Genetic Algorithms for Uncapacitated Network Design

by

Viswanathan Lakshmi

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science
in Operations Research

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1995

© Massachusetts Institute of Technology 1995. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
January 26, 1995

Certified by ....
James B. Orlin
Professor
Thesis Supervisor

Accepted by ....
Robert M. Freund
Acting Co-Director, Operations Research Center

MASSACHUSETTS INSTITUTE

APR 13 1995 Eng.

Genetic Algorithms for Uncapacitated Network Design

by

Viswanathan Lakshmi

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 1995, in partial fulfillment of the requirements for the
degree of Master of Science in Operations Research

Abstract

In this thesis, we address the problem of obtaining a "near optimal" solution to the uncapacitated network design problem. In the uncapacitated network design problem, each arc of a network has a fixed design cost for construction and a variable routing cost for routing each unit of flow. For each pair i,j of nodes, there is a demand of one unit of flow from i to j, and flow may only be sent on constructed arcs.

We develop a genetic algorithm for solving the uncapacitated network design problem. It starts with a population of individuals and the population evolves by iterations. Each iteration of this genetic algorithm has steps called parental selection, crossover and immigration. We also define termination criteria for determining when the population has converged.

The implementation of our genetic algorithm is tested for different problem sizes and several instances in each problem size. The computational results are compared to that of the dual ascent method with drop-add heuristic for the same problem instances. Extensive graphical analysis is performed on the results. We observe that our genetic algorithm is an efficient solution technique to solve uncapacitated network design problems, but is dominated by the dual ascent method with drop-add heuristic.

Thesis Supervisor: James B. Orlin
Title: Professor

To my brother Krishnan

Acknowledgments

First and foremost, I would like to thank my advisor, Jim Orlin, for his invaluable guidance and support during my interactions with him. He has been of immense help to me accomodating to my needs, often, at very short notice.

I would like to express my gratitude to Izumi Sakuta for her patient coding assistances. She has been very nice to set aside her time and efforts to help me run the FORTRAN code.

Special thanks to Tom Magnanti and Georgia Perakis for their help and support during my tough times. I would like to thank all my friends at the OR Center. Special thanks to Raghu for his numerous help and advices (on matters from career counseling to software questions) always delivered with his characteristic smile. Thanks to Laura, Paulette and Cheryl for their continued assistance on administrative details.

Thanks to my past and present suite-mates in Ashdown House for the wonderful times we had together.

I can not express in words the gratitude I owe to my brother Krishnan. He has been a great source of support and confidence to me, without his help I would never have realized my interests and abilities. The love and confidence of my parents made me survive with increased vigour through my tough illnesses-the gratitude I owe them for all their care is beyond the point of expression. Many thanks to my brother Murali for his entertaining, motivating letters and for the fond reminiscences I have of growing up with him. Thanks to Mangai whose love and cheerfulness always helped in reducing my stress.

Millions of thanks to my husband Subash for his continued support during my most stressful times. When it comes to him, the phone lines have no limit on what they can carry-tonnes of love, encouragement, confidence and more!! I cannot thank more my parents (-in-law) for their bounties of love and caring letters. Many thanks to Natesh for his special cards, phone calls and numerous other expressions of love.

Contents

1 **Introduction and Literature Review** **10**

1.1 Introduction 10

1.2 Problem Definition 12

1.3 Literature Review 15

1.4 Overview of Thesis 16

2 **Uncapacitated Network Design** **18**

2.1 Alternate Formulations 18

2.2 Existing Solution Methods 20

2.2.1 Branch and bound method 20

2.2.2 Benders decomposition 22

2.2.3 Route selection algorithm 24

2.2.4 Dual-ascent method 25

2.3 Applications of network design problems 29

3 **Genetic Algorithms for Uncapacitated Network Design Problems** **31**

3.1 Basic terminology 32

3.2 A genetic algorithm for UNDP 32

3.3 Computational Implementation 37

3.3.1 Genetic algorithms implementation 39

3.3.2 Dual-ascent implementation 42

3.4 Computational Results 43
3.4.1 Average gap percentage 49
3.4.2 Computational time 55
3.4.3 Ratio of computational times (GA/DA) 67
3.4.4 Relation of Computational times to shortest path computations 69
3.5 Conclusions 75

4 Conclusions and suggestions for future research 77

Bibliography 79

List of Figures

3-1 percentage gaps for Uniform, Ratio 1 problems 48
3-2 percentage gaps for Uniform, Ratio 5 problems 50
3-3 percentage gaps for Uniform, Ratio 10 problems 50
3-4 percentage gaps for Uniform, Ratio 20 problems 51
3-5 percentage gaps for Uniform, Ratio 50 problems 51
3-6 GA percentage gaps for uniform node problem 52
3-7 DA percentage gaps for uniform node problem 53
3-8 percentage gaps for Normal, Ratio 1 problems 54
3-9 percentage gaps for Normal, Ratio 5 problems 54
3-10 percentage gaps for Normal, Ratio 10 problems 55
3-11 percentage gaps for Normal, Ratio 20 problems 56
3-12 percentage gaps for Normal, Ratio 50 problems 56
3-13 computational time for Uniform, Ratio 1 problems 57
3-14 computational time for Uniform, Ratio 5 problems 58
3-15 computational time for Uniform, Ratio 10 problems 59
3-16 computational time for Uniform, Ratio 20 problems 60
3-17 computational time for Uniform, Ratio 50 problems 61
3-18 computational time for Normal, Ratio 1 problems 62
3-19 computational time for Normal, Ratio 5 problems 63
3-20 computational time for Normal, Ratio 10 problems 64
3-21 computational time for Normal, Ratio 20 problems 65

3-22 computational time for Normal, Ratio 50 problems 66
3-23 Computational time ratio (GA/DA) for different cost ratio, uniform problems 67
3-24 Computational time ratio (GA/DA) for different cost ratio, normal problems 68
3-25 shortest path computations-computational time for problem type 1 . 69
3-26 shortest path computations-computational time for problem type 2 . 70
3-27 shortest path computations-computational time for problem type 3 . 71
3-28 shortest path computations-computational time for problem type 4 . 72
3-29 shortest path computations-computational time for problem type 5 . 72
3-30 shortest path computations-computational time for problem type 6 . 73
3-31 Cost ratio 1, ratio of standard deviation to average fitness-12 data points selected at random 74
3-32 Cost ratio 50, ratio of standard deviation to average fitness-12 data points selected at random 75

List of Tables

3.1 Test Problems 43

3.2 Uniform-Ratio 1 45

3.3 Uniform-Ratio 5 45

3.4 Uniform-Ratio 10 46

3.5 Uniform-Ratio 20 46

3.6 Uniform-Ratio 50 46

3.7 Normal-Ratio 1 47

3.8 Normal-Ratio 5 47

3.9 Normal-Ratio 10 47

3.10 Normal-Ratio 20 48

3.11 Normal-Ratio 50 48

Chapter 1

Introduction and Literature Review

1.1 Introduction

The Uncapacitated network design problem has been studied in depth in the past several decades (see Billheimer and Gray [6], Magnanti and Wong [24], Balakrishnan, Magnanti and Wong [3]). Many real-world problems involve network design of some sort. The construction of telephone lines, road networks and airline routes, capital investment decisions, road repair and cleaning strategies, transit design are some examples of network design applications. More applications are discussed in Section 2.3.

Consider a network G with a node set N, arc set A and a commodity set K. Each arc of A has a construction cost and an usage cost. The construction cost is a one-time cost incurred for construction of the arc while usage cost is a variable cost incurred for each unit of flow along that arc. No flow may be sent in an arc unless it is

constructed. There is one unit of demand from each node i to each node j, and each requirement may be viewed as a different commodity. The network design problem is to satisfy the commodity demands at a minimum cost. In the uncapacitated network design problem, there is no upper bound on the flow in an arc, once it is constructed.

In trying to route a transit system through an array of stations, a designer must resolve the conflict between construction economy and traveler convenience (see Billheimer and Gray [6]). Consider the problem of linking residential stations and business stations of a city. A design solely based on traveler convenience would provide direct routings linking each origin and destination. If the minimization of construction costs were the sole criterion used in designing the required system, a minimum spanning tree would result as the solution model. Although this would minimize the total construction cost, certain origins and destinations will be poorly served. The problem addressed, here, is a network design problem of striking an acceptable balance between construction economy and traveler convenience. The stations model nodes and the transit lines model arcs. The passenger travel demands will represent commodity demands. The construction cost will represent the fixed design cost and a measure of traveler convenience will represent the routing cost. The designer would want to minimize the total cost.

These problems, in their mathematical programming models, are mixed integer programming problems (i.e., they have both integer and continuous decision variables). The integer variables concern construction decisions, whether or not an arc is to be constructed. The continuous variables concern flow decisions (after an arc is constructed how much flow is to pass through this arc?). This model can, therefore, consider trade-offs between flow costs and design costs for satisfying the demands. These problems belong to the class of hard problems called NP-hard problems (Johnson, Lenstra and Rinnooy Kan [20]). Considerable progress has been made in finding exact and approximate algorithms for these problems based on optimization ideas (Magnanti and Wong [24]). As the problems are NP-hard, there are no efficient exact

solution techniques guaranteed to work for large problems.

The absence of efficient exact solution techniques for such combinatorially hard problems has given rise to increased application of random search techniques (Tailard [31], Anderson and Ferris [1]) as alternate solution methods. In this thesis, we develop, for the first time in the literature, an implementation of a search technique called genetic algorithms for the uncapacitated network design problem.

Our implementation starts with an initial population that has individuals (solutions) from the search space of feasible solutions. This population changes from iteration to iteration. Each iteration includes processes called "parental selection", "crossover" and "immigration". While creating the initial population and during subsequent iterations, the genetic algorithm attempts to find successively "fitter" populations while maintaining an appropriate amount of population diversity. The *fitness* of an individual is given by its objective function value.

The iterations do the following. Two individuals are *selected* from the population to *crossover* and form a child. The newly created child is introduced in the population if it meets pre-set criteria related to its feasibility and its objective function value. *Immigration* introduces new individuals into the population. The average fitness of the population improves over iterations and also the fitness of the worst-individual of the population improves over iterations. After a number of iterations, the algorithm typically reaches a solution that is near-optimal.

1.2 Problem Definition

Our model of the uncapacitated network design problem (UNDP) has the following features. It is defined on a network with node set N and an arc set A. Each arc (i, j) of A is uncapacitated and undirected and has a fixed design cost F_{ij} and a variable routing cost c_{ij}^k for each unit of flow sent for commodity k. The construction of more arcs will improve the total routing cost but will increase the total design cost. The

construction of fewer arcs will improve the total design cost but will increase the total routing cost. The objective is to minimize the sum of routing and design costs.

The model allows multiple commodities. Let K be the set of commodities and for each $k \in K$, assume (by scaling, if necessary - see Balakrishnan, Magnanti and Wong [3]) one unit of commodity k must be shipped from its origin $O(k)$ to its destination $D(k)$. We refer to the set of constructed arcs for a solution of this model as a *design or configuration.*

The model has two types of decision variables, y_{ij} and x^k_{ij}; y_{ij} is a binary integer design variable taking a value 1 if arc (i,j) is in the design (i.e., it is constructed) and a value 0 if arc (i,j) is not in the design. x^k_{ij} denotes the flow of commodity k on the arc (i,j).

The model is given by the following mathematical program:

$$\text{Minimize} \sum_{k\in K} \sum_{(i,j)\in A} (c^k_{ij}x^k_{ij} + c^k_{ji}x^k_{ji}) + \sum_{(i,j)\in A} F_{ij}y_{ij} \tag{1.1}$$

subject to

$$\sum_{j\in N}(x^k_{ji}) - \sum_{l\in N}(x^k_{il}) = \begin{cases} -1 & \text{if } i = O(k) \text{ for all } i \in N \text{ and } k \in K \\ 1 & \text{if } i = D(k) \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

$$x^k_{ij} \leq y_{ij} \text{ and}$$

$$x^k_{ji} \leq y_{ij} \text{ for all } (i,j) \in A \text{ and } k \in K \tag{1.3}$$

$$x^k_{ij}, x^k_{ji} \geq 0 \text{ for all } (i,j) \in A \text{ and } k \in K \tag{1.4}$$

$$y_{ij} = 0 \text{ or } 1 \text{ for all } (i,j) \in A \tag{1.5}$$

The constraints of our model are of 4 types. They are *flow demand constraints* (1.2) that require a unit flow for each commodity, *arc usage constraints* (1.3) requiring design (or construction) of an arc before it can be used for flow, *nonnegativity con-*

straints (1.4) requiring continuous variables to take nonnegative values, and *binary integer constraints* (1.5) determining usage status of arcs.

The formulation has $|A|$ integer variables, $2|A||K|$ continuous variables, $|N||K|$ flow demand constraints, $2|A||K|$ arc usage constraints, $2|A||K|$ nonnegativity constraints and $|A|$ binary integer constraints.

To appreciate how the formulation increases with the size of the network, consider a 15-node complete network with complete commodities. By complete network and complete commodities, we mean that this network has an arc between each pair of distinct nodes and a commodity for each pair of distinct nodes of the network . Thus the 15-node network has 105 arcs and 210 commodities. We observe that there are 105 integer variables, 44,100 continuous variables, 3,150 flow demand constraints, 44,100 arc usage constraints, 44,100 nonnegativity constraints and 105 binary integer constraints.

The fixed design cost is F_{ij} associated with each arc (i, j) and the variable routing cost is c_{ij}^k associated with each arc (i, j) and commodity k. The objective is to obtain a design and a corresponding flow that will satisfy commodity demands and will minimize the total cost.

Many network flow and network design problems are special cases of this problem. When the design cost is 0 for each arc, then the UNDP reduces to a set of shortest path problems-one shortest path problem for each commodity. When the commodity set is complete and the routing cost is 0 for each arc, then the UNDP reduces to a minimum spanning tree problem (Magnanti and Wong [24]). The transportation problems and traveling salesman problem (see Magnanti and Wong [24]) are special cases of this model.

1.3 Literature Review

Researchers have developed both exact and approximate solution methods for solving UNDP. Enumerative algorithms like branch and bound, non-enumerative methods like Benders Decomposition, heuristic methods, dual-ascent methods in combination with drop-add heuristics have been tried. Enumerative algorithms reach the optimal solution but in the worst case they may enumerate all feasible solutions. Most non-enumerative methods maintain a lower bound and upper bound for the objective function while searching for a solution, and stop the search when the algorithm is within a pre-specified distance from optimality. The above methods are explained in greater detail in Section 2.2.

Branch and bound algorithms have been proposed by several researchers (Boyce et al. [7], Boyce and Soberanes [8], Dionne and Florian [12], Hoang [15], Leblanc [18], Los and Lardinios [19], Rothengatter [27], and Scott [29]) for network design.

Boyce et.al have suggested an enumerative algorithm for budget design problems (a variation of UNDP) and they noted a bounding mechanism for this algorithm. Hoang has proposed an improvement on the bounding procedure of Boyce et al. Dionne and Florian have noted several ways to improve this algorithm. Los and Lardinios have adapted the Dionne and Florian lower bound for the UNDP.

The branch and bound method is found to be effective for small and mid-size problems. As the problem size grows, the computational time is found to grow exponentially.

Benders [5] decomposition, an algorithm for mixed integer programming problems, has been successfully applied to network design problems (see Magnanti and Wong [24]). Florian et al. [13] have used the algorithm to schedule the movement of railway engines; Richardson [26] has applied this algorithm to the design of airline routes; Magnanti et al. [22] have used the algorithm for fixed charge network design; Geoffrion and Graves [14] applied this procedure to industrial distribution system network design; and Hoang [16] has used an extended version of the algorithm to solve a class

of nonlinear discrete network design models. Methods have been proposed to accelerate the algorithm by using improved cuts called "pareto-optimal" cuts (Magnanti and Wong [23], Magnanti et al. [22]).

The route selection algorithm (drop-add heuristic) has been suggested by Billheimer and Gray [6] for solving network design problems. This algorithm converges to a local optimum. When combined with methods like dual ascent (Balakrishnan, Magnanti and Wong [3]), the drop-add heuristic gives a nearly optimal solution.

Balakrishnan, Magnanti and Wong [3] have applied the dual-ascent procedure with the drop-add heuristic for the UNDP on random test problems with up to 45 nodes, 500 arcs and 1980 commodities and also on some models arising in freight transport. The solutions were typically found to be from 1 to 4% of optimality.

1.4 Overview of Thesis

In this thesis, we develop a genetic algorithm to solve uncapacitated network design problems. Our approach applies several modifications to the traditional genetic algorithms methods. Some of these modifications involve introducing problem specific information to the genetic algorithm.

Chapter two of this thesis considers the UNDP model introduced in Chapter 1 with brief comments on possible modifications in the problem formulation. The solution methods presented in Chapter one to solve uncapacitated network design problem are explained in detail here. The scope and limitations of each method are noted. This chapter also presents some applications of network design problems.

Chapter three gives a brief introduction to notation and terminology in genetic algorithms. Then, it presents a detailed account of our approach in using a genetic algorithm for UNDP. The special steps and variations from traditional methods present in our approach are noted and analyzed. This chapter explains in detail the computational implementation methods of genetic algorithm and dual-ascent (with

drop-add heuristic) for UNDP. Several problem instances from different problem sizes are solved using both methods, and the computational results are presented and interpreted. This chapter also gives a detailed graphical analysis of the results observed.

Chapter four concludes with suggestions for future research in using genetic algorithms for UNDP. Here, we address possible improvements in the traditional genetic algorithms implementation and our implementation that would use information from the problem more effectively. We also present our observations on the scope of applying genetic algorithms to network design problems.

The appendix contains the C++ code for our genetic algorithm implementation for UNDP.

Chapter 2

Uncapacitated Network Design

In this chapter, we consider the UNDP introduced in Chapter one and discuss about the possible alternate formulation methods. Then, we give a presentation of the existing solution methods used to solve UNDP. The solution methods are explained in detail, and their scope and limitations are briefly discussed. Then, we present applications of network design problems in areas including the transportation, urban planning and telecommunication.

2.1 Alternate Formulations

We consider again the UNDP described in Section 1.2. This UNDP can be formulated in alternate ways as follows. The arc usage constraints can be aggregated for each commodity and can be written as one for each arc, thus reducing the $2|A||K|$ constraints to $2|A|$ constraints as follows:

$$\sum_k (x_{ij}^k + x_{ji}^k) \leq 2|K| y_{ij} \text{ for all } (i,j) \in A. \tag{2.1}$$

The arc usage constraints of both formulations ensure usage of an arc for flow only after it is constructed. But, the second formulation has fewer constraints and hence is more compact.

Another formulation method is to aggregate commodities. Instead of having one commodity for each pair of origin-destination nodes, Balakrishnan, Magnanti and Wong [3] suggest a method to aggregate commodities corresponding to each origin (or destination) and to represent them as one commodity. According to this formulation, x_{ij}^k will denote the total flow on arc (i, j) that originates (terminates) at node k. This formulation corresponds to aggregation over all destination nodes of the commodities with origin k. The arc usage constraints will then become

$$\left.\begin{array}{lcl} x_{ij}^k & \leq & |N| y_{ij} \\ x_{ji}^k & \leq & |N| y_{ij} \end{array}\right\} \text{for all } (i, j) \in A \text{ and for all } k \in N \tag{2.2}$$

The flow demand constraints are also altered. For each commodity l (of the original formulation) with $O(l) = k$, there is a requirement of 1 unit at node $D(l)$ for commodity k (of this formulation).

Even though the disaggregate (original) formulation contains more constraints, it is preferable to the aggregate formulations in the following sense. The linear programming relaxation of the disaggregate formulation is tighter than that of the aggregate formulation and hence gives sharper lower bounds for the optimal objective value of the mixed integer programming formulation. Many authors have noted the algorithmic advantages of using tight linear programming relaxations (see Cornuejols, Fisher and Nemhauser [9], Davis and Ray [10], Beale and Tomlin [4], Geoffrion and Graves [14], Mairs et al. [21], Magnanti and Wong [23], Rardin and Choe [25], and Williams [32]).

2.2 Existing Solution Methods

2.2.1 Branch and bound method

The branch and bound method has been suggested as a solution method for network design problems by several researchers (Boyce et al. [7], Boyce and Soberanes [8], Dionne and Florian [12], Hoang [15], Leblanc [18], Los and Lardinios [19], Rothengatter [27], and Scott [29]). The branch and bound method may be summarized as follows: The method maintains the branch and bound enumeration tree consisting of partial solutions. The possible solutions of the network design problem are the leaves of this tree. By using a bounding mechanism, the method branches on this tree in search of a leaf that is an optimal solution for the problem.

Boyce et al. [7] suggested this approach for budget design problems. Budget design problems are network design problems where there is an upper bound on the number of arcs constructed but there is no fixed cost for each arc. Here, both integer and continuous variables appear in the constraints, but only the continuous variables appear in the objective function. In this case, if we know the optimal design y, the optimal objective function value $G(y)$ is the optimal routing cost of this design.

Each node P in the branch and bound enumeration tree represents a partial solution. It has associated disjoint sets F_1 and F_0 of A such that arcs in F_1 are required to be constructed ($\hat{y_{ij}}$=1) and arcs in F_0 are required not to be constructed ($\hat{y_{ij}}$=0). The remaining arcs F_2=A-F_1-F_0 are arcs whose construction status is not yet determined. Node P's decided arcs combined with different construction status for the arcs in F_2 will form the subtree rooted at P. The leaves of this enumeration tree are possible solutions for the problem.

Boyce et al. noted that for any solution y that belongs to the subtree rooted at P, a lower bound for the objective value is given by

$$G(y) \geq G(y^P) \tag{2.3}$$

where y^P denotes the "completion" of P ("completion" of a node P is the configuration in which all its undecided arcs are constructed) with $y_{ij} = 1$ for every arc $(i,j) \in F_2$. This is due to the fact that with the additional arcs in y^P the routing cost can be at most the same as that of y.

Hoang has proposed an improvement on the bounding procedure of Boyce et al. as follow:

$$G(y) \geq G(y^P) + \sum_{(i,j)\in F_2} \bar{y_{ij}} I_{ij}(y^P) \tag{2.4}$$

where $\bar{y_{ij}} = 1 - y_{ij}$. The deletion of arcs from y^P will result in an increase in the shortest route cost and hence increase in the objective function value. The quantity $I_{ij}(y^P)$ denotes the increment to the shortest route cost from node i to node j when we delete (i,j) from the network defined by y^P. The expression (2.4) means that if the arc (i,j) is deleted (set $y_{ij} = 0$ and $\bar{y_{ij}} = 1$) from the network defined by y^P, then the cost of shipping the unit of demand between the nodes i and j must increase by at least $I_{ij}(y^P)$ in the solution y. It might increase by more as other arcs might be deleted as well.

(2.4) gives a lower bound $G(y) \geq G(y^P) + \sum_{(i,j)\in F_2} \bar{y_{ij}} I_{ij}(y^P)$ that applies to every node below P in the branch and bound enumeration tree. Using this lower bound as a fathoming mechanism and branching from node P on a fractionally valued variable, Hoang ultimately reaches a tree solution which is an optimal solution for the problem.

Dionne and Florian [12] suggested improvements for this algorithm such as using specialized and faster algorithms to compute shortest path distances when one arc has been deleted from the network. They also suggested branching on the fractional variable y_{ij}, $(i,j) \in F_2$ with the highest incremental improvement per unit of budget.

Los and Lardinios [19] have generalized the Dionne and Florian lower bound for

the uncapacitated network design problem. They use the inequality

$$G(y) \geq G(y^P) + \sum_{(i,j) \in F_2} \bar{y_{ij}} I_{ij}(y^P) + \sum_{(i,j) \in A} F_{ij} y_{ij} \tag{2.5}$$

For a given solution y, (2.4) will provide a lower bound to the total routing cost and the term $\sum_{(i,j) \in A} F_{ij} y_{ij}$ represents the total fixed charge costs. The sum of the two expressions will thus be a lower bound to the objective value at y.

Los and Lardinios found that this method was effective only for small to medium-sized problems with computation times growing exponentially in the problem size. For example, an 8-node and 23-arc problem required 5.4 seconds on a CDC computer while a 12-node and 40-arc problem required 207 seconds.

2.2.2 Benders decomposition

The Benders decomposition, an algorithm for mixed integer programming problems, has been applied to a variety of network design problems. Florian et al. [13] have used the algorithm to schedule the movement of railway engines. Richardson [26] applied it to the design of airline routes. Magnanti et al. [22] applied this algorithm for fixed charge network design. Hoang [16] used an extended version of the algorithm, called generalized Benders decomposition, to solve a class of nonlinear discrete network design models.

When applied to a network design problem, Benders decomposition proceeds iteratively by choosing a tentative network configuration (i.e., setting values for the integer variables y_{ij}), solving for the optimal routing on this network and using the solution to the routing problem to further constrain the network configuration.

The optimal dual variables for the linear programming routing problem can be used to estimate the maximum possible savings on introduction of an unused arc. Using this, we can get a lower bound (LB) expression for the optimal routing cost R. For any network configuration, $y_{ij} = 1$ if the arc (i, j) is in the configuration and

$y_{ij} = 0$ if (i, j) is not present in the configuration. Thus, the total design cost equals $\sum_{(i,j)\in A} F_{ij} y_{ij}$. Therefore, this expression, added to the previous routing cost lower bound expression (LB) gives a lower bound on the minimum cost v of any design of this problem.

Constraints like the above constraint are called *Benders cuts.* They are derived from the dual optimal solution for the optimal routing calcuation for any tentative network configuration. The Benders algorithm computes the new configuration at each step by minimizing the total network cost v subject to the Benders cuts generated by every previous configuration. This minimization, called the *Benders master problem* (Magnanti and Wong [24]), is a mixed integer program in the integer variables y_{ij} and the single continuous variable v. Since every Benders cut gives a valid inequality involving the optimal solution value v, the optimal value v^* of the Benders master problem is a lower bound on the optimal value v to the original problem. The objective function value corresponding to this network configuration will serve as an upper bound to v. These two bounds provide an assessment of the degree of suboptimality and may be used in heuristics that permit an early termination of the algorithm.

When there are more than one dual optimal solution for the linear programming routing problem, some solution may produce improved cuts (better lower bounds) than other solutions. Such an improved cut called *"pareto-optimal"* cut may be used to accelerate the Benders decomposition. This improvement is generally possible because of degeneracy in the shortest path linear program. When there is degeneracy in this problem, its dual usually has multiple optimal solution. Magnanti and Wong [23] and Magnanti et al. [22] show how to generate improved (pareto-optimal) cuts for arbitrary mixed integer programming problems by solving a linear program to choose from among alternate optimal dual solutions.

The computational experience on a variety of uncapacitated undirected fixed charge design problems were very promising. The Benders decomposition with pre-

processing and pareto optimal, or other improved cuts, found and verified an optimal solution for 19 out of 24 test problems with up to 30 nodes, 130 arcs with 40 arcs fixed open (construction required), and 58 commodities (see Magnanti and Wong 1984 [24]). For all but one of these test problems the algorithm finds feasible solutions that are guaranteed to be within 1.71% of optimality [24].

2.2.3 Route selection algorithm

The route selection algorithm (Billheimer and Gray [6]) applies routines called link elimination and link insertion iteratively to feasible solutions and converges to a local optimum. This algorithm has 3 main components called the assignment routine, the link elimination routine and the link insertion routine. The route selection algorithm is also called drop-add heuristic.

The algorithm starts with a solution where all potential network arcs are constructed. The *assignment routine* computes the shortest path between all nodes, assigns flow to the shortest path and computes an initial objective value. The algorithm then goes to the link elimination routine.

The *link elimination routine* first computes the second shortest path between all node pairs joined by a single link (i, j) in the present configuration U. An improvement parameter $\Delta_{ij} = x_{ij}(U)[d_{ij}(U) - \hat{d_{ij}}(U)] + F_{ij}$, is computed for each node pair combination joined by a single link by assuming that the traffic assigned to the link will be detoured to the second shortest path if (i, j) is eliminated. Here, $x_{ij}(U)$ represents the total flow on link (i, j) for the configuration U, $d_{ij}(U)$ and $\hat{d_{ij}}(U)$ denote respectively the costs of the shortest and second-shortest paths between i and j in configuration U. Thus, Δ_{ij} computes the savings observed in the objective value by eliminating the link (i, j) from U.

After calculating the improvement parameter for all links in U, the link whose elimination brings the greatest improvement in the objective function is removed from U. The improvement parameters are recalculated for the links involved in the

new configuration and the elimination process is repeated. When no additional eliminations appear attractive (i.e., $\Delta_{ij} \leq 0$ for all (i, j) in the configuration), the resulting configuration is subjected to the shortest-route assignment routine, and new values are calculated for detour routes and improvement parameters. If there are still no attractive eliminations, the algorithm goes to the link insertion routine.

The *link insertion routine* considers previously eliminated links for readmission based on an insertion criteria. An improvement parameter δ_{mn} is computed that compares the fixed cost of the candidate insertion arc (m, n) with the variable improvement in the network performance by readmitting the arc.

Here, $\delta_{mn} = \sum_{ij} max[0, d_{ij}(U) - d_{im}(U) - c_{mn} - d_{nj}(U)] - F_{mn}$ where the ij are the commodity origin-destination pairs (they have unit demands). So, the expression $d_{ij}(U) - d_{im}(U) - c_{mn} - d_{nj}(U)$ represents the improvement in variable routing cost between nodes i and j realized by inserting the (m, n) in network U. δ_{mn}, thus, gives the improvement in the objective function by introducing arc (m, n).

Once candidates for insertion have been identified, the link whose insertion will give the highest positive improvement is inserted and a new configuration is formed. The algorithm, then, reperforms the assignment and link elimination routines, ultimately returning to the link insertion routine. The algorithm stops when successive applications of the link elimination and link insertion routines produce no improvement in the objective value.

The computation time appears to increase with the square of the number of nodes $|N|$ when the number of potential links can be expressed as a low multiple of $|N|$ (Billheimer and Gray [6]).

2.2.4 Dual-ascent method

Balakrishnan, Magnanti and Wong [3] have developed a family of dual-ascent algorithms for the uncapacitated network design problem. As the problem is NP-hard, the algorithm focuses on methods for generating good lower bounds and heuristic

solutions rather than solving the problem optimally.

The dual-ascent algorithm considers the dual of the linear programming relaxation of the network design problem which is the following:

Problem DP

$$\text{Maximize } z_D = \sum_{k \in K} v^k_{D(k)} \tag{2.6}$$

subject to

$$\left.\begin{array}{rcl} v^k_j - v^k_i & \leq & c^k_{ij} + w^k_{ij} \\ v^k_i - v^k_j & \leq & c^k_{ji} + w^k_{ji} \end{array}\right\} \text{for all } k \in K \text{ and all } \{i,j\} \in A \tag{2.7}$$

$$\sum_{k \in K} w^k_{ij} + \sum_{k \in K} w^k_{ji} \leq F_{ij} \text{ for all } (i,j) \in A \tag{2.8}$$

$$w^k_{ij}, w^k_{ji} \geq 0 \text{ for all } k \in K \text{ and all } (i,j) \in A \tag{2.9}$$

In this formulation v^k_i is a dual variable corresponding to the flow demand equations (1.2) for commodity k at node i, w^k_{ij} and w^k_{ji} for $k \in K$ and $\{i,j\} \in A$ correspond to the arc usage constraints (1.3).

For any given vector w=$\{w^k_{ij}, w^k_{ji}\}$ that satisfies constraints (2.8) of DP, the best v values are obtained by maximizing (2.6) subject to (2.7). This subproblem decomposes by commodity; The subproblem $SP_k(w)$ corresponding to commodity k has the following structure.

Problem $SP_k(w)$

$$\text{Maximize } v^k_{D(k)} \tag{2.10}$$

subject to

$$\left.\begin{array}{rcl} v_j^k - v_i^k & \leq & \hat{c}_{ij}^k \\ v_i^k - v_j^k & \leq & \hat{c}_{ji}^k \end{array}\right\} \text{for all } k \in K \text{ and all } (i,j) \in A \text{ where} \tag{2.11}$$

$$\hat{c}_{ij}^k = c_{ij}^k + w_{ij}^k \text{ and } \hat{c}_{ji}^k = c_{ji}^k + w_{ji}^k$$

$$\text{for all } (i,j) \in A \text{ and } k \in K.$$

$SP_k(w)$ is the dual of a shortest path problem from origin $O(k)$ to destination $D(k)$ using the *modified arc lengths* $\hat{c}_{ij}^k = c_{ij}^k + w_{ij}^k$ and $\hat{c}_{ji}^k = c_{ji}^k + w_{ji}^k$. For a given set of w-values, setting v_i^k equal to the length of the shortest path from origin $O(k)$ to node i, using $\hat{c}_{ij}^k$ and $\hat{c}_{ji}^k$ as arc length, gives one optimal solution of $SP_k(w)$. In particular, the optimal value of the subproblem $SP_k(w)$ is $v_{D(k)}^k$, the length of the shortest path from origin $O(k)$ to destination $D(k)$. Therefore, we can increase the dual objective function value by increasing the length of the shortest O-D (origin-destination) path for one or more commodities through appropriate increases in w-values (and, hence in $\hat{c}$-values).

The dual-ascent strategy is to iteratively increase one or more w-values so that

a. all constraints of Problem DP remain feasible.

b. the shortest O-D path length $v_{D(k)}^k$ increases for at least one commodity at each stage.

To satisfy condition *a*, Balakrishnan, Magnanti and Wong [3] consider increasing w_{ij}^k and w_{ji}^k values corresponding only to those arcs (i,j) for which constraint (2.10) has slack.

Balakrishnan, Magnanti and Wong adopt a procedure called the *labeling method* to implement this strategy. For a single commodity $k \in K$, this method simultaneously increases several w-values.

For any commodity $k \in K$, this method considers a partition $(N_1(k), N_2(k))$ of the node set N so that $O(k) \in N_1(k)$ and $D(k) \in N_2(k)$. The aim is to identify arcs (i,j) whose w_{ij}^k values must be increased in order to increase the shortest O-D distance

$v^k_{D(k)}$. The method defines a set $A(k)$ as the set of arcs (i,j) incident from $N_1(k)$ to $N_2(k)$. $A(k)$ is called the *cutset* for commodity k induced by the node partition $(N_1(k), N_2(k))$. Increasing the w^k_{ij} values for each of the arcs in $A(k)$ will increase the shortest path distance for commodity k.

However, only a subset of the arcs in the cutset will belong to the shortest paths and it will suffice to increase those w^k_{ij} values alone. This method has mechanisms to identify these arcs (called tight arcs). The other arcs of the cutset may be called non-tight arcs.

Then, it computes the slack s_{ij} for each tight arc (i,j) and the minimum of all s_{ij} is δ_1. It also computes the maximum quantity δ_2 of increase in w-values that will make one or more of the non-tight arcs tight. Then, increasing w^k_{ij} by δ=min$\{\delta_1,\delta_2\}$ for all tight arcs increases all shortest path lengths v^k_l to nodes l of $N_2(k)$ by δ, and improves the lower bound (Balakrishnan, Magnanti and Wong [3]). They refer to this updating procedure as the *simultaneous w-increasing* step.

For this δ, the slacks and the shortest path lengths are updated. Here, there is at least one arc whose slack goes to 0.

The procedure initializes all w-values to zero and initially, $N_1(k) = N\setminus \{D(k)\}$, $N_2(k) = \{D(k)\}$ for all $k \in K$. At each iteration, the algorithm sequentially considers the commodities k for which $O(k) \notin N_2(k)$. For every such commodity, the implementation performs the simultaneous w-increasing step once. If, as a result, the slack for some tight arc (i,j) becomes 0, then node i is transferred from $N_1(k)$ to $N_2(k)$. This augmenting of $N_2(k)$ is called labeling and nodes of $N_2(k)$ are called labeled nodes. The algorithm stops when the origin $O(k)$ is labeled for all commodities $k \in K$.

When the dual-ascent procedure terminates, the design consisting of only zero-slack arcs is feasible (Balakrishnan, Magnanti and Wong [3]). For this design, solving a shortest path problem for each commodity $k \in K$ gives the arc flows. Using this as an initial feasible solution, Balakrishnan, Magnanti and Wong apply drop-add

heuristic (as in route selection algorithm) to get an improved solution. They also use dual-based problem reduction methods to improve the solution. The resulting solution will give an upper bound for the objective value.

To summarize, their algorithm applies dual-ascent labeling method iteratively to get good dual solutions and lower bounds. A heuristic based in these dual solutions gives feasible solutions that serve as starting points for improvement heuristic and problem reduction methods. The resulting solutions are nearly optimal in practice, as the dual ascent lower bounds provide a measure of the distance from optimality.

Their computational results for classes of test problems with up to 500 integer and 1.98 million continuous variables and constraints show that the dual-ascent procedure and an associated drop-add heuristic generate solutions, that, in most cases, are guaranteed to be within 1 to 4% of optimality. The CPU time is not more than 150 seconds on an IBM 3083 computer (Balakrishnan, Magnanti and Wong [3]).

2.3 Applications of network design problems

We can see network design problems in diverse application contexts including package deliveries, communication networks, waste disposal, mail routing, transportation planning, urban traffic planning and transit design (see Billheimer and Gray [6]).

While delivering packages and commodities from one city to another city, nodes model the cities. Arcs model the possible truck (freight) routes. Fixed and variable costs are respectively the cost incurred in running a truck (or other vehicle) and the handling cost per unit of freight. Commodity demands are experienced between cities. The goal is to determine a minimum cost usage of trucks (vehicles) that satisfies the delivery demands.

In the communication networks, nodes model the message centers. Arcs model the connecting lines. The lines have a fixed construction cost and a variable maintenance cost. The commodity demands are message transmission demands. The goal, again,

is to find a collection of lines to be constructed that would result in the minimum total cost.

In waste disposal, the waste collection and discharge points are represented as nodes. The flowing pipelines are represented as arcs. The pipelines have construction costs and maintenance costs. The commodity demands are between collection and discharge points. We want to design and operate the disposal process at minimum cost.

The mail routing is done between post offices which are represented as nodes. The travel routes are represented as arcs. The fixed cost corresponds to establishing and running travel mechanisms. The variable cost relates to handling costs. The commodity demands are the mail routing needs between different post offices. The problem is to satisfy the demands at minimum total cost.

In air-route or road construction between cities, cities are represented as nodes. Arcs model the possible airways or roads. The fixed cost is the construction cost, and the variable cost is maintenance cost. The commodity demands are the travel demands. The goal is to construct and operate the road system (or air-route system) at minimum total cost.

In urban planning, locations in the city are represented as nodes. The travel routes (roads) are represented as arcs. These arcs have construction and maintenance cost. The traffic demands are observed between locations. We want to satisfy the demands at a minimum cost.

In designing transit routes, the stations will be represented as nodes. The arcs will model lines between stations. The demands will be travel demands between stations. The goal is to design transit routes so as to minimize the costs.

Chapter 3

Genetic Algorithms for Uncapacitated Network Design Problems

This chapter starts with a brief introduction to the notation involved in a heuristic search strategy called genetic algorithms. This is followed by a detailed description of a genetic algorithm that we developed to solve UNDP. This description is supported by required background on the corresponding and related methods used in the genetic algorithms literature. This chapter also presents implementation details of the genetic algorithm and the dual ascent method with drop-add heuristic for solving the UNDP. We perform computational testing of our implementation for different problem sizes of UNDP and compare our results with the corresponding results using dual ascent with drop-add heuristic. We analyze the performance in different dimensions and support our analysis with related graphs and charts.

3.1 Basic terminology

Genetic algorithms (Holland [17]) are solution techniques for optimization problems that imitate biological evolution. When a genetic algorithm is used to solve an optimization problem, each feasible solution is referred to as an *individual.* This feasible solution is represented as a *string* (also called a *chromosome*), a genetic representation of the individual. The components of the string (representing different characteristics of the individual) are called *genes.* The possible values, a gene can take, are referred to as its *alleles.*

The pool of all possible solutions (individuals) is called a *search space.* Some representation strategy is adopted to *encode* the individuals into chromosomes.

Each individual has an associated fitness which is typically the objective value of the individual or some variation of it. A genetic algorithm undergoes the following processes in its attempt to find solutions with improved fitness.

A subset of the search space called a *population* is considered. An *initial population* with substantial diversity is selected from the search space. A random selection of individuals is one way to generate an initial population. The individuals of a population undergo iterative transformations. Each iteration may consist of processes (such as selection, crossover, mutation) to "exploit the good genetic material" from the population and to "explore the search space" for new genetic material. The heuristic search terminates when the population has converged (that is, some measure of the diversity of the population goes below a specific threshold or a maximum number of iterations is reached).

3.2 A genetic algorithm for UNDP

We *represent* each feasible solution (individual) of UNDP by a string of $|A|$ genes, one for each arc of the network. An allele is 0 for a gene indicates that the corresponding arc is not present in the solution. Similarly, the allele is 1 if the corresponding arc is

present in the associated solution.

The *fitness* of a solution is its objective value. Here, our problem is a minimization problem and hence an optimum solution has minimum fitness among all feasible solutions. The direction of improvement is therefore the minimizing direction.

We start with an initial *population* whose size is usually determined by experimentation. For our test problems, with $|N|$ varying from 10 to 35, we selected after experimentation, a population size of 200. When the initial population size is higher, the population represents a bigger segment of the search space and hence possibly will lead to better solutions. On the other hand, more members in the initial population would mean more time spent in fitness computations and other processes and hence will worsen the total computational time.

In *creating the individuals* of the population, we combine elements of "randomness" and "greed" in our method. For creating an individual for UNDP, our approach follows one of the following 3 steps.

- With a 10% probability, the algorithm chooses a minimum cost spanning tree of the network as the individual.
- With a 10% probability the algorithm chooses $|N| - 1$ least cost arcs and adds additional arcs to make the design connected.
- With an 80% probability, the algorithm forms a tree (breadth-first spanning tree of the original network not using any cost information) rooted at a node chosen at random and adds additional arcs (number of arcs to be added found by random choice). While adding the arcs, the implementation chooses low design cost arcs with higher probability. Specifically, the arcs are ranked from 1 to $|A|$ according to their design cost (least design cost arc ranked 1), and the probability of choosing an arc ranked $|A| - i + 1$ (i=1 to $|A|$) is i times the probability of choosing the arc ranked $|A|$.

The approach *computes the fitness* of an individual after using Dijkstra's algorithm

to compute the shortest route distances (as restricted to arcs that are in the design).

In most genetic algorithms, the population is transformed iteratively until satisfying a stopping criteria. It undergoes processes such as selection, crossover and mutation at each iteration. In our approach, we substitute "immigration" for mutation.

The *selection* (see Back and Hoffmeister [2]) of an individual is generally the process by which fitter members of the population are given higher representation than the less fit ones. The selection criteria may be based on the individual's fitness proportion (to the total fitness) or the rank of its fitness (in the population). The selection is categorized into different types based on the process used for selection. Some processes used in the literature for selection are as follows:

- *Probabilistic* processes where individuals have a certain probability to get selected vs. *Deterministic* processes where a fixed number (on the basis of fitness) of copies of each individual get selected;

- *Dynamic* methods where probabilities of selection varies between generations vs. *Static* methods where probabilities are the same for each generation;

- *Generational* methods where the whole population undergoes selection vs. *Steady State* methods where only a portion of the population undergoes selection;

- *Universal* methods where selection takes place in one step (say one spin of roulette wheel) vs. *Independent* methods where selection takes place in multiple steps (say several spins of roulette wheel);

- *Replaceable* methods where if an individual is selected once, it is still considered for selection vs. *Nonreplaceable* methods where if an individual is selected, it is not considered for selection in that process anymore;

- *Preservative* methods where each individual has a nonzero probability of getting selected vs. *Extinctive* methods where some individuals have zero probability

of getting selected;

- *Elitist* strategies where the best individual is retained vs. *Pure* strategies where each individual has a probability less than one of getting selected;

Our method of selection called *parental selection* works as follows: The population is ranked according to the fitness of the member individuals. We call the half having better fitness (lower objective value) the better half and we call the other half the worse half. The parental selection method selects two individuals from the population and calls them the *better parent* and *worse parent* according to their relative fitness values. The selection of parents is done as follows: With a probability p_1 (we use p_1=0.75), the method selects at random one parent from each half. With probability $1 - p_1$ it selects at random both parents from the better half.

The selected parents then undergo *crossover*. For each pair of parents, the crossover is a recombination of genetic material within the parents to produce possibly better individuals. The proportion of population selected as parents each time is generally given by the *crossover ratio*. The individuals produced in this process are called children. The diversity and the fitness of the output population is to some extent governed by the method of crossover used. Some methods used for crossover are as follows.

- In *one point crossover*, a particular gene in the chromosome is chosen to be the crossover point. All alleles to the right of this gene are interchanged between the combining individuals, producing two children.

- In *multipoint crossover* (De Jong [11]), there are more than one combination or crossover points, also producing two children.

- In *segmented crossover*, the number of segments to undergo crossover each time is not fixed.

- In *uniform parametrized crossover*, with a probability p an allele is in child I and with probability $(1-p)$ it is in child II.
- In *adaptive crossover*, crossover points evolve as the generations go on.

Our crossover method uses some ideas from *uniform parametrized crossover* (Spears and De Jong [30]). In our method, the parents chosen in the parental selection undergo crossover and each crossover produces only one child. When a gene has the same allele in both the parents, then the corresponding gene in the child gets the same allele. When the alleles corresponding to a gene in the two parents are different then with a probability p_2 (in our implementation p_2 =0.75), the corresponding gene of the child takes the allele from the better parent and with probability $1-p_2$ it takes from the worse parent.

The newly created child is then tested for feasibility (i.e., whether the constructed arcs form a connected spanning subgraph). For a feasible child, its fitness is computed. When the fitness of the child is better than that of the worst member of the present population, the child replaces the worst member.

Over the iterations, each individual P is assigned a value denoted *LastGoodChild(P)*. LastGoodChild(P) is initially set to 0 and is reset to 0 whenever P is a worse parent and a child of P is fitter than both of its parents. LastGoodChild(P) is incremented by 1 whenever P is a worse parent and its child is not fitter than the better parent. If LastGoodChild(P)=3, then P is deleted from the population and replaced by an immigrant (a new individual). This process is used to retain the individuals that may not have a good fitness but may have elements of "goodness" that can be transmitted to the progeny. On the other hand, it removes individuals of lesser fitness that also do not contribute good elements.

Immigration is the method by which we introduce new individuals into the population. In immigration, we create a new individual and compute its fitness. The fitness of the new individual is compared with that of the worst member of the population. If the new individual has a better fitness, it replaces the worst member.

By using crossover and immigration, the population evolves. The change in the population may be due to the introduction of a child or immigrant or the removal of a worse parent or the removal of worst individual in the population.

The successive populations are ranked according to the fitness of their individual members and they undergo the same processes (parental selection, crossover and immigration) until a population satisfies our stopping rule. Our *stopping rule* is based on the following two criteria:

- We set a maximum number of iterations (10000 for our implementations).
- We also set a minimum value for the standard deviation of the fitnesses of the population (average fitness of the present population/1000).

If the minimum standard deviation is obtained, we say that the final population has converged. When the maximum number of iterations is reached, we say that the algorithm was "forced to stop".

3.3 Computational Implementation

We implemented the genetic algorithm in C++ on a SPARC station 10. We used data types and algorithms from LEDA (Library of Efficient Data types and Algorithms) in several places in our implementation and found this library very useful. We implemented the FORTRAN code for dual ascent and heuristic procedure on the same machine. The size of our test problems ranged from 10 nodes, 25 arcs and 90 commodities to 35 nodes, 175 arcs and 1190 commodities. We tested 250 problem instances in all. We solved the same problem instances using genetic algorithm and using dual ascent and drop-add heuristic.

The test problems are defined over random undirected networks with complete demand patterns (a commodity for each node pair). Our network generation method is described later in this section. The nodes of the network are points plotted in

a two dimensional grid. We can have one of two types of node coordinates which we call *uniform node coordinates* and *normal node coordinates.* The arc costs are proportional to Euclidean distances between end nodes of the arcs.

We selected 6 problem types whose sizes are listed in Table 3.1. We consider the routing cost of an arc as the Euclidean distance between the end nodes. The fixed cost is some user specified multiple of routing cost. We refer to this multiple as the *cost ratio.* Our implementations solve test problems for cost ratios of 1, 5, 10, 20 & 50.

For uniform node coordinates, our network generator first selects the required number of nodes n (specified by the user) with node coordinates as random integer pairs chosen from a 100X100 grid. For normal node coordinates, it selects $\lfloor |N|/5 \rfloor$ (which is the closest integer smaller than $|N|/5$) node coordinates that are random integer pairs from a 80X80 grid. Other node coordinates are chosen so that they are normally distributed (see Rubinstein [28]) with mean as one of the selected node coordinates.

The generator, first, forms a random cycle using a randomly selected ordered set of nodes. Then, for a user specified degree d, the generator adds arcs to the network so that the approximate degree of each node is the user specified degree. The total number of arcs m in the network is approximately $nd/2$. The arcs are added such that the arcs connect closer node pairs when possible. Then, the Euclidean distance between the end node pairs and the cost ratio are used to compute the routing and design cost for each arc. The generator also generates commodity origins and destinations where the total number of commodities is also user specified. After forming a random cycle, the addition of arcs is done as follows (degrees updated as and when arcs are added):

1. Form a priority queue of all node pairs (with their relative distances).

2. Choose the node pair (ij) with the smallest distance.

   If the degree of both i and j is less than d and they are not already connected

by an arc, then add arc (i, j) to the network and delete (ij) from the priority queue
Else delete (ij) from the priority queue.

3. Continue (2) until the node-pair priority queue is empty or the required number of arcs are already added.

4. When the node-pair priority queue is empty and more arcs are required for the network, then find a node l with degree less than d. Form a priority queue of nodes of the network (with their distance from l). Choose the node closest to l (say k);
if (l, k) is not already in the network, add this arc to the network and delete k from the node priority queue
else delete k from the node priority queue.

5. Continue (4) until degree of l becomes d or the node priority queue becomes empty or the required number of arcs are added to the network.

6. Perform (4) and (5) for all nodes whose degree is less than d.

3.3.1 Genetic algorithms implementation

The algorithm starts by generating an *initial population*. Number of individuals in the population is user specified. The individuals are formed in one of the following three ways:

1. With 10% probability, the individual is a minimum spanning tree of the original network.

2. With 10% probability $n - 1$ least design cost arcs are chosen from the original network for adding in the individual, then enough arcs are added to ensure connectivity in the individual.

Connectivity correction is done as follows:

A node i is chosen at random.

A breadth-first search is performed from i using arcs of the individual. Nodes that constitute the search tree are referred to as connected nodes, other nodes are called unconnected nodes.

For each unconnected node k, find the closest connected node l.

Add arc (k, l) in the individual.

3. For the remaining 80% cases the following procedure is adopted:

   A node is chosen at random and a spanning tree is formed with the arcs of the original network (not using cost information) using the breadth first search technique.The spanning tree arcs are then added to the individual.

   Then, with a 20% probability a random integer $num - ind - arcs$ is generated between 0 and $m - n + 1$.

   In the remaining 80% cases, $num - ind - arcs$ is a random integer between 0 and $2n + 1$.

   Then, each of the $num - ind - arcs$ arcs are chosen from the original network as follows: The arcs (of the original network) are ranked from 1 to $|A|$ according to their design cost (least design cost arc ranked 1). A random number *choose* is generated between 1 and $|A|(|A| + 1)/2$. Whenever $i(i - 1)/2 < choose \leq i(i + 1)/2$ for (i=1 to $|A|$), then the arc ranked $|A| - i + 1$ is chosen for the individual (provided it is not already in the individual) .

The reason for developing this method for generating the initial population and immigration is that (1) and (2) contribute more to the *fitness* of the solutions, while (3) contributes more to the *diversity* of the population. Our initial experimentations started with more focus on randomness and the whole population was generated using (3). The results obtained had values whose gap percentage (*=100(GA solution-DA lower bound)/DA lower bound*) was higher (than this method) by a factor of 5 to 10 for problems with cost ratio 1. As the cost ratio was increased to 2, the gap

percentage increased by about 50% over the cost ratio 1 gap%.

The *fitness computation* for an individual is started by solving shortest path distances for each commodity (only using arcs in the individual). The shortest path distances thus computed are added to the aggregated design costs of all arcs in the individual. The aggregation of design costs and the shortest distances gives the fitness of the individual.

The population then undergoes iterations of processes called parental selection, crossover and immigration. The methods were explained in detail in section 3.2. The population is updated for rank and fitness after each iteration.

Every 25 iterations, the implementation provides quantities such as best fitness value, average fitness value of the whole population, average fitness of the best ten in the population, average fitness of the best half of the population, standard deviation of the fitness values and the computational time spent for the previous 25 iterations.

We tested different stopping rules using standard deviation during our test runs such as fixing a constant lower bound, a function of population size as the lower bound, and a function of average fitness of the population as a lower bound. We found the average fitness criteria appropriate in experience. The quantity given by $averagefitness/1000$ is set as a lower bound on the standard deviation of the fitness values. If the lower bound is achieved, the genetic algorithm terminates.

After running some test problems, we selected a maximum of 10000 iterations as an upper bound for the number of iterations of the algorithm.

On termination of the algorithm, the implementation gives the following additional quantities:

1. the percentage of feasible children, the percentage of worse parent replacements and the percentage of immigrations effected.

   The sum of the above three quantities gives an expression for the additional fitness computations performed after the fitness computation for the initial population.

2. the total computational time spent excluding network generation.

3. the number of shortest path computations and the fitness computations performed during the whole run.

4. the design of the best individual of the final population and the number of arcs present in this individual.

3.3.2 Dual-ascent implementation

For the dual ascent (with drop-add heuristic) implementation, we use the FORTRAN code written by Balakrishnan, Magnanti and Wong [3]. They used this code on a IBM 3083 computer. With the help of Izumi Sakuta, we made the alterations for using this code on our Sparc station 10. The program prompts the user to specify a parameter NMAX that influences the number of dual ascent cycles that are performed. By a dual ascent cycle they [3] mean a complete execution of the dual-ascent routine (until all commodity origins are labeled), heuristic procedure, and problem reduction test. As the average number of cycles needed ranged from 2 to 4 according to Balakrishnan, Magnanti and Wong [3], we decided to fix the value for NMAX to 5.

The outputs from the dual-ascent implementation includes the following information:

1. a lower bound for the optimal solution (this is generated by cycles of dual-ascent).

2. an upper bound for the optimal solution (this is produced by improving the dual-ascent feasible solutions using the drop-add heuristic).

3. the computational time required for each step and the total computational time.

Table 3.1: Test Problems

Problem Type	No. of nodes	No. of arcs	No. of commodities
1	10	25	90
2	15	60	210
3	20	100	380
4	25	125	600
5	30	150	870
6	35	175	1190

3.4 Computational Results

The problem parameter setting, we tested, were of 6 types. The network dimensions of the different problem types are listed in Table 3.1.

The test outputs are *summarized* in the Tables (3.2 to 3.11). The outputs are presented separately for uniform and normal node coordinates and for different cost ratios. Tables 3.2 to 3.6 consider networks with uniform node coordinates and they are sequentially presented for cost ratios 1, 5, 10, 20 and 50. Tables 3.7 to 3.11 consider networks with normal node coordinates and they are sequentially presented for cost ratios 1, 5, 10, 20 and 50.

For an instance I, let $Z_{GA}(I)$ denote the value of the genetic algorithm's solution. Let $Z_{DA}(I)$ denote the value of the dual ascent lower bound and let $Z_{DC}(I)$ denote the value of the dual ascent cycle solution (applying drop-add heuristic and problem reduction method to the dual ascent feasible solution).

Each table has the following quantities:

1. Problem Type (see Table 3.1).

2. GA ave. gap $(=100(Z_{GA}\text{-}Z_{DA}/Z_{DA}))$ is the % gap of genetic algorithm solution from the dual-ascent lower bound.

3. DA ave. gap $(=100(Z_{DC}\text{-}Z_{DA}/Z_{DA}))$ is the % gap of dual-ascent upper bound

from the dual-ascent lower bound.

4. GA ave. comp. time is the average computational time spent for solving the problems using our genetic algorithm implementation.

5. DA ave. comp. time is the average computational time spent for solving the problems using dual ascent with drop-add heuristic.

6. Ave. No. GA iterations is the average number of iterations it took for GA until termination.

7. Ave. DA asc. cycles is the average number of ascent cycles it took for dual-ascent to reach the corresponding upper bounds and lower bounds.

The averages are computed over 3 to 5 (in most cases 5) problem instances in each case. The different quantities measure the performance of the methods in two dimensions. The *quality* of the solutions is measured by the gap%. The gap% measures the difference from the lower bound. The computational time (CPU time spent for the corresponding method) measures the *speed* of the solution methods.

The performance is *analyzed* using Figures 3-1 to 3-32. Figures 3-1 to 3-12 analyze the percentage gaps observed. Figures 3-13 to 3-24 analyze the computational time spent. Figures 3-25 to 3-30 plot the computational times as a function of shortest path computations performed. Figures 3-31 and 3-32 study the diversity of the initial population.

In the remainder of this section, we use GA and genetic algorithms interchangeably. Similarly, we use DA and dual ascent with drop-add heuristics interchangeably.

Summary of Results

Uniform Node Coordinates

Tables 3-2 to 3-6 give respectively for cost ratios 1,5,10,20 and 50, the results obtained for uniform node coordinate problems in problem types 1 to 6. The average gap for

Table 3.2: Uniform Node Coordinates-Cost Ratio 1

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	.578	.054	45.448	.248	1425	2.2
2	.338	.031	114.44	1.51	2020	3
3	1.146	.025	244.02	6.124	2585	4
4	1.814	.053	272.68	15.592	2055	4.4
5	1.542	.05	429.56	28.868	2295	4.6
6	1.5233	.017	529.17	66.51	2108.33	5

Table 3.3: Uniform Node Coordinates-Cost Ratio 5

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	0.8112	0.477	30.054	0.474	955	3.4
2	1.212	0.583	119.26	3.926	2170	5.6
3	2.03	1.056	222.07	11.624	2420	4.8
4	1.786	0.746	391.55	22.56	3055	4.6
5	3.638	0.468	573.16	50.644	3180	5
6	3.4667	0.433	823.22	69.79	3391.67	4.33

GA ranges from .5 to 9 percentage and the average computational time ranges from 14 to 825 seconds. For DA, the average gap percentage ranges from .7 to 3.9 and the average computational time ranges from .20 to 211 seconds.

Normal Node Coordinates

Tables 3-7 to 3-11 give respectively for cost ratios 1,5,10,20 and 50, the results obtained for normal node coordinate problems in problem types 1 to 6. The average gap for GA ranges from .3 to 5 percentage and the average computational time ranges from 13 to 813 seconds. For DA, the average gap percentage ranges from 0 to 1.5 and the average computational time ranges from .20 to 215 seconds.

Table 3.4: Uniform Node Coordinates-Cost Ratio 10

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	1.368	0.73	18.356	0.402	575	2.6
2	3.886	1.744	77.318	4.18	1420	4.6
3	3.19	1.503	199.34	9.7833	2250	3.67
4	4.5333	2.793	144.35	17.617	1675	4.33
5	2.714	1.41	641.25	64.722	3640	4.6
6	3.9533	1.447	913.24	133.41	3858.33	5

Table 3.5: Uniform Node Coordinates-Cost Ratio 20

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	2.55	0.29	14.42	0.55	450	3
2	4.5975	2.175	68.633	3.7125	1262.5	3.5
3	4.1067	2.793	144.35	17.617	1675	4.33
4	4.8567	2.357	235.02	40.107	1858.33	5
5	3.798	2.344	404.17	76.538	2340	4.6
6	3.3	1.81	642.2	210.52	2766.67	5

Table 3.6: Uniform Node Coordinates-Cost Ratio 50

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	2.3382	1.316	18.356	0.412	585	2
2	4.416	2.97	34.336	4.578	635	3.4
3	9.07	3.5	119.58	20.25	1375	4
4	6.3133	3.843	125.79	51.35	1041.67	5

Table 3.7: Normal Node Coordinates-Cost Ratio 1

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	0.518	0	43.03	0.218	1345	2
2	0.77	0.009	117.06	1.74	2070	3.4
3	0.97	0.069	234.98	8.186	2515	4.2
4	1.738	0.036	270.3	16.292	2020	4.8
5	1.522	0.054	378.59	33.76	2030	4.8
6	1.5667	0.009	454.21	64.607	1800	5

Table 3.8: Normal Node Coordinates-Cost Ratio 5

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	0.319	0.269	32.232	0.386	1030	2.8
2	1.524	0.602	77.856	4.104	1455	4.8
3	1.352	0.414	172.93	12.01	1955	4.4
4	2.16	0.724	379.62	22.488	2975	4.6
5	2.2775	0.38	479.18	55.398	2700	5
6	1.9767	0.173	716.14	82.613	3025	5

Table 3.9: Normal Node Coordinates-Cost Ratio 10

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	0.926	0.28	24.228	0.506	775	3
2	1.862	1.198	73.808	3.976	1390	4
3	1.72	0.87	182.41	16.283	2066.67	4.67
4	2.89	1.02	264.88	35.907	2141.67	4.67
5	2.3433	0.7	412.13	76.797	2400	5
6	3.6933	0.827	812.61	167.91	3541.67	5

Table 3.10: Normal Node Coordinates-Cost Ratio 20

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	0.94	0.11	14.162	0.476	440	2.6
2	3.135	0.933	53.17	4.6075	987.5	4.75
3	4.6467	1.427	121.28	21.997	1425	5
4	2.9433	1.15	182.78	35.187	1450	3.67
5	2.9567	0.997	373.71	49.153	2191.67	3.33
6	3.82	1.49	573.7	214.88	2583.33	5.33

Table 3.11: Normal Node Coordinates-Cost Ratio 50

Problem Type	Ave. gap GA	Ave. gap DA	Ave.CPU time GA	Ave.CPU time DA	Ave. No. GA iterations	Ave. DA ascent cycles
1	1.092	0.222	13.672	0.54	425	2.6
2	2.734	1.45	33.206	5.448	610	3.6
3	4.59	1.38	46.375	16.5	487.5	4

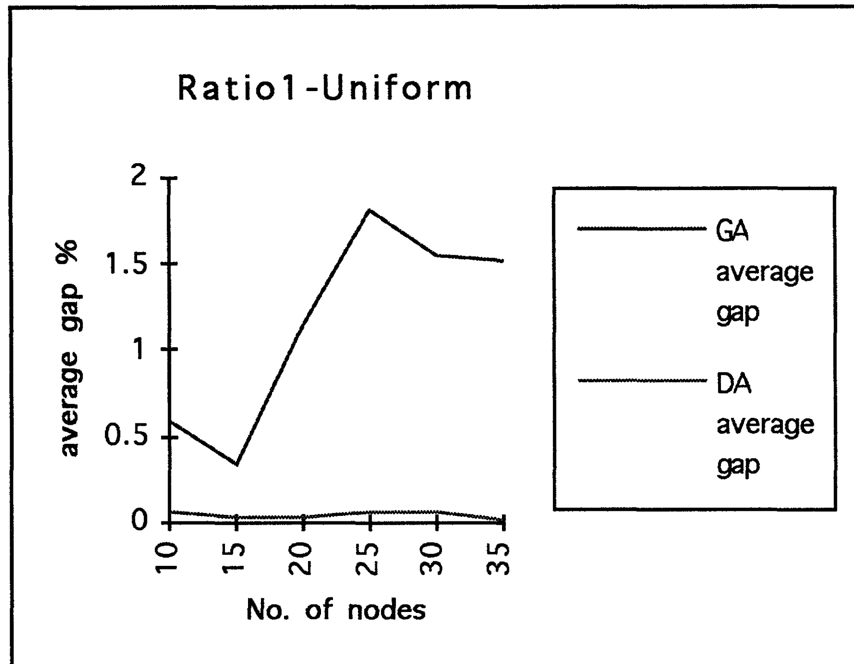

Figure 3-1: percentage gaps for Uniform, Ratio 1 problems

3.4.1 Average gap percentage

Observations

Figures 3-1 to 3-5 and 3-8 to 3-12 analyze the average gap% observed for both GA and DA for different cost ratios in uniform and normal node coordinate problems respectively. For uniform node coordinate problems, figures 3-6 and 3-7 plot respectively for GA and DA, the gap% observed in 5 different ratios (5 curves) for the problem sizes considered.

A rise in gap% is observed for most cases in both methods as the problem size increases. We can find some problem types performing better than smaller sized problems. In most of these problems, we see that both GA and DA do well.

For very low and very high cost-ratios, DA seems to perform much better than GA. But, for intermediate ratios (5,10 and 20) GA presents a comparable performance.

Smaller gaps are observed for normal node coordinate problems in both methods.

Uniform Node Coordinates

Figure 3-1 plots the average gap% observed for both GA and DA for cost-ratio 1 problems in the 6 problem types. The DA average gap% is very close to 0 in all cases. The GA also performs well with the average gap% lying below 2 for all problems.

Figure 3-2 plots the average gap% observed for both GA and DA for cost-ratio 5 problems in the 6 problem types. The DA average gap% is less than 1 in almost all cases. The GA performs with the average gap% less than 4 for all problem types.

Figure 3-3 plots the average gap% observed for both GA and DA for cost-ratio 10 problems in the 6 problem types. The DA average gap% is less than 3 and GA performs with the average gap% below 4 for most problem types.

Figure 3-4 plots the average gap% observed for both GA and DA for cost-ratio 20 problems in the 6 problem types. The DA average gap% is less than 3 and GA performs with the average gap% below 5 for all problem types.

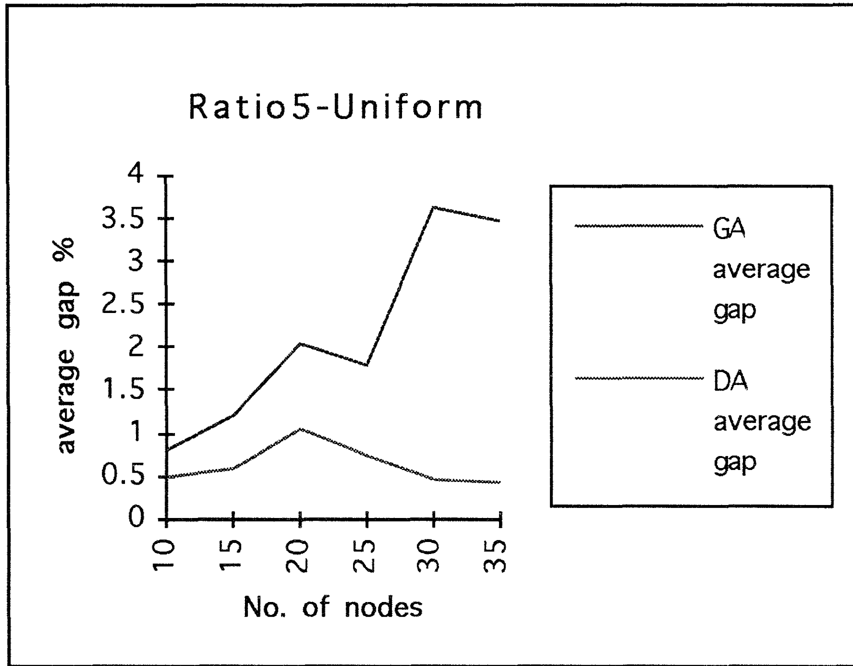

Figure 3-2: percentage gaps for Uniform, Ratio 5 problems

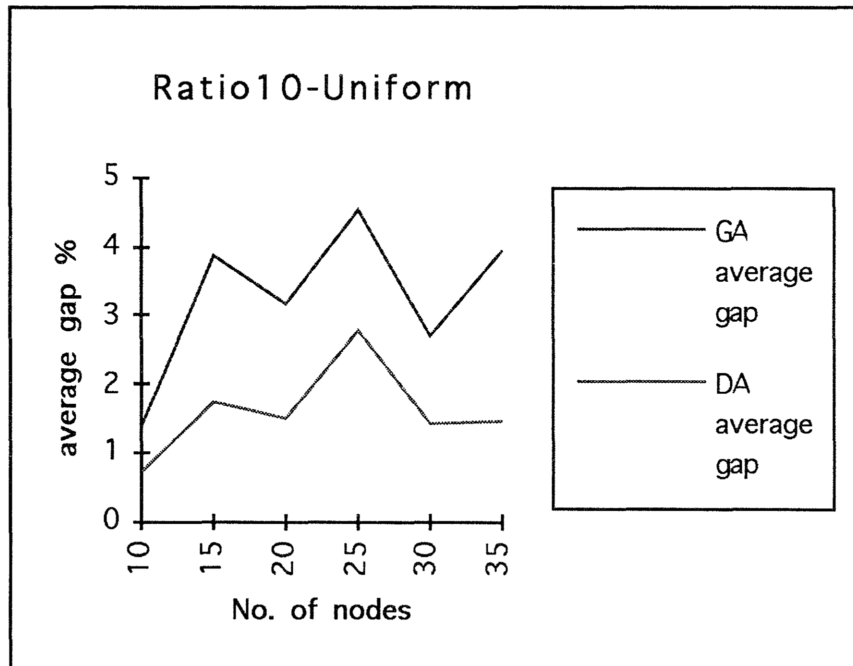

Figure 3-3: percentage gaps for Uniform, Ratio 10 problems

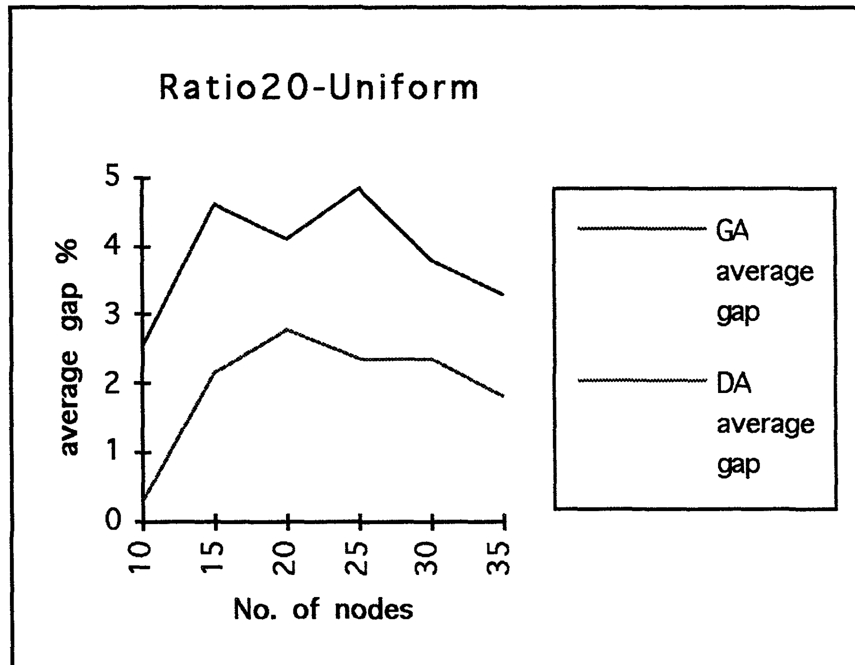

Figure 3-4: percentage gaps for Uniform, Ratio 20 problems

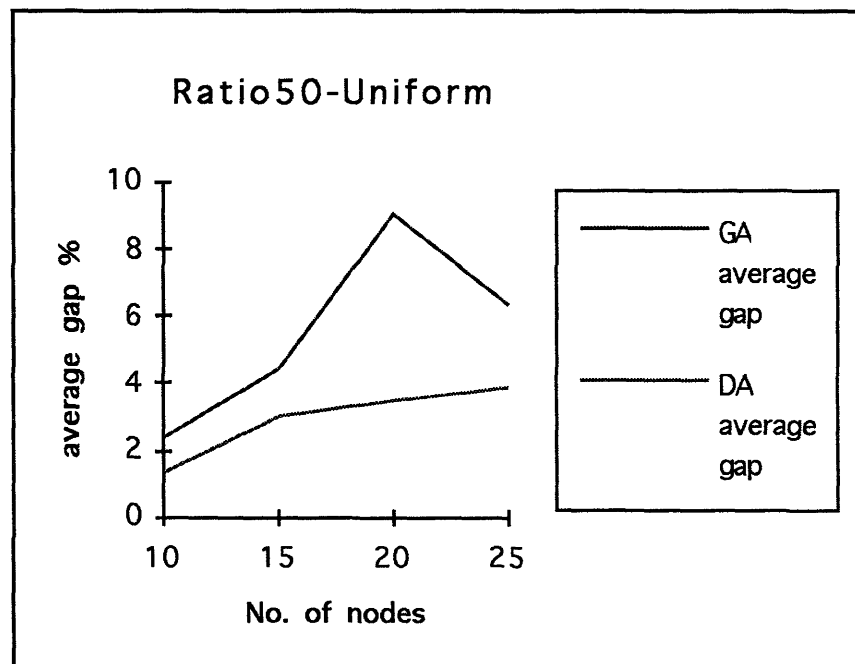

Figure 3-5: percentage gaps for Uniform, Ratio 50 problems

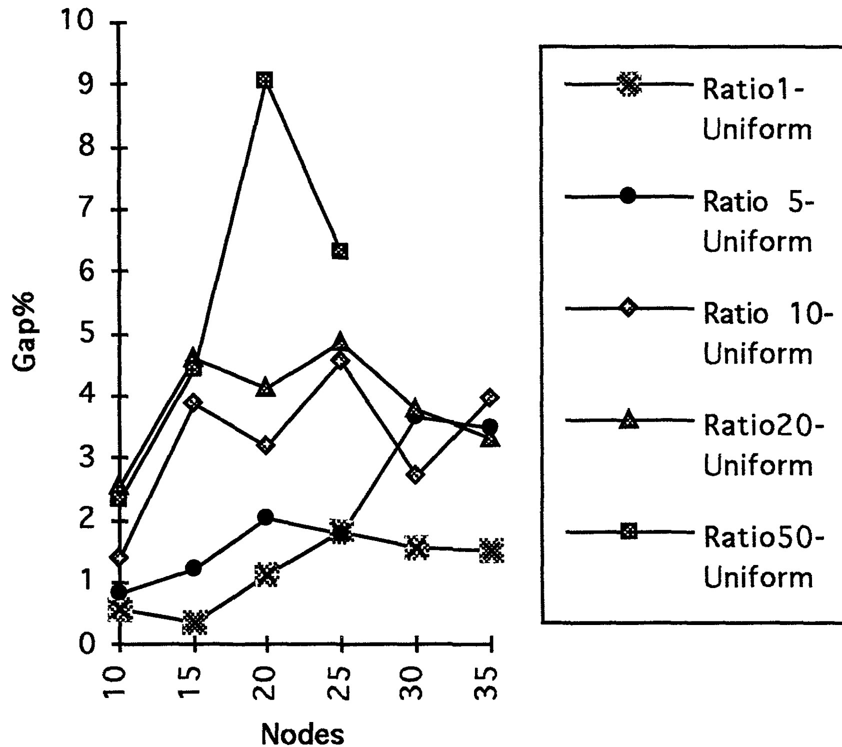

Figure 3-6: GA percentage gaps for uniform node problem

Figure 3-5 plots the average gap% observed for both GA and DA for cost-ratio 50 problems in the first 4 problem types. The DA average gap% is less than 3 and GA performs with the average gap% less than 9 for the different problem types.

Normal Node Coordinates

Figure 3-8 plots the average gap% observed for both GA and DA for cost-ratio 1 problems in the 6 problem types. The DA average gap% is very close to 0 in all cases and GA performs with the average gap% lying below 2 for all problems.

Figure 3-9 plots the average gap% observed for both GA and DA for cost-ratio 5 problems in the 6 problem types. The DA average gap% is less than 1 in all cases. GA performs with the average gap% below 2.5 for all problem types.

Figure 3-10 plots the average gap% observed for both GA and DA for cost-ratio 10 problems in the 6 problem types. The DA average gap% is less than 2 and GA

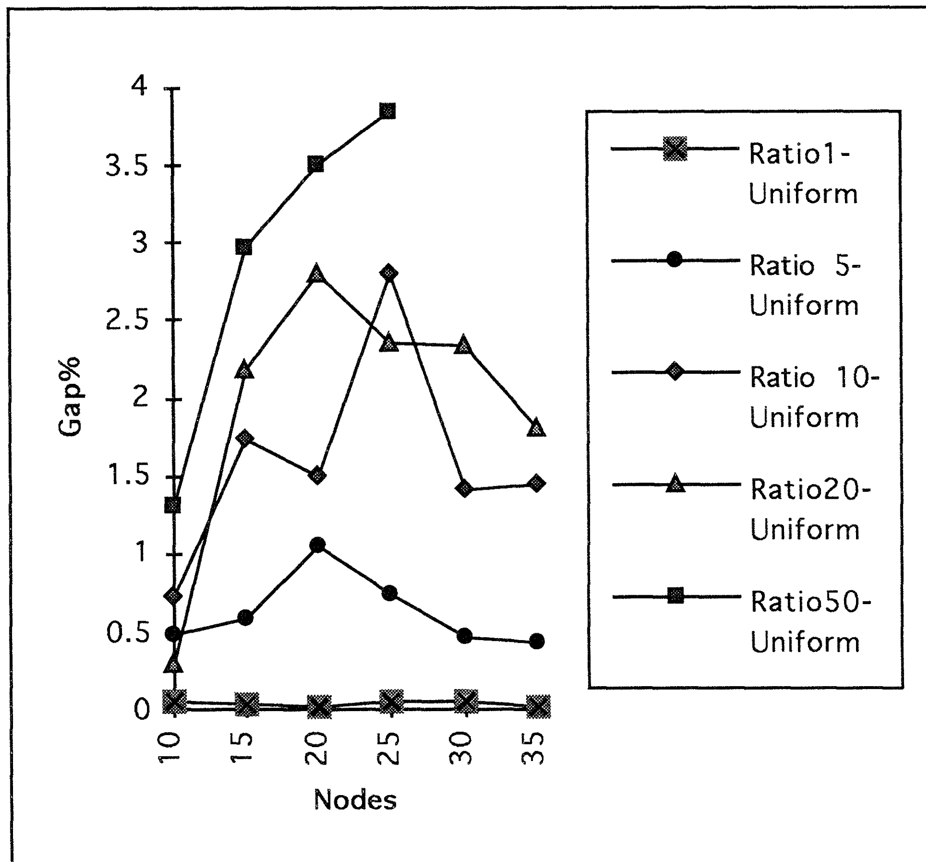

Figure 3-7: DA percentage gaps for uniform node problem

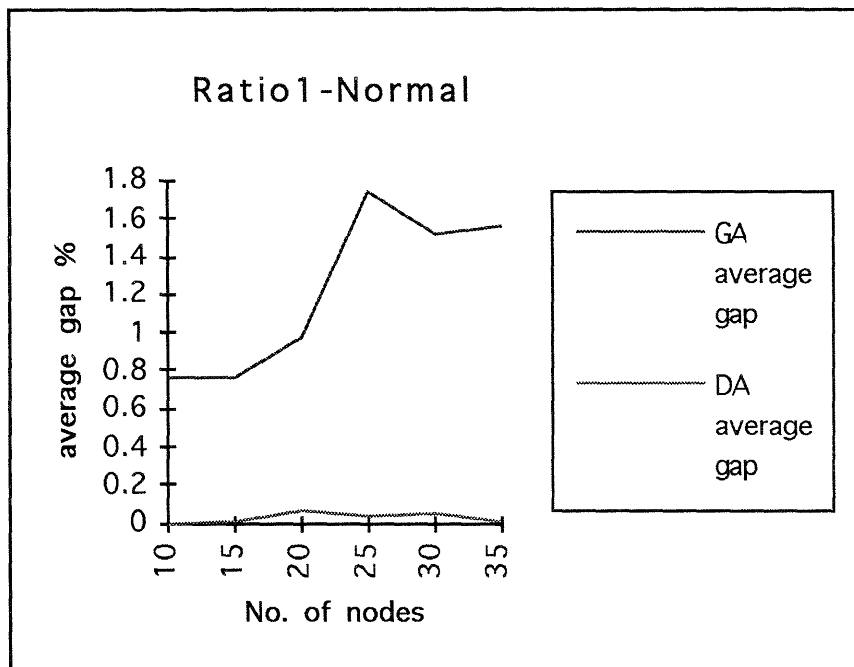

Figure 3-8: percentage gaps for Normal, Ratio 1 problems

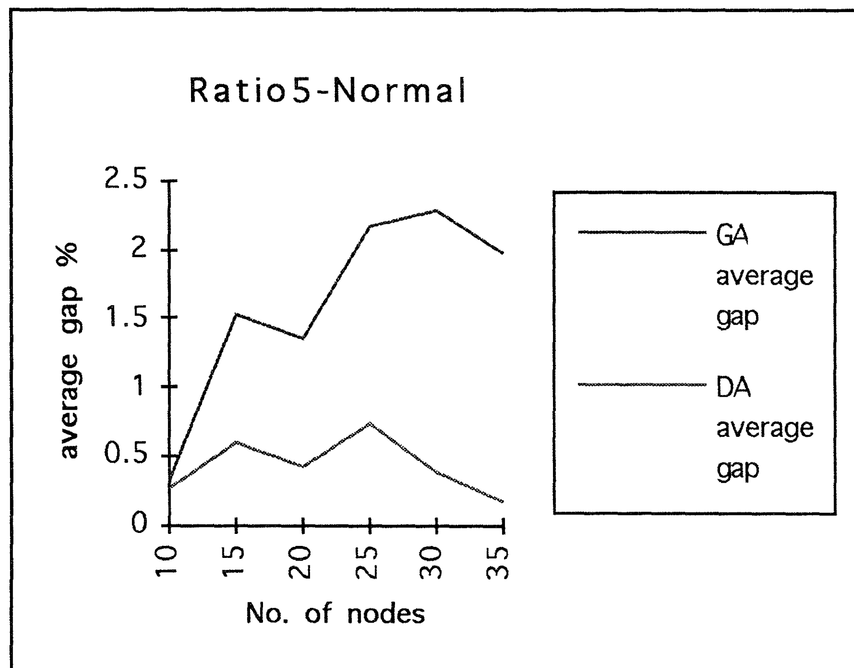

Figure 3-9: percentage gaps for Normal, Ratio 5 problems

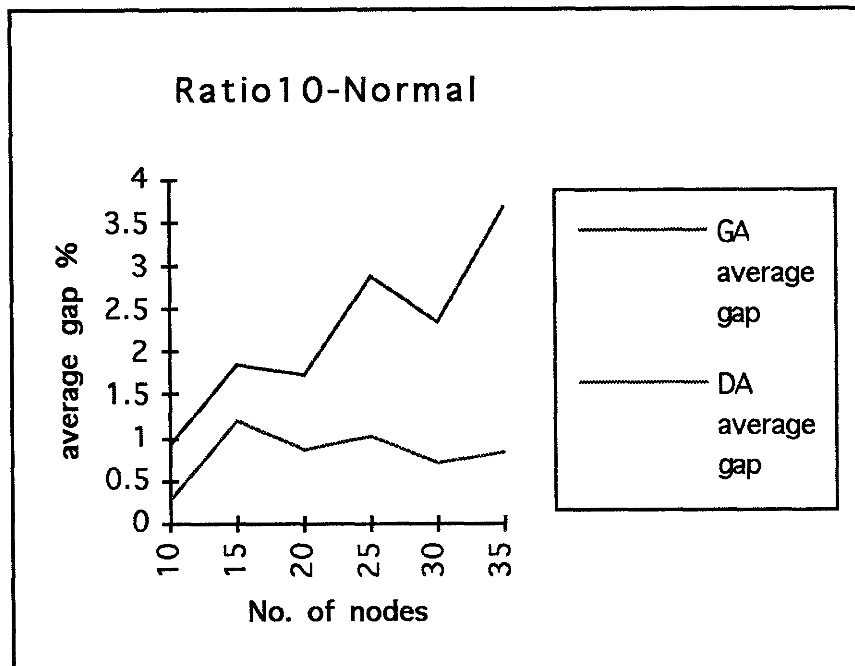

Figure 3-10: percentage gaps for Normal, Ratio 10 problems

performs with the average gap% less than 4 for most problem types.

Figure 3-11 plots the average gap % observed for both GA and DA for cost-ratio 20 problems in the 6 problem types. The DA average gap % is less than 2 and GA performs with the average gap% below 5 for all problem types.

Figure 3-12 plots the average gap% observed for both GA and DA for cost-ratio 50 problems in the first 3 problem types. The DA average gap% is less than 2 and GA performs with the average gap% below 5 for most problem types.

3.4.2 Computational time

Observations

Figures 3-13 to 3-17 and 3-18 to 3-22 analyze the average computational time spent for both GA and DA for different cost ratios in uniform and normal node coordinate problems respectively.

A steady rise in time is observed for both uniform and normal node coordinate

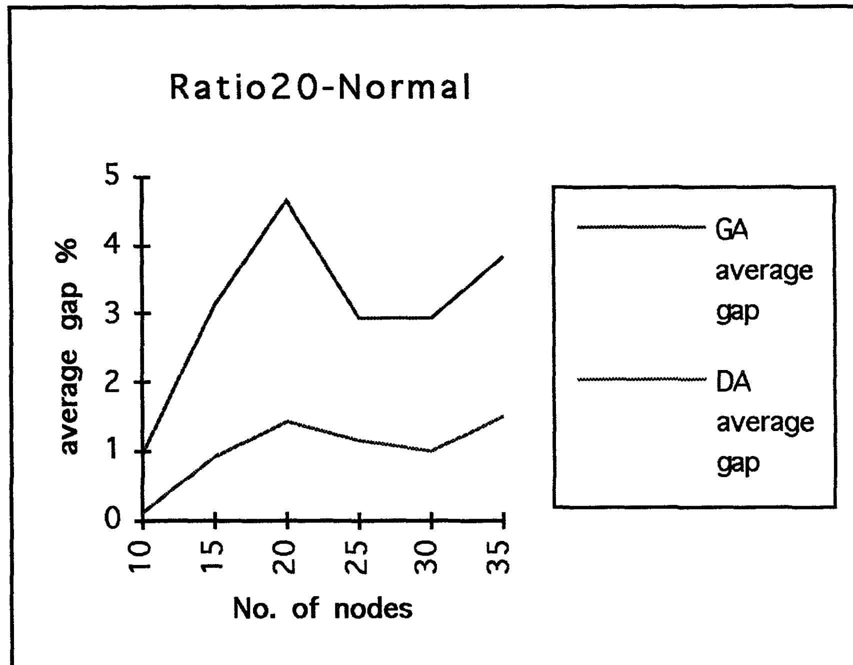

Figure 3-11: percentage gaps for Normal, Ratio 20 problems

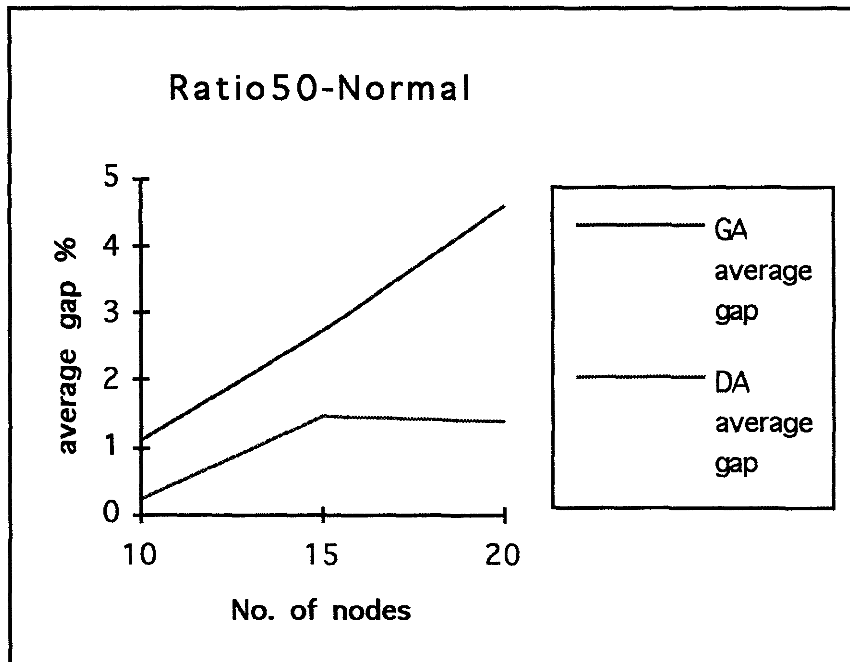

Figure 3-12: percentage gaps for Normal, Ratio 50 problems

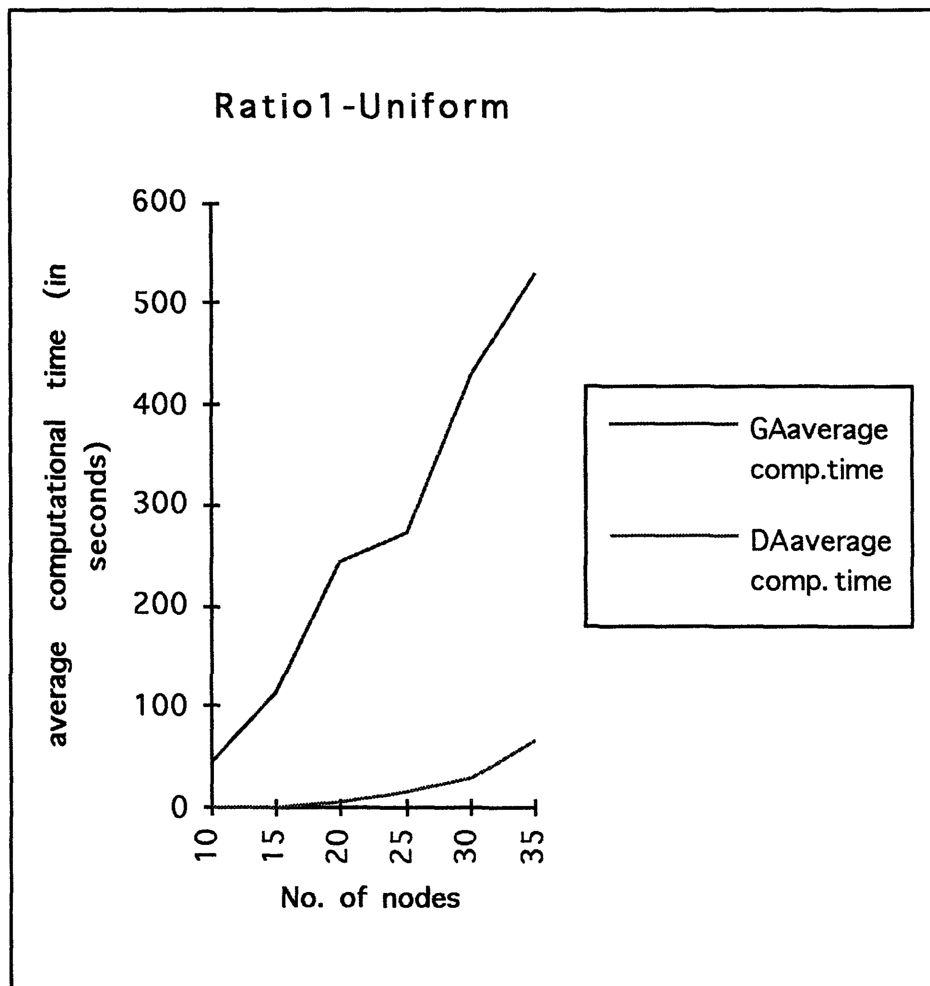

Figure 3-13: computational time for Uniform, Ratio 1 problems

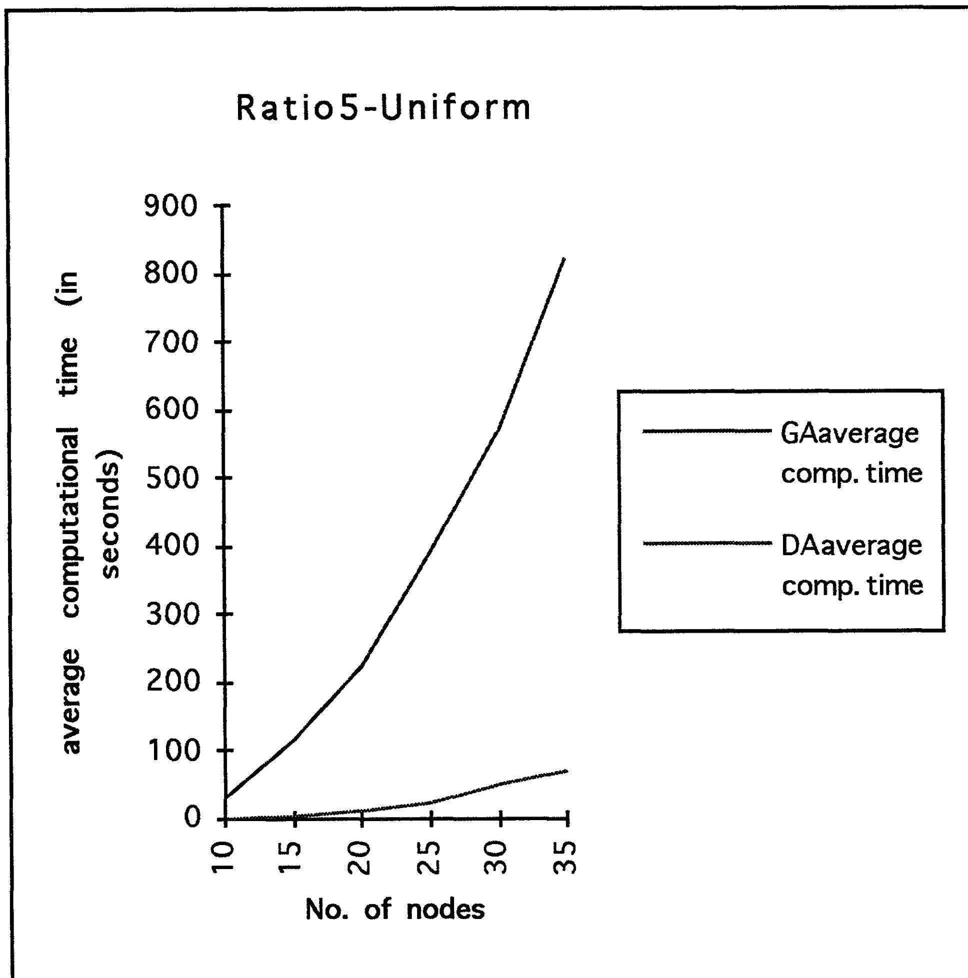

Figure 3-14: computational time for Uniform, Ratio 5 problems

problems as the problem size increases. The computational time spent for solving normal node coordinate problems is comparitively lesser than that on uniform node coordinates. Ratio 1 and 5 problems seem to be particularly easier for DA.

Uniform Node Coordinates

Figure 3-13 plots the computational time spent for both GA and DA for cost-ratio 1 problems in the 6 problem types. The DA average computational time is less than 100 seconds whereas the GA average computation time lies below 600 seconds for all problems.

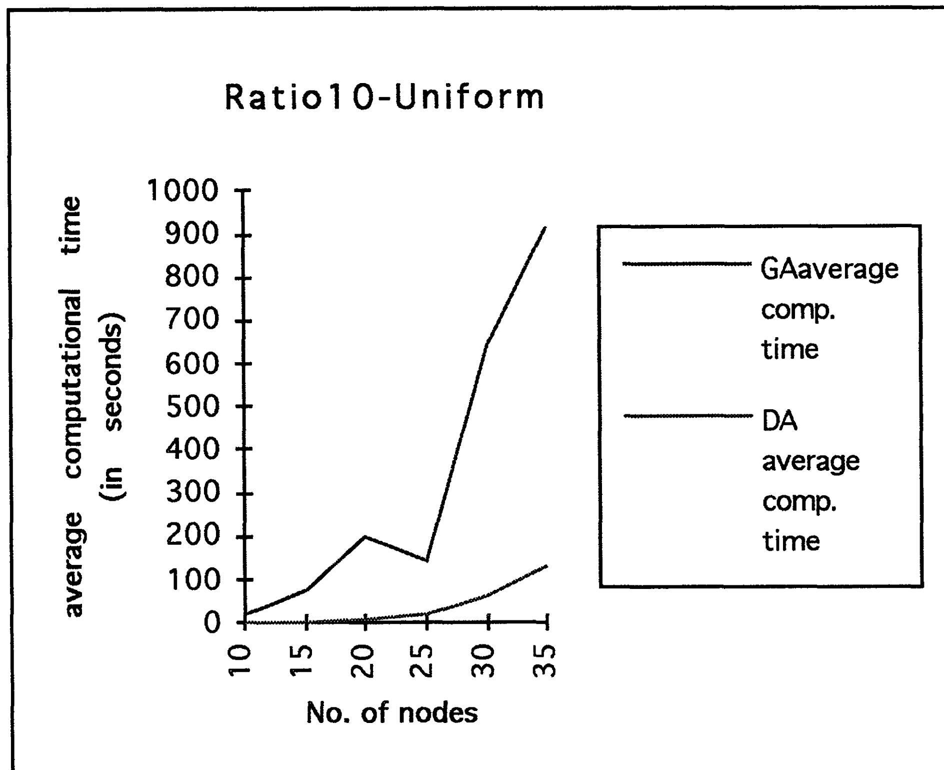

Figure 3-15: computational time for Uniform, Ratio 10 problems

Figure 3-14 plots the computational time spent for both GA and DA for cost-ratio 5 problems in the 6 problem types. The DA average computational time is less than 100 seconds whereas the GA average computation time lies below 900 seconds for all problems.

Figure 3-15 plots the computational time spent for both GA and DA for cost-ratio 10 problems in the 6 problem types. The DA average computational time is less than 200 seconds whereas the GA average computation time lies below 900 seconds in all problem types.

Figure 3-16 plots the computational time spent for both GA and DA for cost-ratio 20 problems in the 6 problem types. The DA average computational time is less than 300 seconds whereas the GA average computation time lies below 700 seconds for all problems.

Figure 3-17 plots the computational time spent for both GA and DA for cost-ratio

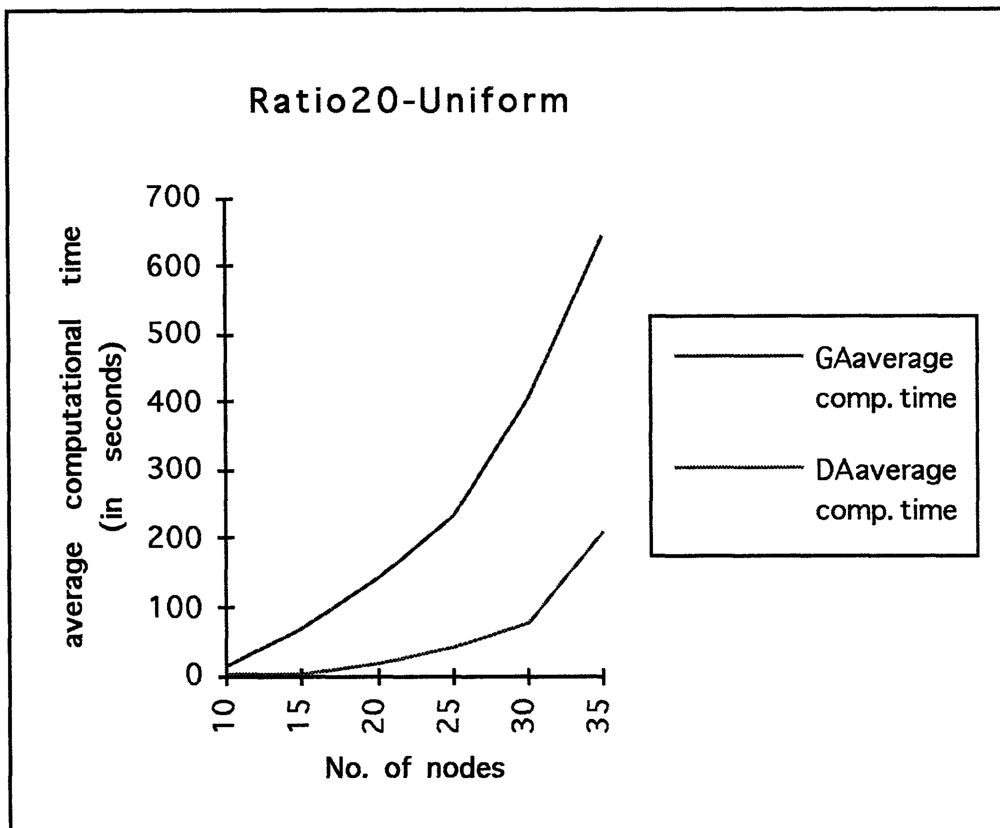

Figure 3-16: computational time for Uniform, Ratio 20 problems

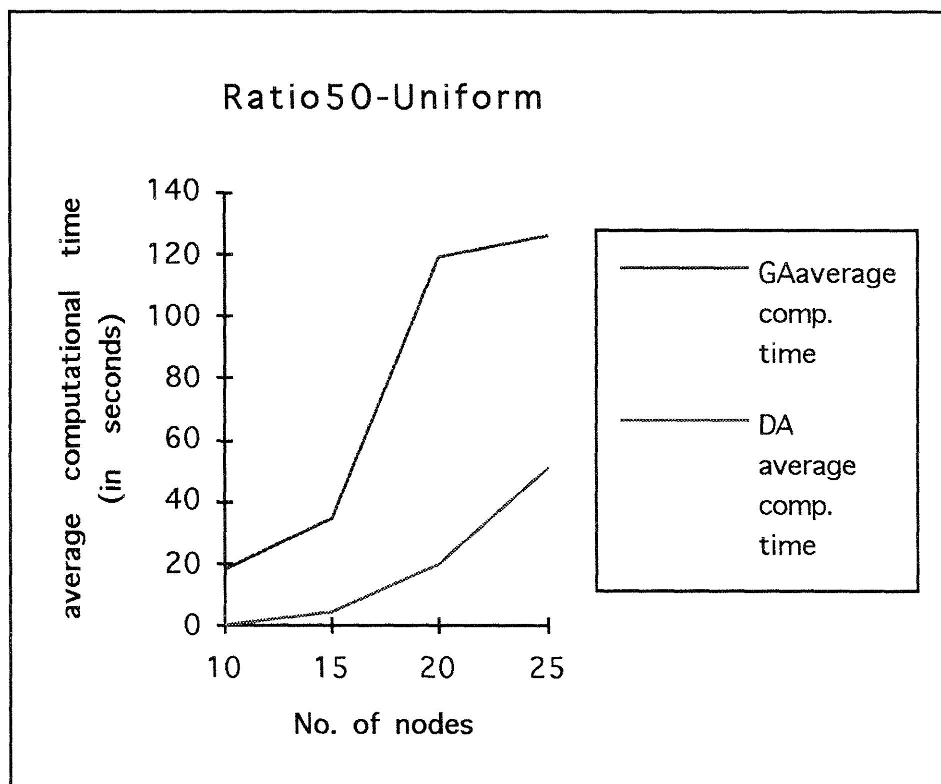

Figure 3-17: computational time for Uniform, Ratio 50 problems

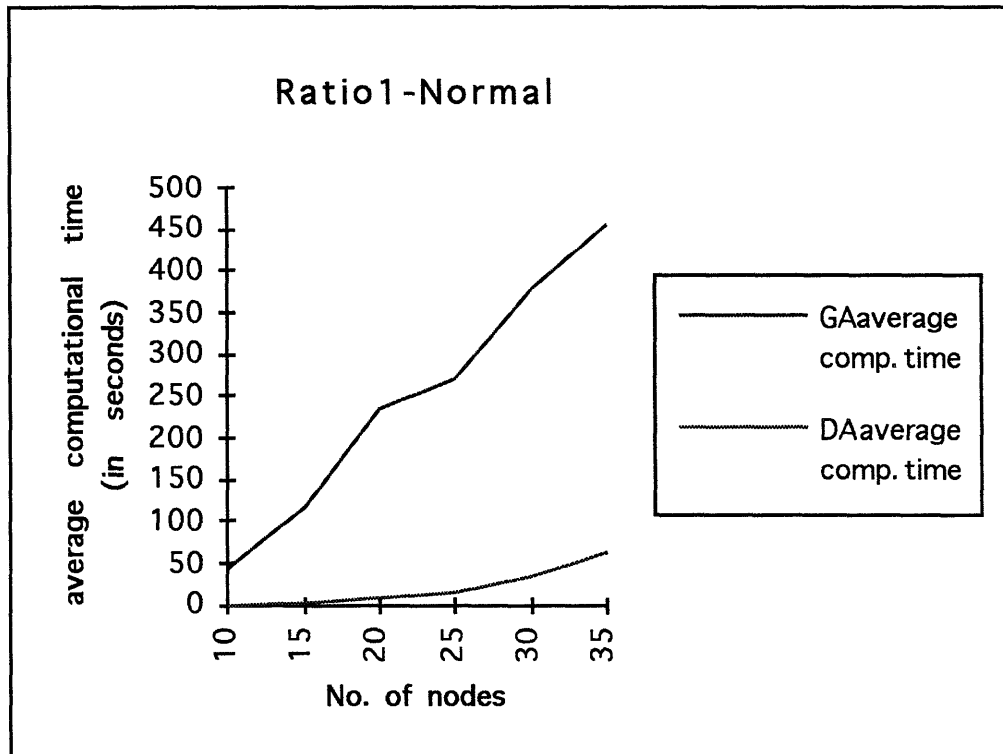

Figure 3-18: computational time for Normal, Ratio 1 problems

50 problems in the first 4 problem types. The DA average computational time is less than 60 seconds while the GA average computation time lies below 140 seconds for all problem types.

Normal Node Coordinates

Figure 3-18 plots the computational time spent for both GA and DA for cost-ratio 1 problems in the 6 problem types. The DA average computational time is less than 100 seconds in all cases while the GA average computation time lies below 450 seconds for all problem types.

Figure 3-19 plots the computational time spent for both GA and DA for cost-ratio 5 problems in the 6 problem types. The DA average computational time is less than 100 seconds whereas the GA average computation time lies below 800 seconds for all problems.

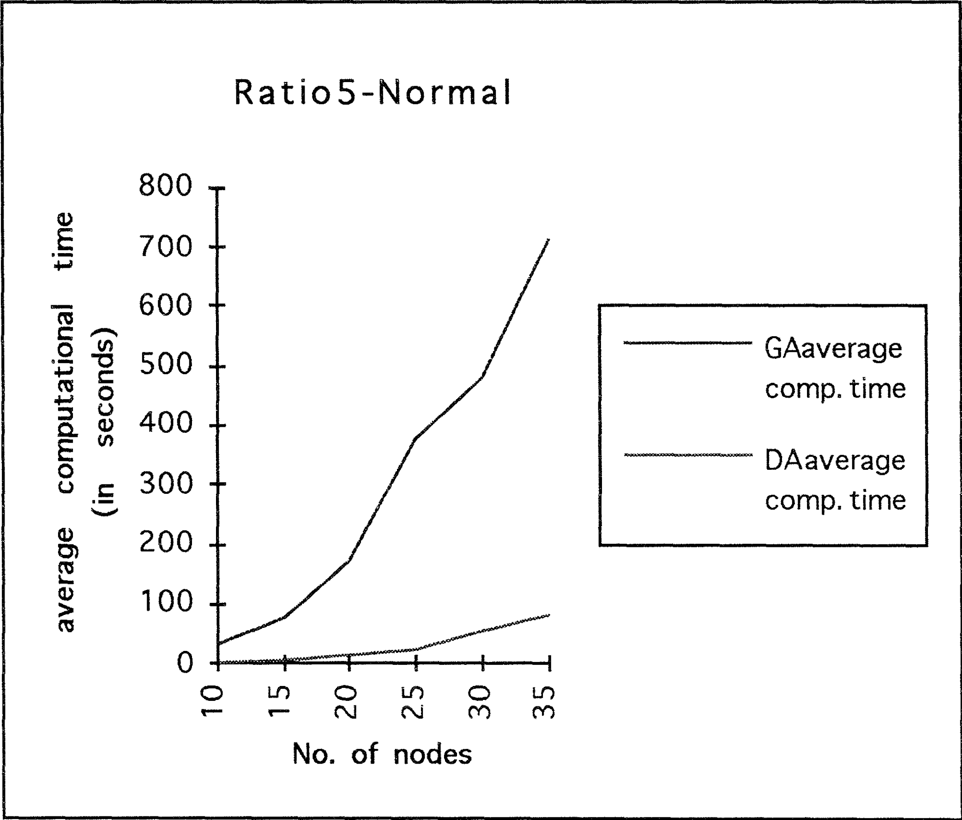

Figure 3-19: computational time for Normal, Ratio 5 problems

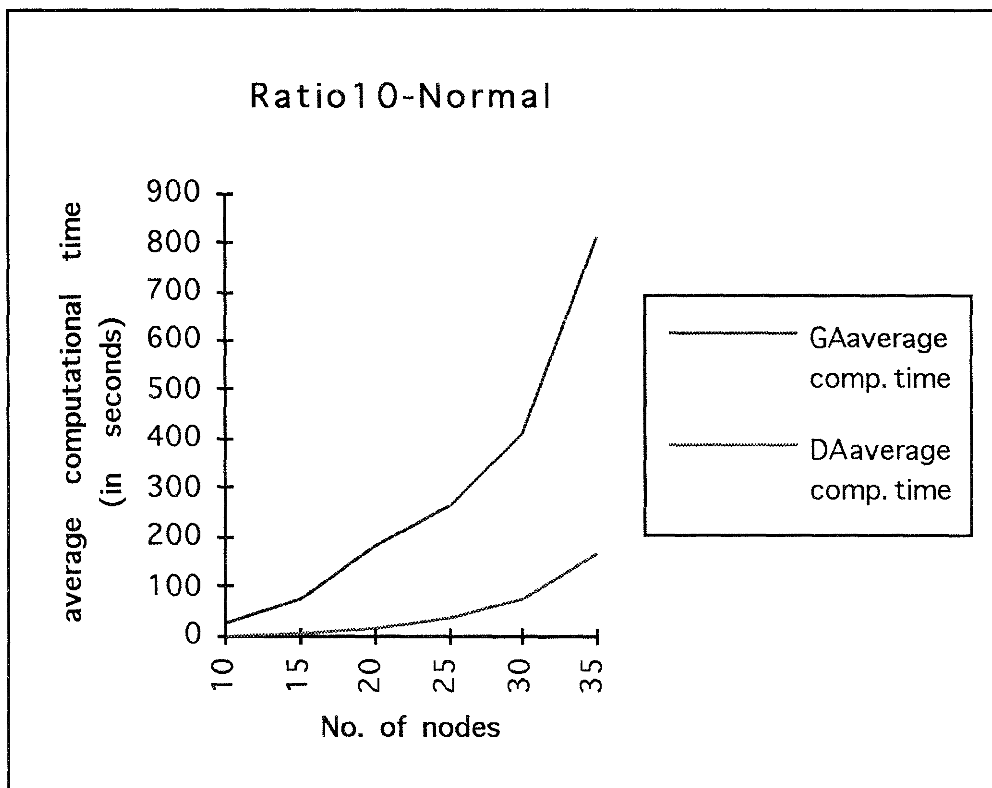

Figure 3-20: computational time for Normal, Ratio 10 problems

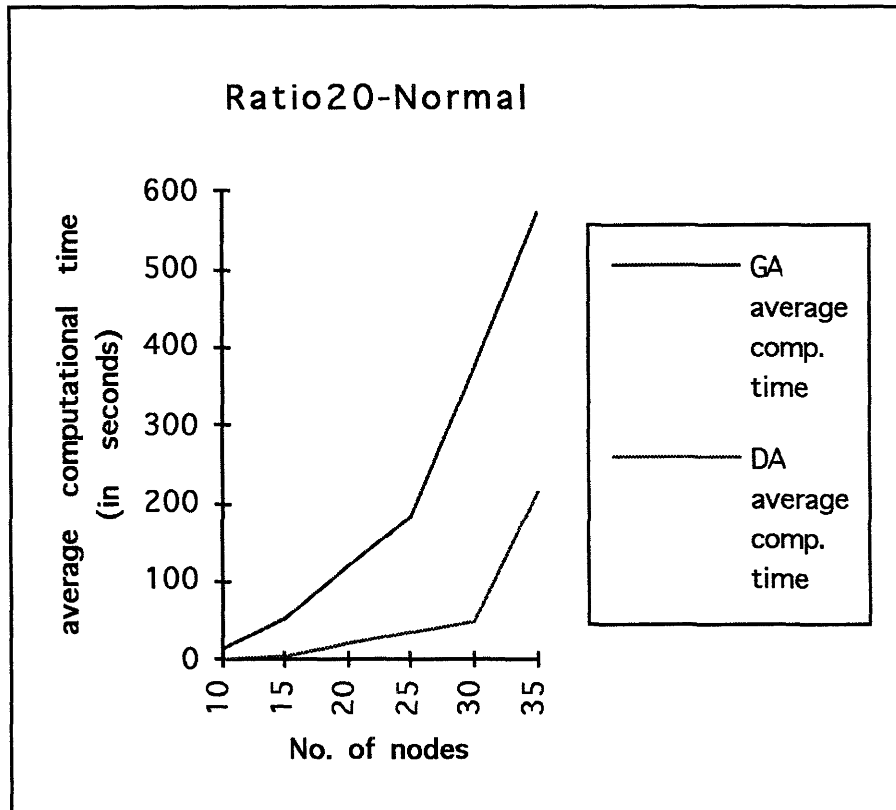

Figure 3-21: computational time for Normal, Ratio 20 problems

Figure 3-20 plots the computational time spent for both GA and DA for cost-ratio 10 problems in the 6 problem types. The DA average computational time is less than 200 seconds whereas the GA average computation time lies below 900 seconds for all problems.

Figure 3-21 plots the computational time spent for both GA and DA for cost-ratio 20 problems in the 6 problem types. The DA average computational time is less than 300 seconds whereas the GA average computation time is less than 600 seconds for all problem types.

Figure 3-22 plots the computational time spent for both GA and DA for cost-ratio 50 problems in the first 4 problem types. The DA average computational time is less than 20 seconds whereas the GA average computation time lies below 50 seconds for all problems.

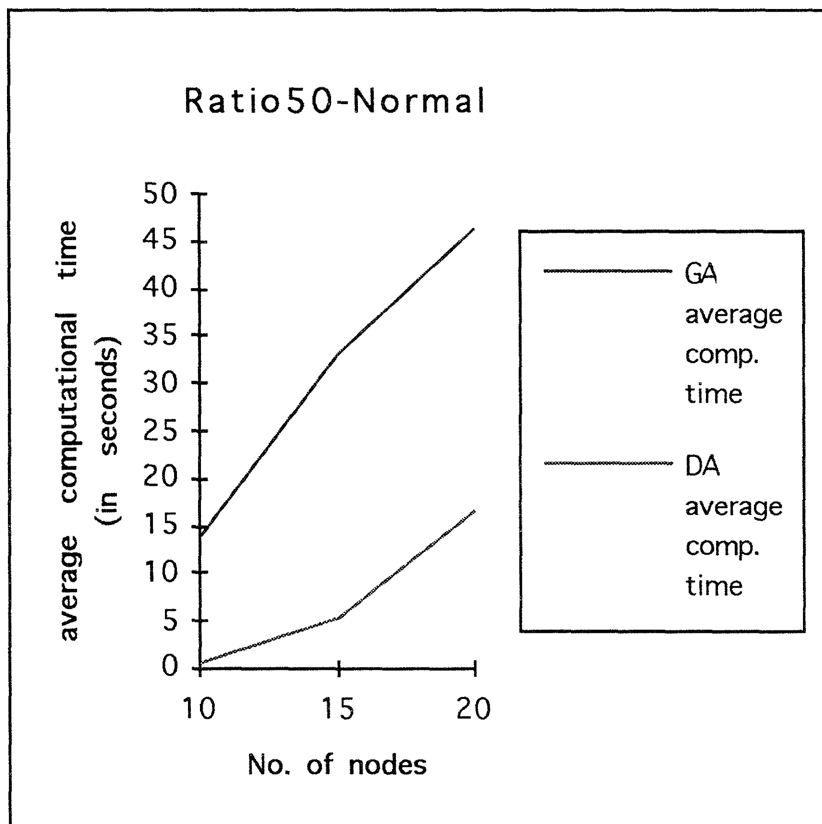

Figure 3-22: computational time for Normal, Ratio 50 problems

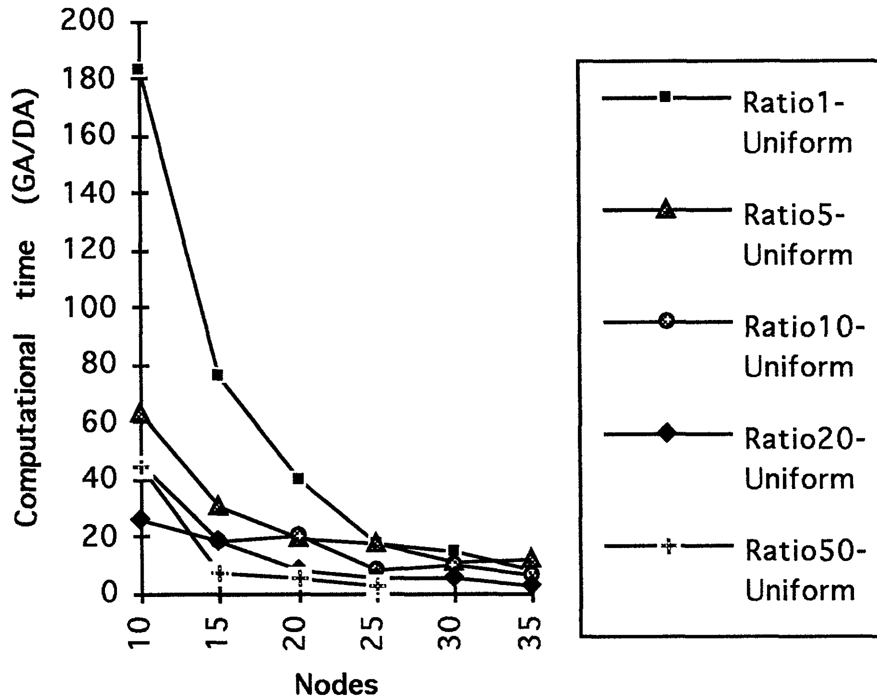

Figure 3-23: Computational time ratio (GA/DA) for different cost ratio, uniform problems

3.4.3 Ratio of computational times (GA/DA)

Figures 3-23 and 3-24 plot respetively for uniform and normal node coordinate problems, the ratio of computational time (GA/DA) observed for differnt cost ratios (5 curves).

Observations

For both uniform and normal node coordinate problems, there is a decline in the ratio curve as the problem size increases. This observation would mean that the proportional time taken by GA (compared to DA) to solve larger problems is observed to be comparatively lesser for both uniform and normal node coordinate problems (i.e., GA computational time gets relatively better with increasing problem size.)

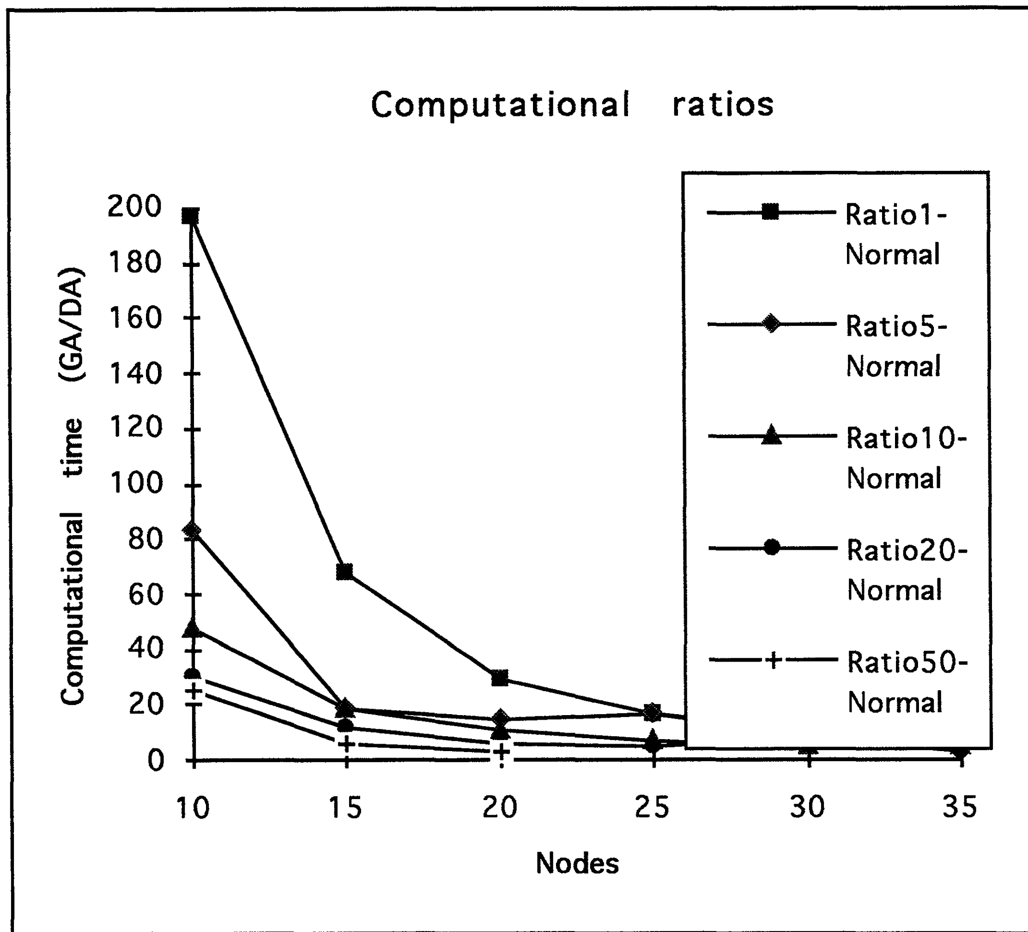

Figure 3-24: Computational time ratio (GA/DA) for different cost ratio, normal problems

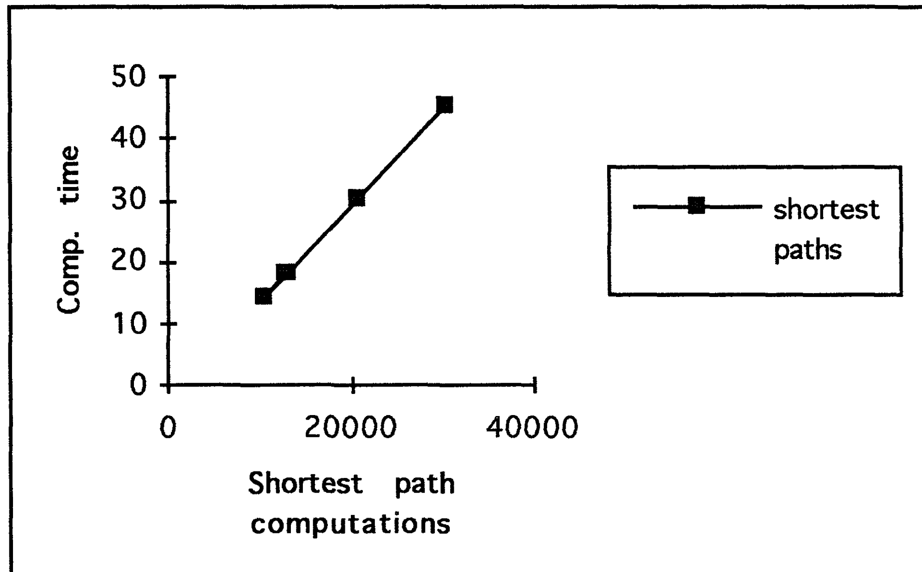

Figure 3-25: shortest path computations-computational time for problem type 1

3.4.4 Relation of Computational times to shortest path computations

The data points plotted (in Figures 3-25 to 3-30, respectively for problem types 1 to 6) are the average computational times observed for each cost-ratio (not particularly in that order) with the corresponding shortest path computations.

Observations

Figures 3-25 to 3-30 support a close linear relation between the computation time spent and the shortest path computations performed. We performed similar analysis for normal node coordinate problems and observed similar results.

We also performed an analysis of variation of computation time to the total number of individuals created in the problem for both uniform and normal node coordinate problems. We observed similar linear (almost) relationship in these cases.

This provides a way to approximately estimate the computational time for a problem when one of the quantities, number of individuals created or shortest path com-

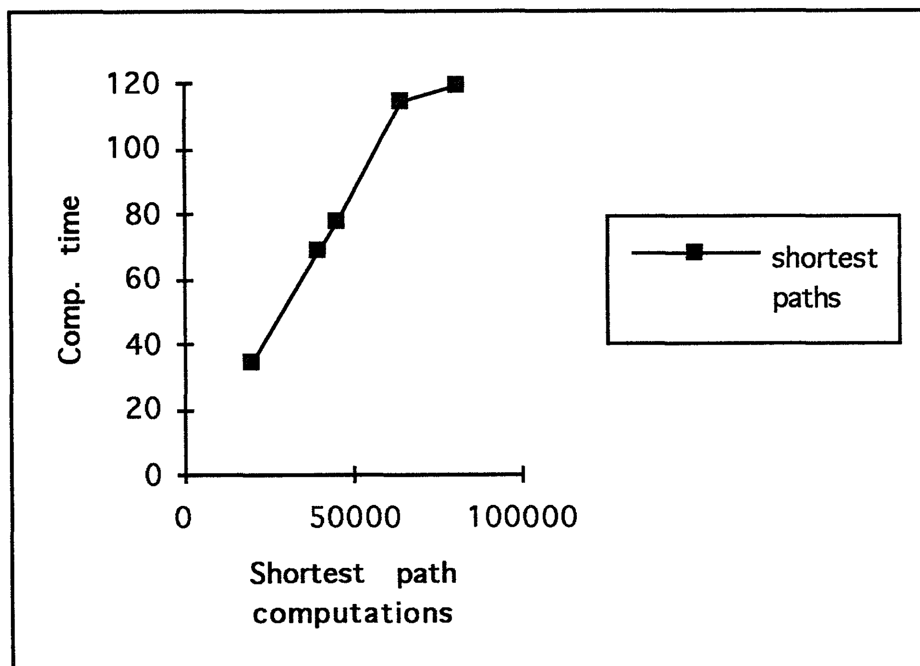

Figure 3-26: shortest path computations-computational time for problem type 2

putations performed, can be estimated.

Figure 3-25 plots for problem type 1 and uniform node coordinates, the variation of computational time with respect to the number of shortest path computations performed. A linear regression of the data points gives an R^2 value of .999.

Figure 3-26 plots for problem type 2 and uniform node coordinates, the variation of computational time with respect to the number of shortest path computations performed. A linear regression of the data points gives an R^2 value of .95.

Figure 3-27 plots for problem type 3 and uniform node coordinates, the variation of computational time with respect to the number of shortest path computations performed. A linear regression performed for the data points gives an R^2 value of .996.

Figure 3-28 plots for problem type 4 and uniform node coordinates, the variation of computational time with respect to the number of shortest path computations performed. A linear regression performed for the data points gives an R^2 value of

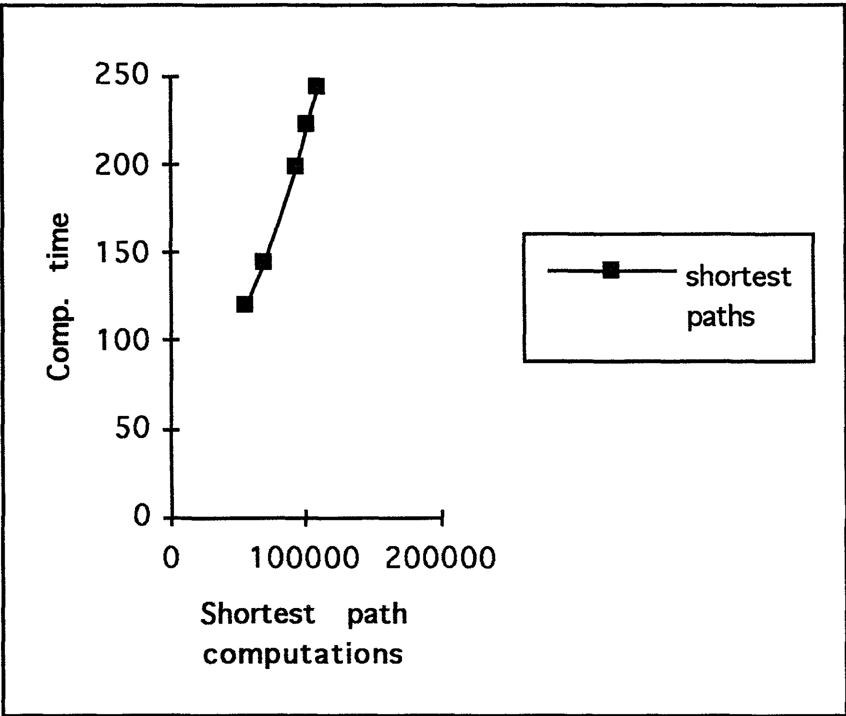

Figure 3-27: shortest path computations-computational time for problem type 3

.992.

Figure 3-29 plots for problem type 5 and uniform node coordinates, the variation of computational time with respect to the number of shortest path computations performed. A linear regression performed for the data points gives an R^2 value of .992.

Figure 3-30 plots for problem type 6 and uniform node coordinates, the variation of computational time with respect to the number of shortest path computations performed. A linear regression performed for the data points gives an R^2 value of .993.

Spread of the population

In a genetic algorithm, diversity of the initial population is needed to ensure proper exploration of the search space. Our implementation measures the standard deviation

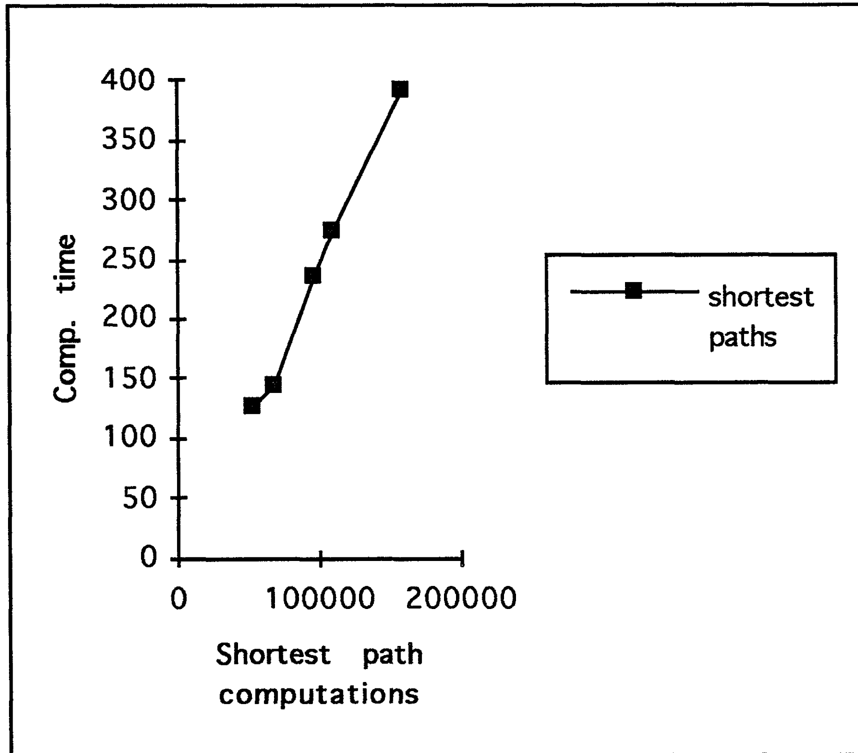

Figure 3-28: shortest path computations-computational time for problem type 4

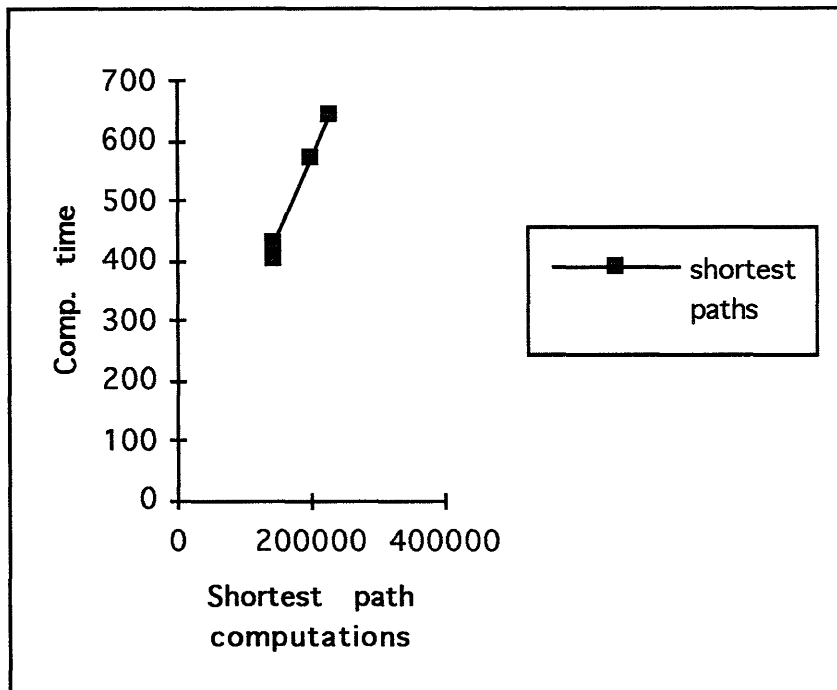

Figure 3-29: shortest path computations-computational time for problem type 5

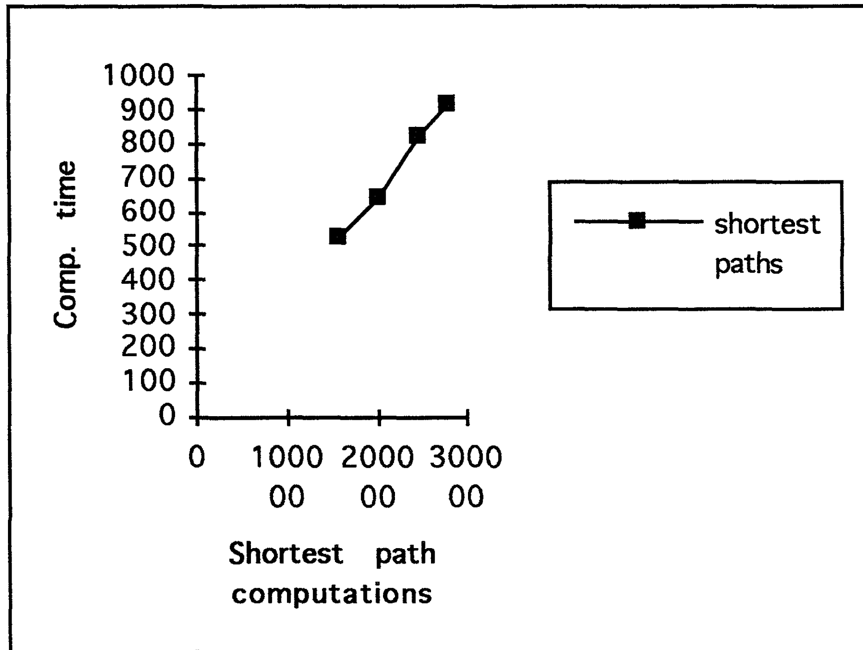

Figure 3-30: shortest path computations-computational time for problem type 6

of the population fitness of the initial population as a measure of diversity. We compare, in the following 2 histograms, this ratio of standard deviation to the average fitness of the initial population to get a measure of the population's diversity.

Observations

The histograms support that the standard deviation of the initial population fitness varies between 10 to 25% of the average fitness. This gives a measure of the spread of the initial population.

Cost ratio 1

Figure 3-31 is a histogram using 12 data points. The data points are the ratio of the standard deviation to the average fitness of the initial population of 12 cost-ratio 1 problems selected at random.

The data points, here, belong to 3 disjoint sets namely

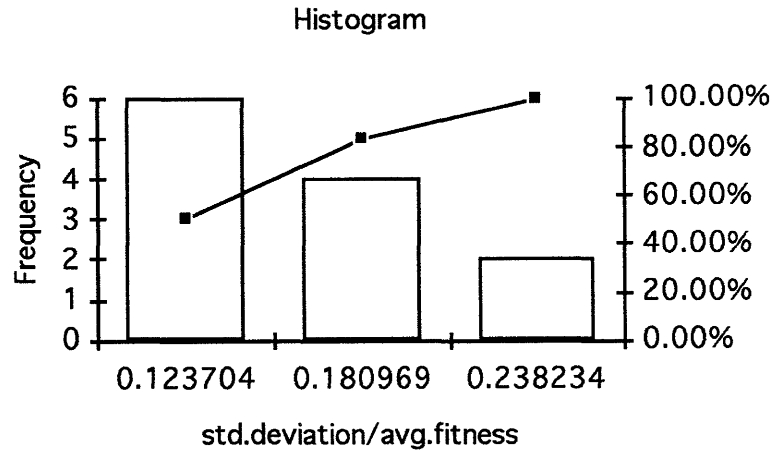

Figure 3-31: Cost ratio 1, ratio of standard deviation to average fitness-12 data points selected at random

1. above .123704 (lowest observation) and below .180969

2. above .180969 and below .238234

3. above .238234 and within .295499376 (highest observation)

In (3), both lower bound and the upper bound are included in the set. In (1) and (2), the lower bound is included but the upper bound is not included in the set.

Cost ratio 50

Figure 3-32 draws a histogram using 12 data points. The data points are the ratio of the standard deviation to the average fitness of the initial population of 12 cost-ratio 50 problems selected at random.

The data points, here, belong to 3 disjoint sets namely

1. above .143383 (lowest observation) and below .179647

2. above .179647 and below .215911

3. above .215911 and within .252175009 (highest observation)

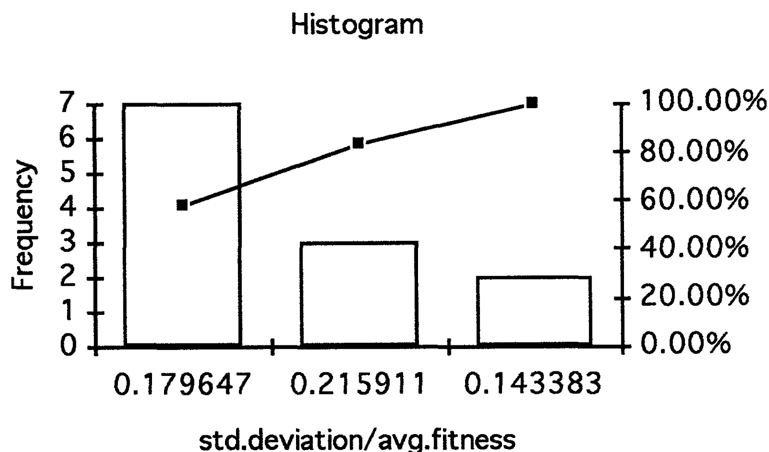

Figure 3-32: Cost ratio 50, ratio of standard deviation to average fitness-12 data points selected at random

In (3), both lower bound and the upper bound are included in the set. In (1) and (2), the lower bound is included but the upper bound is not included in the set.

3.5 Conclusions

In this work, we addressed the problem of finding a "near optimal solution" to the Uncapacitated Network Design Problem (UNDP). Several design problems arising in the areas of telecommunication, transportation and urban planning can be modeled as an UNDP. This problem has been proved to be NP-hard even for very restrictive cases.

We propose a search method called genetic algorithms to solve this problem. As a traditional genetic algorithm doesn't include problem specific information in the solution techniques, we develop a genetic algorithm that includes such information and network properties in many of its steps. For instance, we include steps such as filling a small portion of the initial population with minimum spanning trees, using network algorithms for fitness computation and for checking feasibility of the solutions.

We experimented with alternate methods for each step in order to choose the best

working method. We performed computational testing on 250 problem instances of different sizes (upto 35 node, 175 arcs and 1190 commodities). We tested our method for 5 different design cost-routing cost ratios. The results of our method are compared to that of dual-ascent method with drop-add heuristic.

Based on our test results, we performed graphical analysis of the solution gaps observed and the computational time spent. We also analyzed the variation of the ratio of computational time as the problem size increases.

The percentage gaps (of the solution values) from the dual-ascent lower bound observed is comparable to the results obtained using dual-ascent method with drop-add heuristic. The proportional computational time (compared to dual-ascent with drop-add heuristic) spent by the genetic algorithm is observed to decrease as the problem size increases. This suggests that GAs may be useful for solving larger sized problems.

Chapter 4

Conclusions and suggestions for future research

Improvements

The effectiveness of genetic algorithms for uncapacitated network design problems can be increased by applying improvement methods to the final solution.

One way of improving the solution is by applying the drop-add heuristic to the final solution. This will help decrease the gap of the solution.

Applying alternate strategies for different steps of the algorithm may sometimes lead to considerable improvement (our attempts in altering the population generation methods brought about considerable improvement, see section 3.2).

Including more network specific and problem specific information in the genetic algorithm will improve the effectiveness of the approach.

Other problems

A major avenue for future research is applying genetic algorithms to other related problems. Genetic algorithms can be applied to uncapacitated network design problems with multiple choices and precedence relations. Another class of problems that could be attempted with genetic algorithms is capacitated network design problems.

Bibliography

[1] ANDERSON,EDWARD J., AND MICHAEL C. FERRIS. 1994. Genetic Algorithms for Combinatorial Optimization: The Assembly Line Balancing Problem. *ORSA Journal on Computing.* **Vol.6, No.2,** 161-173.

[2] BACK,T., AND F.HOFFMEISTER. JULY 1991. Extended Selection Mechanisms in Genetic Algorithms. *Proceedings of the Fourth International Conference on Genetic Algorithms.* 92-99.

[3] BALAKRISHNAN,A.,T.L.MAGNANTI AND R.T.WONG.1989. A Dual-ascent procedure for Large-scale Uncapacitated Network Design. *Opns. Res.* **Vol.37, No.5,** 716-740.

[4] BEALE,E.M.L., AND J.A.TOMLIN. 1972.An Integer Programming Approach to a Class of Combinatorial Problems. *Math. Prog.* **3,** 339-344.

[5] BENDERS,J.F. 1962. Partitioning Procedures for Solving Mixed Variable Programming Problems. *Num. Math.* **4,** 238-252.

[6] BILLHEIMER,J., AND P.GRAY. 1973. Network Design with Fixed and Variable Cost Elements. *Trans. Sci.* **7,** 49-74.

[7] BOYCE,D.F.,A.FARHI AND R.WEISCHEDEL. 1973. Optimal Network Problem: A Branch-and-bound Algorithm. *Environ. Plan.* **5,** 519-533.

[8] BOYCE,D.E., AND J.L.SOBERANES. 1979. Solutions to the Optimal Network Design Problem with Shipment Related to Transportation Cost. *Trans. Res.* **13B,** 65-80.

[9] CORNUEJOLS,G., M.L.FISHER AND G.L.NEMHAUSER. 1977. Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms. *Mgmt. Sci.* **23,** 789-810.

[10] DAVIS,P.S., AND T.L.RAY. 1969. A Branch-and-Bound Algorithm for Capacitated Facilities Location Problems. *Naval Res. Logist. Quart.* **16,** 331-344.

[11] K.A.DE JONG. 1975. Analysis of the behaviour of a class of Genetic Adaptive Systems. *Ph. D. Dissertation, University of Michigan, Ann Arbor, MI.*

[12] DIONNE,R., AND M.FLORIAN. 1979. Exact and Approximate Algorithms for Optimal Network Design. *Networks* **9,** 37-59.

[13] FLORIAN,M., G.G.GUERIN AND G.BUSHEL. 1976. The Engine Scheduling Problem on a Railway Network. *INFOR J.* **14,** 121-128.

[14] GEOFFRION,A.M., AND G.GRAVES. 1974. Multicommodity Distribution System Design by Benders Decomposition. *Mgmt. Sci.* **5,** 822-844.

[15] HOANG,H.H. 1973. A Computational Approach to the Selection of an Optimal Network. *Mgmt. Sci.* **19,** 488-498.

[16] HOANG,H.H. 1982. Topological Optimization of Networks: A Nonlinear Mixed Integer Model Employing Generalized Benders Decomposition. *IEEE Trans. Automatic Control* **AC-27,** 164-169.

[17] J.H.HOLLAND. 1975. Adaptation in Natural and Artificial Systems. *University of Michigan Press, Ann Arbor, MI.*

[18] LEBLANC,L.J. 1975. An Algorithm for the Discrete Network Design Problem. *Trans. Sci.* **9,** 283-287.

[19] LOS,M. AND C.LARDINOIS 1980. Combinatorial Programming, Statistical Optimization and the Optimal Transportation Network Problem. *Trans. Sci.* **13B,** 33-48.

[20] JOHNSON,D.S.,J.K.LENSTRA AND H.G.RINNOOY KAN. 1978. The Complexity of the Network Design Problem. *Networks* **8,** 279-285.

[21] MAIRS, T.G., G.W.WAKEFIELD, E.L.JOHNSON AND K.SPIELBERG. 1978. On a Production Allocation and Distribution Problem. *Mgmt. Sci.* **24,** 1622-1630.

[22] MAGNANTI,T.L., P.MIREAULT AND R.T.WONG. Dec.1983. Tailoring Benders Decomposition for Network Design. *Working Paper OR 125-83, Operations Research Center, Massachussetts Institute of Technology.*

[23] MAGNANTI,T.L., AND R.T.WONG. 1981. Accelerating Benders Decomposition: Algorithmic Enhancement and Model Selection Criteria. *Opns. Res.* **29,** 464-484.

[24] MAGNANTI,T.L., AND R.T.WONG.1984. Network Design and Transportation Planning: Models and Algorithms. *Trans. Sci.* **18,** 1-55.

[25] RARDIN,R.L., AND U.CHOE. 1979. Tighter Relaxations of Fixed Charge Network Flow Problems. Technical Report No. J-79-18, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta.

[26] RICHARDSON,R. 1976. An Optimization Approach to Routing Aircraft. *Trans. Sci.* **10,** 52-71.

[27] ROTHENGATTER,W. 1979. Application of Optimal Subset Selection to Problems of Design and Scheduling on Urban Transportation Networks. *Trans. Res.* **13B,** 49-63.

[28] RUBINSTEIN,R.Y. 1981. Simulation and the Monte Carlo Method. *Wiley, New York.*

[29] SCOTT,A.J. 1969. The Optimal Network Problem: Some Computational Procedures. *Trans. Res.* **3,** 201-210.

[30] SPEARS,W.M, AND K.A.DE JONG.1991. On the virtues of Parametrized Uniform Crossover. *Proceedings of the Fourth International Conference on Genetic Algorithms.* 230-236.

[31] TAILLARD, ERIC D. 1994. Parallel Taboo Search Techniques for the Job Shop Scheduling Problem. *ORSA Journal on Computing.* **Vol.6, No.2,** 108-117.

[32] WILLIAMS,H.P. 1974. Experiments in the Formulation of Integer Programming Problems. *Math. Prog. Study* **2,** 180-197.

APPENDIX A

C++ Code

```
/* This is a genetic algorithm for the uncapacitated network design problem*/

#include <LEDA/graph_alg.h>
#include <LEDA/prio.h>
# include <LEDA/graph.h>
# include <LEDA/ugraph.h>
# include <ctype.h>
# include <LEDA/array.h>
# include <LEDA/plane.h>
# include <LEDA/node_pq.h>
# include <LEDA/point_set.h>
# include <LEDA/vector.h>
# include <LEDA/graph_edit.h>
# include <LEDA/queue.h>
# include <LEDA/node_partition.h>
# include <math.h>
# include <stdio.h>
# include <stdlib.h>
# include<fstream.h>
# include<time.h>



//associates n nodes with graph G.
void create_graph(UGRAPH<point,int>& G,
                  int n)
{int i;
for (i=0; i<n; i++)
```

```
G.new_node();}

//determines uniform node coordinates for each node of G.
void uniform_node_coordinates(UGRAPH<point,int>& G,
                              node_array<int>& x_coord,
                              node_array<int>& y_coord)

{ node v;
  forall_nodes(v,G)
  {x_coord[v]=random(1,100);
   y_coord[v]=random(1,100);
   G[v]=point(x_coord[v],y_coord[v]);
  }
}

//initializes the random number generator.
double RandomNumber(bool* First)
{if(*First)
{srandom((int)time(NULL));
*First=!(*First);
}
return((double)random())/((double) 0x7FFFFFFF);
}

//creates an integer normally distributed with mean m and standard deviation s.
int Normal(int m,
               int s,
           bool* First)
{double z,zu,zv;
zu=sqrt(-2.0*log(RandomNumber(First)));
zv=2*M_PI*RandomNumber(First);
z=(zu)*sin(zv);
int normal=int(z*s+m);
return(normal);}
```

```
/*generates some uniform node coordinates and other node coordinates normally
distributed around one of the uniform node coordinate.*/
void Normal_node_coordinates(UGRAPH<point,int>& G,
                         node_array<int>& x_coord,
                         node_array<int>& y_coord)
{int n=G.number_of_nodes();
int unifn=abs(n/5);
int normn=n-unifn;
array<int> x(1,n);
array<int> y(1,n);
int i,j,m1,m2;
bool First=true;

for (i=0;i<=unifn-1;i++)
{x[i*5+1]=random(10,90);
y[i*5+1]=random(10,90);}

for(i=0;i<=unifn-2;i++)
  {m1=x[i*5+1];
  m2=y[i*5+1];
  for (j=2;j<=5;j++)
    {x[i*5+j]=Normal(m1,5,&First);
    y[i*5+j]=Normal(m2,5,&First);
    }
  }

int hold=(unifn-1)*5+1;
int count=n-hold;
m1=x[hold];
m2=y[hold];
for (i=1;i<=count;i++)
{x[hold+i]=Normal(m1,5,&First);
y[hold+i]=Normal(m2,5,&First);}
```

```
node u=G.first_node();
for (i=1;i<=n;i++)
{x_coord[u]=x[i];
 y_coord[u]=y[i];
u=G.succ_node(u);
}
node v;
forall_nodes(v,G)
G[v]=point(x_coord[v],y_coord[v]);

}

//forms a cycle in the graph G.
void form_cycle(UGRAPH<point,int>& G,
                    node_array<int>& x_coord,
                    node_array<int>& y_coord,
                    node_array<int>& deg,
                    node_matrix<int>& yes_edge)

{edge e;
node u,v;
v=G.first_node();
int counter=1;
  do
  {
   u= G.succ_node(v);
   e=G.new_edge(v,u,counter);
   counter++;
   yes_edge(v,u)=1;
   yes_edge(u,v)=1;
   v=u;
  }
  while (v != G.last_node());
```

```
v=G.last_node();
u=G.first_node();
e=G.new_edge(v,u,counter);
yes_edge(v,u)=1;
yes_edge(u,v)=1;

  forall_nodes(v,G)
{yes_edge(v,v)=1;
  deg[v]=2;}



}

/*given node v, this forms a priority queue cost1 of nodes of the graph with respect to their
distance from v.*/
void form_node_pq(UGRAPH<point,int>& G,
                    node_array<int>& x_coord,
                    node_array<int>& y_coord,
                    node_pq<double>& cost1,
                    node v)

{const double INF=10000.00;
double dist;
node u;
int euc;

cost1.clear();
   forall_nodes(u,G)
   {if (u != v)
     { euc=(x_coord[u]-x_coord[v])*(x_coord[u]-x_coord[v])+(y_coord[u]-
y_coord[v])*(y_coord[u]-y_coord[v]);
dist=sqrt((double) euc);
cost1.insert(u,dist);
```

```
    }
  cost1.insert(v, INF);
  }
}

/*forms the graph with specified nodes and arcs.*/
void make_graph(UGRAPH<point,int>& G,
                node_array<int>& x_coord,
                node_array<int>& y_coord,
                node_array<int>& deg,
                node_matrix<int>& yes_edge,
                int d,
                int* num_edges,
                int n)

{priority_queue<point,int> distance;
int i,j;
node u,v;
point p;
edge e;

node_matrix<int> dist(G,0);
u=G.first_node();
array<node> find_node(1,n);

for (i=1;i<=n;i++)
{find_node[i]=u;
u=G.succ_node(u);}

/*dist(u,v) has the distance between nodes u and v*/
forall_nodes(u,G)
forall_nodes(v,G)
{int euc=(x_coord[u]-x_coord[v])*(x_coord[u]-x_coord[v])+(y_coord[u]-
y_coord[v])*(y_coord[u]-y_coord[v]);
```

```
int dist1=(int)sqrt((double) euc);
dist(u,v)=dist1;
}

/*adds to the graph, arcs between closer node pairs maintaining the average degree close to
d*/

/*distance is a priority queue of node pairs with respect to the distance between them. the
following lines add arcs between closer node pairs whose present degree is less than d*/

int num_edge=G.number_of_edges();
  distance.clear();
    for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
    {u=find_node[i];
     v=find_node[j];
     p=point(i,j);
     // if ((deg[u]<d) && (deg[v]<d))
     distance.insert(p,dist(u,v));
     }
while ((num_edge<n*d/2) && (!distance.empty()))
{
  do{
      p=distance.del_min();
      int x=(int)p.xcoord();
      int y=(int)p.ycoord();
      u=find_node[x];
      v=find_node[y];
      }
   while ((yes_edge(u,v)==1) && (!distance.empty()));
   if ((!distance.empty()) && (deg[u]!=d) && (deg[v]!=d))
      {G.new_edge(u,v);
      deg[u]++;
      deg[v]++;
```

```
yes_edge(u,v)=1;
yes_edge(v,u)=1;
num_edge=num_edge+1;
}

}
```

/*when no more arcs can be added by finding the closest node pairs that are not yet connected and the required arc number is not reached, then finds a node v whose degree is less than d and forms priority queue cost1 for v. Connects v to the closest node that is yet unconnected to it until its degree is d. Continues this for each node until reaching the required arc number.*/

```
node_pq<double> cost1(G);
  forall_nodes(v,G)
  if ((deg[v]<d) && (num_edge<n*d/2))
   {form_node_pq(G,x_coord,y_coord,cost1,v);
   int need=d-deg[v];
   int count=0;
     do
     {u=cost1.del_min();
         if(yes_edge(u,v) !=1)
         {G.new_edge(v,u);
         yes_edge(u,v)=1;
         yes_edge(v,u)=1;
         num_edge=num_edge+1;
         count++;
         deg[v]++;
         deg[u]++;
         }
      }while ((count<need) && (!cost1.empty()) && (num_edge<n*d/2));
   }
cout<<"number of edges="<<num_edge<<endl;
int counter=1;
```

```
 forall_edges(e,G)
   {G[e]=counter;
    counter++;}
*num_edges=G.number_of_edges();
}

/*computes design cost and routing cost for each arc of the graph.*/
void compute_edge_costs(UGRAPH<point,int>& G,
                    node_array<int>& x_coord,
                    node_array<int>& y_coord,
                    edge_array<int>& design_cost,
                    edge_array<int>& routing_cost,
                    int* ratio)
{int hold,euc;
edge e;
node u1,v1;
  forall_edges(e,G)
  {u1=G.source(e);
v1=G.target(e);
euc=(x_coord[u1]-x_coord[v1])*(x_coord[u1]-x_coord[v1])+(y_coord[u1]-
y_coord[v1])*(y_coord[u1]-y_coord[v1]);//Euclidean distance
hold=(int)sqrt((double)euc);
routing_cost[e]=hold;
design_cost[e]=(*ratio)*hold;
   }
}

/*creates specified number of commodities*/
void create_commodities(UGRAPH<point,int>& G,
                         array<node>& origin,
                         array<node>& destination,
                         node_matrix<int>& yes_commodity,
                         int num_comm,
                         int n)
```

```
{
node u,v;
int i,j,inf1,inf2;
array<node> number_node(1,n);
u=G.first_node();
for (i=1;i<=n;i++)
{number_node[i]=u;
u=G.succ_node(u);}

forall_nodes(v,G)
yes_commodity(v,v)=1;

int commodity=1;

for (i=1;i<=n;i++)
for (j=1;j<=n;j++)
{u=number_node[i];
v=number_node[j];
if((commodity<=num_comm) && (yes_commodity(u,v)!=1))
{origin[commodity]=number_node[i];
 destination[commodity]=number_node[j];
 commodity++;
 yes_commodity(u,v)=1;

}
}
}

/*sends the generated graph as an input for the dual-ascent program*/
fort_output(UGRAPH<point,int>& G,
            array<node>& origin,
            array<node>& destination,
            edge_array<int>& design_cost,
         edge_array<int>& routing_cost,
```

```
        int n,
        int num_edges,
        int num_comm,
        int problem_number)
{node u,v;
edge e;
int i,j;
float route,design;
FILE *fp;
char input[7];

printf("Enter the fortran input file name:");
scanf("%s",input);

fp=fopen(input, "w");
fprintf(fp,"%3d %3d %3d\n",n,num_edges,problem_number);
node_array<int> node_number(G);
u=G.first_node();
for (i=1;i<=n;i++)
{node_number[u]=i;
u=G.succ_node(u);}
forall_edges(e,G)
{u=G.source(e);
i=node_number[u];
v=G.target(e);
j=node_number[v];
route=(float)routing_cost[e];
design=(float)design_cost[e];
fprintf(fp,"%3d %3d\n",i,j);
fprintf(fp,"%9.5f %9.5f\n",route,design);
}
int counter,k,l;
for (counter=1;counter<=num_comm;counter++)
{k=node_number[origin[counter]];
```

```
l=node_number[destination[counter]];
fprintf(fp,"%3d %3d %3d\n",counter,k,l);
}

fclose(fp);
}

/*forms a priority queue of the edges of the graph with respect to their design costs*/
void form_pq(UGRAPH<point,int>& G,
                priority_queue<edge,int>& d_cost1,
                edge_array<int>& design_cost,
                array<edge>& rank,
                int num_edges)
{edge e,e1;
int i;
d_cost1.clear();
forall_edges(e,G)
d_cost1.insert(e,design_cost[e]);
pq_item it;

//rank edges based on design_cost, least cost-higher rank.

for (i=0; i<num_edges; i++)//for loop
{
it=d_cost1.find_min();
e1=d_cost1.key(it);
rank[num_edges-i]=e1;
d_cost1.del_item(it);}//for loop
}

/*chooses an edge for the creating individual such that the probability of choosing a lower
design cost edge is higher*/
void choose_edge(int num_ind_arcs,
                 int num_edges,
```

```
                    int sum,
                    int* choose)

{int two,s, diff1,diff2;
two=2*random(1,sum);
s=(int)sqrt((double)two);
diff1=two-s*s;
if (diff1>0)//if loop 2.1
{diff2=two-(s+1)*(s+1);
  if (diff1>abs(diff2))
  *choose=s+1;
 else *choose=s;}//if loop 2.1
else                //else loop 2.1
{diff2=two-(s-1)*(s-1);
 if (diff2>abs(diff1))
 *choose=s;
 else *choose=s-1;
}         //else loop 2.1

}
void feasibility_correct(UGRAPH<point,int>& L,
                              edge_array<int>& design,
                    node v)

{
queue<node> Q;
node_array<int> dist(L);
node w,u1,u2;
edge e,e1;

forall_nodes(w,L) dist[w]=-1;

dist[v]=0;
Q.append(v);
```

```
   while (!Q.empty())
   {v=Q.pop();
    forall_adj_edges(e,v)
    if (L[e]==1)
        {u1=L.source(e);
          u2=L.target(e);
          if (u1==v)
          w=u2;
          else
          w=u1;
            if(dist[w]<0)
            {Q.append(w);
          dist[w]=dist[v]+1;
          L[e]=1;
          }
          }
   }

forall_nodes(w,L)
if (dist[w]<0)
{int min=200000;
forall_adj_edges(e,w)
{if (design[e]<min)
{e1=e;
min=design[e];
}
L[e1]=1;
dist[w]=0;
}
}

}
```

```
/*for a randomly chosen node v, forms a tree (for the creating individual) with edges of the
original graph*/
void rand_tree(UGRAPH<point,int>& L,
            node v)

{
queue<node> Q;
node_array<int> dist(L);
node w,u1,u2;
edge e;

forall_nodes(w,L) dist[w]=-1;

dist[v]=0;
Q.append(v);

 while (!Q.empty())
 {v=Q.pop();
  forall_adj_edges(e,v)
     {u1=L.source(e);
       u2=L.target(e);
       if (u1==v)
       w=u2;
       else
       w=u1;
         if(dist[w]<0)
         {Q.append(w);
       dist[w]=dist[v]+1;
       L[e]=1;
       }
       }
  }
}
```

```
void min_spanning_tree(UGRAPH<point,int>& L,
                        edge_array<int>& design)
{node v,w;
edge e;
edge_array<int> counter(L);
int count=1;
forall_edges(e,L)
{counter[e]=count;
count++;}
node_partition Q(L);
L.sort_edges(design);

forall_edges(e,L)
{v=L.source(e);
w=L.target(e);
if (!(Q.same_block(v,w)))
{Q.union_blocks(v,w);
L[e]=1;
}
}
L.sort_edges(counter);
}


/*makes an individual(a design of the original network).*/
void make_individual(UGRAPH<point,int>& G,
                UGRAPH<point,int>& L,
                edge_array<int>& design_cost,
                array<edge>& rank,
                int n,
                int num_edges,
                int d)

{edge e,e1,e2;
```

```
int i,count;
node u;
L.clear();
L=G;
edge_array<int> design(L);
array<int> fixed_cost(1,num_edges);

count=1;
forall_edges(e,G)
{fixed_cost[count]=design_cost[e];
count++;}

count=1;
forall_edges(e,L)
{design[e]=fixed_cost[count];
count++;}


u=L.first_node();

array<node> find_node(1,n);

//find_node[i] finds the ith node of individual...
for (i=1;i<=n;i++)
{find_node[i]=u;
u=L.succ_node(u);}

array<edge> find_edge(1,num_edges);

//find_edge[j] finds the jth edge of individual...
int number=1;
forall_edges(e,L)
{find_edge[number]=e;
number++;}
```

```
edge e3;
forall_edges(e3,L)
L[e3]=0;

//chooses a node u at random and forms tree with this u as root node.
int p=random(1,100);
if (p<=20)
{ if (p<=10)
  min_spanning_tree(L,design);

  else
  {for (i=1;i<=n-1;i++)
    {int inf1;
     e=rank[num_edges-i];
     inf1=G.inf(e);
     e2=find_edge[inf1];
     L[e2]=1;
     }
   u=L.choose_node();
   feasibility_correct(L,design,u);
  }
}

else
{u=L.choose_node();
rand_tree(L,u);
//determine # edges in this design...
int num_ind_arcs;
int k=random(1,100);

if (k<=20)
num_ind_arcs=random(0,(num_edges-n+1)) ;
```

```
else if ((k>20)&&(k<=100))
num_ind_arcs=random(0,n+1);


int choose=0;
/*chooses with higher probability edges with lower design cost for the individual*/

int sum=num_edges*(num_edges+1)/2;
for (i=0;i<num_ind_arcs;i++)
{choose_edge(num_ind_arcs,num_edges,sum,&choose);
int inf1,inf2;
//add chosen edge to design...
e=rank[choose];
inf1=G.inf(e);
e2=find_edge[inf1];
L[e2]=1;
}
}
}



void dijkstra(UGRAPH<point,int>& G,
          node s,
          edge_array<int>&  cost,
          node_matrix<int>& sh_dist,
             array<int>& ind,
             int* node_selections,
             int* distance_updates)

{/*   computes single source shortest paths from node s for
   a non-negative network (G,cost), computes for all nodes v:
   a) dist[v] = cost of shortest path from s to v
   b) pred[v] = predecessor edge of v in shortest paths tree
```

```
 */
priority_queue<node,int> PQ;
node_array<pq_item> I(G);
node_array<int>  dist(G);
node_array<edge> pred(G);
edge_array<int> indicator(G);

  node v;
  edge e;

int counter=1;
forall_edges(e,G)
{indicator[e]=ind[counter];
counter++;
}
forall_nodes(v,G)
  { pred[v] = nil;
    dist[v] = MAXINT;
    }
dist[s] = 0;
I[s] = PQ.insert(s,0);

  while (! PQ.empty())
  { node u = PQ.del_min();
    (*node_selections)++;
    int du = dist[u];
    forall_adj_edges(e,u)
       {
     if (indicator[e]==1)
       {
       v = target(e);
           if (v==u)
           v=source(e);
     int c = du + cost[e];
```

```
    if (c < dist[v])
    { if (dist[v] == MAXINT)
      I[v] = PQ.insert(v,c);
      else
      PQ.decrease_inf(I[v],c);
         dist[v] = c;
         (*distance_updates)++;
      pred[v]=e;

         }
    }
   }
  }
forall_nodes(v,G)
sh_dist(s,v)=dist[v];
}

void compute_fitness(UGRAPH<point,int>& G,
                UGRAPH<point,int>& L,
                    int* pfitness,
                node_matrix<int>& sh_dist,
                  array<int>& ind,
                edge_array<int>& routing_cost,
                    edge_array<int>& design_cost,
                    array<node>& origin,
                array<node>& destination,
                    int num_comm,
                    int* node_selections,
                    int* distance_updates,
                    int* shortest_path)

{/*computes the fitness of an individual L*/
edge e;
node u,v;
```

```
int i;
int counter=1;

forall_nodes(u,G)
forall_nodes(v,G)
sh_dist(u,v)=0;

forall_edges(e,L)
  {ind[counter]=L.inf(e);
   counter++;
  }



*pfitness=0;
list<node> list1;
node_array<int> yes_list(G);

/*forms a list of all nodes that are origin nodes, then computes shortest paths
from the origin nodes to other nodes of the network using dijkstra's algorithm*/


for (i=1;i<=num_comm;i++)
if (yes_list[origin[i]]==0)
{list1.append(origin[i]);
yes_list[origin[i]]=1;}

while (!list1.empty())
{node s1 = list1.pop();
(*shortest_path)++;
dijkstra(G,s1,routing_cost,sh_dist,ind,node_selections,distance_updates);
}

/*computes the fitness of this individual by adding the shortest paths between
each origin-destination node pairs and the design costs of the arcs of this individual*/
```

```
for (i=1;i<=num_comm;i++)
{*pfitness +=sh_dist(origin[i],destination[i]);
}
int count=1;

forall_edges(e,G)
{if (ind[count]==1)
*pfitness +=design_cost[e];
count++;}

}

/*ranks individuals on the basis of their fitness*/
void rank_individuals(priority_queue<int,int>& P,
                       array<int>& fitness,
                           array<int>& rank1,
                           array<int>& rank2,
                       int N)

{
int i,dummy,dummy1;
P.clear();
for (i=1;i<=N;i++)
P.insert(i,fitness[i]);

pq_item it;


for (i=1; i<=N; i++)//for loop
{
it=P.find_min();
dummy=P.key(it);
dummy1=P.inf(it);
rank1[i]=dummy;
```

```
rank2[i]=dummy1;
P.del_item(it);}//for loop

}
/*selects two individuals from the present population as parents for next cross-over*/
void select_parents(UGRAPH<point,int>* L,
                    UGRAPH<point,int>& best,
                    UGRAPH<point,int>& worst,
                     int* l,
                     int* m,
                    array<int>& rank1,
                  int N)
{
/*best half of the population is the half with lower fitness and the worst ahlf is the half with
higher fitness. With probability .75, chooses one parent from the best half of the
population and other parent from the worst half of the population; with probability .25
chooses both parents from the best half of the population; the better fitness parent is the
best parent and the worse fitness parent is the worst parent*/

int number1=random(1,N/2);
int number2=random(1,N/2);
int p=random(1,100);
if (p<=75)
{ *l=rank1[number1];
 *m=rank1[number2+N/2];
best=L[*l];
worst=L[*m];}
else {
int i=Min(number1,number2);
int j=Max(number1,number2);
*l=rank1[i];
 *m=rank1[j];
best=L[*l];
worst=L[*m];
```

```
}

}
/* creates array that will store ranks of the individual based on individual number
order(rankind), fitness of the individuals based on individual number order(fitind),
individual numbers based on rank of individuals order(indrank), and  fitness based on rank
of individuals order(fitind). This is to facilitate
finding the rank of an individual given its individual number etc.*/

void create_arrays( array<int>& rankind,
                    array<int>& fitind,
                    array<int>& indrank,
                    array<int>& fitrank,
                    int N)
{int i,j,k,fit;
array<int> dummy1(1,N);
array<int> dummy2(1,N);

for (i=1;i<=N;i++)
{dummy1[i]=fitrank[i];
dummy2[i]=indrank[i];
}

for (i=1;i<=N;i++)
  {j=dummy2[i];
   rankind[j]=i;}

for (i=1;i<=N;i++)
{fit=dummy1[i];
 k=dummy2[i];
fitind[k]=fit;
}
}
```

```
/*the selected parents cross-over and form a child*/
void create_child(UGRAPH<point,int>& best,
                  UGRAPH<point,int>& worst,
                           UGRAPH<point,int>& child,
                           int num_edges)

{
array<int> ind_best(1,num_edges);
array<int> ind_worst(1,num_edges);
array<int> ind_child(1,num_edges);

edge e;
int count=1;
int i;

forall_edges(e,best)
{ind_best[count]=best.inf(e);
count++;}

count=1;
forall_edges(e,worst)
{ind_worst[count]=worst.inf(e);
count++;}

/*while alleles corresponding to a gene is the same in both parents, child
takes that allele. while alleles corresponding to a gene is different in the two
parents, child takes the allele from the best parent with 75% probability and the allele from
the worst parent with 25% probability*/

for (i=1; i<=num_edges; i++)
{if (ind_best[i]==ind_worst[i])
   ind_child[i]=ind_best[i];
 else
 {int p=75;
```

```
 int choose=random(1,100);
 if (choose<=p)
 ind_child[i]=ind_best[i];
 else ind_child[i]=ind_worst[i];
  }
 }

count=1;
forall_edges(e,child)
{child[e]=ind_child[count];
count++;}
}
/*after the child is formed its feasibility is checked.*/
void feasibility_check(UGRAPH<point,int>& L,
                                  node v,
                                  int* feas)

{
queue<node> Q;
node_array<int> dist(L);
node w,u1,u2;
edge e;

forall_nodes(w,L) dist[w]=-1;

dist[v]=0;
Q.append(v);

  while (!Q.empty())
  {v=Q.pop();
   forall_adj_edges(e,v)
   if (L[e]==1)
       {u1=L.source(e);
          u2=L.target(e);
```

```
        if (u1==v)
        w=u2;
        else
        w=u1;
          if(dist[w]<0)
          {Q.append(w);
        dist[w]=dist[v]+1;
        L[e]=1;
        }
        }
  }

forall_nodes(w,L)
if (dist[w]<0)
*feas=0;
}

/* selects parents, performs crossover to create children, performs immigration*/
void select_cross_immigrate(UGRAPH<point,int>* L,
                UGRAPH<point,int>& G,
                node_matrix<int>& sh_dist,
                array<int>& fitness,
                array<int>& ind,
                array<int>& exist_check,
                array<int>& fitness_from_rank,
                array<int>& ind_from_rank,
                array<edge>& rank,
                edge_array<int>& routing_cost,
                edge_array<int>& design_cost,
                array<node>& origin,
                array<node>& destination,
                int num_comm,
              int num_edges,
                int N,
```

```
            int n,
            int d,
            int* node_selections,
            int* distance_updates,
            int* shortest_path,
            int* fit_computes,
            int* cross_flag,
            int* feas_child,
            int* intro_child,
            int* replace_parent,
            int* num_immigrate)

{priority_queue<int,int> P;
//array<int> ind_from_rank(1,N);

if (*cross_flag=1)
{rank_individuals(P,fitness,ind_from_rank,fitness_from_rank,N);
*cross_flag=0;}

array<int> rank_from_ind(1,N);
array<int> fitness_from_ind(1,N);
int j;
create_arrays(rank_from_ind,fitness_from_ind,ind_from_rank,fitness_from_rank,N);



UGRAPH<point,int> best;
UGRAPH<point,int> worst;
UGRAPH<point,int> child;
int best_parent,worst_parent;

child=G;
/*selects parents for crossover*/
select_parents(L,best,worst,&best_parent,&worst_parent,ind_from_rank,N);
```

```
/*performs crossover and creates a child*/
create_child(best,worst,child,num_edges);


node u;
int feas=1;
/*tests the feasibility of the formed child*/
u=child.choose_node();
feasibility_check(child,u,&feas);
UGRAPH<point,int> form;
int fit_form;
if (feas==1)
{//feas
int fit_child;
(*feas_child)++;
(*fit_computes)++;
/*computes the fitness of a child when it is feasible*/
compute_fitness(G,child,&fit_child,sh_dist,ind,routing_cost,design_cost,origin,destinatio
n,num_comm,node_selections,distance_updates,shortest_path);

int fb,fw,check;
fb=fitness_from_ind[best_parent];
fw=fitness_from_ind[worst_parent];

int i;


int last=ind_from_rank[N];
int last_one=ind_from_rank[N-1];
int last_two=ind_from_rank[N-2];

/*introduces a child if its fitness is better than the worst individual of the
present population */
```

```
int remove=0;
check=0;
if (fit_child<=fitness[last])
{remove=last;
L[remove]=child;
fitness[remove]=fit_child;
exist_check[remove]=0;
(*intro_child)++;
check=1;
}
/*maintains an existence-check for the worst parent. If it produces bad children sequentially
for three times, then it is a candidate for elimination from the population*/
if(fit_child>fitness[best_parent])
exist_check[worst_parent]+=1;
else exist_check[worst_parent]=0;

int check1=0;
if ((exist_check[worst_parent]>=3) && (remove != worst_parent))
{make_individual(G,form,design_cost,rank,n,num_edges,d);
(*fit_computes)++;
(*replace_parent)++;
compute_fitness(G,form,&fit_form,sh_dist,ind,routing_cost,design_cost,origin,destinatio
n,num_comm,node_selections,distance_updates,shortest_path);
L[worst_parent]=form;
fitness[worst_parent]=fit_form;
exist_check[worst_parent]=0;
check1=1;}


/*performs immigration; a new individual is formed, when its fitness is better
than the worst individual of the population it is introduced.*/

make_individual(G,form,design_cost,rank,n,num_edges,d);
(*fit_computes)++;
```

```
compute_fitness(G,form,&fit_form,sh_dist,ind,routing_cost,design_cost,origin,destinatio
n,num_comm,node_selections,distance_updates,shortest_path);
if (fit_form<fitness[last])
{(*num_immigrate)++;
 {if (check==0)

   {if (worst_parent == last)
   {L[last_one]=form;
   fitness[last_one]=fit_form;
      }
   else {L[last]=form;
       fitness[last]=fit_form;
          exist_check[last]=0;
             }
   }

if (check==1)
  {if(worst_parent ==last_one)
   {L[last_two]=form;
    fitness[last_two]=fit_form;
    exist_check[last_two]=0;
        }
   else{L[last_one]=form;
          fitness[last_one]=fit_form;
       exist_check[last_one]=0;
          }
   }

 }
}
}//feas

/*when the child is not feasible, immigration is performed using the following
few lines*/
```

```
else
  {make_individual(G,form,design_cost,rank,n,num_edges,d);
 (*num_immigrate)++;
(*fit_computes)++;
compute_fitness(G,form,&fit_form,sh_dist,ind,routing_cost,design_cost,origin,destinatio
n,num_comm,node_selections,distance_updates,shortest_path);
  if (fit_form<fitness_from_rank[N])
        {int number=ind_from_rank[N];
        L[number]=form;
        fitness[number]=fit_form;
     exist_check[number]=0;
                }
  }
/*ranks the changed population*/
rank_individuals(P,fitness,ind_from_rank,fitness_from_rank,N);

}

run_iterations (UGRAPH<point,int>* L,
                    UGRAPH<point,int>& G,
                    node_matrix<int>& sh_dist,
                    array<int>& fitness,
                    array<int>& ind,
                    array<int>& exist_check,
                    array<edge>& rank,
                    edge_array<int>& routing_cost,
                    edge_array<int>& design_cost,
                    array<node>& origin,
                array<node>& destination,
                    int num_comm,
                  int num_edges,
                    int N,
                    int n,
                    int d,
```

```
                    int* node_selections,
                    int* distance_updates,
                    int* shortest_path,
                    int* fit_computes,
                    FILE* fp)


{float T;
int i,x,j,min_fitness;
int feas_child=0;
int intro_child=0;
int num_immigrate=0;
int replace_parent=0;
double average_fitness,square_fitness,std_fitness,sum_fitness,average_ten,average_half;
array<int> fitness_from_rank(1,N);
array<int> ind_from_rank(1,N);
sum_fitness=0.0;

std_fitness=0.0;
int num_iteration=0;
int cross_flag=1;
for(i=1;i<=N;i++)
sum_fitness+=(double)fitness[i];
average_fitness=sum_fitness/(double)N;

for(i=1;i<=N;i++)
std_fitness+=(fitness[i]-average_fitness)*(fitness[i]-average_fitness);
std_fitness=std_fitness/N;
std_fitness=sqrt(std_fitness);

for(x=1;x<=N;x++)
fprintf(fp,"%3d\n",fitness[x]);
```

```
/*An iteration is composed of selection, cross-over, and immigration; In intervals of 25
iterations, prints output*/
do
{
fprintf(fp,"average fitness=%f\n",average_fitness);
if (cross_flag!=1)
{fprintf(fp,"best fitness=%d\n",fitness_from_rank[1]);
average_ten=0;
average_half=0;
for(x=1;x<=10;x++)
average_ten+=fitness_from_rank[x];
average_ten=average_ten/10;
int half=N/2;
for(x=1;x<=half;x++)
average_half+=fitness_from_rank[x];
average_half=average_half/half;
fprintf(fp,"best 10 average fitness=%f\n",average_ten);
fprintf(fp,"best half average fitness=%f\n",average_half);
}

fprintf(fp,"standard deviation of fitness=%f\n",std_fitness);
fprintf(fp,"\n\n");



for(j=1;j<=25;j++)
{T = used_time();
select_cross_immigrate(L,G,sh_dist,fitness,ind,exist_check,fitness_from_rank,ind_from_
rank,rank,routing_cost,design_cost,origin,destination,num_comm,num_edges,N,n,d,nod
e_selections,distance_updates,shortest_path,fit_computes,&cross_flag,&feas_child,&intro
_child,&replace_parent,&num_immigrate);}
num_iteration+=25;
fprintf(fp,"number of iterations=%3d\n",num_iteration);
float T1=used_time(T);
```

```
fprintf(fp,"time: %6.2f sec\n",T1);
sum_fitness=0.0;
std_fitness=0.0;
for(i=1;i<=N;i++)
sum_fitness+=(double)fitness[i];

average_fitness=sum_fitness/(double)N;

for(i=1;i<=N;i++)
std_fitness+=(fitness[i]-average_fitness)*(fitness[i]-average_fitness);
std_fitness=std_fitness/N;
std_fitness=sqrt(std_fitness);

}while ((std_fitness>=(average_fitness/1000))&&(num_iteration<=10000));

for(x=1;x<=N;x++)
fprintf(fp,"%3d\n",fitness[x]);


fprintf(fp,"average fitness=%f\n",average_fitness);
fprintf(fp,"best fitness=%d\n",fitness_from_rank[1]);
average_ten=0;
average_half=0;
for(x=1;x<=10;x++)
average_ten+=fitness_from_rank[x];
average_ten=average_ten/10;
int half=N/2;
for(x=1;x<=half;x++)
average_half+=fitness_from_rank[x];
average_half=average_half/half;
fprintf(fp,"best 10 average fitness=%f\n",average_ten);
fprintf(fp,"best half average fitness=%f\n",average_half);
fprintf(fp,"standard deviation of fitness=%f\n",std_fitness);
```

```
fprintf(fp,"number of iterations=%d\n",num_iteration);
fprintf(fp,"percentage of feasible children
=%f\n",((float)(feas_child)*100)/(float)(num_iteration));
fprintf(fp,"percentage of parent replacements
=%f\n",((float)(replace_parent)*100)/(float)(num_iteration));

fprintf(fp,"percentage of introduced
children=%f\n",((float)(intro_child)*100)/(float)(num_iteration));
fprintf(fp,"percentage of immigration
effected=%f\n",((float)(num_immigrate)*100)/(float)(num_iteration));

fprintf(fp,"\n\n");

int final_design=ind_from_rank[1];
edge e;
fprintf(fp,"This is the final design\n");

int tot_edge=0;
forall_edges(e,L[final_design])
{fprintf(fp,"%d",L[final_design][e]);
tot_edge+=L[final_design][e];}
fprintf(fp,"\n\n");
fprintf(fp,"number of edges in the final design= %d\n", tot_edge);
newline;
}


main()
{
UGRAPH<point,int> G;
int i,j,k,n,d,num_comm,x,y,N,problem_number,ratio;
double x1,x2,y1,y2;
int validity=6;
/*Interface for input data*/
```

```
cout<<"Type number of nodes(between 5 and 200):";
cin>> n;
if ((n<5) || (n>200))
validity=0;


cout<<"Type degree(even integer greater than 4 and less than "<<n<<" ):";
cin>> d;

if ((d<4) || (d>=n))
validity=1;

cout<<"Type number of commodities(integer greater than 2 and less than or equal to
"<<n*(n-1)<<" ):";

cin>> num_comm;

if ((num_comm<2) || (num_comm>n*(n-1)))
validity=2;

cout<<"Type design-cost,routing-cost ratio(choose 2,10 or 15):";
cin>>ratio;

//if ((ratio!=2) && (ratio!=10) && (ratio!=15))
//validity=3;

cout<<"Type # individuals:(even number please, between 10 and 2000!)";
cin>>N;

if ((N<10) || (N>2000))
validity=4;

char node_indicator;
```

```
cout<<"Do you want normally generated node_coordinates or uniform
node_coordinates?(Type n or u):";
cin>>node_indicator;

if ((node_indicator!='u') && (node_indicator!='n') )
validity=5;

cout<<"Type problem_number:";
cin>>problem_number;


enum{node_invalid,degree_invalid,commodity_invalid,ratio_invalid,individual_invalid,no
de_indicator_invalid,all_valid};
switch (validity)
{case node_invalid:cerr<<"node # not valid"<<endl;
            exit(0);
case degree_invalid:cerr<<"degree # not valid"<<endl;
                         exit(0);
case commodity_invalid:cerr<<"commodity # not valid"<<endl;
                         exit(0);
case ratio_invalid:cerr<<"ratio not valid"<<endl;
                         exit(0);
case individual_invalid:cerr<<"individual # not valid"<<endl;
                         exit(0);
case node_indicator_invalid:cerr<<"node coordinate type not valid"<<endl;
                         exit(0);

case all_valid:break;
}
FILE *fp;
char output[7];

printf("Enter the output file name:");
scanf("%s",output);
```

```
fp=fopen(output,"w");
fprintf(fp,"GENETIC ALGORITHM FOR PROBLEM %3d\n",problem_number);
fprintf(fp,"number of nodes=%3d\n",n);

//creates a graph which will have n nodes;
create_graph(G,n);


node_array<int> x_coord(G), y_coord(G);
node_array<int> deg(G);
node_matrix<int> yes_edge(G,0);
int num_edges;
init_random();
//creates node-coordinates.
if (node_indicator=='u')
uniform_node_coordinates(G,x_coord,y_coord);
else
Normal_node_coordinates(G,x_coord,y_coord);

//forms a cycle in the graph G.
form_cycle(G,x_coord,y_coord,deg,yes_edge);

if (d>2)
make_graph(G,x_coord,y_coord,deg,yes_edge,d,&num_edges,n);

edge_array<int> design_cost(G), routing_cost(G);
compute_edge_costs(G,x_coord,y_coord,design_cost,routing_cost,&ratio);
fprintf(fp,"number of edges=%3d\n",num_edges);
fprintf(fp,"number of commodities=%3d\n",num_comm);
fprintf(fp,"number of individuals=%3d\n",N);
fprintf(fp,"type of node coordinates=%c\n",node_indicator);
fprintf(fp,"design-routing cost ratio=%3d\n",ratio);
fprintf(fp,"\n\n");
```

```
newline;
array<node> origin(1,num_comm);
array<node> destination(1,num_comm);
node_matrix<int> yes_commodity(G,0);
//create specified number of commodities.
create_commodities(G,origin,destination,yes_commodity,num_comm,n);
fort_output(G,origin,destination,design_cost,routing_cost,n,num_edges,num_comm,prob
lem_number);

node u,v;
edge e,e1,e2;
int num;
//priority queue of edges with design_cost...
float T;
T = used_time();

UGRAPH<point,int> L[N+1];

priority_queue<edge,int> d_cost1;
array<edge> rank(1,num_edges);

//create a priority queue of edges with respect to their design costs.
form_pq(G,d_cost1,design_cost,rank,num_edges);

//create individuals for the initial population.
for (num=1;num<=N;num++)//for loop num
make_individual(G,L[num],design_cost,rank,n,num_edges,d);

array<int> ind(1,num_edges);
array<int> fitness(1,N);
node_matrix<int> sh_dist(G);
```

```
array<int> exist_check(1,N);
int node_selections=0;
int distance_updates=0;
int shortest_path=0;
int fit_computes=0;

//computes fitness for individuals of the initial population.
for (j=1;j<=N;j++)
{fit_computes++;
compute_fitness(G,L[j],&fitness[j],sh_dist,ind,routing_cost,design_cost,origin,
destination,num_comm,&node_selections,&distance_updates,&shortest_path);}


//runs iterations of selection, crossover and immigration and oresebts output.
run_iterations(L,G,sh_dist,fitness,ind,exist_check,rank,routing_cost,design_cost,origin,d
estination,num_comm,num_edges,N,n,d,&node_selections,&distance_updates,&shortest
_path,&fit_computes,fp);

fprintf(fp,"total computational time excluding network generation= %6.2f
sec\n",used_time(T));
fprintf(fp,"number of node selections=%3d\n",node_selections);
fprintf(fp,"number of distance updates=%3d\n",distance_updates);
fprintf(fp,"number of shortest path computations=%3d\n",shortest_path);
fprintf(fp,"number of fitness computations=%3d\n",fit_computes);
fclose(fp);

}//end of main
```