01 May 1993

Parallel Genetic Algorithms for the DAG Vertex Splitting Problem

Matthias Mayer

Fikret Erçal
Missouri University of Science and Technology, ercal@mst.edu

PARALLEL GENETIC ALGORITHMS FOR THE DAG VERTEX SPLITTING PROBLEM

M. Mayer* and F. Ercal

CSc-93-10

Department of Computer Science

University of Missouri - Rolla

Rolla, MO 65401 (314)341-4491

*This report is substantially the M.S. thesis of the first author, completed May 1993.

ABSTRACT

Directed Acyclic Graphs are often used to model circuits and networks. The path length in such Directed Acyclic Graphs represents circuit or network delays. In the vertex splitting problem, the objective is to determine a minimum number of vertices from the graph to split such that the resulting graph has no path of length greater than a given δ. The problem has been proven to be NP-hard.

A Sequential Genetic Algorithm has been developed to solve the DAG Vertex Splitting Problem. Unlike a standard Genetic Algorithm, this approach uses a variable chromosome length to represent the vertices that split the graph and a dynamic population size. Two String Length Reduction Methods to reduce the string length and two Stepping Methods to explore the search space have been developed. Combinations of these four methods have been studied and conclusions are drawn.

A parallel version of the sequential Genetic Algorithm has been developed. It uses a fully distributed scheme to assign different string lengths to processors. A ring exchange method is used in order to exchange "good" individuals between processors. Almost linear speed-up and two cases of super linear speed-up are reported.

TABLE OF CONTENTS

Page

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF ILLUSTRATIONS ix

LIST OF TABLES xii

SECTION

I. INTRODUCTION 1

A. GENETIC ALGORITHMS AND RELATED AREAS 1

B. THE DAG VERTEX SPLITTING PROBLEM 2

C. OUTLINE OF THESIS 3

II. BASICS OF A GENETIC ALGORITHM 4

A. DEFINITIONS 4

B. BASIC FUNCTIONS 5

1. The Select Function 5

a. Roulette Wheel Selection 5

b. Tournament Selection 6

c. Ranking Selection 6

2. Crossover 7

a. One-Point Crossover 7

b. Two-Point Crossover 7

c. Uniform Crossover 8

Page

3. Mutation 8

4. Recombination 9

C. THE SCHEMA THEOREM 9

D. PARALLEL GENETIC ALGORITHMS 15

E. PROBLEMS WITH PROPORTIONAL ALLOCATION 17

1. Deceptive Problem 17

2. Premature Convergence 17

3. Genetic Drift 17

F. MODIFICATIONS TO GENETIC ALGORITHMS 18

1. Crowding 18

2. Elitism 18

G. SUMMARY 18

III. THE DAG VERTEX SPLITTING PROBLEM 20

IV. THE GENETIC ALGORITHM FOR THE DVSP 23

A. OBJECTIVE AND CHROMOSOME ENCODING 23

B. GENERAL OUTLINE OF THE GENETIC ALGORITHM 24

C. BINARY APPROXIMATION 25

D. SELECT FUNCTION 27

E. CROSSOVER FUNCTION 27

F. MUTATION FUNCTION 29

G. RECOMBINATION FUNCTION 30

H. TAKE CARE OF ONES 30

Page

I. SUMMARY 31

V. STRING LENGTH REDUCTION AND STEPPING 32

A. STRING LENGTH REDUCTION METHODS 32

1. Preserve Duplicates 32

2. Random Delete 33

B. STEPPING METHODS 33

1. Linear Stepping 33

2. Multiple Binary Stepping 34

C. SUMMARY 35

VI. THE PARALLEL GENETIC ALGORITHM 37

A. OUTLINE OF THE PARALLEL GENETIC ALGORITHM 37

B. EXCHANGE INITIAL STRING LENGTH 40

C. DETERMINE NEW STRING LENGTH 40

D. REMOTE SOLUTION 41

E. ADJUST BOUNDARIES 42

F. EXCHANGE INDIVIDUALS 43

G. SUMMARY 45

VII. EXPERIMENTAL RESULTS 46

A. THE GRAPHS 46

B. THE SEARCH SPACE 47

C. BINARY APPROXIMATION 51

D. STRING LENGTH REDUCTION AND STEPPING 57

Page

E. PARAMETER "ALONE" 59

F. SPEED-UP 69

G. SUMMARY 74

VIII. CONCLUSIONS 77

A 80

B 86

BIBLIOGRAPHY 88

VITA 91

LIST OF ILLUSTRATIONS

Figure		Page
1.	Select individuals using a roulette wheel	6
2.	Example of a One-Point Crossover	7
3.	Example of a Two-Point Crossover	8
4.	Example of a Uniform Crossover	8
5.	Basic structure of a Genetic Algorithm	10
6.	Effect of crossover on schemata	13
7.	Outline of a Parallel Genetic Algorithm	16
8.	DAG with path length 3	21
9.	DAG with vertex 3 split into vertex 3^i and vertex 3^o	21
10.	Outline of the GA for the DVSP	25
11.	Pseudo code of *Binary Approximation*	26
12.	Example of a crossover with duplicated vertices	28
13.	Example of a crossover with separated duplicated vertices	29
14.	Illustration of the *recombination* function	31
15.	Example of a Multiple Binary Search	35
16.	Outline of the Parallel Genetic Algorithm for the DVSP	38
17.	Pseudo code of the function *exchange initial string length*	41
18.	Pseudo code of the function *determine new string length*	41
19.	Pseudo code of the function *exchange individuals*	44
20.	Shape of the search space for graph C432	50

Figure Page

21. Initial string length reduction with Binary Approximation on graph C432 52

22. Initial string length reduction with Binary Approximation on graph C880 53

23. Initial string length reduction with Binary Approximation on graph C1355 54

24. Percentage of reduction in string length for graphs C432, C880, and C1355 55

25. Test run on PGA *without* full distribution and subpopulation size 50 62

26. Test run on PGA *with* full distribution on subpopulation size 50 63

27. Test run on PGA *without* full distribution and subpopulation size 150 65

28. Test run on PGA *with* full distribution and subpopulation size 150 66

29. Total run time of PGAs 67

30. Solution quality with different PGAs 68

31. Speed-up of the Parallel Genetic Algorithm *with* full distribution and total population sizes of 160 and 320. 71

32. Run time and solution quality with different number of processors and a total population size of 160. 73

33. Run time and solution quality with different number of processors and a total population size of 320. 73

34. Initial string length reduction with Binary Approximation on graph C2670 81

35. Initial string length reduction with Binary Approximation on graph C3540 82

36. Initial string length reduction with Binary Approximation on graph C5315 83

37. Initial string length reduction with Binary Approximation on graph C6288 84

Figure Page

38. Initial string length reduction with Binary Approximation on graph C7552 85

39. Percentage of reduction in string length for graphs C2670, C3540, C5315, C6288, and C7552 87

LIST OF TABLES

Table		Page
I.	COMPARISON OF NATURAL GENETICS AND GENETIC ALGORITHM TERMINOLOGY	5
II.	CIRCUIT CHARACTERISTICS OF ISCAS-85 COMBINATIONAL BENCHMARKS	47
III.	POTENTIAL NUMBER OF VERTICES AND TOTAL SIZE OF THE SEARCH SPACE FOR THE BENCHMARK GRAPHS	48
IV.	RANKING OF FOUR GENETIC ALGORITHMS USING DIFFERENT STRING LENGTH REDUCTION AND STEPPING TECHNIQUES	59

I. INTRODUCTION

This chapter gives a short introduction to Genetic Algorithms and related areas like Evolution Strategies and Genetic Programming. It introduces the DAG Vertex Splitting Problem that is solved with sequential and Parallel Genetic Algorithms. The outline of this thesis is also given in this chapter.

A. GENETIC ALGORITHMS AND RELATED AREAS

Genetic algorithms [1] (GAs) are adaptive search techniques that have been shown to be robust optimization algorithms. In contrast to other optimization techniques, genetic algorithms base their progress on the performance of a population of candidate solutions, rather than on one candidate solution. GAs are loosely based upon Darwin's principle of natural selection and natural genetics. They have become increasingly popular in recent years as a method for solving combinatorial optimization problems [2].

Besides Genetic Algorithms there exist other optimization techniques that use natural mechanics. ***Evolution Strategies*** (ES) were first introduced by Reichenberg [3][4]. In [3], Bäck et. al. explains the main difference between Genetic Algorithms and Evolution Strategies, the so called *two-level learning* process. This process allows ESs not only to change the population according to given strategy parameters but also to change the parameters itself. The strategy parameters make up an internal model of the objective

function, which are learned while seeking the optimum without any external controlling instance or additional measure of fitness.

Genetic Programming [5][6] is a fairly new paradigm that uses natural mechanics to create computer programs. In the Genetic Programming paradigm, the individuals in the population are compositions of functions and terminals appropriate to the particular problem domain. The set of functions used typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. The set of terminals used typically includes inputs appropriate to the problem domain and various constants. The functions in the function set must be well defined for any combination of elements from the range of every function and every terminal that it may encounter. The search space is a hyperspace of all possible compositions of functions that can be recursively composed of the available functions and terminals. Koza [5] lists a variety of applications where Genetic Programming has been applied.

B. THE DAG VERTEX SPLITTING PROBLEM

Many applications involving computer networks and electrical circuits can be modeled as a graph and path lengths in these graphs represent circuit or network delays. If the path lengths in a Directed Acyclic Graph are too long and need to be reduced, the reduction can be done by splitting certain vertices in the graph into two vertices which results in a reduction of the path length. The DAG Vertex Splitting Problem is an optimization problem in which the smallest number of vertices in the graph have to be found such that the longest path in the splitted graph is less than or equal a pre-specified

maximum called δ. The DAG Vertex Splitting Problem addressed in this thesis has many applications in the fields of computer science and electrical engineering. An application would be to find the minimum number of placements of signal boosters in a network, where the computers in the network are represented by the vertices of the graph. The placement of flip-flops in partial scan designs [7] is another application. Heuristics [7] have been used earlier to solve the DAG Vertex Splitting Problem. The DAG Vertex Splitting Problem has been identified as NP-hard [7].

C. OUTLINE OF THESIS

This thesis is organized into 8 chapters. Chapter II defines Genetic Algorithms and describes the standard functions used in a Genetic Algorithm. The *Schemata Theorem*, the fundamental theorem in Genetic Algorithms, shows why and how Genetic Algorithms work. Chapter II also mentions problems in Genetic Algorithms and how they can be solved. Chapter III states and discusses the DAG Vertex Splitting Problem. The general outline of the sequential Genetic Algorithm to solve the DAG Vertex Splitting Problem, along with a discussion about the functions that perform reproduction, crossover, and mutation are discussed in Chapter IV. Chapter V explains the string length reduction techniques that delete vertices from the strings (chromosomes) as well as the stepping methods to explore the search space. Chapter VI describes the parallel version of the Genetic Algorithm implemented on the iPSC/2 Intel Hypercube. Experimental results are reported in Chapter VII. Chapter VIII summarizes the results and points to some future research directions.

II. BASICS OF A GENETIC ALGORITHM

A Genetic Algorithm is an adaptive search technique which employs selection of fitter individuals in a population, similar to Darwin's evolution theory [2]. This chapter defines a Genetic Algorithm and its basic outline. It also discusses the Schema Theorem, the most fundamental theorem in Genetic Algorithms. Parallel Genetic Algorithms are a special form of GAs. Genetic Algorithms are not free of problems. Some of these problems are discussed in this chapter along with possible solutions.

A. DEFINITIONS

A Genetic Algorithm requires a ***population*** P to operate. A population is a set of ***individuals*** P_i, taken from a set of possible solutions. Every individual P_i has one or multiple ***chromosomes*** which have the genetic material encoded. Chromosomes consist of ***genes*** and the positions of the genes in the chromosome are called ***loci*** or ***locus*** for a single position. Genes may take on a number of values called ***alleles*** and the genes are taken from an ***alphabet***. The genetic material of an individual is usually encoded binary. Thus, the alphabet consist only of two symbols, '0' and '1'. An example of a binary encoded chromosome with a length of 10 is 0010110001. ***Parents*** are selected from the population in order to ***mate*** and to produce ***offspring***. The offspring and the parents create the ***next generation***. Genetic Algorithms borrow their lingo from natural genetics but there exist synonyms for all the terms used as listed in Table I.

Table I. COMPARISON OF NATURAL GENETICS AND GENETIC ALGORITHM TERMINOLOGY

Natural Genetics	Genetic Algorithms
Chromosome	String
Gene	Character
Locus	String Position
Allele	Feature Value

Each individual has a ***fitness value*** which is determined by the ***fitness*** or ***evaluation function***. The objective of the Genetic Algorithm is to optimize the evaluation function.

B. BASIC FUNCTIONS

This section talks about some of the basic functions used in Genetic Algorithms. The functions are ***selection***, ***crossover***, ***mutation***, and ***recombination***.

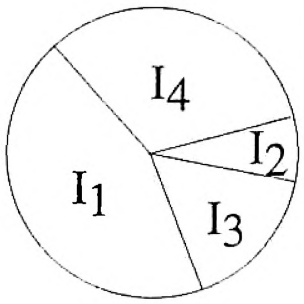
1. The Select Function. The purpose of the ***select*** function is to select individuals from the population for reproduction. Since Genetic Algorithms are based on Darwin's Evolution Theory, also known as ***survival of the fittest method***, those individuals that have a high fitness value also have to have a high probability of getting selected. Several methods have been developed that perform the select.

a. Roulette Wheel Selection. The most commonly used selection method is the so called ***roulette wheel selection*** [1]. Every individual in the population has a slot in the roulette wheel sized in proportion of its fitness value over the cumulated fitness value of

the total population. Thus, every individual has a certain probability of getting selected according to its fitness among the other individuals in the population. Figure 1 shows an example of the roulette wheel selection with four individuals.

b. Tournament Selection. In the ***tournament selection*** [8][9][10], some number of individuals (tournament size) are chosen randomly from the population and the best individual from this group is selected for further genetic processing. Tournaments are often held between pairs of individuals (tournament size of 2), although larger tournaments can be used.

c. Ranking Selection. In the ***ranking selection*** [11] the population is sorted from best to worst. A number is assigned to each individual according to a non-increasing assignment function and then proportional selection is used according to the assigned number.



Individual	Value	Percentage
1	31.5	45 %
2	3.5	5 %
3	14	20 %
4	21	30 %
Total:	70	100 %

Figure 1. Select individuals using a roulette wheel

2. Crossover. The ***crossover*** function is used to create the chromosomes for the offspring from the chromosomes of the parents. There are several different methods of doing the crossover.

a. One-Point Crossover. This method defines a so called ***crossover point***, a place between loci where a chromosome can be split. This crossover point is chosen randomly. Two new chromosomes are created by swapping every gene after or before the crossover point between both parents. Figure 2 shows an example of a One-Point Crossover where the crossover point has been marked by |.

```
Parent 1:       0    1    1    0  |  0    1    0    0
Parent 2:       0    1    0    1  |  1    1    1    0
Crossover Point:
Offspring 1:    0    1    1    0  |  1    1    1    0
Offspring 2:    0    1    0    1  |  0    1    0    0
```

Figure 2. Example of a One-Point Crossover

b. Two-Point Crossover. The ***Two-Point Crossover*** defines two crossover points and the genes are swapped between the two crossover points. Figure 3 gives an example of a Two-Point Crossover. A generalization of this method is the ***N-Point Crossover*** [12] which defines n crossover points.

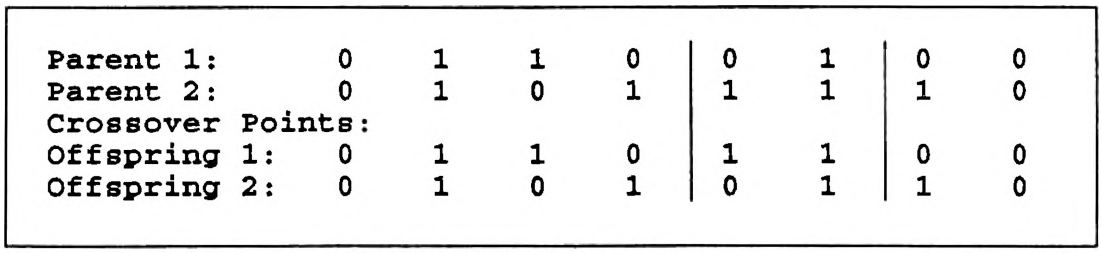


Figure 3. Example of a Two-Point Crossover

c. Uniform Crossover. The ***Uniform Crossover*** [13][14] became very popular recently because it uses a more general approach. Instead of using crossover points the uniform crossover defines a binary ***crossover mask*** with a length equal to the length of the chromosomes. For each loci in the chromosomes of the parents, the genes are swapped if the crossover mask has a one in the same position. Figure 4 illustrates an example of a uniform crossover.

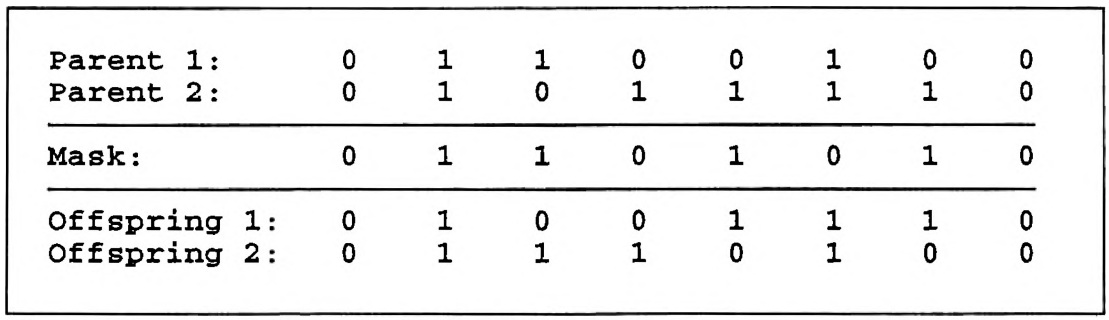


Figure 4. Example of a Uniform Crossover

The crossover mask is chosen randomly with a probability of 50%. Recent research by Spears and DeJong [15] suggests a new type of uniform crossover, the ***Parameterized Uniform Crossover***, where the crossover mask is chosen with a probability other than 50%.

3. Mutation. The ***mutation*** function is another way of altering genes in chromosomes and plays an important role both in natural and artificial genetics. Mutation

is needed because it prevents the loss of potentially useful genetic material. By itself, mutation is a random walk through the chromosomes in the population which changes bits in chromosomes at a very low rate. This rate is called the ***mutation rate*** and is part of the parameter list of a GA.

4. Recombination. The **recombination** function defines how the new generation is to be build from the parents and offspring. There are several ways of doing the recombination. One way of doing it is to replace the parents by the offspring after every crossover. Another way is to replace a certain percentage of the population by offspring during each generation. The parameter, ***generation gab*** G, controls this percentage. That is $N \cdot (1 - G)$ individuals of the population $P(t)$ are chosen randomly to survive into the next generation, where N is the population size. A $G = 1.0$ means that the entire population is replaced by the offspring in each generation. A third way is used in this thesis where the best individuals from the parents and the offspring are selected to form the new generation [16].

The basic structure of a Genetic Algorithm can be seen in Figure 5.

C. THE SCHEMA THEOREM

The ***Schema Theorem*** [1] is one of the most fundamental theorems in Genetic Algorithms. A ***schema*** (plural: ***schemata***) is a similarity template describing a subset of strings with similarities at certain string positions. The theorem enforces to extend the definition of the alphabet by a so called ***don't care*** symbol, **#** for the schemata. Thus, the binary alphabet for schemata consist of three symbols {0, 1, #}. A schema can be

```
create initial population;
evaluate population;
while( termination criteria not reached )
{
      select individuals for reproduction;
      crossover to generate offspring;
      mutate offspring;
      recombine parents and offspring to create new
      generation;
      evaluate new generation;
}
```

Figure 5. Basic structure of a Genetic Algorithm

considered as a pattern matching device in the following sense: a schema matches a particular string if at every location in the schema a 1 matches a 1 in the string, a 0 matches a 0, or a # matches either. For example, the schema #0000 matches two strings, namely {10000, 00000} and the schema #01#0 describes a subset with four members {00100, 00110, 10100, 10110}.

Schemata have two characteristics. The ***order*** of a schema H, denoted by $o(H)$, is the number of fixed (non #) symbols in the schema. The ***defining length*** of a schema H, denoted by $\delta(H)$, is the distance from the first to the last fixed position. For example, the schema 10##0 has order 3 and defining length 4. Schemata and their properties are interesting to study for classifying string similarities.

Suppose at a given time step t, there are m examples of a particular schema H contained in the population $P(t)$ where m is a function of H and t, i.e. $m = m(H, t)$. During reproduction, a string is copied according to its fitness, or more precisely, a string

P_i gets selected with probability $p_i = \frac{f_i}{\sum f_i}$ where f_i is the fitness value of P_i. After picking a nonoverlapping population of size n with replacement from the population $P(t)$, it is expected to have $m(H, t+1)$ representatives of the schema H in the population P at time $t+1$ as given by the equation $m(H, t+1) = m(H, t) \cdot n \cdot \frac{f(H)}{\sum f_i}$, where $f(H)$ is the average fitness of the strings representing schema H at time t. The average fitness of the entire population can be written as $\bar{f} = \frac{\sum f_i}{n}$, then the ***schema growth equation*** can be rewritten as follows:

$$m(H, t+1) = m(H, t) \cdot \frac{f(H)}{\bar{f}} \qquad (1)$$

In other words, a particular schema grows in proportion with the ratio of the average fitness of the schema over the average fitness of the population. Therefore, schemata with fitness values above the population average will receive an increasing number of samples in the next generation, while schemata with fitness values below the population average will receive a decreasing number of samples. It is interesting to note that this behavior is carried out with every schema H contained in a particular population P in parallel.

Suppose there exists a particular schema H whose fitness value is above the population average by an amount $c\bar{f}$ with c being a constant. Under this assumption the schemata growth equation can be rewritten as follows:

$$m(H, t+1) = m(H, t) \cdot \frac{\bar{f} + c\cdot\bar{f}}{\bar{f}} = (1 + c) \cdot m(H, t) \qquad (2)$$

Starting at $t = 0$ and assuming a stationary value of c, the following equation can be obtained:

$$m(H, t) = m(H, 0) \cdot (1 + c)^t \qquad (3)$$

The effect of reproduction is now quantitatively clear. Reproduction allocates ***exponentially*** increasing (decreasing) numbers of trials to above- (below-) average schemata.

Equation (1) considers reproduction as the only operator in a GA. But reproduction alone does not explore new regions in the search space. Only in combination with crossover, new individuals are created. Therefore crossover must be taken into consideration. To see which schemata are affected by crossover, consider the following example of a string of length $l = 7$ and a randomly chosen crossover point shown in Figure 6.

Unless string P_i's mate is identical to P_i at the fixed positions of the schema H_1, schema H_1 will be destroyed because the 1 at position 2 and the 0 at position 7 will be placed in different offspring (they are on opposite sides of the crossover point). Schema H_2 will survive because all fixed positions in H_2 will be copied into a single offspring.

String P_1:	0	1	1	1	0	0	0
Schema H_1:	#	1	#	#	#	#	0
Schema H_2:	#	#	#	1	0	#	#

Figure 6. Effect of crossover on schemata

In general, schema H_1 is less likely to survive crossover than schema H_2 because a crossover point is more likely to fall between the two extreme fixed positions of Schema H_1. Since the crossover point is selected randomly, schema H_1 is destroyed with probability $p_d = \frac{\delta(H_1)}{l-1}$, where $\delta(H_1)$ is the defining length of schema H_1. The probability of survival is $p_s = 1 - p_d$. Considering crossover performed randomly with a probability p_c at a particular mating, the survival probability is given by:

$$p_s \geq 1 - p_c \cdot \frac{\delta(H)}{l-1} \tag{4}$$

Considering reproduction and crossover together, the schema growth equation can be estimated as follows:

$$m(H,t+1) \geq m(H,t) \cdot \frac{f(H)}{\bar{f}} \cdot \left[1 - p_c \cdot \frac{\delta(H)}{l-1}\right] \tag{5}$$

A simple interpretation of the above equation is that a schema is more likely to survive if it has an ***above average performance*** and a ***short defining length***.

The last operator that affects the schema growth is mutation. Mutation is the random alteration of a single position with probability p_m. In order for a schema H to survive, all of the fixed positions must themselves survive. Thus, since a single position survives with probability $(1 - p_m)$, and since each mutation is statistically independent, a particular schema survives when each of the $o(H)$ fixed positions within the schema survive. Thus, the probability of surviving mutation is $(1 - p_m)^{o(H)}$. For small values of p_m ($p_m << 1$), the schema survival probability may be approximated by the expression $1 - o(H) \cdot p_m$. Therefore, the schema growth equation considering reproduction, crossover, and mutation can be expressed by the following equation:

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \cdot \left[1 - p_c \cdot \frac{\delta(H)}{l - 1} - o(H) \cdot p_m \right] \qquad (6)$$

In other words, schemata with a ***low-order***, an ***above average fitness***, and a ***short defining length*** are more likely to survive than others. Equation (6) is called the ***Schema Theorem***, or the ***Fundamental Theorem of Genetic Algorithms***. Since these short, highly-fit, low order schema are important, they have a special name. They are called ***Building Blocks***.

D. PARALLEL GENETIC ALGORITHMS

Genetic Algorithms tend to be slow algorithms, when implemented on a sequential machine. But, they also have a big advantage: they are easier to parallelize than other optimization techniques, like Simulated Annealing. ***Parallel Genetic Algorithms*** (PGAs) are GAs that run on a parallel machine and use the available processors to solve the problem. PGAs have been described in many papers [17][18][19][20] and they become increasingly popular as parallel machines become more available.

Jog, et.al. [17] point out some weaknesses in the design of genetic algorithms as originally proposed by Holland [1]:

- In accordance with nature, it is more natural to view a population as consisting of a set of independent structures, each with its own local behavior, i.e., each has the opportunity to initiate or undergo recombination operators, without the control of a global agent.
- Selection in standard GAs is a global process, i.e., selection of an individual depends on its performance relative to the average performance of the ***entire population***. This is quite different from natural selection and inefficient to parallelize [21]. Parallel Genetic Algorithms therefore introduce ***local selection*** without affecting the performance of the algorithm.
- PGAs allow for asynchronous behavior. This is not possible in standard GAs. This allows different structures to evolve at different speeds which may result in the

global speed-up of the algorithm as well as the maintenance of diversity, which is a critical component for the success of a GA.

A PGA operates on a large population that is distributed into several subpopulations of individuals [19]. Every processor in the multiprocessor system runs a sequential Genetic Algorithm on one of the subpopulations. In order to increase the power of PGAs, processors exchange information via message-passing. This communication consists of exchanging one or many individuals among the processors. At each communication point a certain number of the best individuals of each processor are sent to one of its neighbors. After the individuals have been received, they have to be inserted into the population by either replacing old individuals randomly or by replacing the worst individuals. The modified code for the Parallel Genetic Algorithm is given in Figure 7.

```
create initial subpopulation;
evaluate subpopulation;
while( termination criteria not reached )
{
     communicate with neighbor;
     select individuals for reproduction;
     crossover to generate new offspring;
     recombine parents and offspring to create new
generation;
     evaluate new generation;
}
```

Figure 7. Outline of a Parallel Genetic Algorithm

E. PROBLEMS WITH PROPORTIONAL ALLOCATION

Although assigning probabilities to individuals for selection based on each individual's fitness value relative to the accumulated fitness of the total population increases superior schemata in successive generations, there are several problems associated with this selection system as explained below [22]:

1. Deceptive Problem. The ***deceptive problem*** occurs, when superior individuals in the population do not represent characteristics represented in the optimal value. Thus, the Genetic Algorithm will be led away from the optimal solution.

2. Premature Convergence. A Genetic Algorithm converges prematurely, when the population creates individuals that have already been created and evaluated. Thus, little further exploration can be done. This problem can be reduced by introducing some randomness into the Genetic Algorithm. This is done by mutation. The randomness introduced by mutation will be kept in future generations if it contributes to higher fitness values and it is discarded otherwise.

3. Genetic Drift. ***Genetic Drift*** is a problem that occurs when the ratio between superior and average schemata is low. If genetic drift occurs, the population starts to converge to an arbitrary individual. An attempt of re-introducing lost alleles with a higher mutation rate usually does not work [12].

F. MODIFICATIONS TO GENETIC ALGORITHMS

As these problems with Genetic Algorithms were explored, modifications were designed to correct the problems. This section discusses some of the methods that are used to correct the GA problems associated with GAs [22]:

1. Crowding. One way of slowing down the convergence rate is to use ***crowding*** suggested by DeJong [12]. The population is not a set of offspring generated from the previous population. Instead, each offspring replaces one individual in the population. An individual is replaced with a higher probability if the chromosomes of the individual and of the offspring are similar. This prevents the population from having duplicated genetic material. Therefore the population has a high diversity.

2. Elitism. If the best individual of any population is not represented in the new population, then this best individual is included into the new population as a new member. This method was also suggested by DeJong [12].

G. SUMMARY

This chapter defined a Genetic Algorithm and the terminology used. Based upon simple functions like ***select***, ***crossover***, and ***mutation***, Genetic Algorithms have been shown to be robust optimization algorithms. They differ from other optimization and search techniques in the following ways:

1) GAs work with a coding of the parameter set, not with the parameters themselves.

2) GAs search from a population of points, not a single point.

3) GAs use payoff (objective function) information, not derivatives or other knowledge.

4) GAs use probabilistic transition rules, not deterministic rules.

The *Schemata Theorem* lays the foundation for the GA theory and shows that short, low order, above average schemata receive exponential trials in following generations. Parallel Genetic Algorithms are a way to use Genetic Algorithms in a more natural way than sequential Genetic Algorithms. But, Genetic Algorithms also have problems. Deception, Premature Convergence and Genetic Drift are only some of the problems that have emerged. Modifications were made to Genetic Algorithms in order to correct the problems.

III. THE DAG VERTEX SPLITTING PROBLEM

This chapter defines the DAG Vertex Splitting Problem and its terminology. Solutions to the problem have a wide application in Computer Science and Electrical Engineering.

The DAG vertex splitting problem (DVSP) can be stated as follows [7]: Let $G = (V, E, w)$ be a ***weighted directed acyclic graph*** (WDAG) with vertex set V, edge set E, and edge function w. $w(i, j)$ is the weight of the edge $<i, j> \in E$. $w(i, j)$ is a positive real number for $<i, j> \in E$ and is undefined if $<i, j> \notin$ E. The *delay, d(P)*, on the path P, is the sum of the weights of all the edges on that path P, which can be expressed as follows:

$$d(P) = \sum w(i, j), \ \forall <i, j> \in P \tag{7}$$

The *delay, d(G)*, of the graph G is the maximum path delay in the graph, which is expressed in the following equation:

$$d(G) = \max d(P), \ \forall P \in G \tag{8}$$

Figure 8 shows a DAG with a maximum path length of 3.

Let G/X be the WDAG that results when each vertex v in X is split into two vertices v^i and v^o such that all outgoing edges $<v, j> \in E$ are replaced by edges of the form $<v^o, j>$ and all incoming edges $<i, v> \in E$ are replaced by edges of the form $<i, v^i>$. Outbound edges of v now leave v^o, while inbound edges of v now enter v^i. Figure 9 shows the result, G/X, when splitting vertex 3 of the DAG of Figure 8 into vertex 3^i and vertex

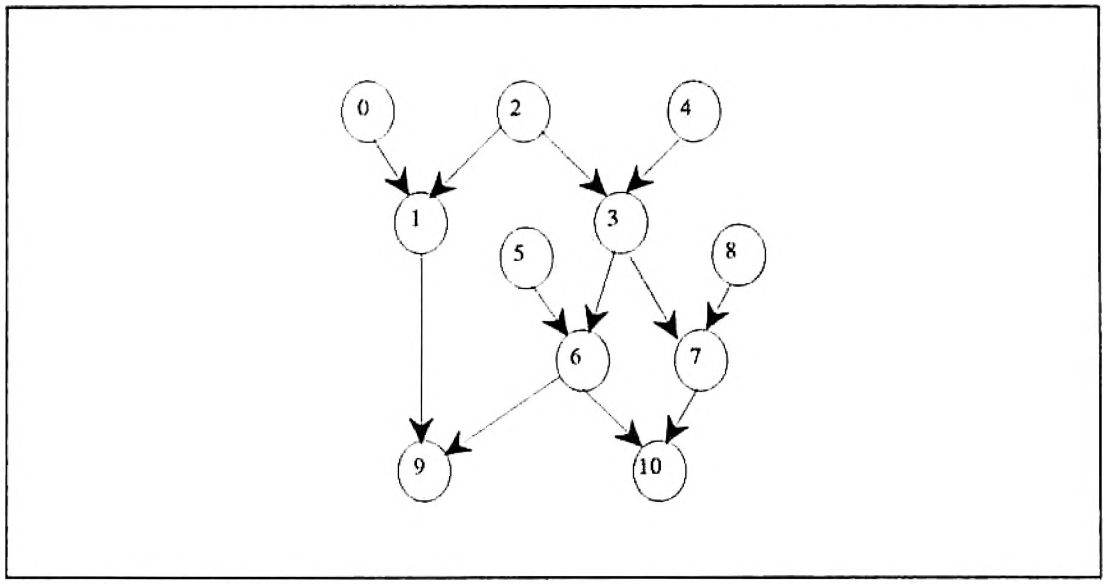


Figure 8. DAG with path length 3

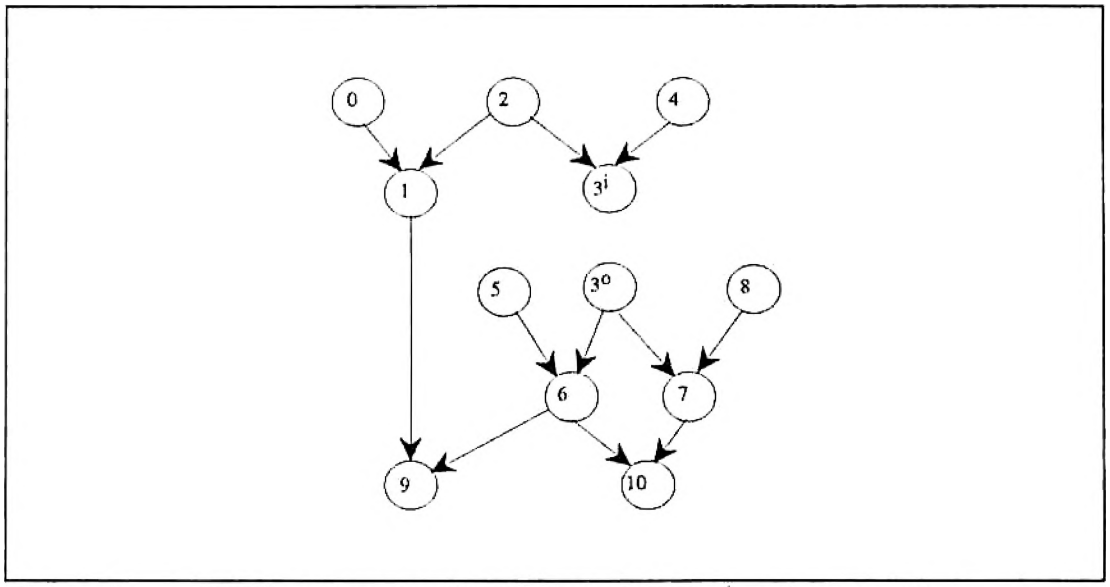


Figure 9. DAG with vertex 3 split into vertex 3^i and vertex 3^o

3^o.

A ***source vertex*** is a vertex with no edge coming into the vertex and a ***sink vertex*** is a vertex with no edge leaving the vertex. Note that splitting either source or sink vertices does not reduce the path length.

The ***DAG vertex splitting problem*** is to find the least cardinality vertex set $X \ni d(G/X) \leq \delta$, where δ is a pre-specified maximum delay. For the DAG of Figure 8 and $\delta = 2$, $X = \{3\}$ is a solution to the DVSP.

Lemma 1: Let $G = (V, E, w)$ be a weighted directed acyclic graph and let δ be a prespecified delay value. Let $MaxEdgeDelay = \max \{ w(i, j) \} \forall \langle i, j \rangle \in E$. Then the DVSP has a solution iff $\delta \geq MaxEdgeDelay$.

Proof: Vertex splitting does not eliminate any edges. So, there is no $X \ni d(G/X) < MaxEdgeDelay$. Further, $d(G/V) = MaxEdgeDelay$. So, for every $\delta \geq MaxEdgeDelay$, there is a least cardinality set $X \ni d(G/X) \leq \delta$.□

If $w(i, j) = 1 \ \forall \langle i, j \rangle \in E$ then the graph has unit weights. It has been proven by Sahni et.al. in [7] that finding a solution for DVSP is ***NP-hard*** for graphs with unit weights and with a $\delta \geq 2$. Since unit weight directed acyclic graphs are just a special case of weighted directed acyclic graphs these results also apply to the WDAG.

IV. THE GENETIC ALGORITHM FOR THE DVSP

This chapter describes the Genetic Algorithm that has been developed to solve the DAG Vertex Splitting Problem. It talks about the data structures that are used by the algorithm and gives a basic outline of the algorithm. This chapter also discusses how the crossover and mutation function have to be adapted in order to work on the algorithm.

A. OBJECTIVE AND CHROMOSOME ENCODING

The ***objective*** for the GA is to find a minimal set of vertices that split the graph such that the resulting graph has no path of length > δ.

The strings (chromosomes) in each individual of the population represent the ***set of splitting vertices*** (or ***split set*** for short) that are used to split the graph. Thus, the smaller the split set the better the solution. The term *split set* or *string* is used from here on instead of chromosome. Theoretically every vertex in the graph, excluding source and sink vertices, can be split and a vertex can only occur once in the split set. But some of these vertices might not be on a path whose length is greater than δ. Thus, it would be worthless to consider them as potential vertices to split. A new set is introduced, called ***potential vertices***, which contains only those vertices that are on paths whose length is greater than δ.

Lemma 2: After all the potential vertices are split, there cannot be any path in the graph whose length is $\geq \delta$.

Proof: Assume that there is such a path. Then, all the vertices on that path must be members of the potential vertex set. On the other hand, all the potential vertices are split and no potential vertex can be on a path $\geq \delta$. Therefore, this is a contradiction.□

B. GENERAL OUTLINE OF THE GENETIC ALGORITHM

The genetic algorithm starts the search with an initial string length and continues with multiple rounds of optimization. Each optimization round tries to find a feasible solution with a fixed size split set (string length), i.e. the vertices in the split set of at least one individual can split the graph in such a way that the resulting graph has a maximum path length which is less than or equal to δ. If a solution is found in a particular round, the next round attempts to shorten the string length and find a new solution with fewer number of vertices. Therefore the string length varies over time. Variable string length has been used before in GAs [23][24][25][26]. The genetic algorithm works only on one certain string length at a time. Different string lengths within a given population are not allowed simultaneously because it makes the GA more complex when applying the select and crossover functions. Basically, the algorithm works as follows:

1) try to find a suboptimal solution by splitting x vertices
2) if a suboptimal solution is found, reduce the number of vertices and try again

3) if no solution has been found within a certain number of generations, expand the number of vertices and try again.

The outline of the Genetic Algorithms for the DVSP is given in Figure 10.

```
Binary Approximation;
Create initial population;

while( !stop )
{
      Determine new string length;
      Reduce or Expand the string length;
      Evaluate population;

      for( # of generations )
      {
            Select individuals for reproduction;
            Crossover to create new offspring;
            Mutate offspring;
            Evaluate offspring;
            Recombine parents and offspring to create
new                   generation;
      }
}
```

Figure 10. Outline of the GA for the DVSP

C. BINARY APPROXIMATION

The Genetic Algorithm has to start the search for an optimal solution with an ***initial string length*** for the initial population. As discussed earlier, a solution to the DVSP exists, if all vertices in the potential vertex set are split. Thus, the initial string length to start the GA would be the cardinality of the set of potential vertices. Since this number might be far away from the global optimum, a method called ***Binary***

Approximation (BA) was developed, to narrow down the area around the global optimum in order to quickly find a better initial string length. The function works as follows:

```
Binary Approximation()
{
     while( upper > lower )
     {
          mid       = ( upper + lower ) / 2;
          found     = random_split( mid );
          if( found )
                    upper     = best_so_far  = mid;
          else
                    lower     = mid;
     }
     return( best_so_far );
}

random_split( mid )
{
     for( # of tries for BA )
     {
          select  mid  unique  vertices  from  the
                    potential vertex set;
          split the graph with the selected vertices;
          determine  the  longest  path  of  the  split
                    graph;
          if( longest path <= delta )
               return( TRUE );
     }
}
```

Figure 11. Pseudo code of *Binary Approximation*

The solution to the DVSP has to lay somewhere between splitting one vertex and splitting all potential vertices. Thus, a binary search is started midway between these two boundaries. A certain number of split sets (strings), determined by the parameter ***number of tries for BA***, are created randomly with a string length halfway between the lower and the upper bound. If a solution is found among these split sets, it is marked as a new upper bound for the BA. If no solution can be found then this is assumed to be a lower bound.

This process is repeated until the difference between the lower and the upper bound is less than one. After termination, the upper bound is assigned to the initial string length. Figure 11 shows the pseudo code for the Binary Approximation.

Experiments[1] have shown that the BA does an extremely good job in narrowing down the area around the global optimum, considering that it is a purely random algorithm. The initial string length returned by the BA is then used to create the initial population.

D. SELECT FUNCTION

The ***select*** function is a standard select using the roulette wheel. Instead of a linear search through the population, a binary search has been implemented which returns the index of the selected individual. Since the goal of optimization is to minimize the longest path in the graph by splitting vertices, the fitness function is defined to be $\frac{1}{LongestPath}$.

E. CROSSOVER FUNCTION

The uniform crossover [13][14] function is used to generate offspring from the parents. The uniform crossover was shown to outperform the one-point and two-point crossover in most cases [13][14]. While applying the uniform crossover, generation of

[1]More information about the experiments in Chapter 7.

multiple copies of the same vertex in a split set of the offspring must be avoided. Consider the following example illustrated in Figure 12 with two parents, a string length of 5, and a randomly generated crossover mask of 01110.

Parent 1:	15	7	8	19	2
Parent 2:	20	5	15	3	7
Crossover Mask:	0	1	1	1	0
Offspring 1:	*15*	5	*15*	3	2
Offspring 2:	20	7	8	19	7

Figure 12. Example of a crossover with duplicated vertices

After the crossover, offspring 1 ends up getting vertex 15 twice and Offspring 2 ends up getting vertex 7 twice. These vertices are called ***duplicated vertices***. Since this is not allowed to happen, as discussed earlier, this situation must be avoided. Thus, all the vertices that appear in both parents are not allowed to undergo crossover. To do this, the duplicated vertices have to be determined in both parents and separated from the remaining vertices by moving them to the end of the split set. This can be done because the order of the vertices in the split set does not change the outcome of the graph after splitting. After the duplicated vertices have been moved to the end of the split set, the uniform crossover can be performed without further changes among the remaining vertices. If these ideas are applied to the previous example, the result would be as shown in Figure 13:

The last two bits in the uniform mask are not really necessary because they do not have any effect on the offspring.

Parent 1:	8	19	2	15	7
Parent 2:	20	5	3	15	7
Crossover Mask:	0	1	1	1	0
Offspring 1:	8	5	3	15	7
Offspring 2:	20	19	2	15	7

Figure 13. Example of a crossover with separated duplicated vertices

Every applied crossover results in two or more offspring, depending on the parameter ***offspring per parents***. For more than two offspring per parents, different crossover mask have to be created, in order to avoid that parents create the same offspring over and over again. The offspring is placed into the population together with its parents which results in a temporary increase in the population. This population is reduced later by the ***recombination*** function. This way of creating a new generation was chosen to ensure that less fit offspring do not overwrite a more fit parent.

F. MUTATION FUNCTION

The ***mutation*** function operates only on the newly created offspring. Once a vertex has been chosen for mutation it is replaced by a new vertex picked from the potential vertex set that is not in the split set of the mutated individual. The parameter ***mutation rate*** determines the probability of mutating a certain vertex in the split set of all offspring.

G. RECOMBINATION FUNCTION

The ***recombination*** function takes the old population and the new offspring and reduces it down to the previous population size. The reduction is based upon the select function which ensures that fit individuals have a higher probability of survival into the new generation. This means that individuals of the old population can survive into the new population while new offspring may die depending on their fitness values. This method ensures that a highly fit parent does not get eliminated by a lower fit offspring. The recombination function also takes care of a certain variety in the new generation by making sure that no individual (parents and offspring) gets selected more than once for the new population. Figure Figure 14 illustrates the recombination function.

H. TAKE CARE OF ONES

It is possible that the DVSP has a solution by splitting only one vertex. This means that the string length is only one. If the crossover function is performed on individuals with a string length of one, no new individuals are introduced into the population. Only mutation can introduce new individuals. Since the mutation rate is very low, there exists a high probability of missing a solution with one vertex. A function called ***Take care of ones*** is used to eliminate this possibility. This function tries every vertex in the potential vertex set one at a time to check if there exists a solution. This function is used before the GA is started. If this function finds a solution by spitting only one vertex, then there is no need to start the GA.

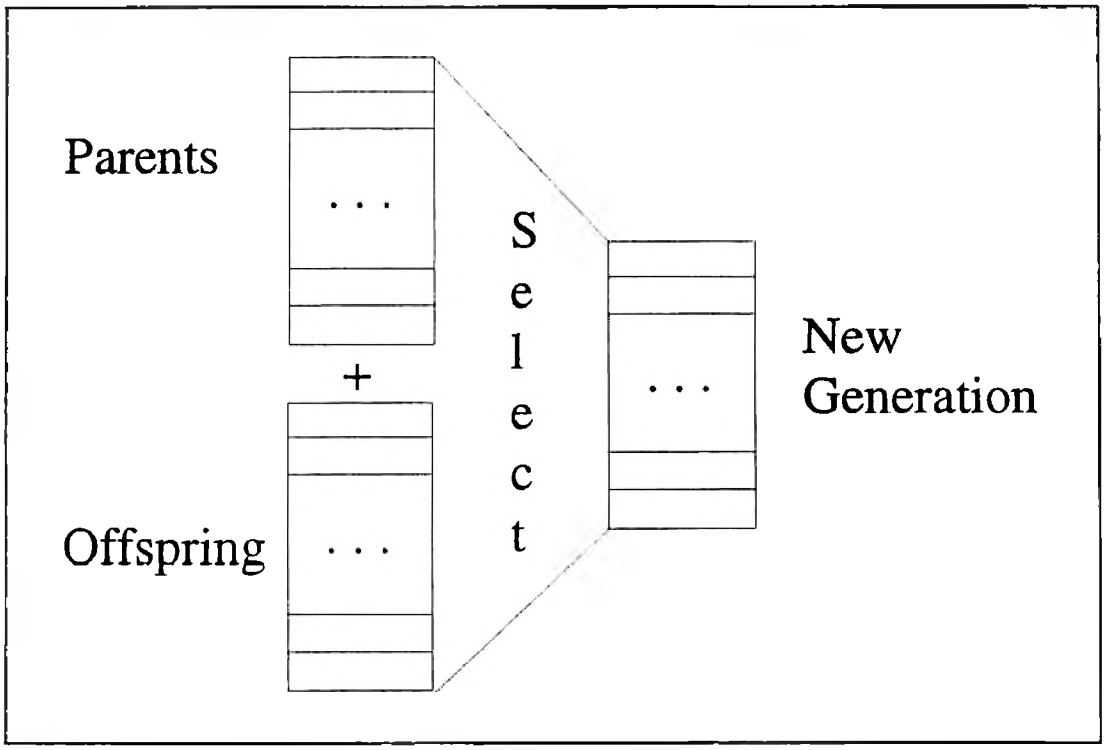


Figure 14. Illustration of the *recombination* function

I. SUMMARY

This chapter describes the Genetic Algorithm that has been developed to solve the DAG Vertex Splitting Problem. The strings (chromosomes) of each individual represent the vertices that are used to split the graph. This leads to a variable string length since the objective of the Genetic Algorithm is to find a minimal set of vertices that split the graph. Binary Approximation is a method to narrow down the search space around the global optimum. Some problems had to be solved in order to use the crossover and mutation function.

V. STRING LENGTH REDUCTION AND STEPPING

Why is a string length reduction necessary? The strings represent the vertices that are used to split the graph. The objective is to find a minimal set of vertices that split the graph. Thus, if the GA finds a suboptimal solution with splitting X vertices, it has to try to find a solution with splitting Y vertices, where $Y < X$.

Any string length reduction method must address the following two problems:

A) How to reduce the size of the split set?

B) What should be the next size of the split set?

Two strategies have been devised for each of the problems stated in A) and B) that are explained below.

A. STRING LENGTH REDUCTION METHODS

To address A), two different ***reduction methods*** were developed.

1. Preserve Duplicates. The first method is called ***Preserve Duplicates***. This method makes sure that the ***duplicated vertices*** in every individual do not get lost when the number of vertices is reduced. The intuition behind this strategy is that the duplicated vertices are important for getting a better fitness value since they have survived in multiple individuals throughout the evolution process. Recall that the duplicate vertices

are located at the end of the split set. Thus, to implement this strategy, the last vertices are moved to the beginning of the split set and the string length is reduced from the end of the split set leaving the duplicate vertices undeleted.

2. Random Delete. The second method deletes a number of vertices randomly out of the split set and is therefore called ***Random Delete***.

B. STEPPING METHODS

After a suboptimal solution has been found, the number of vertices in the split set has to be reduced. Thus, the question that emerges is "How many vertices have to be taken out of the strings for the next round of optimization?" In other words, how does the GA step through the search space? Two different, so called ***stepping methods*** were developed, to determine the string size that ought to be tried next by the GA.

1. Linear Stepping. The first method is called ***Linear Stepping***, which means that the string length is reduced by a positive integer every time a suboptimal solution is found. This integer is part of the parameter list and is called ***strlen decrement***. If no solution can be found with the reduced number of vertices the string length is incremented by one. The one new vertex that has to be introduced is taken from the set of potential vertices. The string length increment is performed until a new solution is found or until the string length exceeds the string length of the best solution found so far.

2. Multiple Binary Stepping. The second method is called ***Multiple Binary Stepping***. This method starts out with a lower bound of two (note that the one vertex case has already been tested) and an upper bound obtained from the BA. The new string length is always determined by the formula $\frac{UpperBound + LowerBound}{2}$. Every time a solution is found, the population that yielded this solution is saved and the upper bound is set to the current string length. If no solution has been found the lower bound is set to the current string length. It is important to note that the split set is always reduced from the latest saved population. This process repeats as long as a new solution with a smaller string length can be found.

Figure 15 gives an example of how the Multiple Binary Search works. The search space is visualized as an array where the indices represent the string length. Assume that the BA has returned an initial string length of seven. Thus, the upper bound is set to seven. The new string length is set to four and the GA is started. In the example, the GA cannot find a solution within a certain number of generations. Thus, the lower bound is set to current string length, which is four. The new string length is then set to five and the GA is started again. This time, a solution is found and the upper bound is set to five. The difference between the lower and the upper bound is only one and therefore the Multiple Binary Stepping is stopped. Since the best solution was reduced from seven to five, a second round is started. The lower bound is reset to two while the upper bound remains. A string length of three is tried by the GA which does not yield a solution. Then a string length of four is tried again, but again no solution can be found. Lower and upper

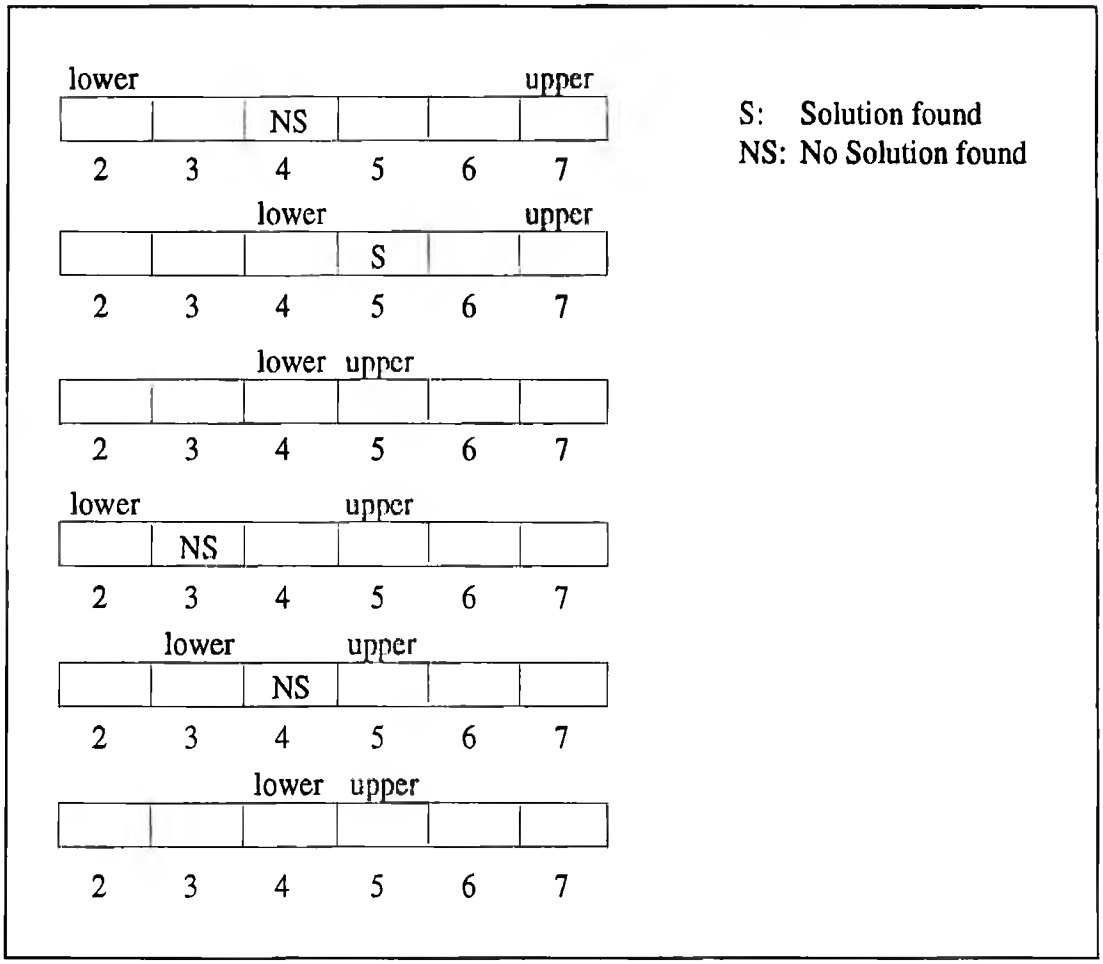


Figure 15. Example of a Multiple Binary Search

bound differ only by one and the Multiple Binary Stepping terminates. Since this round could not reduce the best solution, no new attempt is started to find better solutions.

C. SUMMARY

String length needs to be reduced in order to find better solutions. Two methods of how to reduce the string length are introduced. One method simply picks vertices out of the strings randomly, while the other method tries to prevent duplicated vertices from being deleted which might be of value for better solutions to the DAG Vertex Splitting

Problem in the future. The second question that is addressed in this chapter is how many vertices have to be deleted for the next optimization round. Two techniques were developed that "step" through the search space. One technique reduces the string length by a positive integer. The other technique uses a Multiple Binary Stepping method to explore the search space.

VI. THE PARALLEL GENETIC ALGORITHM

This chapter describes the Parallel Genetic Algorithm (PGA) derived from the sequential Genetic Algorithm to solve the DAG Vertex Splitting Problem. It has been implemented on an iPSC/2 Intel Hypercube with a maximum of 16 processors. The PGA has been developed such that it can run on any number of processors. Section A gives an outline of how the algorithm works. Additional functions are used to exchange information about the state of the progress. The functions are ***exchange initial string length***, ***determine new string length***, ***remote solution***, ***adjust boundaries***, and ***exchange individuals*** and are described in the Sections B through F respectively. Section G summarizes the Parallel Genetic Algorithm.

A. OUTLINE OF THE PARALLEL GENETIC ALGORITHM

The Parallel Genetic Algorithm has been developed such that it can run on any number of processors ≥ 2. The basic idea behind the Parallel Genetic Algorithm is to distribute the available processors over the search space and let each processor operate on a subpopulation of the total population. In other words, every processor operates on a different string length. The outline of the Parallel Genetic Algorithm is given in Figure 16. Every processor starts out by computing the initial string length for the initial population using the ***Binary Approximation*** described in Chapter IV. Then each processor tries to find a solution with just splitting one vertex using the ***Take care of ones*** function. After these two initial steps the processors synchronize and exchange information to find out the minimal initial string length. This function is described in the next section. With

```
Binary Approximation;
Take care of ones;
Exchange initial string length;
create initial population;
save population as best population;

while( upper >= lower )
{
     determine new string length;
     shrink string length from best population to new
               string length;
     evaluate population;

     for( # of generations )
     {
          if( ( generation % alone ) == alone-1 )
          {
               if( remote solution )
                    break;
               exchange individuals;
          }

          Select individuals from subpopulation;
          Crossover to create new offspring;
          Mutate offspring;
          Evaluate offspring;
          if( local solution in offspring )
          {
               save population as best population;
               break;
          }

          Recombine parents and offspring to create
                    new generation;
     }

     if( ! remote solution )
          exist remote solution;

     adjust boundaries;
}
```

Figure 16. Outline of the Parallel Genetic Algorithm for the DVSP

the minimal initial string length, the processors create their initial subpopulation and save it as the currently best population. The lower bound is set to two and the upper bound of

the search space is set to the initial string length. The termination condition for the while loop is determined by the virtual boundaries of the search space. These boundaries are adjusted by the function ***adjust boundaries*** which is discussed in Section G. After entering the while loop every processor uses a new string length. This assignment of the new string length to each processor is fully distributed without any master processor being involved and in a way such that the processors are equally distributed over the search space. Section C describes this algorithm. Every processor reduces the string length in the current population from the best population to the new assigned string length. With the experience gained from the experiments [16] performed on the sequential GA, the ***Random Delete*** reduction method is used to delete vertices out of the split set in the Parallel Genetic Algorithm. The generation loop is similar to the generation loop in the sequential Genetic Algorithm. The only difference is the added communication. The communication is twofold: (1) involves finding out about solutions on other processors. (2) involves exchanging "good" individuals with another processor by replacing "bad" individuals with "good" individuals. The processors spend a certain number of generations evolving the population, determined by the parameter ***alone***. Then the processors synchronize in order to find out, whether another processor has already found a solution. This takes place in the function ***remote solution*** which is explained in Section D. If a remote solution exists, the generation loop is terminated and the upper bound of the search space is adapted to the string length of the processor that found the solution. If more than one processor has found a solution then the minimum of their string lengths is taken for the new upper bound of the search space. In case of no solution, the processors exchange a number of their best individuals. The number of individuals chosen

for exchange is determined by the parameter ***exchange***. A closer look at this function is taken in Section F. If no solution has been found within a certain number of generations, determined by the parameter ***number of generations***, the search space is virtually reduced by increasing the lower bound. This takes place in the function ***adjust boundaries*** which is described in Section G.

B. EXCHANGE INITIAL STRING LENGTH

This function is used to determine the smallest initial string length obtained in all the processors after applying ***Binary Approximation*** and ***Take care of ones***. One of the processors is assigned to be the ***root*** processor. The root processor has to receive the initial string length from all the other processors, determine the smallest string length among the received string lengths, and send the smallest initial string length back to all the processors. All the other processors simply send their initial string length to the root processor and wait for the root to reply back with the smallest initial string length. The pseudo code for this function is given in Figure 17.

C. DETERMINE NEW STRING LENGTH

This functions determines the string length of each processor such that all processors are equally distributed over the search space. The search space can again be viewed as an array with a lower and an upper bound. The pseudo code for this function can be seen in Figure 18.

```
if( ROOT )
{
     for all processors
     {
          receive initial string length;
          if( received value < minimum )
               minimum = received value;
     }
     send minimum to all processors;
}
else
{
     send initial string length to ROOT;
     receive smallest string length;
}
```

Figure 17. Pseudo code of the function *exchange initial string length*

```
diff       = upper - lower;
step       = diff / number of processors;
diff++;
if( step == 0 )
     new_strlen = (upper-node%diff);
else
     new_strlen = (upper-(((node+1)%diff)*step);
```

Figure 18. Pseudo code of the function *determine new string length*

If the difference between the upper and lower bound of the search space is less than the number of processors, then multiple processors are assigned the same string length.

D. REMOTE SOLUTION

This function is used by the processors to determine whether one of the processors has found a solution or not. The function basically works like the ***exchange initial string***

length. One processor is assigned to be the ***root*** processor which receives all the data from the other processors. If one processor has found a solution it sends a TRUE value to the ***root*** along with the string length with which the solution has been found. The ***root*** determines the smallest string length, considering the results received from all the processors and returns this back to the other processors.

E. ADJUST BOUNDARIES

This function sets the upper and lower bounds of the search space depending on whether solutions have been found within a certain number of generations or not. If a solution has been found, the upper limit of the search space is set to the smallest string length that has found the solution. If none of the tested string lengths yielded a solution, the upper limit remains the same, while the lower limit is increased according to the following equation: $lower = \frac{lower + upper}{2} + 1$. This virtually reduces the search space in order to group the processors more around an area where a solution is more likely to exist. This also allows some processors to work on the same string length when the difference between the upper and lower bound is less than the number of processors. The lower limit is continued to be increased if no solution is found in the following optimization rounds. Once the lower limit is higher than the upper limit the PGA is stopped. The lower limit is reset to two if a solution is found.

F. EXCHANGE INDIVIDUALS

This function is used to exchange a number of individuals, determined by the parameter ***exchange***. A different exchange strategy than described in Tanese [19] is used. In a *d*-dimensional Hypercube architecture every processor has *d* neighbors. In Tanese's model, every processor exchanges its best individuals with one of its neighbors. The communication partner changes with every exchange. This exchange scheme does not work in this model because every processor operates on a different string length. Only when the difference between the upper and lower bound of the search space is smaller than the number of available processors, multiple processors are assigned to the same string length. In that case an exchange can take place. Tanese's exchange model uses pairs of neighbors to exchange individuals. This also does not work in this model because there might be an odd number of processors working on one string length which does not allow to create pairs. Therefore a ***ring exchange*** model is used. For example, assume that processor 0, 6, and 12 are assigned to work on the same string length. When the processors exchange individuals, processor 0 sends its individuals to processor 6, processor 6 sends its individuals to processor 12 and processor 12 sends its individuals to processor 0. The ring exchange model has one disadvantage: it does not use ***nearest neighbor communication*** which should be a goal for programs that run on hypercube architectures. Nearest neighbor communication only involves sending messages between processors that are one distance away from each other. In order to achieve nearest neighbor communication between processors that are assigned the same string length, a more complicated distribution strategy has to be used.

The "good" individuals in each subpopulation are selected probabilistically using the ***select*** function. In order to avoid communication overhead by sending every individual in a separate send command, the split sets of every selected individual is copied into a single array which is then send. The fitness values of the selected individuals are also copied into an array and send separately from the split set array. After the data has been exchanged, the "bad" individuals in the subpopulation have to be determined in order to replace them by the received individuals. To select the "bad" individuals in the subpopulation the ***inverse select*** function has to be used. The inverse select function uses the inverse of the fitness value which gives individuals with a bad fitness value a higher probability of getting selected. Figure 19 lists the pseudo code for the function ***exchange individual***.

```
determine whether other processors work on the same
          string length;
if so, determine processor to which to send data;
for( # of individuals to exchange )
{
     select "good" individuals;
     compile split set and fitness value of selected
          individual into send arrays;
}
send compiled arrays;
receive compiled arrays;
for( # of individuals to exchange )
{
     select "bad" individuals;
     replace split set and fitness value of selected
          individual from received arrays;
}
```

Figure 19. Pseudo code of the function *exchange individuals*

G. SUMMARY

This section summarizes the Parallel Genetic Algorithm described in this chapter. The PGA to solve the DAG Vertex Splitting Problem uses a somewhat different approach to parallelize the Genetic Algorithm. It uses subpopulations for the processors as described in [19][18][20]. The main difference is that the processors are evenly distributed over the search space, by assigning each processor a different string length. Processors usually exchange their best individuals with other processors in order to simulate a bigger total population. This approach cannot easily be used in the described PGA. An exchange of individuals can only occur if processors are assigned to the same string length. But even then a pairwise exchange of individuals is not possible because an odd number of processors might be assigned to the same string length. Thus, a so called ***ring exchange*** is used where all the processors working on the same string length exchange individuals in a ring fashion. The fact that the ring exchange does not take advantage of nearest neighbor communication has minor affect on the global run time of the PGA because nearest neighbor communication is only slightly faster than the logical ring. Also, the ring exchange does not occur very often.

VII. EXPERIMENTAL RESULTS

This chapter describes the experiments that were conducted to test the behavior of the parallel and sequential Genetic Algorithms. Section A describes the graphs that were used to run the experiments. The size and shape of the ***search space*** is discussed in Section B. Experiments were run to see how the parameter ***number of tries for BA*** influences the initial string length and the run time. The results of these experiments are reported in Section C. Another set of experiments, reported in Section D, was run to find out which combination of string length reduction method and stepping technique works best. Two sets of experiments were run for the Parallel Genetic Algorithm. One experiment was set up to find out the influence of the parameter ***alone***, which defines the frequency of synchronization between the processors. The results of this experiment are described in Section E. Section F discusses the speed-up experiment which helps to find out whether more processors reduce the run time of the program. A summary and general observations are given in Section G.

A. THE GRAPHS

The graphs used in the experiments for testing the Genetic Algorithms (parallel and sequential) were derived from the ISCAS-85 benchmark combinational circuits [27]. The vertices in the DAG model represent the gates in the circuit and the edges correspond to connections between gates. The delay (weight) for all the edges was set to one. Some characteristics of these circuits (DAGs) are given in Table II.

Table II. CIRCUIT CHARACTERISTICS OF ISCAS-85 COMBINATIONAL BENCHMARKS

Name of Graphs	# of vertices	# of edges	maximal degree	longest path
C17	11	12	2	3
C432	196	336	9	17
C880	443	729	8	24
C1355	587	1064	12	24
C2670	1502	2076	11	32
C3540	1719	2939	16	47
C5315	2485	4386	15	49
C6288	2448	4800	16	124
C7552	3720	6144	15	43

B. THE SEARCH SPACE

Assume that the number of vertices in the potential vertex set is n. Thus, there are

$$\binom{n}{x} = \frac{n!}{x! \cdot (n - x)!} \tag{9}$$

possible combinations for picking x vertices from n potential vertices. Thus, the total size of the search space is

$$TSS(n) = \sum_{i=1}^{n} \binom{n}{i} \tag{10}$$

According to the Binomial Theorem

$$\sum_{j=0}^{n} \binom{n}{j} \cdot x^j \cdot y^{n-j} = (x + y)^n \tag{11}$$

the total search space for a graph with n potential vertices is

$$TSS(n) = \sum_{i=1}^{n} \binom{n}{i} = \sum_{j=0}^{n} \binom{n}{j} - 1 = \sum_{j=0}^{n} (1^j \cdot 1^{n-j} \cdot \binom{n}{j}) - 1 = (1+1)^n - 1 = 2^n - 1 \tag{12}$$

Every graph was tested with a certain maximum delay δ. Table III summarizes some statistics for the graphs for different δ, the resulting potential number of vertices and the total size of the search space.

Table III. POTENTIAL NUMBER OF VERTICES AND TOTAL SIZE OF THE SEARCH SPACE FOR THE BENCHMARK GRAPHS

Graph	delta	# of potential vertices	total search space
C432	5	153	$1.142 \cdot 10^{46}$
C880	10	308	$5.215 \cdot 10^{92}$
C1355	15	514	$5.363 \cdot 10^{154}$
C2670	20	636	$2.852 \cdot 10^{191}$
C3540	30	966	$6.237 \cdot 10^{290}$
C5315	35	522	$1.373 \cdot 10^{157}$
C6288	49	2252	$8.309 \cdot 10^{677}$
C7552	22	1196	$1.076 \cdot 10^{360}$

Figure 20 shows the shape of the search space for graph C432. The thin line in Figure 20 represents the number of possible combinations of selecting X vertices out of the total

number of possible vertices without replacement, which is 153 vertices for the graph C432 and $\delta = 5$. The tick line represents the cumulated number of possible solutions with less than or equal to X vertices. All the other graphs behave in the same way.

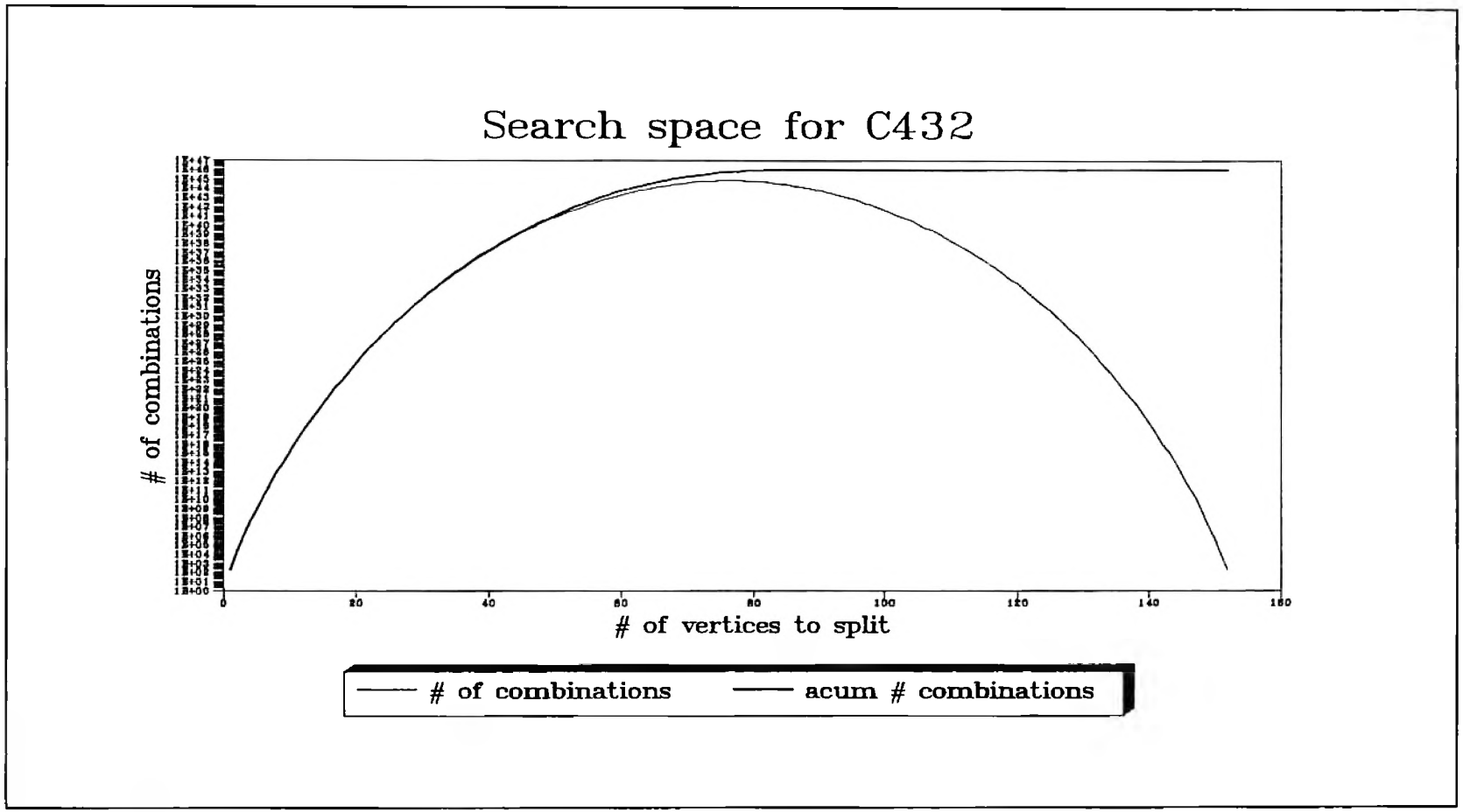


Figure 20. Shape of the search space for graph C432

C. BINARY APPROXIMATION

The purpose of the function Binary Approximation is to determine the ***initial string length*** for the initial population. The function has one parameter ***number of tries for BA***. Experiments were conducted to see how the *initial string length* changes when the parameter *number of tries for BA* changes. The following parameter settings were used:

number of tries for BA = 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500.

Each run was repeated 20 times to get a good average. The tests were run on all graphs with the δ listed in Table III. All the experiments exhibited a similar behavior: a fast jump down to a small *initial string length*, when the *number of tries for BA* = 10 and only smaller reductions in the *initial string length* when the *number of tries for BA* is higher. The run time for the Binary Approximation increases linear with a linear increase in the *number of tries for BA*. Figure 21, Figure 22, and Figure 23 show the results for the graphs C432, C880, and C1355 respectively. The five remaining graphs, C2670, C3540, C5315, C6288, and C7552 are listed in Appendix A. Figure 24 shows the percentage of the initial string length reduction for the first three graphs. The percentage of reduction can be computed according to the following formula:

$$100 - \left(\frac{\textit{initial string length by BA}}{\textit{total number of potential vertices}}\right) \cdot 100 \qquad (13)$$

Appendix B shows the reduction plot for the remaining five graphs.

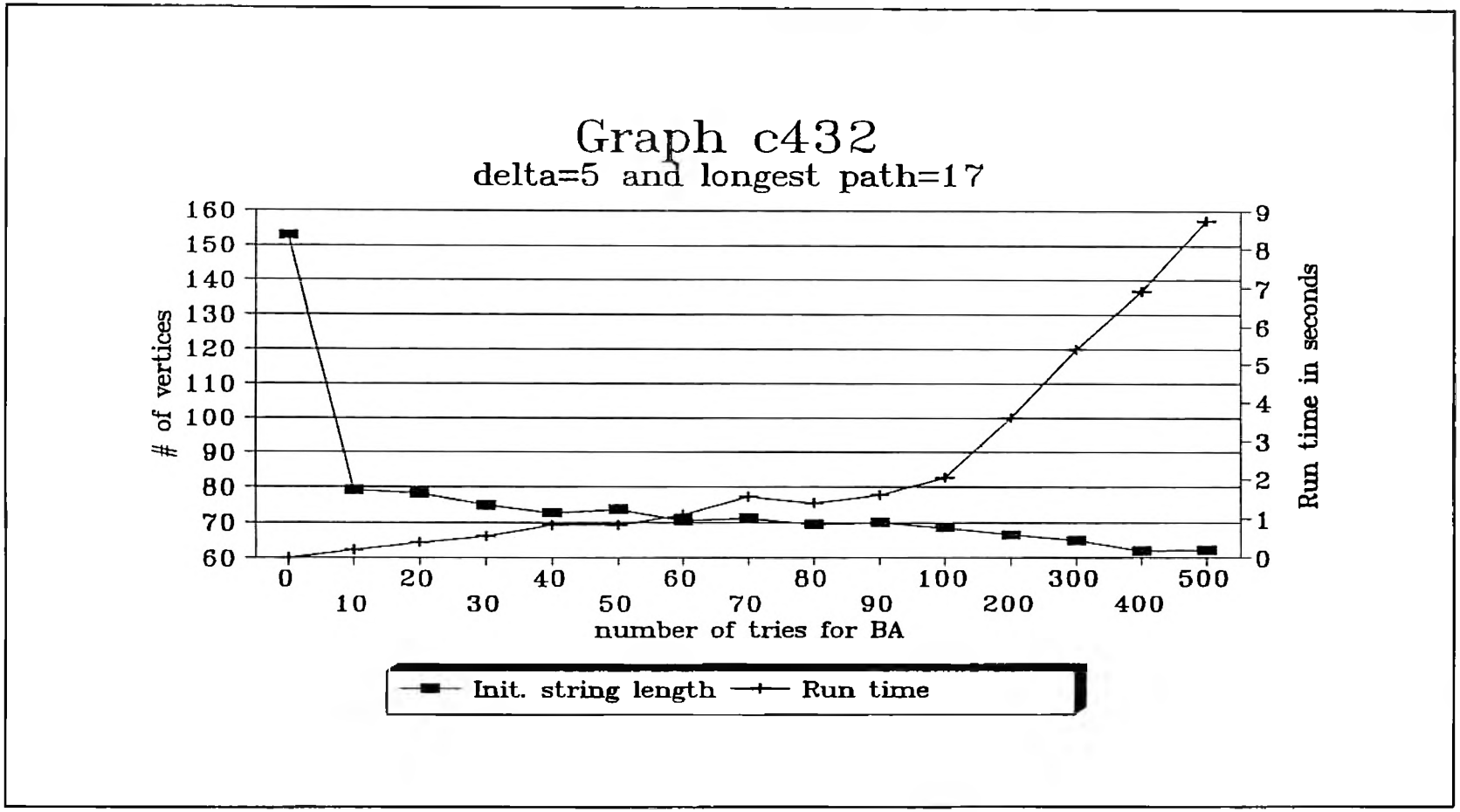


Figure 21. Initial string length reduction with Binary Approximation on graph C432

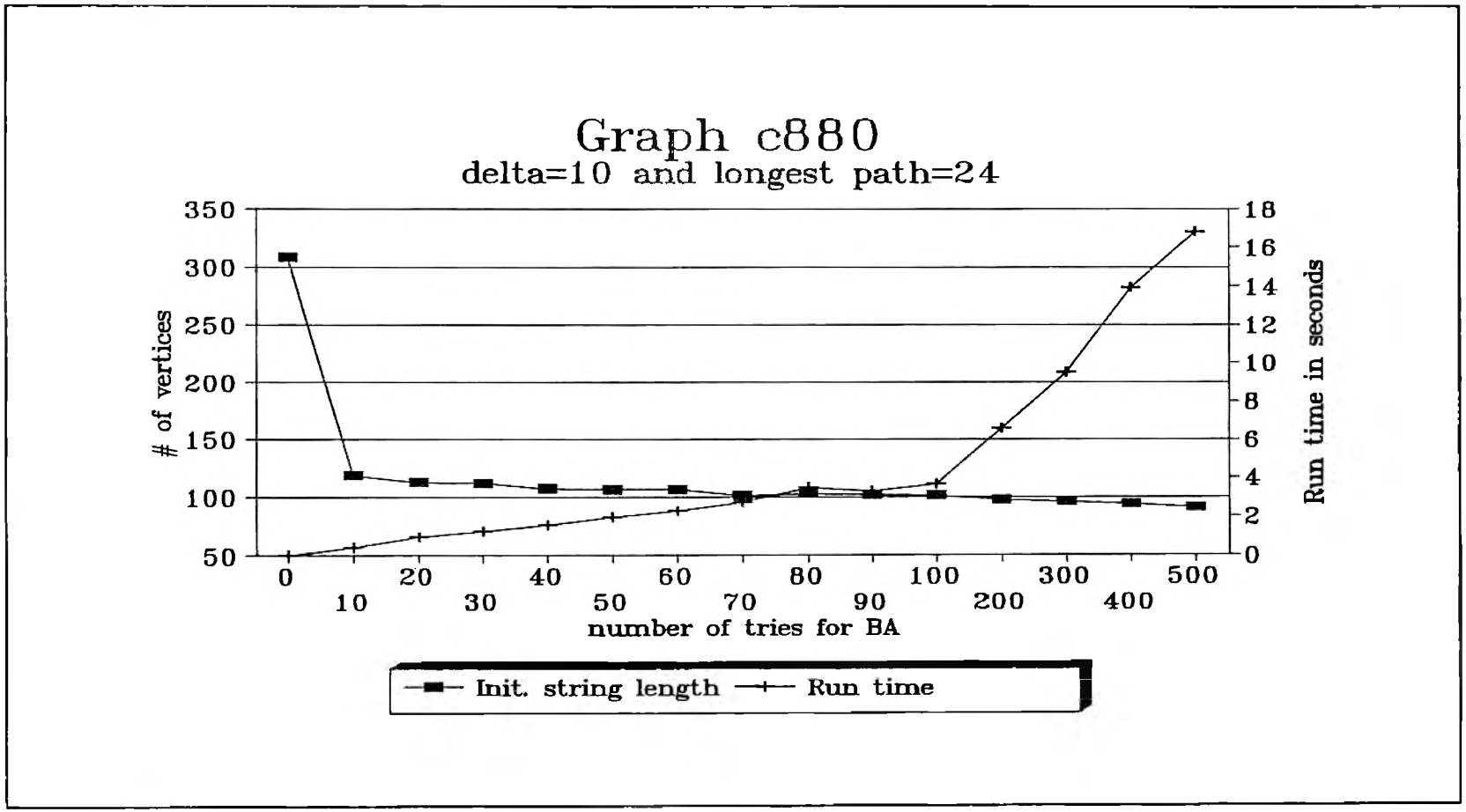


Figure 22. Initial string length reduction with Binary Approximation on graph C880

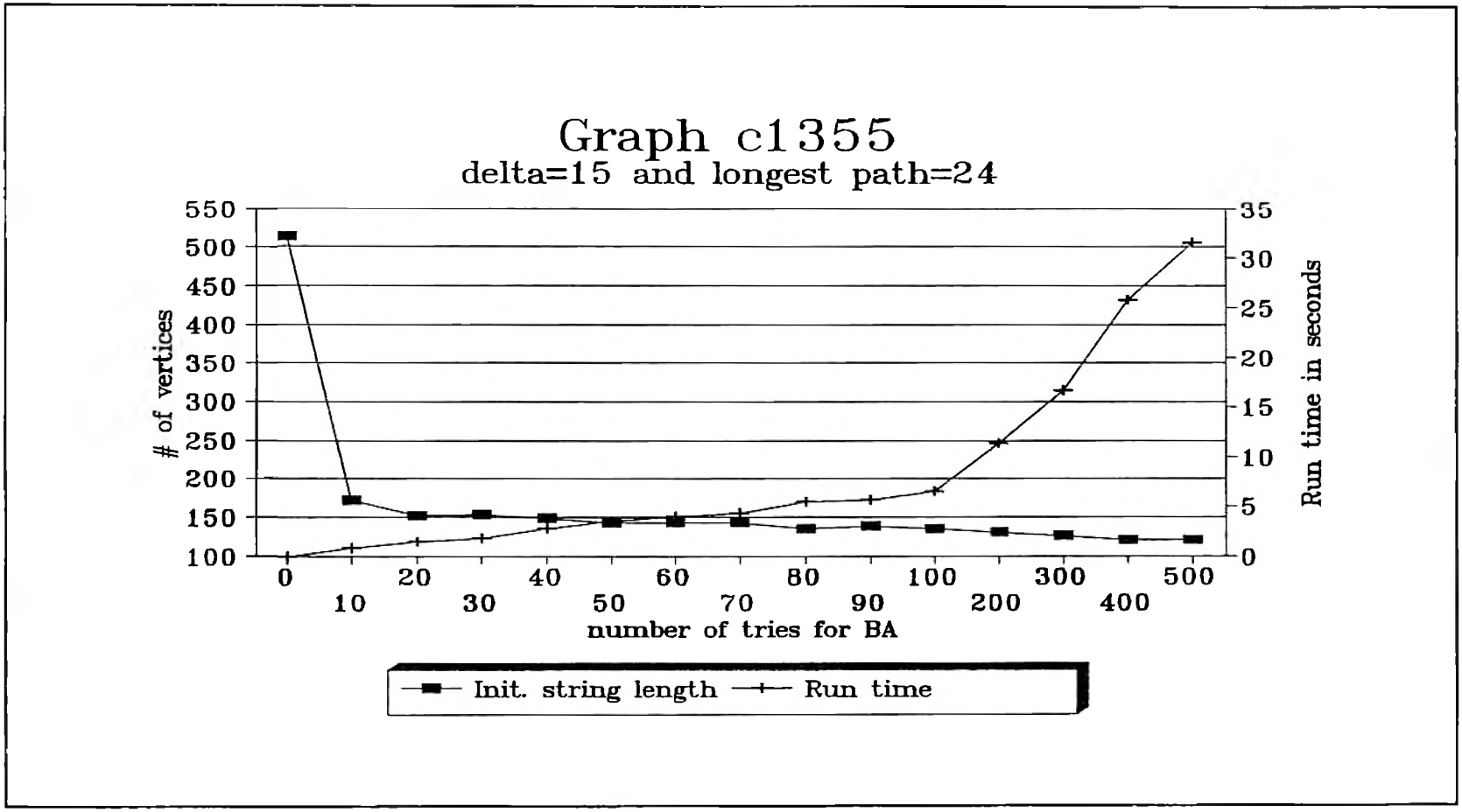


Figure 23. Initial string length reduction with Binary Approximation on graph C1355

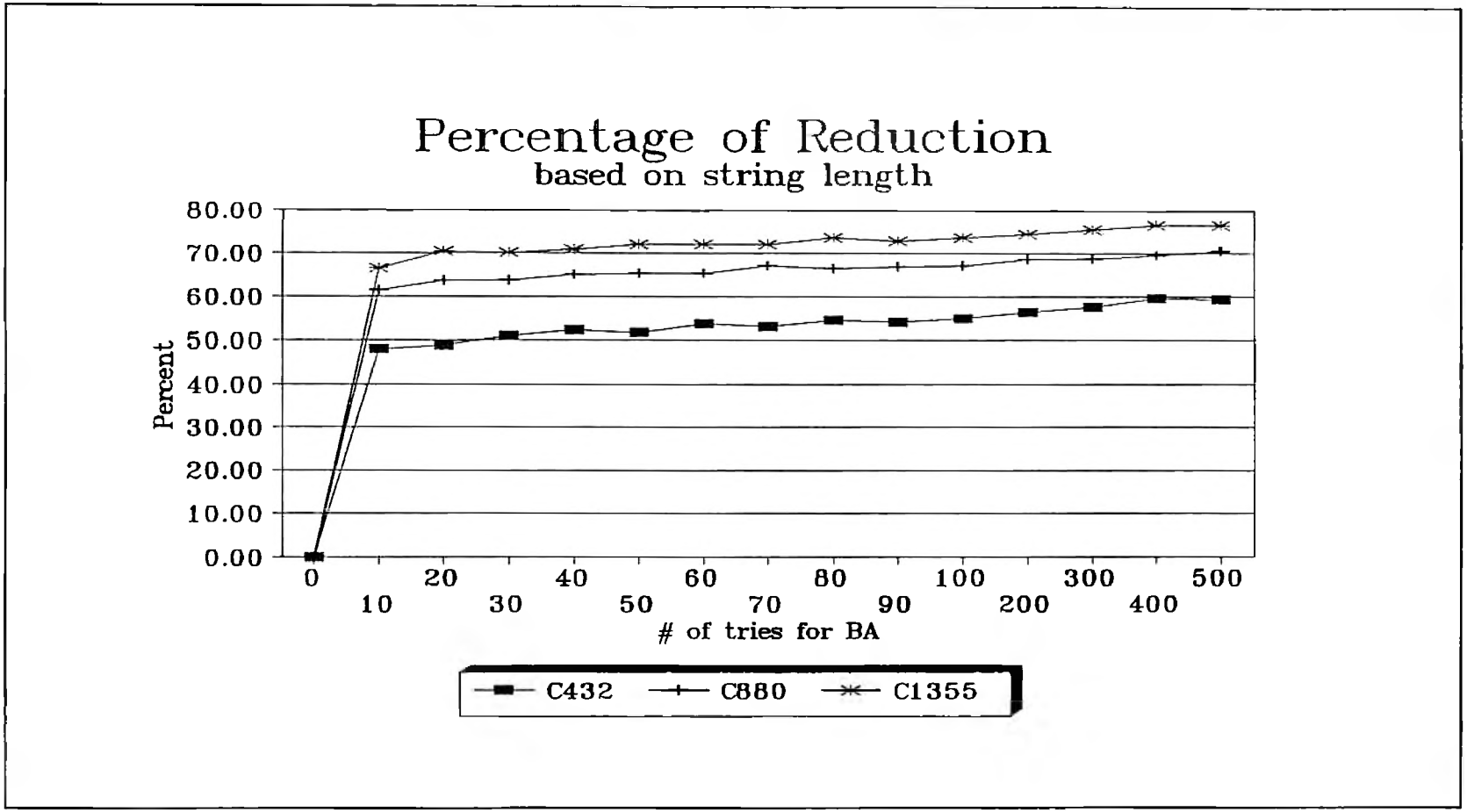


Figure 24. Percentage of reduction in string length for graphs C432, C880, and C1355

How big is the remaining search space after applying the Binary Approximation? The remaining search space (RSS) for a graph with n potential vertices and an initial string length of k is

$$RSS(k, n) = \sum_{i=1}^{k} \binom{n}{i} \tag{14}$$

Tables, listing the Binomial Cumulative Distribution Function [28] can be used to calculate the remaining search space. The Binomial Cumulative Distribution Function is defined as follows:

$$B(k, n, p) = \sum_{i=0}^{k} \binom{n}{i} \cdot p^{i} \cdot (1 - p)^{n-i} \tag{15}$$

Using a probability $p = 0.5$ Equation (15) can be rewritten as follows:

$$B(k,n,0.5) = \sum_{i=0}^{k} \binom{n}{i} \cdot (0.5)^{i} \cdot (0.5)^{n-i} = \sum_{i=0}^{k} \binom{n}{i} \cdot (0.5)^{n} \tag{16}$$

Thus, the remaining search space can be computed using the following equation:

$$\sum_{i=1}^{k} \binom{n}{i} = \sum_{i=0}^{k} \binom{n}{i} - 1 = B(k, n, 0.5) \cdot 2^{n} - 1 \tag{17}$$

Since $2^n - 1$ is the total search space, the percentage of reduction of the search space is equal to $(1 - B(k, n, 0.5)) \cdot 100$ for an initial string length k and n potential vertices.

But there exists one problem when these distribution tables are used to calculate the remaining search space. They are only accurate until the 5th or 6th position after the decimal point. This accuracy is not enough. For example,

$B(35, 153, 0.5) = 0$ [28]. Therefore, the remaining search space would be 0 according to Equation (17). This is not true. The remaining search space is $5.836396 \cdot 10^{34}$ using Equation (14). The distribution table for $B(35, 153, 0.5)$ should have listed a value of

$$\frac{RSS(35, 153)}{TSS(153)} = \frac{5.836396 \cdot 10^{34}}{2^{153}} = 5.111583 \cdot 10^{-12} \qquad (18)$$

but an accuracy of 5 or 6 digits after the decimal point truncates the important digits. A fast way to approximate the remaining search space is to simply calculate $\binom{n}{k}$, $if\ k \leq \frac{n}{2}$, where k is the initial string length and n is the number of potential vertices.

Summarizing the results, it can be concluded that a small *number of tries for BA* result in the best ratio of reduction verses run time and that the reduction in the string length can be up to 90%. It is also true that the remaining search space, even after applying *Binary Approximation*, is still very large.

D. STRING LENGTH REDUCTION AND STEPPING

The intention behind this experiment was to find out which string length reduction method/stepping technique performs better. Therefore combinations of the four methods

described in Chapter V were implemented on a NeXTstation using C. The experiments were divided into four Test Beds (TBs):

- TB 1: Preserve Duplicates and Linear Stepping
- TB 2: Preserve Duplicates and Multiple Binary Stepping
- TB 3: Random Delete and Linear Stepping
- TB 4: Random Delete and Multiple Binary Stepping

Three graphs, C432, C880, and C1355 were selected for the tests. The following parameter settings were used in various combinations [29]:

crossover rate:	0.5, 0.6, 0.9
mutation rate:	0.001, 0.005
# of generations:	100, 200, 1000
population size:	50, 100, 200, 400, 800
offspring per parents:	2, 4, 8
strlen decrement:	3

The tests showed that Test Bed 4 and 2 yielded the best solutions on the average. This is due to the fact that both Test Beds use the ***Multiple Binary Stepping*** which does a more extensive search than Linear Stepping. Because of this more extensive search it also needs more computation time than Linear Stepping. Table IV shows a ranking of all four Test Beds with respect to solution quality and run time. The solutions and run times were averaged over multiple runs.

The results also indicate that the method ***Preserve Duplicates*** (TB 1 and TB 3) does not help much in finding better solutions. It mostly gets trapped in a local optima.

Table IV. RANKING OF FOUR GENETIC ALGORITHMS USING DIFFERENT STRING LENGTH REDUCTION AND STEPPING TECHNIQUES

Test Beds	Solution Quality	Run Time
Test Bed 1	4	1
Test Bed 2	2	3
Test Bed 3	3	2
Test Bed 4	1	3

E. PARAMETER "ALONE"

This experiment was conducted to find out how the parameter ***alone*** influences solution quality and run time for the Parallel Genetic Algorithm. The PGA was implemented on an Intel iPSC/2 Hypercube, which is a distributed memory MIMD (Multiple Instruction Multiple Data) machine. The parameter ***alone*** determines the frequency of synchronization and exchange of individuals between the processors. Two Parallel Genetic Algorithms were used for this experiment.

1) The Parallel Genetic Algorithm as described in Chapter VI, but ***with*** an included full distribution of the best population after the generation loop. The processor which found the best solution distributes its local subpopulation to the other processors. This strategy is called as ***PGA with best distribution*** (PGA-with).

2) The Parallel Genetic Algorithm as described in Chapter VI, ***without*** the full distribution of the best population. This is called ***PGA without best distribution*** (PGA-without).

The two PGA's were run using two different *subpopulation* sizes, 50 and 150 on 16 processors. The parameter ***alone*** was set to 10, 20, 50, and 100. Crossover rate and mutation rate were set to 0.6 and 0.005 respectively. The number of individuals that are exchanged between the processors was always set to be 10% of the subpopulation size. The run time and the *best found solution so far* were recorded in each round and the average of six runs was taken. The experiment was run on graph C432 and the initial string length was set to 80, in order to have a unique starting point for the Genetic Algorithm. Eighty for the initial string length was chosen because the Binary Approximation would have returned this value on the average if the parameter *number of tries for BA* is set to 10^2. Figure 25 and Figure 26 show the results of the experiments with the PGA-***without*** and PGA-***with*** on a subpopulation size 50. Both graphs behave about the same: in the beginning of the run better solutions with fewer number of vertices are found very fast. A ***break*** or ***disruption*** in the graph means that a solution was not found within one generation loop. Therefore, the search space is virtually reduced, as described in the previous chapter, and a new generation loop is started. A disruption can be seen in both graphs with *alone* = 10. Looking at just the best solutions found, both PGAs and an *alone* of 10 results in solutions which are worse than the other alone settings. This is due to the fact that both graphs with *alone* = 10 have a disruption earlier

[2]See experiment with Binary Approximation in this chaper.

in the test run. Thus, it can be concluded that a disruption is not good for finding good solutions. Another reason, why *alone* = 10 finds worse solutions than the others, is the frequent test among the processors for checking if a solution has been found. Therefore less time is given to the processors to evolve the population which can result in a *premature convergence.*

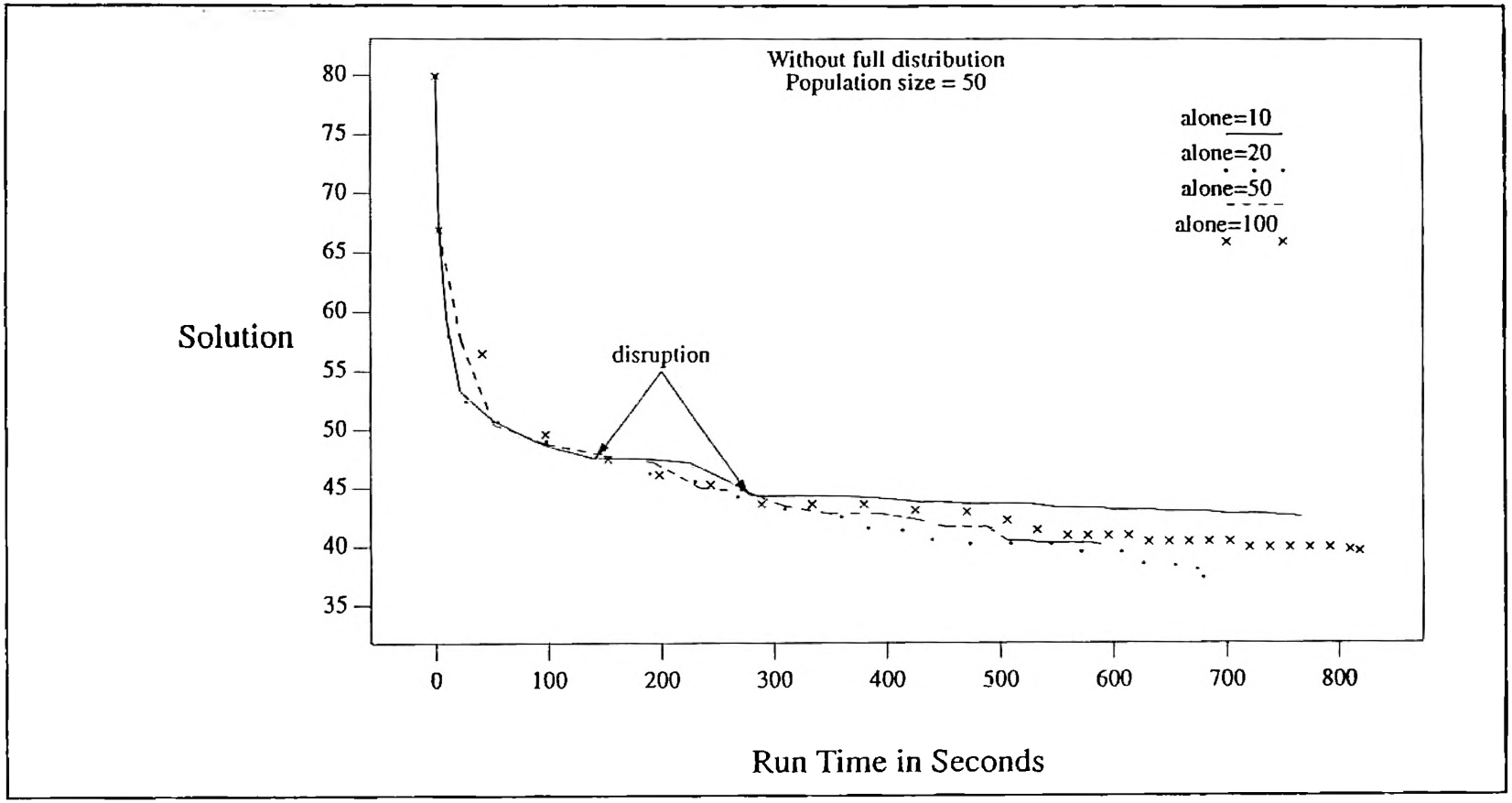


Figure 25. Test run on PGA *without* full distribution and subpopulation size 50

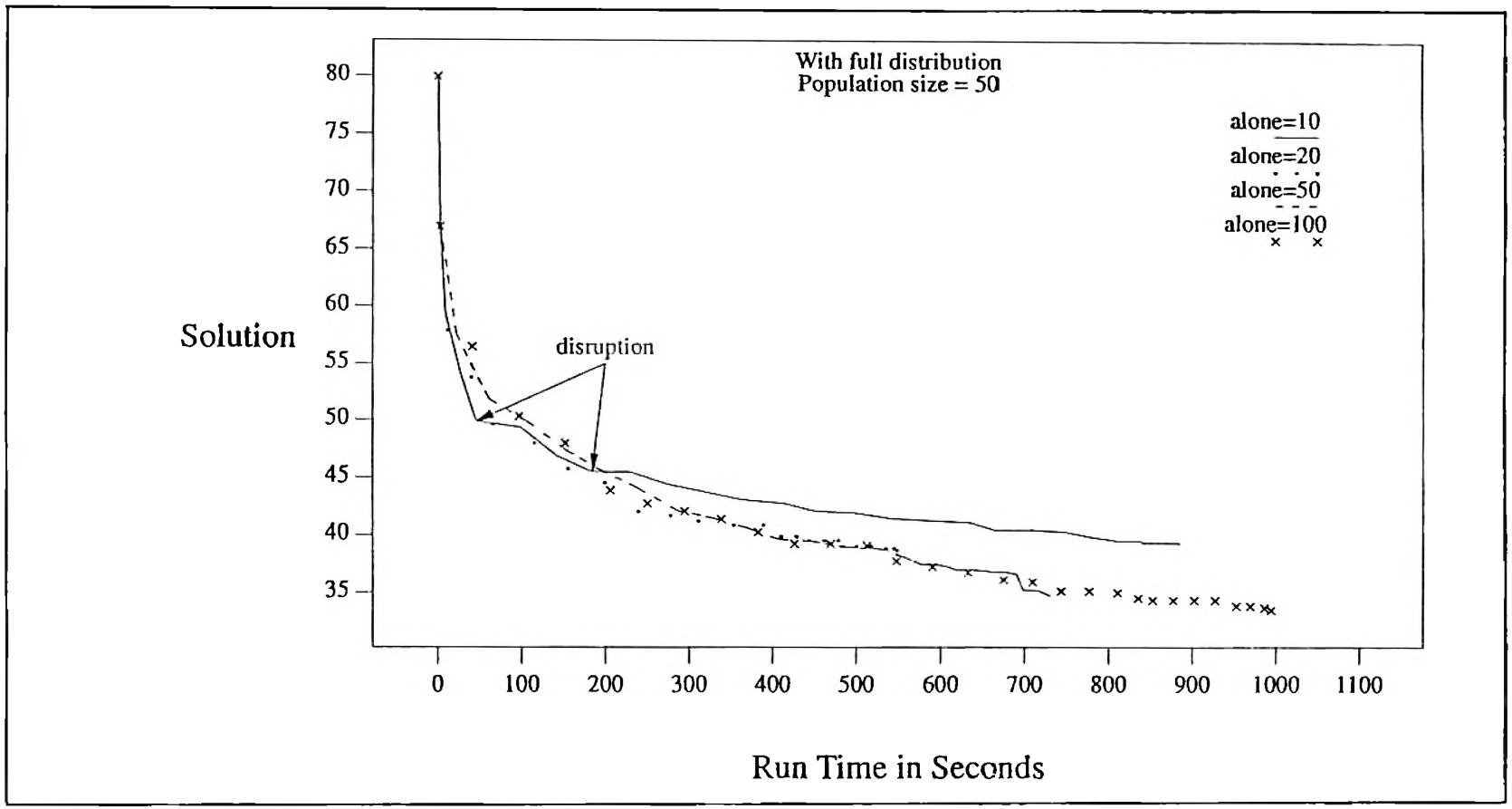


Figure 26. Test run on PGA *with* full distribution on subpopulation size 50

Figure 27 and Figure 28 show the plots with 150 individuals in each subpopulation for the PGA-*without* and PGA-***with***. Again it can be observed that better solutions are found very fast in the beginning. If *alone* is set to 10, the solutions found are again worse than the other alone settings in the PGA-*without*. However, the PGA-*with* and 150 individuals in each processors subpopulation finds the best solutions with *alone* set to 10. The explanation for this results is again the disruption in the *without* PGA. The *with* PGA also shows a disruption with *alone* being 10, but this disruption is not as strong as the one for the PGA-*without*. Therefore, the PGA-*with* finds better solutions with *alone* being set to 10. Thus, it can be concluded that *disruption* is not necessarily bad for finding good solutions but it certainly reduces the probability of finding good solutions. It must be distinguished between *weak* disruptions and *strong* disruptions because it appears that a strong disruption influences the PGA more than a weak disruption.

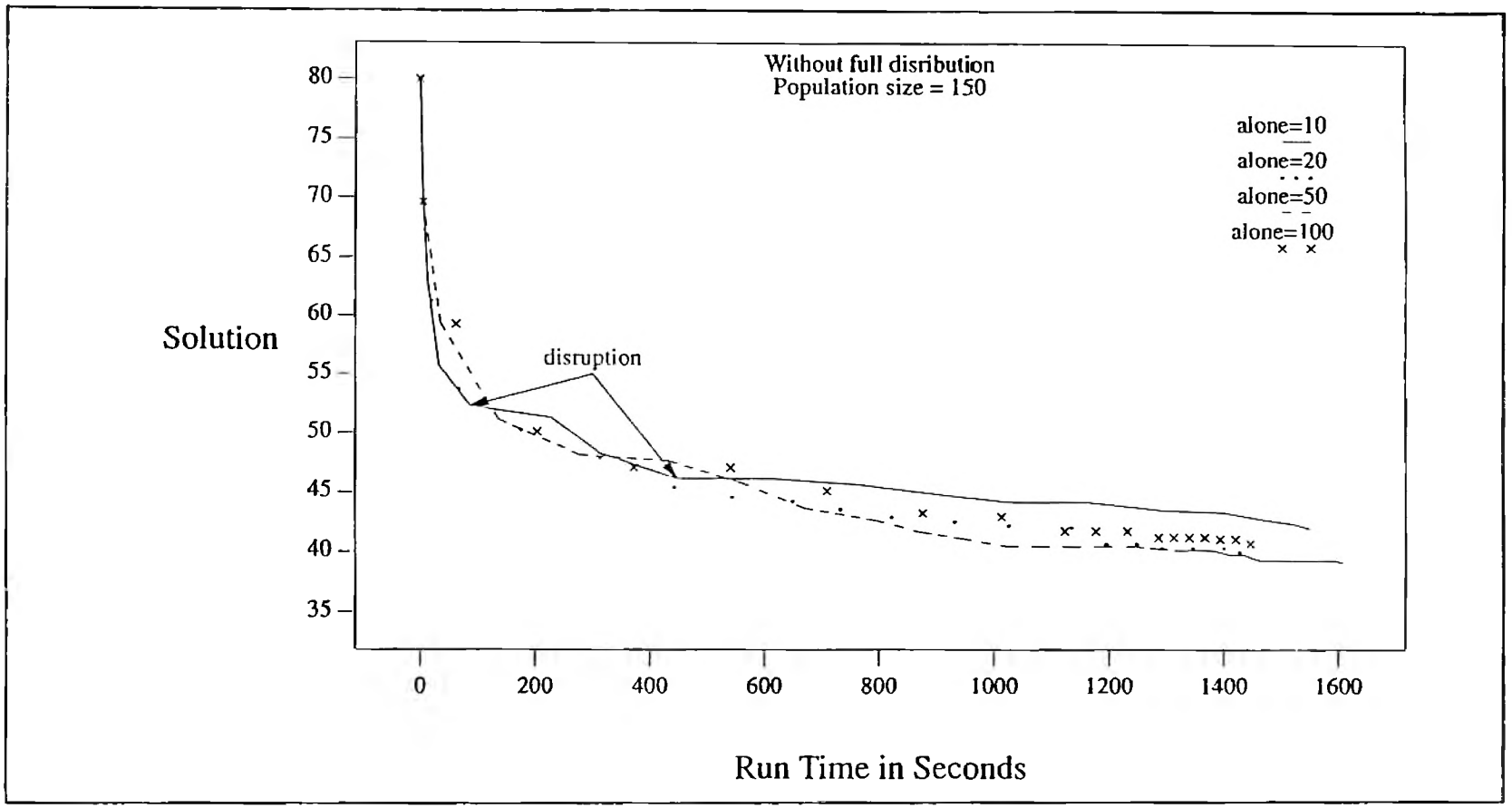


Figure 27. Test run on PGA *without* full distribution and subpopulation size 150

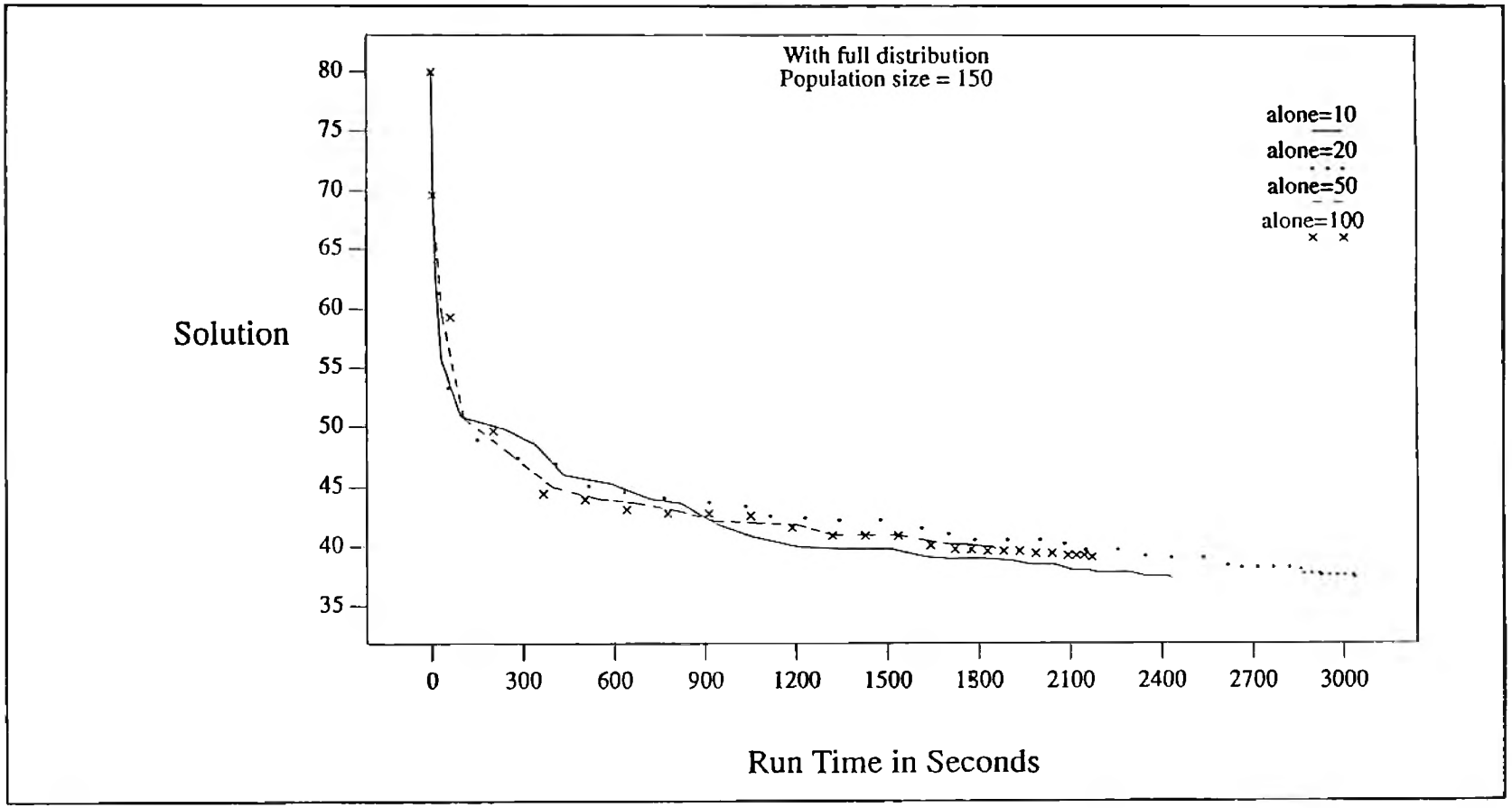


Figure 28. Test run on PGA *with* full distribution and subpopulation size 150

Other important observations can be made if only the total run time is considered. Figure 29 shows the total run time according to the PGA used. *wo* represents the PGA *without* full distribution, while *w* represents the PGAs with *full* distribution of the population. The number after the / represents the number of individuals in the subpopulations.

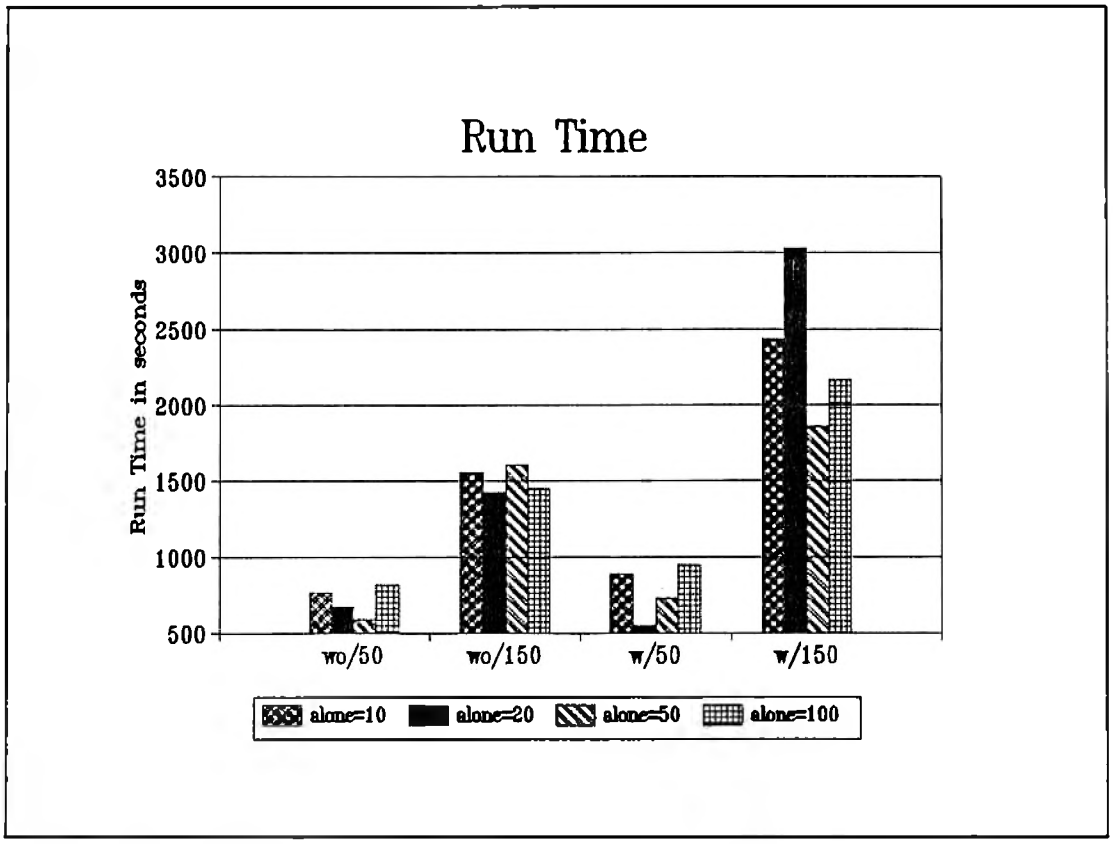


Figure 29. Total run time of PGAs

It can be seen that the average run time for both PGAs *with* distribution is higher than the run time for the PGA *without* full distribution. This is not surprising, because of the added communication overhead caused by the distribution of the best population. The overhead is higher for a subpopulation size of 150 because more individuals need to be distributed. It also can be seen that on the average the best run time is obtained when

alone is set to either 20 or 50. This can be explained as follows: a small *alone* has more communication overhead than a large *alone* because it has more frequent synchronization which requires messages to be send back and forth. But on the other hand, if a solution is found, a small *alone* can act much faster to this new situation and reassign the processors according to the reduced search space. If *alone* is set to a high number, good individuals are not exchanged very often among the processors which causes the subpopulations to be isolated and therefore it is a disadvantage for the PGA.

Figure 30 plots the solutions obtained verses the used PGA.

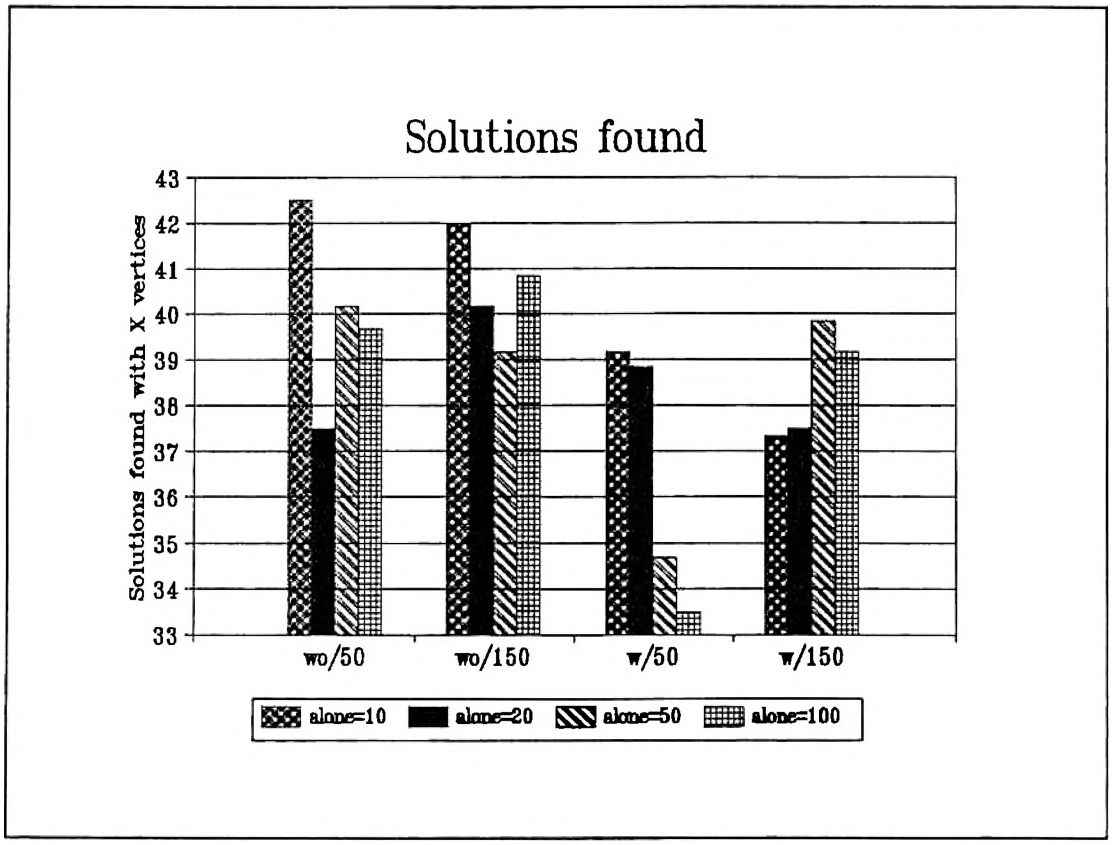


Figure 30. Solution quality with different PGAs

On the average, the best solutions were found by the PGA-*with*. The reason for this behavior can be explained by the fact that the PGA-*with* shares the best solutions among the processors while the PGA-*without* does not. On the other hand, the sharing of solutions by distributing the full populations may be risky, because if the population that has found a solution does not contain the genetic material for better solutions, the PGA might get trapped in a local optima. It is also interesting to note that subpopulations with 50 individuals yield better solutions than 150 individuals. This is due to the fact that a smaller population size allows to select superior individuals easier than in the bigger populations. However, a smaller population size might not have the diversity in the population that is needed to obtain good solutions. The best solutions were found with the following combination of parameters: PGA-*with*, 50 individuals in each subpopulation and *alone* set to be 100.

F. SPEED-UP

The ***Speed-Up*** is on of the most important performance measures for the parallel algorithm. It is computed according to the following equation:

$$\textit{speed-up} = \frac{\textit{Run time of the best sequential algorithm}}{\textit{Run time of the parallel algorithm using X processors}} \qquad (19)$$

Thus, the better the speed-up, the better the parallel algorithm. *Linear Speed-Up* occurs if *speed-up = # processors*. However, there exists one problem with Equation (19). The best sequential Genetic Algorithm is not known. Therefore the equation needs to be

modified in order to calculate the speed-up. Usually the run time of the sequential algorithm is measured on one processor. Therefore the speed-up is

$$speed\text{-}up = \frac{Run\ time\ of\ one\ processor}{Run\ time\ of\ parallel\ algorithm\ on\ P\ processors} \tag{20}$$

Since the described PGA is designed to work on multiple processors, the run time comparison between the PGA with just one processor and the PGA with more than one processor is not possible. Therefore, linear speed-up is assumed between one processor and two processors and the speed-up is calculated using the formula below:

$$speed\text{-}up = \frac{Run\ time\ of\ two\ processors \cdot 2}{Run\ time\ of\ parallel\ algorithm\ using\ P\ processors} \tag{21}$$

The experiment to measure the speed-up for the PGA was run on a PGA with *full* distribution of the population. The parameter *alone* was set to 100 because it got the best results for small population sizes[3]. The *total population size* was set to 160 and 320. Crossover and mutation rate were set to 0.6 and 0.005 respectively. The number of individuals to be exchanged between the processors was constantly set to be 10 and each test was run 10 times to get an average. It can be seen from Figure 31 that the PGA with a total population size of 320 yields almost linear speedup. The reason why it is below the linear speed-up line is because of the communication overhead. The PGA with 160 individuals in the total population shows what is called *super linear speed-up* for four and eight processors. This is due to the fact that Genetic Algorithms are probabilistic algorithms and therefore they can sometimes find better solutions faster.

[3]See previous section.

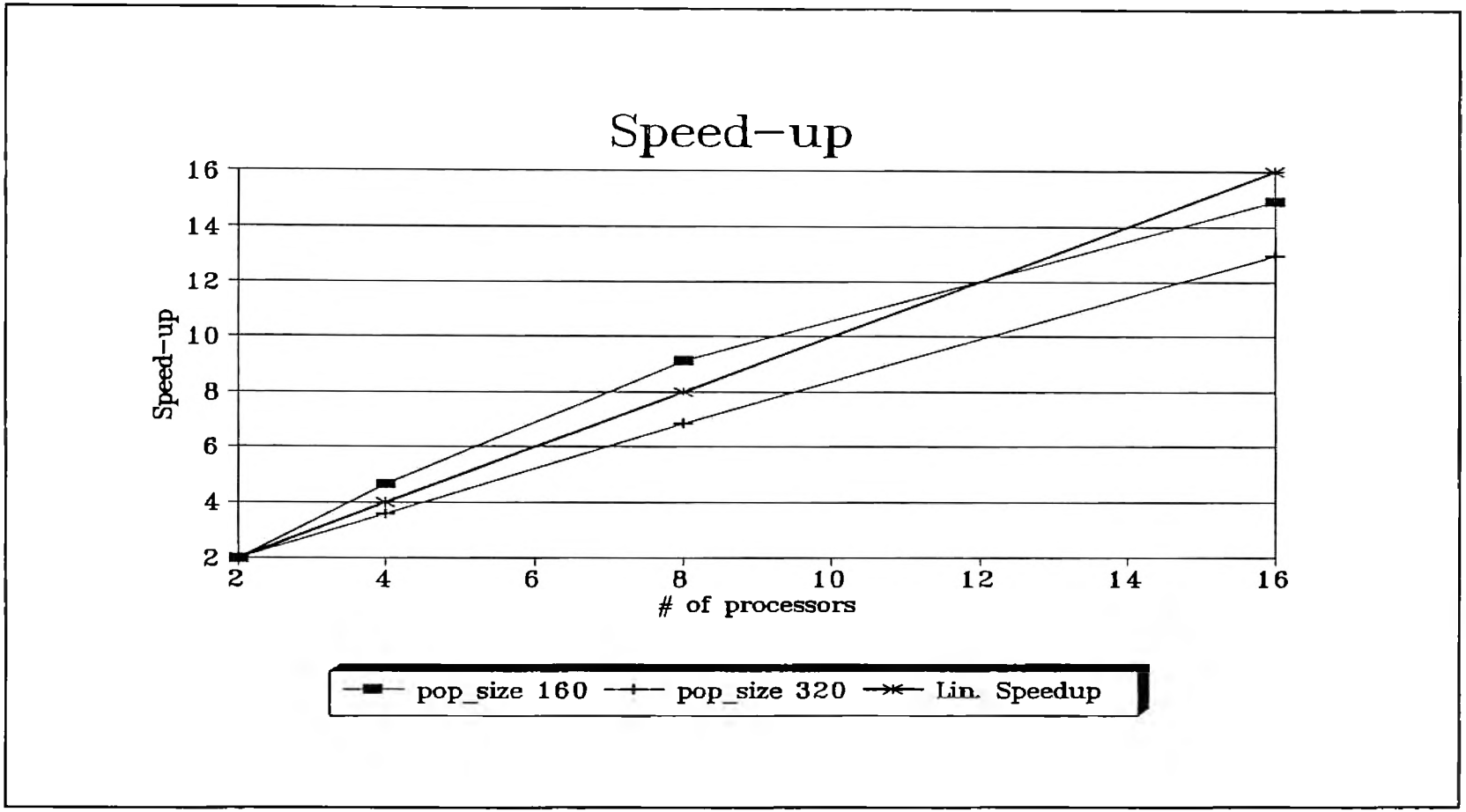


Figure 31. Speed-up of the Parallel Genetic Algorithm *with* full distribution and total population sizes of 160 and 320.

Figure 32 and Figure 33 show the graphs for total population sizes of 160 and 320 respectively with respect to the number of processors. In most cases better solutions were found with a higher number of processors. The reason for this behavior can be explained as follows: Dividing the total population among more processors means that every processor has fewer number of individuals in its local subpopulation. With fewer individuals in the subpopulation it is easier to select superior individuals. On the other hand, if the subpopulation size gets too small, it is harder to create offspring with new genes because of the fewer amount of genetic material in the subpopulation. This result is even more important than achieving a linear speed-up for the Parallel Genetic Algorithm. Therefore, the usage of the PGA has two big advantages:

1) less run time with more processors

2) better solutions with more processors

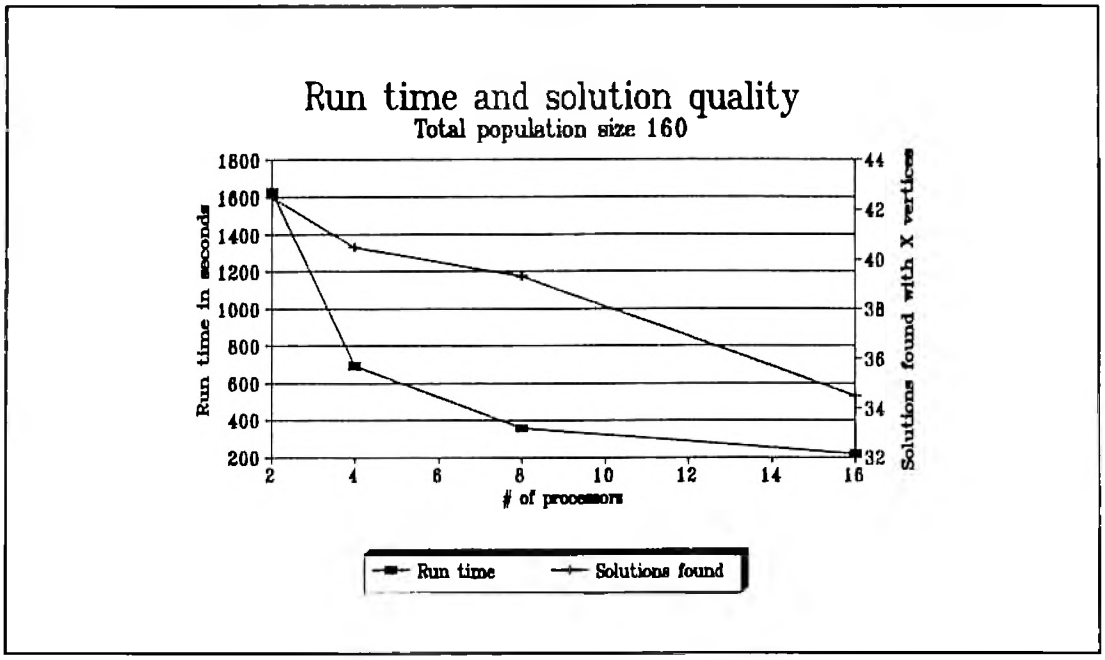


Figure 32. **Run time and solution quality with different number of processors and a total population size of 160.**

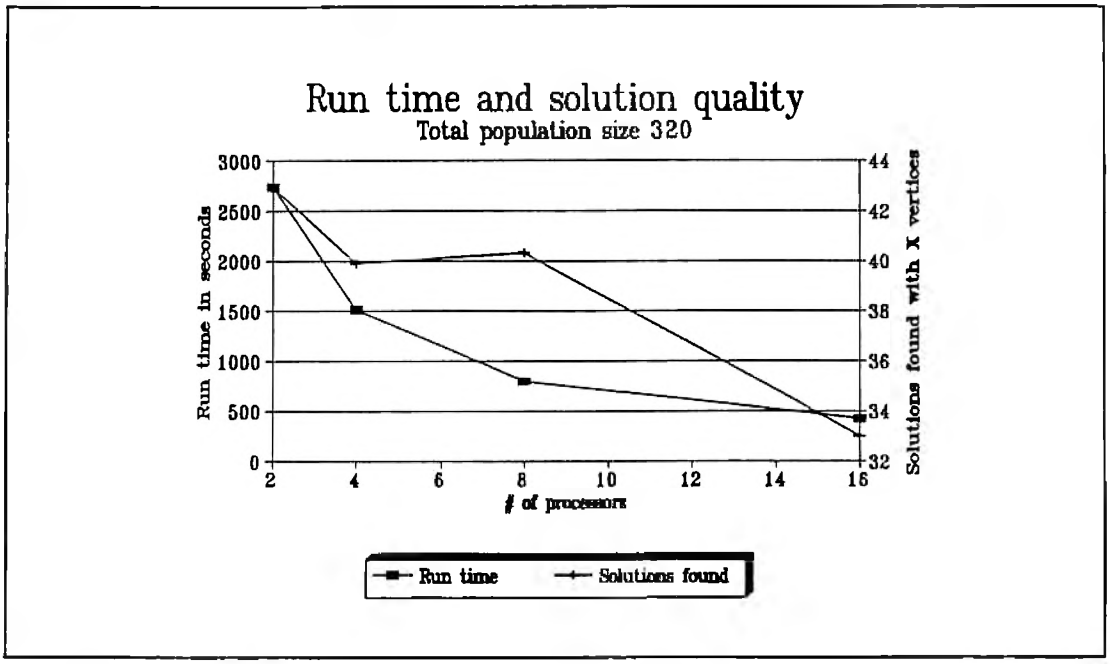


Figure 33. **Run time and solution quality with different number of processors and a total population size of 320.**

G. SUMMARY

Several experiments were run to study the behavior of the parallel and sequential Genetic Algorithm. The analysis of the search space determined that the search space for the DAG Vertex Splitting Problem is $2^n - 1$, where n is the number of vertices in the potential vertex set. The results obtained from the experiments with the *Binary Approximation* are summarized as follows:

1) the initial string length can be reduced by up to 90% when compared to the potential number of vertices.
2) a small *number of tries for BA* results in the best ratio of reduction versus run time.
3) even with a high percentage of reduction in the string length, the remaining search space is still very high.

Experiments with the sequential Genetic Algorithm using different string length reduction methods and different stepping techniques showed that the deletion of vertices in the strings using the *Preserve Duplicates* method biases the GA too much which leads to premature convergence.

All the experiments exhibited one interesting behavior. The Genetic Algorithms had a difficult time finding good solutions if the solutions are obtained fast. This suggests that GAs need time to evolve in order to obtain better solutions. Two short experiments were run to find out how much evolvement is good for the GA. One experiment started the GA with a maximal initial string length without using *Binary Approximation*. The GA

spent too much time evolving in areas which were usually excluded by the *Binary Approximation* and therefore converged prematurely yielding too poor solutions. In the second experiment, the initial string length was set close to the best solution and the GA was run evolving for many generations. The GA was not able to find any better solutions. These two experiments lead to the conclusion that neither too much nor too few evolvement is good for the GA in order to find good solutions. This makes the *Binary Approximation* even more valuable, because it helps to skip over areas in the search space where solutions are easy to find. It also aborts early enough in order not to get too close to the best solution.

Two experiments were run to measure the performance of the Parallel Genetic Algorithm. The first experiment was set up to find out how the frequency of synchronization among the processors influences the total run time and solution quality. The result of the experiment turns out that neither a high nor a low frequency of synchronization are good for the total run time of the Parallel Genetic Algorithm. The total run time for the PGA *with* full distribution of the best population has a longer run time because of communication overhead. However, PGA-*with* found the best solutions compared with the PGA-*without*. An analysis of how the subpopulation size influences the solutions yields to the conclusion that a smaller subpopulation size finds better solutions than a bigger subpopulation size. This experiment also showed that *disruption* during the execution of the PGA, caused by a generation loop that did not find a solution, is disadvantageous for the PGA in most cases.

The PGA showed an almost linear speed-up for a total population of 320 and a super linear speed-up for four and eight processors for a total population of 160. The super linear speed-up can be explained with the probabilistic behavior of the Genetic Algorithms. This similar behavior has also been reported in [18][30]. Not only the PGA exhibits linear speed-up, it is also capable of finding better solutions with more processors.

VIII. CONCLUSIONS

Adaptive search algorithms are capable of adjusting efficiently to their environments. Nature has proven that it is a very good adaptive system. Therefore, search algorithms can be enhanced if natural behavior can be included into them. Genetic Algorithms are such adaptive search algorithms that are based on natural behavior. Nature is represented by a population of individuals and these individuals mate and produce offspring. The reproduction is based on the relative fitness of each individual in the population. Every individual has its genetic material encoded in its chromosomes and with each reproduction, the chromosomes of both parents are crossed to generate the chromosomes for the offspring. The Schema Theorem states that, as the process evolves, the number of individuals with superior characteristics increases exponentially while the number of individuals with inferior characteristics decrease exponentially. Superiority (Inferiority) is based on schemas. Schemas describe similarities in the chromosomes at certain positions. A number of problems can occur when Genetic Algorithms are used, including deception, premature convergence and genetic drift. Solutions such as crowding and elitism have been proposed by other researchers to correct these problems.

This thesis introduced the DAG Vertex Splitting Problem (DVSP) and described a Genetic Algorithm to solve the DVSP. The described GA differs from a standard GA. It uses a variable string length and a variable population size. The chromosomes in each individual represent the vertices that are used to split the graph. The objective of the Genetic Algorithm is to find a minimal set of split vertices (minimal string length) such that the longest path in the splitted graph is less than or equal to a prespecified value.

Therefore the string length must be reduced in order to find better solutions. Two different deletion methods are used to determine the vertices that are to be deleted in conjunction with two different stepping methods to determine the new string length. Experiments on official benchmark graphs have shown that the *Multiple Binary Stepping* outperforms *Linear Stepping* in obtaining better solutions. On the other hand, *Linear Stepping* has a better run time performance than *Multiple Binary Stepping*. The experiments have also shown that the method *Preserve Duplicates* does not help the GA in finding better solutions.

Another set of experiments was run to determine the behavior of the *Binary Approximation*. It is used to determine the initial string length for the initial population. It tries to find solutions by randomly picking vertices. The experiments turn out that just a small number of random tries can reduce the initial string length by up to 90%.

A Parallel Genetic Algorithm is a GA where the total population is subdivided into equal pieces and distributed over the processors. Every processor in the multi-processor system runs a sequential Genetic Algorithm. In addition to the sequential GA, the processors exchange the best individuals from their subpopulation and replace the worst individuals by them. This has two effects: 1) a global population is simulated instead of having many isolated subpopulations. 2) superior individuals are injected to the other processors. A Parallel Genetic Algorithm to solve the DVSP was derived from the sequential GA and implemented on an iPSC/2 Intel Hypercube. The described approach distributes the available processors equally over the search space, meaning that every

processor works on a different string length. The exchange of individuals among processors can only take place if the processors work on the same string length. Usually pairs of processors exchange individuals. This approach uses a *ring exchange* where the processors exchange the individuals in a logical ring. The frequency of exchange is determined by the parameter *alone*. An experiment was set up to see how the PGA behaves with different alone values. The best run time has been achieved with a medium size alone. The best solutions were found with a smaller subpopulation size and a *full* distribution of the best population. A second experiment was run on the PGA to determine the speed-up of the PGA. A super linear speed-up is reported for four and eight processors for a total population size of 160. An almost linear speed-up is obtained for runs with a population size of 320. Analysis is made to explain both results.

Some heuristics have been proposed to solve the DVSP [7]. Future research will incorporate some of these heuristics into various functions of the GA, like select, crossover and mutate [31]. Different *select* strategies as suggested by Baker in [11] should also help to solve the problem better.

APPENDIX A

Test results from the experiment on *Binary Approximation* for the graphs C2670, C3540, C5315, C6288, and C7552.

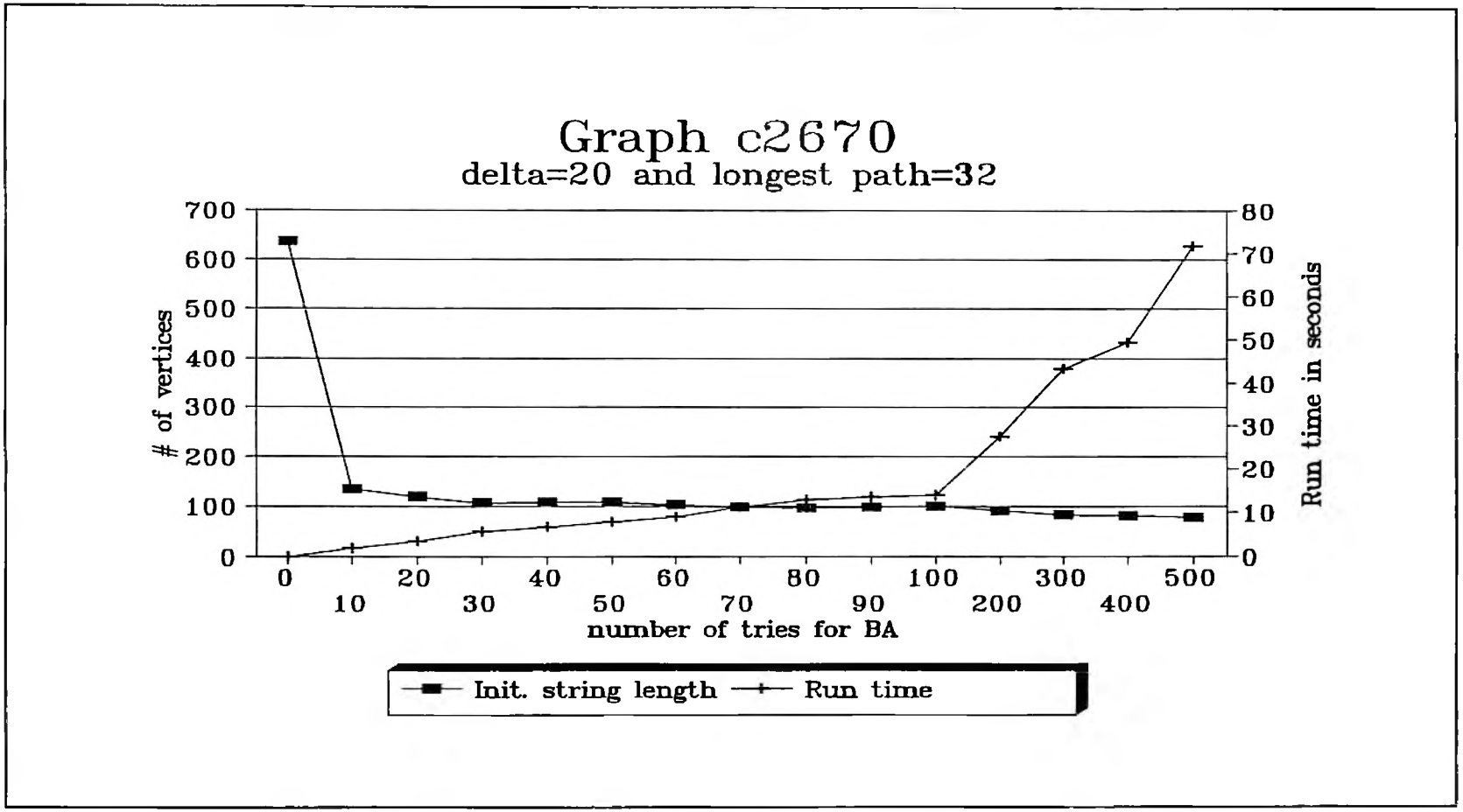


Figure 34. Initial string length reduction with Binary Approximation on graph C2670

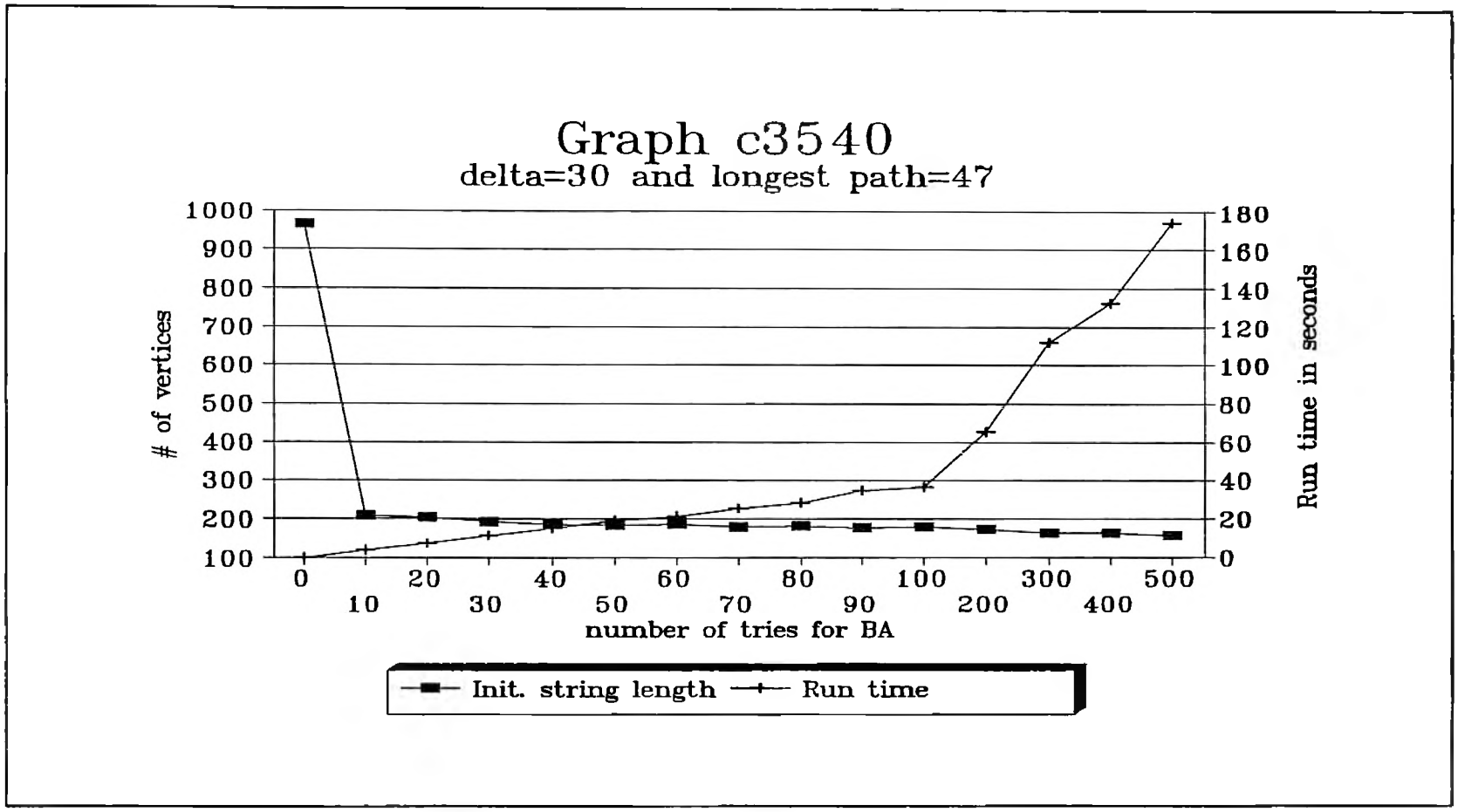


Figure 35. Initial string length reduction with Binary Approximation on graph C3540

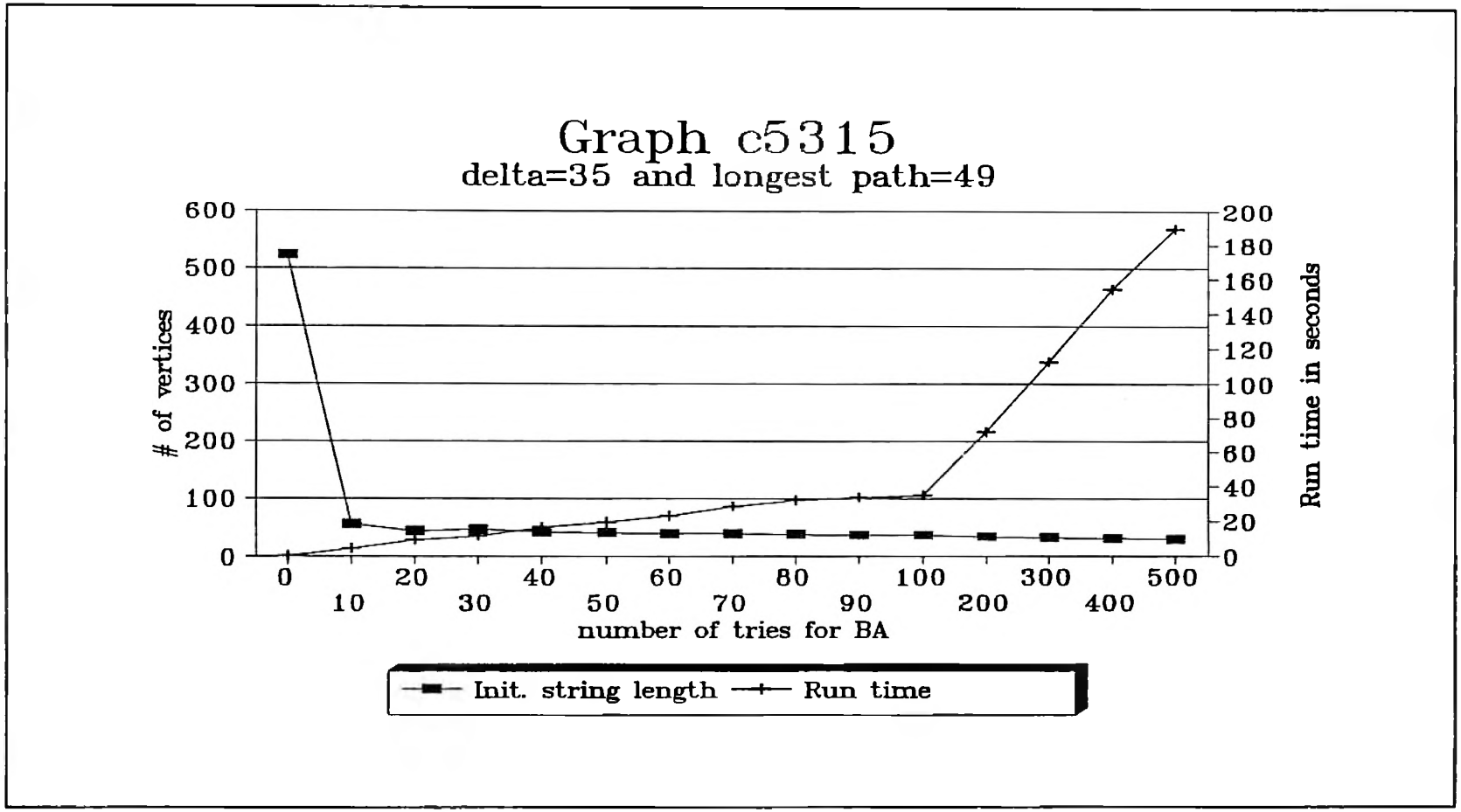


Figure 36. Initial string length reduction with Binary Approximation on graph C5315

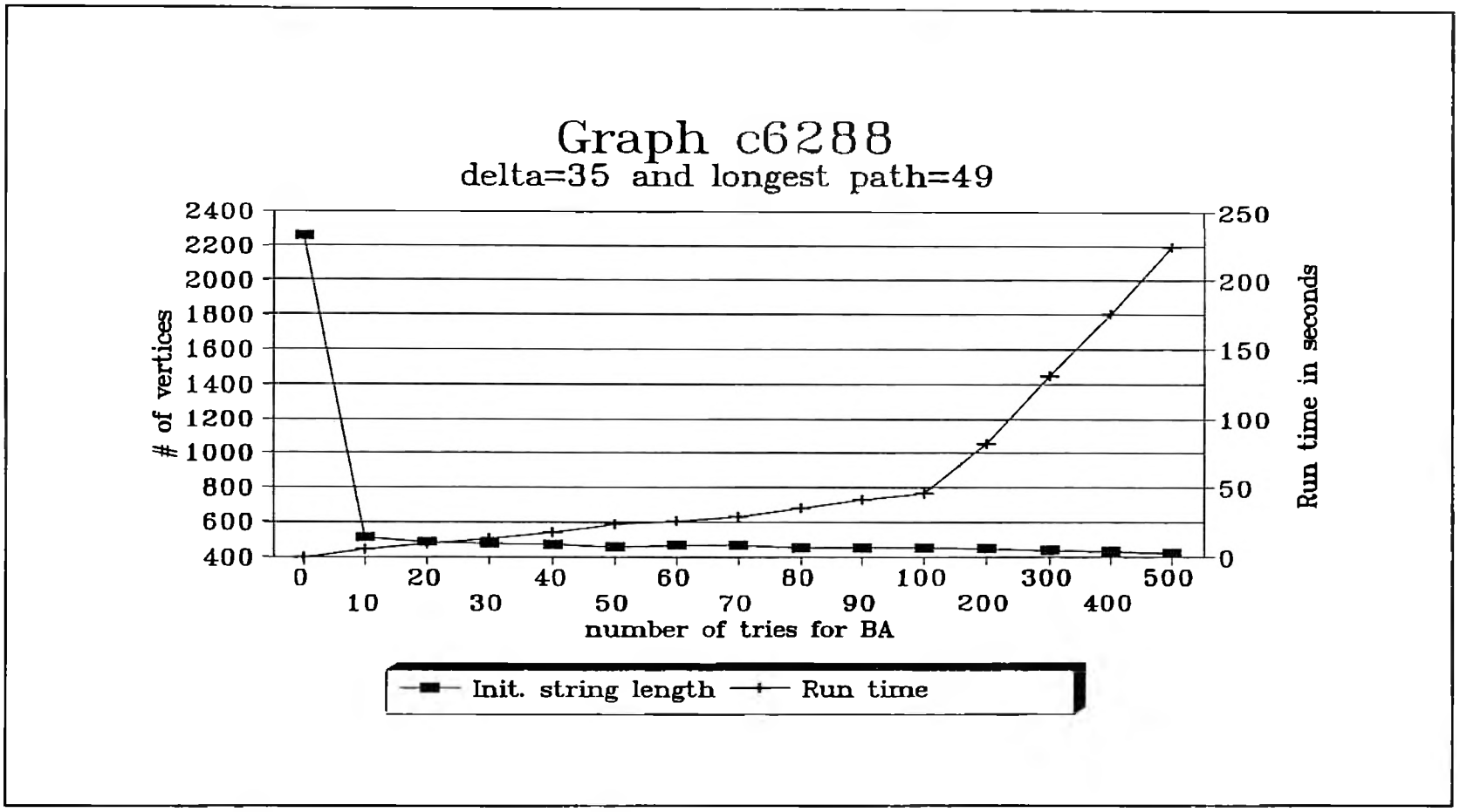


Figure 37. Initial string length reduction with Binary Approximation on graph C6288

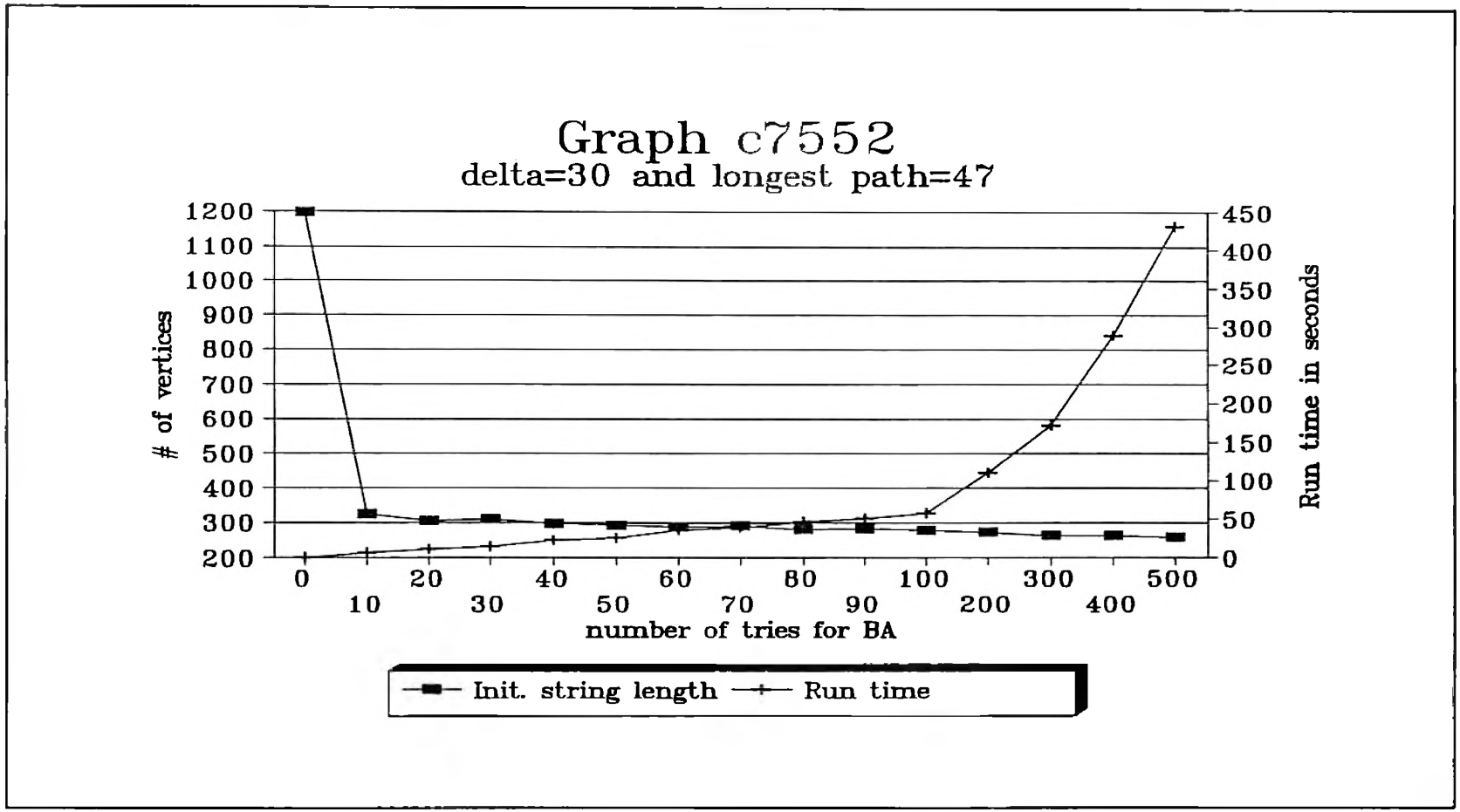


Figure 38. Initial string length reduction with Binary Approximation on graph C7552

APPENDIX B

Percentage of reduction in the initial string length for the graphs C2670, C3540, C5315, C6288, and C7552.

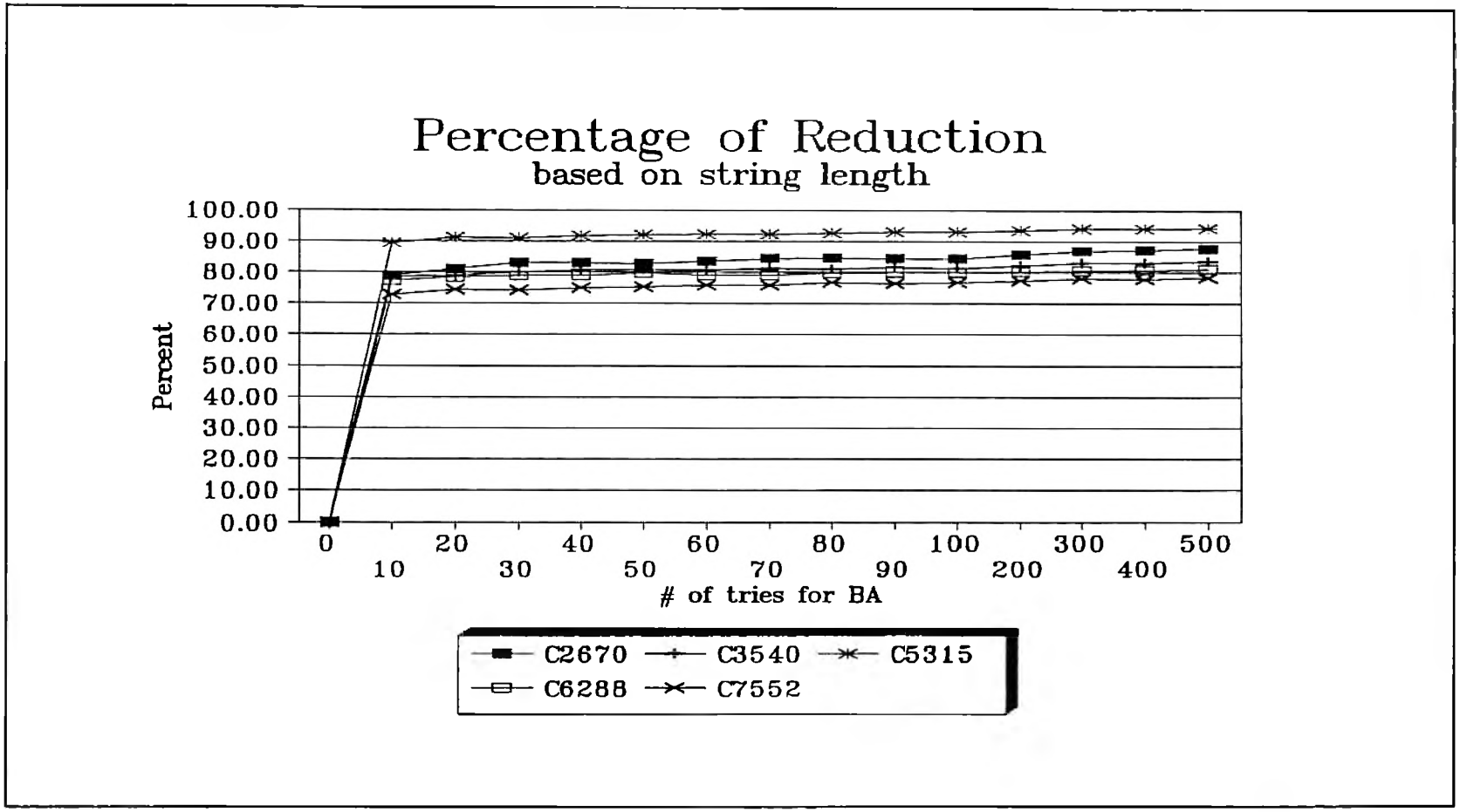


Figure 39. Percentage of reduction in string length for graphs C2670, C3540, C5315, C6288, and C7552

BIBLIOGRAPHY

1. Holland, J.H. (1975). *Adaption in natural and artificial systems.* Ann Arbor: The University of Michigan Press.

2. Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, 1989.

3. Bäck, T., Hoffmeister, F., Schwefel, H-P. (1991). *A Survey of Evolution Strategies.* Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, pp. 2-9.

4. Reichenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Prommann-Holzboog Verlag, Stuttgart.

5. Koza, J.R. (1991). *Evolving a Computer Program to Generate Random Numbers Using the Genetic Programming Paradigm.* Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, pp. 37-44.

6. Koza, J.R. (1992). *Genetic Programming: on the programming of computers by means of natural selection.* The MIT Press (1992).

7. Paik, D., Reddy, S., Sahni, S. (1990). *Vertex Splitting in Dags and Applications to Partial Scan Designs and Lossy Circuits.* Technical Report TR90-034, University of Florida.

8. Goldberg, D.E. and Deb, K. (1990). *A Comparitative Study of Selection Schemes Used in Genetic Algorithms.* TCGA Report No. 90007, University of Alabama.

9. Brindle, A. (1981). *Genetic Algorithms for function optimization.* Unpublished doctoral dissertation, University of Alberta, Edmonton.

10. Wetzel, A. (1983). *Evaluation of the effectiveness of genetic algorithms in combinatorial optimization.* Unpublished manuscript, University of Pittsburgh, Pittsburgh.

11. Baker, J.E. (1985). *Adaptive selection methods for genetic algorithms.* Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp. 101-111.

12. De Jong, K.A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* (Doctoral dissertation, University of Michigan). Dissertation Abstracts International 36(10), 514B. (University Microfilms No. 76-9381) Applications, pp. 100-100.

13. Syswerda, G. (1989). *Uniform Crossover in Genetic Algorithms.* Proceedings of the Third International Conference on Genetic Algorithms. Morgan Kaufmann Publishers, pp. 2-9.

14. Spears, W.M. and Anand, V. (1991). *A Study of Crossover Operators in Genetic Programming.* Sixth International Symposium on Methodologies for Intelligent Systems, Charlotte, NC, pp. 409-418.

15. Spears, W.M., De Jong, K.A. (1991). *On the Virtues of Parameterized Uniform Crossover.* Proceedings of the Fourth International Conference on Genetic Algorithms. Morgan Kaufmann Publishers, pp. 230-326.

16. Mayer, M., Ercal, F. (1993). *Genetic Algorithms for Vertex Splitting in DAGs.* To appear in Proceedings of the 5th International Conference on Genetic Algorithms. Also, Technical Report TR93-02. University of Missouri-Rolla.

17. Jog, P., Suh, J.Y., and Van Gucht, D. (1990). *Parallel Genetic Algorithms Applied to the Traveling Salesman Problem.* Technical Report NO. 314, Indiana University, Bloomington.

18. Mühlenbein, H., Schomisch, M., and Born, J. (1991). *The parallel genetic algorithm as function optimizer.* Parallel Computing, vol. 17, pp. 619-632.

19. Tanese, R. (1987). *Parallel Genetic Algorithm for a Hypercube.* Proceedings of the Second International Conference on Genetic Algorithms, pp. 177-183.

20. Petty, C.B., Leuze, M.R., and Grefenstette, J.J. (1987). *A Parallel Genetic Algorithm.* Proceedings of the Second International Conference on Genetic Algorithms. pp. 155-161.

21. Jog, P. (1989). *Parallelization of probabilistic sequential search algorithms.* Ph.D. Thesis, Indiana University, Bloomington.

22. Lewchuk, M.J. (1992). *Genetic Invariance: A New Type of Genetic Algorithms.* Master's Thesis, University of Alberta, Canada.

23. Cramer, N.L. (1985). *A representation for the adaptive generation of simple sequential programs.* Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp. 183-187.

24. Goldberg, D.E., Korb, B., Deb, K. (1989). Messy Genetic Algorithms: Motivation, Analysis, and First Results. Complex Systems, vol. 3, pp. 493-530.

25. Goldberg, D.E., Deb, K., Korb, B. (1990). *Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale.* Complex Systems, vol. 4, pp. 415-444.

26. Jujiko, C., Dickinson, J. (1987). *Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma.* Proceedings of the Second International Conference on Genetic Algorithms, pp. 236-240.

27. Brglez F. and Fujiwara, H. (1985). *A Neutral Netlist of Ten Combinational Benchmark Circuits and a Target Translator in FORTRAN.* Proceedings IEEE Symposium on Circuits & Systems, pp. 663-666.

28. Aiken, H.H. (1955). *Tables of the cumulative Binomial Probability Distribution.* Harvard University Press.

29. Grefenstette, J.J. (1986). *Optimization of Control Parameters for Genetic Algorithms.* Transactions on Systems, Man, and Cybernetics, vol. SMC-16, no. 1, pp. 122-128.

30. Talbi, E-G. and Bessière, P. (1991). *A Parallel Genetic Algorithm for the Graph Partitioning Problem.* ACM International conference on Supercomputing, Cologne, Germany, July 1991. pp. 663-666.

31. Grefenstette, J.J. (1987). *Incorporating Problem Specific Knowledge into Genetic Algorithms.* Genetic Algorithms and Simulated Annealing, L. Davis, ed. (Pitman, London, 1987), pp. 42-60.