27836	3.75505	6.2768	5.81574	2.25024	4.70411	7.12394	
8	0	4.99279	4.16813	4.35586	5.55458	3.25136	4.18887	
9	4.99279	0	9.15995	3.76703	2.23714	2.65465	3.37585	
10	4.16813	9.15995	0	4.42425	2.25449	3.20761	4.26805	
11	4.35586	3.76703	4.42425	0	3.44797	17.3785	10.2591	
12	5.55458	2.23714	2.25449	3.44797	0	2.81481	3.75115	
13	3.25136	2.65465	3.20761	17.3785	2.81481	0	13.3583	
14	4.18887	3.37585	4.26805	10.2591	3.75115	13.3583	0	

Table F.10: Network design problem 5 – traffic (Gbit/s)

F.3 Mesh Network Topology Design

F.3.1 Initial Topology Experiments

The non-default parameters values common to the three GAs discussed in §5.1.1 on p. 93 are listed in Table F.11. Typical topology results, showing progress and operator-probability adaptation over single runs of both GA1 and GA2 on one network design problem are illustrated in Fig. F.4. Equivalent traces for GA3, as well as a comparison of the lowest cost curves for all three GAs, were given in Fig. 5.1 on p. 95. The complete results from applying the GAs to all five of the network design problems are recorded in Tables F.12–F.14 (these were summarised in Table 5.2 on p. 97).

As an example of the application of the non-parametric sign test used throughout Ch. 5, consider Table F.15. In its second and third columns, this records the topology cost results from applying GA1 and GA2 to Problem 1, reproduced from the second columns of Tables F.12 and F.13, respectively. Each row in the table therefore represents two GA runs using the same random seed, and thus starting from exactly the same initial population; they differ only in the GA used. The fourth column of the table records whether, for each row, the result from GA1 was greater than that for GA2. Taking as the *null hypothesis* [307] that the two sets of results are drawn from the same distribution, and therefore that there is no median difference between the two GAs, the expected number of rows with GA1 greater than GA2 is 5 out of 10 (equivalent to tossing an unbiased coin). The actual distribution is binomial, with the probability that a GA1 result is greater, $p = 0.5$. Thus the probability of $(x_1, x_2) = (0, 10)$, where x_1 is the number of rows in which GA1 is greater and x_2 the number of rows in which GA2 is greater, is:

$$\text{prob}[(x_1, x_2) = (0, 10)] = \begin{pmatrix} 10 \\ 0 \end{pmatrix} p^0 p^{10} = p^{10} = 0.098\% \tag{F.6}$$

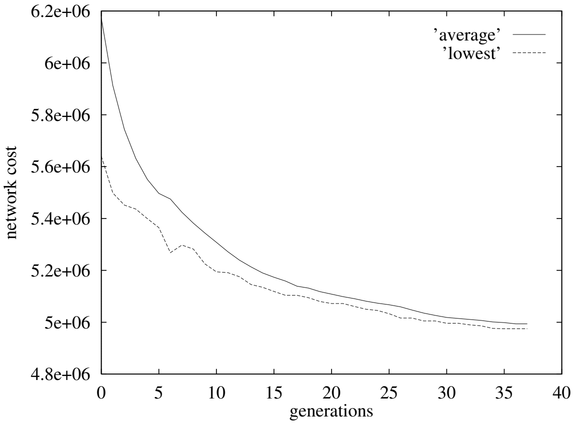
Similarly, for $(x_1, x_2) = (1, 9)$, $(2, 8)$ and $(3, 7)$:

$$\text{prob}[(x_1, x_2) = (1, 9)] = \begin{pmatrix} 10 \\ 1 \end{pmatrix} p^1 p^9 = 10p^{10} = 0.977\% \tag{F.7}$$

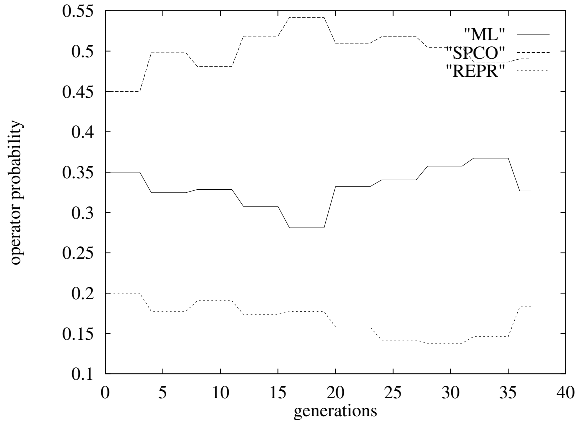
$$\text{prob}[(x_1, x_2) = (2, 8)] = \begin{pmatrix} 10 \\ 2 \end{pmatrix} p^2 p^8 = 45p^{10} = 4.39\% \tag{F.8}$$

Parameter	Value	Description
`ales.eMin`	`70.0`	Constant K_0, shared with `AlesRouter`
`ales.eLink`	`35.0`	Constant K_n, shared with `AlesRouter`
`ales.nnPenalty`	`200,000.0`	Penalty P_N
`ales.apPenalty`	`100,000.0`	Penalty P_R
`maxTrials`	`5,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`200`	Population size
`pop.select`	`tournament`	Selection operator

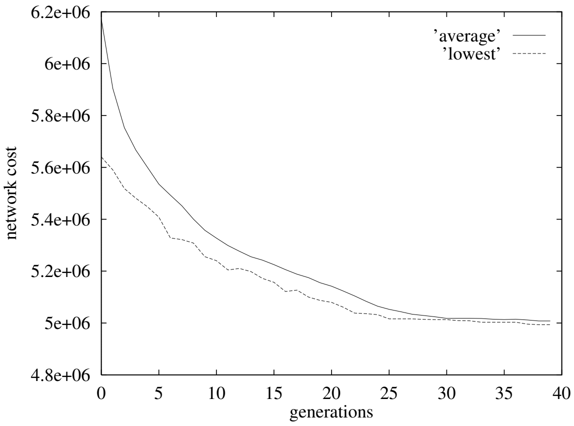
Table F.11: Non-default parameters for GA1, GA2 and GA3



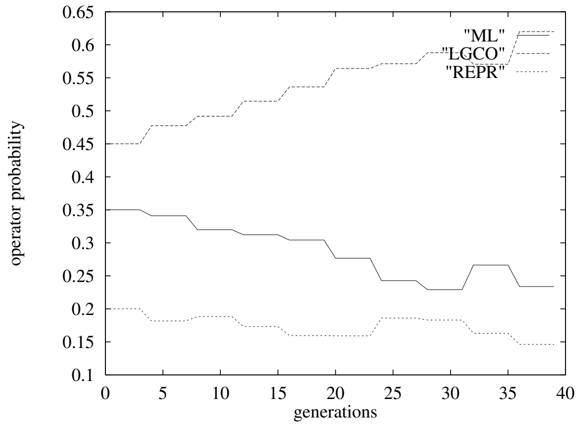
(a) Progress over a single GA1 run



(b) Operator-probability adaptation for GA1



(c) Progress over a single GA2 run



(d) Operator-probability adaptation for GA2

Figure F.4: Typical initial topology results (Problem 1, Run 1)

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.975	4.806	4.487	4.619	4.522
2	4.997	4.803	4.492	4.649	4.489
3	4.982	4.786	4.469	4.635	4.461
4	4.990	4.793	4.489	4.646	4.471
5	4.999	4.785	4.450	4.635	4.481
6	5.017	4.785	4.478	4.706	4.487
7	5.017	4.772	4.446	4.658	4.501
8	4.982	4.773	4.485	4.618	4.486
9	5.029	4.801	4.487	4.652	4.479
10	4.982	4.818	4.486	4.634	4.477
Best	4.975	4.772	4.446	4.618	4.461
Median	4.994	4.789	4.486	4.641	4.484

Table F.12: Topology cost results for GA1

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.994	4.785	4.416	4.619	4.463
2	4.954	4.774	4.443	4.610	4.449
3	4.971	4.764	4.415	4.619	4.467
4	4.978	4.751	4.425	4.607	4.427
5	4.993	4.752	4.434	4.605	4.448
6	4.953	4.751	4.423	4.616	4.448
7	4.982	4.781	4.417	4.613	4.424
8	4.969	4.765	4.426	4.602	4.450
9	4.959	4.775	4.419	4.593	4.451
10	5.005	4.761	4.418	4.618	4.496
Best	4.953	4.751	4.415	4.593	4.424
Median	4.975	4.764	4.421	4.612	4.450

Table F.13: Topology cost results for GA2

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.960	4.782	4.425	4.621	4.441
2	4.990	4.755	4.432	4.608	4.487
3	4.999	4.782	4.435	4.591	4.467
4	4.958	4.776	4.404	4.611	4.445
5	4.977	4.760	4.448	4.604	4.459
6	4.969	4.772	4.417	4.621	4.452
7	4.954	4.767	4.439	4.625	4.438
8	4.994	4.756	4.429	4.618	4.469
9	4.976	4.767	4.422	4.606	4.455
10	4.978	4.768	4.459	4.621	4.452
Best	4.954	4.755	4.404	4.591	4.438
Median	4.977	4.767	4.430	4.614	4.454

Table F.14: Topology cost results for GA3

Run	GA1 ($\times 10^6$)	GA2 ($\times 10^6$)	GA1 > GA2 ?
1	4.975	4.994	0
2	4.997	4.954	1
3	4.982	4.971	1
4	4.990	4.978	1
5	4.999	4.993	1
6	5.017	4.953	1
7	5.017	4.982	1
8	4.982	4.969	1
9	5.029	4.959	1
10	4.982	5.005	0
Best	4.975	4.953	
Median	4.994	4.975	

Table F.15: Applying sign test to topology cost results for GA1 and GA2 on Problem 1

$$\text{prob}[(x_1, x_2) = (3, 7)] = \begin{pmatrix} 10 \\ 3 \end{pmatrix} p^3 p^7 = 120p^{10} = 11.72\% \tag{F.9}$$

Consequently, the cumulative prob$[x_1 \leq 0] = 0.1\%$, prob$[x_1 \leq 1] = 1.1\%$ and prob$[x_1 \leq 2] = 5.5\%$. Now, in hypothesis testing, if an event occurs of sufficiently low probability, given the null hypothesis, this is taken to be significant, and the null hypothesis rejected. In this case, that would indicate there was indeed a median difference between the two GAs. Conventionally, a probability of 0.1% is taken as very highly significant, 1% as highly significant, and 5% as significant; values greater than 5% are not significant[2]. In this thesis, as comparisons were made on the basis of 10 runs, 1.1% was taken as highly significant, and 5.5% as significant. Thus it is these cumulative probabilities which are recorded in the appropriate tables in Ch. 5. For example, in Table F.15, $x_1 = 2$, and so a value of 5.5% is recorded for $s_{1,2}$, the significance level of the comparison of GA1 with GA2, in Table 5.2 on p. 97. Using a non-parametric sign test in this way allows median differences to be detected using only 10 runs where a much greater number would be needed if a Gaussian-based test were used instead. In addition, a non-parametric test does not assume an underlying Gaussian distribution in the samples (even in the limit), which in any case is unlikely here, given that they are all final results from a minimisation process, and consequently their distributions are very likely to be skewed (asymmetric), with a longer tail on the upper side.

[2]Strictly, the rejection of the null hypothesis at this level—a one-tail test—does not indicate whether it is GA1 which is better than GA2 or not. Significance levels only half those indicated would be needed for such a two-tail test; however, this technicality has not been observed in the thesis.

F.3.2 Test Networks

The non-default parameters values used with GA2 and GA3 to generate the two additional test networks described in §5.1.2 are listed in Tables F.16 and F.17. The complete results from then applying the GAs employing the two parameter sets to Problem 1 are detailed in Tables F.18 and F.19. The particular GA runs selected to provide both the original five test networks and the two additional ones (illustrated in Fig. 5.3 on p. 99 and Fig. 5.4 on p. 99, respectively) are recorded in Table F.20. In addition, a list of network links for all seven test networks (using the node ids introduced in §F.2) is given in Table F.21.

Parameter	Value	Description
`ales.eMin`	`70.0`	Constant K_0, shared with `AlesRouter`
`ales.eLink`	`35.0`	Constant K_n, shared with `AlesRouter`
`ales.nnPenalty`	`200,000.0`	Penalty P_N
`ales.apPenalty`	`100,000.0`	Penalty P_R
`ales.alpha`	`0.8`	Constant α
`maxTrials`	`5,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`200`	Population size
`pop.select`	`tournament`	Selection operator

Table F.16: Reduced α parameter set for GA2 and GA3

Parameter	Value	Description
`ales.eMin`	`100.0`	Constant K_0, shared with `AlesRouter`
`ales.eLink`	`800.0`	Constant K_n, shared with `AlesRouter`
`ales.nnPenalty`	`1,000,000.0`	Penalty P_N
`ales.apPenalty`	`500,000.0`	Penalty P_R
`ales.alpha`	`0.8`	Constant α
`maxTrials`	`5,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`200`	Population size
`pop.select`	`tournament`	Selection operator

Table F.17: Increased node cost parameter set for GA2 and GA3

GA	Run				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)
2	2.981	2.991	2.972	3.011	2.990
3	2.982	2.987	2.990	2.986	3.002

Table F.18: Reduced α topology results for Problem 1

GA	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)
2	30.32	30.78	30.54	30.38	30.91
3	30.71	30.58	30.31	30.48	30.25

Table F.19: Increased node cost topology results for Problem 1

Network	GA	Run	Cost ($\times 10^6$)
1	2	2	4.954
2	2	4	4.751
3	3	4	4.404
4	3	3	4.591
5	2	4	4.427
6	2	3	2.972
7	3	5	30.25

Table F.20: Test network selection

Network	Links							
1	(0,2)	(0,4)	(0,8)	(0,11)	(1,4)	(1,10)	(1,12)	(1,14)
	(2,5)	(2,7)	(2,8)	(2,9)	(2,11)	(2,12)	(3,5)	(3,8)
	(3,9)	(3,11)	(3,12)	(4,8)	(4,12)	(4,13)	(4,14)	(5,6)
	(5,7)	(5,9)	(6,7)	(6,13)	(6,14)	(7,9)	(7,13)	(8,11)
	(8,12)	(9,10)	(9,11)	(9,13)	(10,12)	(10,13)	(10,14)	(13,14)
2	(0,2)	(0,3)	(0,6)	(0,9)	(1,2)	(1,4)	(1,5)	(1,7)
	(1,9)	(1,10)	(1,13)	(2,7)	(2,9)	(3,4)	(3,6)	(3,11)
	(4,8)	(4,11)	(4,12)	(4,14)	(5,8)	(5,10)	(5,13)	(5,14)
	(6,9)	(6,12)	(6,13)	(7,9)	(7,10)	(8,10)	(8,12)	(8,14)
	(9,12)	(9,13)	(11,13)	(11,14)	(12,13)	(12,14)		
3	(0,1)	(0,2)	(0,3)	(0,5)	(0,8)	(1,2)	(1,5)	(1,9)
	(1,10)	(2,3)	(2,8)	(3,5)	(3,6)	(3,8)	(3,13)	(4,10)
	(4,11)	(4,13)	(4,14)	(5,9)	(5,11)	(5,12)	(6,7)	(6,9)
	(6,12)	(6,13)	(6,14)	(7,8)	(7,12)	(7,14)	(8,11)	(8,12)
	(9,10)	(9,11)	(10,11)	(11,13)	(12,14)	(13,14)		
4	(0,1)	(0,6)	(0,7)	(0,12)	(0,14)	(1,2)	(1,3)	(1,4)
	(1,9)	(1,11)	(2,3)	(2,5)	(2,10)	(3,4)	(3,9)	(3,10)
	(4,8)	(4,11)	(4,13)	(5,10)	(5,12)	(5,14)	(6,7)	(6,8)
	(6,12)	(7,8)	(7,9)	(7,12)	(7,13)	(8,11)	(8,13)	(9,10)
	(9,13)	(10,13)	(10,14)	(12,14)	(13,14)			
5	(0,2)	(0,3)	(0,6)	(0,12)	(0,13)	(1,6)	(1,8)	(1,12)
	(1,14)	(2,3)	(2,11)	(2,13)	(2,14)	(3,6)	(3,12)	(3,14)
	(4,5)	(4,7)	(4,9)	(4,10)	(5,7)	(5,11)	(5,13)	(5,14)
	(6,7)	(6,9)	(6,12)	(7,10)	(7,14)	(8,9)	(8,10)	(8,12)
	(9,10)	(10,11)	(11,13)	(13,14)				
6	(0,2)	(0,4)	(0,8)	(0,11)	(1,4)	(1,10)	(1,12)	(1,14)
	(2,5)	(2,7)	(2,9)	(2,11)	(3,6)	(3,8)	(3,9)	(3,12)
	(4,8)	(4,14)	(5,7)	(5,9)	(5,13)	(6,7)	(6,10)	(6,13)
	(7,13)	(8,11)	(9,10)	(10,14)	(11,12)	(13,14)		
7	(0,2)	(0,4)	(0,8)	(0,11)	(1,5)	(1,10)	(1,14)	(2,4)
	(2,6)	(2,9)	(3,8)	(3,12)	(3,14)	(4,10)	(4,14)	(5,6)
	(5,8)	(6,7)	(6,13)	(7,10)	(7,12)	(8,11)	(9,11)	(9,12)
	(10,13)							

Table F.21: Test network links

F.3.3 Local Search Operators

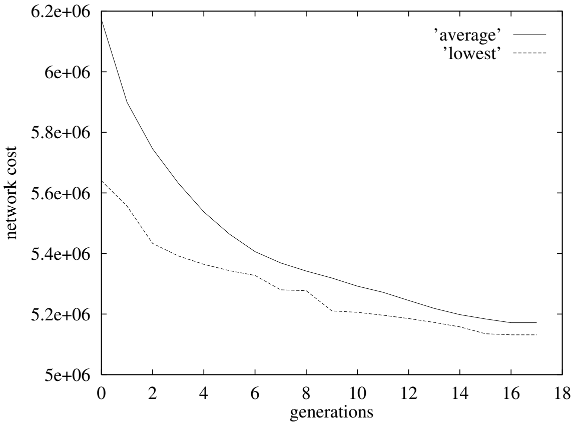
The non-default parameters values common to the nine GAs discussed in §5.1.3 on p. 100 are listed in Table F.22. Typical topology results, showing progress and operator-probability adaptation over single runs of GA5-GA12 on one network design problem are illustrated in Figures F.5–F.8. Equivalent traces for GA4, as well as a comparison of the lowest cost curves for five of the GAs, were given in Fig. 5.5 on p. 102. (It should be noted that Fig. 5.5(c) and (d) have been repeated here as Fig. F.7(b) and (d) for ease of reference.) The complete results from applying the GAs to all five of the network design problems are recorded in Tables F.23–F.31 (these were summarised in Tables 5.5 and 5.6 on p. 103).

Parameter	Value	Description
`ales.eMin`	`70.0`	Constant K_0, shared with `AlesRouter`
`ales.eLink`	`35.0`	Constant K_n, shared with `AlesRouter`
`ales.nnPenalty`	`200,000.0`	Penalty P_N
`ales.apPenalty`	`100,000.0`	Penalty P_R
`maxTrials`	`15,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`200`	Population size
`pop.select`	`tournament`	Selection operator
`opp.fitCredDiv`	—	Credit divisor: `children` or `trials`

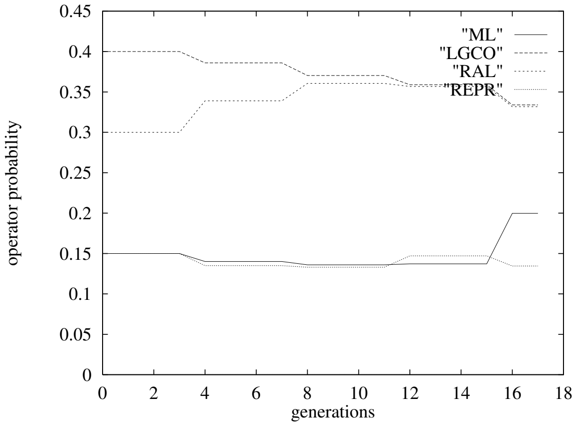
Table F.22: Non-default parameters for GA4–GA12

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.960	4.746	4.413	4.605	4.429
2	4.953	4.756	4.442	4.598	4.449
3	4.970	4.761	4.409	4.596	4.430
4	4.947	4.746	4.425	4.604	4.427
5	4.963	4.752	4.434	4.591	4.430
6	4.953	4.751	4.414	4.609	4.443
7	4.957	4.765	4.409	4.605	4.424
8	4.969	4.765	4.409	4.592	4.441
9	4.956	4.755	4.412	4.593	4.428
10	4.967	4.752	4.410	4.607	4.446
Best	4.947	4.746	4.409	4.591	4.424
Median	4.958	4.753	4.412	4.601	4.430

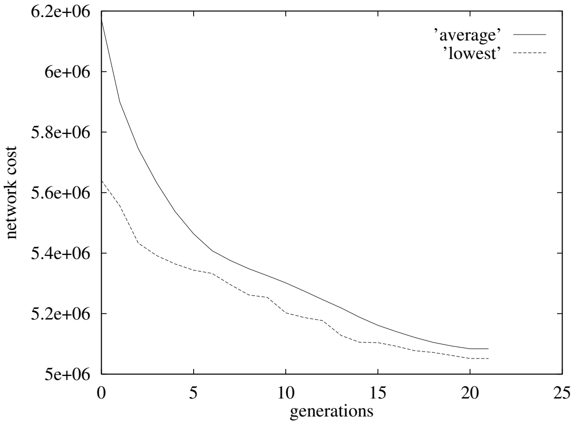
Table F.23: Topology cost results for GA4



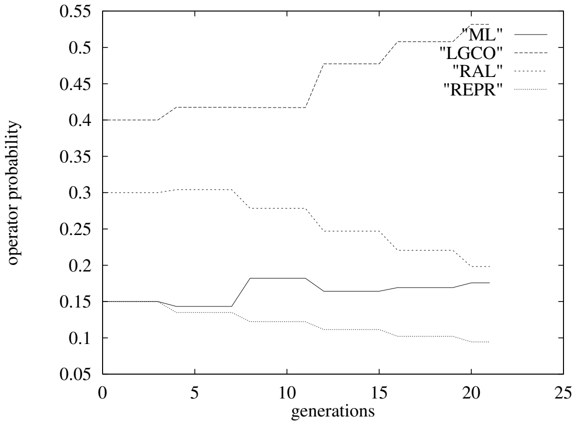
(a) Progress over a single GA5 run



(b) Operator-probability adaptation for GA5



(c) Progress over a single GA6 run

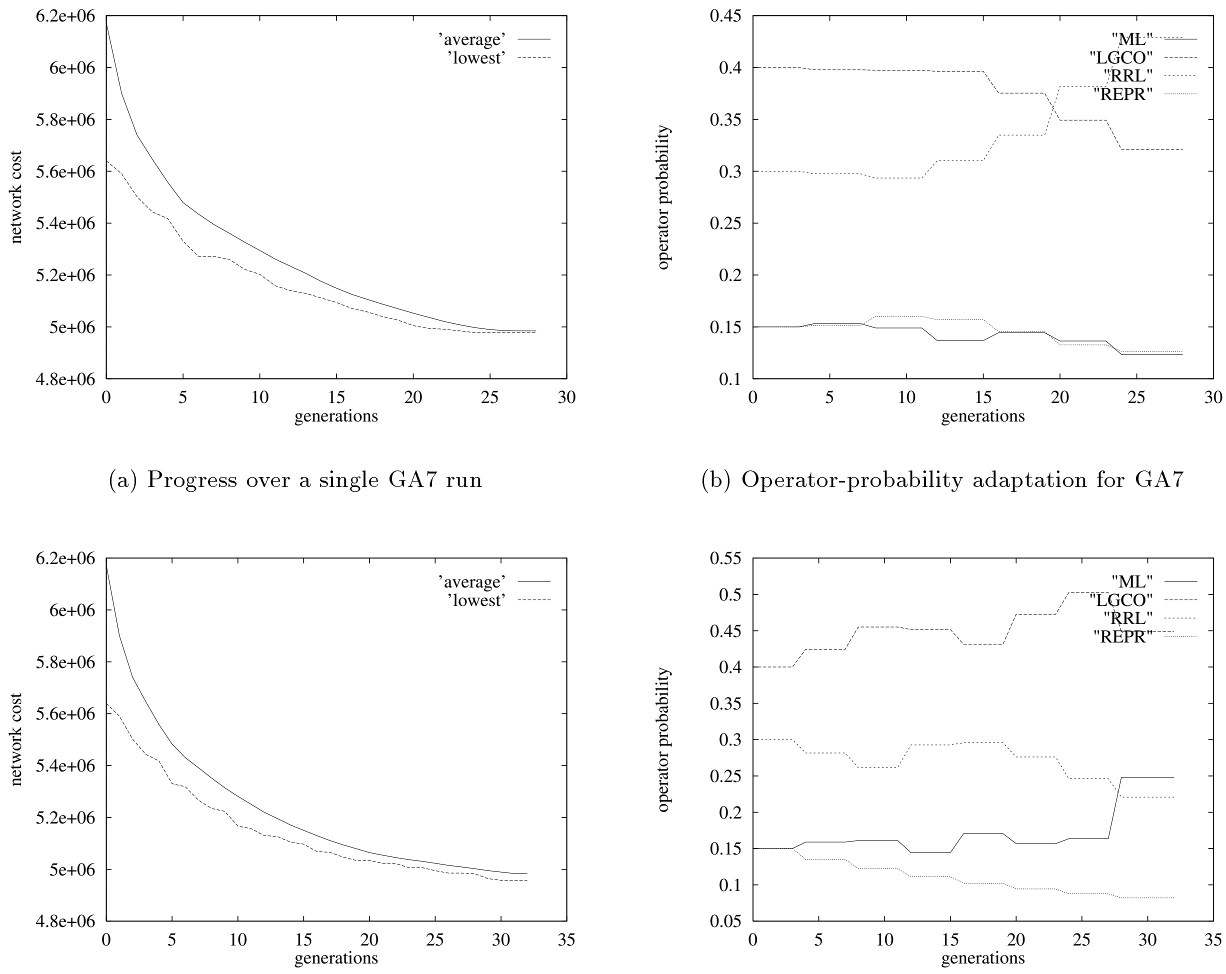


(d) Operator-probability adaptation for GA6

Figure F.5: Typical topology results for GA5 and GA6 (Problem 1, Run 1)

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	5.132	4.838	4.506	4.719	4.531
2	5.073	4.858	4.542	4.735	4.573
3	5.092	4.899	4.564	4.716	4.555
4	5.050	4.899	4.556	4.716	4.566
5	5.087	4.844	4.474	4.713	4.584
6	5.019	4.883	4.544	4.692	4.628
7	4.988	4.878	4.604	4.766	4.549
8	5.046	4.917	4.485	4.694	4.513
9	5.090	4.849	4.536	4.725	4.617
10	5.042	4.864	4.529	4.748	4.579
Best	4.988	4.838	4.474	4.692	4.513
Median	5.062	4.871	4.539	4.718	4.569

Table F.24: Topology cost results for GA5



(a) Progress over a single GA7 run

(b) Operator-probability adaptation for GA7

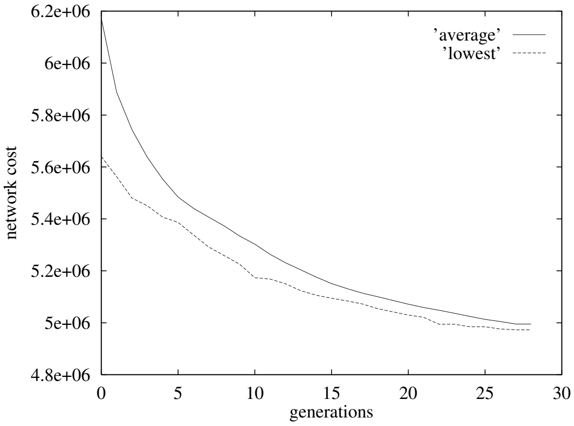
(c) Progress over a single GA8 run

(d) Operator-probability adaptation for GA8

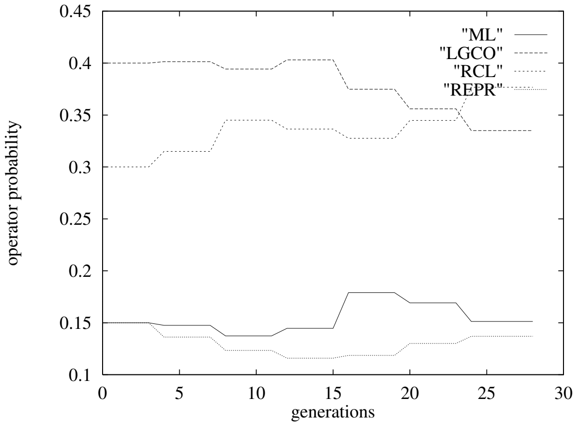
Figure F.6: Typical topology results for GA7 and GA8 (Problem 1, Run 1)

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	5.052	4.857	4.480	4.702	4.452
2	5.037	4.838	4.537	4.732	4.536
3	5.087	4.850	4.510	4.670	4.558
4	5.008	4.842	4.533	4.616	4.553
5	5.034	4.855	4.466	4.674	4.524
6	5.006	4.831	4.485	4.641	4.611
7	4.987	4.816	4.502	4.693	4.479
8	5.003	4.872	4.495	4.666	4.491
9	5.059	4.812	4.497	4.719	4.530
10	4.996	4.809	4.487	4.647	4.486
Best	4.987	4.809	4.466	4.616	4.452
Median	5.021	4.840	4.496	4.672	4.527

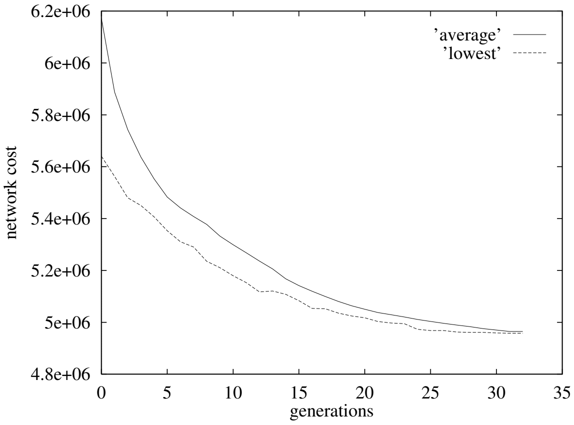
Table F.25: Topology cost results for GA6



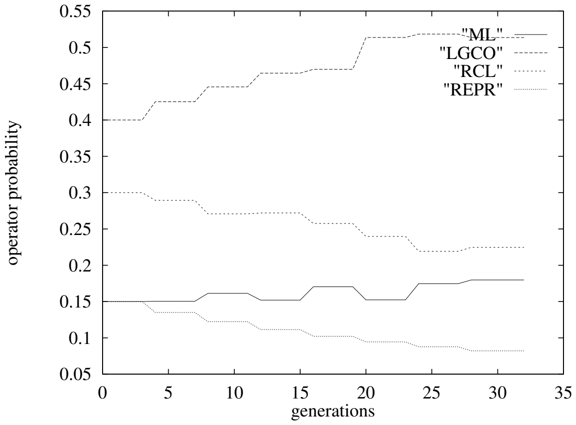
(a) Progress over a single GA9 run



(b) Operator-probability adaptation for GA9



(c) Progress over a single GA10 run

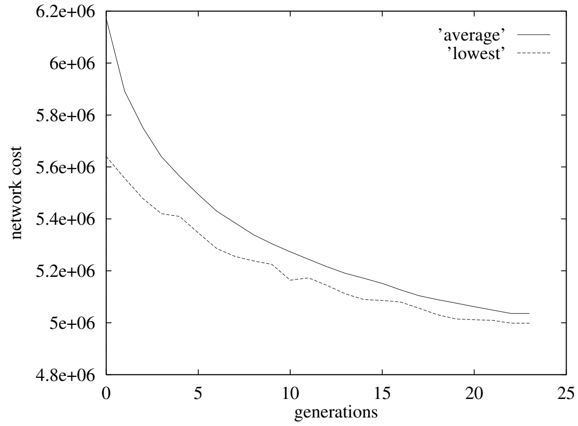


(d) Operator-probability adaptation for GA10

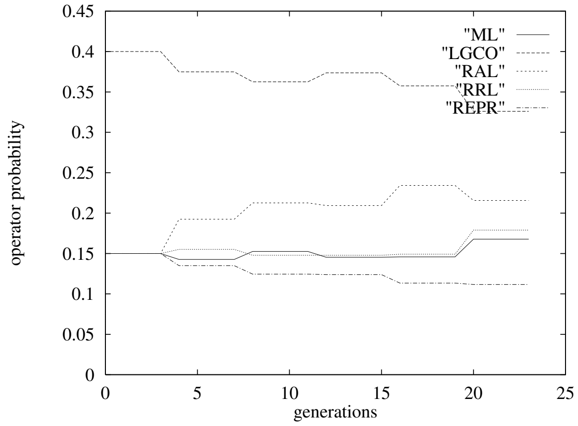
Figure F.7: Typical topology results for GA9 and GA10 (Problem 1, Run 1)

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.978	4.747	4.426	4.596	4.448
2	4.953	4.755	4.425	4.609	4.471
3	4.988	4.779	4.433	4.600	4.445
4	4.993	4.770	4.421	4.607	4.459
5	4.972	4.780	4.418	4.592	4.439
6	4.961	4.762	4.436	4.622	4.441
7	4.983	4.753	4.458	4.592	4.456
8	4.984	4.759	4.423	4.607	4.466
9	4.965	4.786	4.424	4.649	4.468
10	5.016	4.774	4.433	4.659	4.435
Best	4.953	4.747	4.418	4.592	4.435
Median	4.980	4.766	4.425	4.607	4.452

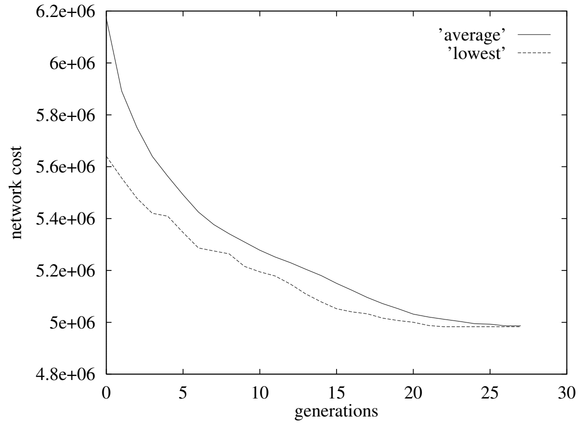
Table F.26: Topology cost results for GA7



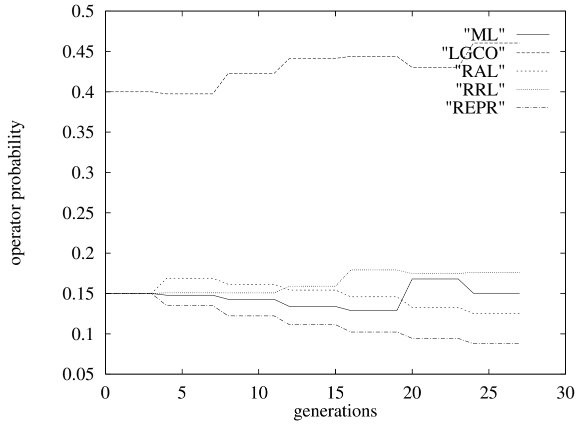
(a) Progress over a single GA11 run



(b) Operator-probability adaptation for GA11



(c) Progress over a single GA12 run



(d) Operator-probability adaptation for GA12

Figure F.8: Typical topology results for GA11 and GA12 (Problem 1, Run 1)

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.956	4.745	4.407	4.609	4.454
2	4.968	4.779	4.439	4.622	4.468
3	4.970	4.765	4.438	4.602	4.451
4	4.989	4.759	4.421	4.607	4.477
5	4.962	4.764	4.418	4.600	4.434
6	4.961	4.767	4.438	4.598	4.437
7	4.964	4.761	4.407	4.602	4.434
8	4.991	4.762	4.423	4.616	4.487
9	4.975	4.794	4.413	4.623	4.479
10	4.998	4.751	4.409	4.611	4.446
Best	4.956	4.745	4.407	4.598	4.434
Median	4.969	4.763	4.420	4.608	4.452

Table F.27: Topology cost results for GA8

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.973	4.753	4.434	4.618	4.438
2	4.980	4.756	4.414	4.607	4.439
3	4.952	4.767	4.423	4.599	4.438
4	4.962	4.757	4.400	4.594	4.454
5	4.962	4.763	4.439	4.598	4.430
6	4.952	4.759	4.419	4.607	4.447
7	4.964	4.763	4.412	4.608	4.461
8	4.973	4.761	4.418	4.601	4.459
9	4.950	4.750	4.423	4.595	4.441
10	4.947	4.747	4.452	4.612	4.417
Best	4.947	4.747	4.400	4.594	4.417
Median	4.962	4.758	4.421	4.604	4.440

Table F.28: Topology cost results for GA9

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.958	4.748	4.415	4.612	4.448
2	4.953	4.760	4.408	4.592	4.440
3	4.946	4.750	4.416	4.589	4.426
4	4.957	4.754	4.409	4.594	4.435
5	4.962	4.755	4.422	4.595	4.467
6	4.963	4.758	4.414	4.607	4.444
7	4.965	4.768	4.409	4.601	4.434
8	4.949	4.752	4.409	4.607	4.451
9	4.955	4.772	4.420	4.606	4.427
10	4.979	4.748	4.427	4.602	4.436
Best	4.946	4.748	4.408	4.589	4.426
Median	4.957	4.755	4.414	4.602	4.438

Table F.29: Topology cost results for GA10

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.998	4.750	4.450	4.686	4.439
2	4.989	4.777	4.459	4.628	4.515
3	5.073	4.772	4.447	4.710	4.474
4	4.978	4.761	4.452	4.619	4.467
5	5.001	4.875	4.466	4.625	4.455
6	4.970	4.766	4.484	4.637	4.508
7	5.015	4.792	4.484	4.634	4.470
8	4.962	4.810	4.490	4.641	4.478
9	4.983	4.847	4.462	4.676	4.536
10	4.957	4.772	4.464	4.686	4.471
Best	4.957	4.750	4.447	4.619	4.439
Median	4.986	4.774	4.463	4.639	4.473

Table F.30: Topology cost results for GA11

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.983	4.754	4.417	4.621	4.458
2	4.951	4.767	4.423	4.616	4.478
3	4.981	4.757	4.415	4.642	4.472
4	4.976	4.755	4.440	4.615	4.451
5	4.972	4.798	4.434	4.662	4.433
6	4.980	4.753	4.421	4.639	4.460
7	4.974	4.758	4.433	4.593	4.423
8	4.954	4.768	4.446	4.616	4.466
9	4.978	4.799	4.427	4.608	4.490
10	4.975	4.760	4.477	4.602	4.445
Best	4.951	4.753	4.415	4.593	4.423
Median	4.975	4.759	4.430	4.616	4.459

Table F.31: Topology cost results for GA12

F.3.4 Iterative Heuristic Operators

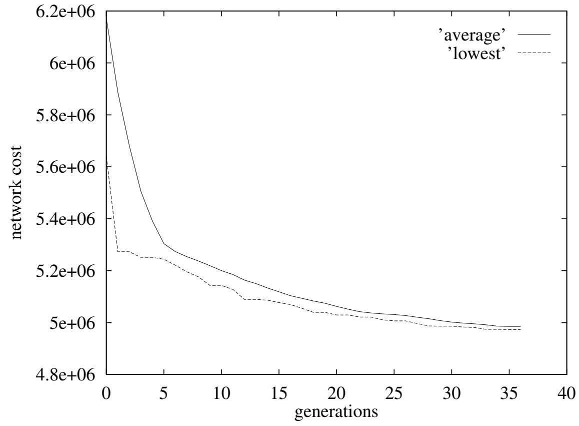
The non-default parameters values common to the three GAs discussed in §5.1.4 on p. 105 are listed in Table F.32. Typical topology results, showing progress and operator-probability adaptation over single runs of GA13-GA15 on one network design problem are illustrated in Fig. F.9. A comparison of the lowest cost curves for all three of the GAs was given in Fig. 5.6 on p. 107. In addition, the complete results from applying the GAs to all five of the network design problems are recorded in Tables F.33–F.35 (these were summarised in Table 5.11 on p. 107).

Parameter	Value	Description
`ales.eMin`	`70.0`	Constant K_0, shared with `AlesRouter`
`ales.eLink`	`35.0`	Constant K_n, shared with `AlesRouter`
`ales.nnPenalty`	`200,000.0`	Penalty P_N
`ales.apPenalty`	`100,000.0`	Penalty P_R
`maxTrials`	`40,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`200`	Population size
`pop.select`	`tournament`	Selection operator
`opp.fitCredDiv`	`trials`	Credit divisor

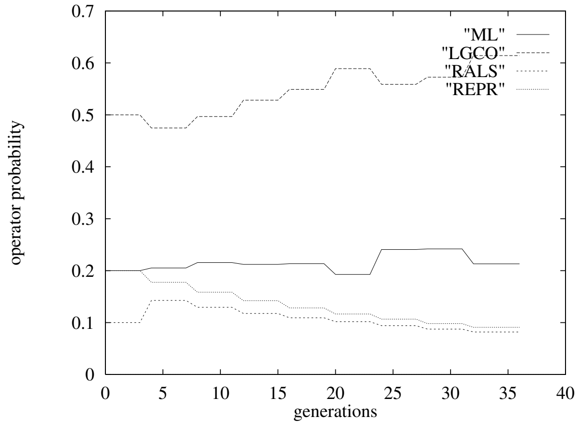
Table F.32: Non-default parameters for GA13–GA15

Run	Problem				
	1	2	3	4	5
	($\times 10^6$)	($\times 10^6$)	($\times 10^6$	($\times 10^6$)	($\times 10^6$)
1	4.973	4.752	4.444	4.607	4.430
2	4.956	4.755	4.431	4.591	4.434
3	4.949	4.760	4.437	4.607	4.459
4	4.959	4.753	4.408	4.595	4.419
5	4.936	4.758	4.426	4.597	4.443
6	4.952	4.756	4.421	4.603	4.440
7	4.960	4.749	4.420	4.593	4.445
8	4.947	4.758	4.434	4.604	4.435
9	4.947	4.747	4.427	4.604	4.450
10	4.951	4.759	4.419	4.605	4.441
Best	4.936	4.747	4.408	4.591	4.419
Median	4.951	4.755	4.426	4.603	4.441

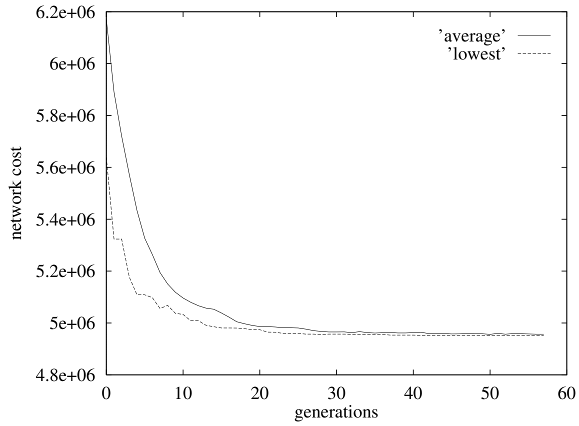
Table F.33: Topology cost results for GA13



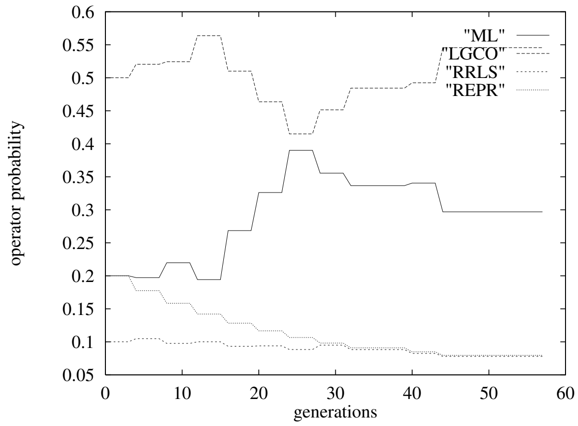
(a) Progress over a single GA13 run



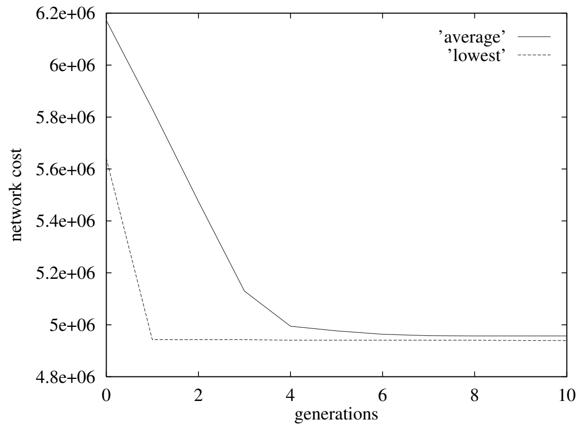
(b) Operator-probability adaptation for GA13



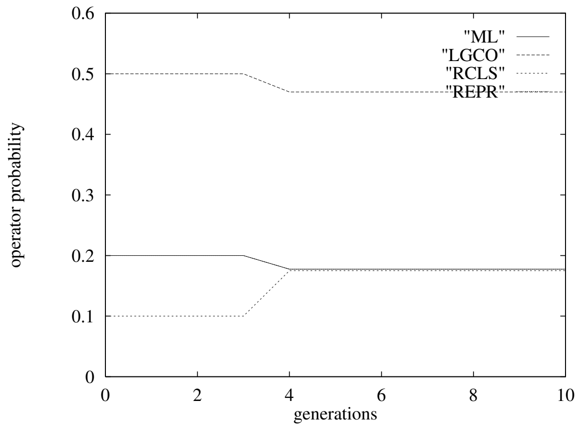
(c) Progress over a single GA14 run



(d) Operator-probability adaptation for GA14



(e) Progress over a single GA15 run



(f) Operator-probability adaptation for GA15

Figure F.9: Typical topology results for GA13, GA14 and GA15 (Problem 1, Run 1)

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.952	4.748	4.416	4.590	4.442
2	4.936	4.757	4.439	4.592	4.427
3	4.955	4.739	4.409	4.600	4.457
4	4.944	4.757	4.408	4.594	4.419
5	4.941	4.755	4.400	4.590	4.426
6	4.954	4.765	4.408	4.595	4.431
7	4.942	4.748	4.416	4.596	4.421
8	4.937	4.756	4.418	4.607	4.440
9	4.970	4.743	4.406	4.590	4.432
10	4.969	4.761	4.413	4.595	4.428
Best	4.936	4.739	4.400	4.590	4.419
Median	4.948	4.756	4.411	4.594	4.430

Table F.34: Topology cost results for GA14

Run	Problem				
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$	4 ($\times 10^6$)	5 ($\times 10^6$)
1	4.939	4.744	4.400	4.588	4.419
2	4.935	4.736	4.404	4.587	4.418
3	4.944	4.739	4.404	4.587	4.418
4	4.936	4.744	4.404	4.588	4.418
5	4.936	4.744	4.404	4.587	4.420
6	4.942	4.744	4.404	4.588	4.417
7	4.937	4.743	4.400	4.589	4.421
8	4.938	4.741	4.400	4.587	4.418
9	4.935	4.741	4.400	4.585	4.416
10	4.941	4.741	4.400	4.587	4.419
Best	4.935	4.736	4.400	4.585	4.416
Median	4.937	4.742	4.402	4.587	4.418

Table F.35: Topology cost results for GA15

F.4 Routing, Fibre & Wavelength Allocation

F.4.1 Minimum Network Wavelength Requirement

The non-default parameters values for the two GAs discussed in §5.2.1 on p. 109 are given in Tables F.36 and F.37. The complete network wavelength requirement (NWR) results from applying them to Networks 1–5 are then recorded in Tables F.38 and F.39. The best of each of these groups of five runs was included in Table 5.15 on p. 112.

The non-default parameters values for the three heuristics discussed in §5.2.1 are given in Table F.40. The NWR results from applying one of these, Wuttisittikulkij & O'Mahony's heuristic ($k = 8$, sorted order) [298] (W&O), to Networks 1–5 are listed in Table F.41. The best of each of these groups of five runs was again included in Table 5.15.

Parameter	Value	Description
`net.metric`	`NWR`	Metric used for network self-assessment
`net.router`	`WNOP`	Router used to route the network prior to self-assessment
`maxTrials`	`500,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`500`	Population size
`pop.select`	`tournament`	Selection operator
`pop.initOp`	`WRKS`	Population initialisation operator
`pop.phyNetwork`	`true`	Construct the individual using a `PhyNetwork` (`true`) or a `Network` (`false`)
`wrks.theK`	`8`	Number of alternative paths, k
`wrks.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkm.theK`	`8`	Number of alternative paths, k
`wkm.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkco.theK`	`8`	Number of alternative paths, k
`wkco.firstFibreOnly`	`false`	Use only first fibre on each cable?

Table F.36: Non-default parameters for GA16

Parameter	Value	Description
`net.metric`	`NWR`	Metric used for network self-assessment
`net.router`	`WNOP`	Router used to route the network prior to self-assessment
`maxTrials`	`75,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`500`	Population size
`pop.select`	`tournament`	Selection operator
`pop.initOp`	`WRKS`	Population initialisation operator
`pop.phyNetwork`	`true`	Construct the individual using a `PhyNetwork` (`true`) or a `Network` (`false`)
`wrks.theK`	`8`	Number of alternative paths, k
`wrks.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkm.theK`	`8`	Number of alternative paths, k
`wkm.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wwrr.theK`	`8`	Number of alternative paths, k
`wwrr.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wwso.theK`	`8`	Number of alternative paths, k
`wwso.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkco.theK`	`8`	Number of alternative paths, k
`wkco.firstFibreOnly`	`false`	Use only first fibre on each cable?

Table F.37: Non-default parameters for GA17

Run	Network					
	1		2		3	
	NWR	Trials	NWR	Trials	NWR	Trials
1	14	400,181	11	299,331	10	448,235
2	13	424,437	10	469,338	11	484,945
3	13	351,264	10	401,919	10	467,209
4	14	245,101	10	496,945	11	345,812
5	14	330,278	12	345,997	11	333,983
Best	13	351,264	10	401,919	10	448,235

Run	Network					
	4		5			
	NWR	Trials	NWR	Trials		
1	12	321,042	13	469,673		
2	12	445,261	12	436,962		
3	11	429,485	12	428,962		
4	12	480,841	13	339,015		
5	12	365,958	13	314,122		
Best	11	429,485	12	428,962		

Table F.38: Minimum NWR results for GA16

Run	Network					
	1		2		3	
	NWR	Trials	NWR	Trials	NWR	Trials
1	9	31,874	7	46,952	8	29,420
2	9	35,150	8	27,508	7	44,467
3	9	39,550	7	38,291	8	36,178
4	9	33,726	7	42,283	7	53,688
5	9	34,351	7	39,823	7	62,726
Best	9	31,874	7	38,291	7	44,467
Run	Network					
	4		5			
	NWR	Trials	NWR	Trials		
1	8	48,304	9	34,523		
2	9	29,391	9	36,118		
3	8	37,997	9	33,571		
4	8	40,284	8	48,448		
5	8	38,939	9	34,393		
Best	8	37,997	8	48,448		

Table F.39: Minimum NWR results for GA17

Parameter	Value	Description
`ww95.theK`	8	Number of alternative paths, k
`ww95.sortPaths`	`true`	Allocate the paths in hop count order?
`ww95.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wadi.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wahr.theK`	8	Number of alternative paths, k
`wahr.firstFibreOnly`	`false`	Use only first fibre on each cable?

Table F.40: Non-default parameters for W&0, W&D1 and W&D2 heuristics (minimum NWR)

Run	Network				
	1	2	3	4	5
1	8	7	8	8	9
2	8	7	8	8	9
3	8	8	8	8	9
4	8	8	8	8	9
5	8	7	8	8	9
Best	8	7	8	8	9

Table F.41: Minimum NWR results for W&O

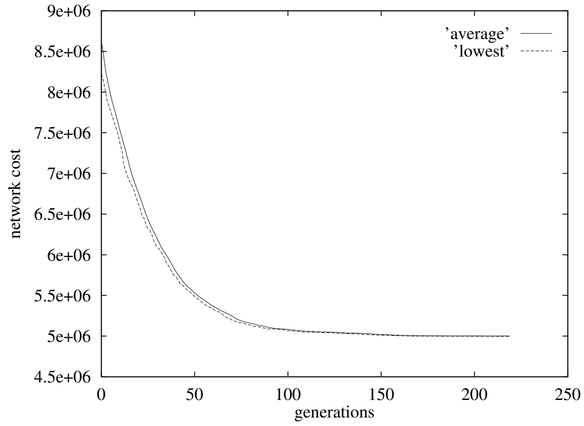
F.4.2 Minimum Cost Allocation

The non-default parameters values common to the three GAs discussed in §5.2.2 on p. 113 are given in Table F.42. Typical results, showing progress and operator-probability adaptation over single runs of GA19 and GA20 on Network 1, for different combinations of (α, β, γ), are illustrated in Figures F.10–F.13. Equivalent traces for GA18 were given in Figures 5.8 and 5.9 on p. 114. The complete results from applying all three GAs to Networks 1–7 are recorded in Tables F.43–F.48. The best of each of these groups of five runs was included in Tables 5.17–5.22 on pp. 115–116.

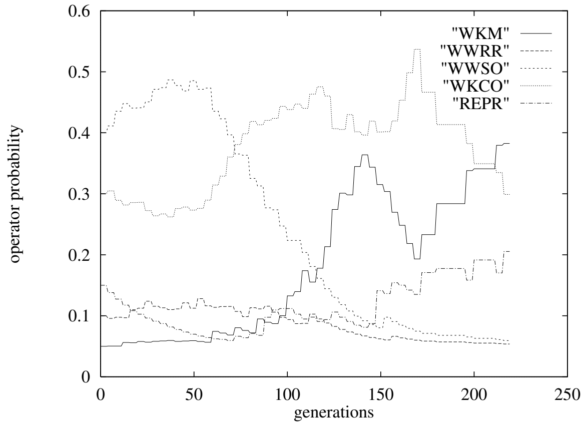
The non-default parameters values for the three heuristics discussed in §5.2.2 on p. 113 are given in Table F.49. The allocation results from applying one of these, Wuttisittikulkij & O'Mahony's heuristic [298], to Networks 1–7, for six different combinations of the Network Cost Model B parameters (α, β, γ), are listed in Tables F.50–F.55. The best of each of these groups of five runs was again included in Tables 5.17–5.22.

Parameter	Value	Description
`net.metric`	`PNCT`	Metric used for network self-assessment
`net.router`	`WNOP`	Router used to route the network prior to self-assessment
`maxTrials`	`100,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`500`	Population size
`pop.select`	`tournament`	Selection operator
`pop.initOp`	`WRKS`	Population initialisation operator
`pop.phyNetwork`	`true`	Construct the individual using a `PhyNetwork` (`true`) or a `Network` (`false`)
`wrks.theK`	8 or 4	Number of alternative paths, k
`wrks.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkm.theK`	8 or 4	Number of alternative paths, k
`wkm.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wwrr.theK`	8 or 4	Number of alternative paths, k
`wwrr.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wwso.theK`	8 or 4	Number of alternative paths, k
`wwso.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkco.theK`	8 or 4	Number of alternative paths, k
`wkco.firstFibreOnly`	`false`	Use only first fibre on each cable?

Table F.42: Non-default parameters for GA18–GA20

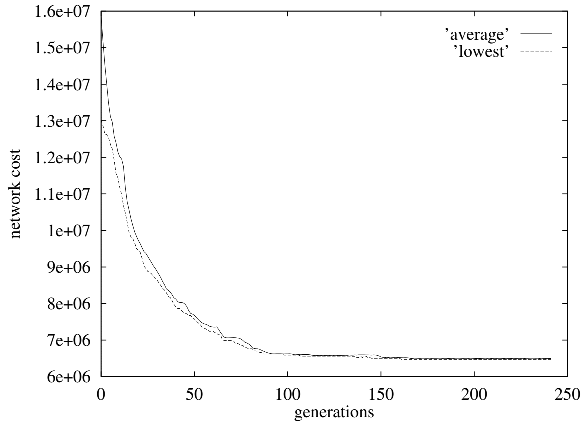


(a) Progress over a single run

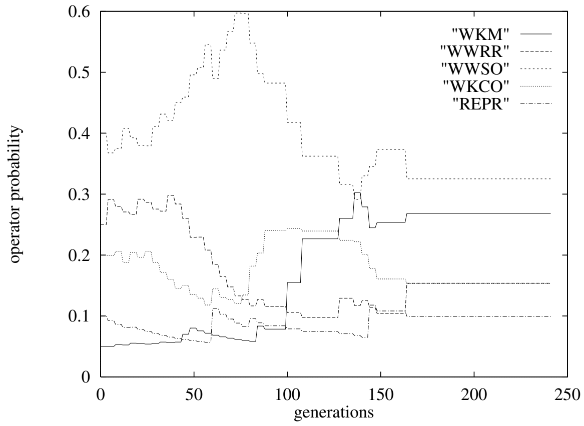


(b) Operator-probability adaptation

Figure F.10: Typical results for GA19, $\alpha = \beta = 1$ and $\gamma = 1$ (Network 1, Run 1)

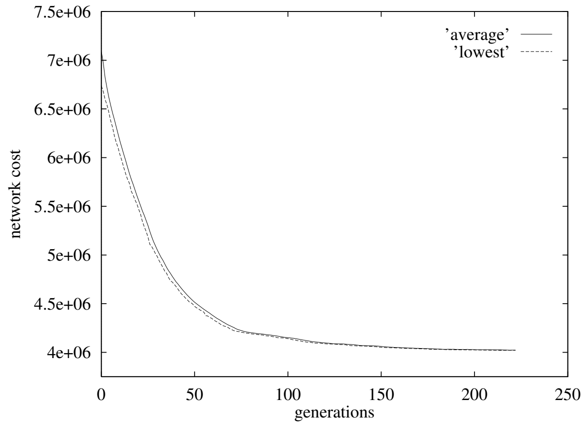


(a) Progress over a single run

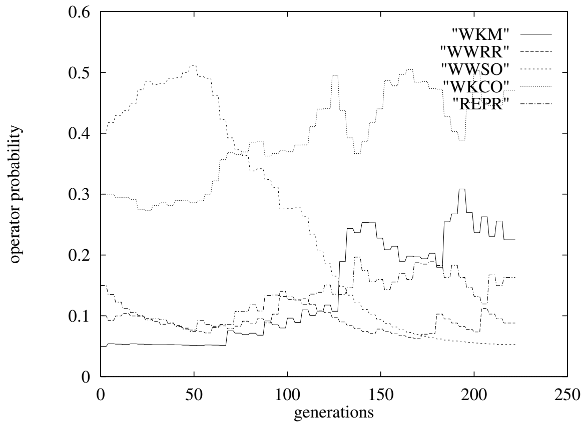


(b) Operator-probability adaptation

Figure F.11: Typical results for GA20, $\alpha = \beta = 1$ and $\gamma = 0$ (Network 1, Run 1)

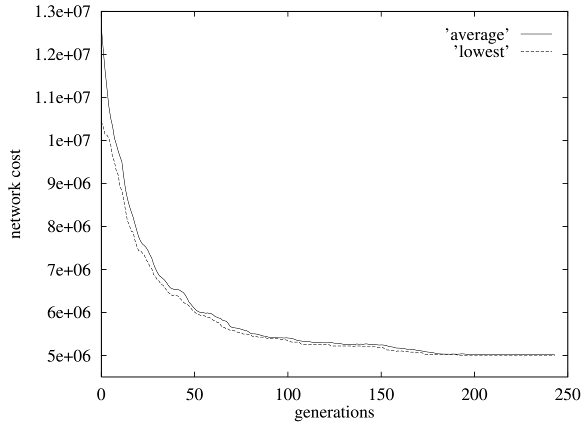


(a) Progress over a single run

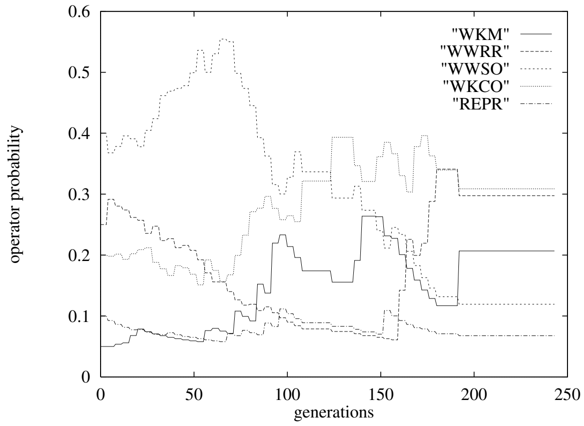


(b) Operator-probability adaptation

Figure F.12: Typical results for GA19, $\alpha = \beta = 0.8$ and $\gamma = 1$ (Network 1, Run 1)



(a) Progress over a single run



(b) Operator-probability adaptation

Figure F.13: Typical results for GA20, $\alpha = \beta = 0.8$ and $\gamma = 0$ (Network 1, Run 1)

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	5.579	4.930	5.013	5.117	5.182	5.370	5.751
2	5.694	4.969	5.016	5.201	5.202	5.434	5.656
3	5.732	4.905	4.979	5.188	5.206	5.361	5.757
4	5.679	4.938	4.979	5.189	5.150	5.344	5.681
5	5.629	4.886	4.976	5.248	5.108	5.435	5.651
Best	5.579	4.886	4.976	5.117	5.108	5.344	5.651

Table F.43: Minimum cost allocation results for $\alpha = \beta = 1$ and $\gamma = 0.5$ using GA18

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	4.996	4.436	4.395	4.645	4.641	4.753	5.228
2	4.996	4.437	4.396	4.645	4.640	4.753	5.228
3	4.996	4.436	4.394	4.645	4.640	4.753	5.228
4	4.995	4.438	4.394	4.645	4.640	4.753	5.228
5	4.994	4.438	4.394	4.646	4.641	4.753	5.228
Best	4.994	4.436	4.394	4.645	4.640	4.753	5.228

Table F.44: Minimum cost allocation results for $\alpha = \beta = 1$ and $\gamma = 1$ using GA19

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	6.471	5.338	5.435	5.861	5.561	5.958	6.007
2	6.188	5.417	5.362	5.499	5.684	5.787	6.001
3	6.310	5.454	5.598	5.910	5.858	6.108	6.029
4	6.116	5.428	5.384	5.672	5.523	5.781	6.005
5	6.217	5.228	5.422	5.749	5.714	5.954	6.294
Best	6.116	5.228	5.362	5.499	5.523	5.781	6.001

Table F.45: Minimum cost allocation results for $\alpha = \beta = 1$ and $\gamma = 0$ using GA20

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	4.616	4.011	4.017	4.136	4.123	4.098	4.001
2	4.450	3.941	3.866	4.093	4.117	4.065	3.982
3	4.627	3.909	4.016	4.062	4.040	4.169	3.965
4	4.532	3.909	3.928	4.144	4.151	4.185	4.034
5	4.650	3.933	3.918	4.103	4.076	4.110	4.000
Best	4.450	3.909	3.866	4.062	4.040	4.065	3.965

Table F.46: Minimum cost allocation results for $\alpha = \beta = 0.8$ and $\gamma = 0.5$ using GA18

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	4.020	3.531	3.541	3.718	3.717	3.633	3.686
2	4.014	3.531	3.541	3.717	3.716	3.633	3.686
3	4.015	3.531	3.545	3.717	3.717	3.633	3.686
4	4.016	3.533	3.541	3.732	3.717	3.633	3.686
5	4.018	3.532	3.541	3.716	3.717	3.633	3.686
Best	4.014	3.531	3.541	3.716	3.716	3.633	3.686

Table F.47: Minimum cost allocation results for $\alpha = \beta = 0.8$ and $\gamma = 1$ using GA19

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	5.005	4.518	4.597	4.444	4.625	4.585	4.300
2	4.910	4.335	4.281	4.546	4.581	4.346	4.381
3	5.147	4.292	4.271	4.463	4.363	4.491	4.228
4	4.895	4.249	4.374	4.473	4.474	4.344	4.237
5	4.946	4.265	4.463	4.556	4.524	4.624	4.230
Best	4.895	4.249	4.271	4.444	4.363	4.344	4.228

Table F.48: Minimum cost allocation results for $\alpha = \beta = 0.8$ and $\gamma = 0$ using GA20

Parameter	Value	Description
`ww95.theK`	8 or 4	Number of alternative paths, k
`ww95.sortPaths`	`true`	Allocate the paths in hop count order?
`ww95.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wadi.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wahr.theK`	8 or 4	Number of alternative paths, k
`wahr.firstFibreOnly`	`false`	Use only first fibre on each cable?

Table F.49: Non-default parameters for W&0, W&D1 and W&D2 heuristics (minimum cost allocation)

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	5.919	5.151	5.185	5.445	5.524	5.732	6.093
2	5.980	5.231	5.172	5.434	5.696	5.692	6.077
3	6.032	5.161	5.213	5.413	5.441	5.811	6.164
4	5.917	5.171	5.153	5.416	5.424	5.662	6.101
5	5.909	5.169	5.154	5.477	5.526	5.761	6.232
Best	5.909	5.151	5.153	5.413	5.424	5.662	6.077

Table F.50: Minimum cost allocation results for $\alpha = \beta = 1$ and $\gamma = 0.5$ using W&O

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	5.790	5.081	5.053	5.367	5.386	5.540	5.966
2	5.845	5.159	5.067	5.320	5.543	5.516	5.907
3	5.897	5.109	5.089	5.313	5.338	5.569	5.960
4	5.704	5.104	5.051	5.330	5.302	5.533	5.968
5	5.782	5.088	5.078	5.365	5.411	5.615	6.025
Best	5.704	5.081	5.051	5.313	5.302	5.516	5.907

Table F.51: Minimum cost allocation results for $\alpha = \beta = 1$ and $\gamma = 1$ using W&O

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	6.048	5.222	5.317	5.523	5.662	5.925	6.220
2	6.115	5.303	5.278	5.548	5.850	5.868	6.247
3	6.167	5.213	5.337	5.513	5.544	6.053	6.368
4	6.131	5.237	5.254	5.503	5.547	5.790	6.234
5	6.036	5.251	5.229	5.589	5.641	5.907	6.439
Best	6.036	5.213	5.229	5.503	5.544	5.790	6.220

Table F.52: Minimum cost allocation results for $\alpha = \beta = 1$ and $\gamma = 0$ using W&O

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	4.667	4.090	4.162	4.328	4.392	4.295	4.215
2	4.715	4.146	4.159	4.317	4.527	4.260	4.203
3	4.755	4.089	4.186	4.312	4.329	4.354	4.263
4	4.662	4.107	4.138	4.312	4.313	4.246	4.217
5	4.653	4.105	4.140	4.350	4.394	4.315	4.306
Best	4.653	4.089	4.138	4.312	4.313	4.246	4.203

Table F.53: Minimum cost allocation results for $\alpha = \beta = 0.8$ and $\gamma = 0.5$ using W&O

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	4.565	4.036	4.058	4.272	4.283	4.148	4.124
2	4.605	4.088	4.072	4.230	4.402	4.123	4.082
3	4.651	4.050	4.090	4.230	4.248	4.170	4.115
4	4.489	4.056	4.056	4.242	4.216	4.149	4.123
5	4.552	4.041	4.081	4.260	4.302	4.203	4.157
Best	4.489	4.036	4.056	4.230	4.216	4.123	4.082

Table F.54: Minimum cost allocation results for $\alpha = \beta = 0.8$ and $\gamma = 1$ using W&O

Run	Network						
	1 ($\times 10^6$)	2 ($\times 10^6$)	3 ($\times 10^6$)	4 ($\times 10^6$)	5 ($\times 10^6$)	6 ($\times 10^6$)	7 ($\times 10^6$)
1	4.768	4.144	4.266	4.384	4.501	4.441	4.306
2	4.824	4.204	4.245	4.404	4.651	4.398	4.323
3	4.859	4.128	4.283	4.395	4.410	4.538	4.412
4	4.836	4.158	4.220	4.382	4.409	4.343	4.311
5	4.755	4.169	4.200	4.439	4.487	4.427	4.455
Best	4.755	4.128	4.200	4.382	4.409	4.343	4.306

Table F.55: Minimum cost allocation results for $\alpha = \beta = 0.8$ and $\gamma = 0$ using W&O

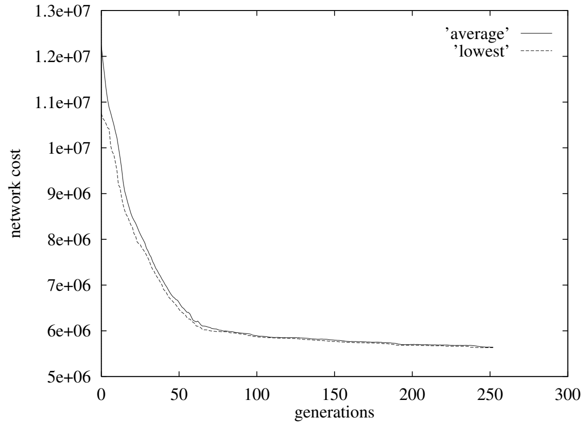
F.5 Operator-probability Adaptation

F.5.1 Basic Adaptation Algorithm

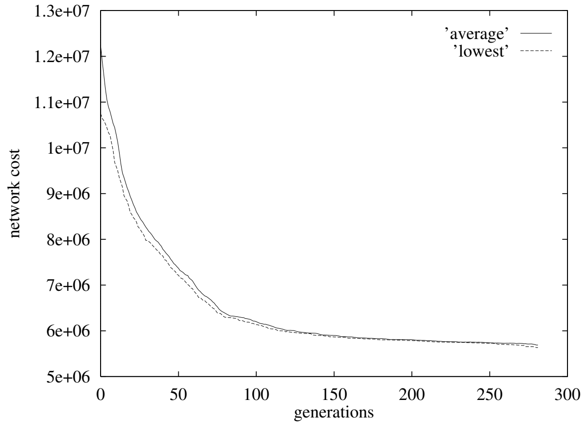
The non-default parameters values common to the five GAs discussed in §5.3.1 on p. 118 are given in Table F.56. Typical results, showing progress over single runs of GA21–GA25, each with a different combination of adaptation parameters (ψ, κ), on Network 1 are illustrated in Fig. F.14. The corresponding operator-probability adaptation curves for GA21, GA23–GA25 are given in Fig. 5.10 on p. 120. Moreover, the complete results from applying all five GAs to Networks 1, 3 & 5 are recorded in Tables F.57–F.61 (these were summarised in Tables 5.24 and 5.25 on p. 121).

Parameter	Value	Description
`net.metric`	`PNCT`	Metric used for network self-assessment
`net.router`	`WNOP`	Router used to route the network prior to self-assessment
`maxTrials`	`100,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`500`	Population size
`pop.select`	`tournament`	Selection operator
`pop.initOp`	`WRKS`	Population initialisation operator
`pop.phyNetwork`	`true`	Construct the individual using a `PhyNetwork` (`true`) or a `Network` (`false`)
`opp.adaptProp`	`0.00`, `0.15` or `0.30`	Adaptation proportion, ψ; $0 \leq \psi \leq 1$
`opp.creditK`	`0.0`, `0.5` or `1.0`	Credit constant, κ; must be ≥ 0
`wrks.theK`	`8`	Number of alternative paths, k
`wrks.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkm.theK`	`8`	Number of alternative paths k
`wkm.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wwrr.theK`	`8`	Number of alternative paths, k
`wwrr.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wwso.theK`	`8`	Number of alternative paths, k
`wwso.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkco.theK`	`8`	Number of alternative paths, k
`wkco.firstFibreOnly`	`false`	Use only first fibre on each cable?

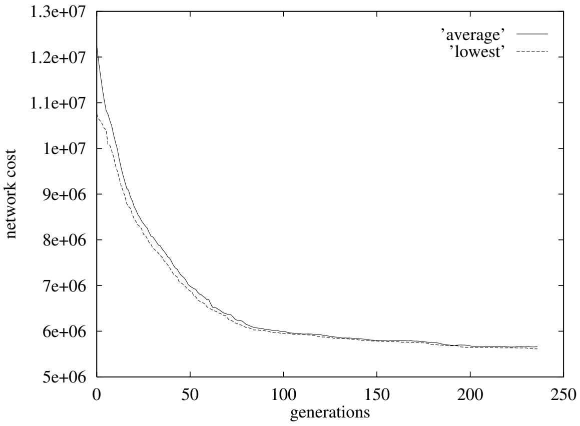
Table F.56: Non-default parameters for GA21–GA25



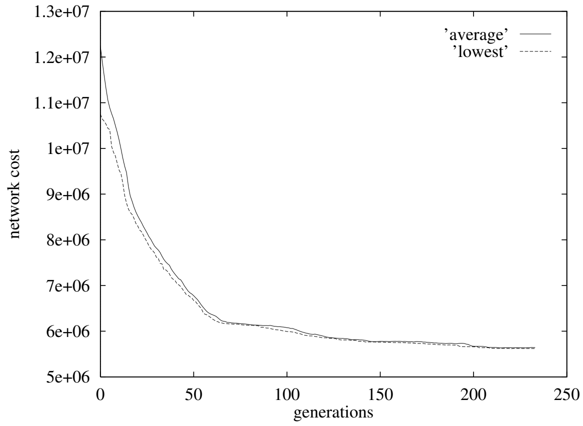
(a) GA21, $(\psi, \kappa) = (0.15, 0.5)$



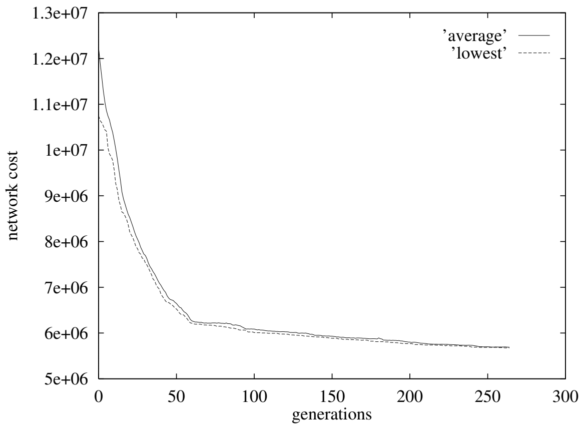
(b) GA22, $(\psi, \kappa) = (0.00, 0.5)$



(c) GA23, $(\psi, \kappa) = (0.30, 0.5)$



(d) GA24, $(\psi, \kappa) = (0.15, 0.0)$



(e) GA25, $(\psi, \kappa) = (0.15, 1.0)$

Figure F.14: Typical progress over single runs of GA21–GA25 (Network 1, Run 1)

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.222	5.468	5.954	5.632	4.960	5.126
2	6.582	5.524	6.075	5.614	4.889	5.111
3	6.506	5.838	5.869	5.645	4.911	5.256
4	6.528	5.576	5.913	5.780	4.991	5.091
5	6.442	5.635	6.272	5.717	4.967	5.081
6	6.454	5.618	5.896	5.582	4.879	5.161
7	6.379	5.511	5.979	5.699	4.950	5.199
8	6.663	5.591	6.034	5.680	4.940	5.233
9	6.595	5.684	5.922	5.673	5.028	5.249
10	6.682	5.649	6.074	5.597	4.928	5.101
Best	6.222	5.468	5.869	5.582	4.879	5.081
Median	6.517	5.604	5.967	5.659	4.945	5.144

Table F.57: Operator-probability adaptation results for GA21

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.783	5.654	6.147	5.634	4.871	5.225
2	6.780	5.700	6.293	5.736	4.934	5.063
3	6.682	5.850	5.922	5.743	4.861	5.179
4	6.709	5.747	5.953	5.703	4.885	5.127
5	6.543	5.730	6.318	5.718	4.976	5.115
6	6.630	5.778	6.085	5.621	4.924	5.144
7	6.551	5.788	6.030	5.680	4.871	5.238
8	6.754	5.642	6.085	5.682	4.965	5.188
9	6.678	5.692	6.055	5.671	4.910	5.164
10	6.527	5.713	6.042	5.636	4.937	5.194
Best	6.527	5.642	5.922	5.621	4.861	5.063
Median	6.680	5.722	6.070	5.681	4.917	5.171

Table F.58: Operator-probability adaptation results for GA22

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.628	5.535	6.115	5.614	4.978	5.276
2	6.306	5.625	6.172	5.609	4.913	5.183
3	6.727	5.763	5.784	5.630	4.908	5.090
4	6.534	5.800	5.972	5.650	5.016	5.108
5	6.484	5.818	6.207	5.717	4.853	5.165
6	6.703	5.820	6.002	5.667	4.981	5.142
7	6.442	5.691	6.074	5.701	4.936	5.175
8	6.700	5.586	6.073	5.735	4.971	5.135
9	6.581	5.824	5.845	5.707	4.867	5.138
10	6.442	5.714	5.984	5.647	4.833	5.090
Best	6.306	5.535	5.784	5.609	4.833	5.090
Median	6.558	5.739	6.038	5.658	4.924	5.140

Table F.59: Operator-probability adaptation results for GA23

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.331	5.461	6.029	5.613	4.934	5.209
2	6.529	5.566	6.205	5.811	4.932	5.207
3	6.635	5.743	5.875	5.695	4.936	5.242
4	6.598	5.662	6.157	5.681	4.985	5.140
5	6.379	5.607	6.168	5.694	4.926	5.191
6	6.621	5.725	5.929	5.624	4.905	5.141
7	6.463	5.674	6.023	5.655	4.887	5.138
8	6.511	5.460	5.927	5.648	4.933	5.175
9	6.611	5.583	5.884	5.637	4.921	5.109
10	6.382	5.613	6.109	5.662	4.937	5.079
Best	6.331	5.460	5.875	5.613	4.887	5.079
Median	6.520	5.610	6.026	5.659	4.932	5.158

Table F.60: Operator-probability adaptation results for GA24

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.222	5.546	5.944	5.675	4.976	5.113
2	6.364	5.678	6.254	5.641	5.019	5.150
3	6.679	5.748	5.819	5.635	4.887	5.113
4	6.680	5.668	6.144	5.646	4.974	5.150
5	6.359	5.627	6.001	5.548	4.875	5.140
6	6.460	5.605	5.995	5.445	4.890	5.118
7	6.413	5.636	5.950	5.635	4.880	5.153
8	6.620	5.578	6.009	5.656	4.961	5.079
9	6.492	5.714	5.904	5.614	4.895	5.215
10	6.562	5.545	6.184	5.608	5.001	5.229
Best	6.222	5.545	5.819	5.445	4.875	5.079
Median	6.476	5.631	5.998	5.635	4.928	5.145

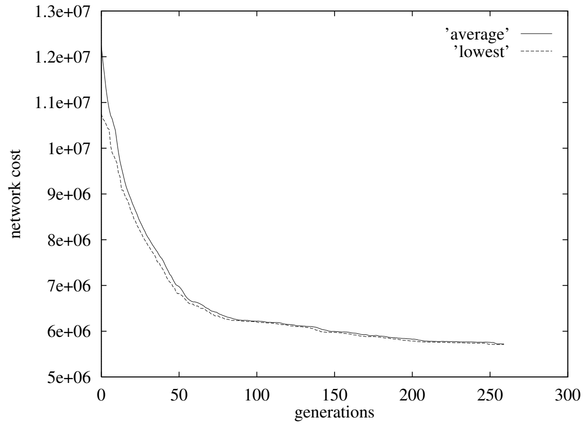
Table F.61: Operator-probability adaptation results for GA25

F.5.2 Variants on the Basic Algorithm

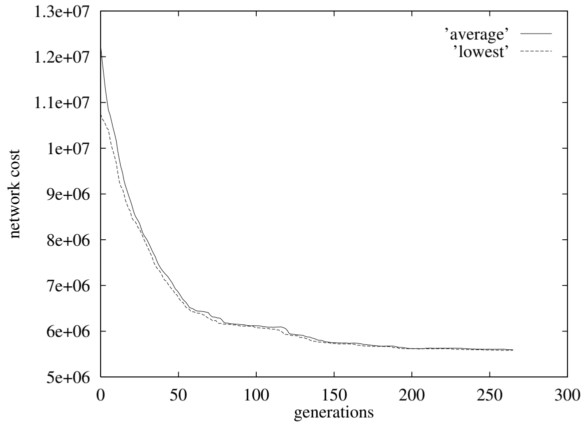
The non-default parameters values common to the eight GAs discussed in §5.3.2 on p. 122 are given in Table F.62. Typical results, showing progress over single runs of GA26-GA29, each with a different combination of adaptation parameters (ψ, κ) and using parental credit assignment, on Network 1 are illustrated in Fig. F.15. The corresponding operator-probability adaptation curves are given in Fig. 5.11 on p. 124. Similarly, for median credit assignment, typical results showing progress over single runs of GA30-GA33 on Network 1 are presented in Fig. F.16. The corresponding operator-probability adaptation curves to these are shown in Fig. 5.12 on p. 127. In addition, the complete results from applying all eight GAs to Networks 1, 3 & 5 are recorded in Tables F.63–F.70 (these were summarised in Tables 5.27–5.30 on pp. 125–127).

Parameter	Value	Description
`net.metric`	`PNCT`	Metric used for network self-assessment
`net.router`	`WNOP`	Router used to route the network prior to self-assessment
`maxTrials`	`100,000`	Maximum number of trials; must be ≥ 1
`pop.size`	`500`	Population size
`pop.select`	`tournament`	Selection operator
`pop.initOp`	`WRKS`	Population initialisation operator
`pop.phyNetwork`	`true`	Construct the individual using a `PhyNetwork` (`true`) or a `Network` (`false`)
`opp.adaptProp`	`0.15` or `0.30`	Adaptation proportion, ψ; $0 \leq \psi \leq 1$
`opp.creditK`	`0.0`, `0.5` or `1.0`	Credit constant, κ; must be ≥ 0
`opp.fitCredType`	`parent` or `median`	Credit type: `best`, `winBest`, `median` or `parent`
`wrks.theK`	`8`	Number of alternative paths, k
`wrks.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkm.theK`	`8`	Number of alternative paths, k
`wkm.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wwrr.theK`	`8`	Number of alternative paths, k
`wwrr.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wwso.theK`	`8`	Number of alternative paths, k
`wwso.firstFibreOnly`	`false`	Use only first fibre on each cable?
`wkco.theK`	`8`	Number of alternative paths, k
`wkco.firstFibreOnly`	`false`	Use only first fibre on each cable?

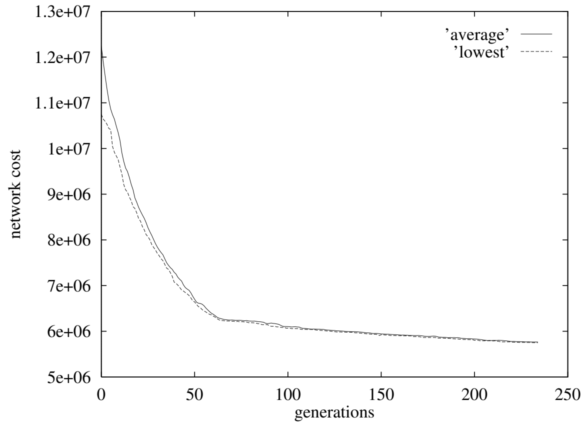
Table F.62: Non-default parameters for GA26–GA33



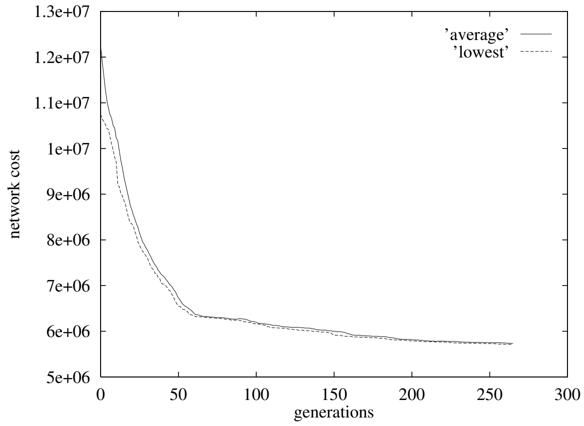
(a) GA26, $(\psi, \kappa) = (0.15, 0.5)$



(b) GA27, $(\psi, \kappa) = (0.30, 0.5)$



(c) GA28, $(\psi, \kappa) = (0.15, 0.0)$

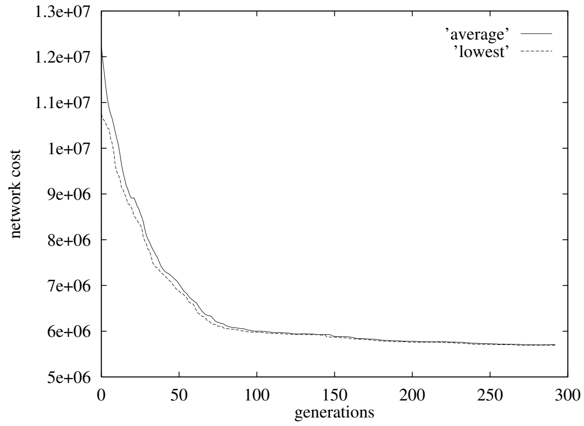


(d) GA29, $(\psi, \kappa) = (0.15, 1.0)$

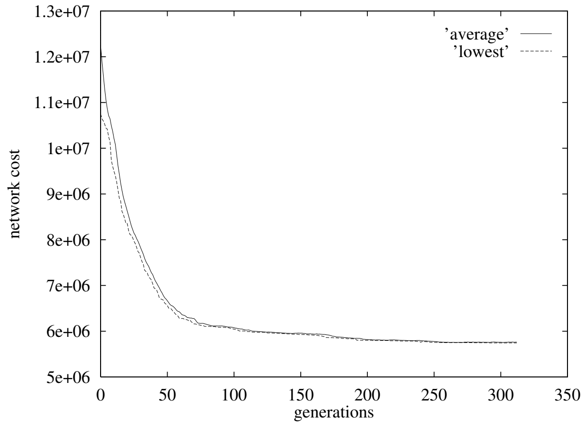
Figure F.15: Typical progress over single runs of GA26–GA29 (Network 1, Run 1)

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.557	5.523	5.990	5.710	4.968	5.112
2	6.461	5.639	6.035	5.642	4.875	5.124
3	6.579	5.838	5.806	5.731	4.958	5.137
4	6.684	5.613	5.982	5.584	4.904	5.095
5	6.371	5.605	5.999	5.636	4.892	5.164
6	6.652	5.651	6.073	5.678	4.878	5.189
7	6.526	5.571	6.024	5.638	4.981	5.165
8	6.439	5.527	5.823	5.704	4.874	5.198
9	6.510	5.599	5.707	5.771	4.933	5.173
10	6.545	5.553	5.895	5.655	4.911	5.137
Best	6.371	5.523	5.707	5.584	4.874	5.095
Median	6.536	5.602	5.986	5.666	4.907	5.150

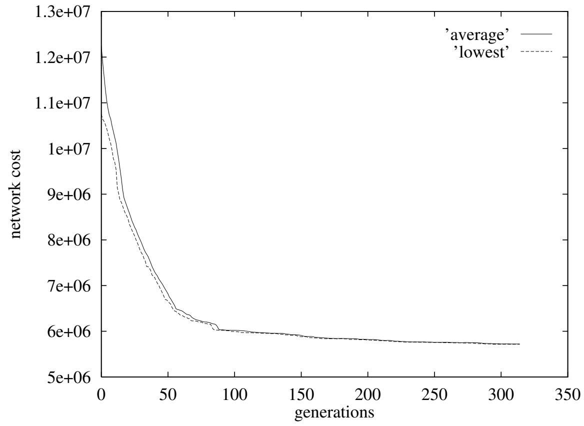
Table F.63: Operator-probability adaptation results for GA26



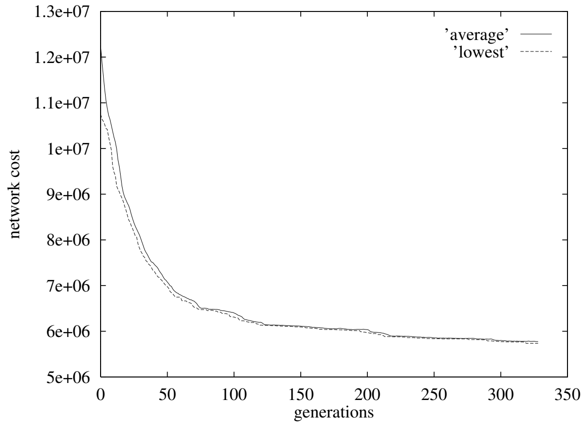
(a) GA30, $(\psi, \kappa) = (0.15, 0.5)$



(b) GA31, $(\psi, \kappa) = (0.30, 0.5)$



(c) GA32, $(\psi, \kappa) = (0.15, 0.0)$



(d) GA33, $(\psi, \kappa) = (0.15, 1.0)$

Figure F.16: Typical progress over single runs of GA30–GA33 (Network 1, Run 1)

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.427	5.470	5.932	5.583	5.051	5.183
2	6.521	5.622	5.944	5.697	5.006	5.211
3	6.509	5.801	5.751	5.668	4.941	5.143
4	6.499	5.545	5.841	5.647	4.894	5.204
5	6.434	5.686	6.109	5.642	5.000	5.167
6	6.402	5.655	6.065	5.602	4.915	5.166
7	6.450	5.587	6.107	5.644	4.923	5.161
8	6.550	5.515	5.952	5.668	4.869	5.138
9	6.499	5.613	5.955	5.786	4.840	5.155
10	6.420	5.615	5.862	5.576	4.950	5.180
Best	6.402	5.470	5.751	5.576	4.840	5.138
Median	6.474	5.614	5.948	5.646	4.932	5.166

Table F.64: Operator-probability adaptation results for GA27

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.332	5.541	5.969	5.750	4.941	5.180
2	6.468	5.586	6.069	5.628	4.941	5.095
3	6.564	5.887	5.760	5.815	4.961	5.139
4	6.519	5.798	5.964	5.803	4.997	5.118
5	6.378	5.601	6.234	5.563	4.981	5.171
6	6.462	5.738	6.024	5.616	4.912	5.095
7	6.370	5.665	5.909	5.589	4.919	5.091
8	6.529	5.551	5.949	5.608	4.958	5.248
9	6.542	5.640	6.004	5.698	4.884	5.171
10	6.515	5.536	5.969	5.608	4.896	5.129
Best	6.332	5.536	5.760	5.563	4.884	5.091
Median	6.491	5.621	5.969	5.622	4.941	5.134

Table F.65: Operator-probability adaptation results for GA28

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.312	5.589	6.017	5.713	4.930	5.229
2	6.506	5.539	6.154	5.626	5.001	5.213
3	6.667	5.883	5.765	5.592	4.939	5.206
4	6.657	5.666	6.049	5.568	4.971	5.159
5	6.219	5.490	5.967	5.637	4.965	5.168
6	6.473	5.727	6.010	5.676	4.937	5.148
7	6.409	5.645	5.962	5.662	4.983	5.123
8	6.653	5.563	6.050	5.603	4.893	5.100
9	6.452	5.565	5.884	5.744	5.035	5.042
10	6.502	5.530	6.039	5.630	4.994	5.266
Best	6.219	5.490	5.765	5.568	4.893	5.042
Median	6.487	5.577	6.013	5.634	4.968	5.164

Table F.66: Operator-probability adaptation results for GA29

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.400	5.520	5.943	5.693	4.956	5.255
2	6.396	5.500	6.092	5.760	4.976	5.319
3	6.420	5.774	5.900	5.719	4.919	5.169
4	6.576	5.690	5.970	5.834	4.966	5.134
5	6.463	5.658	5.949	5.721	4.973	5.113
6	6.491	5.628	5.969	5.733	4.946	5.253
7	6.401	5.580	6.103	5.668	4.987	5.126
8	6.567	5.589	5.877	5.678	5.136	5.153
9	6.508	5.395	5.896	5.694	5.002	5.147
10	6.396	5.590	5.871	5.595	0.000	5.199
Best	6.396	5.395	5.871	5.595	4.919	5.113
Median	6.442	5.590	5.946	5.706	4.971	5.161

Table F.67: Operator-probability adaptation results for GA30

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.249	5.420	5.916	5.745	4.935	5.153
2	6.381	5.441	6.108	5.720	5.035	5.254
3	6.626	5.643	5.733	5.700	4.959	5.135
4	6.517	5.743	5.895	5.705	4.899	5.264
5	6.437	5.545	5.924	5.780	4.982	5.168
6	6.427	5.659	5.913	5.678	4.900	5.346
7	6.530	5.594	5.833	5.657	4.906	5.190
8	6.423	5.666	6.138	5.818	4.982	5.194
9	6.409	5.524	5.788	5.778	4.890	5.115
10	6.390	5.523	5.892	5.637	4.914	5.107
Best	6.249	5.420	5.733	5.637	4.890	5.107
Median	6.425	5.569	5.904	5.712	4.925	5.179

Table F.68: Operator-probability adaptation results for GA31

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.305	5.450	5.971	5.716	4.913	5.183
2	6.449	5.615	6.079	5.698	4.904	5.203
3	6.393	5.687	5.791	5.701	4.972	5.126
4	6.601	5.833	5.928	5.752	5.086	5.126
5	6.443	5.691	6.147	5.720	5.047	5.136
6	6.571	5.594	5.939	5.674	4.918	5.242
7	6.370	5.616	6.052	5.660	4.975	5.099
8	6.502	5.478	5.892	5.652	4.890	5.133
9	6.611	5.598	5.823	5.623	4.952	5.159
10	6.392	5.496	5.886	5.657	4.960	5.119
Best	6.305	5.450	5.791	5.623	4.890	5.099
Median	6.446	5.607	5.933	5.686	4.956	5.134

Table F.69: Operator-probability adaptation results for GA32

Run	Network					
	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)	1 ($\times 10^6$)	3 ($\times 10^6$)	5 ($\times 10^6$)
	25,000 trials			100,000 trials		
1	6.664	5.475	5.895	5.737	4.882	5.174
2	6.455	5.608	6.243	5.670	5.026	5.258
3	6.431	5.770	5.827	5.578	4.840	5.148
4	6.550	5.789	5.923	5.649	5.018	5.177
5	6.578	5.523	5.980	5.682	4.942	5.075
6	6.488	5.579	5.946	5.668	4.931	5.200
7	6.401	5.536	6.085	5.632	4.999	5.205
8	6.683	5.531	6.034	5.680	4.982	5.180
9	6.511	5.607	5.940	5.547	5.007	5.086
10	6.523	5.571	5.884	5.740	4.935	5.118
Best	6.401	5.475	5.827	5.547	4.840	5.075
Median	6.517	5.575	5.943	5.669	4.962	5.175

Table F.70: Operator-probability adaptation results for GA33