Genetic Algorithms and Multiple Distinct Solutions

Patrick Alasdair Turner

MSc Information Technology: Knowledge Based Systems

Department of Artificial Intelligence

University of Edinburgh

1994

Abstract

A commonly encountered feature of genetic algorithms is that they tend to offer just one candidate solution to a problem. This project investigates methods to increase the number of solutions generated by genetic algorithms in the domain of simple timetabling problems.

Spatial selection [Collins & Jefferson 91], islands [Tanese 89], crowding [De Jong 75], sharing [Deb & Goldberg 89,Goldberg & Richardson 87], some simple combinations of these methods and a new variant of spatial selection called 'tribes' were tested on four basic timetabling problems.

For problems with large numbers of solutions a simple GA run many times was found to produce more distinct solutions in a faster time than any of the specialised methods. On problems with low numbers of solutions spatial selection and tribes were found to be the most efficient at finding distinct solutions.

Acknowledgements

It is with my sincerest gratitude that I thank my supervisors Dr. Graeme Ritchie and Dave Corne. They have helped me to the utmost of their ability and been patient and understanding at all times.

I must also give thanks to Richard Reeve, who has also been extremely patient while explaining statistical methods to me, and to Louise Dennis for words, coffee, chocolate and sympathy.

Finally, I thank Brown & Root Limited who provided financial support during my year of study.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The idea of the genetic algorithm (GA) was first proposed by Holland in [Holland 75]. The GA Holland proposed was based on an evolutionary approach to problem solving in changing environments. A solution is evolved from a population of many potential solutions through a period of generations, emulating the breeding of animals and plants through many generations. Since 1975, Holland's GA has formed the basis for much research into GAs. GAs have been discovered to be efficient at searching large search spaces and have been successfully applied to many problem domains, domains as varied as optimisation of manufacturing schedules and design of nuclear power plant fuel configurations[1].

Despite starting with a *gene pool* of potential solutions, GAs suffer from a tendency to converge on a single solution after a number of generations. An observation on natural evolution is that there is a huge variety of solutions that coexist on the planet. There seems to be a dichotomy between GAs and the evolution they emulate. Is it possible to adapt GAs so that they also produce more solutions?

[1] See [Hedberg 94] for a tourist guide to current GA applications.

1.2 A brief introduction to genetic algorithms

> Genetic algorithms are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure [...]
> *Darrell Whitley, [Whitley 93].*

Before elaborating the objectives of this project I intend to give a very brief introduction to GAs. The information here is very limited: any reader new to the field of GAs is strongly recommended to read a more detailed introduction such as can be found in [Whitley 93] or [Goldberg 89].

This introduction explores simple GAs, similar to the one proposed by Holland. It is split into two: a section on representation is necessary owing to the terminology it introduces and a section on implementing GAs shows how this representation is used to solve a problem.

1.2.1 Representing problems with GAs

A GA encodes a *population* of *individuals*[2] as a set of potential solutions to a problem. The physical description of an individual within a population is its *phenotype*, and the encoded information which maps to this phenotype is a string of genes. We refer to the string of genes as a *genotype*. In this manner, a GA might represent a particular phenotype with one or more genotypes. The term *chromosome*, as I will use it, is an instance of a particular *genotype*. There can be one or more chromosomes in a population each of which share a particular genotype.

The genes on the genotype are placed in a linear string at a set of *loci*. Each *locus* contains one gene. The specific gene at the locus is referred to as an *allele*. In a simple implementation of a GA each gene is either the allele 0 or the allele 1; the genotype is a bit-string of these alleles. There is some mapping function which takes the position and values of the alleles to compute a phenotype. A simple example of this mapping is to treat a bit-string as a binary code and to calculate a corresponding numerical value. Figure 1–1 demonstrates this idea. Of course, the genotype does not have to consist of just binary alleles, or even integer alleles, although this is what GAs tend to use in practice.

In order for the GA to find a solution to a problem, each individual is assigned a *fitness*. The fitness of an individual is some measure of how good a solution to a problem that particular individual is. The evaluation of the fitness might be a function of the genotype or the phenotype. Obviously, this requires some way of

[2]individuals are referred to by [Heitkoetter & Beasley 94].

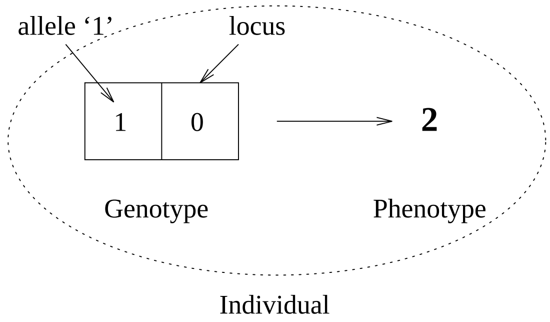


Figure 1–1: Representing a numeric value using a bit-string chromosome

evaluating how good the individual is for the purpose of the problem — if it is not possible to make this evaluation then the problem cannot normally be represented using a GA, although in such a case it may suffice to use an approximate measure of fitness.

1.2.2 How a simple GA works

The aim of the GA itself is to produce solutions from a population of individuals. To achieve this goal the GA attempts to increase the fitness of the members of a randomly chosen population by replacing the chromosomes within it with new, fitter chromosomes. This process is carried out iteratively in *generations*. Each generation comprises selecting *parent* chromosomes from the population, recombining them in some way, perhaps mutating or inverting the resultant *children* and inserting the children into the population. The following description is of a simple GA[3] with *steady-state reproduction*, whereby one child is inserted into the population per generation.

A simple GA takes two parent chromosomes from the population. A *selection operator* is used to select parent chromosomes from the population. Usually, this operator will select chromosomes according to the fitness of the members of the population. A typical selection operator is *rank based selection*. In rank based selection, a parent chromosome is chosen according to the rank of its fitness within the population. Each chromosome gets a chance of being selected as a parent proportional to how well ranked its fitness is.

Having decided on parent chromosomes, the parents are combined using a *recombination* (or *crossover*) operator. An example crossover operator is uniform crossover. Here there is a probability (usually 50%) that the genes at each locus of the parents will be swapped (see figure 1–2) to produce two child chromosomes.

---

[3]In this discussion, a 'simple GA' is not a technical term, merely a simple GA.

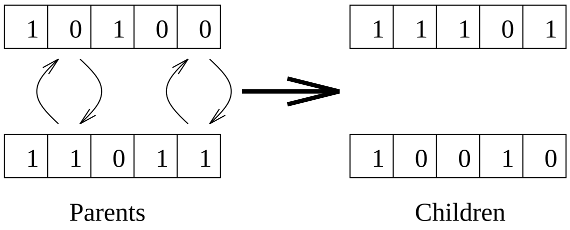


Figure 1–2: Uniform crossover: genes at each locus are swapped according to a fixed probability

Various other operators can be applied to the children. A frequently applied operator is *mutation*, where the gene at each locus is mutated with some probability p_m. The mutation might either change the allele at the locus to a different allele, or to one of the possible alleles (including the allele already present) for that locus.

Different methods for inserting the children into the population are available. Commonly, one child is selected to be inserted into the population. If this child is fitter than the least fit chromosome in the population, then the child is inserted into the population in place of that chromosome.

The cycle is then repeated: selection, recombination, mutation, insertion. The process of generations is repeated until some terminating condition occurs: an optimal solution or, in the case of this project, a population whose average fitness has stayed static for a certain number of generations.

This section has described an implementation of a simple GA. There are many variations on this this theme; however, for the purposes of this dissertation knowledge of the operators introduced here is sufficient.

1.3 Why obtain multiple distinct solutions?

Finding many solutions allows a choice. This choice is helpful in many domains. For example, when preparing timetables, an unforeseen problem with one timetable might rule out its use. In this case it is useful to possess a second timetable, especially if problems may only be found when the timetable is distributed to those it concerns. The freedom of choice of solutions is not only helpful with timetables however: any problem which is likely to involve modification — a travelling salesperson may find that one route is blocked, or a manager may find that one phase of a plan has not been completed on time — will benefit from the ability to make a choice. In some cases, careful specification of a problem can eliminate the unforeseen difficulties — it is possible to specify in advance that there should be an overlap interval in a plan, or even that route X for the travelling salesperson has some percentage chance of being blocked. However, including every aspect of a problem in the specification is not always possible, let alone desirable — for example, the data for when route X is closed may not be available; even if we did have the data, route X may reduce the journey length significantly and therefore a 'route X open' and 'route X closed' journey plan would be useful. Of course, sometimes we might just wish to choose one solution over another for aesthetic reasons — two different designs of bridge or differently coloured foods. In some cases, the matter of choice is not an issue: we may simply want all the solutions to a given problem — what are the roots of some polynomial equation, or which types of tremor are likely to precede a major earthquake?

Multiple distinct solutions also help when we cannot achieve ideal solutions, or the perfect case is not decidable. This is related to the discussion above: not being able, or not wanting, to include certain aspects of a problem in the specification of that problem could be construed as not being able to decide the perfect case. If we are unable to reach the optimal solution then finding a set of suboptimal solutions seems better than just selecting one suboptimal solution. Of the bridges in the previous paragraph, we might have one which is weaker and one that is more brittle. To decide between these suboptimal solutions is certainly possible given more specification: e.g. X, Y, Z and brittle is better than A, B, C and weak. However, as alluded to above, the extra specification may be too complex to include, difficult to predict in advance of finding a solution or just not realised as being pertinent to finding a solution.

I hope, from this discussion, that it is clear that multiple distinct solutions are relevant to many domains. The domain of timetabling is of particular relevance to the University of Edinburgh. Due to the success of timetabling genetic algorithms developed by Ross, Fang and Corne (see chapter 2.1 for a full description of the representation they have employed) the University of Edinburgh Department of Artificial Intelligence now uses genetic algorithms to timetable both MSc lectures and examinations. As has been mentioned, potential timetables often suffer unforeseen problems: more students than expected may choose a certain course, a lecturer may find that they have another commitment on a certain day, or students and lecturers may simply prefer one solution over another. For reasons such as these, multiple timetables to choose from are of interest to course organisers.

As a final word, this is not to overestimate the importance of multiple distinct solutions. Obviously, *a* solution is required at first; however, having found a solution, another is always desirable.

1.4 Objectives of this project

The aim of this project is to find adaptations of GAs which can generate multiple distinct solutions in the domain of timetabling problems. The project will test some previously proposed and some new methods for obtaining more than one solution on timetabling problems.

As was mentioned in the previous section, timetabling problems are particularly relevant to current research at the University of Edinburgh. Before going on to outline the methods employed in the course of the project and some of the theory behind them, I will introduce the domain of timetabling problems in this section. Chapter 2.1 will go into more detail and describe one method to represent timetabling problems with GAs.

In addition to considering the domain of timetabling problems, it is necessary to think about how any multiple solutions obtained should be classified. This section also overviews the issues of classifying distinct solutions; again, there is a more detailed discussion about criteria for judging the relative merits of different sets of distinct solutions found in chapter 2.2.

1.4.1 Timetabling problems

A timetabling problem can be defined as trying to fit a set of events into a set of slots. There may be different types of slots: time slots, location slots or lecturer slots. The events are governed by a set of constraints specifying how they can be placed in these slots. For example, a tutorial might be required to follow a lecture. If we consider all the constraints necessary in a particular case, then we can formulate a timetable problem from those constraints.

In [Corne *et al.* 93], Corne, Fang and Mellish describe how the Modular Exam Scheduling Problem (MESP) was tackled in the Department of Artificial Intelligence at the University of Edinburgh using GAs. Previously, exam timetables had been compiled by hand, starting with each department's teaching and course timetables. In a modular course the exam timetables for each department were merged to form a complete modular exam schedule. However, since students on modular courses take modules from different departments, the complete timetable can involve many clashes of exams. The resolution of this problem was to draft and redraft the timetable painstakingly as complaints of clashes from students were registered. Corne, Fang and Mellish loaded the constraints of the problem into an evaluation function and weighted it according to how many students take each course. Using this evaluation function, a GA is very successful at eliminating the clashes, and then minimising the number of students who have to take consecutive exams, three exams in a day, and so on.

Chapter 2.1 discusses a variant of the representation Corne, Fang and Mellish used for timetabling problems chapter 2.1, while chapter 4.1 describes a test set of four simple timetabling problems which were artificially constructed to have from 10 to 120 solutions. These problems were used to test GAs adapted to produce multiple distinct solutions.

1.4.2 Classifying distinct solutions

Given the objective of trying to find multiple distinct solutions, it will be necessary to decide to have some indicators of how well a GA has performed at finding many solutions. We will need to be able to classify distinct solutions and also determine the rate at which these solutions are found.

Some distinct solutions may be very similar and some very different. Ideally, when searching for many solutions, more of the different solutions would seem desirable. For example, two timetables might have just two events swapped or the whole arrangement of the timetables may 'look' completely different. But how different? Figure 1–3 demonstrates some of the difficulty in making this judgement even with very simple timetables.

	1	2	3	4
6.00am	SLEEP	SLEEP	SLEEP	RELAX
10.00am	WORK	RELAX	SLEEP	SLEEP
2.00pm	EAT	EAT	WORK	EAT
8.00pm	RELAX	WORK	EAT	WORK
Midnight	SLEEP	SLEEP	RELAX	SLEEP

Figure 1–3: Some simple timetables: how much do they differ?

To us as observers, the solutions 1 and 3 are quite similar, a simple rotation has been performed, while 1 and 4 are quite different. However, both pairs only have one event in common. Questions arise here about how we recognise similarity between the timetables. An evaluation of 'similar' solutions will actually involve our perception of what 'similar' means. When do they stop being similar? If we reverse all the events, as in solution 2, is this any less similar to timetable 1 than swapping any two events? After all, this is just another way that solution 2 could be perceived. I do not intend to answer these questions of perception of similarity in this project. Instead, chapter 2.2 looks in more detail at pragmatic ways to judge how different timetables are. Chapter 4.2 describes how these metrics were applied to gauge the success of GAs at producing multiple distinct solutions.

1.5 Obtaining multiple distinct solutions with GAs

> The main idea is that there should be a mechanism that will control and prevent an unbalanced proliferation of genotypes. It is clear that such an operator can be imposed through an explicit testing of genotype diversity but that such a mechanism is computationally prohibitive.
> *Yuval Davidor, [Davidor 91].*

Proposing a method for producing multiple distinct solutions requires an idea of why GAs tend to generate only one solution.

The unbalanced proliferation of genotypes that Davidor mentions is the norm for simple GAs. Fitter chromosomes are given much higher chances of being selected from the population. When they have been selected their offspring have similar genotypes. With uniform crossover, typically at least 50% of the alleles will be the same, usually more. If the chromosome contains a fit *schema* this schema will tend to survive. A schema is a subsection of a genotype, where some alleles are specified and others are left as wild-cards. There is a fundamental theorem of GAs, the *Schema Theorem*, which proves that the numbers of chromosomes with proportionally fitter schemata in the population will increase. The fit schemata not only increase in numbers, but tend to eradicate the lesser schemata very quickly. This happens because chromosomes possessing a fit schema will replace less fit chromosomes which do not possess that schema. In subsequent generations the child chromosomes will join the others with the schema at the top of the list ready for selection. The trick to obtaining multiple distinct solutions is, then, to encourage chromosomes with different schemata to reproduce or, alternatively, to discourage reproduction of chromosomes with a single schema.

Davidor states that it is possible to encourage a general diversity of genotypes in the population by explicitly testing the diversity of the population, but the computational cost of explicit testing is too high to allow us to use it in practice. Instead, the method we do use to sustain the diversity of a population must not only be computationally feasible, but all also ensure that the GA still finds solutions.

This project investigates various methods proposed to increase the number of solutions obtained from a GA. *Spatial selection* has been proposed by Collins and Jefferson [Collins & Jefferson 91] and Davidor [Davidor 91] as a method to encourage diversity of genotypes by isolating some chromosomes from others. Spatial selection is an adaptation of *island models* or *distributed genetic algorithms* which have been proposed by Tanese [Tanese 89] among others. Island models isolate members of populations in subpopulations, allowing some interaction between these subpopulations. *Crowding* was proposed by De Jong in [De Jong 75] and is a method to ensure that chromosomes only replace chromosomes with similar genotypes in the population. A mixture of crowding and spatial selection was created for this project. *Sharing* has been implemented by Deb, Goldberg and Richardson in [Deb & Goldberg 89] and [Goldberg & Richardson 87]. Sharing works by lowering the fitness of individuals which share some common feature. Typically, the fitness of individuals in the population with similar genotypes might be reduced. Sharing was also combined with spatial selection for this project.

Finally, a new method is proposed here to be combined with spatial selection, although there is no reason why the method cannot be used with a simple GA as well. *Tribes* prevents chromosomes from being inserted into the population if their genotype is too dissimilar to those they are replacing.

Most of the methods this project investigates have seen little research. Certainly, research conducted so far has been on a limited test set of problems without a general comparison with the other methods. This project uses simple timetabling problems to investigate their behaviour in a pragmatic domain.

1.6 Survey of findings

The four timetabling problems tested had 10, 30, 60 and 120 solutions, and for each of the methods a metric was used to determine the number of distinct solutions found and the rate at which they were produced.

While all the methods (with the exception of spatial sharing) increased on the number of distinct solutions found by a simple GA, they were almost always less efficient at finding the solutions than a simple GA.

Sharing found the greatest number of distinct solutions on the problem set, while also being the most inefficient method at finding solutions. Sharing was closely followed by tribes for an ability to find distinct solutions. However, in contrast to sharing, tribes was the most efficient method at finding distinct solutions on a problem with 10 solutions. Spatial selection and spatial crowding were also found to be more efficient than a simple GA at finding solutions on a problem with 10 solutions. As the number of solutions was increased all the methods proposed to find distinct solutions became less efficient as compared to a simple GA until, by the 120 solution problem, a simple GA was found to be the most efficient method to find distinct solutions.

1.7 Outline

The basics of genetic algorithms and multiple distinct solutions have been discussed in this chapter. The chapter gave a brief introduction to GAs and why obtaining multiple distinct solutions might be useful. Next, a section was given to the objectives of the project: investigating methods to produce multiple distinct solutions on simple timetabling problems; the section asked what timetabling problems were and introduced the issues of classifying distinct solutions. The following section looked at how methods to find multiple distinct solutions might be implemented. Finally, a survey of findings was presented.

Chapter 2, *Foundations*, will pick up the thread from the objectives section. More detail will be presented on how to represent timetabling problems with GAs, and some methods for classifying distinct solutions will be introduced.

Chapter 3, *Methods for Obtaining Multiple Distinct Solutions*, will take the introduction to methods given in section 1.4.2 a stage further and discuss the details of the methods tested in the project.

Chapter 4, *Outline of Test Plan*, combines the theory in chapters 2 and 3 to form a plan for the experimentation to be conducted. It defines the timetabling problems to be used, the implementations of the MDS methods and the metrics for judging their success.

Chapter 5, *Discussion of Results*, looks at how the methods performed in practice and offers some explanation of the reasons why they succeeded or failed.

Chapter 6, *A new method: Tribes*, looks at some of the lessons learnt from chapter 5 in order to propose a new method to be tried. It discusses how this method was tested, reports the results and gives an analysis of them.

Chapter 7, *Conclusion*, ties together the discussions of results from chapters 5 and 6 in order to make some conclusions about finding multiple distinct solutions. Some suggestions for further work are presented; finally, a short section assesses what the project managed to achieve.

1.8 Program notes

The PGA program v2.7 written by Peter Ross and Geoffery H. Ballinger in the C programming language formed the experimental platform for this project. This program was adapted for work with some of the methods tested; a manual for the modifications made is included in appendix B. A program listing for the entire code is available in appendix C. Please note that PGA v2.7 is copyright of Peter Ross and Geoffery H. Ballinger and it is reproduced here in accordance with section 3 of the GNU General Public License (version 2).

Chapter 2

Foundations

2.1 Representing timetable problems with GAs

Chapter 1.4.1 introduced timetabling problems. This section outlines a GA representation of timetabling problems, as developed by Ross, Corne and Fang in [Ross *et al.* 94].

As discussed in chapter 1.4.1, a timetabling problem can be described by a number of constraints on events. If we have n events, then we can represent the events as positions on a genotype so that the genotype is n genes long. If we have m time slots, we can arrange the genotype so each allele corresponds to one of the m slots that is available. We might want to include location slots on the same chromosome if there is some interaction between place and time (for example, it may take ten minutes to get from one location to another). For this, we repeat the events on the same genotype and assign p alleles, one for each location, for this portion of the genotype. An example of a chromosome generated in this manner is shown in 2–1.

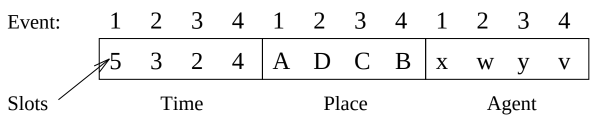


Figure 2–1: Example of a timetabling chromosome

Having formed a representation like this, we can formulate a fitness function to assess how well a certain timetable genotype fits the constraints. We define constraints of different types such as *edge constraints* (where two events cannot overlap). The edge constraints may be weighted according to how important it is that the two events cannot overlap. Other constraints can be defined in the same way; for example, one event that must precede another. The number of constraints that the genotype breaks is easy to calculate, as the representation states which events are in which slots. We can evaluate all the constraints that a particular genotype breaks, and sum them to form a fitness function. The fitness is defined as:

$$f(x) = 1/(1 + \sum_{i=1}^{C} P_i v_i(x)) \tag{2.1}$$

C is the number of constraints. P_i is the penalty associated with breaking the ith constraint, $v_i(x)$ is a δ-function determining whether individual x has broken the ith constraint. This function ranges from ~ 0 when lots of constraints are broken to 1 when no constraints are broken.

2.2 Methods for classifying distinct solutions

There are natural difficulties in the comparison of distinct solutions. As was mentioned in chapter 1.4.2, how distinct are any two solutions? Are five similar solutions better than two very different solutions? In addition to the innate problems of comparing distinct solutions, there are also method specific problems with comparison. How do we compare the time taken to reach distinct solutions? Are a few solutions found quickly better than many solutions in a longer time? This sections identifies some metrics for comparing the success of different methods: the number of solutions, the amount of diversity and the time taken to reach solutions.

2.2.1 The number of solutions obtained

The number of solutions obtained gives only a rough guide to how well a GA has performed. Using the number of solutions says nothing about how diverse those solutions are. However, as a rough guide to how the GA has performed it seems sensible to see how many solutions are produced. This section introduces *effective trials*, a new metric which gives a better assessment of how well a certain GA run has performed than counting the number of solutions it produces.

Judging methods simply by counting the number of solutions produced will naturally hinder methods that generate large numbers of solutions, since, as more solutions are found, it becomes harder to find new distinct solutions. For example, rolling a die will take one throw to find a single distinct face. To find another distinct face, on average, a further 6/5 throws will be required (since out of 6 throws only 5 will reveal a new face) and so on.

This toughness of finding new solutions can be accommodated simply by looking at the equivalent 'average number of throws' required to find a solution rather than the actual number of solutions. For a q solution problem, to find all the solutions this number is:

$$\text{Throws} = 1 + \frac{q}{q-1} + \ldots + \frac{q}{2} + q \tag{2.2}$$

1 solution is equivalent to an average of 1 throw, 2 solutions are equivalent to an average of $1 + \frac{q}{q-1}$ throws, and so on. However, this formula is only useful when considering integral numbers of solutions. If we have a method that produces an average of, for example, 3.2 solutions over 10 runs, how many throws is this equivalent to? For this, we will require a continuous function.

Reversing equation 2.2 gives:

$$\text{Throws} = q\left(1 + \frac{1}{2} + \ldots + \frac{1}{q-1} + \frac{1}{q}\right) = q\sum_{i=1}^{q}\frac{1}{i} \tag{2.3}$$

Noting that:

$$q\int_a^b \frac{1}{x}\,dx < q\sum_{i=a+1}^{b}\frac{1}{i} < q\int_a^b \frac{1}{x+1}\,dx \tag{2.4}$$

an approximation to $\sum_{i=a+1}^{b}\frac{1}{i}$ is:

$$q\sum_{i=a+1}^{b}\frac{1}{i} \approx q\int_a^b \frac{1/x + 1/(x+1)}{2}\,dx = \left[q\ln\sqrt{x^2+x}\right]_a^b \tag{2.5}$$

Hence, if we find p solutions, the equivalent number of throws is approximately:

$$\text{Throws} \approx \left[q\ln\sqrt{x^2+x}\right]_{q-p}^{q} \tag{2.6}$$

Equation 2.6 shows the effective number of 'Throws' taken or *effective trials* that have been conducted if p solutions are found. The effective trials provides a fairer way to compare how well GAs have performed than simply looking at the number of solutions, because the worth of finding each extra solution is taken into account. We may use equation 2.6 to define the effective trials t_{eff} as:

$$t_{\mathit{eff}} = \left[q\ln\sqrt{x^2+x}\right]_{q-p}^{q} \tag{2.7}$$

There are a couple of caveats to do with measuring the effective trials that should be noted. Firstly, the effective trials does *not* measure the diversity of the solutions found. For this, a diversity metric is needed; some are introduced in the next section. Secondly, the effective trials was originally conceived in a case where it was only possible outcome of a throw was a solution. In the case of a GA, a possible outcome is 'no solution found'. However, the effective trials handles the case where there are less the 1 solution: it tends to 0 as the solutions tend to 0. Because t_{eff} is continuous to 0, the fact that there is a possible 'no solution found' outcome at the end of a GA run is not a major problem for the effective trials.

In summary, measuring the effective trials is only possible when the number of solutions is known and the distribution of the solutions is assumed to be even. Given these restrictions, the effective trials allows an easy to calculate approximation to how well a particular method is doing.

2.2.2 Measuring the diversity of solutions

This section starts with a little background on measuring diversity and then introduces Hamming distances. It goes on to describe the methods used by Collins and Jefferson in [Collins & Jefferson 91] and Deb and Goldberg in [Deb & Goldberg 89] to measure diversity. It describes why each is inappropriate for use with general timetabling problems. Finally, the section discusses a method proposed by Corne, the *allele count diversity*.

Counting the number of solutions will not provide us directly with any information about how different a solution is. With a diversity metric we hope to be able to say a little about the difference between two solutions. As was pointed out in the introduction, the difference between two solutions may come down to personal taste. However, what is certainly true is that a difference in genotype of solution does not necessarily correspond to a difference in phenotype. The many to one relationship between genotype and phenotype means that two very different genotypes could represent the same phenotype. When I come to discussing diversity below I will concentrate on methods which compare genotypes. Is this a good idea? Well, in general, probably not. However, in the case of timetables, the genotype to phenotype relationship is very direct. The genotype could even be viewed as a phenotype. So, in the case of timetables, measuring the diversity of solutions through measuring the diversity of genotypes is reasonable.

An important factor of a diversity metric is that it should have some notion of how diverse two solutions are. Again, the point is that it is quite difficult to make this distinction. However, we can try to analyse genotypes in a way that will work in favour of very different genotypes and against very similar genotypes. Similar might mean a number of things, although there is one important metric that we can use: *Hamming distance*. Hamming distance is used in a number of places in this dissertation and it is important to know the concept.

Hamming distance

The Hamming distance, d_H, is the sum of the *bit difference* between alleles at each locus on a genotype. If the alleles are the same then the bit difference is

0 otherwise it is 1. As the name implies, bit difference was originally applied to binary chromosomes. In the case of n-ary chromosome[1] the bit difference follows the same rule: if the two alleles are different then the bit difference is 1. The definition of the bit difference, δ_{ij} is:

$$\delta_{ij} = \begin{cases} 0 & \text{if } \alpha_{(1)i} \equiv \alpha_{(2)j} \\ 1 & \text{otherwise} \end{cases} \tag{2.8}$$

Where i and j are loci, and $\alpha_{(n)i}$ is the allele at locus i on genotype γ_n. Using this definition, the definition of Hamming distance is, for a genotype of length ℓ:

$$d_H(\gamma_1, \gamma_2) = \sum_i^{\ell} \delta_{ii}. \tag{2.9}$$

Unfortunately, although the Hamming distance is suitable for telling how diverse two genotypes are, it is difficult to see how it should be used to calculate how different a set of genotypes is. For this reason, Collins and Jefferson, Deb and Goldberg and Corne have all introduced diversity metrics. These methods are described in the following sections.

Allele diversity

In [Collins & Jefferson 91], Collins and Jefferson describe the measurement of the *allele diversity*. By finding how well distributed each allele is at each locus in the solutions found, we can tell to what extent the GA has converged to a single genotype.

The allele diversity compares the observed frequency of a particular allele with the expected frequency of that allele in an evenly distributed population of alleles. The allele diversity at each locus is

$$D_{(allele)i} = 1 - \left(\frac{F_i^{(e)} - F_i^{(o)}}{F_i^{(e)}} \right)^2 \tag{2.10}$$

where $F^{(e)}$ is the expected frequency of the allele and $F^{(o)}$ is the observed frequency. For a binary chromosome, at each locus, there are just two possible alleles: either 0 or 1, and therefore in a binary chromosome $F^{(e)}$ is simply 0.5 for any single allele.

[1] An n-ary chromosome is a chromosome with n possible alleles at each locus.

If there is just one solution then the observed frequency of each allele will be either 1.0 or 0.0 at each locus. In both cases this gives $D_{(allele)i} = 0$ for every locus on the genotype. Similarly, at maximum diversity there will be a frequency of 0.5 of each allele and hence $D_{(allele)}i = 1$ for every locus on the genotype.

The allele diversity is useful when considering binary chromosomes: in this case the alleles are complementary, and $D_{(allele)(0)} = D_{(allele)(1)}$ at every locus. Hence a single value of $D_{(allele)}$ for the whole chromosome may be obtained by summing $D_{(allele)}$ for one or other allele for every locus on the chromosome. However, when considering a non-binary chromosome this is not the case. A single value of $D_{(allele)}$ for a locus cannot be obtained, and summing a certain allele for the whole chromosome will be misleading.

Because timetable problems will frequently have more than two alleles, the method is not really suitable for application in this project, although it is useful where binary chromosomes are concerned.

'Chi-square-like' diversity

Deb and Goldberg use a 'chi-square-like' quantity in [Deb & Goldberg 89] to measure diversity.

The 'chi-square-like' quantity uses a set of discrete possible outcomes for a run to form an ideal distribution and then measures the observed χ^2 deviation from that distribution.

The ideal distribution is calculated by taking the number and spacing of the solutions. If there are q solutions and a population of N chromosomes, then the mean number of chromosomes for each peak should be N/q and the variance is $N\frac{1}{q}(1 - \frac{1}{q})$. Another discrete result is included: 'not on any peak'. This has a mean of 0 and variance of $\sum_{i=1}^{i=q} N\frac{1}{q}(1-\frac{1}{q})$. Note that the mean and variance of the set of results for 'peaks' is not independent of the mean and variance of the result 'not on any peak', although this independence is assumed by Deb and Goldberg.

The failing of the method for our purposes is that it is *not* a diversity metric in the sense that we have been talking about. It says nothing about how diverse the solutions found actually are. What it is measuring is the difference between an ideal distribution and an empirical distribution of a set of solutions. We might use an example similar to that used for the effective trials metric to show elucidate the meaning of the metric. Suppose we throw a die 100 times, how many times do we expect to see each face? The answer is clearly $\frac{100}{6}$. We then throw the die 100 times and discover how different the actual distribution is from the one we expected — i.e., measure χ^2. This ideal distribution is just as for the effective trials: solutions are assumed to be positioned evenly across the search-space. We are not actually saying: "How diverse are the solutions?" but, rather, "How different is the *distribution* of the solutions from what we expect?"

Allele count diversity

Allele count diversity was suggested by Dave Corne, [Corne 94], as a simple to apply diversity metric for n-ary chromosomes. It involves simply counting the number of alleles at each locus in the set of solutions. An example of the allele count diversity calculation for three genotypes is shown in figure 2–2.

3	4	1	3	2	0

2	4	0	3	2	1

3	4	3	3	1	4

Diversity= 2 + 1 + 3 + 1 + 2 + 3 = 12

Figure 2–2: Allele count diversity for three genotypes

With binary chromosomes this method is very restrictive, since the maximum for a genotype of length ℓ is only $2 \times \ell$ and the minimum ℓ. However, even in this case it is still possible to make judgements about the diversity of solutions, provided the allele count diversity does not approach its maximum. The reason for the proviso that we should not trust allele count diversity near its maximum is that the difference new solutions make at this stage might not be proportional to how different they actually are. Of course, with more alleles to choose from, many more solutions are required before the metric approaches this limit.

2.2.3 Measuring the speed of finding solutions

Issues with speed are primarily method specific. How do we compare methods that produce solutions with rates that vary? Or methods that produce methods consistently against those that produce them erratically? This is quite a complex matter, and I will look at a few issues that arise. These are arranged into two main sections: when to stop a GA run and what period of the GA run to time. Finally, a word of hindsight is added.

When to stop

Firstly, when do we stop a GA run? This is simple if every chromosome in the population converges to a single genotype. However, a successful method might keep enough diversity in the population to stop convergence. How do we know when to stop looking at this method? It might suddenly produce a flurry of solutions after a drought for many generations. One method is to wait a certain number of generations to see if the population fitness is rising. We can measure the maximum of the average fitness of the population and detect whether this has risen in the last n generations. After a certain period it is time to give up waiting for more solutions. This will inevitably miss any sudden flurry of solutions from time to time, but, provided the number of generations is chosen carefully (in excess of the time would normally be prepared to wait with nothing happening), we should catch most solutions that are going to appear. The main point here is that we have to realise that this method *is* limited: however, without waiting forever, any run must be terminated at some point.

What to time

Secondly, what time do we measure as the 'time the GA took to find solutions'? What happens when one solution is found quickly, then it takes much longer to find the next solution? At some point a line has to be drawn and the definitive time to find solutions decided. I chose to look at the time it took to find peak diversity. This is simple to apply but not necessarily fair to the cases where solutions are not produced at a constant rate.

There is a problem with taking the time to the peak diversity: how long does a GA that finds no solution take to reach its 'peak' diversity? We cannot allocate it zero time, since this implies that the method is somewhat efficient. Equally, we cannot allocate infinity, since we did not in fact wait to find out. As a pragmatic choice, I took the total time the run took before it was stopped: so the amount of time taken for the zero solution case depends on how long we are prepared to wait for before giving up on finding solutions.

Time issues with hindsight

I have presented a simple method for timing GA runs, and this method was used while experimenting on the different forms of GAs. The justifications rely on intuitive arguments for how to measure the success of different types of GA. The arguments can be applied whether or not we know the number of solutions attainable. However, standing back it is possible to see a logical argument for how to measure time when the number of solutions is known. The key is to realise that we are, in fact, trying to find the *efficiency* of methods at generating solutions, not the time for solutions to be found: the methods that are best at producing multiple distinct solutions are going to be those which are most efficient at finding distinct solutions.

With this in mind, when do we stop a GA? The answer is to stop when we cannot possibly beat the maximum efficiency found so far. If the control method has found 1 solution in 10 seconds (0.1 solution per second) and there are 20 solutions possible, then waiting over 200 seconds for another method is pointless.

What time to measure is also dictated by efficiency: when is the GA performing at maximum efficiency? If a method finds 1 solution in 10 seconds, but has to wait for 30 seconds to produce 2 solutions, then the method works best finding 1 solution. A caveat, before we go too fast: this also depends on the number of solutions; if there are only 2 solutions then, according to the effective trials, the 2 solution case shown here is just as efficient as the 1 solution case. However, despite this caveat, we can still say of a GA that, with a diversity / solution metric A, the peak efficiency is B. This peak efficiency is what should be measured, and the period timed is the period for which the GA in question works at peak efficiency. Finally, note that the problem of how much time the 'no solution' case should be allocated is also problematic for this method of measuring time.

Chapter 3

Methods for obtaining Multiple Distinct Solutions

This chapter describes various methods which have been proposed for obtaining multiple distinct solutions and some tentative methods to be investigated, combining some of these existing methods.

3.1 Spatial Selection

In [Wright 69], Wright proposed the theory of gene frequencies in populations. Spatial selection is based loosely on some of the tenets of Wright's shifting balance theory of evolution. The idea is to produce distinct solutions is to by leaving units within the population (*demes*) isolated in some way from other units within the population; each deme is allowed to find a solution in relative isolation from the other demes, in the hope that each deme will converge on a separate solution.

Populations can be subdivided with hard edges into islands. The individuals on the islands might be *panmictic*; that is, parents in the subpopulation may be chosen from anywhere within its island. The subpopulation on the island forms a *deme*, so a deme is a panmictic community of individuals within the total population.

The hard edges between the islands can be removed and 'stepping stones' placed between them, so that individuals can cross over from one subpopulation to another, that is, from one deme to another. We can blur the edges more, so that more and more individuals are allowed to cross the borders, until eventually the only separation between what were distinct subpopulations is a physical distance. One deme is *isolated by distance* from another.

The islands model, with stepping stones between islands of demes can be translated into a GA islands model. Islands models are discussed in the next section. This section looks at the stage further than islands, where demes are isolated by distance.

By being somewhat isolated from each other, the demes will be allowed to reproduce in relative peace, occasionally reproducing with a parent from a neighbouring deme, not far removed by distance, but generally reproducing within themselves, panmictically in a localised area. I will talk of demes becoming *established* in this way. An established deme will have interbred for a number of generations to produce a set of individuals with similar genotypes.

The idea with spatial selection is to bring into play this isolation by distance, so that many demes can converge on different solutions concurrently, while still sharing genetic information between each other. Collins and Jefferson, [Collins & Jefferson 91] and Davidor, [Davidor 91] propose similar methods to emulate the formation of demes in Wright's theory.

Spatial selection involves notionally placing chromosomes in an n-dimensional grid, which simulates the placing of individuals in the environment. For reproduction a location on the grid is picked at random. Parents are chosen by starting at this chosen location and taking R random steps over the grid (see figure 3–1). The grid is wrapped into a toroid to make it continuous.

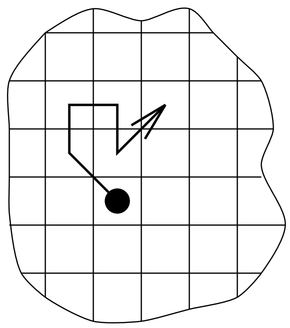


Figure 3–1: A five step random walk over a 2-D grid

The parent chromosome is the fittest chromosome encountered during this random walk across the grid (or most recent found if two or more chromosomes on the walk have are both fittest). Two parents may be chosen by two consecutive walks over the grid. The child is produced, as usual in a GA, by applying genetic operators to the parents or child. The child chromosome is inserted into the population at the originally chosen location.

If there are R steps in a random walk and the grid is n-dimensional, then any chosen location can find parents from a hyper-cube of $(2R+1)^n$ chromosomes placed around it. However, the average distance the random walk will cover from the chosen location will be much less than R, so the average number of chromosomes 'seen' as potential parents will be much smaller than $(2R+1)^n$. As we increase n, the number of chromosomes in this set of potential parents rapidly becomes larger, until we are in a position where, unless we also increase the population with n, the number of chromosomes in the potential parents set becomes near the number of chromosomes in the population. In the limit, we are effectively taking a tournament group of R chromosomes from the population. So, to maintain effective isolation by distance, where each chromosome can only see a small portion of the total, a low value of n is required. However, note that choosing n to be one is very restrictive, leaving only a few chromosome locations available as parent locations.

Typically n is chosen as 2, as it was for the spatial selection method used for this project. The value of R must also be relatively low, since the formation of a deme will depend on a set of chromosomes forming a subdivision of similar genotypes on the spatial grid which is not interrupted by continual addition of parents from outside the deme.

In this project the chromosome, the chromosome in the randomly chosen location is only replaced if the child chromosome generated is fitter than it. This will allow fit demes to establish themselves relatively quickly, since any parents chosen from the deme will tend to find their children are more often successfully inserted into the chosen location than had the parents been chosen from another nearby less fit deme.

3.2 Islands

Islands were not originally implemented as a model to create multiple distinct solutions. Tanese [Tanese 89] investigated *distributed genetic algorithms* as a method to run many GAs in parallel. The GA Tanese proposed was made up of subpopulations of chromosomes. Each subpopulation develops in isolation and after a *migration interval*, a period of generations, some of the chromosomes from one subpopulation are moved to another subpopulation.

Despite not being used to obtain multiple distinct solutions by Tanese, islands models are clearly very similar to Wright's islands with stepping-stones. Rather than allowing demes to form on their own as in spatial selection, we may enforce the size of panmictic populations with a number of islands. In theory, this should preserve diversity in the population as a whole by allowing subpopulations to evolve separately.

A way of implementing islands, as used in this project, is to select a chromosome from the one of the subpopulations (using a standard selection operator) every migration interval and to try to insert it into all the other subpopulations.

3.3 Crowding

Crowding uses a different approach to maintaining the diversity of a population. Rather than restricting the choice of parents, crowding affects how the child is inserted into the population. A crowding scheme was developed by De Jong in [De Jong 75, in [Davidor 91]]. The essential feature of crowding is that a child chromosome replaces a member of the population with a similar genotype. A simple GA will often replace the least fit member of the population and destroy potentially useful genotypic information in doing so. By using crowding, we can retain the this information. In addition, it will prevent convergence of the population to one genotype since any genotype chosen as a parent is more likely to be replaced through crowding; as mentioned in the introduction, a child will usually have over 50% of the genes of its parent, and therefore the parent is likely to be the chromosome crowded out of the population.

De Jong's scheme uses a crowding factor (CF) variable which is used to decide which chromosome a child will replace. The child is compared to CF chromosomes chosen at random from the population. The chromosome from this set with the most similar (in terms of Hamming distance) genotype to that of the child is replaced.

A method for combining this with spatial selection[1] might be to take a random walk from the position of the location chosen as the start selection random walk. We take a random walk CF steps long and the most similar chromosome found on the random walk is replaced.

This is not the only way to implement a spatial version of crowding (for example, CF individuals could be chosen from the immediate vicinity or simply from the population as a whole); however, it does appear to be a natural extension to the Collins and Jefferson method of spatial selection.

---

[1] Due to Dave Corne, [Corne 94]

3.4 Sharing

The idea behind sharing is to share the benefits of a particular facet of an individual among all the individual in a population which share that facet. This reduces the possibility that one set of individuals with a certain characteristic will dominate the population.

3.4.1 An introduction to sharing

Sharing is usually introduced by reference to a 'two-armed bandit' (see [Holland 75]), but this is a bit restrictive for this discussion; instead, imagine that, on some planet, there is a source of food located on land and another source in the sea. If the animals on the planet are either entirely land-living or sea-living then they occupy separate food *niches*. That is, any animal in one food niche shares food with other animals in that niche, for example, two land-living animals; the food available to an animal is not affected by the consumption of food by an animal in a separate niche.

Deb and Goldberg, [Deb & Goldberg 89] (also [Goldberg & Richardson 87]), propose a *sharing function* to work out whether or not two animals have to share their food. For the moment, this is a trivial function: if the two animals are both of the same phenotype, i.e., both land-living or both sea-living then they must share food with each other; if two animals are in separate niches then they do not have to share food with each other.

To formulate the sharing function Deb and Goldberg introduce a *niche separation distance*. The niche separation distance, σ_{share}, is the 'distance' that fully separates two niches. In our example, the 'distance' might be a sort of 'phenotype distance' line which extends from land-living to sea-living animals. If two animals are both the same species then on this phenotype distance line they have a separation of 0. Land-living animals are separated from other land-living animals by a distance of 0; the land-living and sea-living animals are separated by the niche separation distance. Now, a land-living animal, say it has a phenotype ϕ_L, will have to share with another land-living animal completely: the sharing function for two animals separated by 0 'phenotypes' is 1 (land food eating animal equivalent). We can write this symbolically: $Sh_{simple} = 1$ if $d_\phi(\phi_1, \phi_2) = 0$. For the case of a land-living and a sea-living animal the separation distance is σ_{share} 'phenotypes', sharing function for the land-living animal will have a value of 0 (land food eating animal equivalents). So, $Sh_{simple} = 1$ if $d_\phi(\phi_1, \phi_2) = \sigma_{share}$. We may define a simple sharing function with these properties:

$$Sh_{simple}(\phi_1, \phi_2) = 1 - \frac{d_\phi(\phi_1, \phi_2)}{\sigma_{share}} \tag{3.1}$$

To find out how much land located food a single land-living animal, say A, receives we may divide the land located food by the total number of land-living animals. We can calculate this number of animals by summing the sharing function applied to A and all the animals in the population (including A itself), and thus divide the food. Say $f(\phi_A)$ food is the total amount of food A can see, i.e., the total land food, $f'(\phi_A)$ is the food share A actually receives and there are N animals in the population:

$$f'(\phi_A) = \frac{f(\phi_A)}{\sum_j^N Sh_{simple}(\phi_A, \phi_j)} \tag{3.2}$$

All this may seem a bit complicated for our simple example, but the sharing function is introduced so that we do not have to have just land-living and sea-living animals, each in their own niche. We might posit an amphibian that might eat some land food and some sea food, and therefore A will have to share food with an amphibian. However, the amphibian does not count as 'one' land-living animal, because it spends some time eating sea located food. Figure 3–2 shows the animals in the example on this line. Now the sharing function tells us that if the amphibian species (with phenotype ϕ_A) spends most of its time eating land food, i.e., if $d_\phi(\phi_L, \phi_A)$ is small, then the land-living animal will have to share a lot of the land located food with the amphibian. Similarly, if $d_\phi(\phi_L, \phi_A)$ is large, $Sh(\phi_L, \phi_A)$ is small. An intuitive justification for this sharing function is that if, for example, $d_\phi(\phi_L, \phi_A) = 1/3$, that is, the amphibian spends two thirds of it time eating land food and one third sea food, then it seems fair that it should count as 2/3 (land food eating animal equivalents), which agrees with what the sharing function tells us.

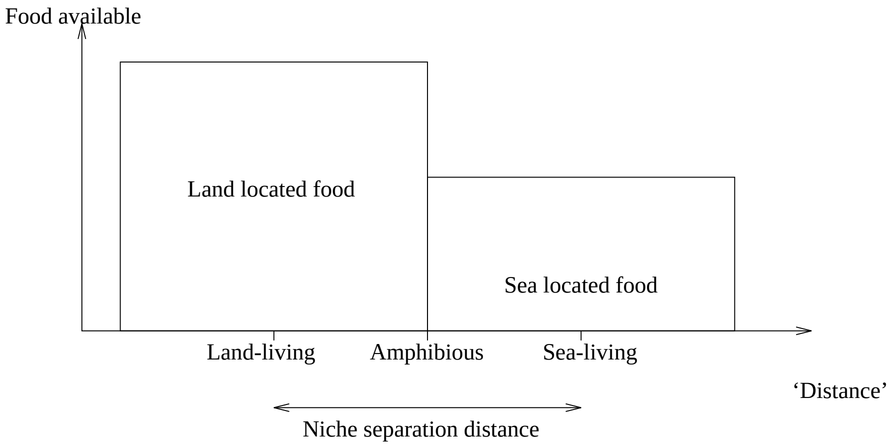


Figure 3–2: A simple sharing example

Before moving on to how the sharing function relates to GAs there are a couple more modifications to make to the current simple sharing function. Firstly, what happens when the animals are separated by more than the niche separation distance? Say, for example there is a source of deep-sea food that amphibians cannot reach and some deep-sea-living species exploits. Now, the land-living animals and sea-living animals are still σ_{share} apart, so the land-living animals and deep-sea-living animals must be separated by more than the σ_{share}. As it stands, this gives a negative sharing function between land-living animals and deep-sea-living animals. This does not make sense: land-living animals do not gain food simply because another species far removed from itself exists on the planet. For this reason, we specify that σ_{share} cannot be negative. A second modification is to allow the amount of sharing to vary non-linearly with distance. For example, an almost land-living amphibian may find it hard to eat sea food, and therefore have a sharing function which is not proportional to how much time it spends eating food. For this purpose, a power α is introduced to the sharing function. Equation 3.3 shows a general sharing function with these modifications in place:

$$Sh(x_i, x_j) = \begin{cases} 1 - \left(\frac{d_x(x_i,x_j)}{\sigma_{share}}\right)^{\alpha} & \text{if } d_x(x_i, x_j) < \sigma_{share} \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

Now the shared fitness assigned to an individual i, or food in the example, is given by:

$$f'(x_i) = \frac{f(x_i)}{\sum_{j=1}^{N} Sh(x_i, x_j)} \tag{3.4}$$

Finally in this introduction, a caveat. In the example, we always calculated the sharing function between land-living animals and other animals. That is, the sharing function between a niche centre and some phenotype distance away from that niche centre. The sharing function itself is intended to be applied to any individual, not just those at the niche centre. For this the fitness must be known at the point of a notional niche centre for the individual (in the example, how much food is in the notional 'amphibious available food' niche?). It is not clear that this shared fitness at a notional niche centre will actually give a true representation of how much fitness really should be shared with other individuals.

3.4.2 GAs and sharing

As previously stated, when applied to GAs the method reduces the value of the fitness of an individual according to the number of individuals in a population which share some 'facet'. Which particular facet of an individual this should be is determined by the distance metric used to calculate d_x: it might be the genotype, the phenotype or even the fitness. Deb and Goldberg look consider two approaches: genotype and phenotype sharing. The functions Deb and Goldberg use are Euclidean distance for phenotypic sharing (where an n-dimensional distance between phenotypes makes sense) and Hamming distance for genotypic sharing (see section 2.2.2 for a related discussion of diversity metrics).

When applied to GAs, we would like the GA to converge to as many different solutions as there are in the search space. So we choose a niche separation distance such that the search space is divided in some way into the number of solutions required. This way, individuals converging on one solution do not affect those converging on another. By the same token, any individuals converging on the same solution will have to share their fitness among themselves.

Given a distance function, a value for σ_{share} can be calculated using the number of niches that we want to group the chromosomes into. Usually we do not know the number of solutions to a problem and hence the corresponding number of niches in the phenotype or genotype space. However, since it would be desirable to find a rough value for σ_{share} given a test problem, section 3.4.3 discusses how σ_{share} may be calculated for genotype (Hamming distance) sharing. A similar calculation for phenotype sharing is more simple, but this project concerns solving timetabling problems which cannot be represented in n-dimensional space, and are therefore not suited to phenotypic sharing using Euclidean distance.

Throughout, it is important to remember that we are sharing fitness among individuals in virtue of them having a 'similar (Hamming distance) genotype' or a 'similar (Euclidean distance) phenotype' and so on, and that these characteristics of an individual may have little relation with the nature of the search space. For example, we might use phenotypic sharing with Euclidean distance over five niches, but this will be of little use if we are trying to find a set of twenty peaks which are logarithmically spaced; of course, if we know the peaks are logarithmically spaced, there is no reason why we should not use a logarithmic distance function. It should be clear that problems will not usually have maxima which are uniformly distributed as regards Hamming distance or Euclidean distance.

As sharing has been described, it involves a great many calculations. For every generation the shared fitness of every member of the population must be recalculated to allow for the introduction of a new chromosome. Typically, this will involve calculating d_{ij} for every member of the population (at the least, all genotypes or all phenotypes within the population). That is, N calculations of d_{ij} are required per generation.

If the spatial selection model is combined with the sharing model the number of calculations may be slightly reduced by assuming that isolation by distance controls the size of demes (see section 3.1): only the shared fitness of chromosomes located in the vicinity of a chromosome to be replaced need be calculated at each generation. However, the actual size of any deme in spatial selection is undefined: how much a particular chromosome is isolated may vary. When estimating the number of chromosomes to consider in the sharing calculation, if we consider a deme to be very large, then we lose any advantage in decreasing the number of calculations. On the other hand, if we underestimate the deme size, the penalty imposed on a particular successful genotype or phenotype will be lower than usual.

3.4.3 Evaluating σ_{share} for genotype sharing

The following discussion is an expansion of the analysis offered by Deb and Goldberg in [Deb & Goldberg 89] from the binary chromosome case to to an n_a allele (or n-ary) chromosome.

In genotype sharing we can measure the distance between two genotypes γ_i and γ_j with the Hamming distance function, $d_H(\gamma_i, \gamma_j)$. σ_{share} divides the set of all possible genotypes into niches using this function. However, there is no direct mapping of Hamming distance to a set of genotypes. In order to evaluate σ_{share}, we must correlate a Hamming distance with the number of genotypes in the set it covers, $\mathcal{N}(\sigma_{share})$.

For an n_a allele chromosome with ℓ loci, there are ${n_a}^{\ell}$ possible genotypes. Hence, if there are q niches, we want to cover ${n_a}^{\ell}$ genotypes with q sets:

$$\mathcal{N}(\sigma_{share}) = \frac{{n_a}^{\ell}}{q} \tag{3.5}$$

$\mathcal{N}(\sigma_{share})$ can be calculated as follows. If we pick a random genotype from the space of all genotypes, at $d_H(\gamma_i, \gamma_j) = 1$ there are ${}^{\ell}C_1$ loci which might contain a different allele[2]. At the locus which is different we can place a possible $n_a - 1$ alleles. Therefore, if we choose to cover the set of all possible genotypes with $\sigma_{share} = 1$, each set will contain $1 + (n_a - 1)\,{}^{\ell}C_1$ genotypes. The '1' is the genotype at the centre of the niche, and the '$(n_a - 1)\,{}^{\ell}C_1$' are the possible genotypes at $d_H(\gamma_i, \gamma_j) = 1$ from that genotype.

Similarly, $\sigma_{share} = 2$ will cover $1 + (n_a - 1)\,{}^{\ell}C_1 + (n_a - 1)^2\,{}^{\ell}C_2$ possible genotypes. In the general case where $\sigma_{share} = k$:

$$\mathcal{N}(\sigma_{share}) = 1 + (n_a - 1)\,{}^{\ell}C_1 + (n_a - 1)^2\,{}^{\ell}C_2 + \ldots + (n_a - 1)^k\,{}^{\ell}C_k \tag{3.6}$$

If we divide through by the number of total possible genotypes, ${n_a}^{\ell}$, this becomes a cumulative binomial distribution. Hence, from equations 3.5 and 3.6:

$$\sum_{i=0}^{\sigma_{share}} \left(1 - \tfrac{1}{n_a}\right)^i \tfrac{1}{n_a}^{(\ell - i)}\,{}^{\ell}C_i = \frac{1}{q} \tag{3.7}$$

In this manner, for genotype sharing, we can calculate the number of niches corresponding to each value of σ_{share} through calculation of the cumulative binomial distribution on the left hand side of equation 3.7. An example of this method in use is given in chapter 4.3.2, when a rough value of σ_{share} is calculated for use with the test set problems.

[2] ${}^{a}C_b$ is equivalent to the notation $\binom{a}{b}$.

Chapter 4

Outline of Test Plan

This chapter presents an overview of the tests performed. It takes the timetabling problem representation from chapter 2.1 to form a test set of problems, the theory of classify distinct solutions from chapter 2.2 to form some criteria for judging the success of methods and the description of the methods in chapter 3 to form the experimental set up of the methods to be tested. Finally, these parts of the test plan are drawn together as a whole. A full specification of the test plan is given in appendix A.

4.1 Test Set Problems

As discussed in chapter 1.4.1, one of the real problem application areas of genetic algorithms is timetabling. A successful representation for timetable problems has been developed by Peter Ross, Dave Corne and Hsiao–Lan Fang (presented in [Ross *et al.* 94] among others and outlined in chapter 2.1). Using this representation, timetable problems can be constructed artificially to give any number of solutions. Four timetabling problems with varying numbers of solutions were devised for the purpose of experimentation, and their construction is outlined below.

Artificial timetable problems are chosen primarily because they are easy to create with known numbers of solutions. It is also hoped that a GA will behave in a similar way when run on artificial problems or real problems. There are, however, some major differences between the artificial problems and the nature of real problems. The edge-constraints in the artificial problems are far more uniform and far fewer than in real problems. In real problems the edge-constraints are very often specified by a list of separate events; for example, events 1, 3, 16 and 21 must be separate slots (i.e., edge constraints between 1-3, 1-16, 1-21, 3-16, 3-21 and 16-21). In the artificial problems, the emphasis is slightly different: event 1 must be in a separate slot from whatever slots the events 3, 4, 5 and 9 occupy (i.e., edge-constraints between 3-4, 3-5 and 3-9). It is hoped that the resultant set of edge-constraints, after many specifications such as these, form a set of solutions with a similar distribution across the search space for both artificial and real problems.

The problems devised consisted of 20 events to fit into 5 slots[1]. From this four problems were constructed, each with different numbers of solutions:

- 120 solution problem (`tt120`)
- 60 solution problem (`tt60`)
- 30 solution problem (`tt30`)
- 10 solution problem (`tt10`)

The 120 solution problem split the timetable so the first four events had to be in one slot, the next four events in another slot, and so on (see figure 4–1). Hence there are $5! = 120$ solutions to this problem.

0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
0	0	0	0	1	1	1	1	2	2	2	2	4	4	4	4	3	3	3	3
0	0	0	0	1	1	1	1	3	3	3	3	2	2	2	2	4	4	4	4
0	0	0	0	1	1	1	1	3	3	3	3	4	4	4	4	2	2	2	2
⋮																			
⋮																			
4	4	4	4	3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0

Figure 4–1: Solutions to the 120 solution timetable problem

The edge constraints to specify this are quite simple: event 0 must be in a separate slot from events 4 to 19, similarly event 1 must be in a separate slot from events 4 to 19, ..., event 6 in a separate slot from events 0 to 3 and 8 to 19, and so on.

If we introduce the constraints that events 0, 1, 2 and 3 must precede events 4, 5, 6 and 7 we halve the number of solutions to 60. The number of solutions can be reduced to 30 by also specifying that events 12, 13, 14 and 15 precede events 16, 17, 18 and 19.

Finally, to reduce the problem to have just 10 solutions, the additional constraints that events 4, 5, 6 and 7 precede events 8, 9, 10 and 11 may be added.

The set of constraints used to define the problems are recorded in appendix D.

[1]The timetable problems originate from a set devised by Dave Corne, [Corne 94]

4.2 Criteria for judging success

Chapter 2.2 considered three metrics for comparing the number of solutions found by different methods. These metrics were based on the number of solutions found, the diversity of the solutions and the time taken to find the solutions. Chapter 2.2 looked at a few different instances of each type of metric, and justified the use of three in particular: the effective number of trials, the allele count diversity and the time taken to reach peak diversity.

These metrics do not say much about the performance of the GA in themselves. Although it is interesting to look at how many effective trials or how much diversity a GA is producing, it would be more informative to see the efficiency of a GA at producing multiple distinct solutions. For this we can divide the the diversity or effective trials by the time to give the rate at which diversity or effective trials are increasing. This rate will tell us whether it is better to run a simple GA, say, twice to gain three distinct solutions or to run sharing once to find three distinct solutions. For this reason the diversity, the diversity per second, the effective trials and the effective trials per second were compared for the various methods.

It should be realised that, although the metric 'diversity per second' is used, there is no easy no way to tell that a method that finds diversity 20 in 3 seconds will find diversity 40 in 6 seconds. For example, with the 120 solution problem this is unlikely — the mean separation between solutions in Hamming distance is 16 and the variance[2] is 4, so on average on the 120 solution problem, the method will only find diversity 36 in 6 seconds. However, provided that the solutions are evenly placed within the search space the effective trials per second will be a fair comparison of two methods, as the effective trials metric takes into account the relative difficulty of finding a new solution; although, it should be noted that in general (in the experiments, all but the `tt120` problem), the solutions will not be evenly placed throughout the search space.

Finally, we also need to define when the GA should be stopped. Chapter 2.2.3 gave a justification for leaving the GA for a set number of generations without an increase in the maximum of the average fitness before stopping the GA. To choose this number of generations a few runs of a simple GA were made to find how long it was taking the population to converge to a single genotype. For the problems with a few simple operators defined it took the GA in the region 500 to 2500 generations to converge to a single genotype. From these tests, it seemed that the stopping condition of 1000 generations without improvement in fitness was a sensible choice.

[2]The variance has been determined empirically; the mean agrees with the binomial distribution I would have expected.

4.3 GA configurations

The following methods, discussed in the previous chapter, were tested for obtaining multiple distinct solutions. A simple GA has been added as a control.

1. Simple GA
2. Islands
4. Spatial Selection
4a. Crowding
4b. Spatial Crowding (crowding combined with spatial selection)
5a. Sharing
5b. Spatial Sharing (sharing combined with spatial selection)

The configurations used for each method are outlined below, with a discussion of the choice and values of parameters.

4.3.1 Template configuration

Potentially there is a vast number of GA configurations which may tested given the methods above. For example, factors which might affect sharing include not just the niche separation distance and α (the arbitrary power of d_{ij}/σ_{share}), but also the population size, the type of reproduction, the method of selection and the crossover and mutation operators.

In order to keep the project manageable, a basic GA configuration was chosen as a template (shown below) to be used for all the experiments, and the multiple distinct solutions methods were laid on top of this configuration. This template GA was used as the control simple GA (method 1).

Template

Crossover operator:	Uniform
Mutation rate:	$1/\ell$
Selection operator:	Rank
Reproduction operator:	Steady-state

In [Syswerda 89] Syswerda showed that using uniform crossover is almost always superior to using one or two-point crossover at finding solutions. Choosing a mutation rate of $1/\ell$ (where ℓ is the length of the genome) allows approximately one mutation on the child chromosome per generation. The rank selection operator was only used with non-spatial methods since the spatial selection model includes a selection operator (see chapter 3.1). Steady-state reproduction was used in favour of generational reproduction (where the whole population is replaced each generation) primarily because there is less change to the population each generation; a slower rate of change in the population allows easier analysis.

It should be noted that while the use of a template reduces the number of experiments and the amount of data produced significantly, it should be realised that working with a smaller parameter set may miss important results. However, the template chosen is small and uncomplicated, so the methods should not be biased too much by its use. In addition, the operators picked for the template have been used previously with timetabling problems (see [Corne *et al.* 94]), so they would seem a sensible choice to use.

4.3.2 MDS configurations

Given the template, just one parameter per method was selected to be varied. For each value of the parameter the method was tested on the four problems described in section 4.1 and with five population sizes.

A range of population sizes were experimented on in order to be fair to all the methods: previous experiments on spatial selection have used very large populations [Collins & Jefferson 91], whereas sharing experiments have used much smaller populations [Deb & Goldberg 89]. A typical population size for solving timetabling problems is 50 (see [Corne *et al.* 93] or [Ross *et al.* 94]). Therefore a minimum population size of 49 (7^2 to allow easy translation to spatial selection) was chosen, to a maximum of 625 (to give a large range of population sizes). Slightly different population sizes were used in conjunction with the islands model to allow whole numbers of chromosomes per island.

The following sections detail the parameter values chosen for each method.

Spatial Selection

With spatial selection, a random walk length with a small population size will cover much more of the spatial selection grid, on average, than the same random walk length with a large population size. For this reason, population divisors are introduced. The population divisor is inversely proportional to the area of coverage of a random walk, the idea being that every population size is segmented into the same number of approximate area regions for each divisor. The relationship between the population divisor, $p_d iv$, and the random walk length, R, for a population of N chromosomes is shown in equation 4.1:

$$R = \frac{\sqrt{N}}{p_{div}} \text{ (rounded to nearest integer)} \tag{4.1}$$

Collins and Jefferson in [Collins & Jefferson 91] use random walk lengths in the range 1 to 30 on population sizes much larger than those used in this project.

Given that the population sizes used by Collins and Jefferson were much larger than we will use, a range of population divisors was chosen to give a minimum random walk length of 1 with $N = 49$ to a maximum of 17 with $N = 625$. Table 4–1 shows the random walk length for each population divisor / population size combination.

Population Size	Population Divisor		
	1.5	2.5	5.0
49	5	3	1
100	7	4	2
225	10	6	3
400	13	8	4
625	17	10	5

Table 4–1: Random walk lengths for each population divisor / population size combination

Islands

2, 3 and 5 islands were used on population sizes of 60, 120, 240, 360 and 600 (population sizes were chosen to allow easy subdivision). The choice of these values stems from a similar choice of parameter values for spatial selection. Tanese in [Tanese 89] chose to investigate subpopulation sizes from 8 to 128 on a total population size of 256. With 2, 3 and 5 islands, the range of subpopulation sizes tested varies from 48 to 120. Tanese found that a subpopulation of size 8 still gave good results. However, Tanese was trying to find a solution rather than many solutions. Given this, he was using relatively high migration intervals and rates, so small populations were viable. In the case of finding multiple distinct solutions, we actually want to keep the migration intervals and rates quite low, and therefore each subpopulation will have to be able to try to find solutions while semi-isolated. Without the aid of new genotypes being added to the subpopulation as regularly (a migration interval of 10% as opposed to 20% or 30%), it was decided to concentrate on slightly larger subpopulations. Given the limited time for this project, the migration interval and migration rate were left constant in order to restrain the model to a single parameter choice.

Crowding

Crowding factors of 2, 3 and 5 were used. De Jong [De Jong 75, in [Davidor 91]] found that the empirically optimal crowding factor was about 3. The range used gives a range either side of the optimal crowding factor found by De Jong.

Spatial Crowding

Chapter 3.3 gave a proposal for an implementation of spatial crowding, whereby the crowding factor gives the length of a random walk from the site of the location chosen for reproduction. Note that the first step of the random walk is considered to be the chosen location on the grid, so a crowding factor of 1 is no different from standard spatial selection. Crowding factors of 2, 3 and 5 were also used for this method. The method was used in conjunction with the best performing spatial selection population divisor (this was found to be 5.0 — see chapter 5.3 for details).

Sharing

For sharing, the obvious parameter to use is the niche separation distance. Since there was limited time, it was decided to keep α (the arbitrary power) constant.

Before moving on, the niche separation distance, σ_{share}, was introduced using the units of Hamming distance, following [Deb & Goldberg 89]. However, to allow σ_{share} to be used more generally, I will express σ_{share} in units of *normalised* Hamming distance from now on. The normalised Hamming distance d_N is simply the Hamming distance d_H divided by the length of the genome ℓ:

$$d_N = \frac{d_H}{\ell}. \tag{4.2}$$

The use of normalised Hamming distance as the units for σ_{share} means that, whatever the length of the genome, we can express σ_{share} as a number in the range 0 to 1.

Chapter 3.4.3 showed how it is possible to obtain a theoretical value for the niche separation distance, given the number of solutions for a problem, the genotype length and number of alleles for each locus. Section 4.1 in this chapter considered the timetabling problems tested. The timetabling problems consist of 20 events with 5 slots. Genotypes of length 20 (a locus for each event) with 5 alleles (an allele for each slot) are used to represent the timetables. That is, $\ell = 20$ and $n_a = 5$.

Given that we know the number of solutions for the problems, it would seem appropriate to calculate an approximate value for σ_{share} to allow us to make a sensible choice of the parameter σ_{share}. Although in the case of a real problem this calculation would be impossible, it allows us to make an estimate for what values of the niche separation distance work. For example, to see whether 'looking for' 1, 30 or 10000 solutions is generally a sensible idea.

Equation 3.7 (which is repeated below with the values of ℓ and n_a for the test set problems inserted) gave the relationship for the number of solutions required with the niche separation distance (q is the number of solutions):

$$\sum_{i=0}^{\sigma_{share}} \left(1 - \tfrac{1}{5}\right)^i \tfrac{1}{5}^{(20-i)} \, {}^{20}C_i \; = \; \frac{1}{q} \tag{4.3}$$

From this equation we can plot σ_{share} against q. This plot is shown in figure 4–2. Note that this is a plot of a cumulative binomial distribution and the lines between the points mean nothing. The lines are shown in order to show roughly what the relationship between σ_{share} and q is between the points.

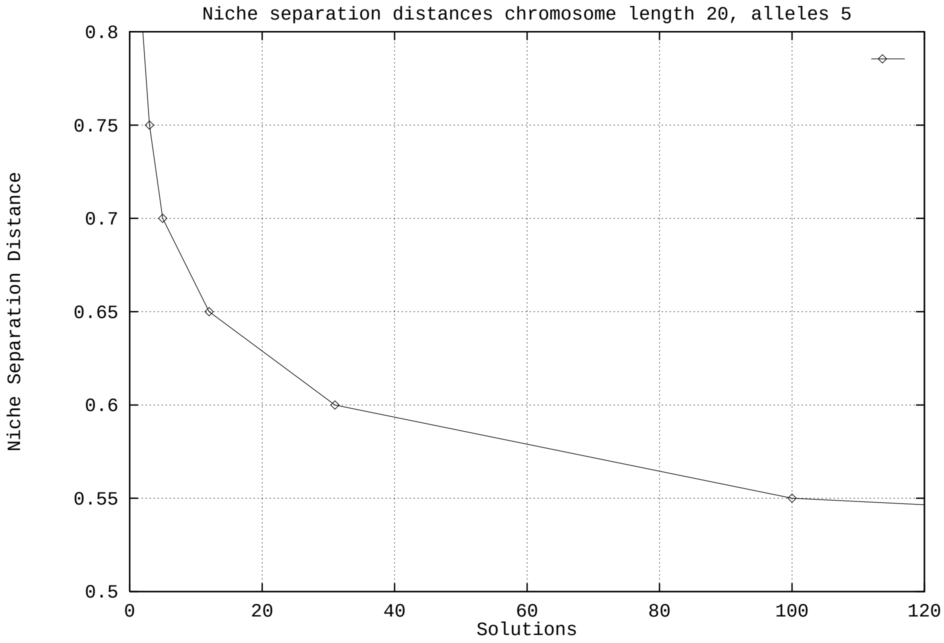


Figure 4–2: Number of solutions against niche separation distance

The figure shows that for problems with numbers of solutions in the range 10 to 120 correspond to $\sigma_{share} \approx 0.6$. To give a range of niche separation distances the values 0.4 ($\sim 10\,000$ solutions), 0.6 (~ 30 solutions) and 0.8 (~ 1 solution) were chosen for the values of σ_{share} to be tested. These values are really quite arbitrary. The value of 0.8 might be justified by the fact that the solutions to the `tt120` problem have an average normalised Hamming distance separation of 0.8 (see section 4.2).

Spatial Sharing

Spatial sharing was introduced at the end of section 3.4.2 as a method which would reduce the calculation overhead of sharing when used in conjunction with spatial selection. σ_{share} was chosen as the parameter to be varied for the method with values 0.4, 0.6 and 0.8 as for standard sharing. The area to be used for the 'sharing zone' — that is, the number of chromosomes around each location in the grid to be included in the sharing calculation — was chosen as $(2R+1)^2$ (R is the length of the random walk). $(2R+1)^2$ corresponds to taking a square centered around each location extending R in each direction. The hope was that this would correspond to roughly to the size of a deme, although see the results in the next chapter to see how misguided this turned out to be. Like spatial crowding, the method was used in conjunction with the best performing spatial selection population divisor (this was found to be 5.0 — see chapter 5.3 for details).

4.4 Summary of Test Plan

A detailed test plan is given in appendix A. The preceding sections have looked at the problems to test, the criteria for judging success and the methods tested. Table 4–2 gives a summary of the test plan, showing the combinations of method, population sizes and parameters tried on each of the four test set problems.

		Method	Population sizes	Parameter	Values tested
1		Simple GA	49, 100, 225, 400, 625	None	
2		Islands	60, 120, 240, 360, 600	Number of islands	2, 3, 5
3		Spatial Selection	49, 100, 225, 400, 625	Population Divisor	1.5,3.5,5.0
4	(a)	Crowding	49, 100, 225, 400, 625	Crowding Factor	2, 3, 5
	(b)	Crowding (spatial)			
5	(a)	Sharing	49, 100, 225, 400	σ_{share}	0.4,0.6,0.8
	(b)	Sharing (spatial)			

Table 4–2: Summary of test plan

Chapter 5

Discussion of results

This chapter discusses the results obtained from testing. Section 5.1 explains the form of the results: four graphs for each method for each problem. Next, a section is given to each method tested. A summary draws together the various methods, compares the results as a whole and gives a justification for a new method to be tried: tribes.

5.1 Form of results

The metrics used for comparing the methods, as discussed in the last chapter, were effective trials, effective trials per second, diversity and diversity per second. Data actually collected from the genetic algorithm runs were the peak diversity, time at which peak diversity was reached and number of solutions at the peak diversity. If no solutions were found, the total time for the run was taken rather than the time to peak diversity. Averages of these values were taken for ten different random seeds to the genetic algorithm and from these averages the values of the metrics were calculated. A precise description of this method of data collection is included in appendix A.

A value of each metric was obtained for each method, problem, method specific parameter and population size. These were plotted onto graphs of the metric against population size for each method and problem, including plots for different method specific parameters on the same graph. For each method this gave four graphs per method per problem, and all of these graphs together can be found in appendix A.

The graphs discussed below are of the effective trials and effective trials per second without reference to the diversity. The reason for doing this is that the results obtained for diversity and solutions are very similar. Figure 5–1 shows an example of the effective trials graphs and diversity graphs taken from the crowding method for the `tt120` problem. As can be verified from the all the results in appendix A, any trends shown in the effective trials graphs also show in the diversity graphs and vice versa. Any use of a diversity graph can only make the same point as use of the corresponding effective trials graph.

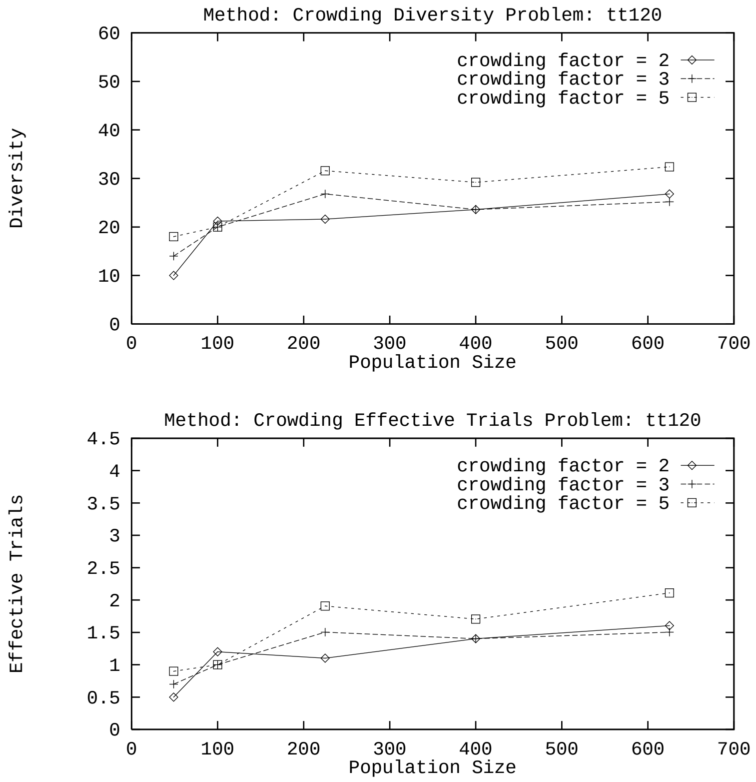


Figure 5–1: Comparing diversity and effective trials for a crowding run

Noticing that the diversity metric does not differ much from the effective trials is an interesting point in itself. Section 2.2.1 discussed why looking at the effective number of trials said nothing about the diversity of the solutions found, and section 2.2.2 spent an appreciable amount of time looking at the justifications for different types of diversity metric. Why, after this, should measuring the diversity be so similar to measuring the number of solutions?

First of all, notice that the the 120 solution problem is built from considering placing 5 groups of events into 5 slots. There are no conditions on the placement of the 5 groups of events, so they can be placed in any of the 5! slot permutations, which, since every permutation is covered, must be evenly spaced. As was mentioned in chapter 4.3.2, the mean Hamming distance between any two of these solutions is 16. Therefore, for any two solutions ($t_{eff} \approx 2$), the mean allele count diversity will be $20 + 16 = 36$. Also note that the minimum distance between two solutions is 8, so if a method was taking the easiest route to find another solution after first selecting one, it would find a diversity of 28 at 2 solutions. Looking at figure 5–1, the empirical diversity is approximately 34 at $t_{eff} = 2$, which is very similar to what we would expect for randomly chosen solutions from the space. That is, the methods are actually finding distinct solutions at random from the space (note, however, that this is a qualitative assessment and also that the figure of 36 only applies to the `tt120` problem). The graphs in appendix A confirm this figure of diversity 34 for 2 effective trials for most of the methods and parameters.

The methods are selecting solutions from over the covering whole search space rather than working up from a single solution and adding to it.

5.2 Control

5.2.1 Simple GAs only find one solution

As anticipated, the control method was very bad at producing many solutions. For all problems the method produced only around 1 effective trial. As the number of possible solutions were increased the method, if anything, got slightly worse at finding multiple distinct solutions (see figure 5–2). However, the effective trials certainly do not increase and the method converges on around 1 effective trial whatever number of solutions is available and whatever population size is used.

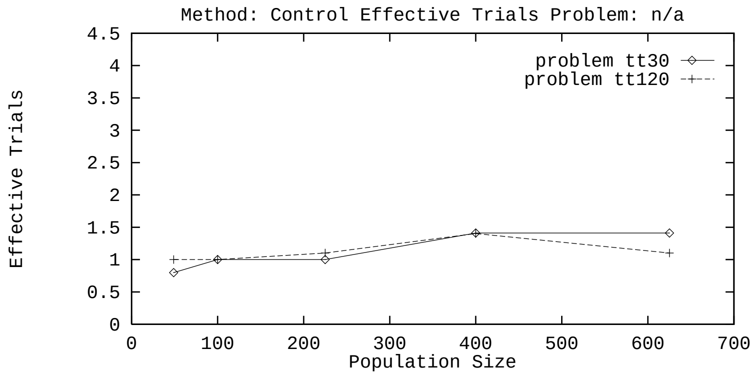


Figure 5–2: Effective trials for control method

5.2.2 Efficiency is much poorer when solutions cannot be found

The advantage, however, of the control method is its speed. Even when it is not producing solutions regularly, it is still fast. Figure 5–3 compares the efficiency on the `tt30` and `tt120` problems. The major difference is the result at a population of 49. Here the method consistently finds solutions on the `tt120` problem but not the `tt30` problem. Referring to 5–2 again shows that the control method is producing an average of $t_{eff} = 1.0$ at population 49 on the `tt120` problem and an average of $t_{eff} = 0.8$ solutions on the `tt30` problem at the same population size, which would not seem to be a big enough difference to account for the difference in efficiency seen in figure 5–3 at a population size of 49.

It should be remembered at this stage that if a solution is not found, the time used as the 'time to reach peak diversity' is the time of the complete run, but

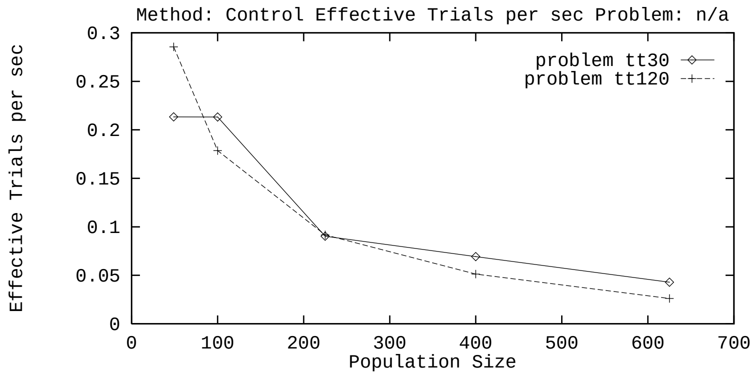


Figure 5–3: Efficiency of control method

when a solution is found, the 'time reach to peak diversity' will almost certainly be a time of at least 1000 generations less than this. The reason for this being that 1000 generations of no increase in average fitness are allowed before the GA run is stopped (see chapter 4.2 to review the testing method and appendix A for technical details). Hence, any method that is just failing to produce a solution will always tend to be much slower. Recalling the original discussion of the speed metric in chapter 2.2.3, this is intentionally included in the metric: we do not want a method that often produces no results at all to score highly.

The reason for the increase in ability to find a solution at population size 49 can be attributed to the increase in solution density from the `tt30` problem to the `tt120` problem: it is easier to find a solution in the `tt120` problem.

5.2.3 Efficiency decreases as population increases

Apart from the difference at population size 49, the results in figure 5–3 are very similar. The speed that effective trials are found decreases rapidly as the size of the population increases. This is due to the slower convergence of larger population sizes. Not only are slightly more solutions found, but also the number of generations to find each solution has been slowed. In the case of the control method this increase in generations, and therefore time, far outweighes the benefits of the extra solutions found.

5.2.4 Summary

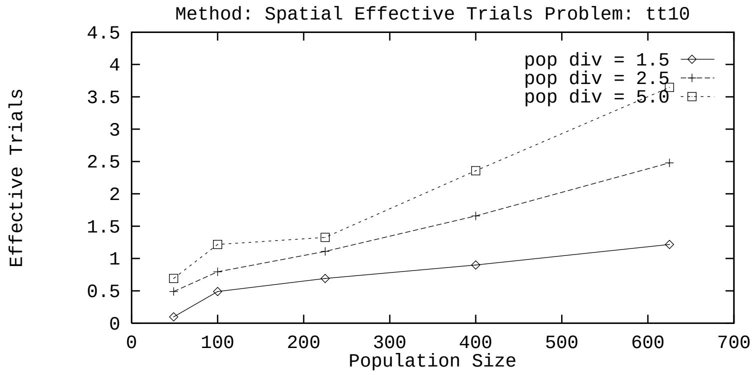
The comments made here are fairly general points that relate to most of the methods. The poor effective trials per second rating of a method which cannot find a single solution and the general decrease in efficiency as the population size increases both occur to varying degrees with every method.

Finally, this experimental evidence confirms previous findings: a simple GA does tend to converge on a single solution.

5.3 Spatial Selection

5.3.1 Effective trials increase as population size increases

Figure 5–4 shows the effective trials for spatial selection for the `tt10` and `tt120` problems. The first point to note is the increase in solutions as the populations increases (for all population divisors and problems). This is true in most of the methods tested; as the population size increases so the diversity of the initial population increases. Also convergence is slowed since for one genotype to dominate the population it must displace, for example, 624 rather than just 48 other genotypes. This leads to an increase in the amount of time given to less fit genotypes to reproduce and hence a longer time for these genotypes to contribute to producing a potentially fitter genotype.



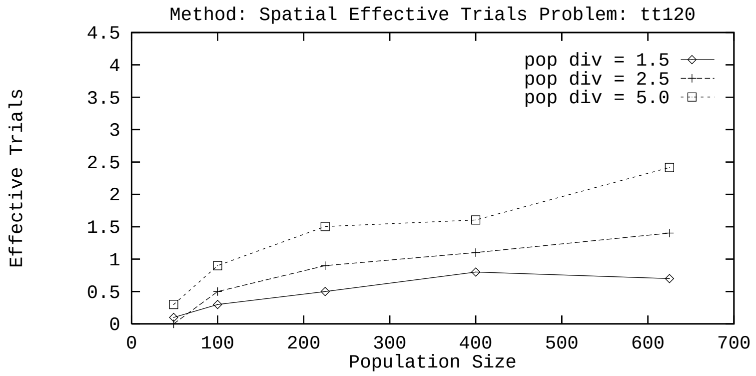


Figure 5–4: Effective trials for Spatial Selection

5.3.2 t_{eff} is higher for small numbers of solutions

It is very noticeable that the effective trials are much higher for the `tt10` problem than the `tt120` problem for all the population divisors shown in figure 5–4. Looking at the top end of the scale, at population size 625 and population divisor 5.0, on the `tt10` problem effective trials of 3.6 corresponds to 3.2 solutions and on the `tt120` problem effective trials of 2.4 corresponds to 2.4 solutions. Evidently, the method not only produces higher effective trials but also more solutions when there are only 10 solutions.

Before moving on to a discussion of why spatial selection finds more solutions when there are fewer available, there is a point to be made about the effective trials. Effective trials were introduced in chapter 2.2.1, however little was said about how the metric varies with the number of solutions available. Recall the intuitive discussion of how the effective trials are derived. When the number of solutions possible is low, then the difference between finding, say 2 or 3 solutions is quite high — with only 3 solutions possible, the difference between finding 2 and 3 solutions is, on average, 3 throws (the total average number of throws is $3/3 + 3/2 + 3/1$). When there are many solutions this factor is much reduced: by the time there are 120 solutions possible the difference between 2 and 3 solutions is just 120/118 throws (by the same argument as was used in chapter 2.2.1, the total average number of throws is $120/120+120/119+120/118$). This is, of course, quite fair. Finding an extra solution with just 3 possible is much better than only one extra solution when 120 are possible (provided we assume that they are evenly spread across the search space).

Spatial selection is not only finding higher effective trials, though. There is a definite increase in the number of solutions as well. An initial hypothesis might be to suggest that this is due to the solution density as with a simple GA. If there is a high solution density then a potential solution will be found quickly and the whole population will converge on to this solution before other solutions have been given a chance to develop. However, this seems to be refuted by a second piece of evidence. Figure 5–5 shows the effective trials per second for spatial selection on the `tt10` and `tt120` problems. For a population divisor of 1.5 the results are almost identical on both problems. But for the other population divisors the efficiency of producing effective trials is much better on the the `tt10` problem than the `tt120` problem. Too much better, in fact, to be accounted for simply by the increased number of effective trials. The method is finding more solutions faster when just 10 solutions are available. A clue comes from the population size 225, population divisor 5.0 case. Comparing figures 5–4 and 5–5, there is only a slight decrease in effective trials, yet there is a very large decrease in the effective trials per second at this point. When spatial selection does not find many solutions it performs more slowly. Why? To answer this we must understand a little about how demes establish themselves.

Collins and Jefferson in [Collins & Jefferson 91] give examples of how areas of single solutions develop in a population. A fit deme spreads out across the grid of chromosomes, encountering other demes as it does so, either forming a border with other fit demes, or simply overrunning the current occupant chromosomes in the

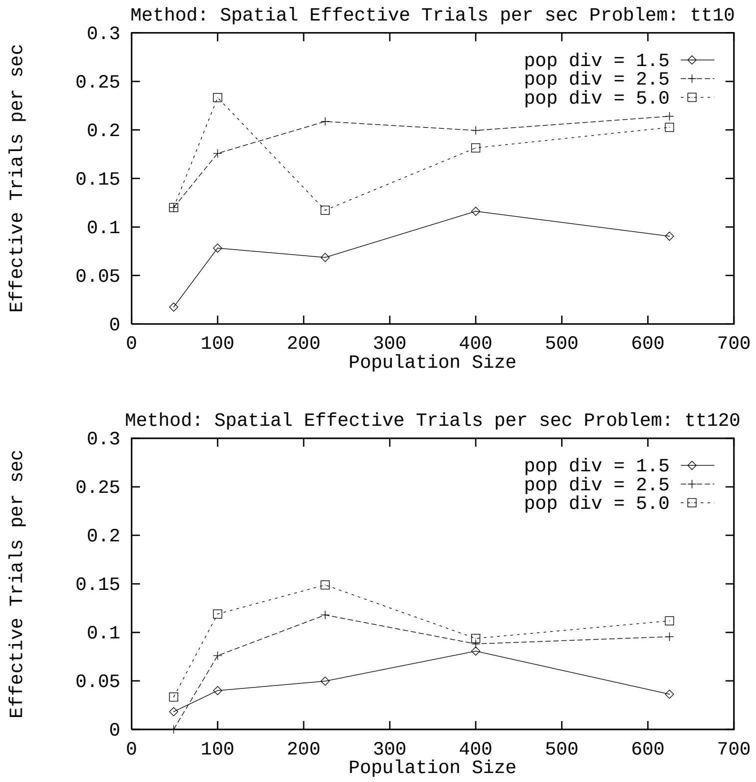


Figure 5–5: Efficiency of Spatial Selection

grid if they are less fit. Figures 5–6 to 5–8 show an example of this effect (produced by using a population of 400 and a random walk of 4 on the `tt10` problem) by showing how the fit chromosomes 'A' and 'B' take over the spatial selection grid.

The solutions expand across the grid until the whole population has converged on two solutions. This spreading of genotypes across the spatial grid provides a theory of how spatial selection is finding solutions in the `tt10` and `tt120` problems. In the `tt10` problems there are only a few solutions to find, a deme can establish itself and the genotypes within it can spread across the grid. In the `tt120` there are many more solutions and many demes may be working towards different solutions. These demes are crammed together on the grid and the genotypes within them are constantly being interbred with the surrounding demes. Figure 5–9 shows the theory of this in operation — areas of similar genotype are shaded. The diagram shows larger areas of similar genotypes being able to concentrate on single solutions in the low solution number case, and hence becoming fitter, while in the high solution number case the genotypes from partially established demes (approach one or other solution) across the grid are interbred, forming children which are unlikely to be fitter the parents. This is a theory which fits the empirical data: in general, spatial selection will find more solutions in a quicker time on the `tt10` problem than on the `tt120` problem.

```
generation = 4500

 A . . . . B B B B B B . B . . . . . A .
 . . . B . B . B B B B . B . . . . . . .
 A A A . . . . . . B B B B . . . . . . .
 . A . . B . . . . . B . . . . . . A . .
 A . . . . . . . . . . . . . . . . . . .
 A . A . . . . . . . B . . . . . . . . .
 . A . . . . . . . . B . . . . . . . . .
 A A . A A . . A . . . . . . . . . . . .
 . A A A A A A . A . . . . A A . . . . .
 A A . A A A . A . . . . . A . . . . . .
 . A A A A A A . . . A . . A A A . . . .
 A A A A A A . A A A . . A A A A A A . .
 . A A A A . . . A A . . . A . A . A . .
 . A . A A . . . . . . . A . A A A A A .
 A . . A A . . . . . . . . . . . A A . .
 . . . . B . B . . B . . . A . . A . . .
 . . . . B B . B B . . . . . . . . . . .
 . . . . B B B B B B . . . B . . . . . .
 . . . . . . B . B B B . B . . B . . . .
 . . . . . . . . B B B . . . . . . . . .


    Number   Fitness        Genotype
A:   82    1.000000000 00003333444411112222
B:   45    1.000000000 00001111444422223333
total no of distinct chromosomes = 113
```

Figure 5–6: First stage of spatial selection grid dominance

5.3.3 Larger population divisors produce more solutions

Returning to figure 5–4, it can be seen that the larger the population divisor the larger the number of solutions produced. The population divisor was introduced in 4.3.2 as a means of providing a similar area coverage for random walks with different population sizes. The higher the population divisor the shorter the the random walk.

So, the shorter the random walk, the more solutions that are found. This is as we would expect: a shorter random walk protects the diversity of the gene pool, since the chances of coming across a fit genotype from a position further away on the grid are reduced. In terms of demes, the demes are less likely to interact with each other and so tend to be preserved. This allows more solutions to be formed within the population as whole.

What is surprising is that the shorter random walk is more efficient as well. In the `tt10` problem shown in figure 5–5 the population divisors 2.5 and 5.0 compete for the best efficiency, while by the 120 solution case the population divisor 5.0 is clearly better. The reasons for the population divisor 5.0 being efficient are the same as for the previous section: separate demes are allowed to develop in areas isolated by distance, so many solutions can be developed concurrently in the same population. With 120 solutions available, each individual solution is easier to find, so smaller demes can work in relative isolation to produce each solution, favouring short random walks. However, in the 10 solution case, it is harder to find each individual solution, hence allowing more inter-deme breeding should allow solutions to be formed faster, favouring longer random walks. This is indeed the case: a population divisor of 2.5 competes well with a population divisor of 5.0 on the `tt10` problem because it can 'see' a greater number of chromosomes at

```
generation = 5500

 A B . . . B B B B B B B B . B B . . A A
 A . B B B B B B B B B B B B . . . B A A
 A A A . . B B B B B B B B B A . . . A .
 A A . B B B B B B B B B B B A . . A . A
 A A B . . . . A B B B B A B . A B . A A
 A A A . . . . . . . B B B A A . A A A A
 A A A A A . A A B . B . . A A A A . A A
 A A A A A A A A A B B B . A A A A A A A
 A A A A A A A A A . . . B A A A A A A A
 A A A A A A A A A A A A . A A A A A A A
 A A A A A A A A A A A . A A A A A A . .
 A A A A A A A A A A A . A A A A A A A .
 . A A A A . A A A A A . A A A A A A A A
 A A A A A A A A A A . . A A A A A A A A
 A A A A A A A A A A A . A A B A A A . A
 A A A A B A B B B B . . . A . . A A . A
 A . . B B B . B B B B . B . . . B A A A
 . . A B B B B B B B . B B B B B A A A .
 . . B B B B B B B B B B B B . B . . A .
 . . B . B B . B B B B B B B . B . . . .


    Number   Fitness        Genotype
A:  201     1.000000000 00003333444411112222
B:  111     1.000000000 00001111444422223333
total no of distinct chromosomes = 45
```

Figure 5–7: Second stage of spatial selection grid dominance

each generation, however the population divisor 2.5 is less efficient than 5.0 for the `tt120` problem for all population sizes. However, the population divisor of 1.5 is producing randoms walks which are too long. The isolation by distance is reduced too far, a continuum is reached and the result is disastrous for spatial selection. Due to the deme break up the population divisor of 1.5 converges too fast, and usually before finding a solution.

5.3.4 Summary

Spatial selection suffers from a tendency for one particular genotype to overrun the selection grid quickly, stifling the diversity of the gene pool. Shorter random walks help prevent this to some extent. Short random walks are in general more efficient at maintaining diversity of the gene pool and hence tend to find more solutions. Since many solutions can be developed simultaneously, using a shorter random walk also keeps the efficiency of finding solutions high.

5.4 Islands

5.4.1 Islands produce similar results to simple GAs

Figure 5–10 shows the results for islands for the `tt120` problem. The number of solutions is not much higher than for the control method. In fact, for all numbers of subpopulations the results do not vary much. It is possible that the migration rate was chosen to be too high, since only a few solutions are produced by even

```
generation = 6500

 A B A B . B B B B B B B B B B B B B A A
 A B B B B B B B B B B B B B A . B B A A
 A A A B B B B B B B B B B B A . . B A A
 A A B B B B B B B B B B B B A A A A A A
 A A B . . B A A B B B B A B . A B A A A
 A A A . A A . B . B B B B A A A A A A A
 A A A A A A A A B A B A B A A A A . A A
 A A A A A A A A A B B B . A A A A A A A
 A A A A A A A A A A . A B A A A A A A A
 A A A A A A A A A A A A . A A A A A A A
 A A A A A A A A A A A A A A A A A A . A
 A A A A A A A A A A A A A A A A A A A .
 A A A A A A A A A A A . A A A A A A A A
 A A A A A A A A A A A A A A A A A A A A
 A A A A A A A A A A A A A A B A A A A A
 A A A A B A B B B B . . A A . B A A . A
 A . B B B B . B B B B . B . . A B A A A
 . B A B B B B B B B B B B B B B A A A .
 A A B B B B B B B B B B B B B B B B A A
 B . B . B B B B B B B B B B B B A . A B


    Number   Fitness        Genotype
A:  233    1.000000000 00003333444411112222
B:  136    1.000000000 00001111444422223333
total no of distinct chromosomes = 23
```

Figure 5–8: Third stage of spatial selection grid dominance

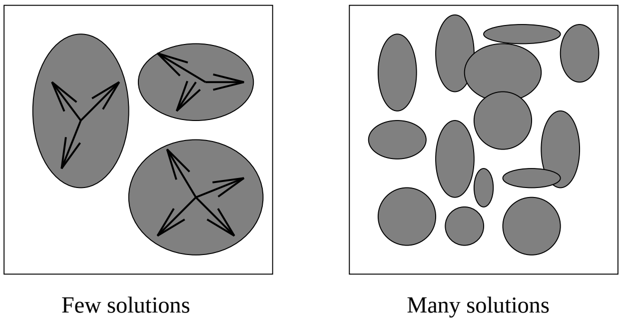


Figure 5–9: A theory for spatial selection deme formation

5 subpopulations. At a population of 600, if 5 subpopulations were fully isolated then these would correspond to 5 individual populations of 120 chromosomes. Since an individual population of 120 chromosomes in a simple GA consistently produces one solution, we would expect 5 populations to produce an effective number of trials of 5. This is clearly not happening; hence, chromosomes entering the population through migration must be influencing the population already there. We would actually expect this even at low migration rates: if one subpopulation converges, then any chromosome migrating from this population will be the solution chromosome it has converged on. If the population it migrates to has not got a chromosome which can compete with it in terms of fitness, this chromosome will quickly take over this population in addition since the rank selection operator will prefer to choose this chromosome for breeding. This is not helped by choosing a chromosome to be migrated into a population by using the standard selection operator: it might be better to choose the chromosome for migration at random instead.

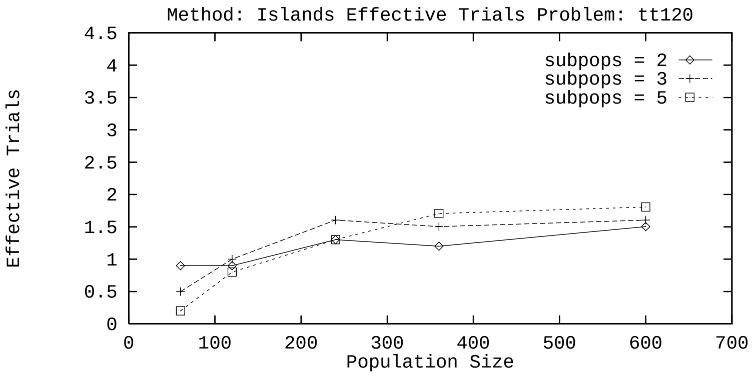


Figure 5–10: Effective Trials for Islands

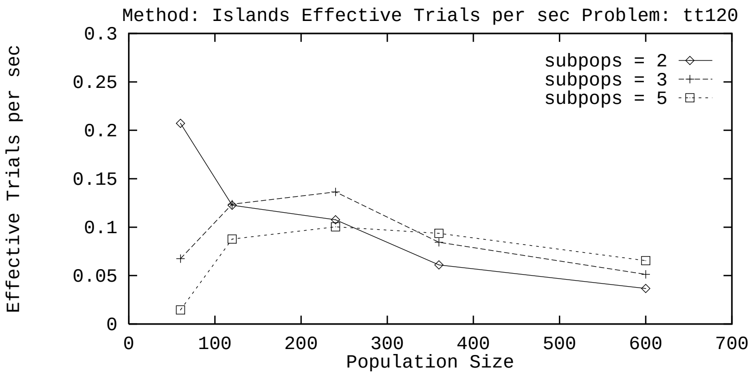


Figure 5–11: Efficiency of Islands

The efficiency of the methods is again close to a simple GA. Figure 5–11 shows the the effective number of trials per second for the `tt120` problem. At low populations the methods suffer from not having enough of a gene pool for each population. The 2 subpopulation case manages to perform like a poor version of the simple GA, while the 3 and 5 subpopulation cases begin with a smaller gene pool and therefore find it hard to produce solutions at lower population sizes. As the population size increases they benefit slightly from being able to converge on separate solutions in different in separate subpopulations; however, the effect is not nearly as pronounced as it is with spatial selection.

5.5 Crowding

5.5.1 Large Crowding Factors are effective but inefficient

All crowding factor experiments showed the same trends. Figures 5–12 and 5–13 show the results for the `tt10` problem. A crowding factor of 5 produced higher effective trials, while the other crowding factors performed similarly to the control method. Unfortunately, in all cases the efficiency was reduced. The higher the crowding factor, the more inefficient the method became. The crowding factor 5 case shows that some of the gene pool diversity is maintained as would be expected. The crowding can replace a chromosome that it sufficiently similar to itself; however, for each new chromosome a Hamming distance test must be performed, and this test reduces the overall efficiency.

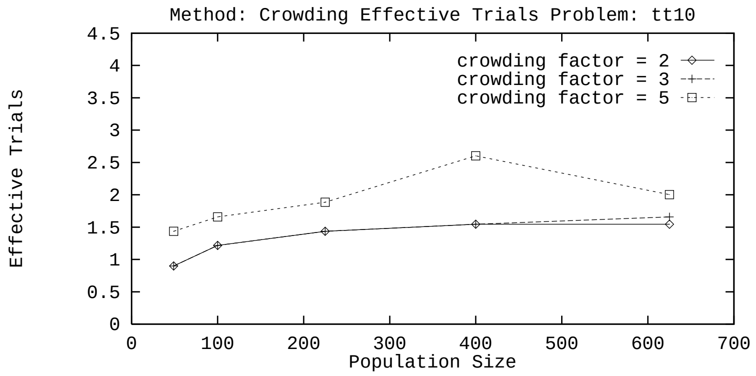


Figure 5–12: Effective Trials for Crowding

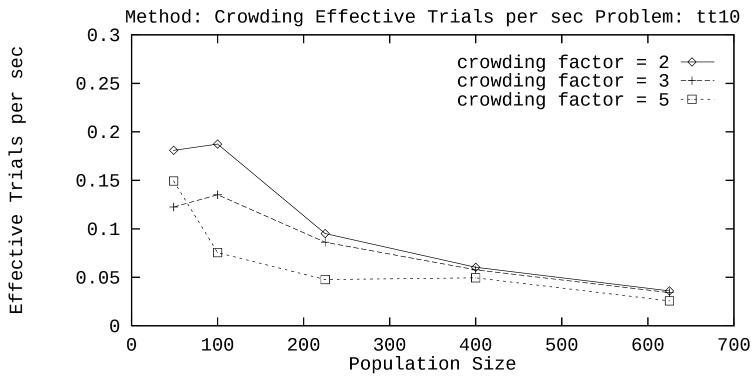


Figure 5–13: Efficiency of Crowding

5.6 Spatial Crowding

All spatial crowding experiments were conducted with the best performing spatial selection method: population divisor 5.0. It could be argued that perhaps 2.5 could also have been tried; however, this was not possible due to limited time available.

5.6.1 Spatial Crowding similar to Spatial Selection

Figures 5–14 and 5–15 show the results for the `tt10` problem. Here the crowding factors of 3 and 5 show an improvement to the small population case in spatial selection. With a population of 49 the effect is similar to that already described for spatial selection: the diversity is maintained long enough to allow solutions to be found. However, at higher populations the effects of crowding are to reduce the spatial selection method's capabilities slightly. An explanation for this might be that while crowding may enhance diversity within a deme, it has little effect further from the centre of a deme. This would cause a general destabilisation of the demes. With demes finding it harder to find a solution, eventually fewer solutions will be found. The effect is the same at lower populations, but the increase in diversity still allows the ability to find more than one solution. In general, the graph will be flattened, which is what is observed in figure 5–14. The efficiency is improved for lower population sizes, and the cost of implementing crowding is similar to that of standard crowding. This has the effect that the overall efficiency stays similar to spatial selection.

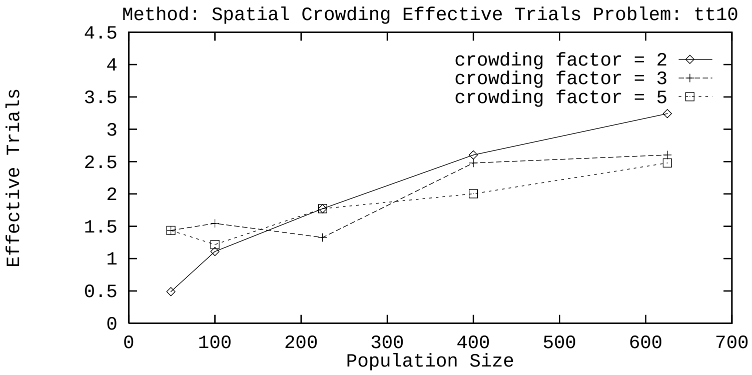


Figure 5–14: Effective Trials for Spatial Crowding

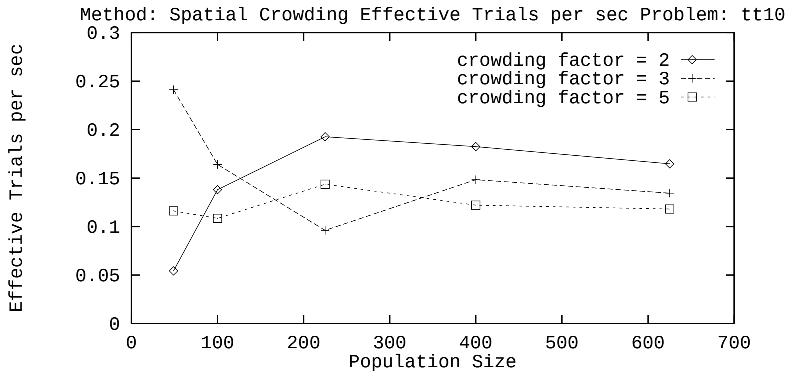


Figure 5–15: Efficiency of Spatial Crowding

5.7 Sharing

5.7.1 Careful selection of σ_{share} produces many solutions

Figure 5–16 shows the results of sharing on the `tt10` and `tt60` problem. Surprisingly, the niche separation distance of 0.6 calculated in section 4.3.2 performs worse than $\sigma_{share} = 0.8$ in both cases.

Firstly, though, why does $\sigma_{share} = 0.4$ perform so badly? Looking at the graphs for $\sigma_{share} = 0.4$, on the 10 solution problem the method manages to find solutions at low populations but not high populations. A similar explanation to that for spatial crowding is possible. Because the niche is quite narrow, any group of chromosomes converging on the same solution quickly lose any fitness benefit because they all have to share the benefits of being *near* that solution. The population becomes very diverse, but no one genotype can gather any momentum of numbers. What is worse in the sharing model than in the spatial crowding model is that gathering numbers simultaneously reduces the fitness of that genotype. Because the niche is narrow, other genotypes are unaffected and become more likely to be selected. In small populations this can still be a benefit: the diversity of the population is increased, but at the same time the numbers of chromosomes needed in a 'group' to give that group a better chance of selection is fewer. However, with larger populations, because the number of chromosomes needed to give a good chance of selection is higher, their fitness will be too far reduced by the time they would have achieved this chance. This can be demonstrated by an example. Say the fittest genotype has fitness 0.2 and there are 2 chromosomes in a population of 49 with this genotype. No other chromosomes are in this niche. Therefore these chromosomes have a shared fitness of 0.1. To give the same percentage of chromosomes with this genotype with a population of 400 chromosomes there would have to be about 16 chromosomes with this genotype, but by this time the shared fitness is reduced to 0.0125.

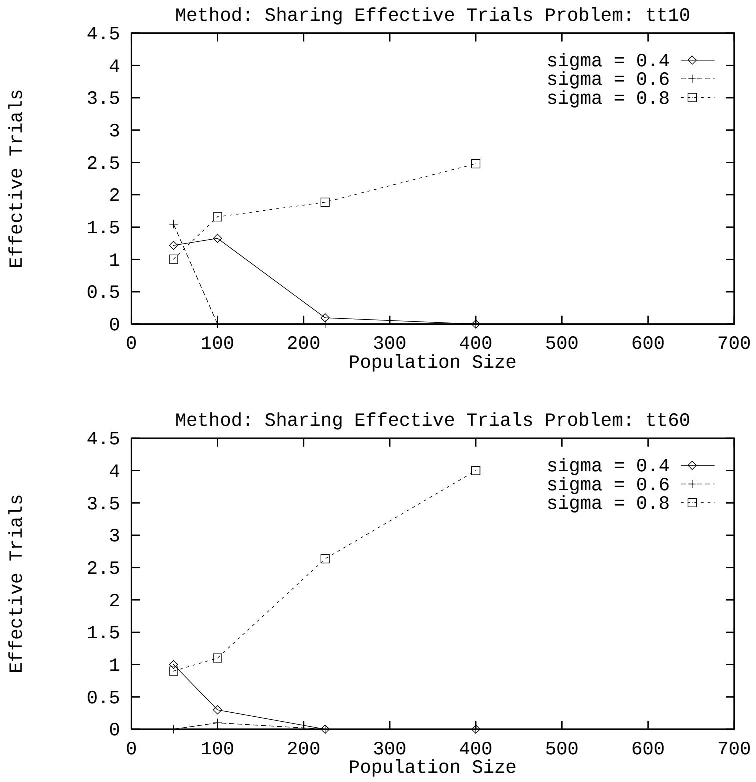


Figure 5–16: Effective Trials for Sharing

The $\sigma_{share} = 0.6$ case is very similar to the $\sigma_{share} = 0.4$ case. The wider niches ensure that less chromosomes are left without some sort of sharing factor, giving more of a chance to any relatively fit groups of chromosomes forming.

By $\sigma_{share} = 0.8$ this situation has clearly changed. Now the fitness is increasing with population size. What has happened? We have clearly passed some limit in niche size, so that now the chromosomes in a niche are not hindered by excessive penalties for their numbers. At least, chromosomes are given enough of a chance to form a solution. As the population size is increased more diversity is allowed throughout the population so there is more 'room' in the population to allow different niches to develop; each niche is less hindered since any chromosome will tend to have a sharing factor reducing its fitness, the more chromosomes in the population the more chromosomes will be overlapping. The reason that sharing works well with $\sigma_{share} = 0.8$ is that no one of these niches can become too strong.

Is there some reason why the niche separation distance 0.8 lies in a different regime from $\sigma_{share} = 0.6$? The problem with $\sigma_{share} = 0.6$ is that it separates the genotype space into almost exactly 30 niches. This number is just too many to sustain the effort of finding any solution. In retrospect, being more conservative with the solutions we are attempting to find would have been more sensible. 0.8 corresponds to just over 1 niche. A point worth noting is that a normalised Hamming distance of 0.8 corresponds to the mean separation distance of the solutions in the `tt120` problem.

5.7.2 Sharing is very inefficient

Figure 5–17 shows a graph of the effective trials per second for the `tt10` as an example of the poor efficiency of the method. The method never achieves very good efficiency since there is a huge number of calculation of Hamming distances to be made. At every generation calculation of the Hamming distance from the child chromosome to every chromosome in the population is made. Although, with a better implementation than mine the number of calculations needed may be decreased, the problem remains that the method will always have to make approximately this many calculations of Hamming distance per generation.

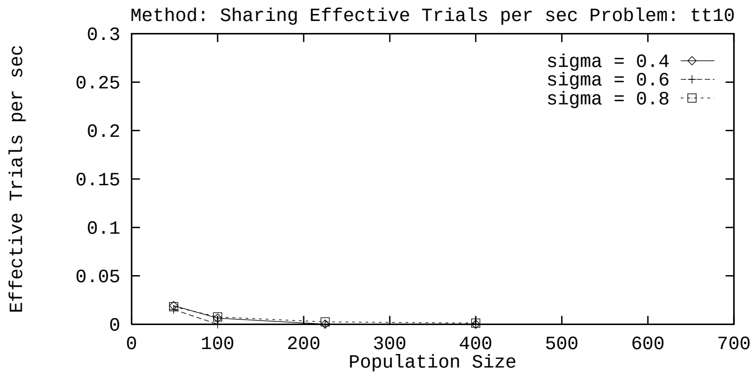


Figure 5–17: Efficiency of Sharing

5.7.3 Summary

In conclusion, sharing can be used to find more solutions if used with a well chosen value of σ_{share}. As can be seen from the results, choosing a fairly conservative value is more helpful than hoping to be able to find too many solutions. However, the method in this form is too inefficient to warrant close investigation.

5.8 Spatial Sharing

As with spatial crowding, all spatial sharing experiments were conducted with population divisor 5.0.

5.8.1 Spatial Sharing stops demes becoming established

Spatial sharing performs very badly. Figure 5–18 shows the best effective trials achieved. These were obtained on the `tt10` problem. The graph resembles that for the $\sigma_{share} = 0.4$ or 0.6 cases for sharing, only worse. The problem here is the method of spatial sharing that I proposed. As discussed in 4.3.2, the spatial sharing zone is the random walk length in each direction. The trouble with using this area though is that any one deme cannot establish itself. Any second instance of a chromosome of a certain genotype formed near another will immediately suffer a local reduction in fitness. Since, in spatial selection, this chromosome will almost always be near the other, no slightly fitter genotype can establish itself in the population: selection will be limited to a finding virtually random fitness parents. This is not helped by the population divisor chosen. With a short random walk, the local area is much smaller so, for example, with a population of 49 the random walk is 1 and the area of the spatial sharing zone is just 9. When choosing parents, 2 chromosomes with same genotype will have to compete against 7 others. With 9 chromosomes in the sharing zone, it is unlikely that these other 7 chromosomes will be in the same niche, or even if they are (when $\sigma_{share} = 0.8$) the sharing factor will be minimal. Hence, the 2 chromosomes fare much worse than if they were sharing over a zone equivalent to the whole population size.

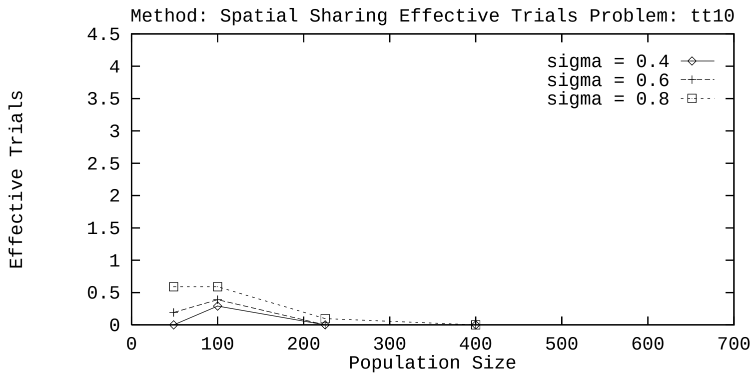


Figure 5–18: Effective Trials for Spatial Sharing

The methods hardly ever produce results, and so the efficiency was negligible. The results for the effective trials per second can be found in appendix A.

5.9 Summary of results

The preceding sections have discussed the issues for each method alone. This section will tie together the methods and compare the results obtained. Note that the parameter values used to demonstrate the effective trials are not always the same as those used when comparing the efficiency.

5.9.1 Comparison of effective trials generated

Figures 5–19 and 5–20 at the end of this chapter compare the results for all the methods on problems `tt10` and `tt120` respectively. In each case, the parameter that produced the highest effective trials is used.

As can be seen, all the methods actually produced very similar effective trials, both on the 10 solution problem and the 120 solution problem. Spatial selection, crowding, spatial crowding and sharing all performed about the same with the 10 solution problem, while sharing was by far the best method for the 120 solution problem.

The operators, with the exception of sharing, all really fail to produce results with the 120 solution problem. 2 effective trials is hardly much better than a simple GA.

5.9.2 Comparison of efficiency of effective trials generation

Figures 5–21 and 5–22 show the effective trials per second for the methods on the `tt10` problem and the `tt120` problem. Here the parameter that produced the highest efficiency is shown.

It is noticeable that all methods tend to work better at low population sizes and, not only this, the peak efficiency is displayed when very few effective trials are being produced. Typically, the methods are only generating 1 to 1.5 effective trials when they are at their most efficient. This actually means that creating multiple distinct solutions is more about creating any solutions fast than producing more than one solution per run.

In efficiency terms there are two regimes. Spatial selection and spatial crowding work well on the `tt10` problem, while the control method and islands work much better with the `tt120` problem.

Spatial selection methods have the edge on the lower solution problems because of the way they exploit the formation of demes. However, by the 120 solution problem, two factors have weighed against spatial selection methods. The first is that spatial selection finds fewer effective trials with higher numbers of solutions available. The second is that the simple GA methods have started to find solutions at lower population values.

In conclusion, all methods do not improve considerably on a simple GA in terms of number of solutions found. However, on problems with low numbers of solutions an increase in efficiency of conducting effective trials can be achieved by using spatial selection methods.

A final word of caution is necessary though: I have not offered a quantitative analysis of the results. As can be seen, there are very few points on the graphs. Also, the lines between the points are erratic. In fact, there are really too few points to be joined in this way: the lines just help to distinguish the methods. The impressions here are just that: impressions and not conclusive evidence.

5.9.3 Setting the stage for a new method

Finally, when referring to spatial selection methods, I have often talked of demes being able to establish themselves, or preventing demes from being overrun by various means. That spatial selection methods are the only methods that have been seen to improve significantly on simple GAs warrants some further investigation into them. The theory that demes require help to establish themselves leads naturally to the idea that some operator should be introduced that protects deme formation. Such an operator, tribes, is introduced in the next chapter.

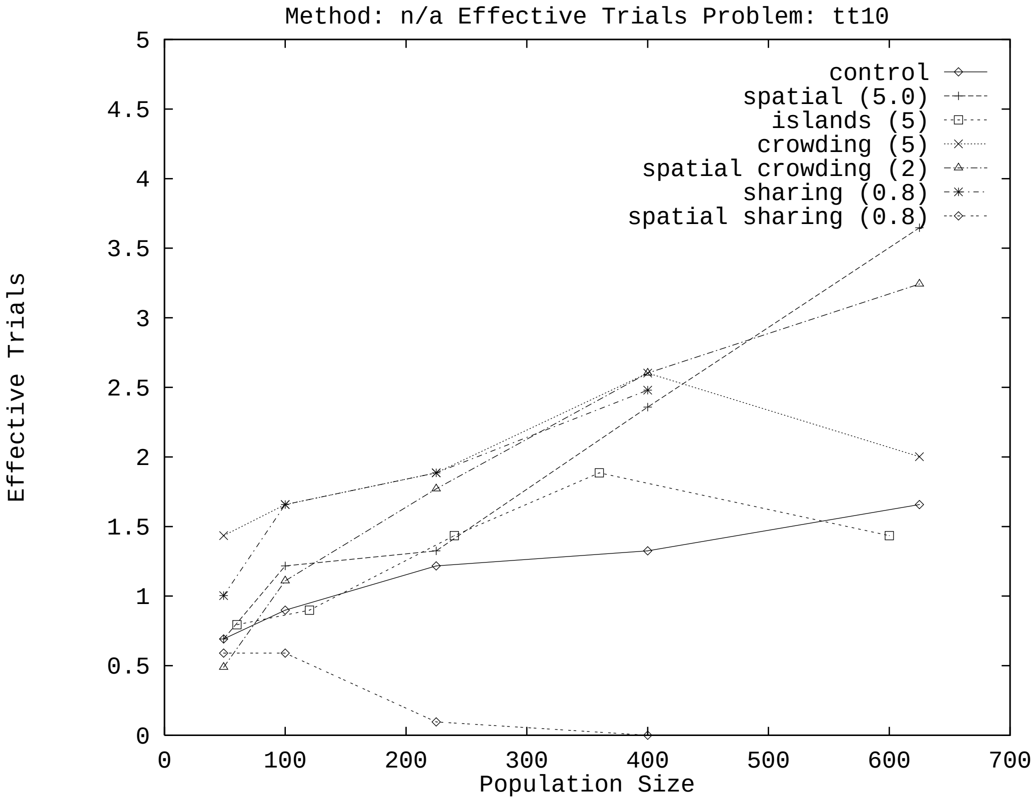


Figure 5–19: Effective trials for methods on `tt10` problem

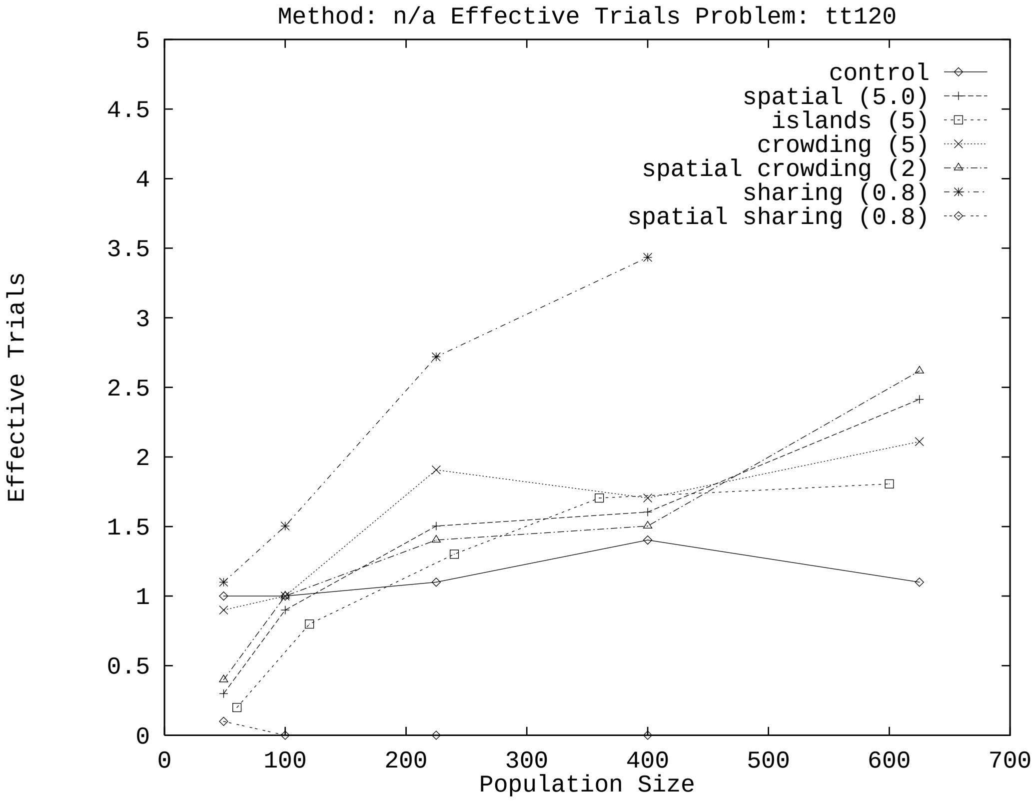


Figure 5–20: Effective trials for methods on `tt120` problem

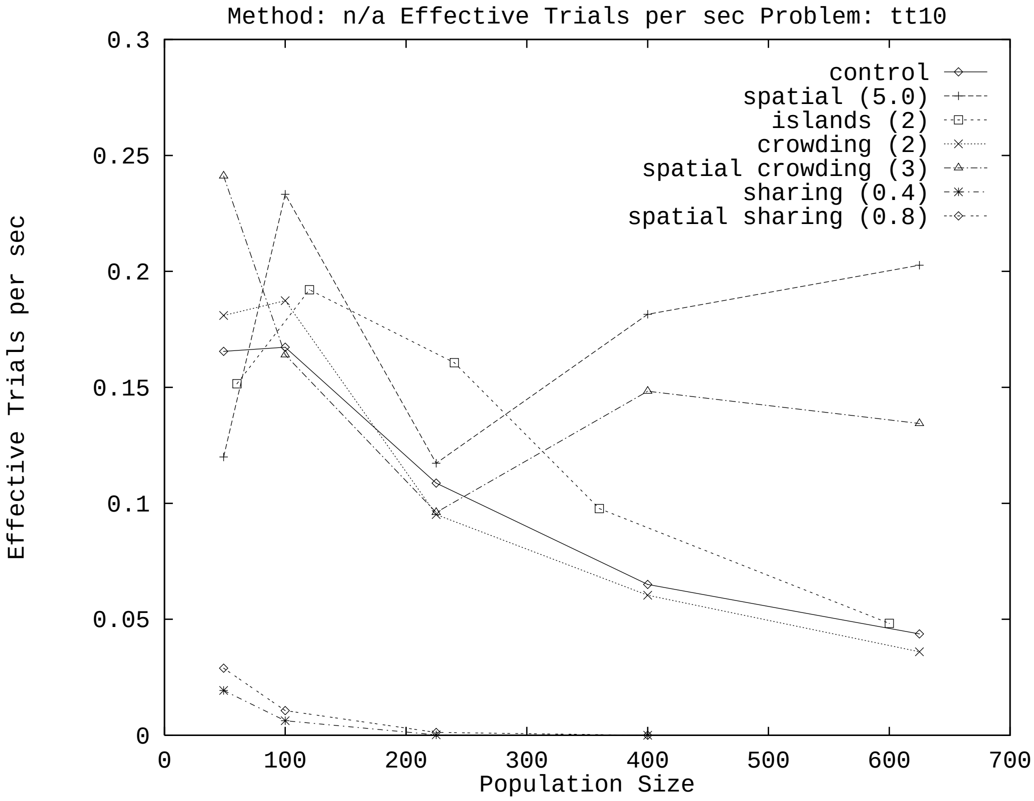


Figure 5–21: Efficiency of methods on `tt10` problem

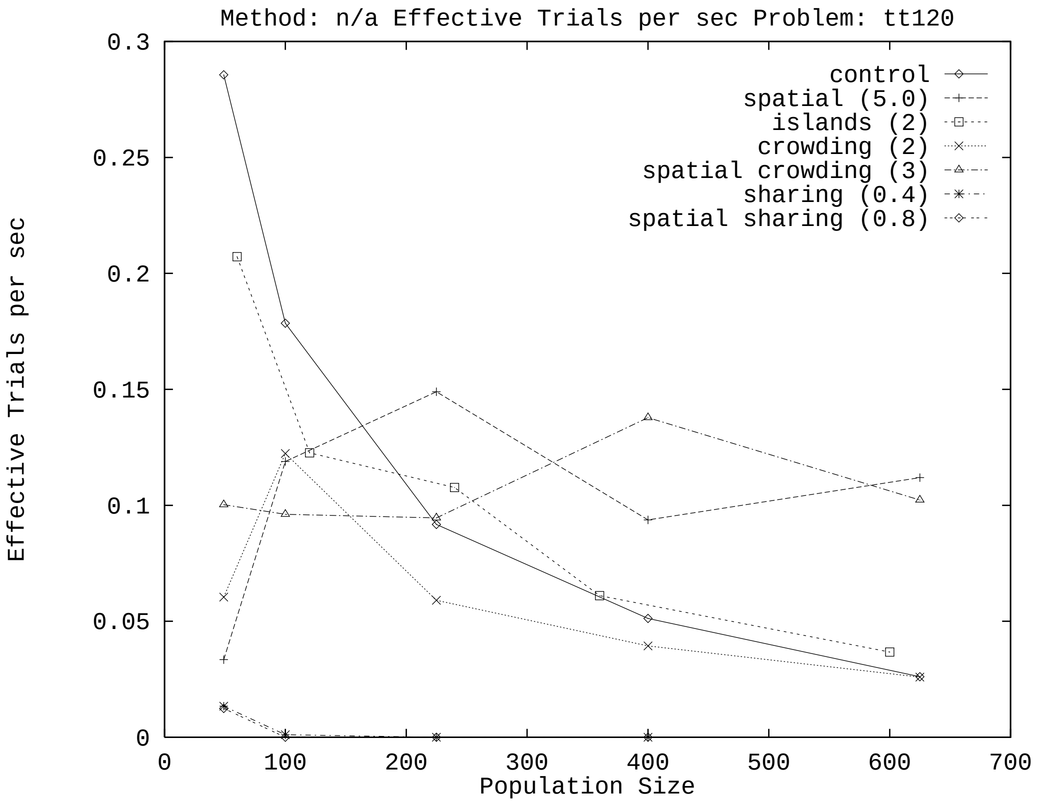


Figure 5–22: Efficiency of methods on `tt120` problem

Chapter 6

A new method: Tribes

6.1 Introduction

As was explained in chapter 5.3 spatial selection methods suffer from successful genotypes rapidly dominating the whole spatial selection grid, destroying nascent demes as they advance. A method is required to allow demes to form without danger of being destroyed in this manner.

The operator also needs to be efficient. Sharing especially, and crowding to a lesser extent, both involve a lot of calculation which reduces their efficiency. It would seem sensible to posit a method which requires as few calculations as possible to increase or sustain the diversity of a population.

6.2 The tribes method

An operator which should stop a genotype alien to the local environment taking the place of the resident genotype is to allow replacement of a chromosome on a spatial selection grid only if that genotype is sufficiently like the current occupant of the position. In this way, only members of the same 'tribe' can be inserted into the population at any location.

Tribes uses the Hamming distance to determine whether the a child chromosome is like the current occupant at the chosen location on the grid. If the Hamming distance from the child to the current occupant in the grid is over a certain distance then the child is rejected and discarded. This distance will be called the cutoff distance, d_c, and it will be measured using the normalised Hamming distance, d_N, introduced in chapter 4.3.2.

In order to keep the efficiency high, the child is not even evaluated if it is rejected. Hence, the generation consists of selecting parents, deciding whether the child has a chance of insertion, if it does then evaluating it and inserting if fitter. The revised genetic algorithm can be written in pseudo code as follows:

```
LOOP
    SELECT location
    SELECT parents FROM NEAR location
```

```
    CROSSOVER parents TO FORM child
    MUTATE child
    IF d_N(γ_child, γ_location chromosome) > d_c THEN
        REJECT child
    ELSE
        EVALUATE child TO GIVE child fitness
        IF child fitness > location chromosome fitness THEN
            INSERT child IN PLACE OF location chromosome
        END IF
    END IF
END LOOP
```

6.3 Testing of tribes

Tribes was tried on top of the basic spatial selection GA with a population divisor of 5.0. A population divisor of 5.0 was chosen because this showed the best overall performance with spatial selection when used on the test set problems (see chapter 5.3 for details).

Choosing a value of d_c might be considered similar to choosing a value for the niche separation distance used in sharing. The value of σ_{share} was chosen because it reflected the distance between different niches in the search-space. With d_c, we want to perform the same kind of operation: if the child looks as though it comes from a different tribe, it is equivalent to saying that it is converging to a different solution from the deme in which it is being placed. Something which is converging to another solution, in the theory of sharing, is in a separate niche. Therefore, it would seem sensible to choose similar values for d_c as we chose for σ_{share}. Values for d_c of 0.4, 0.6 and 0.8 were used — the same as the values chosen for σ_{share}.

Tribes was tested on the same problems as were used with the other methods, introduced in chapter 4. Full details of the test plan method appear in appendix A.

6.4 Discussion of results

6.4.1 Lower cutoff distances raise t_{eff} but are less efficient

Figure 6–1 shows the effective trials and efficiency of tribes for the `tt60` problem case. As can be seen from the figure, the method performs very well at producing effective trials and lower values of d_c generate consistently higher values for t_{eff} for all population sizes. This is to be expected: the lower the value for d_c, the less easy it is for a foreign genotype to be placed within a deme and therefore the

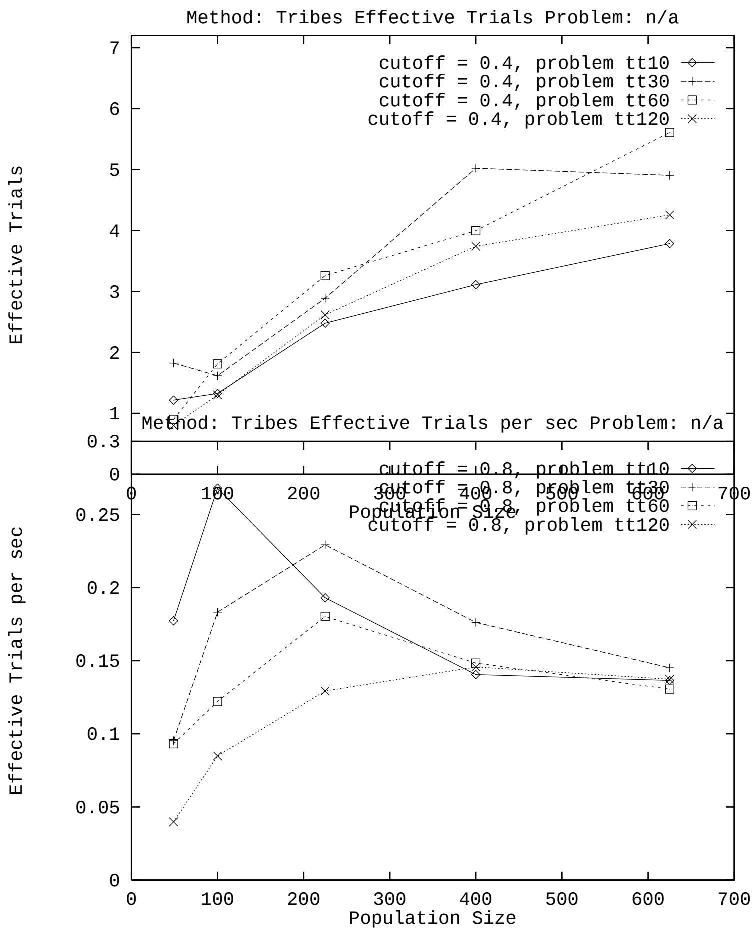


Figure 6–1: Effective trials and efficiency on the `tt60` problem

population will remain more diverse. The demes become established more easily, allowing separate demes to converge on different solutions.

Unfortunately, although t_{eff} is raised with lower values of d_c, the efficiency is also reduced. It takes more time for the population to converge on the distinct solutions. As has been mentioned when discussing criteria for judging success in chapter 2.2, with the metrics used there is no easy way of telling whether the first solutions are found equally quickly with both a low cutoff distance and a high cutoff distance, or whether a low cutoff distance finds solutions at a constantly slower rate than a high cutoff distance. The latter is probably the case: convergence is generally slowed down by selecting low values of d_c. Because children have to reflect the genotype of the deme, it is more difficult for any new, fitter genotype to be inserted into the population the lower d_c is set.

6.4.2 Tribes generates more effective trials in the midrange of solution numbers

An interesting feature of tribes is that the peak values of t_{eff} are found when there are an intermediate amount of solutions available (intermediate, of course, in the range chosen for the project). Figure 6–2 shows how many effective trials were found with $d_c = 0.4$ for the `tt10`, `tt30`, `tt60` and `tt120` problems.

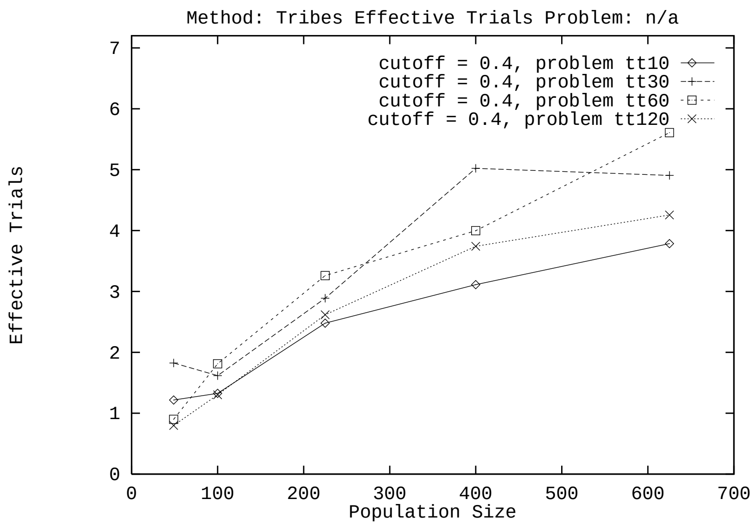


Figure 6–2: Effective trials for tribes with $d_c = 0.4$

There is not a huge difference between the effective trials produced, but there are definitely higher numbers of effective trials on the `tt30` and `tt60` problems than the `tt10` and `tt120` problems. Why should this be? Referring to the discussion of spatial selection results (chapter 5.3), notice that the way genotypes expand across the spatial selection grid means that more solutions are found in the 10 solution case than the 120 solution case. However, by introducing tribes we have restricted the expansion of dominant genotypes across the spatial selection grid. A deme only accepts a new genotype if it is within its 'tribe'. This means that the amount of inter-deme bred children that are inserted has been reduced. When there are only 10 solutions available this prevents tribes working at its best: each deme is more isolated than it was before and their spread across the spatial grid is stalled; it would make more sense for each deme to cover more area and therefore have access to a larger gene pool, and since there are fewer solutions available this extended deme size would not hinder the amount of solutions being found. With more solutions available it makes sense to reduce the deme size, allowing each deme to work towards one of more separate goals. So with more solutions, the stabilised spatial selection grid works in favour of tribes. However, by the 120 solution case, there are just too many solutions and the amount of deme interaction cannot be reduced enough. Now, for the same reasons as spatial selection finds less solutions on the `tt120` problem, tribes also finds less solutions.

6.4.3 Efficiency decreases with the number of solutions available

Although there are slightly higher numbers of effective trials found in the `tt30` and `tt60` problems than in the `tt10` problem, the efficiency of tribes seems to work in a very similar way to standard spatial selection. Figure 6–3 shows the efficiency of $d_c = 0.8$ on all the problems.

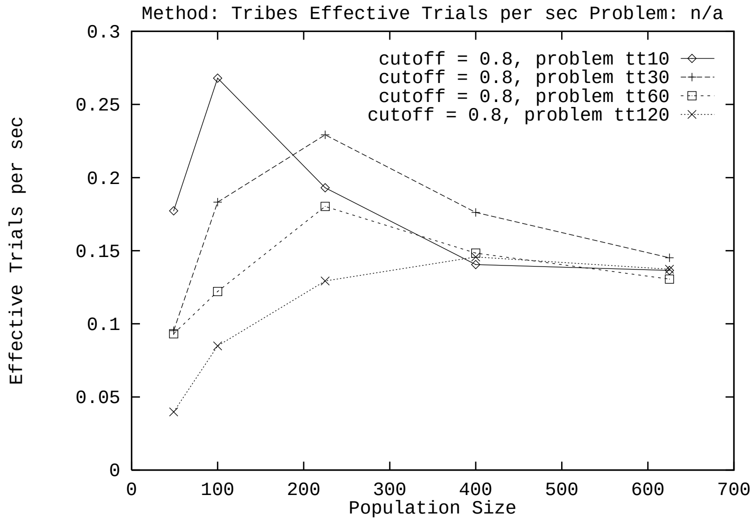


Figure 6–3: Efficiency of tribes with $d_c = 0.8$

As stated, the reason for the general decrease in efficiency as the number of solutions increases probably works with a similar mechanism as that with spatial selection. As the number of solutions is increased the method finds solutions more slowly. This happens because larger sections of the spatial grid work towards the same solution when lower numbers of solutions are available. With higher numbers of solutions available the efficiency is slowed since there are many competing fit genotypes on the spatial selection grid, all forming separate demes. These competing demes find solutions more slowly as they inter-breed when partially converged on one solution with other demes which have started to converge on another solution. The resultant is likely to be either a mutant and be rejected through the tribes operator, or less fit and again rejected.

6.4.4 Comparison with other methods

How tribes compares on effective trials

Figures 6–4 and 6–5 at the end of this chapter show t_{eff} for the `tt10` and `tt120` problems for the control method and some of the better methods at finding effective trials. As can be seen tribes is extremely good at finding effective trials and the method is performing almost as well as sharing, and better than any of the others even for these problems, where it is not at its peak.

It should be noted that although sharing produces effective trials marginally better than tribes, tribes (cutoff distance 0.4) is over 5 times as efficient as sharing (niche separation distance 0.8).

How tribes compares on efficiency

Figure 6–6 at the end of the chapter shows the effective trials per second for a selection of the more efficient methods. Note that despite the fact that tribes produces large numbers of solutions, it is still more efficient with higher values of d_c; therefore when tribes is most efficient it is producing a number of solutions more comparable with the number being generated by spatial selection at its most efficient.

Tribes produces solutions more efficiently than any of the other methods at when there are 10 possible solutions as can be seen from the figure. However, by the 120 solutions case, the efficiency has dropped; the simple GA methods still work more efficiently when there are more solutions to be found.

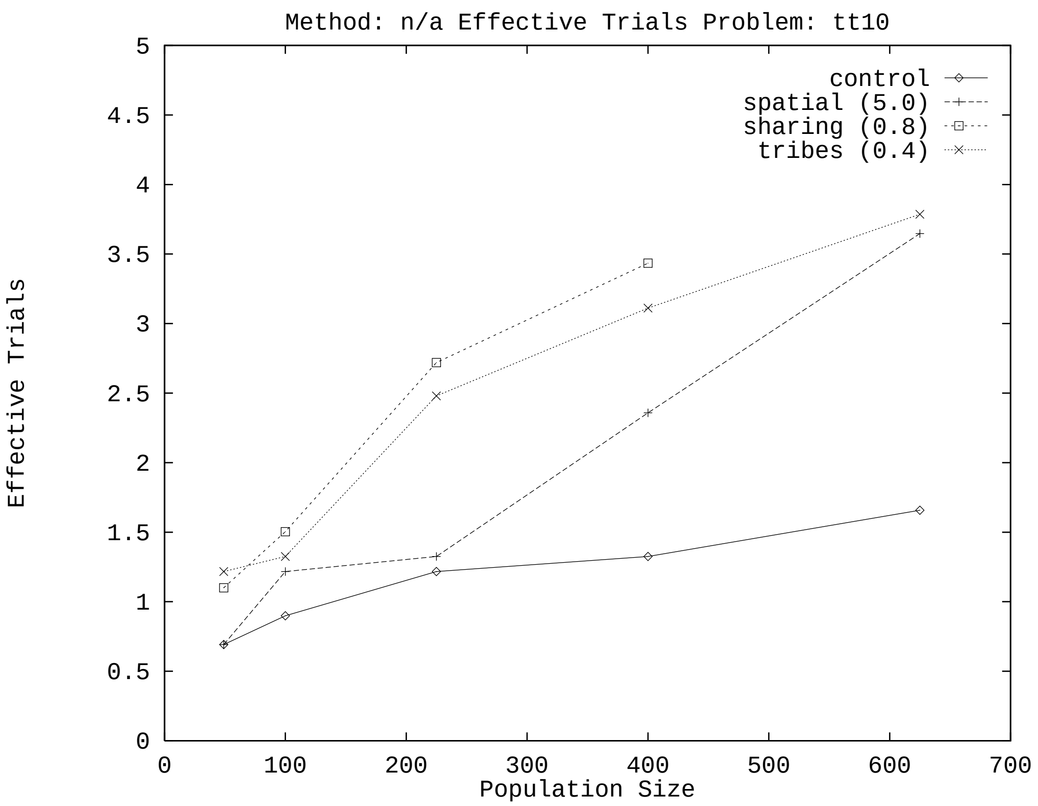


Figure 6–4: Comparing tribes effective trials on the `tt10` problem

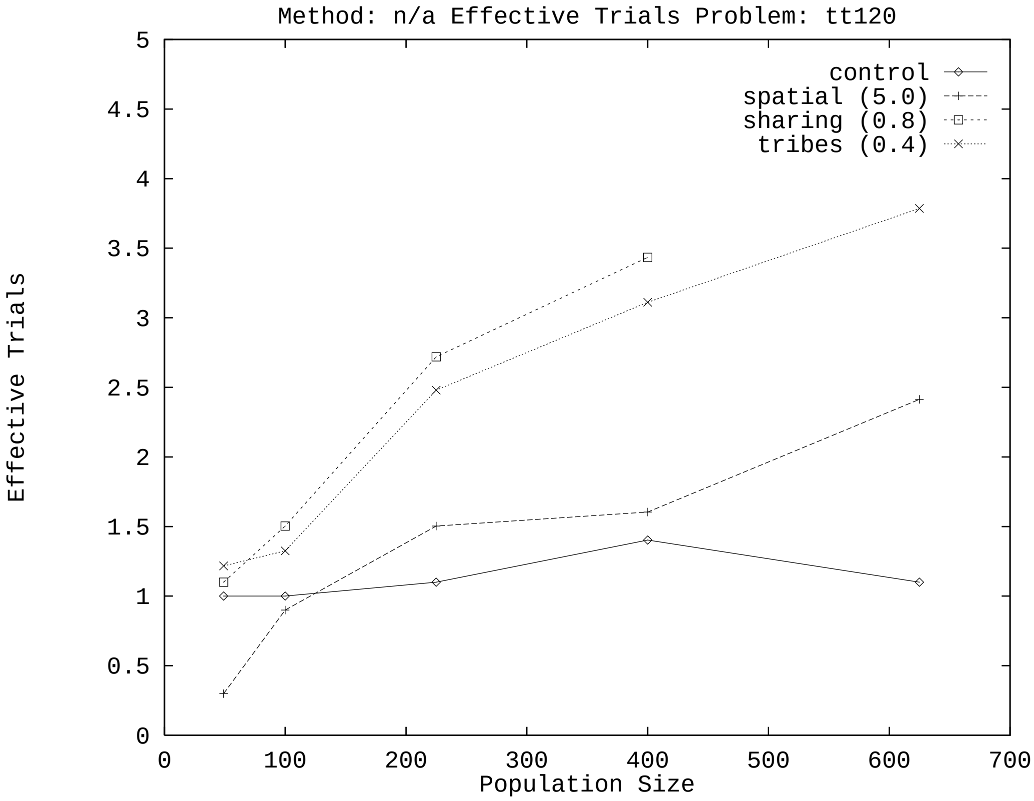


Figure 6–5: Comparing tribes effective trials on the `tt120` problem

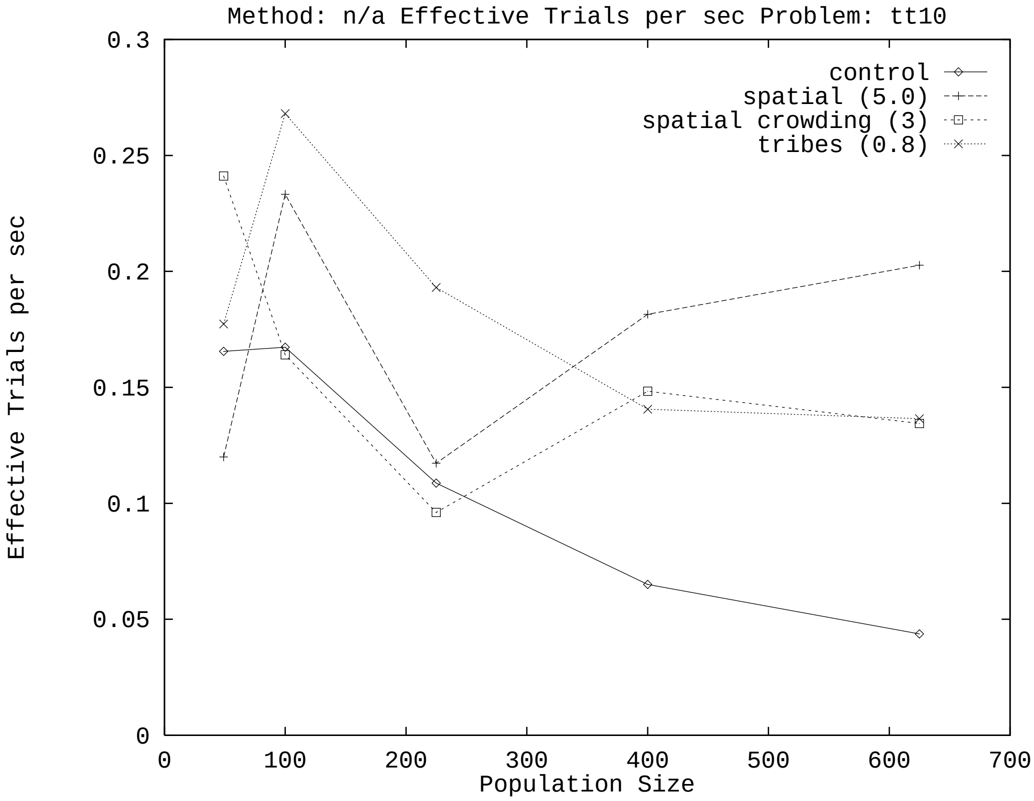


Figure 6–6: Comparing tribes efficiency on the `tt10` problem

Chapter 7

Conclusion

7.1 Finding multiple distinct solutions

This project set out to compare the ability of adaptations to GAs to find multiple distinct solutions. It looked at eight methods in all: spatial selection proposed by [Collins & Jefferson 91], islands [Tanese 89], crowding [De Jong 75], spatial crowding, sharing [Deb & Goldberg 89,Goldberg & Richardson 87], spatial sharing and tribes.

The methods were tested on a set of four simple timetabling problems [Corne 94] using four metrics to judge their success. The timetable problems were devised so that they had 10, 30, 60 and 120 solutions. The criteria used to judge their success were allele count diversity [Corne 94], allele count diversity, effective trials, effective trials per second.

The project found that producing multiple distinct solutions on simple timetabling problems cannot be achieved easily with genetic algorithms, whatever adaptations have been made to them. When distinct solutions are found in a single run, there tend to be few of them and in most cases finding them also results in a decrease in efficiency of the method in question; so much so that more distinct solutions would be found faster by running a simple GA many times or the same method with parameters and population sizes which produce fewer solutions per run.

Simple GAs are always more efficient at finding distinct solutions when there are many solutions available. However, spatial selection works faster when there are only a few solutions on the problems. There has not been enough experimentation performed here to be able to tell whether spatial selection is working more efficiently when the number of solutions in the search space is low or when the density of solutions in the search space is low.

One method that finds many distinct solutions and remains efficient is tribes. Even so, tribes still tends to work more efficiently with smaller populations, with parameters set so it finds fewer solutions.

A separate issue that has been brought to light is the metrics used to judge the success of a single GA run. Valuable lessons were learned about how to time a run: the important point for future research is to find out the peak *efficiency* of a method.

Finally, not enough work has been conducted to find out how general these findings are. The experimentation covered a very large number of combinations of methods and so missed much fine detail. What has been presented is a broad overview of the area of finding multiple distinct solutions in a restricted domain — that of simple timetabling problems.

7.2 Further work

There are several areas of further research that have presented themselves. They are listed below.

Tribes

Tribes is perfectly capable of being implemented with a simple GA rather than with spatial selection. The reason tribes was used with spatial selection was the natural link to deme formation. However, tribes should be just as good at maintaining the diversity of a standard gene pool, so one idea to try is to implement tribes with a simple GA.

There is one major advantage to tribes: the fact that it does not always evaluate the child chromosome. This would suggest that if the evaluation function was more costly to apply, such as in a real timetabling problem, then it may start to show significant improvements over a simple GA. This is a very tentative idea, but it should be fairly easy to implement a metric which compares, say, effective trials per evaluation for tribes and a simple GA.

Trying different metrics

One reason that effective trials was employed was to take in account how much harder it was to find solutions as more were found. The allele count diversity suffers more than the solutions found for this disproportionate increase as diversity increases. More thought would be needed, but a similar 'taking into account amount found so far' modification to the allele count diversity could be made. As a general point, I think that more effort in general should be put in to looking at metrics for judging success, as the techniques available are currently very limited.

Testing problems with a range of diversity of solutions

My choice of uniform problem distribution came to note in a number of places, particularly in chapter 5.1. It would be interesting to devise a set of test problems that do not have such a uniform distribution of solutions across the search space, to see if there is really much difference in the way methods find solutions.

Improving Sharing Efficiency

Having found sharing to be so inefficient, I have doubts about suggesting further work in this area. However, in [Deb & Goldberg 89], Deb and Goldberg indicate that sharing might be tried with taking random samples of loci to share rather than the whole genotype. This seems like one good idea that could be tried.

In addition, taking only a small random sample from the population and calculating the sharing function according to this small sample might be an idea. Having experienced the problems with spatial sharing highlighted in chapter 5.8, caution has to be exercised here. The reason that this should be better than spatial sharing is that the sample is *random*. In spatial sharing we were hindered by destabilising a local area of genotypes trying to converge, not just trying to find representative examples of what the whole population was doing.

7.2.1 Concluding comments

This project set out to compare the ability of adaptations to GAs to find multiple distinct solutions. In the sense of finding distinct solutions in a single run, I have succeeded in finding two very able methods: sharing and tribes.

However, I realise that the number of distinct solutions found in a single run is not the only indicator of how well a method is performing. A more important measure of success is how efficiently distinct solutions can be found.

Due to the small test set of problems that I used in the project and the range of methods that I tried to cover, I restricted the amount of evidence I was able to collect, and so I have been unable to demonstrate any irrefutable facts about GAs. However, because the coverage of the project was so broad, I hope that I have managed to give a general feel for what kind of operators should be applied to produce multiple distinct solutions.

In tribes, I have presented a good example of the kind of operator that should be used, if any, to improve on a simple GA: it must work fast and maintain diversity while still allowing the GA to converge on solutions.

Finally, note the 'if any': one thing that this project has shown is that it is very difficult to improve on a simple GA.

Bibliography

[Collins & Jefferson 91] Robert J. Collins and David R. Jefferson. Selection in massively parallel genetic algorithms. In *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991.

[Corne 94] Dave Corne. Personal communication, 1994.

[Corne *et al.* 93] Dave Corne, Hsiao-Lan Fang, and Chris Mellish. Solving the modular exam scheduling problem with genetic algorithms. In *Proceedings of the Sixth International Conference in Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*. Gordon and Breach Science Publishers, 1993.

[Corne *et al.* 94] Dave Corne, Peter Ross, and Hsiao-Lan Fang. Fast practical evolutionary timetabling. In *Proceedings of the 1994 AISB Workshop on Evolutionary Computation*. Springer Verlag, 1994.

[Davidor 91] Yuval Davidor. A naturally occuring niche & species phenomenon: The model and first results. In *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991.

[De Jong 75] Kenneth A. De Jong. *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. Unpublished PhD thesis, University of Michigan, 1975. *Dissertation Abstracts International, 36*(10), 5104B.

[Deb & Goldberg 89] Kalyanmoy Deb and David E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the 3rd International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[Goldberg & Richardson 87] David E. Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the 2nd International*

Conference on Genetic Algorithms. Lawrence Erlbaum Publishers, 1987.

[Goldberg 89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1989.

[Hedberg 94] Sara Hedberg. Emerging genetic algorithms. *AI Expert*, September 1994.

[Heitkoetter & Beasley 94] Joerg Heitkoetter and David Beasley. The hitch-hiker's guide to evolutionary computation: A list of frequently asked questions (FAQ). USENET : comp.ai.genetic, 1994.

[Holland 75] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[Ross *et al.* 94] Peter Ross, Dave Corne, and Hsiao-Lan Fang. Successful lecture timetabling with evolutionary algorithms. In *Proceedings of the ECAI Workshop on Evolutionary Algorithms*, 1994.

[Syswerda 89] Gilbert Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[Tanese 89] Reiko Tanese. Distributed genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[Whitley 93] Darrell Whitley. A genetic algorithm tutorial. Technical Report CS-93-103, Department of Computer Science, Colorado State University, 1993.

[Wright 69] Sewall Wright. *Evolution and the Genetics of Populations. Volume 2: The Theory of Gene Frequencies*. University of Chicago Press, 1969.

Appendix A

Method and Results

This appendix is split into two sections: the first section specifies the method used and the second contains graphs of results for all the operators tested.

A.1 Method

All methods used the PGA program developed by Peter Ross and Geoffery H. Ballinger, modified for the purposes of this project. Documentation for this adaptation is included in appendix B and a full program listing is included in appendix C.

All experiments were conducted on Sun ELC machines running SunOS v4.1.3.

The PGA program was run with the following template flags:

```
-l250 -i10 -m0.05 -Cuniform -ett -N -X1000
```

These are: full report every 250 generations, report and measure average fitness every 10 generations, mutate at a rate of 0.05, use uniform crossover, use timetabling problems, non-interactive and stop after 1000 generations without the maximum of the average fitness having increased.

In addition the methods used the following flags:

- Control: -P1 -p<x> -rone -srank One subpopulation, population size <x>, <x> $= 49, 100, 225, 400, 625$, steady-state reproduction, rank selection.

- Spatial Selection: -P1 -p<x> -rssone<y> One subpopulation, population size <x>, <x> $= 49, 100, 225, 400, 625$, steady-state spatial selection, random walk <y>. The random walk lengths for each population size are shown in table A–1.

- Islands: -P<y> -p<x> -rone -srank <y> subpopulations, <y> $= 2, 3, 5$, population size <x>, <x> $= 60, 120, 240, 360, 600$, steady-state reproduction, rank selection. The default PGA migration interval is 10.

- Crowding: -P1 -p<x> -rone -srank -Db<y> One subpopulation, population size <x>, <x> = 49, 100, 225, 400, 625, steady-state reproduction, rank selection, crowding factor <y>, <y> = 2, 3, 5.

- Spatial Crowding: -P1 -p<x> -rssone<z> -Db<y> One subpopulation, population size <x>, <x> = 49, 100, 225, 400, 625, steady-state spatial selection, random walk <z> (the random walk lengths for each population size are shown in table A–1, under population divisor 5.0), crowding factor <y>, <y> = 2, 3, 5.

- Sharing: -P1 -p<x> -rone -srank -Ea<y> One subpopulation, population size <x>, <x> = 49, 100, 225, 400, 625, steady-state reproduction, rank selection, sharing with σ_{share} <y>, <y> = 0.4, 0.6, 0.8.

- Spatial Sharing: -P1 -p<x> -rssone<z> -Eb<y> One subpopulation, population size <x>, <x> = 49, 100, 225, 400, 625, steady-state spatial selection, random walk <z> (the random walk lengths for each population size are shown in table A–1, under population divisor 5.0), sharing with σ_{share} <y>, <y> = 0.4, 0.6, 0.8 over the chromosomes on the grid within the random walk length of each individual.

- Tribes: -P1 -p<x> -rssone<z> -Y<y> One subpopulation, population size <x>, <x> = 49, 100, 225, 400, 625, steady-state spatial selection, random walk <z> (the random walk lengths for each population size are shown in table A–1, under population divisor 5.0), tribes with σ_{share} <y>, <y> = 0.4, 0.6, 0.8.

Population Size	Population Divisor		
	1.5	2.5	5.0
49	5	3	1
100	7	4	2
225	10	6	3
400	13	8	4
625	17	10	5

Table A–1: Random walk lengths for each population divisor / population size combination

For each of the methods above the following problems were tested:

- 120 solutions, tt120

- 60 solutions, tt60

- 30 solutions, tt30

- 10 solutions, tt10

The data files of constraints specifying these problems, `ttdata.120`, `ttdata.60`, `ttdata.30` and `ttdata.10`, were loaded into PGA with the `-F` option. These data files are included in with the program code in C.

For each method, parameter, population size and problem PGA was run with ten different seeds using the `-S` flag. The seeds were: 2866, 3435, 4464, 13455, 17925, 20863, 21957, 25902, 26275, 30301.

For each method, parameter, population size and problem averages of the peak diversity reached, solutions when peak diversity was attained and time to reach peak diversity were recorded. These were averaged for the 10 seeds. If the peak diversity was 0 then the time for the whole run was taken to be the time to reach peak diversity.

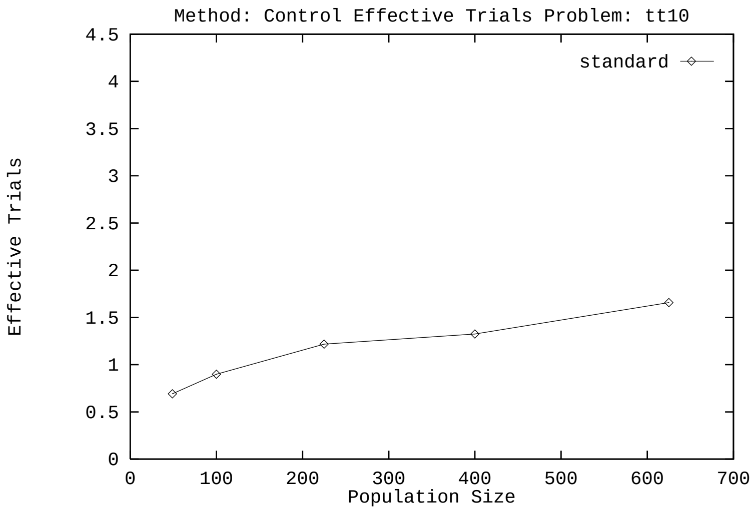
The average of the solutions was used to convert to the effective number of trials for the method. The efficiency values were obtained by dividing the average of peak diversity or the average of effective trials (as relevant) by the average time to reach peak diversity.

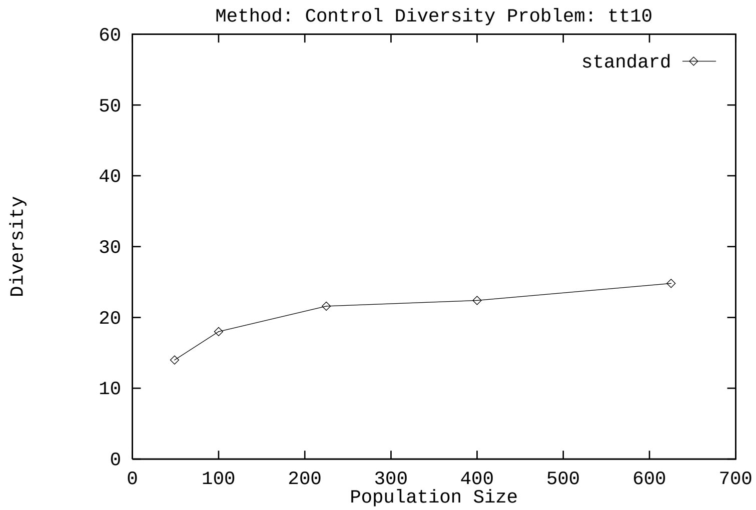
In the next section there follow graphs of plotting the data derived:

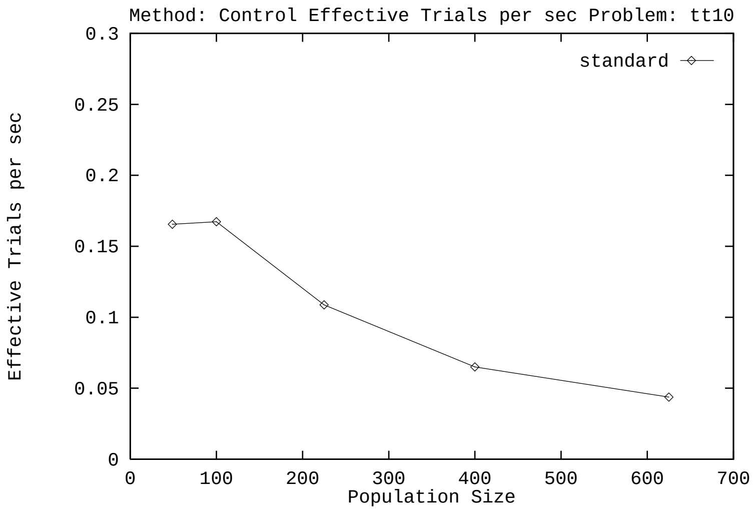
- Peak Diversity (labelled Diversity)
- Peak Diversity divided by time to reach peak diversity (labelled Diversity per second)
- Effective Trials (labelled Effective Trials)
- Effective Trials divided by time to reach peak diversity (labelled Effective Trials per second)

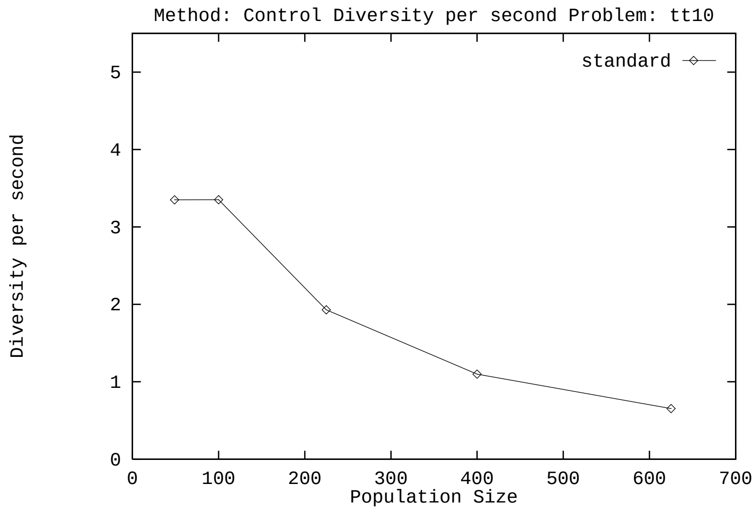
A.2 Results

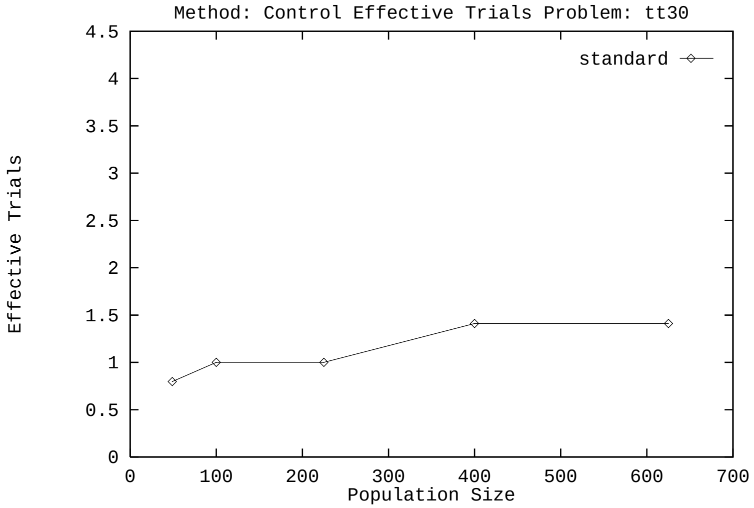
This section provides graphs of all the experiments conducted. There derivation is explained in the previous section. Each method has four graphs for each problem it was tested on.

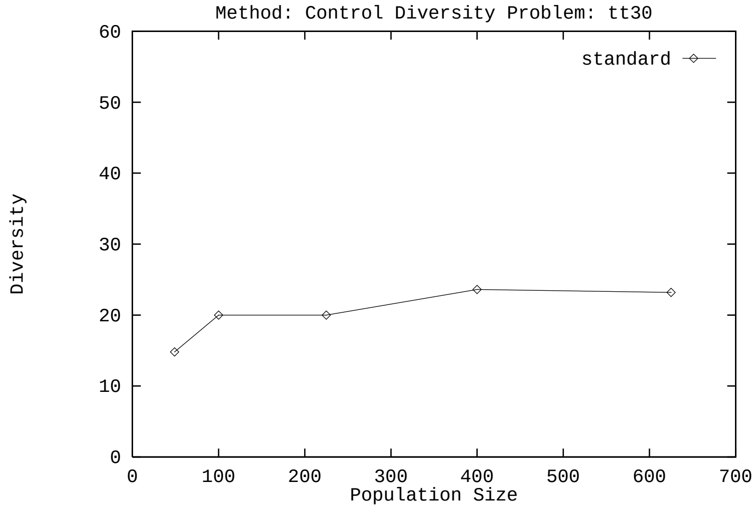

Method: Control Effective Trials Problem: tt10
Effective Trials
Population Size
standard
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
100
200
300
400
500
600
700

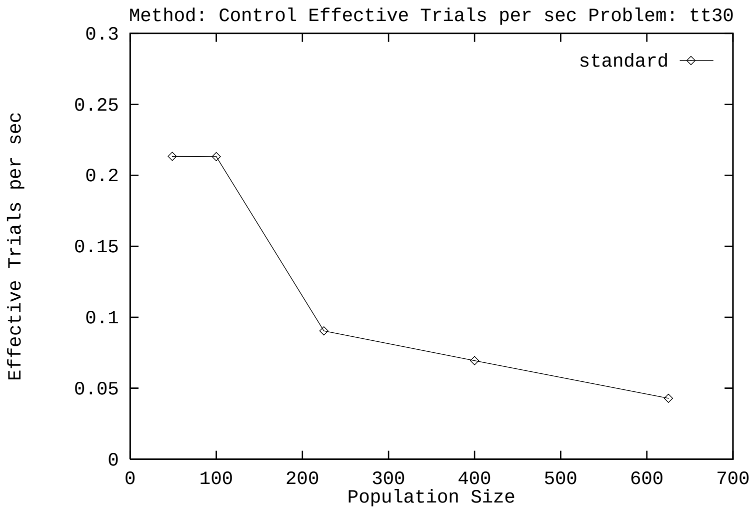

Method: Control Diversity Problem: tt10
Diversity
Population Size
standard
0
10
20
30
40
50
60
100
200
300
400
500
600
700

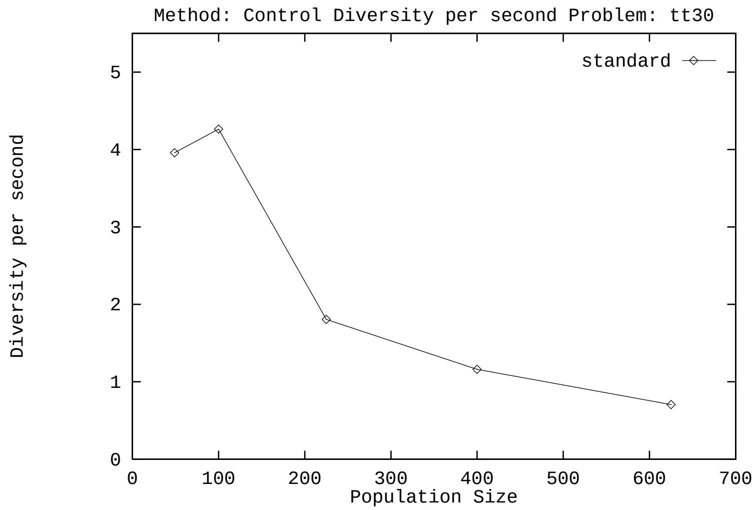

Method: Control Effective Trials per sec Problem: tt10
Effective Trials per sec
Population Size
standard
0
0.05
0.1
0.15
0.2
0.25
0.3
100
200
300
400
500
600
700

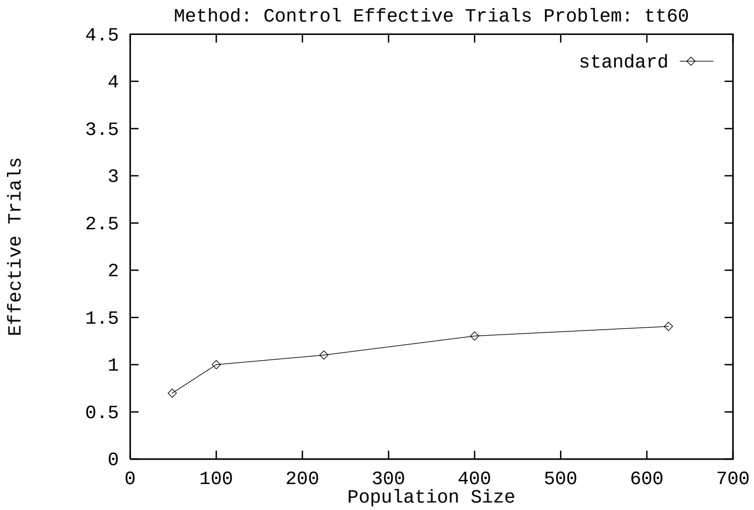

Method: Control Diversity per second Problem: tt10
Diversity per second
Population Size
standard
0
1
2
3
4
5
100
200
300
400
500
600
700

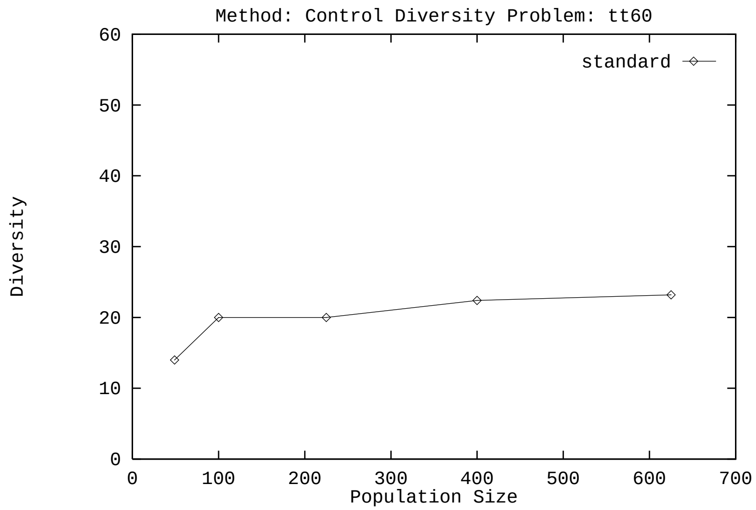

Method: Control Effective Trials Problem: tt30
Effective Trials
standard
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
0
100
200
300
400
500
600
700
Population Size

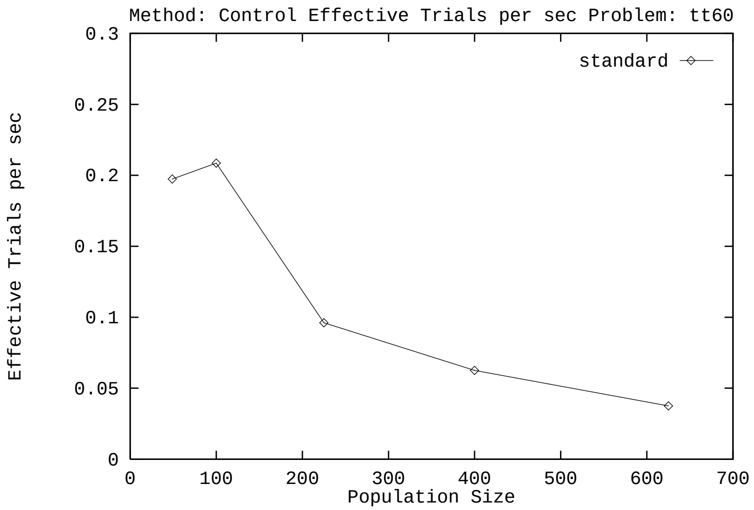

Method: Control Diversity Problem: tt30
Diversity
standard
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700
Population Size

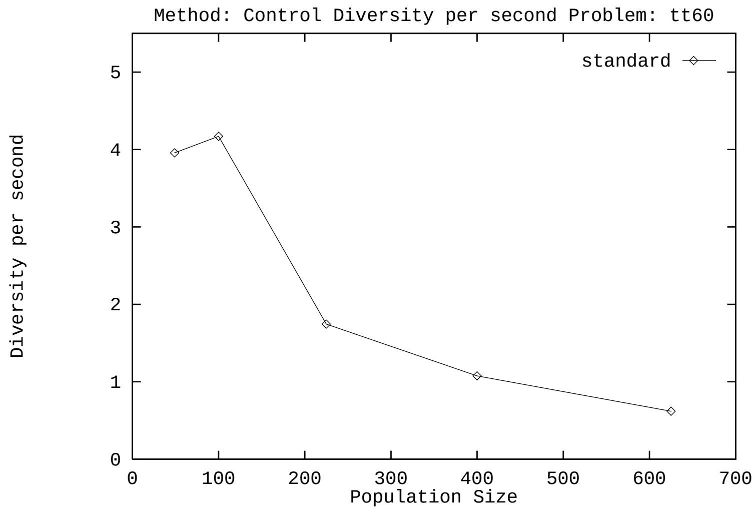

Method: Control Effective Trials per sec Problem: tt30
Effective Trials per sec
standard
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700
Population Size

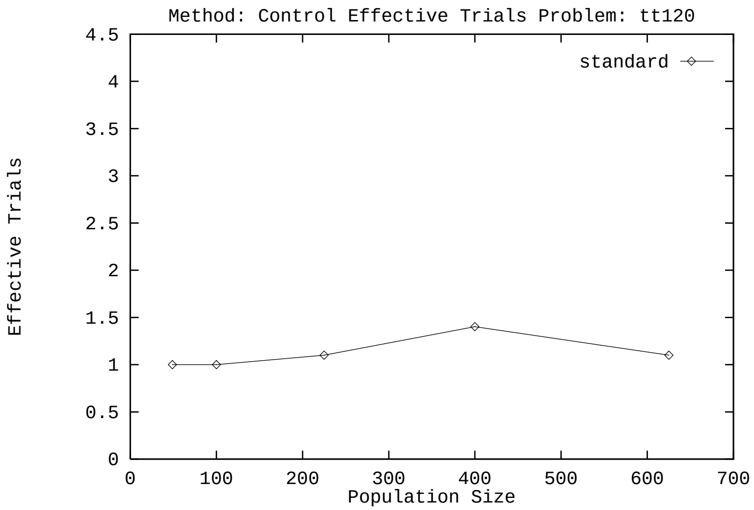

Method: Control Diversity per second Problem: tt30
Diversity per second
standard
0
1
2
3
4
5
0
100
200
300
400
500
600
700
Population Size

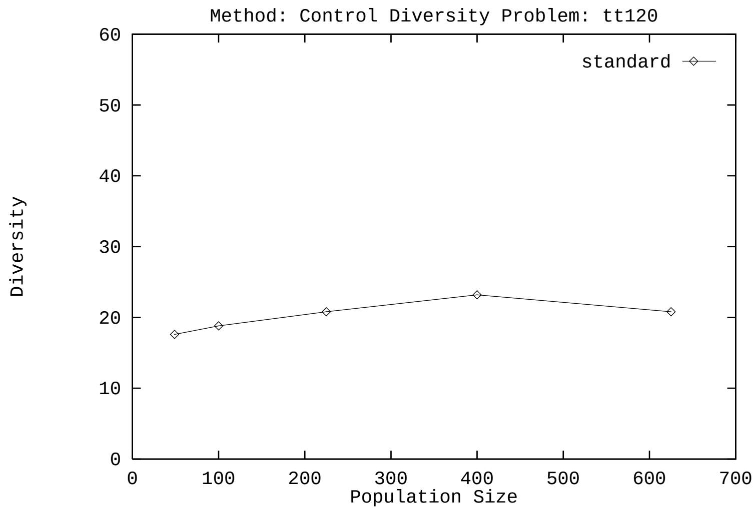

Method: Control Effective Trials Problem: tt60
standard
Effective Trials
Population Size
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
100
200
300
400
500
600
700

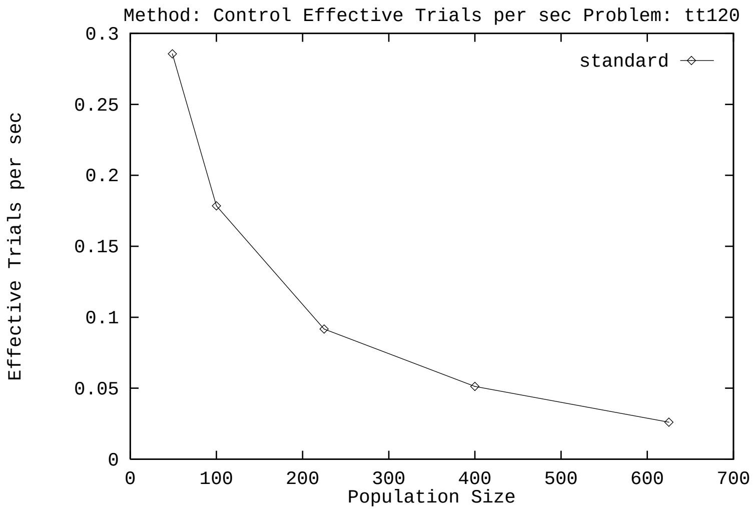

Method: Control Diversity Problem: tt60
standard
Diversity
Population Size
60
50
40
30
20
10
0
0
100
200
300
400
500
600
700

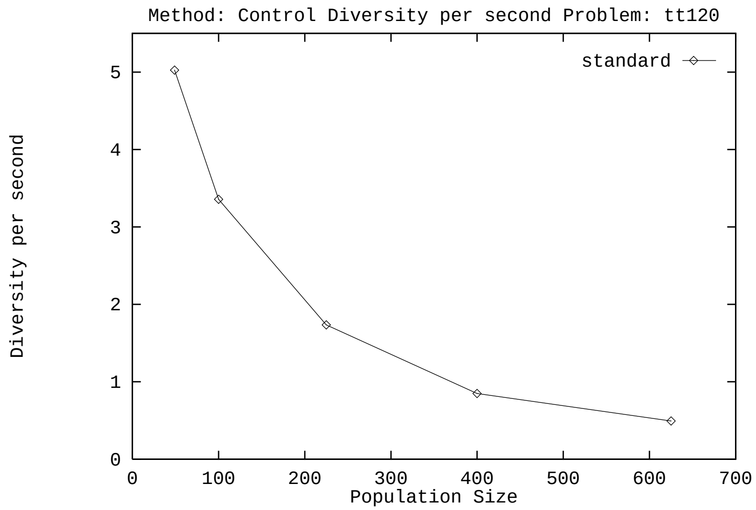

Method: Control Effective Trials per sec Problem: tt60
standard
Effective Trials per sec
Population Size
0.3
0.25
0.2
0.15
0.1
0.05
0
0
100
200
300
400
500
600
700

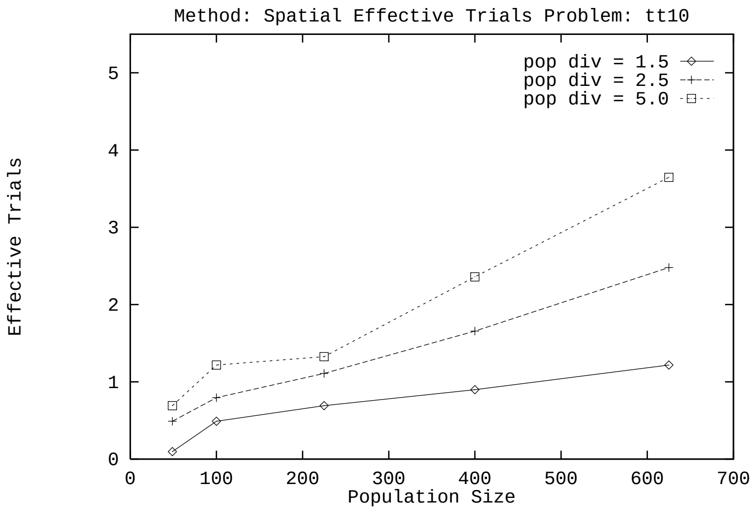

Method: Control Diversity per second Problem: tt60
standard
Diversity per second
Population Size
5
4
3
2
1
0
0
100
200
300
400
500
600
700

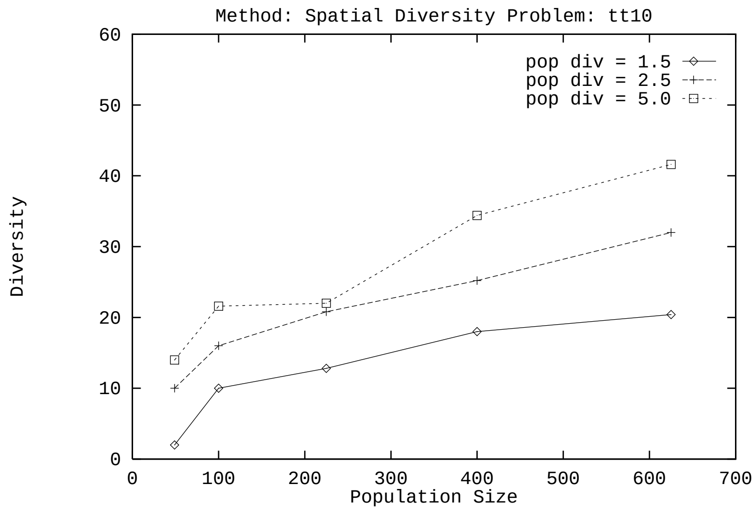

Method: Control Effective Trials Problem: tt120
Effective Trials
Population Size
standard
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
0
100
200
300
400
500
600
700

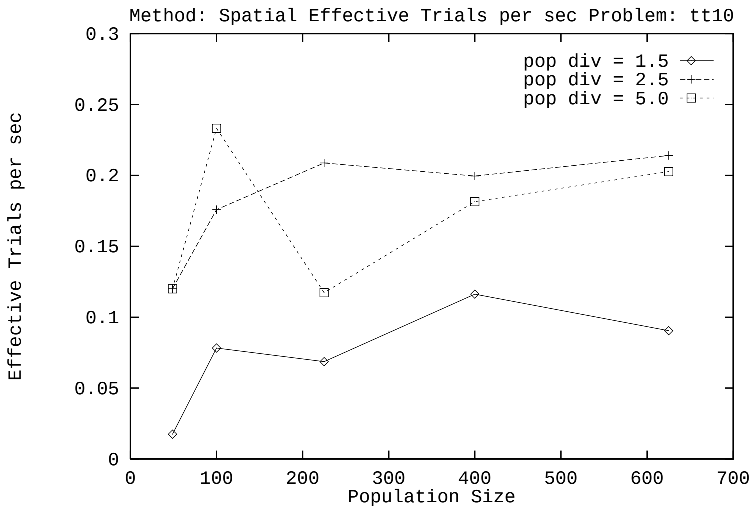

Method: Control Diversity Problem: tt120
Diversity
Population Size
standard
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

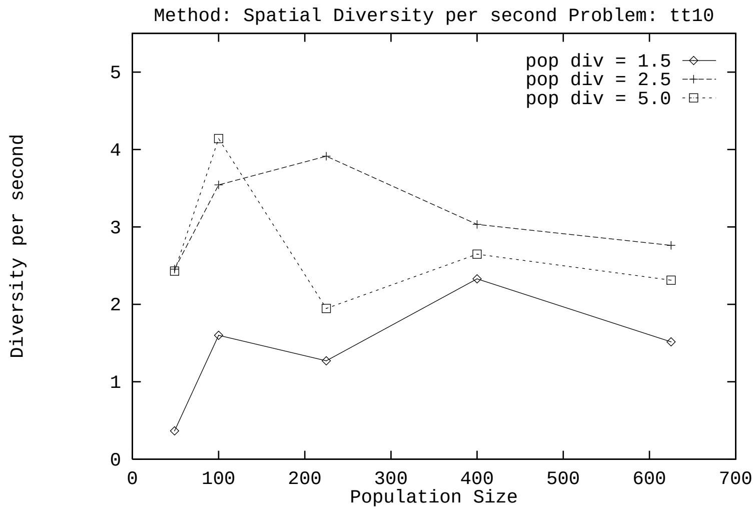

Method: Control Effective Trials per sec Problem: tt120
Effective Trials per sec
Population Size
standard
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

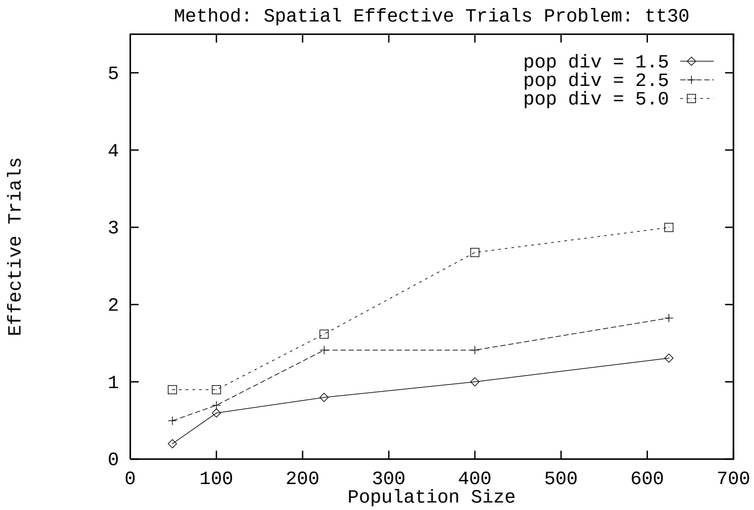

Method: Control Diversity per second Problem: tt120
Diversity per second
Population Size
standard
0
1
2
3
4
5
0
100
200
300
400
500
600
700

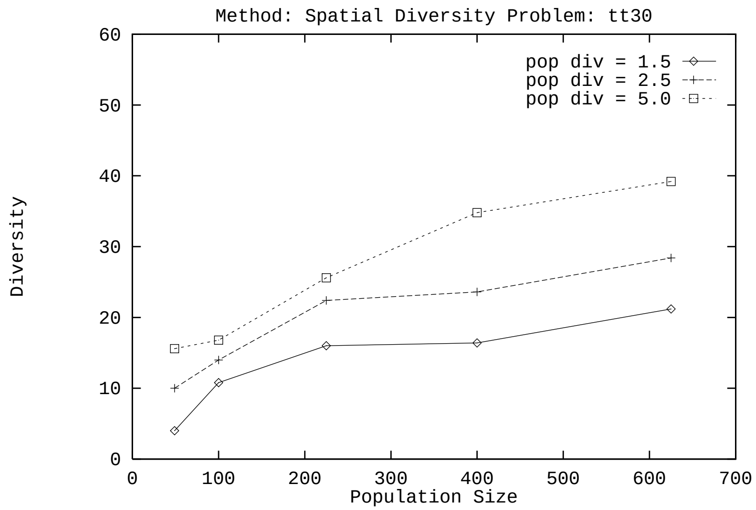

Method: Spatial Effective Trials Problem: tt10
Effective Trials
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0

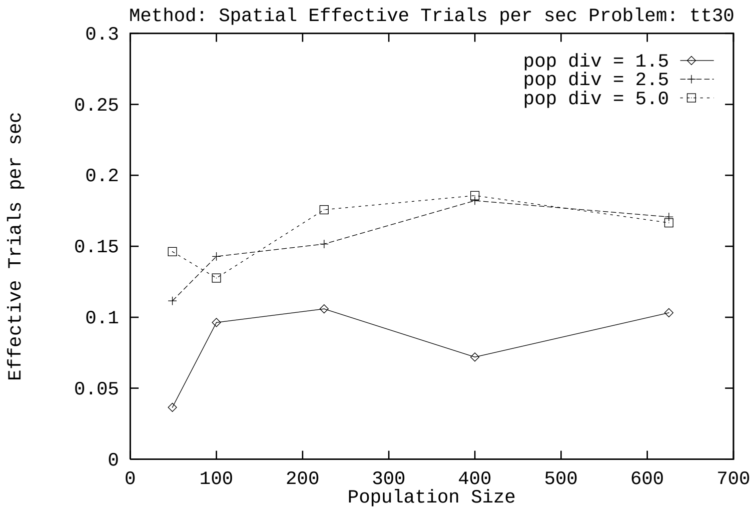

Method: Spatial Diversity Problem: tt10
Diversity
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0

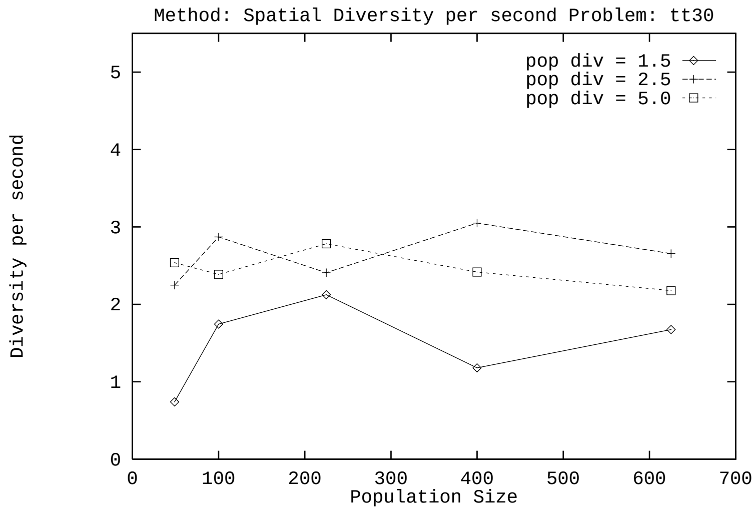

Method: Spatial Effective Trials per sec Problem: tt10
Effective Trials per sec
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0

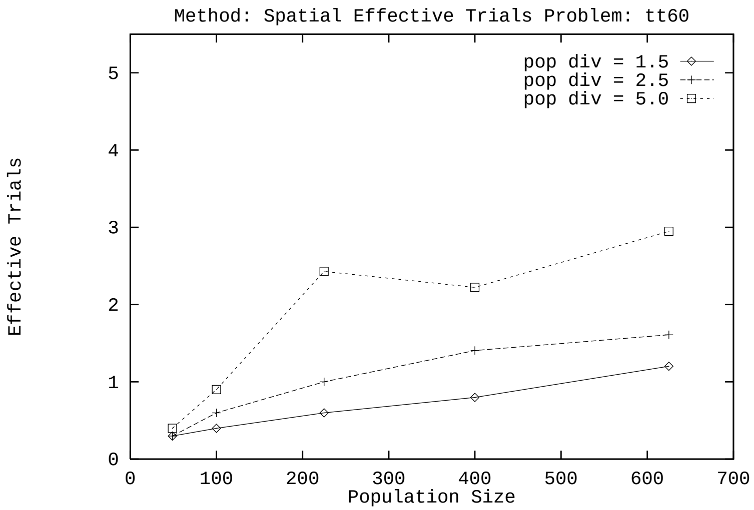

Method: Spatial Diversity per second Problem: tt10
Diversity per second
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0

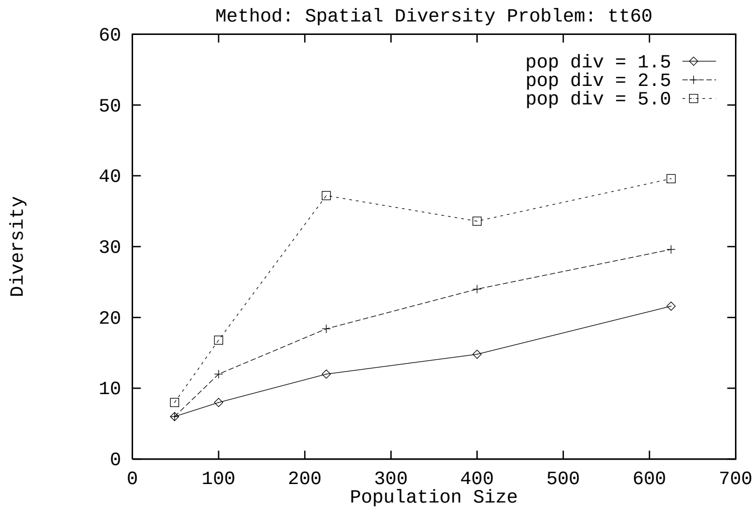

Method: Spatial Effective Trials Problem: tt30
Effective Trials
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0
0
1
2
3
4
5
0
100
200
300
400
500
600
700

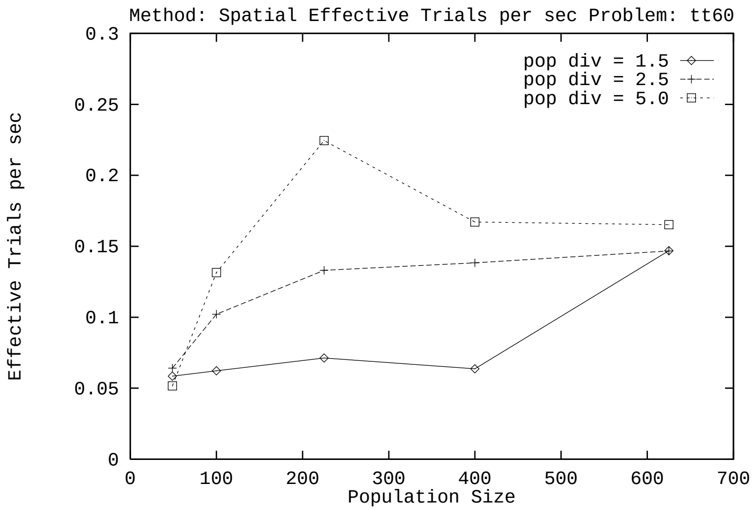

Method: Spatial Diversity Problem: tt30
Diversity
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

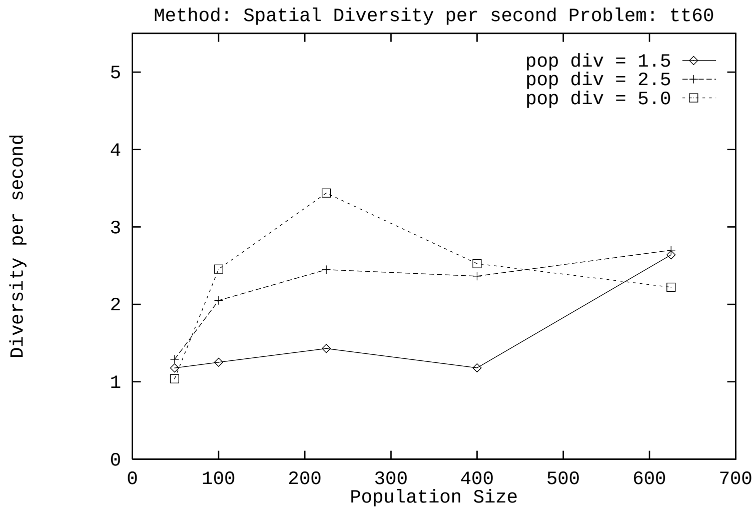

Method: Spatial Effective Trials per sec Problem: tt30
Effective Trials per sec
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

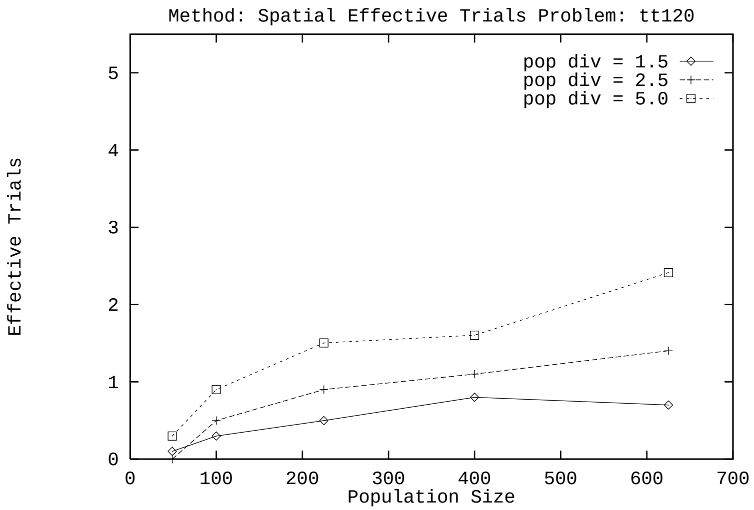

Method: Spatial Diversity per second Problem: tt30
Diversity per second
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0
0
1
2
3
4
5
0
100
200
300
400
500
600
700

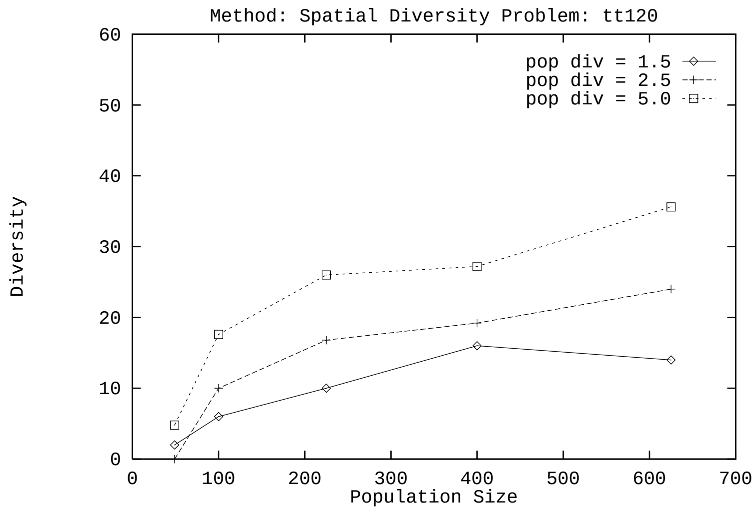

Method: Spatial Effective Trials Problem: tt60
Effective Trials
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0
0
1
2
3
4
5
100
200
300
400
500
600
700

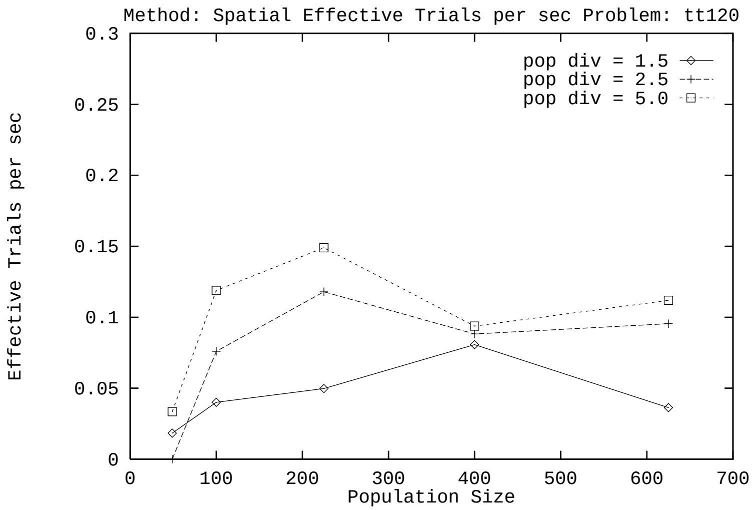

Method: Spatial Diversity Problem: tt60
Diversity
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0
0
10
20
30
40
50
60
100
200
300
400
500
600
700

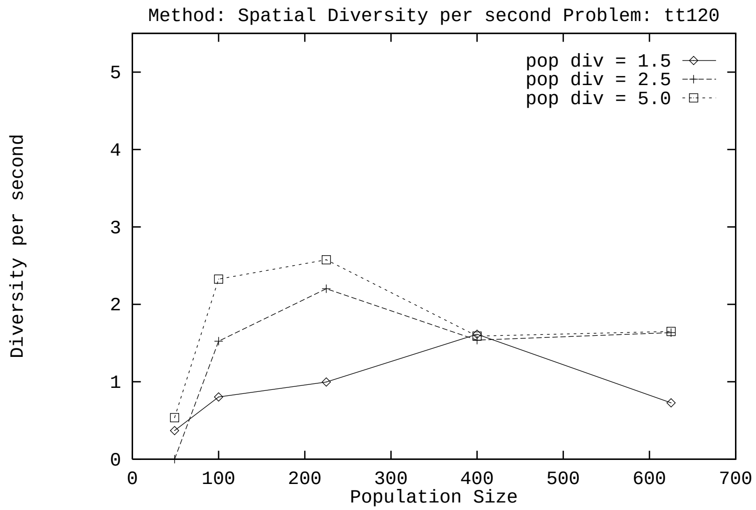

Method: Spatial Effective Trials per sec Problem: tt60
Effective Trials per sec
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0
0
0.05
0.1
0.15
0.2
0.25
0.3
100
200
300
400
500
600
700

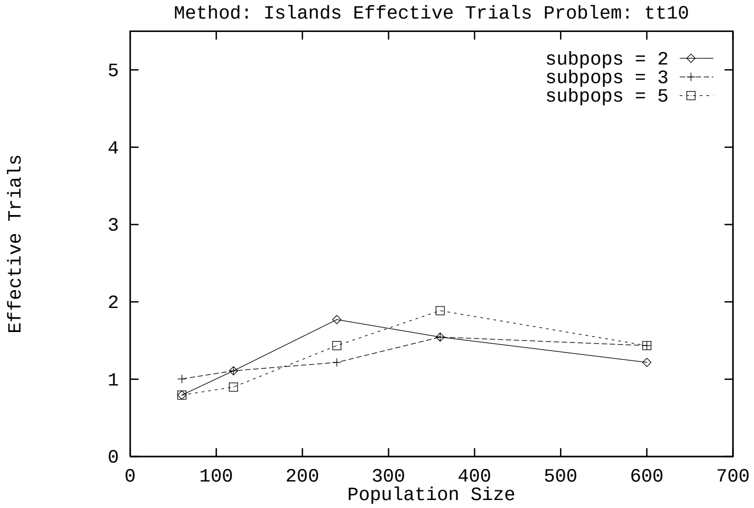

Method: Spatial Diversity per second Problem: tt60
Diversity per second
Population Size
pop div = 1.5
pop div = 2.5
pop div = 5.0
0
1
2
3
4
5
100
200
300
400
500
600
700

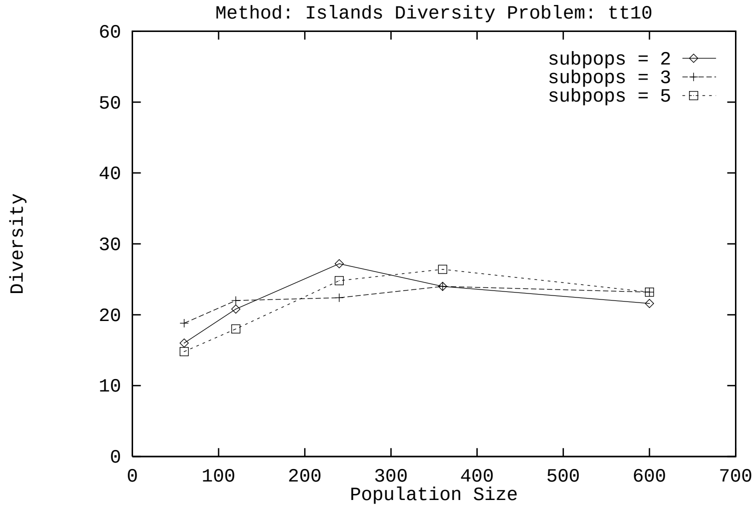

Method: Spatial Effective Trials Problem: tt120
pop div = 1.5
pop div = 2.5
pop div = 5.0
Effective Trials
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

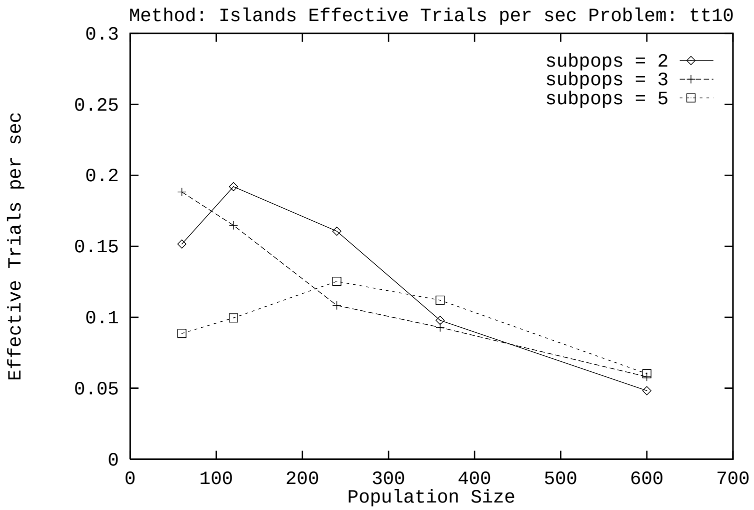

Method: Spatial Diversity Problem: tt120
pop div = 1.5
pop div = 2.5
pop div = 5.0
Diversity
Population Size
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

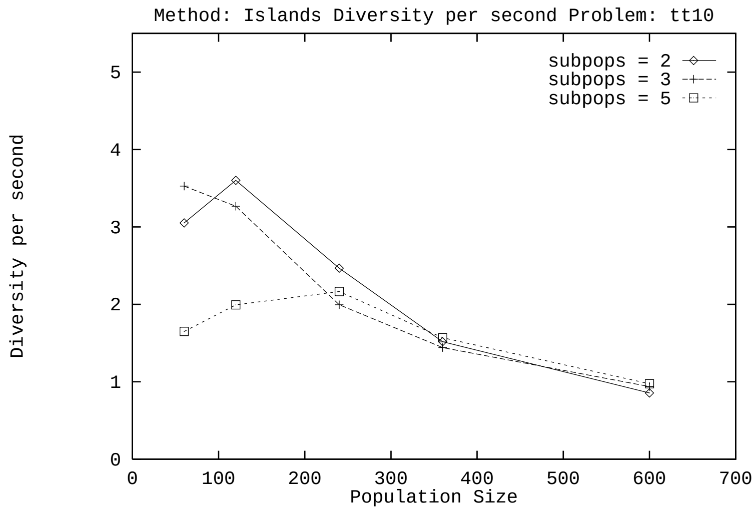

Method: Spatial Effective Trials per sec Problem: tt120
pop div = 1.5
pop div = 2.5
pop div = 5.0
Effective Trials per sec
Population Size
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

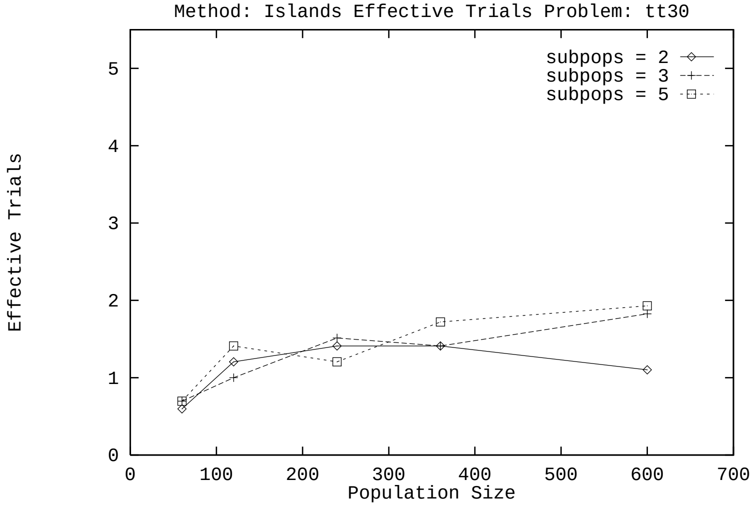

Method: Spatial Diversity per second Problem: tt120
pop div = 1.5
pop div = 2.5
pop div = 5.0
Diversity per second
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

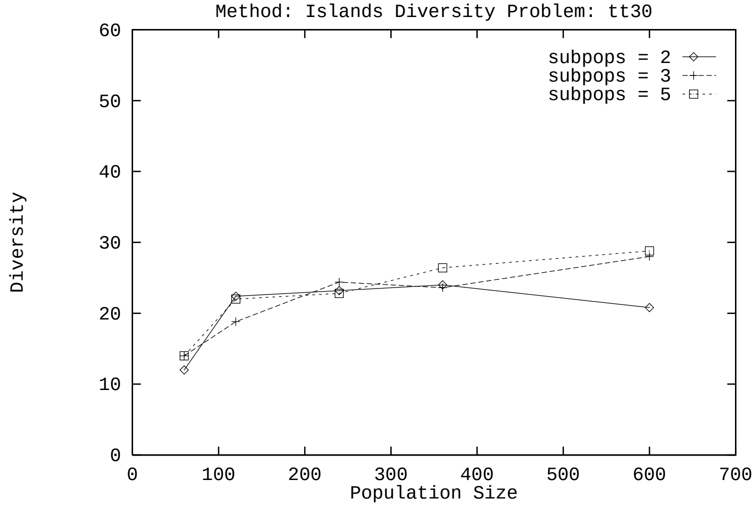

Method: Islands Effective Trials Problem: tt10
Effective Trials
Population Size
subpops = 2
subpops = 3
subpops = 5
0
1
2
3
4
5
0
100
200
300
400
500
600
700

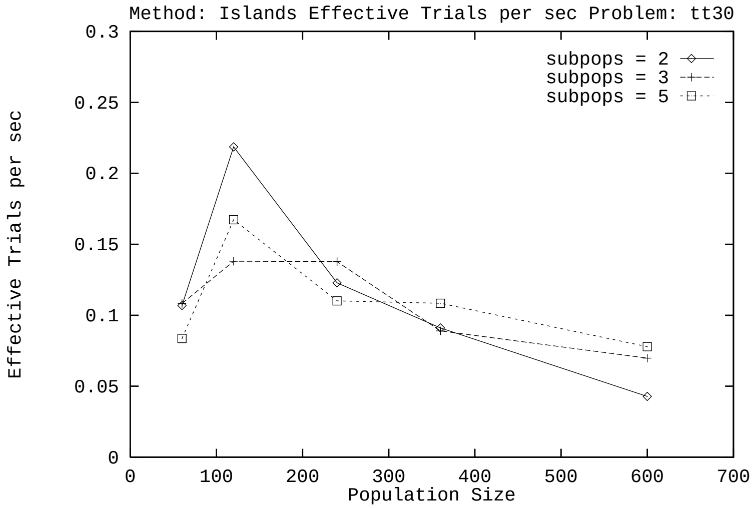

Method: Islands Diversity Problem: tt10
Diversity
Population Size
subpops = 2
subpops = 3
subpops = 5
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

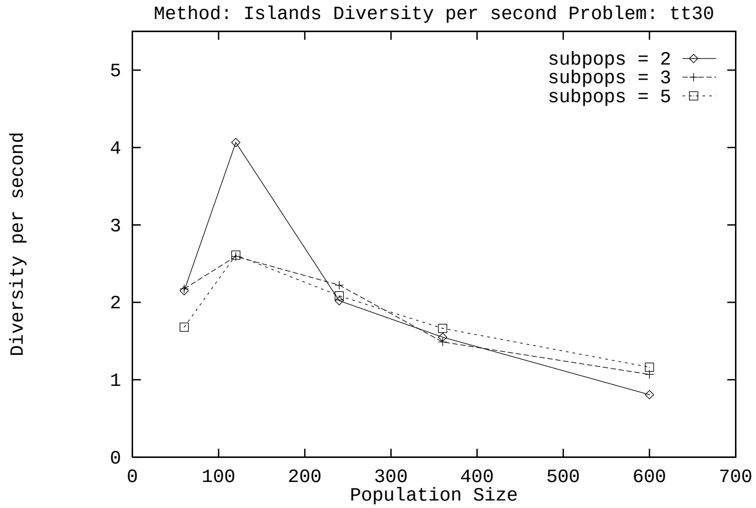

Method: Islands Effective Trials per sec Problem: tt10
Effective Trials per sec
Population Size
subpops = 2
subpops = 3
subpops = 5
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

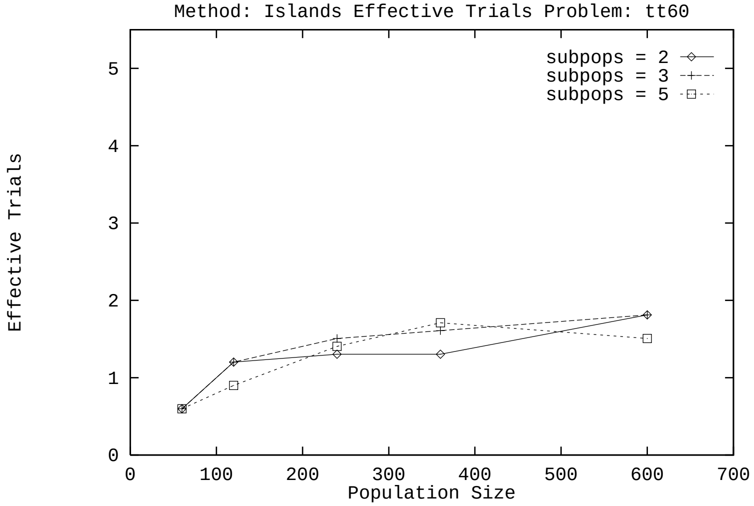

Method: Islands Diversity per second Problem: tt10
Diversity per second
Population Size
subpops = 2
subpops = 3
subpops = 5
0
1
2
3
4
5
0
100
200
300
400
500
600
700

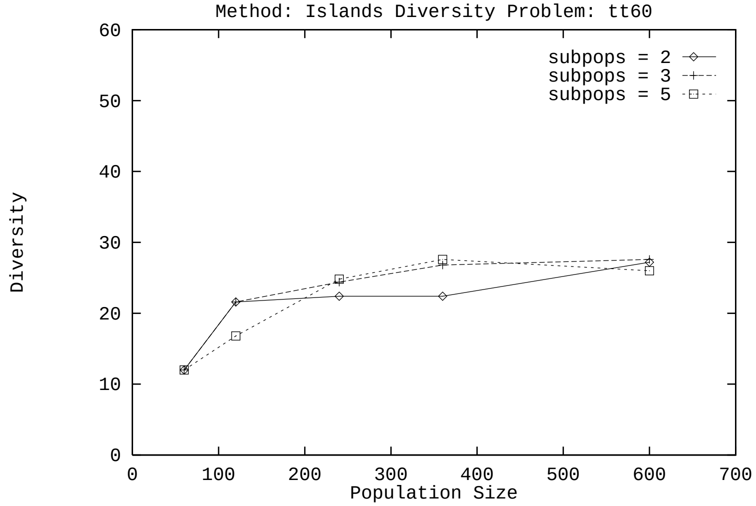

Method: Islands Effective Trials Problem: tt30
Effective Trials
Population Size
subpops = 2
subpops = 3
subpops = 5
0
1
2
3
4
5
0
100
200
300
400
500
600
700

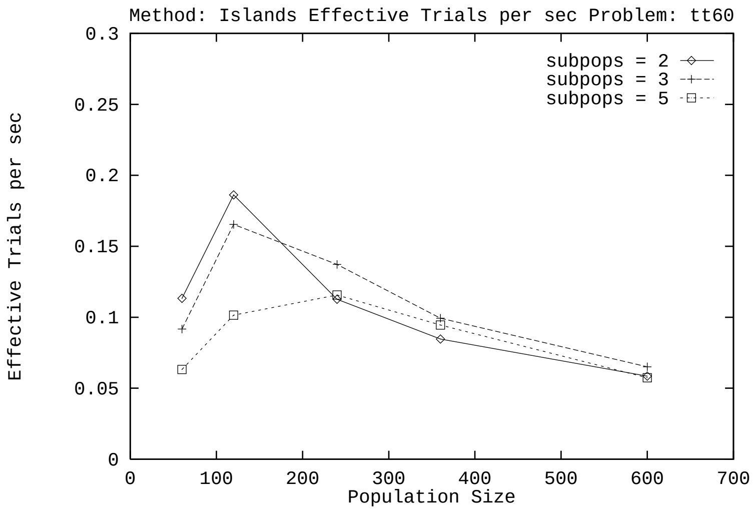

Method: Islands Diversity Problem: tt30
Diversity
Population Size
subpops = 2
subpops = 3
subpops = 5
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

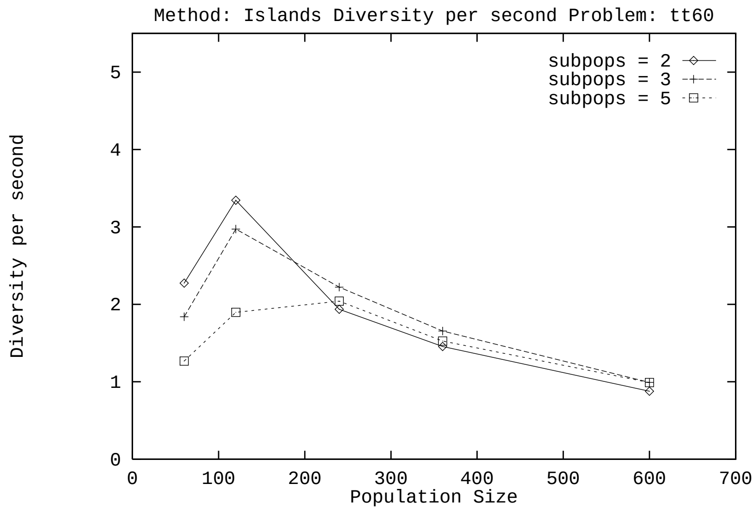

Method: Islands Effective Trials per sec Problem: tt30
Effective Trials per sec
Population Size
subpops = 2
subpops = 3
subpops = 5
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

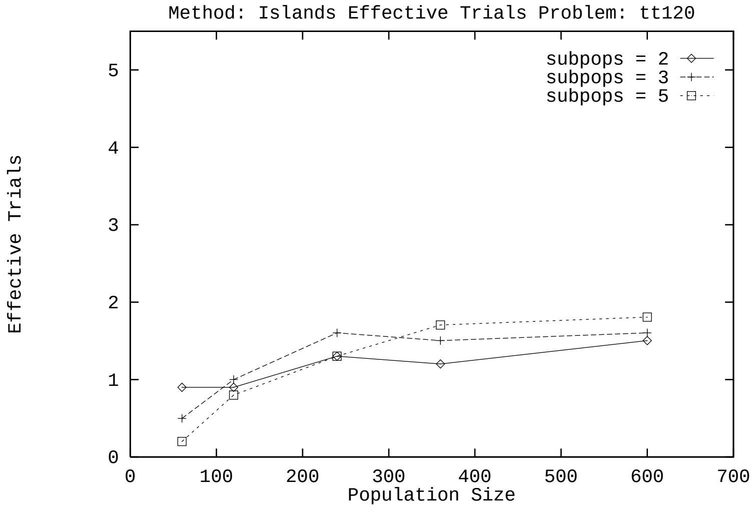

Method: Islands Diversity per second Problem: tt30
Diversity per second
Population Size
subpops = 2
subpops = 3
subpops = 5
0
1
2
3
4
5
0
100
200
300
400
500
600
700

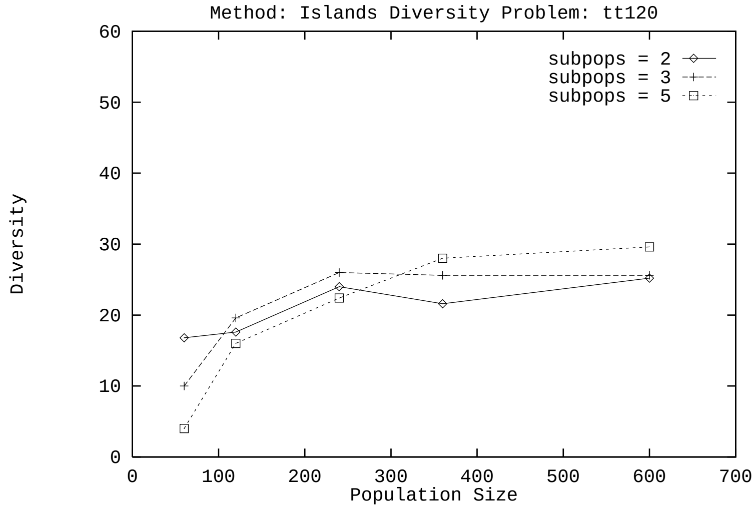

Method: Islands Effective Trials Problem: tt60
Effective Trials
Population Size
subpops = 2
subpops = 3
subpops = 5

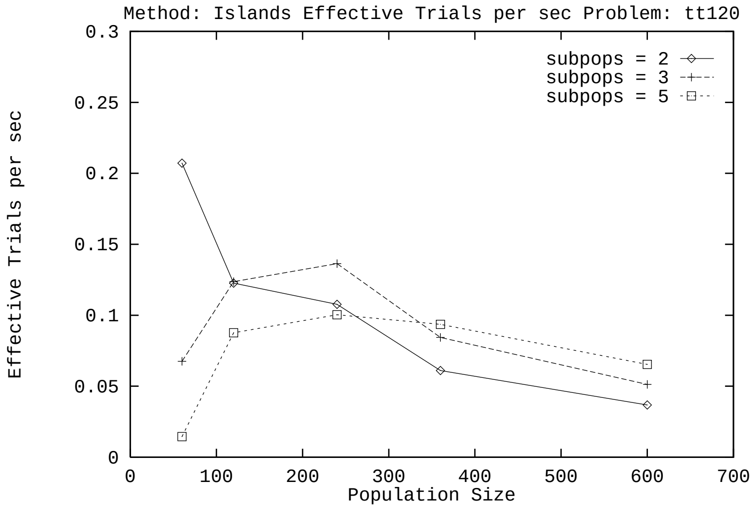

Method: Islands Diversity Problem: tt60
Diversity
Population Size
subpops = 2
subpops = 3
subpops = 5

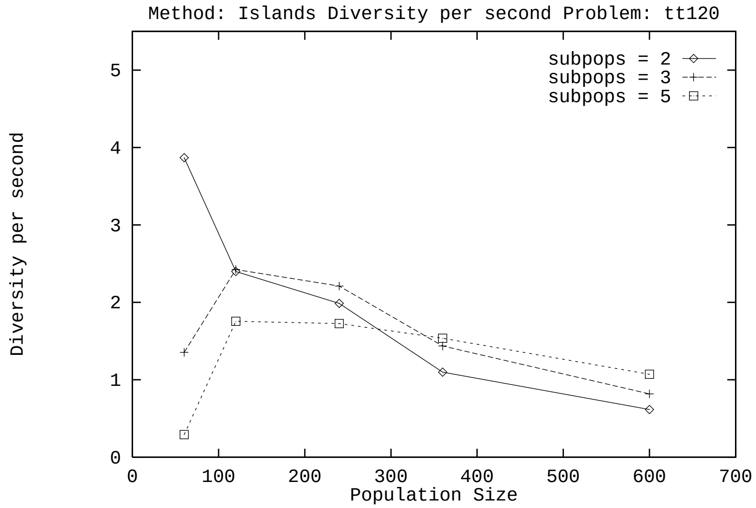

Method: Islands Effective Trials per sec Problem: tt60
Effective Trials per sec
Population Size
subpops = 2
subpops = 3
subpops = 5

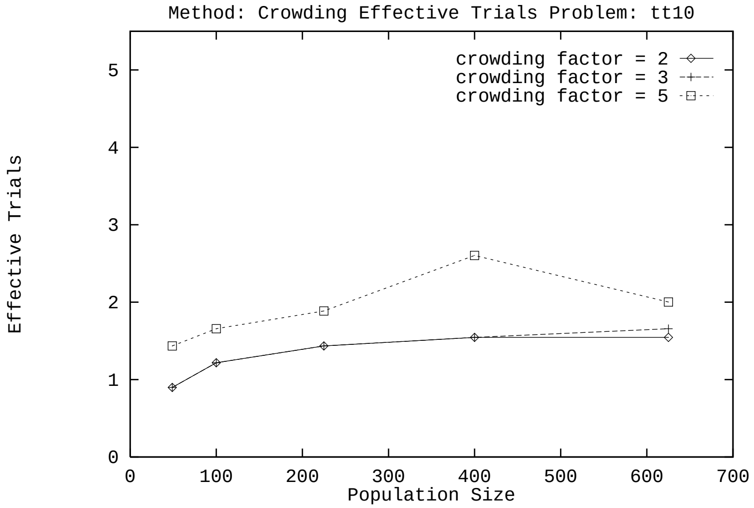

Method: Islands Diversity per second Problem: tt60
Diversity per second
Population Size
subpops = 2
subpops = 3
subpops = 5

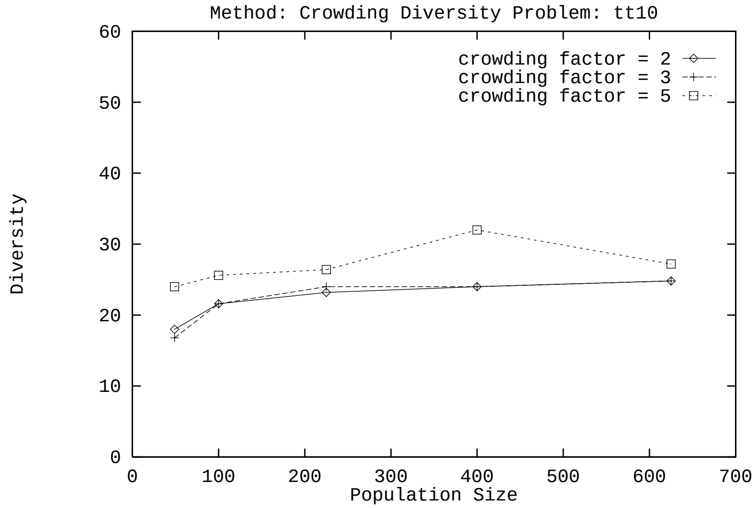

Method: Islands Effective Trials Problem: tt120
subpops = 2
subpops = 3
subpops = 5
Effective Trials
Population Size

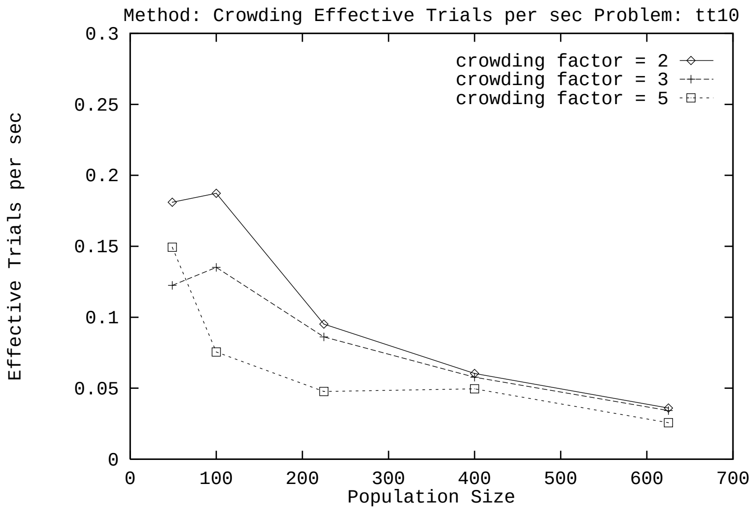

Method: Islands Diversity Problem: tt120
subpops = 2
subpops = 3
subpops = 5
Diversity
Population Size

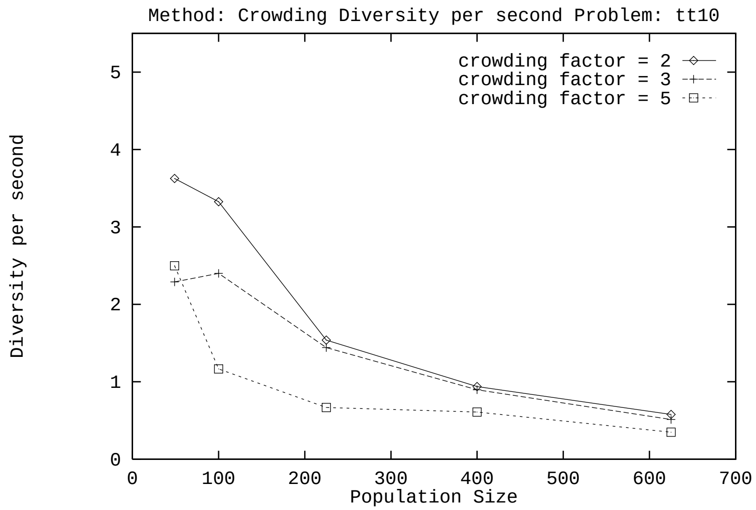

Method: Islands Effective Trials per sec Problem: tt120
subpops = 2
subpops = 3
subpops = 5
Effective Trials per sec
Population Size

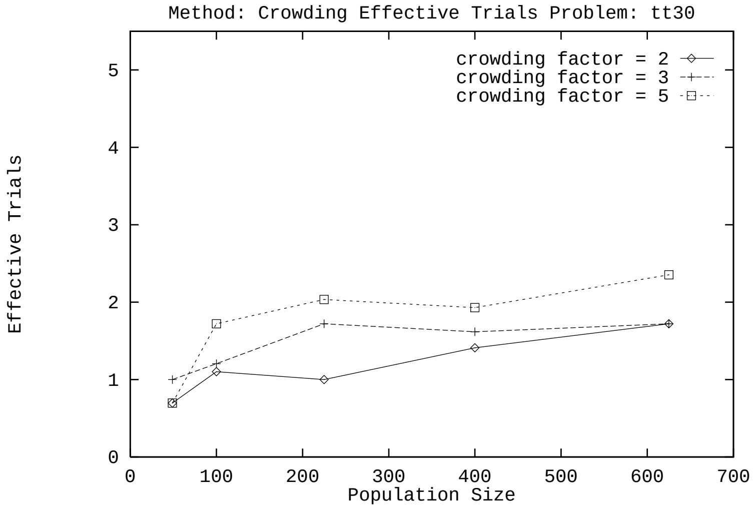

Method: Islands Diversity per second Problem: tt120
subpops = 2
subpops = 3
subpops = 5
Diversity per second
Population Size

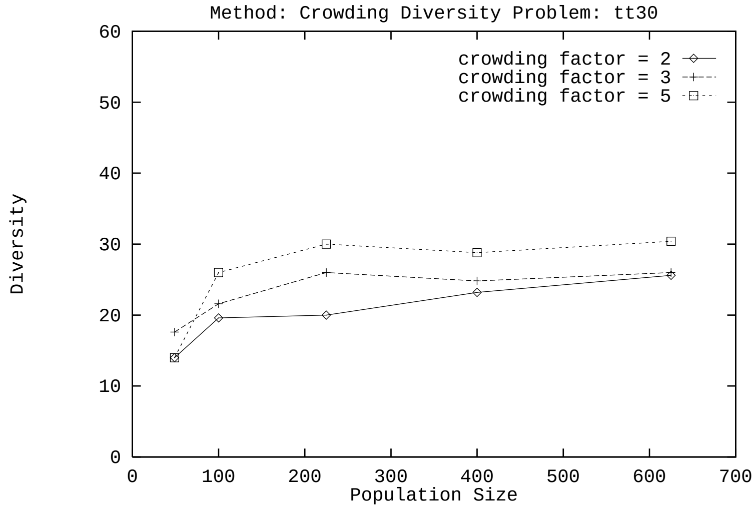

Method: Crowding Effective Trials Problem: tt10
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

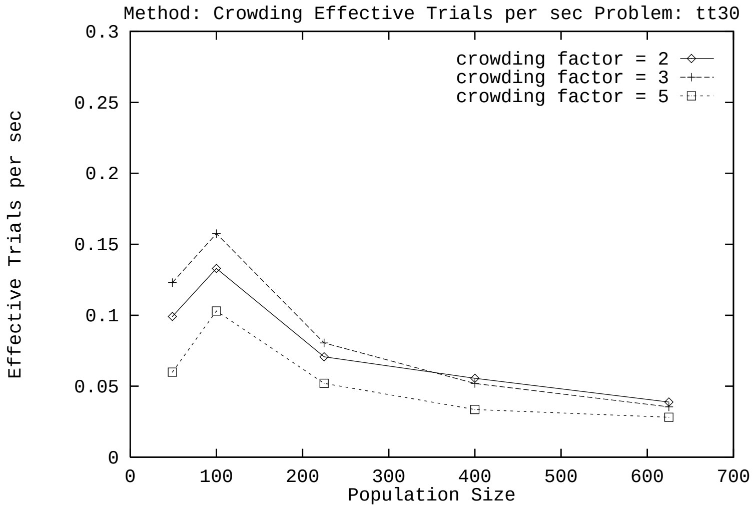

Method: Crowding Diversity Problem: tt10
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity
Population Size
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

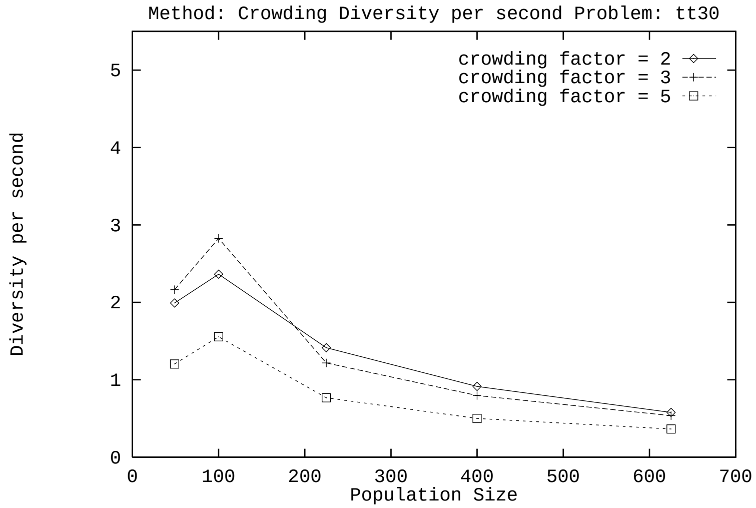

Method: Crowding Effective Trials per sec Problem: tt10
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials per sec
Population Size
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

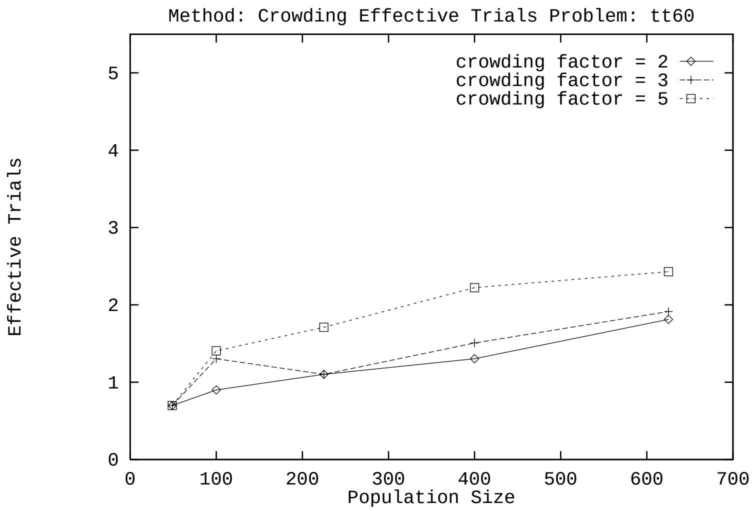

Method: Crowding Diversity per second Problem: tt10
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity per second
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

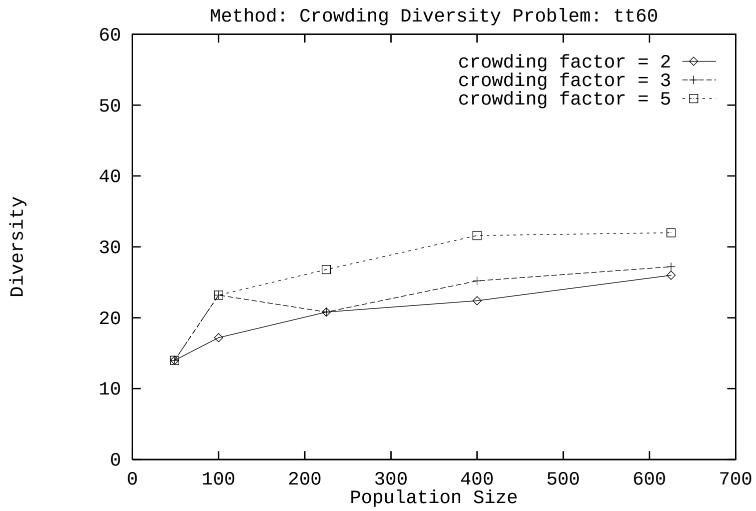

Method: Crowding Effective Trials Problem: tt30
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials
Population Size

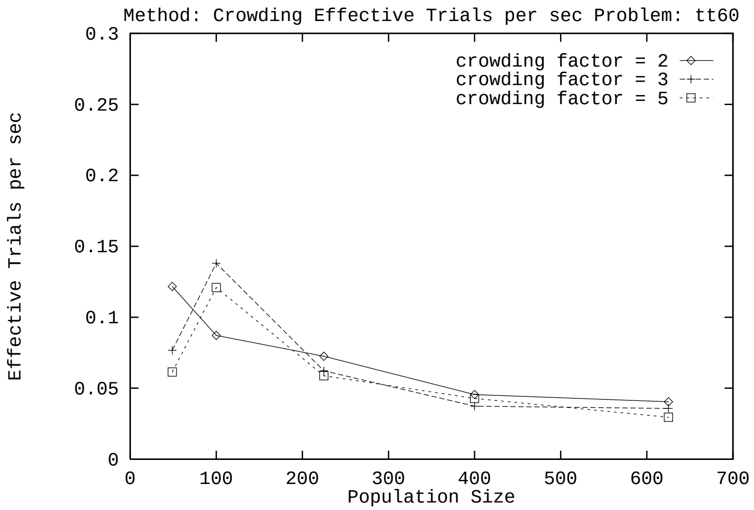

Method: Crowding Diversity Problem: tt30
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity
Population Size

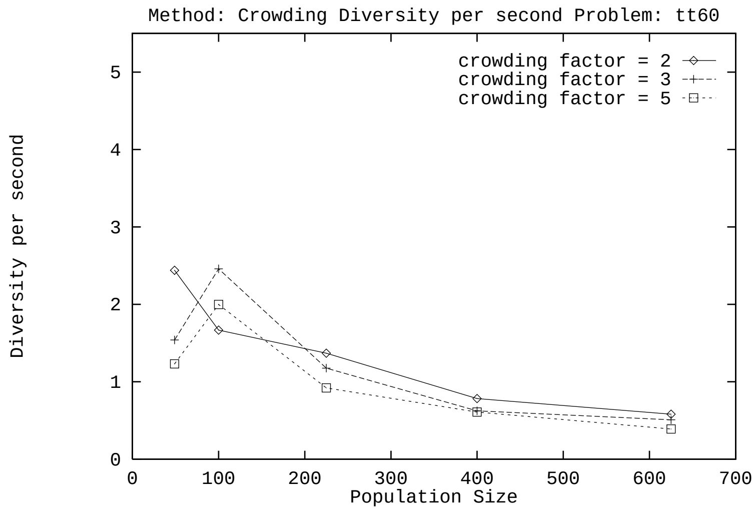

Method: Crowding Effective Trials per sec Problem: tt30
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials per sec
Population Size

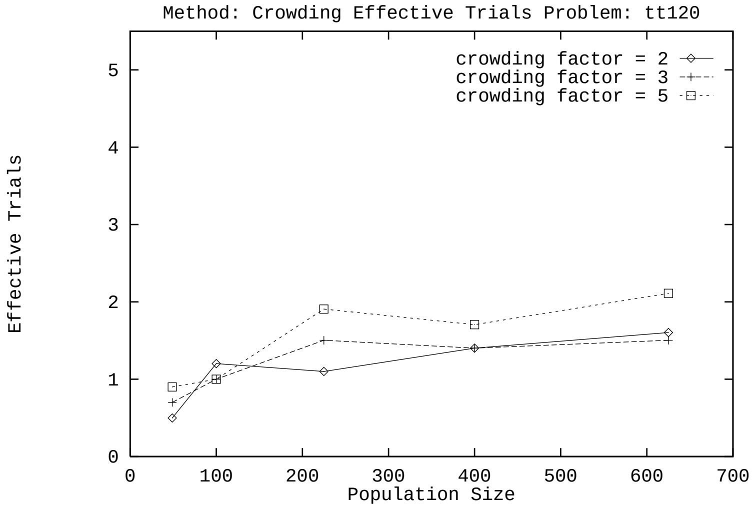

Method: Crowding Diversity per second Problem: tt30
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity per second
Population Size

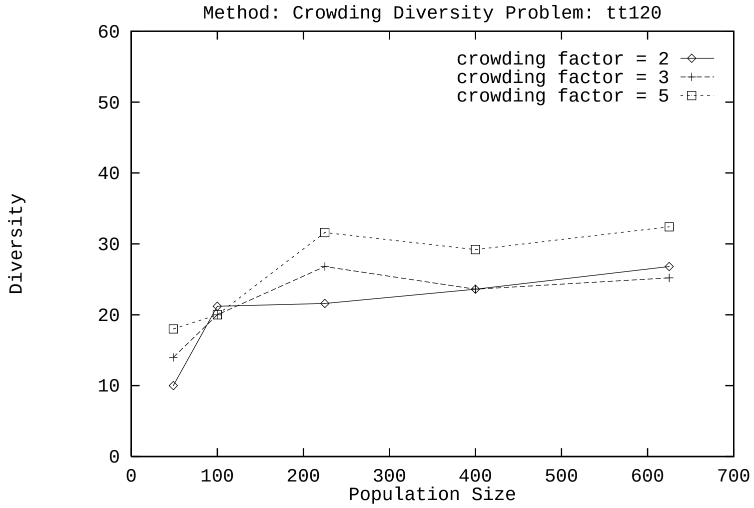

Method: Crowding Effective Trials Problem: tt60
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

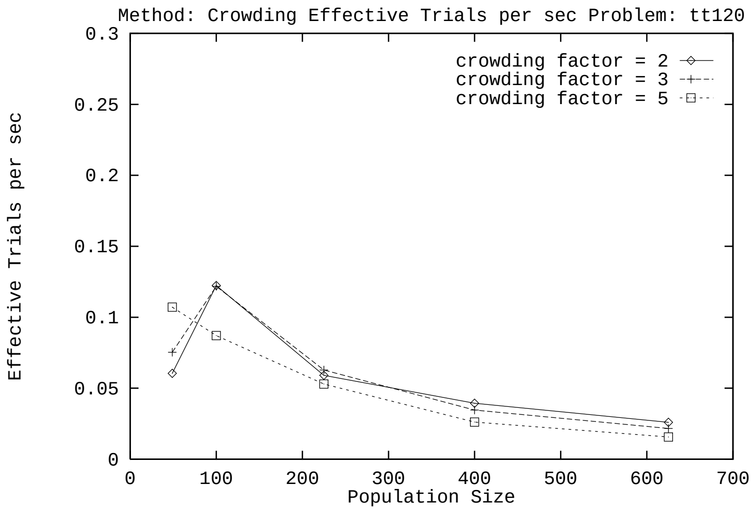

Method: Crowding Diversity Problem: tt60
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity
Population Size
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

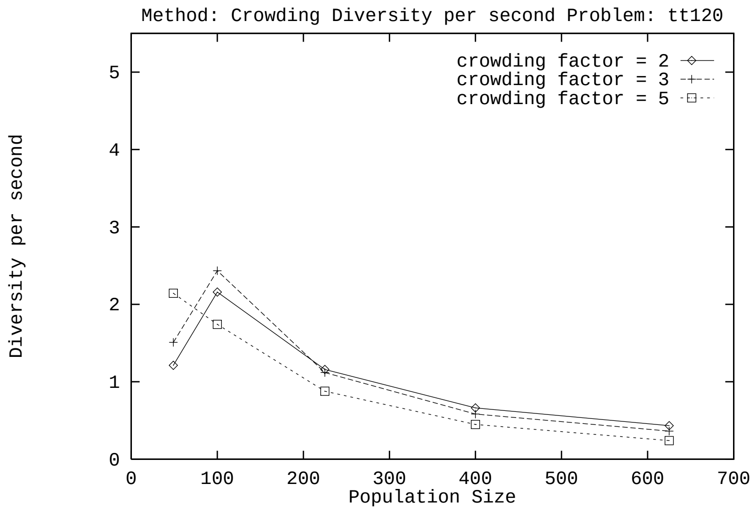

Method: Crowding Effective Trials per sec Problem: tt60
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials per sec
Population Size
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

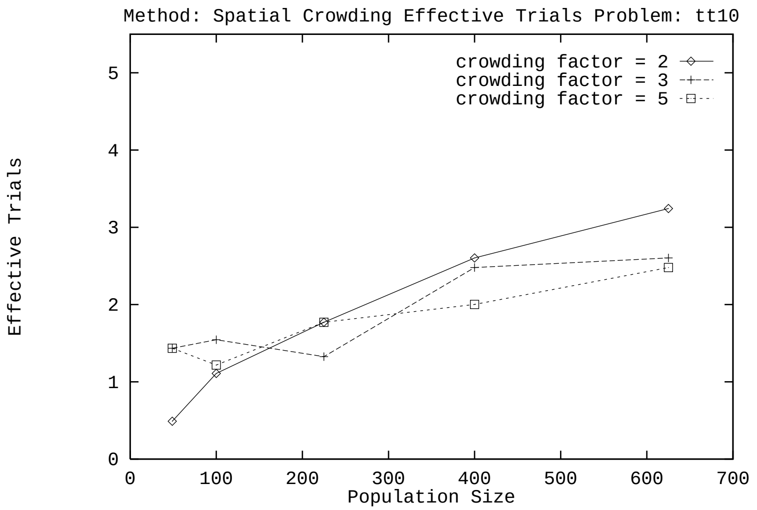

Method: Crowding Diversity per second Problem: tt60
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity per second
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

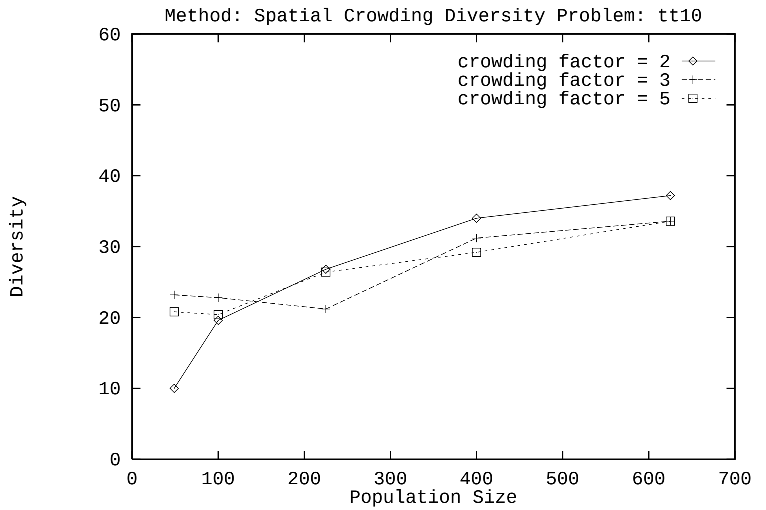

Method: Crowding Effective Trials Problem: tt120
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

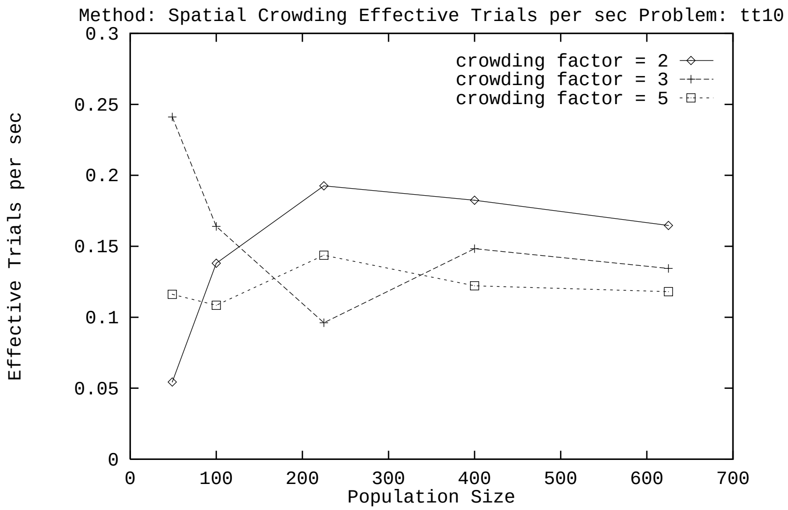

Method: Crowding Diversity Problem: tt120
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity
Population Size
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

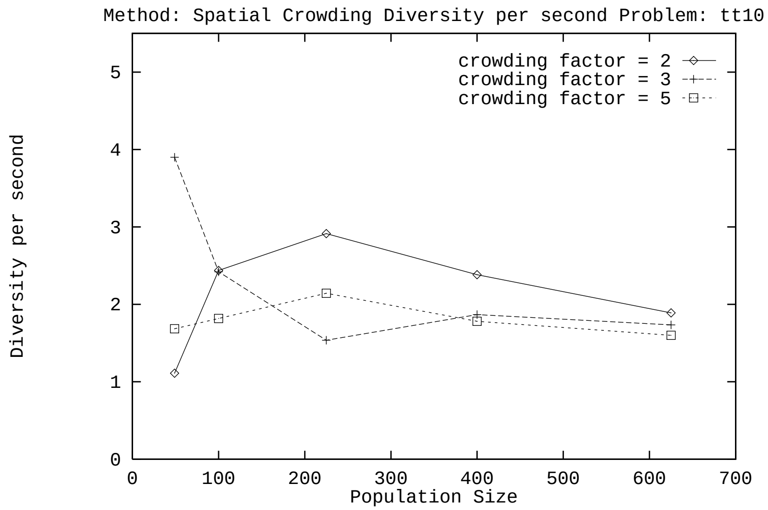

Method: Crowding Effective Trials per sec Problem: tt120
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials per sec
Population Size
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

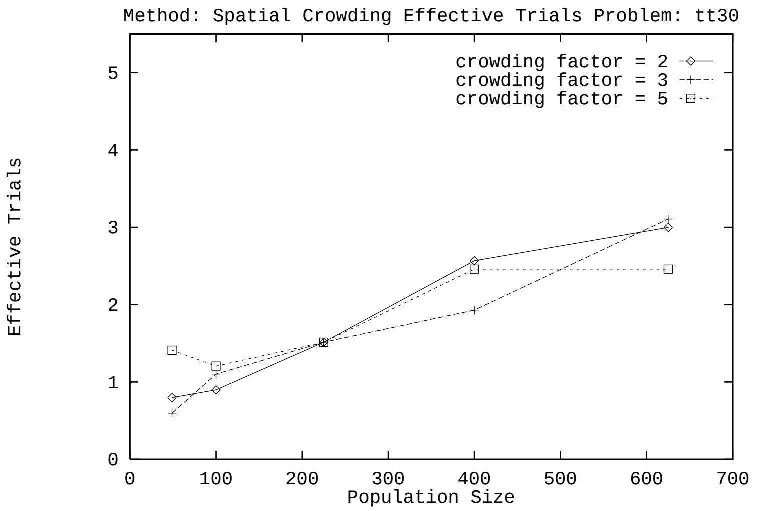

Method: Crowding Diversity per second Problem: tt120
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity per second
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

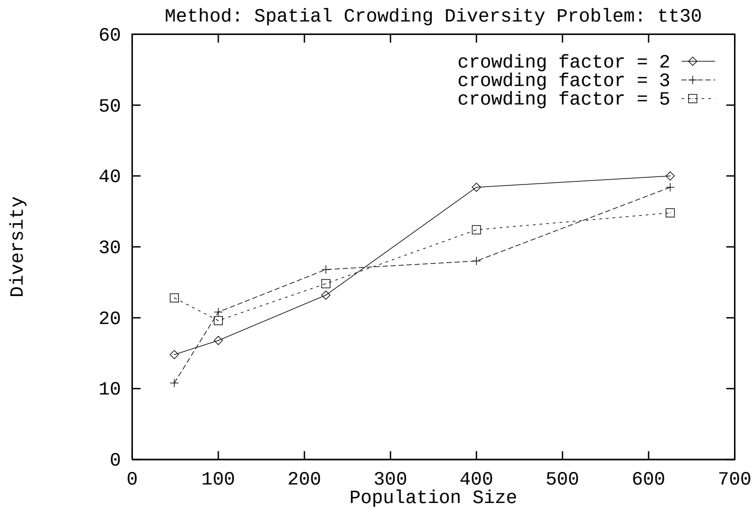

Method: Spatial Crowding Effective Trials Problem: tt10
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

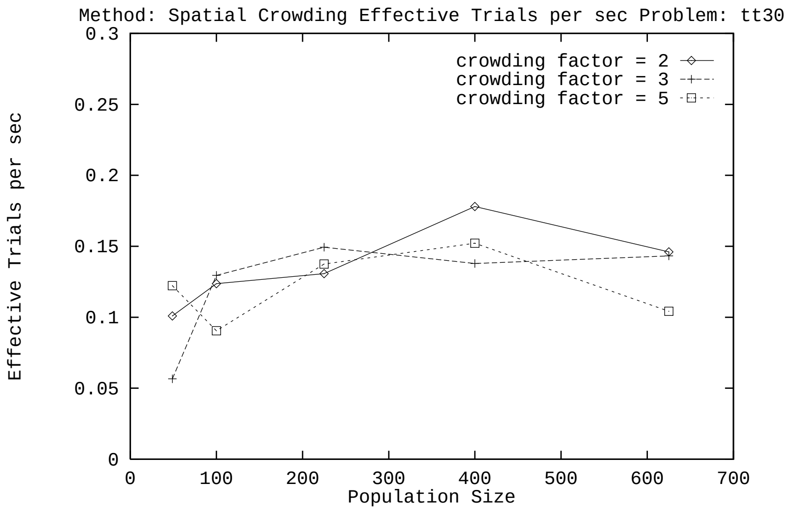

Method: Spatial Crowding Diversity Problem: tt10
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity
Population Size
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

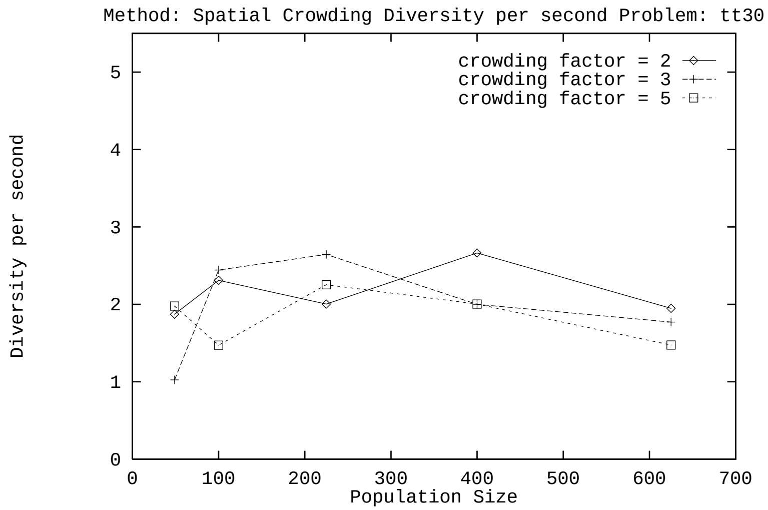

Method: Spatial Crowding Effective Trials per sec Problem: tt10
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials per sec
Population Size
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

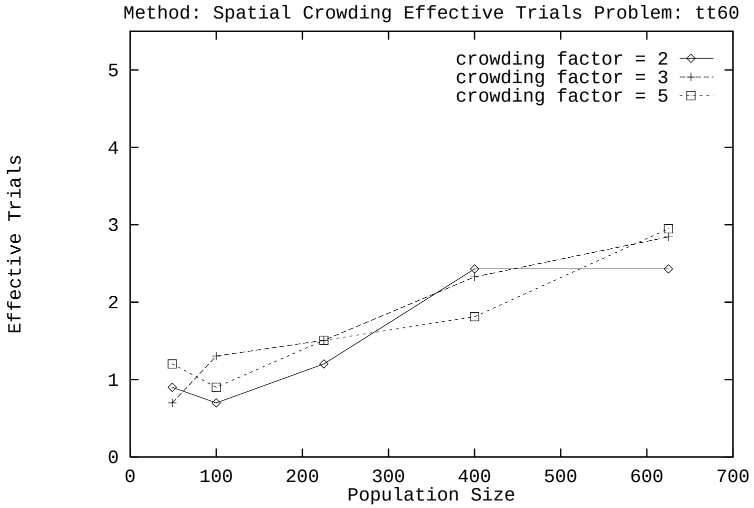

Method: Spatial Crowding Diversity per second Problem: tt10
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity per second
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

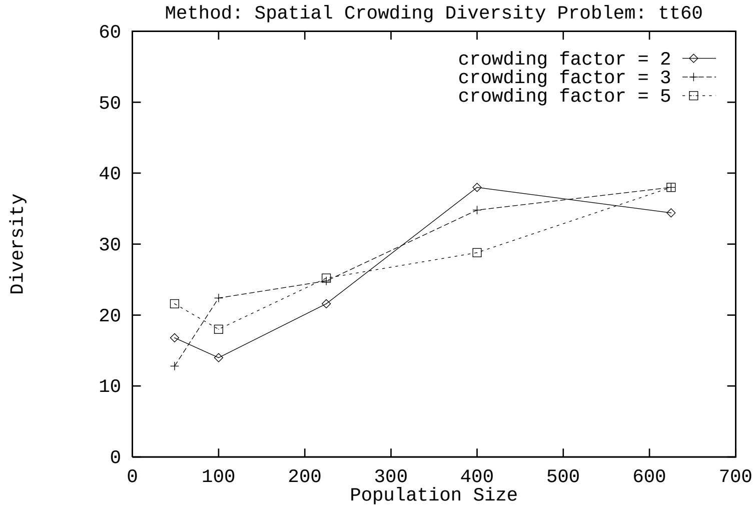

Method: Spatial Crowding Effective Trials Problem: tt30
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

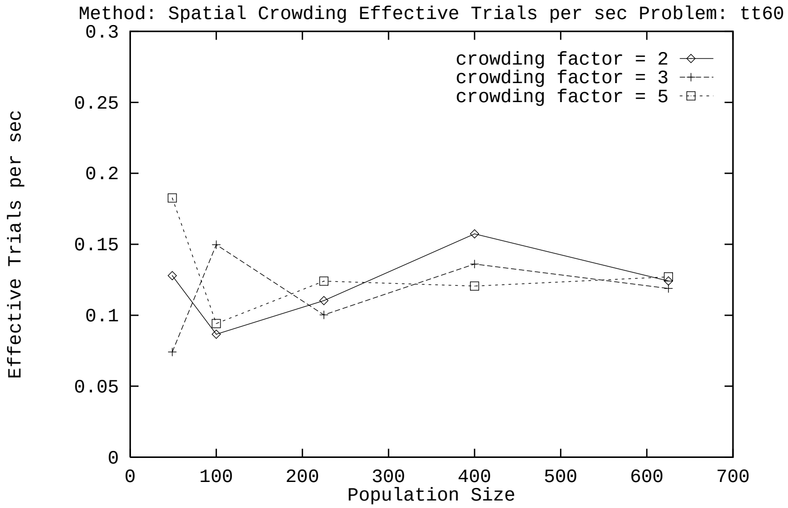

Method: Spatial Crowding Diversity Problem: tt30
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity
Population Size
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

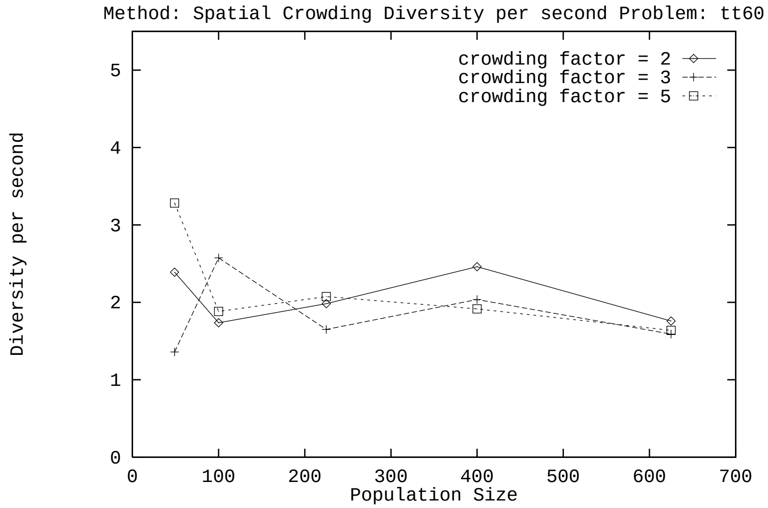

Method: Spatial Crowding Effective Trials per sec Problem: tt30
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials per sec
Population Size
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

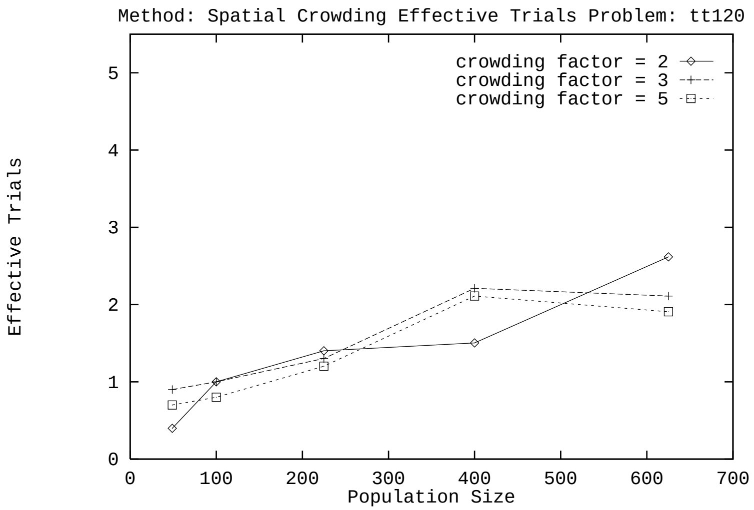

Method: Spatial Crowding Diversity per second Problem: tt30
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity per second
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

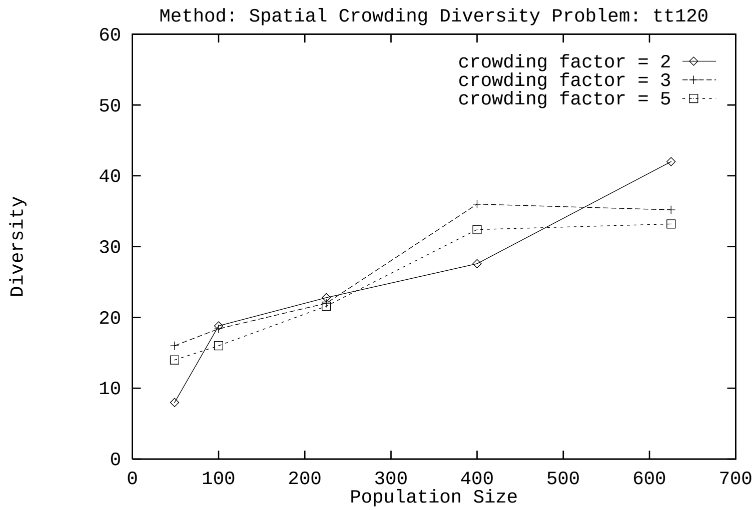

Method: Spatial Crowding Effective Trials Problem: tt60
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

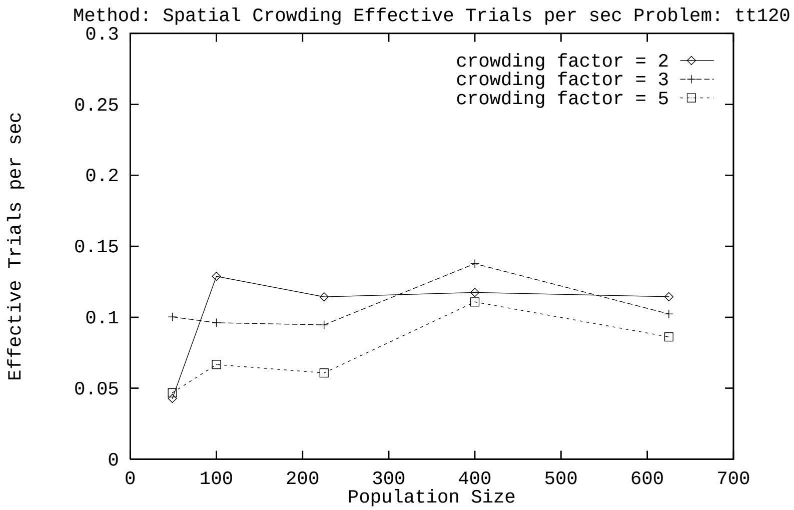

Method: Spatial Crowding Diversity Problem: tt60
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity
Population Size
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

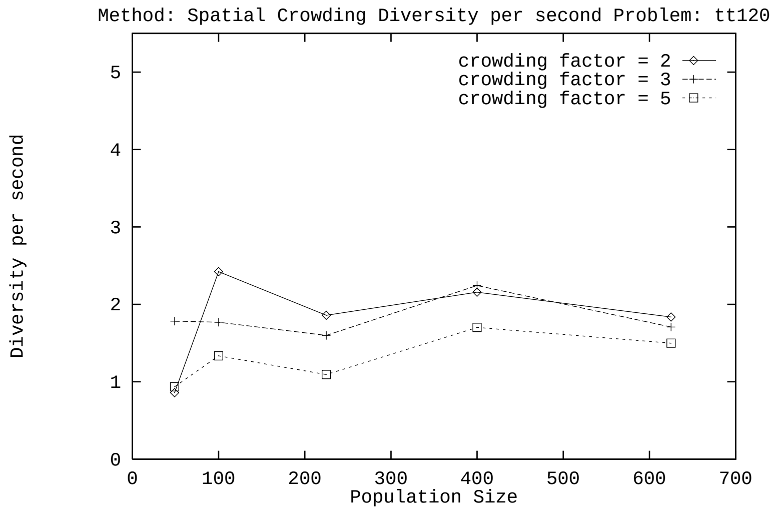

Method: Spatial Crowding Effective Trials per sec Problem: tt60
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials per sec
Population Size
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

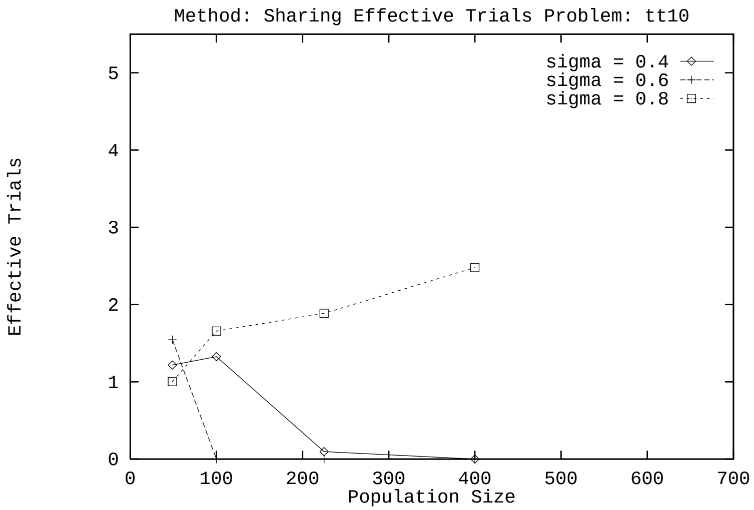

Method: Spatial Crowding Diversity per second Problem: tt60
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity per second
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

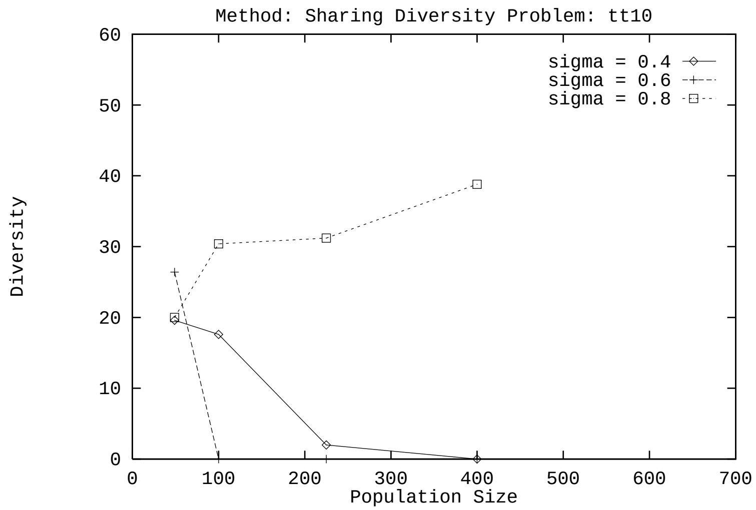

Method: Spatial Crowding Effective Trials Problem: tt120
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials
0
1
2
3
4
5
0
100
200
300
400
500
600
700
Population Size

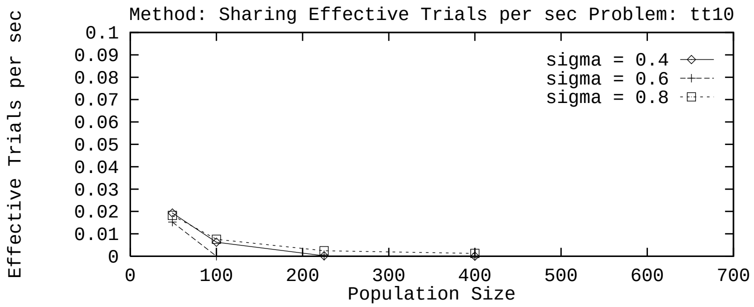

Method: Spatial Crowding Diversity Problem: tt120
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700
Population Size

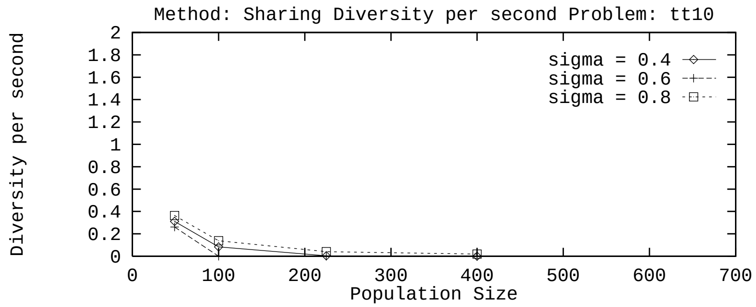

Method: Spatial Crowding Effective Trials per sec Problem: tt120
crowding factor = 2
crowding factor = 3
crowding factor = 5
Effective Trials per sec
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700
Population Size

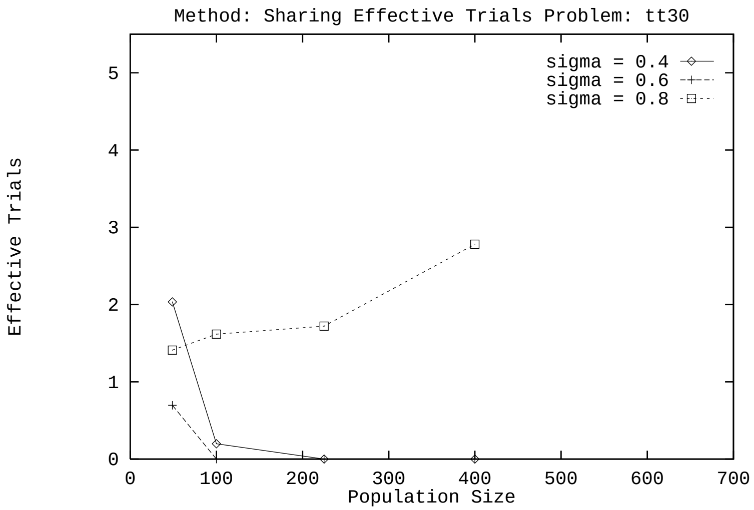

Method: Spatial Crowding Diversity per second Problem: tt120
crowding factor = 2
crowding factor = 3
crowding factor = 5
Diversity per second
0
1
2
3
4
5
0
100
200
300
400
500
600
700
Population Size

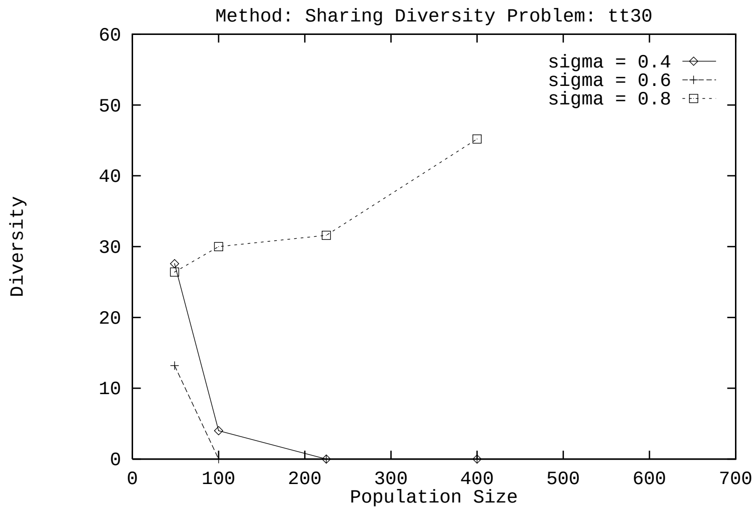

Method: Sharing Effective Trials Problem: tt10
Effective Trials
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

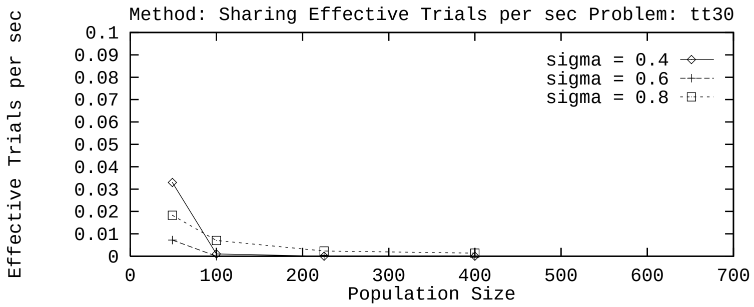

Method: Sharing Diversity Problem: tt10
Diversity
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

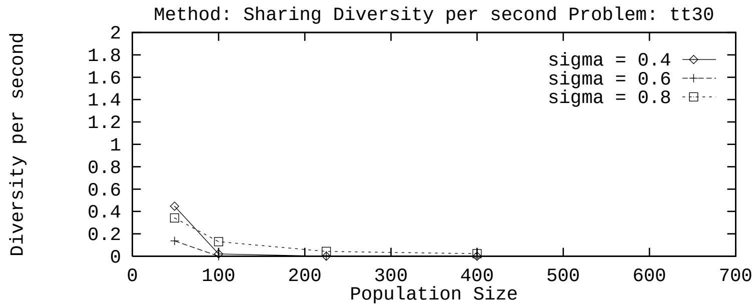

Method: Sharing Effective Trials per sec Problem: tt10
Effective Trials per sec
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

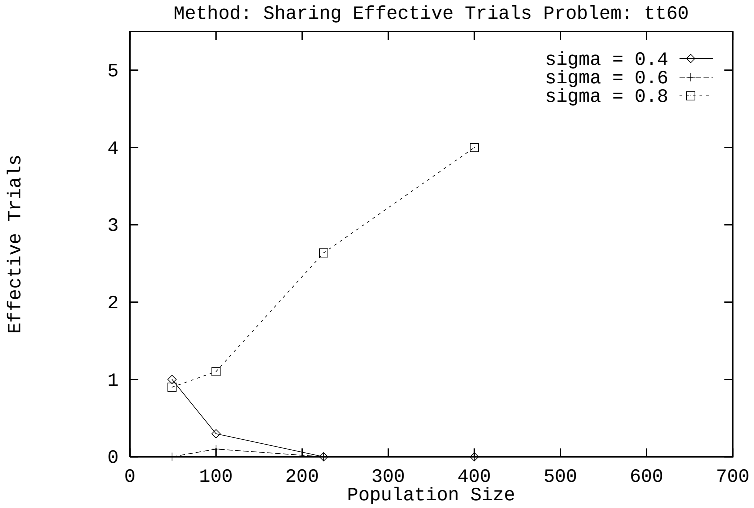

Method: Sharing Diversity per second Problem: tt10
Diversity per second
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

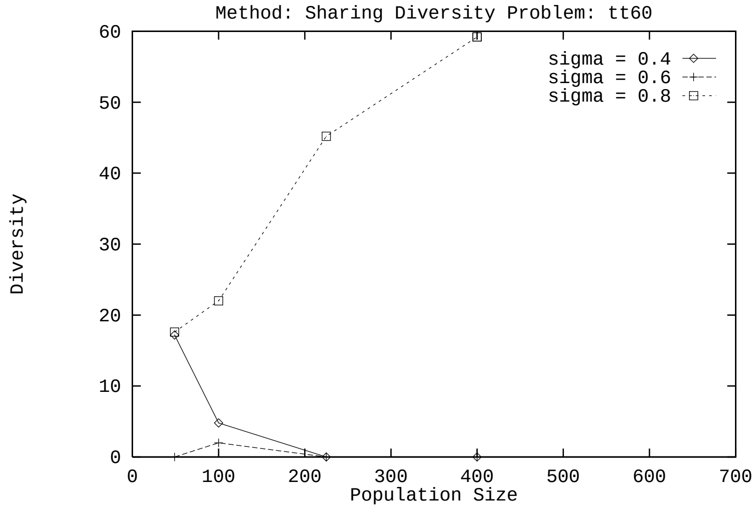

Method: Sharing Effective Trials Problem: tt30
Effective Trials
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
1
2
3
4
5
0
100
200
300
400
500
600
700

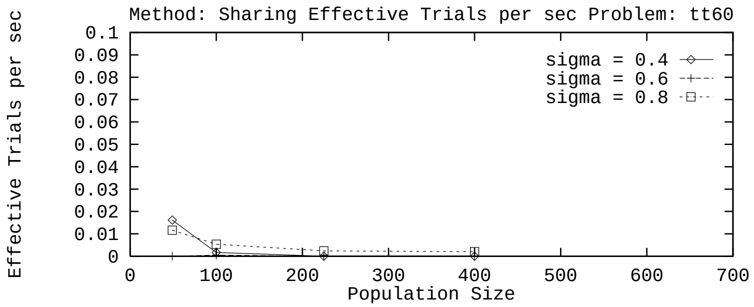

Method: Sharing Diversity Problem: tt30
Diversity
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

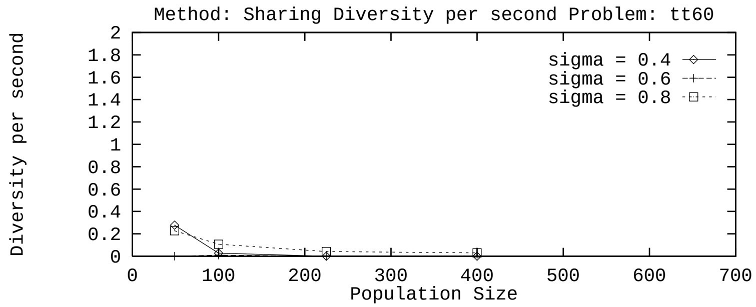

Method: Sharing Effective Trials per sec Problem: tt30
Effective Trials per sec
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0
100
200
300
400
500
600
700

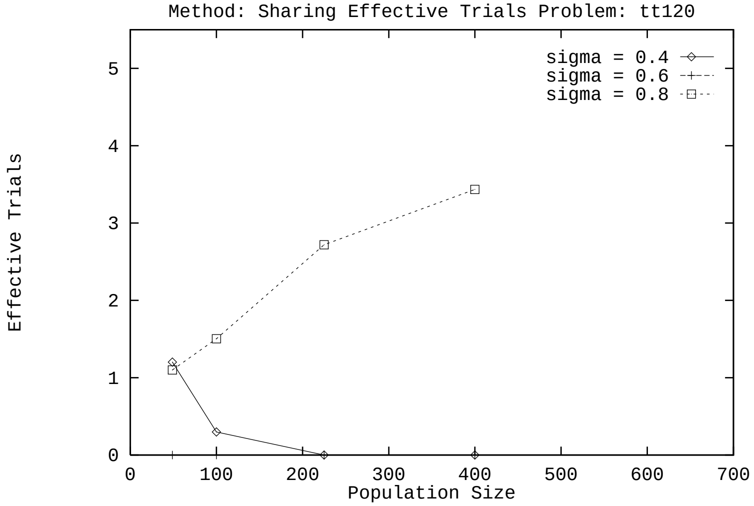

Method: Sharing Diversity per second Problem: tt30
Diversity per second
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
0
100
200
300
400
500
600
700

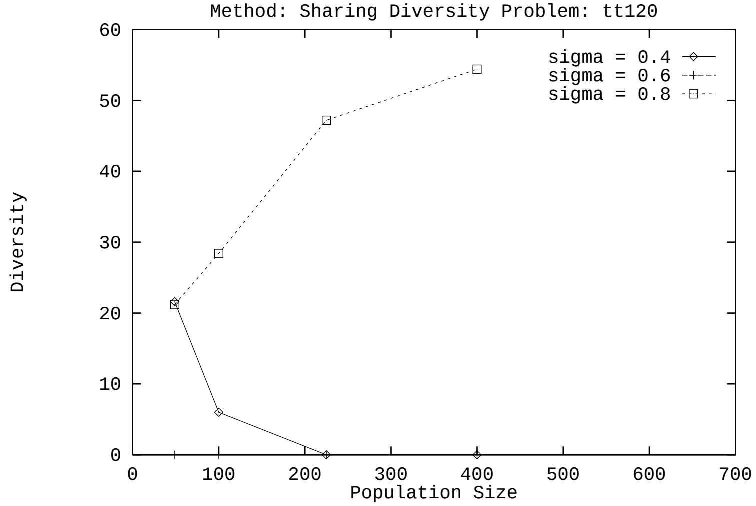

Method: Sharing Effective Trials Problem: tt60
Effective Trials
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
1
2
3
4
5
0
100
200
300
400
500
600
700

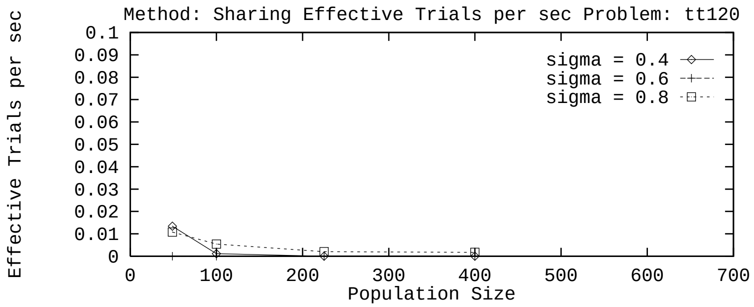

Method: Sharing Diversity Problem: tt60
Diversity
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

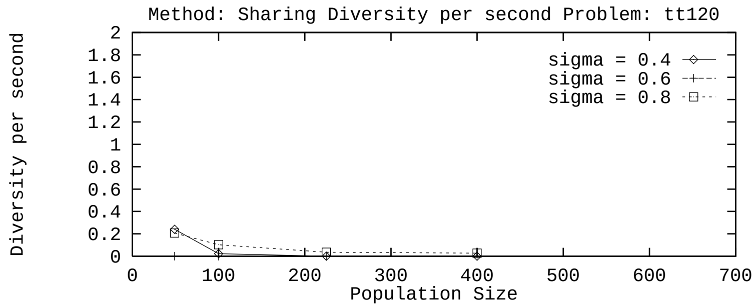

Method: Sharing Effective Trials per sec Problem: tt60
Effective Trials per sec
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0
100
200
300
400
500
600
700

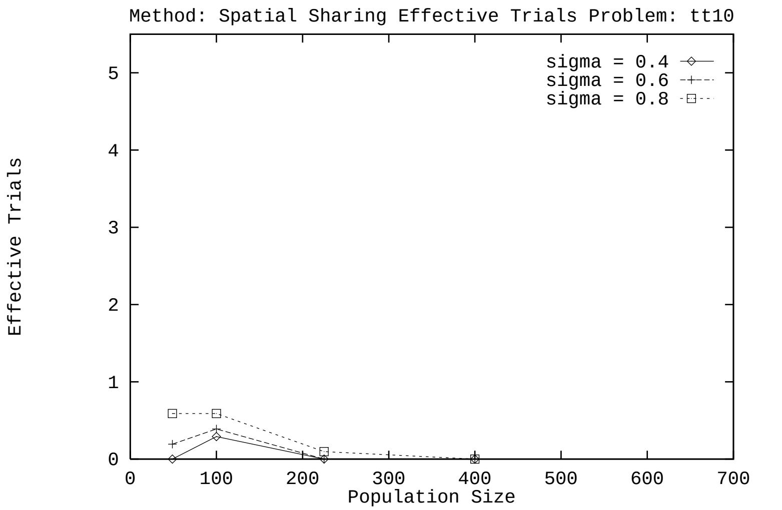

Method: Sharing Diversity per second Problem: tt60
Diversity per second
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
0
100
200
300
400
500
600
700

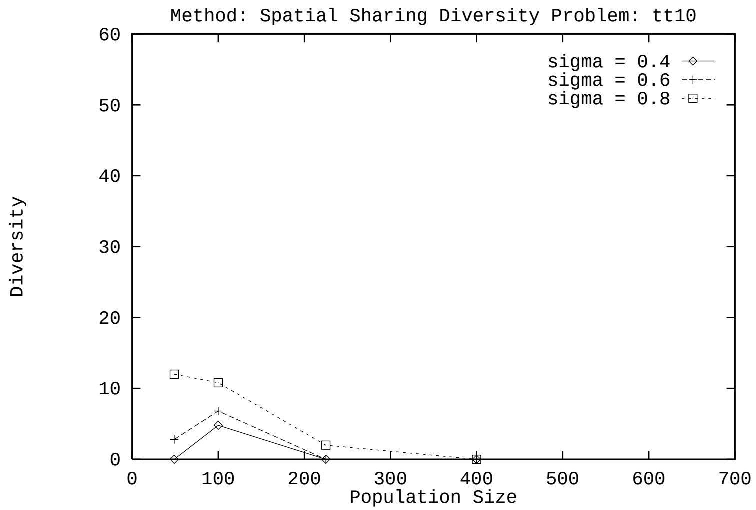

Method: Sharing Effective Trials Problem: tt120
Effective Trials
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

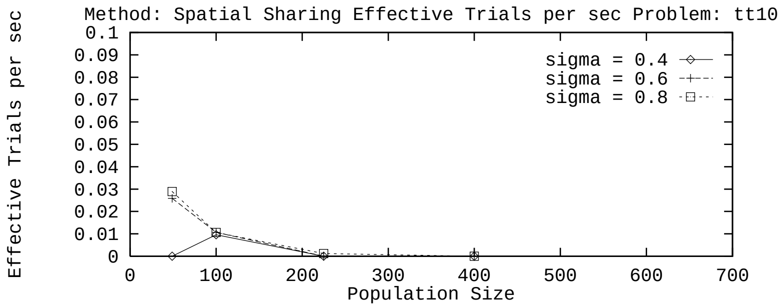

Method: Sharing Diversity Problem: tt120
Diversity
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

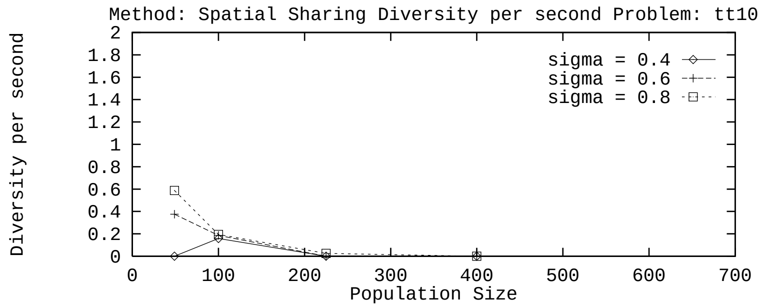

Method: Sharing Effective Trials per sec Problem: tt120
Effective Trials per sec
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

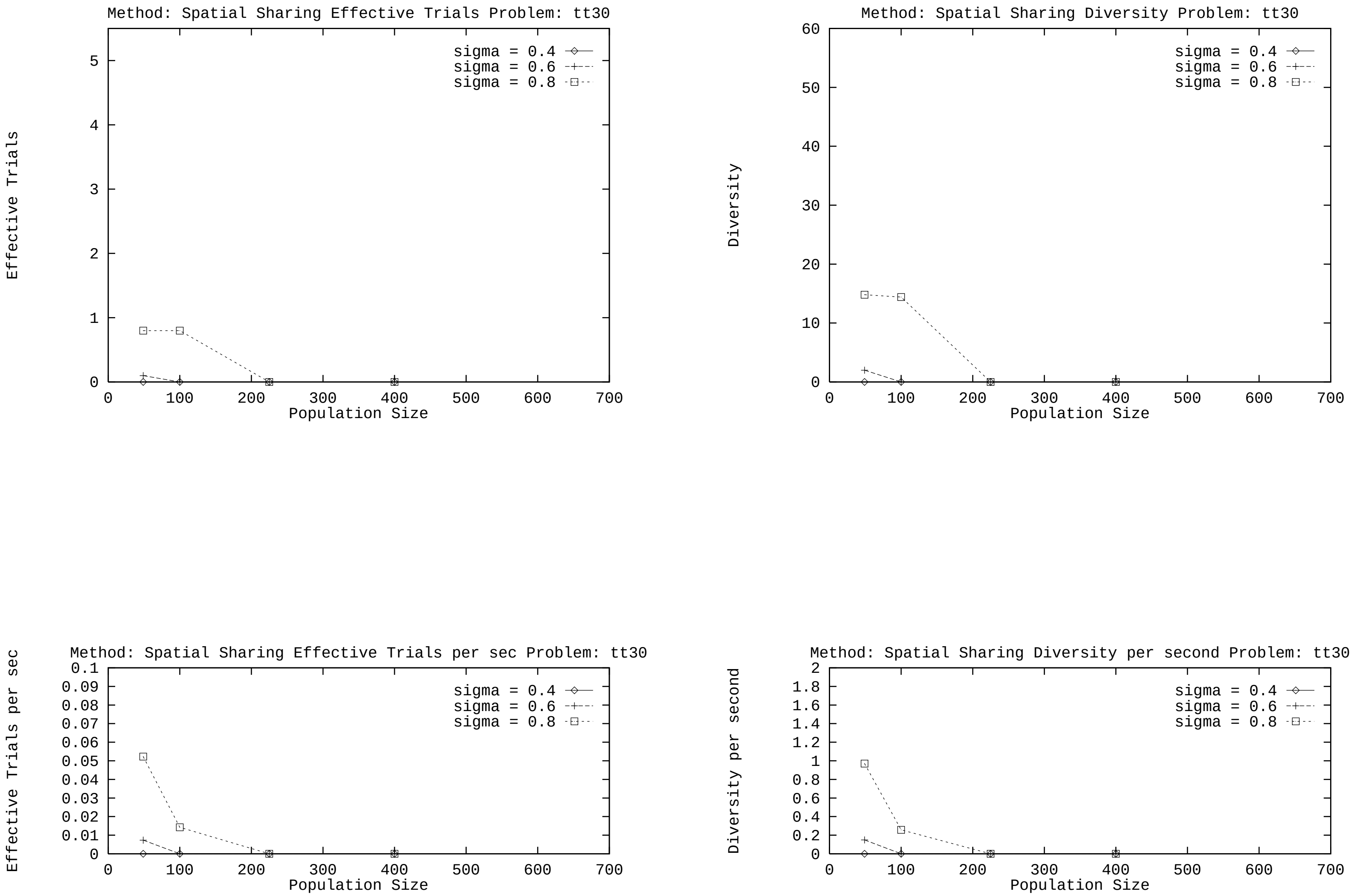

Method: Sharing Diversity per second Problem: tt120
Diversity per second
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8


Method: Spatial Sharing Effective Trials Problem: tt10
Effective Trials
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

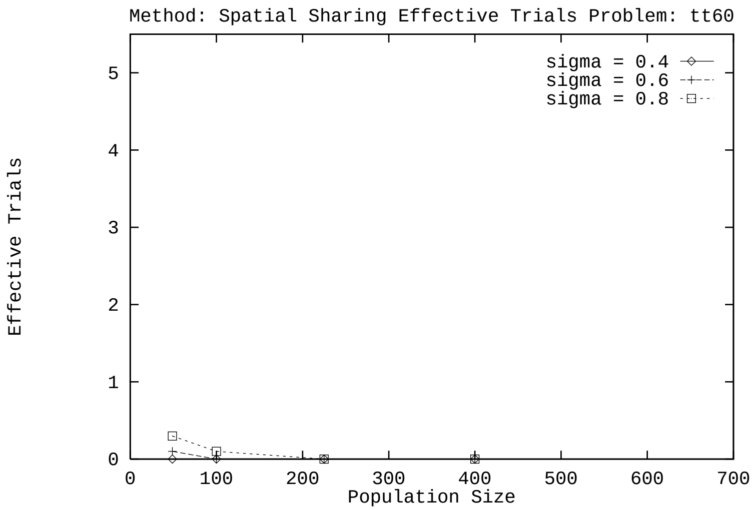

Method: Spatial Sharing Diversity Problem: tt10
Diversity
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

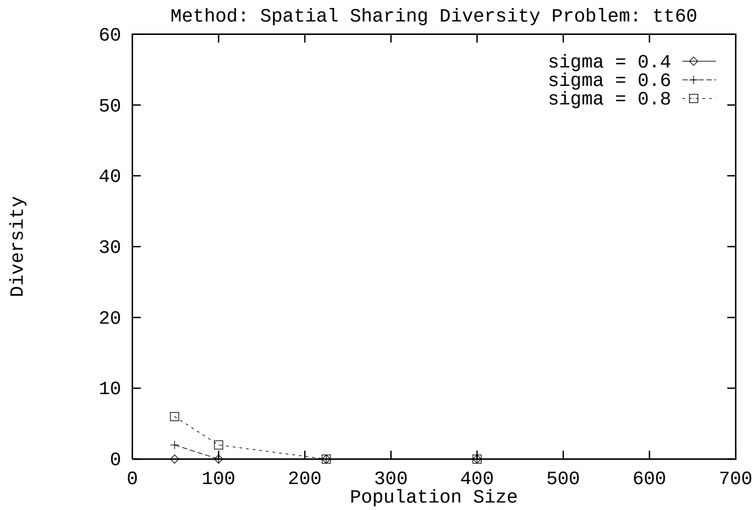

Method: Spatial Sharing Effective Trials per sec Problem: tt10
Effective Trials per sec
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

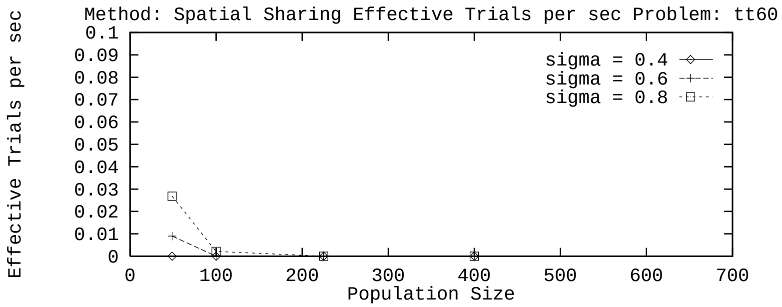

Method: Spatial Sharing Diversity per second Problem: tt10
Diversity per second
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

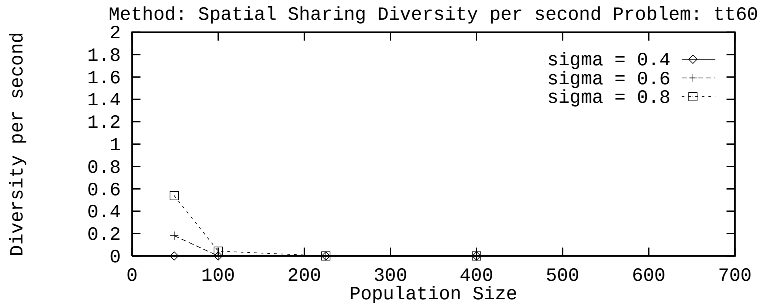


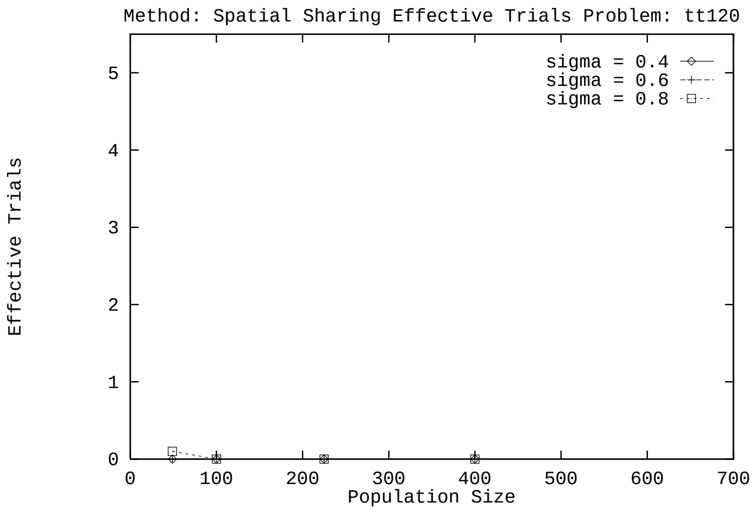
Method: Spatial Sharing Effective Trials Problem: tt30
Effective Trials
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
1
2
3
4
5
100
200
300
400
500
600
700
Method: Spatial Sharing Diversity Problem: tt30
Diversity
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
10
20
30
40
50
60
Method: Spatial Sharing Effective Trials per sec Problem: tt30
Effective Trials per sec
Population Size
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
Method: Spatial Sharing Diversity per second Problem: tt30
Diversity per second
Population Size
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

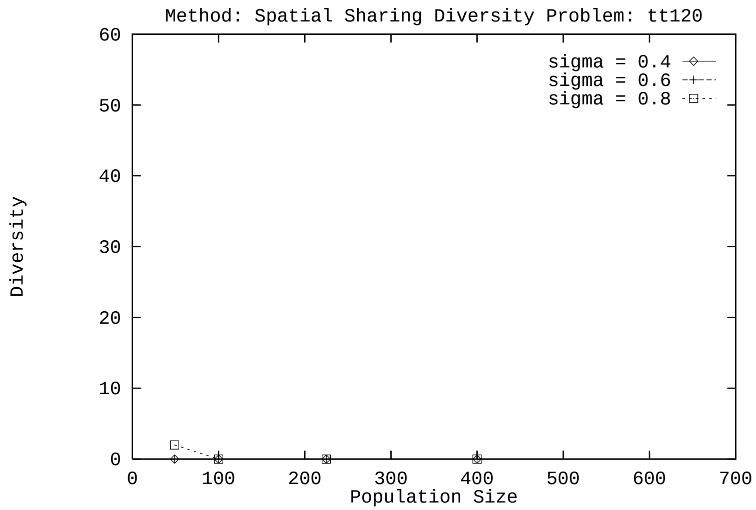

Method: Spatial Sharing Effective Trials Problem: tt60
Effective Trials
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
1
2
3
4
5
0
100
200
300
400
500
600
700

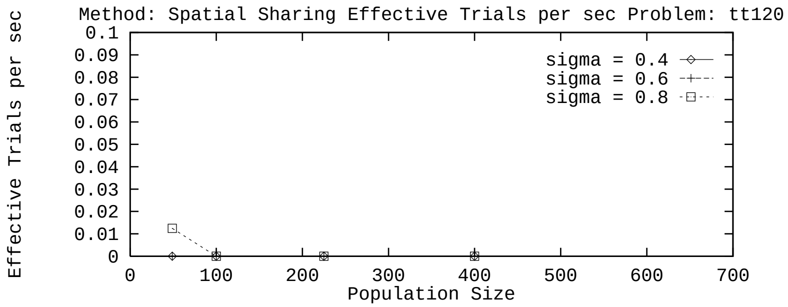

Method: Spatial Sharing Diversity Problem: tt60
Diversity
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700

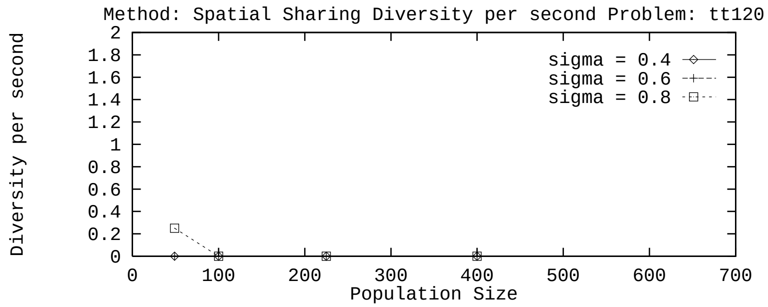

Method: Spatial Sharing Effective Trials per sec Problem: tt60
Effective Trials per sec
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0
100
200
300
400
500
600
700

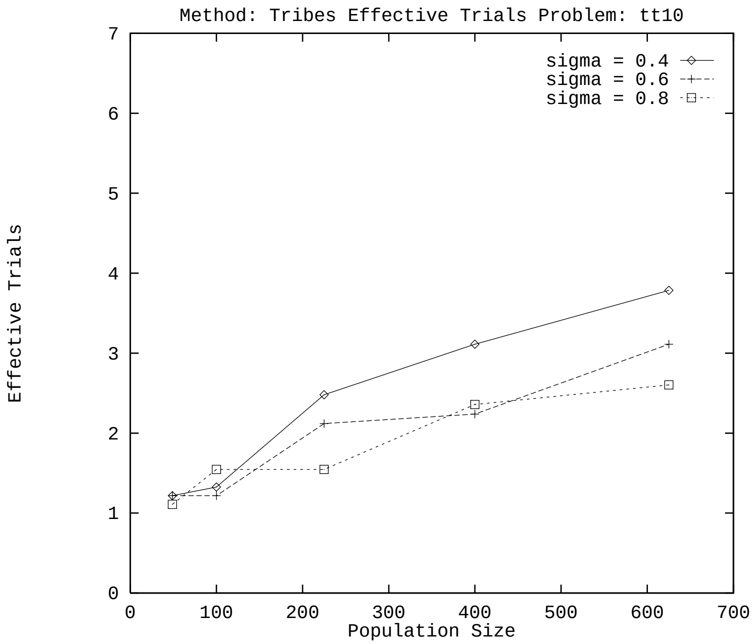

Method: Spatial Sharing Diversity per second Problem: tt60
Diversity per second
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
0
100
200
300
400
500
600
700

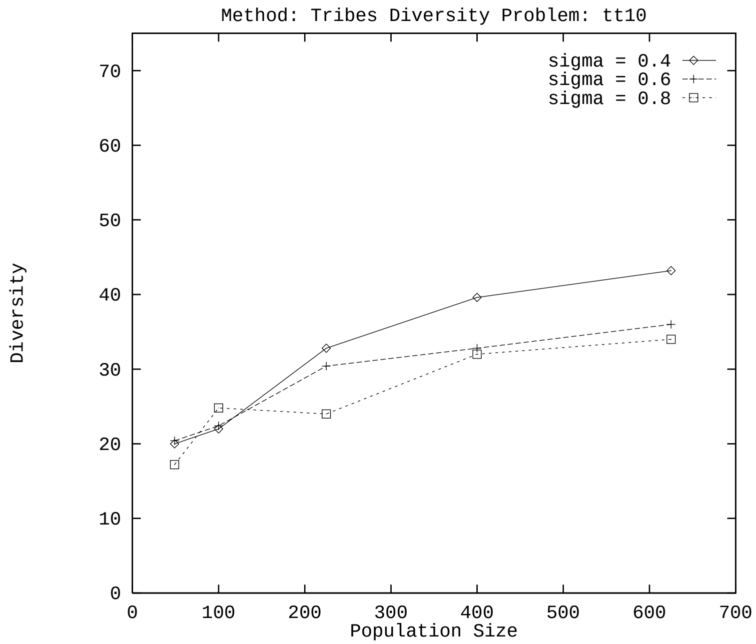

Method: Spatial Sharing Effective Trials Problem: tt120
sigma = 0.4
sigma = 0.6
sigma = 0.8
Effective Trials
Population Size

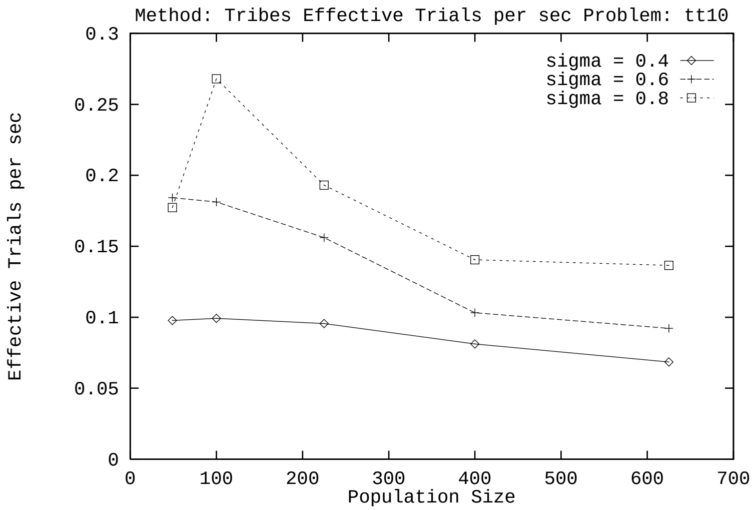

Method: Spatial Sharing Diversity Problem: tt120
sigma = 0.4
sigma = 0.6
sigma = 0.8
Diversity
Population Size

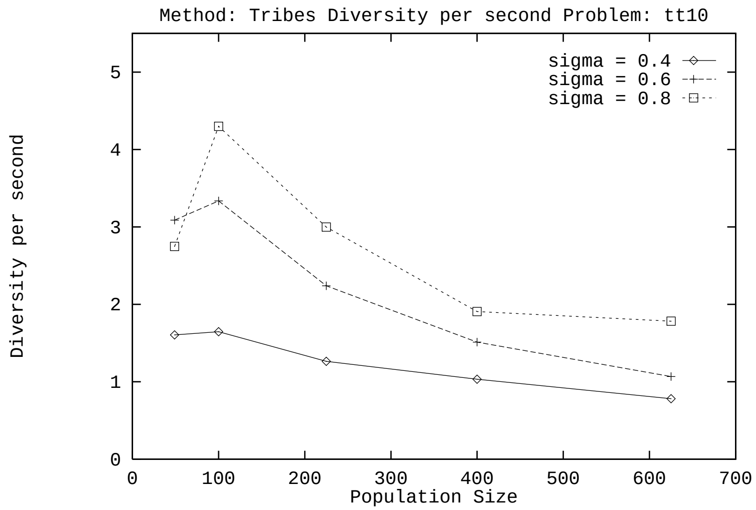

Method: Spatial Sharing Effective Trials per sec Problem: tt120
sigma = 0.4
sigma = 0.6
sigma = 0.8
Effective Trials per sec
Population Size

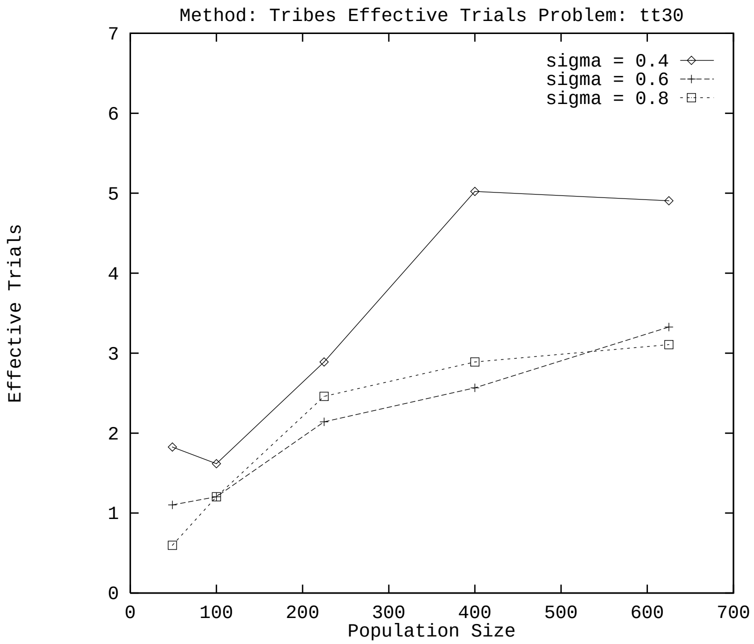

Method: Spatial Sharing Diversity per second Problem: tt120
sigma = 0.4
sigma = 0.6
sigma = 0.8
Diversity per second
Population Size

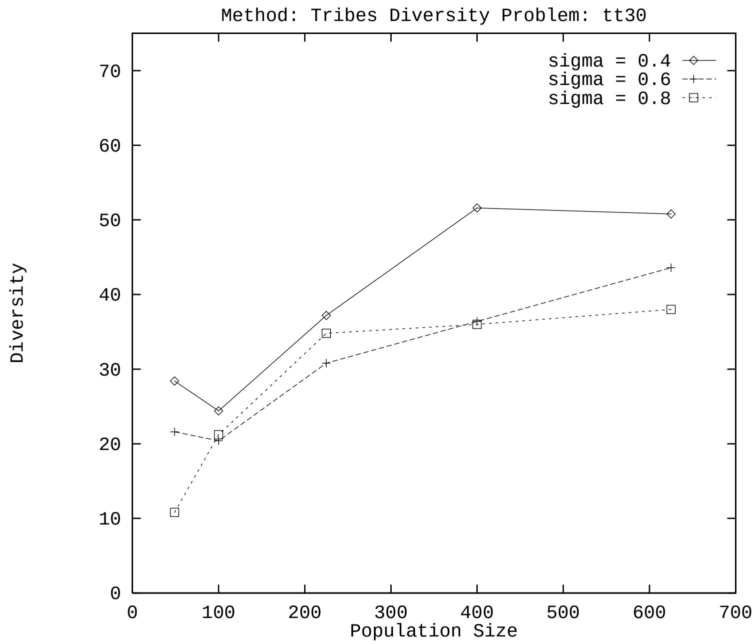

Method: Tribes Effective Trials Problem: tt10
Effective Trials
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
1
2
3
4
5
6
7
0
100
200
300
400
500
600
700

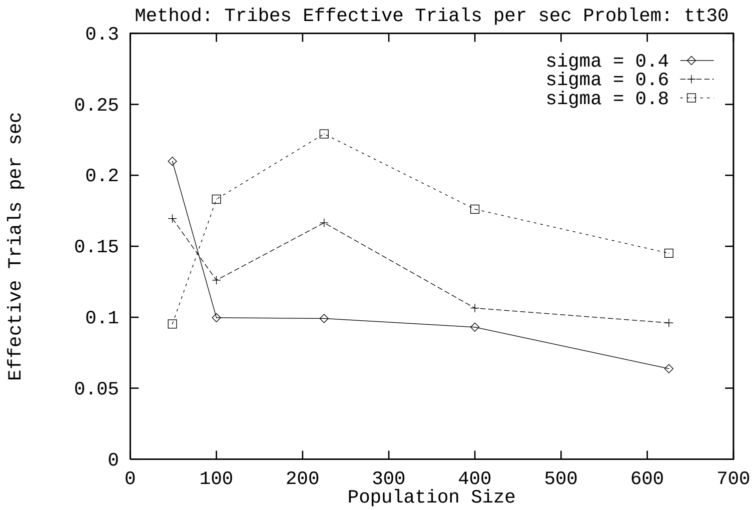

Method: Tribes Diversity Problem: tt10
Diversity
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
10
20
30
40
50
60
70
0
100
200
300
400
500
600
700

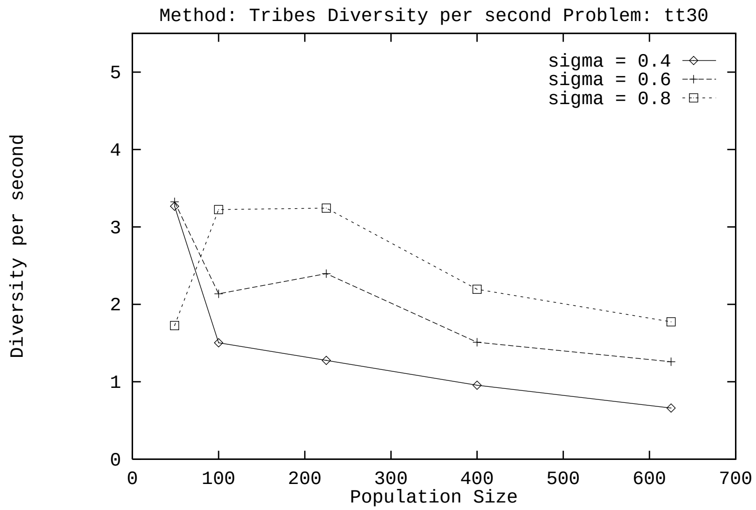

Method: Tribes Effective Trials per sec Problem: tt10
Effective Trials per sec
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

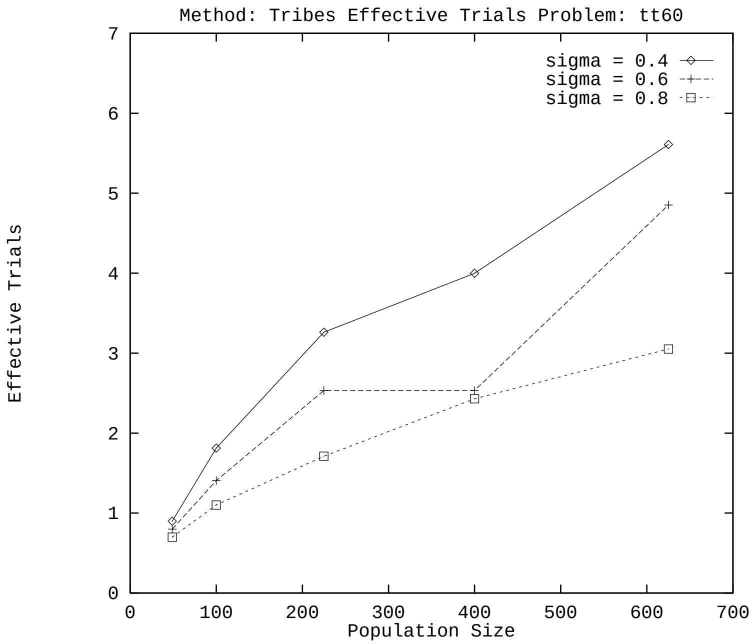

Method: Tribes Diversity per second Problem: tt10
Diversity per second
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8
0
1
2
3
4
5
0
100
200
300
400
500
600
700

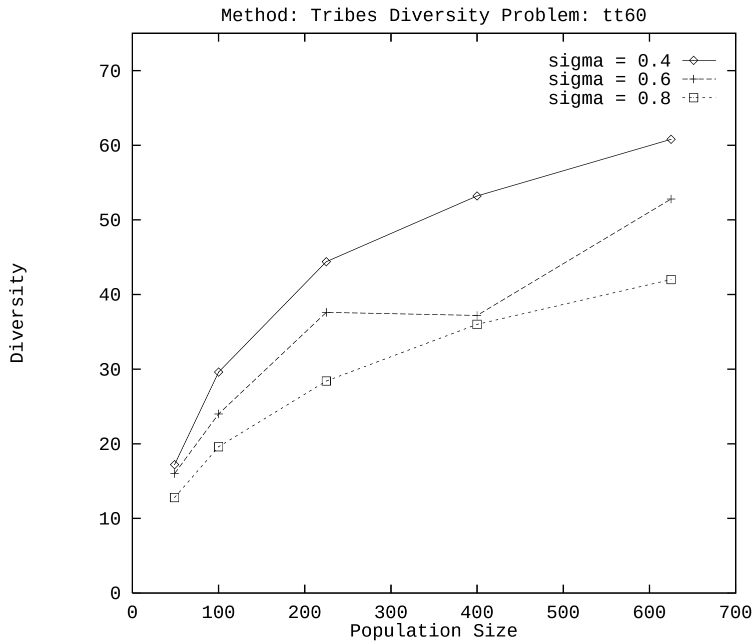

Method: Tribes Effective Trials Problem: tt30
sigma = 0.4
sigma = 0.6
sigma = 0.8
Effective Trials
Population Size
0
1
2
3
4
5
6
7
0
100
200
300
400
500
600
700

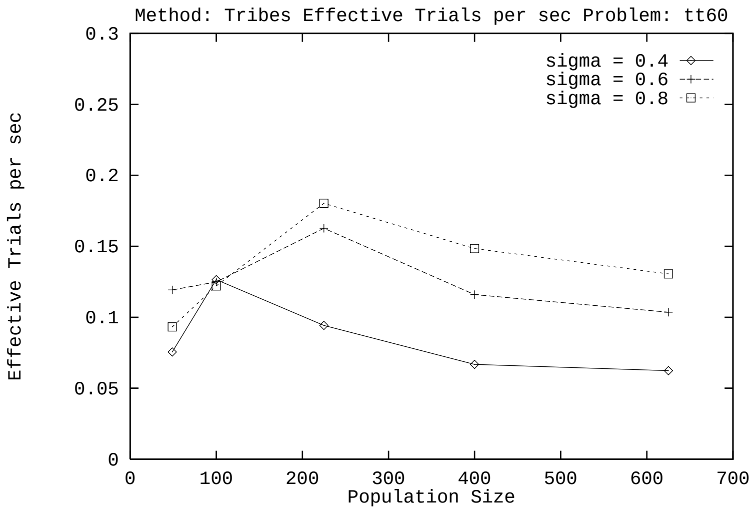

Method: Tribes Diversity Problem: tt30
sigma = 0.4
sigma = 0.6
sigma = 0.8
Diversity
Population Size
0
10
20
30
40
50
60
70
0
100
200
300
400
500
600
700

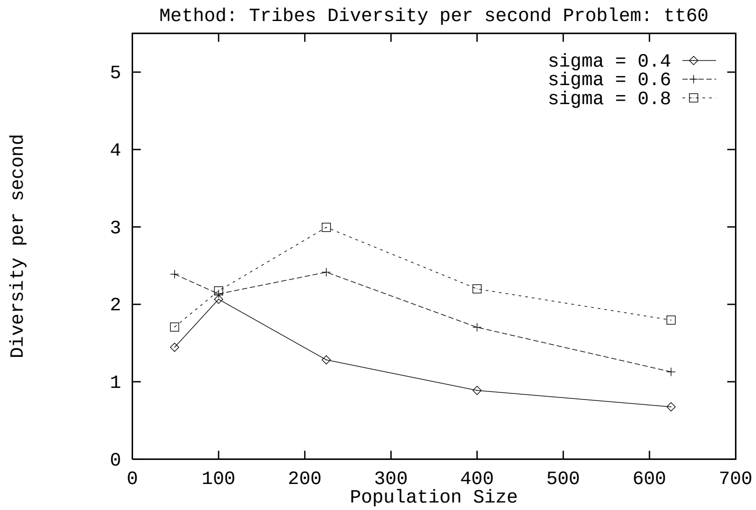

Method: Tribes Effective Trials per sec Problem: tt30
sigma = 0.4
sigma = 0.6
sigma = 0.8
Effective Trials per sec
Population Size
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700

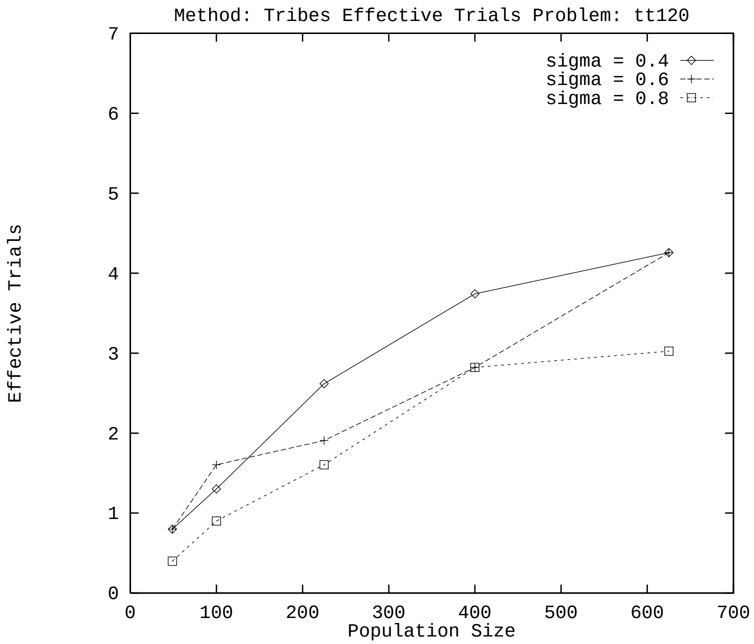

Method: Tribes Diversity per second Problem: tt30
sigma = 0.4
sigma = 0.6
sigma = 0.8
Diversity per second
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700

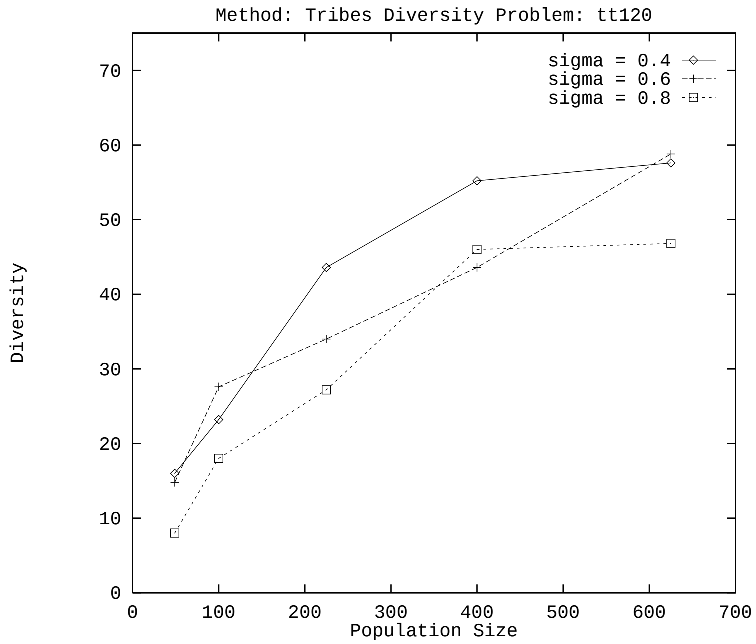

Method: Tribes Effective Trials Problem: tt60
sigma = 0.4
sigma = 0.6
sigma = 0.8
Effective Trials
Population Size
0
1
2
3
4
5
6
7
0
100
200
300
400
500
600
700

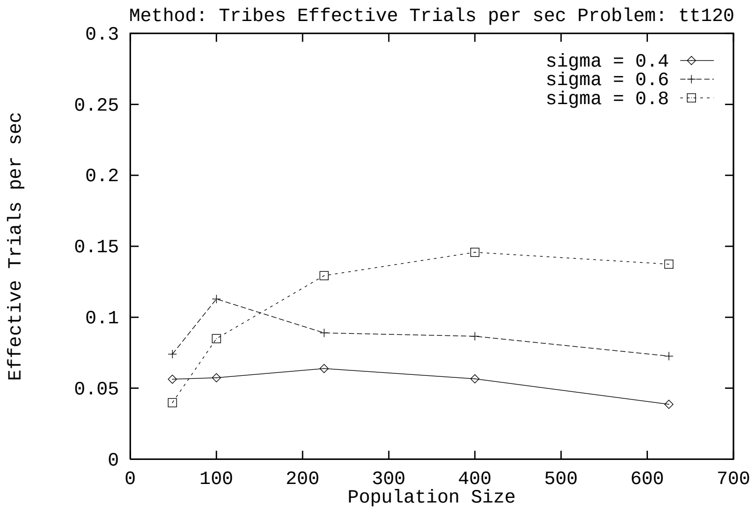

Method: Tribes Diversity Problem: tt60
sigma = 0.4
sigma = 0.6
sigma = 0.8
Diversity
Population Size
0
10
20
30
40
50
60
70
0
100
200
300
400
500
600
700

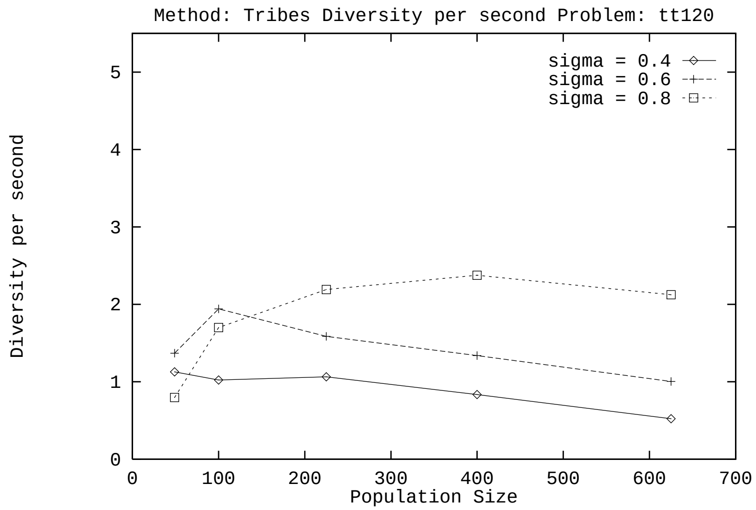

Method: Tribes Effective Trials per sec Problem: tt60
sigma = 0.4
sigma = 0.6
sigma = 0.8
Effective Trials per sec
Population Size
0
0.05
0.1
0.15
0.2
0.25
0.3
0
100
200
300
400
500
600
700


Method: Tribes Diversity per second Problem: tt60
sigma = 0.4
sigma = 0.6
sigma = 0.8
Diversity per second
Population Size
0
1
2
3
4
5
0
100
200
300
400
500
600
700


Method: Tribes Effective Trials Problem: tt120
Effective Trials
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8


Method: Tribes Diversity Problem: tt120
Diversity
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8


Method: Tribes Effective Trials per sec Problem: tt120
Effective Trials per sec
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8


Method: Tribes Diversity per second Problem: tt120
Diversity per second
Population Size
sigma = 0.4
sigma = 0.6
sigma = 0.8

Appendix B

Modifications to PGA

B.1 Introduction

This document outlines the changes made to PGA (version 2.7) for Multiple Distinct Solutions project work. It is designed to be used in conjunction with the document *About PGA v2.5* by Peter Ross and also the README file supplied with version 2.7 of the PGA code. Some knowledge of genetic algorithms (GAs) is assumed.

The modifications are:

- a new GA convergence test.
- additional reporting facilities.
- an implementation of crowding (similar to the method used in [De Jong 75]).
- an implementation of sharing (as used in [Deb & Goldberg 89]).
- an implementation of tribes

B.2 GA Convergence

Command Line Flag: -X<n>

<n> Number of static generations before convergence is declared.
(integer <n> and <n> ≥ 1, <n> defaults to the reporting stage interval.)

B.2.1 General Description

The original convergence tests in PGA differed according to whether PGA was run interactively or non-interactively, and the non-interactive convergence tests were not suitable for finding multiple distinct solutions. The convergence tests have been replaced by a single convergence test: when the average fitness of the population has not increased above its maximum for n generations (which can be set with the `-X` flag), then the population is considered to have converged.

B.2.2 Technical Details

The average population fitness is calculated at each reporting interval (the reporting interval is set with the `-i` flag and defaults to 10 generations). If this exceeds the previous maximum average fitness of the population then the maximum average fitness of the population is replaced and a counter, the *GAstatic* count, is reset to zero. The *GAstatic* count is incremented every generation, and if this reaches the maximum number of static generations, the population is flagged as converged. When all the populations have converged then the PGA run is completed. The maximum number of static generations may be set with the `-X` flag and defaults to the reporting stage interval. (The reporting stage interval is set with the `-l` flag — see *About PGA v2.5* for more details.)

B.3 Additional Reporting to File

B.3.1 General Description

In addition to the reporting to file in PGA v2.7, the modified version of PGA reports the following every reporting stage interval (set with the `-l` flag):

- The fittest genotypes in the population
- The fitness of the these genotypes
- The number of these genotypes in the population
- The allele count diversity these genotypes in the population
- The time of the run so far

B.3.2 Technical Details

The current fittest genotypes in the population are those with fitness within 5×10^{-9} of the fittest genotype found in the population. All are reported. The fitness of this group of genotypes is the fitness of the fittest genotype in the population.

The number of genotypes in this group is simply how many genotypes there are in the group. The allele count diversity is the allele count diversity (as defined in chapter 2.2.2 of this thesis).

The time of the run so far is the time of the initialisation plus the time of applying genetic operators for at each generation.

B.4 Crowding

Command Line Flag: `-D<opt><n>`

`<opt> = a` Replace chromosome regardless of fitness

`<opt> = b` Replace chromosome only if child is fitter

`<n>` Crowding Factor (CF)
(integer `<n>` and `<n>` ≥ 1)

B.4.1 General Description

Two methods of crowding are employed: standard crowding and spatial crowding. If a non-spatial selection operator is being used then standard crowding is used, otherwise spatial crowding is used.

Standard crowding follows a method similar to that employed by De Jong in [De Jong 75]. Crowding Factor (CF) chromosomes are selected from the population and the one which is most similar to the child is replaced.

Spatial selection in PGA follows [Collins & Jefferson 91]: a chromosome at some location on a 2-D plane is chosen as a 'nominated' chromosome and two random walks from this chromosome location determine the parents. Usually the child replaces the nominated chromosome. However, in the spatial crowding method a random walk of CF length is taken from the nominated chromosome location, and the most similar chromosome to the child encountered on the walk is replaced.

B.4.2 Technical Details

`<opt>` either `a` or `b`. `a` results in 'crowded' chromosome being replaced regardless of fitness of child chromosome. `b` results in 'crowded' chromosome being replaced only if the $fitness(child) > fitness(crowded)$.

`<n>` (integer `<n>` and `<n>` ≥ 1) is the CF.

If `-rone` is selected then CF chromosomes are picked from the current population (one at a time, with replacement) and the chromosome with the lowest

Hamming distance is chosen as the 'crowded' chromosome. If more than one chromosome is found with the same hamming distance then the first one chosen is preferred. The crowded chromosome is replaced by the child according to the value of `<opt>`.

If `-rssone` is selected then CF chromosomes are chosen by random walk from the current location (that used for selection of the parents). The random walk type used is the same as that used for selection of the parents. Hamming distance comparison is the same as with `-rone`. If more than one chromosome is found with the same hamming distance then the first one found on the random walk is preferred.

B.5 Sharing

Command Line Flag: `-E<opt><n>`

`<opt> = a` Standard sharing method

`<opt> = b` Spatial sharing method

`<n>` Niche separation distance (σ_{share})
(real `<n>` and $0 <$ `<n>` < 1)

B.5.1 General Description

The method for the standard sharing method is taken from [Deb & Goldberg 89]. Rather than using the absolute fitness for a chromosome, a shared fitness is used. The shared fitness takes into account chromosomes with similar genotypes or phenotypes. To calculate the shared fitness a sharing function is used. The sharing function gives the degree of overlap of two chromosomes in terms of whether the occupy separate *niches*. This is given by the formula:

$$Sh(x_i, x_j) = \begin{cases} 1 - \left(\frac{d_x(x_i,x_j)}{\sigma_{share}}\right)^{\alpha} & \text{if } d_x(x_i, x_j) < \sigma_{share} \\ 0 & \text{otherwise} \end{cases}$$

$d(x_i, x_j)$ is the distance function between two chromosomes. In its simplest form this is the Hamming distance. σ_{share} is the distance between niches (the *niche separation distance*) — this is chosen with the value of `<n>`. *alpha* is an arbitrary power not implemented as yet.

For each chromosome the shared fitness (f') is simply the standard fitness over the sum of the sharing functions for all the chromosomes in the population:

$$f'(x_i) = f(x_i) / \sum_{j=1}^{N} Sh(x_i, x_j)$$

There is no reason why this method should not be used with spatial selection; however, the amount of calculation can be significantly reduced when using spatial selection if only the chromosomes in reach (through random walk) of the 'nominated' chromosome are considered in the sum of sharing functions.

B.5.2 Technical Details

`<opt>` either `a` or `b`. `a` is standard sharing. Note that there is no reason why this method should not be used with spatial selection. `b` is a spatial sharing method.

`<n>` (real `<n>` and `<n>` > 0) is the niche separation distance σ_{share}.

At present, genotype sharing (i.e. d_x = Hamming distance) is used for all the problems.

In standard sharing, the fitness of each chromosome is defined as above:

$$f'(x_i) = f(x_i) / \sum_{j=1}^{N} Sh(x_i, x_j)$$

In spatial sharing, if the location of a chromosome is (a, b) and R is the length of the random walk, then the fitness of that chromosome is defined as:

$$f(x_{a,b})' = f(x_{a,b}) / \sum_{i=a-R}^{a+R} \sum_{j=b-R}^{b+R} Sh(x_{a,b}, x_{i,j})$$

The other details for both methods, as follow, are the same.

When the populations are initialised the shared fitness is calculated for every chromosome in the population. At each generation, when a potential child has been generated and a potential chromosome to be replaced been chosen, a trial shared fitness is calculated for the child (at the same time, potential updated shared fitnesses are calculated for the other members of the population). This trial shared fitness is compared with the shared fitness of the chromosome to be replaced. If the trial shared fitness of the child is greater than the shared fitness of the chromosome to be replaced, then the chromosome to be replaced is replaced (and the other chromosomes in the population updated to reflect this change):

$$\text{If} \frac{f_{new}}{\sum Sh_{new}^{trial}} > \frac{f_{old}}{\sum Sh_{old}^{original}} \text{then replace } C_{old} \text{ with } C_{new}$$

B.6 Tribes

Command Line Flag: `-Y<n>`

`<n>` Cutoff distance (d_c)
(real `<n>` and 0 < `<n>` < 1)

B.6.1 General Description

Tribes is described in this thesis, chapter 6.2. Tribes is currently only implemented with spatial selection. If tribes is defined and the child chromosome is further than d_c (a Hamming distance) from the chromosome it tries to replace then the child is rejected without evaluation.

B.6.2 Technical Details

The chromosome to be replaced is chosen using whatever operators are currently in effect and the child chromosome is generated, again with whatever operators are currently in effect. The normalised Hamming distance between the chromosome to be replaced and the child chromosome is calculated. If this normalised Hamming distance is greater than `<n>` (d_c) then the child fitness is set to -1000000.0 without evaluation otherwise the child fitness is evaluated as normal.

The generation cycle continues as normal after this point.

Appendix C

Program Code

This appendix gives the adapted program code for PGA v2.7 by Peter Ross and Geoffery H. Ballinger, written in C. The code here is not intended to be a complete listing for PGA, but just the modules that have either been created or modified for use with this project. The modules included here are:

pga.h
main.c
mds.h
mds_functions.h
mds_functions.c
mds_report.h
mds_report.c
mds_crowding.h
mds_crowding.c
mds_sharing.h
mds_sharing.c
reprod.c
select.c
init.c
help.c

PGA v2.7 is available through Peter Ross. Please note that PGA v2.7 is copyright of Peter Ross and Geoffery H. Ballinger and it is reproduced here in accordance with section 3 of the GNU General Public License (version 2).

```
/**********************************************************************
 *                                                                    *
 *            PGA - Parallel Genetic Algorithm Testbed                *
 *                                                                    *
 *        "A Parallel Implementation of a genetic algorithm"          *
 *                                                                    *
 *             By Geoffrey H. Ballinger (geoff@ed.ac.uk),             *
 *                Peter Ross (peter@ed.ac.uk)                         *
 *                                                                    *
 *             Modified  Alasdair Turner, September 1994              *
 *                                                                    *
 **********************************************************************/
```

```
/*
   Copyright (C) 1993, Peter Ross and Geoffrey H. Ballinger.
   Copyright (C) 1994 Alasdair Turner.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/

/*
 * This program has been modified for use with MDS solutions code
 *
 * Overview of changes   Alasdair Turner   September 1994
 *
 * The type `chromosome' now includes a `share' and a `trial share'
 * The these are used with sharing to store the shared function for
 * that chromosome.
 *
 * The `share' is the current sharing function, while `trial share' is
 * used when a new child chromosome is generated for the value of what
 * the sharing function would be if that chromosome were inserted into
 * the population
 */

typedef struct genodata {     /* genotype string plus reference count    */
  char *genotype;
  int refcount;
  int id;
  int parent1, parent2;
} GENODATA;

typedef struct chromosome {   /* A chromosome consists of a genotype and */
  GENODATA *gdp;              /* a fitness.  For MDS purposes includes   */
  double fitness;             /* a `share' and a `trial_share'           */
  double share;
  double trial_share;
} CHROMOSOME;

typedef struct chromopair {   /* Returned by crossover, contains one or */
  CHROMOSOME child1;          /* two children depending on -t flag      */
  CHROMOSOME child2;
} CHROMOPAIR;

/* If a genotype is longer than this (set by -n) then use
   a double in the decode routine, because a long will overflow.
   A long is 32 bits on most systems, so a value of 60 assumes
   that every call of decode() will be trying to split the
   genotype into at least two numbers, so won't be collecting
   up more than 30 bits at one go. */

#define CRUCIAL_GENO_SIZE 60

/* User actions possible */

#define QUIT      1
#define CONTINUE  2
#define RESTART   3
#define ASK_AGAIN 4

/* data input files */

#define WEIGHTSFILE  "weights"  /* name of file for knapsack data */
#define RRDATAFILE   "rrdata"   /* name of file for Royal Road data */
```

```
#define TTDATAFILE   "ttdata"   /* Name of file for timetable data */

/* On some operating systems you will need a different random (etc) */
#if hpux || SOLARIS
#ifdef SOLARIS
#include <stdlib.h>
#endif
#define random() lrand48()
#define srandom(seed) srand48((long) seed)
#endif hpux || SOLARIS

/*****************************/
/* Timetabling-specific stuff */
/*****************************/

/* The constraint types: */

enum constraint_type {
  CLASH,        /* E1 and E2 must not clash: blame E1 */
  CLASH_COPY,   /* .. but keep a copy for later info with E2 */
  BEFORE,       /* Event E must precede another: but blame E */
  EXCLUDE,      /* Event E must not be in slot S: blame E */
  PRESET        /* Event E must be in slot S: just make it so! */
};

struct constraint       {  /* used for timetable constraints of     */
  struct constraint *next; /* various sorts: e1 != e2, e1 not in s, */
  int e1;                  /* e1 must be in s, e1 must precede e2   */
  int e2;
  int s;
  double cpenalty;         /* what this constraint costs to break   */
  enum constraint_type ctype;    /* what type of constraint this is */
};

/* The following enum defines symbols for each type of entry in */
/* the timetable data file. The struct following it associates  */
/* keywords with each type, thus making it easy to add other    */
/* keywords just by adding extra lines to the struct - code     */
/* should NOT need changing.                                    */

enum entry_type {
        END_LIST,           /* (marks end of struct array below)         */
        DATA_EVENTS,        /* Number of events is #1                    */
        DATA_SLOTS,         /* Number of slots (NOT per day) is #1       */
        DATA_DAY,           /* Total number of days is #1                */
        DATA_SPREAD,        /* Near-clashes if <= #1 apart               */
        DATA_CLASHTYPE,     /* Clash penalties are per-student? type=1..3 */
        PENALTY_CLASH,      /* Penalty for clash is #1                   */
        PENALTY_ORDER,      /* Penalty for order violation is #1         */
        PENALTY_EXCLUDE,    /* Penalty for event in bad slot for it is #1 */
        PENALTY_NEARCLASH,  /* Penalty for near-clash is #1              */
        CONSTRAIN_CLASH,    /* Event #1 and event #2 in different slots  */
        CONSTRAIN_BEFORE,   /* Event #1 must precede event #2            */
        CONSTRAIN_EXCLUDE,  /* Event #1 must not be in slot #2           */
        CONSTRAIN_PRESET    /* Event #1 must be in slot #2               */
};

#define CHUNK 10  /* The array where numbers get stashed first will    */
                  /* increase by this size whenever necessary. MUST BE */
                  /* AT LEAST 2.                                       */
#define ALL -1    /* MUST BE NEGATIVE: means "get all numbers on line" */

struct tt_params {
  char *param;
  enum entry_type param_type;
  int args_wanted;
};

#define SLOTVAL(x) ((x)-'0')


------------------------------------------------------------------------------
/************************************************************************
 *                                                                      *
```

```
 *               PGA - Parallel Genetic Algorithm Testbed               *
 *                                                                      *
 *           "A Parallel Implementation of a genetic algorithm"         *
 *                                                                      *
 *                By Peter Ross (peter@ed.ac.uk),                       *
 *                   Geoffrey H. Ballinger (geoff@ed.ac.uk)             *
 *                                                                      *
 *                Modified  Alasdair Turner, September 1994             *
 *                                                                      *
 ************************************************************************/

/*
   Copyright (C) 1993, 1994 Peter Ross and Geoffrey H. Ballinger.
   Copyright (C) 1994 Alasdair Turner.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/

/*
 * This program has been modified for use with MDS solutions code
 *
 * Overview of changes   Alasdair Turner   September 1994
 *
 * MDS code header file is included for MDS functions and variables.
 *
 * main:
 *   Prints to output file / screen the status of the MDS variables:
 *     Crowding --- set with the -D flag
 *       Crowding (off, replace regardless, replace if fitter)
 *       Crowding Factor (integer >= 1)
 *     Sharing  --- set with the -E flag
 *       Sharing (off, standard, spatial)
 *       Niche Separation Distance (0 < double < 1)
 *     Tribes   --- set with the -Y flag
 *       Tribes (off, on)
 *       Niche Separation Distance (0 < double < 1)
 *   Times the intialisation and main loop using mds_clock
 *   Main loop now similar for both non-interactive and interactive modes
 *   Reporting facility changed to MDS specific
 *     function 'mds_report' rather than 'report'.
 *   New termination condition for main loop:
 *     The ga_static variable is incremented every iteration of the main
 *     loop and is reset by the mds_report function is the GA is not static.
 *     If ga_static >= n_for_stop (a user defined number of generations to
 *     stop after, set with -X flag, defaults to full reporting interval)
 *
 * report:
 *   This function has been replaced by mds_report.  Different convergence
 *   tests are now applied.
 *
 * handle:
 *   Has been modified so that it can handle input of MDS variables:
 *     Crowding: -D<opt><n>
 *       <opt> a or b, a = replace regradless, b = replace if fitter
 *       <n>   crowding factor, integer >= 1, out of range -> 1
 *     Sharing:  -E<opt><n>
 *       <opt> a or b, a = standard, b = spatial
 *       <n>   niche separation distance, 0 < double < 1, out of range -> 0.5
 *     Tribes:   -Y<n>
 *       <n>   niche separation distance, 0 < double < 1, out of range -> 0.5
 *
 * insert:
```

```
 *   The function has been rewritten to use MDS functions
 *   It now includes using crowding and sharing if appropriate
 *   The new function `mds_insert' includes the old functionality of the
 *   `insert' function.
 */

#include <stdio.h>          /* Include the IO and maths header files. */
#include <math.h>
#include <curses.h>

#include "version.h"        /* title string and version number */
#include "pga.h"            /* defines genodata and chromosome structs */
#include "screen.h"         /* screen layout configuration */

void handle();              /* Prototypes; defined in this file. */
int report();
void setprompt();
void free_pops();
int compchromo();
void insert();
void dump_chromos();
double drandom();

/* Defined in init.c */

extern CHROMOSOME *init_pops_sorted();
extern CHROMOSOME *init_pops_unsorted();

/* Defined in eval.c */

extern double eval_max();
extern double eval_dj1();
extern double eval_dj2();
extern double eval_dj3();
extern double eval_dj4();
extern double eval_dj5();
extern double eval_bf6();
extern double eval_mcb();
extern double eval_knap();
extern int read_tt_datafile();
extern double eval_tt();
extern int read_weights_file();
extern double eval_rr();
extern void read_rrdata_file();
extern void long_decode();
extern void double_decode();

/* Defined in select.c */

extern CHROMOSOME rank_select();
extern CHROMOSOME fitprop_select();
extern CHROMOSOME tm_select();
extern CHROMOSOME tn_select();

/* Defined in reprod.c */

extern void one_reproduction();
extern void gen_reproduction();
extern void ss_gen_reproduction();
extern void ss_one_reproduction();

/* Derfined in cross.c */

extern CHROMOPAIR one_pt_cross();
extern CHROMOPAIR two_pt_cross();
extern CHROMOPAIR uniform_cross();
extern CHROMOPAIR no_cross();

/* Defined in mutate.c */

extern CHROMOSOME mutate_01();
extern CHROMOSOME mutate_multi();
extern CHROMOSOME mutate_tt();
extern double adaptive();
```

```
extern double nonadaptive();
/* Use one of these in timetable mutation...*/
extern int find_worst_event();
extern int find_random_event();
extern int find_event_by_tn();
/* ...and also one of these: */
extern int find_free_slot();
extern int find_random_slot();
extern int find_slot_by_tn();

/* Defined in MDS package */

#define MAIN_C
#include "mds.h"
#undef  MAIN_C

/* Defined in help.c */

extern void pga_help();

/* Standard C lib stuff */

extern FILE *fopen();
extern int fscanf();
extern char *malloc();
extern char *calloc();
extern char *strdup();
extern long random();
#if SOLARIS || hpux
extern int sprintf();
#else
extern char *sprintf();
#endif SOLARIS || hpux

/* Globals used in various files, but declared here  */

int numpop = 5,             /* Number of populations */
    numchromo = 50,         /* Number of chromos in a population */
    geno_size = 32,         /* Length of a chromosome */
    gens_stage = 100,       /* How many times to cycle before asking user */
    repinterval = 10,       /* Interval between screen/file reports */
    isadaptive = FALSE,     /* Using adaptive mutation? */
    usegray = FALSE,        /* Treat bit patterns as Gray codes in decode */
    ss_task = FALSE,        /* A spatial selection run (ssgen or ssone)? */
    pause_first = FALSE,    /* Allow user to read complaints about tt file? */
    last_line = 24,         /* Size of screen; actually found at run time */
    rr_blocks,              /* RR number of low-level blocks (no default) */
    rr_blocksize = 8,       /* RR default block size */
    rr_gap = 7,             /* RR default gap */
    rr_mstar = 4,           /* RR default no of rewardable bits, low-level */
    migint = 10,            /* Interval between migrations */
    max_allele = 1,         /* In most problems, alleles in range 0..this */
    tournament_size = 10,   /* Default size in tournament selection */
    tt_maxevents = 0,       /* Default # of events in timetabling */
    tt_maxslots = 0,        /* Defualt # of slots in timetabling */
    tt_days = 1,            /* Default # of days in timetabling */
    tt_slots_per_day = 1,   /* Default # of slots per day in tiemtabling */
    tt_spread = 1,          /* Default definition of near-clash */
    tt_clash_type = 1,      /* Clashes don't count per-student (and gripe) */
    tt_mtn1 = 3,            /* Event tournament size in timetable mutation */
    tt_mtn2 = 3,            /* Slot tournament size in timetable mutation */
    chromo_id = 1;          /* Chromosome IDs start from this value */
int randseed;               /* The random seed */
long *sticks;               /* Used in knapsack prob: array of sizes */
long target;                /* Used in knapsack prob: target */
FILE *output = (FILE *)NULL;        /* Output file if any */
FILE *chromofile = (FILE *)NULL;    /* Chromo file, if any output file */
int interactive = TRUE;             /* Interactive? */
int finaldump = FALSE;              /* Noninteractive: do final chromodump? */
int chromoutput = FALSE;            /* Any chromosome output? */
int twins = FALSE;          /* Crossover produces pairs, or just one child? */
char *datafilename = (char *)NULL;
char outfilename[256], chromofilename[256];
char prompt[80];
```

```
double *tt_scores;          /* Used in timetabling, smart mutation */
char *geno_size_string;     /* String naming size of a chromo (hack) */
char *rr_blockarray;        /* Used in RR for higher-level block scoring */
double mute_rate = 0.02,    /* Default mutation rate */
       cross_rate = 0.6,    /* Default crossover rate if applicable */
       bias = 1.5,          /* Default bias for rank selection */
  tt_penalty_clash = 10.0,  /* TT: penalty for a clash */
  tt_penalty_order = 3.0,   /* TT: penalty for an order violation */
  tt_penalty_exclude = 5.0,/* TT: penalty for an exclusion violation */
  tt_penalty_consec = 1.0,  /* TT: penalty for a near-clash */
       rr_ustar = 1.0,      /* RR default reward for first block */
       rr_u = 0.3,          /* RR default reward for later blocks */
       rr_v = 0.02;         /* RR default bonus/penalty for low-level bit */
char *eval_name,            /* String giving name of problem */
     *maxfitness,           /* String giving max fitness (if known) */
     *cross_name,           /* String naming crossover type */
     *select_name,          /* String naming selection type */
     *repro_name;           /* String naming reproduction type */
double *total_abs_fitness; /* Used in roulette selection: size of wheel */
double ga_max_average;      /* Used to see if pop av fitness is increasing */
int evals;                  /* How often eval has been called */
int ss_xmax,ss_ymax,ss_n;  /* Size of grid and length of walk in ssN */
double user_threshold;      /* Obsolete */
int n_for_stop;             /* How many must generations static to stop */

/* Timetabling structures */

/* This will be an array where the chains get rooted: */

struct constraint **tt_event_data = (struct constraint **)NULL;

/* And this is where the preset events get noted: */

struct constraint *tt_presets = (struct constraint *)NULL;

/* This struct matches keywords to symbols, and says how many numbers */
/* ought to be picked up after the keyword is found. A value of ALL   */
/* in the args_wanted field means gather up as many numbers as can be */
/* found on the input line, rather than a fixed number.               */

#define CHUNK 10  /* The array where numbers get stashed first will    */
                  /* increase by this size whenever necessary. MUST BE */
                  /* AT LEAST 2.                                        */
#define ALL -1    /* MUST BE NEGATIVE: means "get all numbers on line" */

struct tt_params tt_paramlist[] = {
  {"events",       DATA_EVENTS,         1  },
  {"slots",        DATA_SLOTS,          1  },
  {"days",         DATA_DAY,            1  },
  {"near-clash",   DATA_SPREAD,         1  },
  {"clash-type",   DATA_CLASHTYPE,      1  },
  {"p-clash",      PENALTY_CLASH,       1  },
  {"p-order",      PENALTY_ORDER,       1  },
  {"p-exclude",    PENALTY_EXCLUDE,     1  },
  {"p-near",       PENALTY_NEARCLASH,   1  },
  {"separate",     CONSTRAIN_CLASH,     ALL},
  {"order",        CONSTRAIN_BEFORE,    ALL},
  {"exclude",      CONSTRAIN_EXCLUDE,   ALL},
  {"preset",       CONSTRAIN_PRESET,    2  },
  {NULL, END_LIST, 0  }
};


/* Default bindings for function pointers */

CHROMOSOME *(*initialise)() = init_pops_sorted;
CHROMOSOME (*selection)() = rank_select;
CHROMOPAIR (*crossover)() = two_pt_cross;
CHROMOSOME (*mutate)() = mutate_01;
void (*reproduction)() = one_reproduction;
void (*decode)() = long_decode;
double (*mrate)() = nonadaptive;
double (*evaluate)() = eval_max;
int (*find_bad_event)() = find_worst_event;
```

```
int (*find_good_slot)() = find_free_slot;


/**********************************************************************/
/* The main program.                                                  */
/**********************************************************************/

main(argc,argv)
int argc;
char *argv[];
{
  int command,chromo,pop,gen,mignext,action,gen_total,fileoutput,i;
  double average;
  char c;
  int mapped, saved, converged, satisfied;
  CHROMOSOME *pops,migchromo;

  randseed = (int) time(0);

  mignext=0;
  n_for_stop = -1;
  satisfied = FALSE;
  geno_size_string = (char *)malloc(20);
  sprintf(geno_size_string,"%d",geno_size);
  maxfitness = geno_size_string;
  eval_name = "max";
  cross_name = "two-point";
  select_name = "rank";
  repro_name = "one";

  for (command=1; command<argc; command+=1) {
    handle(argv[command]);                    /* Handle any arguments. */
  }

  srandom(randseed);

  if(n_for_stop < 0)
    n_for_stop = gens_stage;

  if(geno_size > CRUCIAL_GENO_SIZE) {
    decode = double_decode;
    if(usegray) {
      fprintf(stderr, "Chromosome length looks a bit big for Gray code\n");
      fprintf(stderr, "option - admittedly just a crude check.\n");
      exit(1);
    }
  }
  if(evaluate == eval_knap) {
    sticks = (long *)calloc(geno_size,sizeof(long));
    if(datafilename == (char *)NULL)
      datafilename = WEIGHTSFILE;
    if(!read_weights_file()) {
      fprintf(stderr,"No weights file accessible. It should hold\n");
      fprintf(stderr,"the target value (an integer) and up to\n");
      fprintf(stderr,"%d further integers. The aim is to find a\n", geno_size);
      fprintf(stderr,"subset with sum as close to the target as possible.\n");
      exit(1);
    }
  } else if(evaluate == eval_rr) {
      int n;
      double max;
      if(datafilename == (char *)NULL)
        datafilename = RRDATAFILE;
      read_rrdata_file();
      rr_blockarray = malloc(rr_blocks);
      maxfitness = (char *)malloc(20);
      n = rr_blocks;
      max = 0.0;
      while(n>0) {
        max += rr_ustar + (n-1)*rr_u;
        n /= 2;
      }
      sprintf(maxfitness,"%.6f", max);
  } else if(evaluate == eval_tt) {
```

```
    if(datafilename == (char *)NULL)
      datafilename = TTDATAFILE;
    if(!read_tt_datafile()) {
      fprintf(stderr,"See the documentation for details of the data file\n");
      exit(1);
    }
}

if((reproduction == ss_gen_reproduction)
   || (reproduction == ss_one_reproduction)) {
      ss_task = TRUE;
      ss_xmax = (int) sqrt((double)numchromo);
      ss_ymax = numchromo/ss_xmax;
      numchromo = ss_xmax*ss_ymax;
      migint = 0;
}

if(mutate == mutate_tt) {
  tt_scores = (double *)malloc(geno_size*sizeof(double));
}

if(pause_first) {
  fprintf(stderr,"**** Press RETURN to continue..");
  while(getchar() != '\n');
}

fileoutput = (output == (FILE *)NULL)? FALSE : TRUE;

if(interactive) {
  stdscr = initscr();
  last_line = tgetnum("li");
  if(last_line < 0) last_line = 24;

  mvprintw(POP_ROW,POP_COL,      "          Populations: %-6d",numpop);
  if((reproduction == ss_gen_reproduction)
     || (reproduction == ss_one_reproduction))
    mvprintw(CPP_ROW,CPP_COL,      "  Chromosomes per pop: %-4d (%dx%d)",
                                 numchromo,ss_xmax,ss_ymax);
  else
    mvprintw(CPP_ROW,CPP_COL,      "  Chromosomes per pop: %-4d",numchromo);
  mvprintw(EVLIM_ROW,EVLIM_COL,  "Generations per stage: %-6d",gens_stage);
  mvprintw(REPINT_ROW,REPINT_COL,"   Reporting interval: %-6d",repinterval);
  mvprintw(MIG_ROW,MIG_COL,      "   Migration interval: %-6d",migint);
  mvprintw(CTYPE_ROW,CTYPE_COL,  "   Crossover type: %s, %s", cross_name,
                                        (twins)? "twins" : "one child");
  if(!strcmp(repro_name,"one")
     || !strncmp(repro_name,"ssone",5))
    mvprintw(CROSS_ROW,CROSS_COL,"   Crossover rate: n/a");
  else
    mvprintw(CROSS_ROW,CROSS_COL,"   Crossover rate: %3.2f",cross_rate);
  if(!strcmp(select_name,"rank"))
    mvprintw(BIAS_ROW,BIAS_COL,  "   Selection bias: %3.2f",bias);
  else
    mvprintw(BIAS_ROW,BIAS_COL,  "   Selection bias: n/a");
  if(isadaptive)
    mvprintw(MUT_ROW,MUT_COL,    "    Mutation rate: adaptive %.4f",
                                                        mute_rate);
  else
    mvprintw(MUT_ROW,MUT_COL,    "    Mutation rate: %.6f",mute_rate);
  mvprintw(CRL_ROW,CRL_COL,      "Chromosome length: %d",geno_size);
  if(fileoutput)
    mvprintw(FNAME_ROW,FNAME_COL,"    Log file name: %s",outfilename);
  mvprintw(EVAL_ROW,EVAL_COL,    "Eval function: %s",eval_name);
  mvprintw(SELECT_ROW,SELECT_COL,"    Selection: %s", select_name);
  mvprintw(REPRO_ROW,REPRO_COL,  " Reproduction: %s",repro_name);
  mvprintw(POPTABLE_ROW,POPTABLE_COL,
             "Pop.......Average.........Best.(max = %s)",maxfitness);
  mvprintw(GEN_ROW,GEN_COL,"Generation: ");
  clrtoeol();
  mvprintw(EVCOUNT_ROW,EVCOUNT_COL,"Evaluations so far: ");
  clrtoeol();
  refresh();
  cbreak();
  noecho();
```

```
}

total_abs_fitness = (double *)malloc(numpop*sizeof(double));
ga_max_average = -1000000.0;
mds_clock(TRUE);
pops = initialise(numpop,numchromo);
mds_clock(FALSE);
gen_total = 1;
evals=0;
action = CONTINUE;

if(fileoutput) {
  fprintf(output,  "\n\n===================================\n");
  fprintf(output,  "               eval = %s\n",eval_name);
  fprintf(output,  "        max-fitness = %s\n",maxfitness);
  fprintf(output,  "       reproduction = %s\n",repro_name);
  fprintf(output,  "             select = %s\n",select_name);
  if(!strcmp(select_name,"fitprop"))
    fprintf(output, "     selection-bias = n/a\n");
  else
    fprintf(output, "     selection-bias = %.6f\n",bias);
  fprintf(output,  "  chromosome-length = %d\n",geno_size);
  fprintf(output,  "               pops = %d\n",numpop);
  if((reproduction == ss_gen_reproduction)
     || (reproduction == ss_one_reproduction))
    fprintf(output,  "chromosomes-per-pop = %d : %d x %d\n",
                     numchromo,ss_xmax,ss_ymax);
  else
    fprintf(output,  "chromosomes-per-pop = %d\n",numchromo);
  fprintf(output,  " migration-interval = %d\n",migint);
  fprintf(output,  "     crossover-type = %s\n",cross_name);
  if(!strcmp(repro_name,"one")
    || !strncmp(repro_name,"ssone",5))
    fprintf(output, "     crossover-rate = n/a\n");
  else
    fprintf(output, "     crossover-rate = %.6f\n",cross_rate);
  if(isadaptive)
    fprintf(output, "      mutation-rate = adaptive\n");
  else
    fprintf(output, "      mutation-rate = %.6f\n", mute_rate);
  fprintf(output,  " reporting-interval = %d\n", repinterval);
  if(twins)
    fprintf(output, "              twins = yes\n");
  else
    fprintf(output, "              twins = no\n");

  /* Print MDS variables */
  if(crowding == 1)
    fprintf(output, "      crowding (CF) = %d (replace regardless)\n",
            crowding_factor);
  else if (crowding == 2)
    fprintf(output, "      crowding (CF) = %d (replace if fitter)\n",
            crowding_factor);
  else
    fprintf(output, "      crowding (CF) = no\n");
  if(sharing == 1)
    fprintf(output, "      sharing (NSD) = %4.2f (standard)\n", sigma_share);
  else if (sharing == 2)
    fprintf(output, "      sharing (NSD) = %4.2f (spatial)\n", sigma_share);
  else
    fprintf(output, "      sharing (NSD) = no\n");
  if(sharing == 1)
    fprintf(output, "       tribes (NSD) = %4.2f\n", dist_cutoff);
  else
    fprintf(output, "       tribes (NSD) = no\n");

  fprintf(output,  " random-number-seed = %d\n", randseed);
  fprintf(output,  "\n");
}
mds_report(0,pops,numchromo,numpop,fileoutput,TRUE);
mapped = saved = FALSE;
i = 0; /* used to gather up any digits typed in */

while(action != QUIT) {
```

```
if(interactive) {                                       /* Interactive */
  do {
        setprompt(fileoutput,ss_task,mapped,saved,satisfied);
        mvprintw(0,1,prompt);
        clrtoeol();
        refresh();
        switch(c=getchar()) {
              case 'a': case 'A':
                      printw("starting again..");
                      refresh();
                      move(GEN_ROW,GEN_COL+12);
                      clrtoeol();
                      move(EVCOUNT_ROW,EVCOUNT_COL+20);
                      clrtoeol();
                      refresh();
                      free_pops(pops,numpop*numchromo);
                      gen_total = 1;
                      ga_max_average = -1000000.0;
                      mds_clock(TRUE);
                      pops = initialise(numpop,numchromo);
                      mds_clock(FALSE);
                      evals = 0;
                      if(fileoutput) {
                        fprintf(output,
                                "\n\n------ re-starting ------\n\n");
                      }
                      mds_report(0,pops,numchromo,numpop,fileoutput,1);
                      satisfied = FALSE;
                      mapped = saved = FALSE;
                      action = ASK_AGAIN;
                      i = 0; /* reset number input */
                      break;
              case 'c': case 'C':
                      if(satisfied)
                        action = ASK_AGAIN;
                      else {
                        printw("continuing.."); refresh();
                        action = CONTINUE;
                        mapped = saved = FALSE;
                        if(i>0) {
                           gens_stage = i;
                           mvprintw(EVLIM_ROW,EVLIM_COL,
                                    "Generations per stage: %-6d",
                                    gens_stage);
                           i=0;
                        }
                      }
                      break;
              case 'q': case 'Q':
                      action = QUIT;
                      break;
              case 's': case 'S':
                      if(fileoutput && !saved) {
                         printw("saving chromosomes.."); refresh();
                         dump_chromos(pops,gen_total-1);
                         saved = TRUE;
                      } else if(!saved) {
                         strcpy(outfilename,"unnamed");
                         printw("saving chromosomes in unnamed.chr..");
                         refresh();
                         dump_chromos(pops,gen_total-1);
                         saved = TRUE;
                      }
                      action = ASK_AGAIN;;
                      i = 0; /* reset number input */
                      break;
              case 'm': case 'M':
                      printw("writing grid map.."); refresh();
                      if(fileoutput && ss_task && !mapped) {
                        ss_map(outfilename,
                               (gen_total-1),pops,numpop,numchromo);
                        mapped = TRUE;
                      }
                      action = ASK_AGAIN;
```

```
                        i = 0; /* reset number input */
                        break;
                case '0':
                case '1':
                case '2':
                case '3':
                case '4':
                case '5':
                case '6':
                case '7':
                case '8':
                case '9':
                        i = 10*i + c - '0';
                        action = ASK_AGAIN;
                        break;
                default:
                        action = ASK_AGAIN;
                        i = 0; /* reset number input */
                        break;
          }
      } while(action == ASK_AGAIN);
    }
    else
    {                                                   /* Non interactive */
      if(satisfied) {
        if(fileoutput && finaldump)
          dump_chromos(pops,gen_total-1);
        action = QUIT;
      }
    }
    if(action == CONTINUE) {
      mds_clock(TRUE);
      for(gen=1; gen <= gens_stage && !satisfied; gen++, gen_total++) {
        for(pop=0; pop<numpop; pop+=1) {            /* For each generation */
          reproduction(pops,pop*numchromo,numchromo);        /* Reproduce. */
        }
        if (numpop>1 && migint > 0 && gen%migint==0) {       /* Migrate if */
          migchromo=selection(pops,mignext*numchromo,numchromo);
          for(pop=0; pop<numpop; pop+=1) {                   /* necessary. */
            if(pop != mignext)              /* Don't insert to its source */
              insert(pops,migchromo,pop*numchromo,numchromo);
          }
          mignext=(mignext+1)%numpop;
        }
        ga_static++;              /* ga_static is zeroed by mds_report if */
        if (ga_static >= n_for_stop) {      /* pop av fitness is improving */
          satisfied = TRUE;
        }
        if (gen_total%repinterval==0 || satisfied) {             /* Report */
          mds_clock(FALSE);
          mds_report(gen_total,pops,numchromo,numpop,fileoutput,
                     (gen_total%gens_stage == 0 || satisfied));
          mds_clock(TRUE);
        }
      }
      mds_clock(FALSE);
    }
}

/* All done, tidy up */
if(interactive) {
  echo();
  nocbreak();
  move(last_line-1,0);
  refresh();
  endwin();
}

if(fileoutput)
   fclose(output);
if(chromoutput)
   fclose(chromofile);

free_pops(pops,numpop*numchromo);
```

```
  exit(0);
}

/**********************************************************************/
/* This sets the prompt for next interaction                          */
/**********************************************************************/

int stringcat(s1,s2,n)
char *s1, *s2;
int n;
{
  int i,j;
  for(i=n,j=0; s2[j] != '\0'; i++, j++) {
    s1[i] = s2[j];
  }
  s1[i] = '\0';
  return(i);
}

void setprompt(filing,sstask,mapped,saved,satisfied)
int filing,sstask,mapped,saved,satisfied;
{
  int i;
  for(i=0;i<80;i++) prompt[i]='\0';

  i=stringcat(prompt,"(A)gain, (Q)uit", 0);
  if(filing && !saved)  i=stringcat(prompt, ", (S)ave", i);
  if(sstask && !mapped) i=stringcat(prompt, ", (M)ap", i);
  if(!satisfied)        i=stringcat(prompt, ", (C)ontinue",i);
  stringcat(prompt, ": ", i);
}

/**********************************************************************/
/* This updates the screen and, if necessary, the output file         */
/* If interactive, returns the number that have converged.            */
/* If NOT interactive, returns the number that have passed the supplied */
/* threshold value.                                                   */
/**********************************************************************/

/* OBSOLETE */

/* Note that the report function has been *replaced* for multiple
 * distinct solutions code.  The new function 'mds_report' is used in
 * its place.  The old meanings of 'threshold' and 'convergence' have
 * changed somewhat.  The user still sets the 'n_for_stop', but
 * this is taken to mean the number of generations for which any
 * population has not seen an increase in average fitness. */

int report(gen_total,fileoutput,pops,threshold)
int gen_total, fileoutput;
CHROMOSOME *pops;
double threshold;
{
  int pop, chromo;
  int n_at_threshold = 0;
  char conv_char;
  double average, best, tmp;

  if(interactive) {
    mvprintw(GEN_ROW,GEN_COL+12,"%d",gen_total);         /* necessary */
    mvprintw(EVCOUNT_ROW,EVCOUNT_COL+20,"%d", evals);
  }
  if(fileoutput) {
    fprintf(output, "\nGenerations = %d  Evaluations-so-far = %d\n",
                    gen_total,evals);
  }

  for(pop=0; pop<numpop; pop+=1) {
    average=0.0;
    best = -1000000.0;
    for(chromo=0; chromo<numchromo; chromo+=1) {
        /* Note that best is usually pops[pop*numchromo] since */
        /* the population is sorted - but not sorted with -rssN */
        tmp = pops[pop*numchromo+chromo].fitness;
```

```
        average+=tmp;
        if(best < tmp) best = tmp;
    }
    average/=(double)numchromo;
    if(fabs(average-best) < 1e-9) {
      conv_char = '=';
      n_at_threshold++;
    } else {
      conv_char = ' ';
      if(!interactive && best >= threshold)
        n_at_threshold++;
    }
    if(interactive) {
      mvprintw(POPTABLE_ROW+1+pop,POPTABLE_COL,
               "%-4d%c    %12.7f     %12.7f",
               pop, conv_char, average, best);
    }
    if(fileoutput) {
      fprintf(output, "  %-4d    %12.7f     %12.7f\n",
                         pop,    average,   best);
    }
  }
  if(interactive)
    refresh();
  return(n_at_threshold);
}


/**********************************************************************/
/* This frees whole population                                        */
/**********************************************************************/

void free_pops(whole_pop,num)
CHROMOSOME *whole_pop;
{
  int i;

  for(i=0; i<num; i++) {
    /* Avoid freeing anything twice; matters for some operating systems */
    if(whole_pop[i].gdp->refcount == 1) {
      free(whole_pop[i].gdp->genotype);
      free(whole_pop[i].gdp);
    } else
      whole_pop[i].gdp->refcount--;
  }
  free(whole_pop);
}

/**********************************************************************/
/* This resets the defaults according to the argument 'arg'.          */
/**********************************************************************/

void handle(arg)
char *arg;
{
  int newint;
  double newdouble;
  char tt_eventchoice[16], tt_slotchoice[16];

  if (*arg=='-') {               /* Either a switch ...        */
    switch (*(arg+1)) {
    case 'P' :
      if(sscanf(arg,"-P%d",&newint) == 1) {
        if (newint<1) {
          fprintf(stderr,"PGA: You must have at least 1 population.\n");
          exit(1);
        }
        else {
          numpop=newint;
        }
      }
      break;
    case 'p' :
      if(sscanf(arg,"-p%d",&newint) == 1) {
```

```
      if (newint<2) {
        fprintf(stderr,
            "PGA: You must have at least 2 chromosomes per population.\n");
        exit(1);
      }
      else {
        numchromo=newint;
      }
    }
    break;
  case 'n' :
    if(sscanf(arg,"-n%d",&newint) == 1) {
      if (newint<12) {
        fprintf(stderr,"PGA: Chromosome length must be at least 12.\n");
        exit(1);
      }
      else {
        geno_size=newint;
        sprintf(geno_size_string,"%d",geno_size);
      }
    }
    break;
  case 'N' :
    interactive = FALSE;
    break;
  case 'l' :
    if(sscanf(arg,"-l%d",&newint) == 1) {
      if (newint<1) {
        fprintf(stderr,
                "PGA: Less than one generation per stage isn't sensible!\n");
        exit(1);
      }
      else {
        gens_stage=newint;
      }
    }
    break;
  case 'i' :
    if(sscanf(arg,"-i%d",&newint) == 1) {
      if (newint<1) {
        fprintf(stderr,
              "PGA: A report every %d generations isn't sensible!\n",newint);
        exit(1);
      }
      else {
        repinterval=newint;
      }
    }
    break;
  case 'M' :
    if((reproduction == ss_gen_reproduction)
       || (reproduction == ss_one_reproduction)) {
      fprintf(stderr,
              "PGA: cannot use -M or -s with -rssoneN or -rssgenN\n");
      exit(1);
    }
    if(sscanf(arg,"-M%d",&newint) == 1) {
      if (newint<0) {
        fprintf(stderr,
                "PGA: A migration interval of %d makes no sense.\n",newint);
        exit(1);
      }
      else {
        migint=newint;
      }
    }
    break;
  case 'S' :
    if(sscanf(arg,"-S%d",&newint) == 1) {
      if(newint<0) {
        fprintf(stderr,"PGA: random seed must be non-negative, set to 1.\n");
        newint = 1;
      }
      randseed = newint;
```

```
    }
    break;
  case 'X' :
    if(sscanf(arg,"-X%d",&newint) == 1) {
      if(newint<0) {
        fprintf(stderr,
                "PGA: Number of static generations must be positive\n");
        newint = -1;
      }
      n_for_stop = newint;
    }
    break;
  case 'a' :
    isadaptive=!isadaptive;
    if (isadaptive) mrate=adaptive;
    else mrate=nonadaptive;
    break;
  case 'g' :
    usegray = TRUE;
    break;
  case 't' :
    twins = !twins;
    break;
  case 'F' :
    datafilename=strdup(arg+2);
    if(strlen(datafilename) == 0) {
      fprintf(stderr,"PGA: use -Fdatafile, no space between F and name\n");
      exit(1);
    }
    break;
  case 'm' :
    if(sscanf(arg,"-m%lf",&newdouble) == 1) {
      if (newdouble<0 || newdouble>1) {
        fprintf(stderr,"PGA: The mutation rate must be between 0 and 1.\n");
        exit(1);
      }
      else {
        mute_rate=newdouble;
      }
    }
    break;
  case 'c' :
    if(sscanf(arg,"-c%lf",&newdouble) == 1) {
      if (newdouble<0 || newdouble>1) {
        fprintf(stderr,"PGA: The crossover rate must be between 0 and 1.\n");
        exit(1);
      }
      else {
        cross_rate=newdouble;
      }
    }
    break;
  case 'b' :
    if(sscanf(arg,"-b%lf",&newdouble) == 1) {
      if (newdouble<1.0 || newdouble>2.0) {
        fprintf(stderr,"PGA: The bias must be strictly between 1 and 2.\n");
        exit(1);
      }
      else {
        bias=newdouble;
      }
    }
    break;
  case 's' :
    if((reproduction == ss_gen_reproduction)
       || (reproduction == ss_one_reproduction)) {
      fprintf(stderr, "PGA: cannot use -s or -M with -rssN\n");
      exit(1);
    }
    if (!strcmp(arg,"-srank")) {
      selection = rank_select;
      select_name = "rank";
    }
    else if (!strcmp(arg,"-sfitprop")) {
```

```
      selection = fitprop_select;
      select_name = "fitprop";
    }
    else if (!strncmp(arg,"-stm",4)) {
      selection = tm_select;
      select_name = strdup(arg+2);
      tournament_size = atoi(arg+4);
      if (tournament_size < 1) {
        fprintf(stderr,"PGA: tm selection requires arg > 1; set to 10\n");
        tournament_size = 10;
      }
    }
    else if (!strncmp(arg,"-stn",4)) {
      selection = tn_select;
      select_name = strdup(arg+2);
      tournament_size = atoi(arg+4);
      if (tournament_size < 1) {
        fprintf(stderr,"PGA: tn selection requires arg > 1; set to 10\n");
        tournament_size = 10;
      }
    }
    else {
      fprintf(stderr,"PGA: I don't know about %s selection.\n",arg+2);
      exit(1);
    }
    break;
  case 'r' :
    if (!strcmp(arg,"-rone")) {
      reproduction = one_reproduction;
      repro_name = "one";
    }
    else if (!strcmp(arg,"-rgen")) {
      reproduction = gen_reproduction;
      repro_name = "gen";
    }
    else if (!strncmp(arg,"-rssgen",7)) {
      reproduction = ss_gen_reproduction;
      repro_name = strdup(arg+2);
      initialise = init_pops_unsorted;
      select_name = "*spatial*";
      ss_n = atoi(arg+7);
      if(ss_n < 1) {
        fprintf(stderr, "PGA: ss random walk must be of length > 0, set to 5\n");
        ss_n = 5;
      }
    }
    else if (!strncmp(arg,"-rssone",7)) {
      reproduction = ss_one_reproduction;
      repro_name = strdup(arg+2);
      initialise = init_pops_unsorted;
      select_name = "*spatial*";
      ss_n = atoi(arg+7);
      if(ss_n < 1) {
        fprintf(stderr, "PGA: ss random walk must be of length > 0, set to 5\n");
        ss_n = 5;
      }
    }
    else {
      fprintf(stderr,"PGA: I don't know about %s reproduction.\n",arg+2);
      exit(1);
    }
    break;
  case 'C':
    if (!strcmp(arg,"-Cone")) {
      cross_name = "one-point";
      crossover = one_pt_cross;
    } else if (!strcmp(arg,"-Ctwo")) {
      cross_name = "two-point";
      crossover = two_pt_cross;
    } else if (!strcmp(arg,"-Cuniform")) {
      cross_name = "uniform";
      crossover = uniform_cross;
    } else if (!strcmp(arg,"-Cnone")) {
      cross_name = "none";
```

```
      crossover = no_cross;
    }
    break;
  case 'D':
    if(sscanf(arg,"-Da%d",&newint) == 1) {
      crowding = 1;
      crowding_factor = newint;
    } else if (sscanf(arg,"-Db%d",&newint) == 1) {
      crowding = 2;
      crowding_factor = newint;
    } else {
      fprintf(stderr, "PGA: -D requires option `a' or `b' and integer CF\n");
      exit(1);
    }
    if(crowding_factor < 1) {
      fprintf(stderr, "PGA: Crowding factor must be 1 or more. Set to 5\n");
      crowding_factor = 5;
    }
    break;
  case 'E':
    if(sscanf(arg,"-Ea%lf",&newdouble) == 1) {
      sharing = 1;
      sigma_share = newdouble;
    } else if (sscanf(arg,"-Eb%lf",&newdouble) == 1) {
      sharing = 2;
      sigma_share = newdouble;
    } else {
      fprintf(stderr, "PGA: -E requires option `a' or `b' and double NSD\n");
      exit(1);
    }
    if(sigma_share <= 0.0 || sigma_share >= 1.0) {
      fprintf(stderr, "PGA: NSD must be between 0 and 1. Set to 0.5\n");
      sigma_share = 0.5;
    }
    break;
  case 'Y':
    if(sscanf(arg,"-Y%lf",&newdouble) == 1) {
      landowner = 1;
      dist_cutoff = newdouble;
    }
    if(dist_cutoff <= 0.0 || dist_cutoff >= 1.0) {
      fprintf(stderr, "PGA: NSD must be between 0 and 1. Set to 0.5\n");
      dist_cutoff = 0.5;
    }
    break;
  case 'e' :
    eval_name = strdup(arg+2);
    if (!strcmp(arg,"-emax")) {
      evaluate = eval_max;
      sprintf(geno_size_string,"%d",geno_size);
      maxfitness = geno_size_string;
    }
    else if (!strcmp(arg,"-edj1")) {
      evaluate = eval_dj1;
      maxfitness = "100";
    }
    else if (!strcmp(arg,"-edj2")) {
      evaluate = eval_dj2;
      maxfitness = "1000";
    }
    else if (!strcmp(arg,"-edj3")) {
      evaluate = eval_dj3;
      maxfitness = "55";
    }
    else if (!strcmp(arg,"-edj5")) {
      evaluate = eval_dj5;
      maxfitness = "approx 499.001997";
    }
    else if (!strcmp(arg,"-ebf6")) {
      evaluate = eval_bf6;
      maxfitness = "approx 0.994007";
    }
    else if (!strcmp(arg,"-eknap")) {
      evaluate = eval_knap;
```

```
      maxfitness = "1.0";
    }
    else if (!strncmp(arg,"-ett",4)) {
      evaluate = eval_tt;
      mutate = mutate_multi;
      if(strlen(arg) > 4 && *(arg+4) == ':') {
        mutate = mutate_tt;
        if(sscanf(arg+5,"%[^+]+%[^+]",tt_eventchoice,tt_slotchoice) != 2) {
          fprintf(stderr,"PGA: mangled tt argument %s\n", arg);
          exit(1);
        } else {
          switch(tt_eventchoice[0]) {
          case 'r':  /* random */
            find_bad_event = find_random_event;
            break;
          case 'w':
            find_bad_event = find_worst_event;
            break;
          case 't':
            find_bad_event = find_event_by_tn;
            tt_mtn1 = atoi(tt_eventchoice+1);
            if(tt_mtn1 < 1) {
              fprintf(stderr,"PGA: -ett:tN+.. has silly N\n");
              exit(1);
            }
            break;
          default:
            fprintf(stderr,"PGA: -ett:.. unrecognised event chooser\n");
            exit(1);
          }
          switch(tt_slotchoice[0]) {
          case 'r':  /* random */
            find_good_slot = find_random_slot;
            break;
          case 'f':
            find_good_slot = find_free_slot;
            break;
          case 't':
            find_good_slot = find_slot_by_tn;
            tt_mtn2 = atoi(tt_slotchoice+1);
            if(tt_mtn2 < 1) {
              fprintf(stderr,"PGA: -ett:..+tN has silly N\n");
              exit(1);
            }
            break;
          default:
            fprintf(stderr,"PGA: -ett:.. unrecognised slot chooser\n");
            exit(1);
          }
        }
      }
      maxfitness = "1.0";
    }
    else if (!strcmp(arg,"-err")) {
      evaluate = eval_rr;
      /* NB, maxfitness set after arg processing, once params known */
    }
    else if (!strncmp(arg,"-emcb",5)) {
      evaluate = eval_mcb;
      maxfitness = geno_size_string;
      mutate = mutate_multi;
      max_allele = atoi(arg+5);
      if(max_allele < 1 || max_allele > 9) {
        fprintf(stderr,"PGA: mcbK: K should be in [1..9], defaulted to 2\n");
        max_allele = 2;
      }
    }
    else {
      fprintf(stderr,"PGA: unknown evaluation function: %s\n",
                      arg+2);
      exit(1);
    }
    break;
  default :
```

```
      if (*(arg+1)!='h') fprintf(stderr,"Unknown argument %s.\n\n",arg);
      pga_help();
      exit(0);
    }
  }
  else {                         /* ... or an output filename. */
    output=fopen(arg,"a");
    if (output==NULL) {
      fprintf(stderr,"PGA: can't open %s for writing.\n",arg);
      exit(1);
    }
    strcpy(outfilename,arg);
  }
}

/************************************************************************/
/* Dumps chromosomes to a file. Filename is outfilename.chr, eg foo.chr */
/************************************************************************/

void dump_chromos(pops,g)
CHROMOSOME *pops;
int g;
{
  int i,j;

  if(!chromoutput) {
     sprintf(chromofilename,"%s.chr",outfilename);
     chromofile = fopen(chromofilename,"a");
     chromoutput = (chromofile != (FILE *)NULL);
  }

  if(chromoutput) {
    for(i=0;i<numpop;i++) {
      for(j=0;j<numchromo;j++) {
        fprintf(chromofile,
                "%-2d %-4d %-5d %12.7f  %s  %5d %5d %5d\n",
                 i,   j,   g,    pops[i*numchromo+j].fitness,
                                 pops[i*numchromo+j].gdp->genotype,
                                 pops[i*numchromo+j].gdp->id,
                                 pops[i*numchromo+j].gdp->parent1,
                                 pops[i*numchromo+j].gdp->parent2);
      }
      fprintf(chromofile,"\n\n");
    }
  }
}

/************************************************************************/
/* This function inserts 'chromo' into the correct place (sorted by     */
/* fitness) within the subpopulation defined by 'base' and 'range' of   */
/* the populations array 'pops'.                                        */
/************************************************************************/

/* Functionality superceded by `mds_insert':
 *
 * Now calls `mds_insert' to insert `chromo' into the correct place
 *
 * Also calls MDS functions as appropriate:
 *   If crowding the chromosome to be replaced is no longer the least fit,
 *   but a chromosome chosen from the population using crowding.
 *
 *   If sharing the shared fitness of the whole population is calculated as
 *   if the new chromosome were to be inserted.  This save two sets of
 *   calculation of shared fitness if the chromosome is actually inserted
 */

void insert(pops,chromo,base,range)
CHROMOSOME *pops,chromo;
int base,range;
{
  CHROMOSOME *repl_chromo;
  int pop;

  pop = base/numchromo;         /* which pop we're hitting */
```

```
  repl_chromo = pops+base+range-1;  /* The last chromo in the population */

  if (crowding)
  {
     repl_chromo = mds_crowded_chromo(pops+base, range, &chromo);
  }

  if (sharing)
  {
    mds_trial_share(pops+base, range, &chromo, repl_chromo);
  }

  mds_insert(pops+base, range, &chromo, repl_chromo, pop);
}


/************************************************************************/
/* Returns >0 if chromo2 is fitter than chromo1, <0 if chromo 1 is      */
/* fitter than chromo2, and 0 if they are equally fit. This is passed   */
/* to qsort as its comparison function.                                 */
/************************************************************************/

int compchromo(chromo1,chromo2)
CHROMOSOME *chromo1,*chromo2;
{
  if((chromo2->fitness) > (chromo1->fitness))
     return(1);
  else if((chromo2->fitness) == (chromo1->fitness))
     return(0);
  else
     return(-1);
}

/***********************************************************************/
/* Returns a random double in range 0.0 to 1.0 (never 1.0 exactly)     */
/***********************************************************************/
double drandom()
{
  return((double)random()/2147483647.0);
}

/* end of pga.c */

-------------------------------------------------------------------------------
/* mds.h
 *
 * Multiple Distinct Solutions
 *
 * Header file for Multiple Distinct Solutions package.
 *
 * A module for the PGA program by Peter Ross and Geoffery H. Ballinger
 *
 * Copyright (C) 1994 Alasdair Turner
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see file COPYING
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

/*
 * This is a heaser file for the MDS function set:
 *
 * mds_insert:  A new insert function to allow sharing replacement and
 *              replacement of a chromosome at any place within the
 *              rank list of the population
 *
 * mds_crowded_chromo:
 *              A function to find a chromosome for replacement using
 *              De Jong's crowding sheme
```

```
 *
 * mds_ss_crowded_chromo:
 *             A function to find a chromosome for replacement using
 *             De Jong's crowding sheme modified for use with spatial
 *             selection
 *
 * mds_trial_share:
 *             Calculate the potential sharing value for a population
 *             and child as if the child were to be inserted into the
 *             population
 *
 * mds_ss_trial_share:
 *             Calculate the potential sharing value for the area around
 *             the chosen location and child as if the child were to be
 *             inserted into the chosen location
 *
 * mds_init_sharing_pop:
 *             Calculate the initial sharing value for all members of the
 *             population
 *
 * mds_init_ss_sharing_pop:
 *             Calculate the initial local area sharing value for all
 *             members of the population
 *
 * compchromo_sharing:
 *             A sharing version of compchromo for sorting the population
 *             when sharing
 *
 * mds_report:  A new reporting function to report the number and values
 *             of the fittest genotypes in the populations and reset
 *             the ga_static variable if the maximum of the average fitness
 *             of the population has increased
 *
 * mds_clock:   Turn on and off the timer
 *
 * mds_n_distance:
 *             Calculate the normalised Hamming distance between two
 *             genotypes
 *
 * Globals defined:
 *
 * For any module using mds.h
 *
 *    crowding     crowding flag
 *    sharing      sharing flag
 *    landowner    tribes flag
 *
 *    dist_cutoff  tribes cuttoff distance
 *
 * For use with main.c and internal to MDS modules
 *
 *    crowding_factor  crowding factor for crowding
 *    sigma_share      niche separation distance for sharing
 *
 *    ga_static        how many generations since last improvement in maximum
 *                     of average fitness
 *    ga_runtime       total runtime so far
 *    ga_inittime      time of initialisation (not used)
 *    ga_max_average   the maximum of the average fitness
 */

#define BOOLEAN int

extern BOOLEAN mds_insert(CHROMOSOME *pop_base,
                          int         pop_size,
                          CHROMOSOME *new_chromo,
                          CHROMOSOME *old_chromo,
                          int         pop);

extern CHROMOSOME *mds_crowded_chromo(CHROMOSOME *pop_base,
                                      int         pop_size,
                                      CHROMOSOME *new_chromo);

extern CHROMOSOME *mds_ss_crowded_chromo(CHROMOSOME *pop_base,
```

```
                                          int        x,
                                          int        y,
                                          CHROMOSOME *new_chromo);

extern void mds_trial_share(CHROMOSOME *pop_base,
                            int        pop_size,
                            CHROMOSOME *add_chromo,
                            CHROMOSOME *rem_chromo);

extern void mds_ss_trial_share(CHROMOSOME *pop_base,
                               int        x,
                               int        y,
                               CHROMOSOME *add_chromo,
                               CHROMOSOME *rem_chromo);

extern void mds_init_sharing_pop(CHROMOSOME *pop_base,
                                 int        pop_size,
                                 int        pop);

extern void mds_init_ss_sharing_pop(CHROMOSOME *pop_base,
                                    int        pop_size,
                                    int        pop);

/* qsort doesn't like this prototyped when properly */

extern int compchromo_sharing( /* CHROMOSOME *chromo1,
                                  CHROMOSOME *chromo2 */ );

extern void mds_report(int        gen_total,
                       CHROMOSOME *pop_base,
                       int        pop_size,
                       int        num_pops,
                       BOOLEAN    fileoutput,
                       BOOLEAN    fare_stage);

extern void mds_clock(BOOLEAN clock_start);

extern double mds_n_distance(char *genotype_1, char *genotype_2);

/*
 * Global variables for code.  If this is being called from main.c
 * then set initial values of variables, otherwise just let the module
 * see necessary variables.
 */

#ifdef MAIN_C

   int    crowding        = 0;
   int    crowding_factor = 5;

   int    sharing         = 0;
   double sigma_share     = 0.0;

   int    landowner       = 0;
   double dist_cutoff     = 0.0;

   int    ga_static       = 0;
   double ga_runtime      = 0.0;
   double ga_inittime     = 0.0;
   double ga_max_average  = 0.0;

#else

   extern int    landowner;
   extern double dist_cutoff;

   extern int    crowding;
   extern int    sharing;

#endif
```

```
/* mds_functions.h
 *
 * Multiple Distinct Solutions
 *
 * Header file for MDS generic functions, mds_functions.c
 *
 * A module for the PGA program by Peter Ross and Geoffery H. Ballinger
 *
 * Copyright (C) 1994 Alasdair Turner
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see file COPYING
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */


#define BOOLEAN int
#define TRUE    1
#define FALSE   0

/* External globals and functions required */

extern int geno_size,
           ss_task,
           crowding,
           sharing,
           landowner;

extern double *total_abs_fitness, ga_runtime, sigma_share_2;

extern void mds_sharing_replace(CHROMOSOME *pop_base,
                                int         pop_size,
                                CHROMOSOME *rem_chromo,
                                CHROMOSOME *add_chromo);

extern double fabs(double x);

/* Definitions in mds_functions.c */

int mds_h_distance(char *genotype_1,
                   char *genotype_2);

double mds_n_distance(char *genotype_1,
                      char *genotype_2);

void mds_clock(BOOLEAN clock_start);

BOOLEAN mds_insert(CHROMOSOME *pop_base,
                   int         pop_size,
                   CHROMOSOME *add_chromo,
                   CHROMOSOME *rem_chromo,
                   int         pop);

void mds_replace(CHROMOSOME *pop_first,
                 int         pop_size,
                 CHROMOSOME *rem_chromo,
                 CHROMOSOME *add_chromo);

-------------------------------------------------------------------------------
/* mds_functions.c
 *
 * Multiple Distinct Solutions
 *
 * MDS generic functions code.
 *
 * A module for the PGA program by Peter Ross and Geoffery H. Ballinger
 *
 * Copyright (C) 1994 Alasdair Turner
 *
 * This is free software; you can redistribute it and/or modify it under the
```

```
 * terms of the GNU General Public License, see file COPYING
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */


#include "pga.h"

/*
 * Standard functions
 */

#include "mds_functions.h"

/*
 * Function:  mds_h_distance
 *
 * Hamming distance
 */

int mds_h_distance(char *genotype_1, char *genotype_2)
{
   int j;
   int h_dist = 0;

   for(j=0; j < geno_size; j++)
   {
      if (genotype_1[j] != genotype_2[j])
      {
         h_dist++;
      }
   }
   return h_dist;
}

/*
 * Function:  mds_n_distance
 *
 * Normalised Hamming distance
 */

double mds_n_distance(char *genotype_1, char *genotype_2)
{
   double n_dist;

   n_dist = (double)mds_h_distance(genotype_1, genotype_2) / (double)geno_size;

   return n_dist;
}

/*
 * Function:  mds_clock
 *
 * This function starts a clock running with mds_clock(TRUE) and stops
 * it with mds_clock(FALSE).  The global ga_runtime is incremented
 * when the clock is stopped.
 */

void mds_clock(BOOLEAN start_flag)
{
   static double time_start = 0.0;       /* A bit of built in function data. */

   if (start_flag)
   {
      time_start = (double)clock() / 1e6;
   }
   else
   {
      ga_runtime += ((double)clock() / 1e6) - time_start;
   }
```

```
   return;
}

/*
 * Function:  mds_insert
 *
 * Insert new chromosome into population in place of old chromosome.
 */

BOOLEAN mds_insert(CHROMOSOME *pop_base,
                   int         pop_size,
                   CHROMOSOME *add_chromo,
                   CHROMOSOME *rem_chromo,
                   int         pop)
{
   BOOLEAN inserted;
   double add_fitness, rem_fitness;

   add_fitness = add_chromo->fitness;
   rem_fitness = rem_chromo->fitness;

   if (sharing)                          /* Divide fitness by share */
   {
      add_fitness /= add_chromo->share;
      rem_fitness /= rem_chromo->share;
   }

   if (crowding == 1)                    /* Crowding type 'a' replaces */
   {                                     /* Regardless of fitness      */
      add_fitness = 1.0;
      rem_fitness = 0.0;
   }

/*
   if (landowner)
   {
      if (mds_n_distance(add_chromo->gdp->genotype,
                         rem_chromo->gdp->genotype) > sigma_share_2)
      {
         add_fitness = 0.0;
         rem_fitness = 1.0;
      }
   }
*/

   if (add_fitness > rem_fitness)
   {
      add_chromo->gdp->refcount++;

      if (rem_chromo->gdp->refcount == 1)
      {
         free(rem_chromo->gdp->genotype);
         free(rem_chromo->gdp);
      }

      if (sharing)
      {
         total_abs_fitness[pop] += fabs(add_chromo->fitness/add_chromo->share)
                                 - fabs(rem_chromo->fitness/rem_chromo->share);

         mds_sharing_replace(pop_base, pop_size, rem_chromo, add_chromo);
      }
      else
      {
         total_abs_fitness[pop] += fabs(add_chromo->fitness)
                                 - fabs(rem_chromo->fitness);

         if (ss_task)
         {
            *rem_chromo = *add_chromo;
         }
         else
         {
            mds_replace(pop_base, pop_size, rem_chromo, add_chromo);
```

```
         }
      }

      inserted = TRUE;
   }
   else
   {
      if(add_chromo->gdp->refcount == 0)
      {
         free(add_chromo->gdp->genotype);
         free(add_chromo->gdp);
      }

      inserted = FALSE;
   }

   return inserted;
}

/*
 * Function:  mds_replace
 *
 * What I've done is to adapt the main.c insert for an all purpose
 * replacement of a chromosome
 *
 * The code works as follows: the remove chromo is zapped, and the rest of
 * of the chromosomes are shifted up or down one at a time until the new
 * chromosome can be slotted into place (in fitness order).
 *
 * Note that the `while' statements must conform to ANSI C left to
 * right interpretation, since (chromo - 1) or (chromo + 1) are not defined
 * at chromo = pop_first and chromo = pop_last respectively.
 */

void mds_replace(CHROMOSOME *pop_first,
                 int         pop_size,
                 CHROMOSOME *rem_chromo,
                 CHROMOSOME *add_chromo)
{
   CHROMOSOME *chromo, *pop_last;

   chromo   = rem_chromo;
   pop_last = pop_first + pop_size - 1;

   if (add_chromo->fitness == rem_chromo->fitness)
   {
      *chromo = *add_chromo;
   }
   else if (add_chromo->fitness > rem_chromo->fitness)
   {
      while (chromo > pop_first && add_chromo->fitness > (chromo - 1)->fitness)
      {
         *chromo = *(chromo - 1);

         chromo--;
      }
      *chromo = *add_chromo;
   }
   else
   {
      while (chromo < pop_last && add_chromo->fitness < (chromo + 1)->fitness)
      {
         *chromo = *(chromo + 1);

         chromo++;
      }
      *chromo = *add_chromo;
   }

   return;
}

-------------------------------------------------------------------------------
/* mds_report.h
```

```
 *
 * Multiple Distinct Solutions
 *
 * Header file for MDS reporting code, mds_report.c
 *
 * A module for the PGA program by Peter Ross and Geoffery H. Ballinger
 *
 * Copyright (C) 1994 Alasdair Turner
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see file COPYING
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */


#define SHOW_TOP_N 5
#define BOOLEAN int

extern int     interactive,
               geno_size,
               gens_stage,
               ga_static,
               evals,
               max_allele;
extern double  ga_runtime, ga_max_average;
extern FILE   *output;

/*
 * Type:      genostat
 *
 * Used to count and identify distinct genotypes in a population
 */

typedef struct genostat
{
   char   *genotype;
   double  fitness;
   int     count;
} GENOSTAT;

void mds_report(int         gen_total,
                CHROMOSOME *pop_base,
                int         pop_size,
                int         num_pops,
                BOOLEAN     fileoutput,
                BOOLEAN     fare_stage);

BOOLEAN mds_match(char *genotype_1,
                  char *genotype_2);

GENOSTAT *mds_member_geno_stats(char     *genotype,
                                GENOSTAT *gs_base,
                                int       gs_size);

void mds_init_report(int     gen_total,
                     int     evaluations,
                     BOOLEAN fileoutput,
                     BOOLEAN fare_stage);

void mds_close_report(BOOLEAN fileoutput,
                      BOOLEAN fare_stage);

void mds_report_standard(int    fileoutput,
                         int    pop,
                         double average,
                         double best);

void mds_report_diversity(double    best,
                          GENOSTAT *geno_stat_base,
                          int       num_geno_stats);
```

```
void mds_report_runtime(void);

int mds_prepare_geno_stats(CHROMOSOME *chromo,
                           GENOSTAT   *geno_stat_base,
                           int         num_geno_stats);

--------------------------------------------------------------------------------
/* mds_report.c
 *
 * Multiple Distinct Solutions
 *
 * MDS reporting code.
 *
 * A module for the PGA program by Peter Ross and Geoffery H. Ballinger
 *
 * Copyright (C) 1994 Alasdair Turner
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see file COPYING
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

#include <stdio.h>
#include <math.h>
#include <curses.h>

#include "pga.h"
#include "screen.h"

#include "mds_report.h"

/* n_for_stop is reinterpretted as `number of generations static for stop'
 *
 * The geno_stat array is declared static to give a permanent workspace.
 * (Yes, I know it takes up a lot of space!)
 */

/*
 * Function:  mds_report
 *
 * Report PGA info as previous report plus every full report stage print out
 * best genotype set, count, diversity and time
 *
 * If maximum of average fitness has increased then reset ga_static
 */

void mds_report(int         gen_total,
                CHROMOSOME *pop_base,
                int         pop_size,
                int         num_pops,
                BOOLEAN     fileoutput,
                BOOLEAN     fare_stage)
{
   static GENOSTAT *geno_stat_base = NULL;
   int         pop, num_geno_stats;
   CHROMOSOME *pop_ptr, *chromo;
   double      best, pop_best, average, pop_average, fitness;

   if (geno_stat_base == NULL)                         /* Allocate workspace */
   {
      geno_stat_base = (GENOSTAT *)
                       malloc(pop_size * num_pops * sizeof(GENOSTAT));
   }

   mds_init_report(gen_total, evals, fileoutput, fare_stage);

   average        = 0.0;
   best           = -1000000.0;
   num_geno_stats = 0;
```

```
   for(pop = 0, pop_ptr = pop_base; pop < num_pops; pop++, pop_ptr += pop_size)
   {
      pop_average = 0.0;
      pop_best    = -1000000.0;

      for(chromo = pop_ptr; chromo < pop_ptr + pop_size; chromo++)
      {
         fitness = chromo->fitness;
         pop_average += fitness;

         if (fitness > pop_best)
         {
            pop_best = fitness;
         }

         if (fare_stage && fileoutput)
         {
            num_geno_stats =
               mds_prepare_geno_stats(chromo, geno_stat_base, num_geno_stats);
         }
      }

      pop_average /= (double)pop_size;

      average += pop_average;

      if (pop_best > best)
      {
         best = pop_best;
      }

      if (fare_stage || interactive)
      {
         mds_report_standard(fileoutput, pop, pop_average, pop_best);
      }
   }

   average /= (double)num_pops;

   if (average > ga_max_average)
   {
      ga_max_average = average;
      ga_static      = 0;
   }

   if (fare_stage && fileoutput)
   {
      mds_report_diversity(best, geno_stat_base, num_geno_stats);

      mds_report_runtime();
   }

   mds_close_report(fileoutput, fare_stage);

   return;
}

/*
 * Function:  mds_match
 *
 * Returns true if two genotypes are identical, false otherwise
 */

BOOLEAN mds_match(char *genotype_1,
                  char *genotype_2)
{
   int j;

   for (j = 0; j < geno_size; j++)
   {
      if (genotype_1[j] != genotype_2[j])
      {
         return FALSE;
```

```
      }
   }
   return TRUE;
}

/*
 * Function:  mds_member_geno_stats
 *
 * Returns pointer to the position a genotype if stored in the genotype
 * list if it is in the list, otherwise returns NULL
 */

GENOSTAT *mds_member_geno_stats(char     *genotype,
                                GENOSTAT *gs_base,
                                int       gs_size)
{
   GENOSTAT *gs_ptr;

   for (gs_ptr = gs_base; gs_ptr < gs_base + gs_size; gs_ptr++)
   {
      if (mds_match(genotype, gs_ptr->genotype))
      {
         return gs_ptr;
      }
   }

   return NULL;
}

/*
 * Function:  mds_init_report
 *
 * Make start of report look pretty, get cursor to right place on screen
 */

void mds_init_report(int     gen_total,
                     int     evaluations,
                     BOOLEAN fileoutput,
                     BOOLEAN fare_stage)
{
   if (interactive)
   {
      mvprintw(GEN_ROW,GEN_COL+12,"%d",gen_total);         /* necessary */
      mvprintw(EVCOUNT_ROW,EVCOUNT_COL+20,"%d", evaluations);
   }

   if (fileoutput && fare_stage)
   {
      fprintf(output, "\n-----------------------------------------------");

      fprintf(output, "\nGenerations = %d  Evaluations-so-far = %d\n\n",
              gen_total, evaluations);
   }

   return;
}

/*
 * Function: mds_close_report
 *
 * Make end of report look pretty, make sure screen's displaying right stuff
 */

void mds_close_report(BOOLEAN fileoutput,
                      BOOLEAN fare_stage)
{
   if (interactive)
   {
      refresh();
   }

   if (fileoutput && fare_stage)
   {
      fprintf(output, "-----------------------------------------------\n");
```

```
   }

   return;
}

/*
 * Function: mds_report_standard
 *
 * Old stuff that report used to do
 */

void mds_report_standard(int    fileoutput,
                         int    pop,
                         double average,
                         double best)
{
   char conv_char;

   if (fabs(average-best) < 1e-9)
   {
      conv_char = '=';
   }
   else
   {
      conv_char = ' ';
   }

   if (interactive)
   {
      mvprintw(POPTABLE_ROW+1+pop,POPTABLE_COL,
               "%-4d%c    %12.7f     %12.7f",
               pop, conv_char, average, best);
   }

   if (fileoutput)
   {
      fprintf(output," %-4d    %12.7f     %12.7f\n",
                     pop,    average,   best);
   }

   return;
}

/*
 * Function:  mds_report_diversity
 *
 * Report details of genotype set: the genotypes within 5e-9 of the fittest,
 *                                 their fitness, number in this set, and
 *                                 their allele count diversity
 */

void mds_report_diversity(double    best,
                          GENOSTAT *gs_base,
                          int       num_geno_stats)
{
   GENOSTAT *gs_ptr;
   char *allele_d_base, *allele_d, *allele;
   int   genome_diversity, distinct_solutions;
   int   size = (max_allele + 2) * geno_size;

   allele_d_base = (char *)malloc(sizeof(char) * size);

   for (allele = allele_d_base; allele < allele_d_base + size; allele++)
   {
      *allele = '\0';
   }

   fprintf(output, "\nChromosomes with fitness within 5e-9 of best.\n");

   genome_diversity   = 0;
   distinct_solutions = 0;

   for (gs_ptr = gs_base; gs_ptr < gs_base + num_geno_stats; gs_ptr++)
   {
```

```
      if (fabs(gs_ptr->fitness - best) < 5e-9)  /* 5e-9 is resolution we see */
      {
         for (allele = gs_ptr->genotype, allele_d = allele_d_base;
              allele < gs_ptr->genotype + geno_size;
              allele++, allele_d += (max_allele + 2))
         {
            if (strchr(allele_d, *allele) == NULL)
            {
               *(allele_d + strlen(allele_d)) = *allele;
            }
         }

         fprintf(output, "%s, fitness %12.7f, %d occurence(s)\n",
                 gs_ptr->genotype, gs_ptr->fitness, gs_ptr->count);

         distinct_solutions++;
      }
   }

   for (allele_d = allele_d_base;
        allele_d < allele_d_base + size;
        allele_d += (max_allele + 2))
   {
      genome_diversity += strlen(allele_d);
   }

   fprintf(output, "\nFitness   %13.8f\n", best);
   fprintf(output, "Solutions %4d\n", distinct_solutions);
   fprintf(output, "Diversity %4d\n", genome_diversity);

   free(allele_d_base);

   return;
}

/*
 * Function:  mds_report_runtime
 *
 * Report total runtime
 */

void mds_report_runtime(void)
{
   fprintf(output, "Runtime   %8.3f\n", ga_runtime);

   return;
}

/*
 * Function:  mds_prepare_geno_stats
 *
 * Given a chromosome, add it's genotype to the genostats list if it
 * isn't already in the list, or, if it is, add 1 to the number of
 * chromosomes found with that genotype
 */

int mds_prepare_geno_stats(CHROMOSOME *chromo,
                           GENOSTAT   *geno_stat_base,
                           int         num_geno_stats)
{
   GENOSTAT *gs_ptr;

   gs_ptr = mds_member_geno_stats(chromo->gdp->genotype,
                                  geno_stat_base, num_geno_stats);

   if (gs_ptr == NULL)
   {
      num_geno_stats++;

      gs_ptr = geno_stat_base + num_geno_stats - 1;

      gs_ptr->genotype = chromo->gdp->genotype;
      gs_ptr->fitness  = chromo->fitness;
      gs_ptr->count    = 1;
```

```
   }
   else
   {
      gs_ptr->count++;
   }

   return num_geno_stats;
}
```

```
--------------------------------------------------------------------------------
/* mds_crowding.h
 *
 * Multiple Distinct Solutions
 *
 * Header file for MDS crowding code, mds_crowding.c
 *
 * A module for the PGA program by Peter Ross and Geoffery H. Ballinger
 *
 * Copyright (C) 1994 Alasdair Turner
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see file COPYING
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */


#define  CHROMO_POS(x,y)  (pop_base + ((y) * ss_xmax) + (x))

extern int crowding_factor,
           ss_xmax,
           ss_ymax;

extern int rand_dir(void);
extern int mds_h_distance(char *genotype_1, char *genotype_2);

CHROMOSOME *mds_crowded_chromo(CHROMOSOME *pop_base,
                               int         pop_size,
                               CHROMOSOME *new_chromo);

CHROMOSOME *mds_ss_crowded_chromo(CHROMOSOME *pop_base,
                                  int         x,
                                  int         y,
                                  CHROMOSOME *new_chromo);
```

```
--------------------------------------------------------------------------------
/* mds_crowding.c
 *
 * Multiple Distinct Solutions
 *
 * MDS crowding code.
 *
 * A module for the PGA program by Peter Ross and Geoffery H. Ballinger
 *
 * Copyright (C) 1994 Alasdair Turner
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see file COPYING
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */
```

```
#include "pga.h"

/*
 * Crowding
 */

#include "mds_crowding.h"

/*
 * Hamming distance is used to compare similarity
 *
 * Choosing a similar chromosome is an interesting problem - the (?)
 * simplest method is used here.
 */

/*
 * Function:  mds_crowded_chromo
 *
 * Choose a chromosome to be crowded out of the population using De Jong's
 * crowding scheme
 */

CHROMOSOME *mds_crowded_chromo(CHROMOSOME *pop_base,
                               int         pop_size,
                               CHROMOSOME *new_chromo)
{
   int min_h_dist, h_dist, i;

   CHROMOSOME *chromo, *crowded_chromo;

   crowded_chromo = (pop_base + 1) + (random() % (pop_size - 2));

   min_h_dist     = mds_h_distance(new_chromo->gdp->genotype,
                                   crowded_chromo->gdp->genotype);

   for(i=0; i < (crowding_factor - 1); i++)
   {
      chromo = (pop_base + 1) + (random() % (pop_size - 2));

      h_dist = mds_h_distance(new_chromo->gdp->genotype,
                              chromo->gdp->genotype);

      if (h_dist <= min_h_dist)
      {
         crowded_chromo = chromo;
         min_h_dist     = h_dist;
      }
   }

   return crowded_chromo;
}

/*
 * Function:  mds_ss_crowded_chromo
 *
 * In the spatial selection version, we taken a random walk of crowding
 * factor length to decide which chromosome is replaced.
 *
 * The random walk method is taken from reprod.c
 */

CHROMOSOME *mds_ss_crowded_chromo(CHROMOSOME *pop_base,
                                  int         x,
                                  int         y,
                                  CHROMOSOME *new_chromo)
{
   int min_h_dist, h_dist, i;

   CHROMOSOME *chromo, *crowded_chromo;

   crowded_chromo = CHROMO_POS(x,y);
   min_h_dist     = mds_h_distance(new_chromo->gdp->genotype,
                                   crowded_chromo->gdp->genotype);
```

```
   for(i=0; i < (crowding_factor - 1); i++)
   {
      switch(rand_dir())
      {
      case 0:      y--; break;
      case 1: x++; y--; break;
      case 2: x++;      break;
      case 3: x++; y++; break;
      case 4:      y++; break;
      case 5: x--; y++; break;
      case 6: x--;      break;
      case 7: x--; y--; break;
      }

      x = (x+ss_xmax) % ss_xmax;
      y = (y+ss_ymax) % ss_ymax;

      chromo = CHROMO_POS(x,y);
      h_dist = mds_h_distance(new_chromo->gdp->genotype,chromo->gdp->genotype);

      if (h_dist <= min_h_dist)
      {
         crowded_chromo = chromo;
         min_h_dist     = h_dist;
      }
   }

   return crowded_chromo;
}

------------------------------------------------------------------------------
/* mds_sharing.h
 *
 * Multiple Distinct Solutions
 *
 * Header file for MDS sharing code, mds_sharing.c
 *
 * A module for the PGA program by Peter Ross and Geoffery H. Ballinger
 *
 * Copyright (C) 1994 Alasdair Turner
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see file COPYING
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */


#define BOOLEAN int

#define  CHROMO_POS(x,y)  (pop_base + ((y) * ss_xmax) + (x))

extern int     geno_size,
               ss_task,
               ss_n,
               ss_xmax,
               ss_ymax;

extern double  sigma_share,
              *total_abs_fitness;

extern double mds_n_distance(char *genotype_1,
                             char *genotype_2);

extern double fabs(double x);

/* Defined in mds_sharing.c */

void mds_trial_share(CHROMOSOME *pop_base,
                       int         pop_size,
```

```
                       CHROMOSOME *add_chromo,
                       CHROMOSOME *rem_chromo);

void mds_ss_trial_share(CHROMOSOME *pop_base,
                          int        x,
                          int        y,
                          CHROMOSOME *add_chromo,
                          CHROMOSOME *rem_chromo);

void mds_add_chromo_share(CHROMOSOME *chromo_ptr,
                          CHROMOSOME *add_chromo,
                          CHROMOSOME *rem_chromo);

void mds_sharing_replace(CHROMOSOME *pop_base,
                         int        pop_size,
                         CHROMOSOME *rem_chromo,
                         CHROMOSOME *add_chromo);

void mds_init_sharing_pop(CHROMOSOME *pop_base,
                          int        pop_size,
                          int        pop);

void mds_init_ss_sharing_pop(CHROMOSOME *pop_base,
                             int        pop_size,
                             int        pop);

/* qsort doesn't like this being prototyped properly */

int compchromo_sharing( /* CHROMOSOME *chromo1,
                           CHROMOSOME *chromo2  */ );


------------------------------------------------------------------------------
/* mds_sharing.c
 *
 * Multiple Distinct Solutions
 *
 * MDS sharing code.
 *
 * A module for the PGA program by Peter Ross and Geoffery H. Ballinger
 *
 * Copyright (C) 1994 Alasdair Turner
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see file COPYING
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

#include "pga.h"

/*
 * Sharing
 */

#include "mds_sharing.h"

/*
 * This uses a single pass of the population to perform the following
 * operations:
 * 1. Calculate a trial_share value for a chromosome to be added.
 * 2. Increment trial_share for each member of the population according
 *    to distance to chromosome to be added.
 * 3. Decrement trial_share for each member of the population according
 *    to distance to chromosome to be removed.
 */

/*
 * Function:  mds_trial_share
 *
 * Move through whole population, calculating the sharing function for
```

```
 * each chromosome
 */

void mds_trial_share(CHROMOSOME *pop_base,
                     int         pop_size,
                     CHROMOSOME *add_chromo,
                     CHROMOSOME *rem_chromo)
{
   CHROMOSOME *chromo_ptr;

   add_chromo->share = 1.0;

   for (chromo_ptr = pop_base; chromo_ptr < pop_base + pop_size; chromo_ptr++)
   {
      mds_add_chromo_share(chromo_ptr, add_chromo, rem_chromo);
   }

   return;
}

/*
 * Similar to mds_trial_share --- only mds_ss_trial_share only looks at a
 * subset of the population at any time
 */

void mds_ss_trial_share(CHROMOSOME *pop_base,
                        int         x,
                        int         y,
                        CHROMOSOME *add_chromo,
                        CHROMOSOME *rem_chromo)
{
   CHROMOSOME *chromo_ptr;
   int         i, j;

   add_chromo->share = 1.0;

   for (i = x + ss_xmax - ss_n; i <= x + ss_xmax + ss_n; i++)
   {
      for (j = y + ss_ymax - ss_n; j <= y + ss_ymax + ss_n; j++)
      {
         chromo_ptr = CHROMO_POS((i % ss_xmax), (j % ss_ymax));

         mds_add_chromo_share(chromo_ptr, add_chromo, rem_chromo);
      }
   }

   return;
}

/*
 * Function:  mds_add_chromo_share
 *
 * Add the child-this_chromo sharing function to the child sharing value.
 * Add the child-this_chromo sharing function to the this_chromo trial share.
 * Remove the rem_chromo-this_chromo sharing function from the this_chromo
 *   trial share
 */

void mds_add_chromo_share(CHROMOSOME *chromo_ptr,
                          CHROMOSOME *add_chromo,
                          CHROMOSOME *rem_chromo)
{
   double add_dist, rem_dist;

   add_dist = mds_n_distance(add_chromo->gdp->genotype,
                             chromo_ptr->gdp->genotype);
   rem_dist = mds_n_distance(rem_chromo->gdp->genotype,
                             chromo_ptr->gdp->genotype);

   chromo_ptr->trial_share = chromo_ptr->share;

   if (add_dist < sigma_share && chromo_ptr != rem_chromo)
   {
      add_chromo->share += (1.0 - (add_dist / sigma_share));
```

```
      chromo_ptr->trial_share += (1.0 - (add_dist / sigma_share));
   }

   if (rem_dist < sigma_share)
   {
      chromo_ptr->trial_share -= (1.0 - (rem_dist / sigma_share));
   }

   return;
}

/*
 * Function:  mds_sharing_replace
 *
 * Replace chromo with child, update population from trial share to actual
 * share and resort population (horrible, but true --- think what happens
 * to all the fitnesses when the sharing value has been changed for the
 * population)
 */

void mds_sharing_replace(CHROMOSOME *pop_base,
                         int         pop_size,
                         CHROMOSOME *rem_chromo,
                         CHROMOSOME *add_chromo)
{
   CHROMOSOME *chromo_ptr;

   add_chromo->trial_share = add_chromo->share;

   *rem_chromo = *add_chromo;

   for (chromo_ptr = pop_base; chromo_ptr < pop_base+pop_size; chromo_ptr++)
   {
      chromo_ptr->share = chromo_ptr->trial_share;
   }

   if (!ss_task)
   {
      qsort((char *)pop_base, pop_size, sizeof(CHROMOSOME),compchromo_sharing);
   }

   return;
}

/*
 * Function:  mds_init_sharing_pop
 *
 * Initialise a sharing population
 *
 * Note: this function overwrites total_abs_fitness
 */

void mds_init_sharing_pop(CHROMOSOME *pop_base,
                          int         pop_size,
                          int         pop)
{
   CHROMOSOME *chromo_1, *chromo_2;
   double dist;

   total_abs_fitness[pop] = 0.0;

   for (chromo_1 = pop_base; chromo_1 < pop_base + pop_size; chromo_1++)
   {
      chromo_1->share = 0.0;

      for (chromo_2 = pop_base; chromo_2 < pop_base + pop_size; chromo_2++)
      {
         dist = mds_n_distance(chromo_1->gdp->genotype,
                               chromo_2->gdp->genotype);

         if (dist < sigma_share)
         {
            chromo_1->share += (1.0 - (dist / sigma_share));
```

```
         }
      }

      chromo_1->trial_share = chromo_1->share;

      total_abs_fitness[pop] += fabs(chromo_1->fitness / chromo_1->share);
   }

   return;
}

/*
 * Function:  mds_init_ss_sharing_pop
 *
 * Initialise a spatial sharing population
 */

void mds_init_ss_sharing_pop(CHROMOSOME *pop_base,
                             int         pop_size,
                             int         pop)
{
   CHROMOSOME *chromo_1, *chromo_2;
   int x, y, i, j;
   double dist;

   total_abs_fitness[pop] = 0.0;

   for (x = 0; x < ss_xmax; x++)
   {
      for (y = 0; y < ss_ymax; y++)
      {
         chromo_1 = CHROMO_POS(x, y);
         chromo_1->share = 0.0;

         for (i = x + ss_xmax - ss_n; i <= x + ss_xmax + ss_n; i++)
         {
            for (j = y + ss_ymax - ss_n; j <= y + ss_ymax + ss_n; j++)
            {
               chromo_2 = CHROMO_POS((i % ss_xmax), (j % ss_ymax));

               dist = mds_n_distance(chromo_1->gdp->genotype,
                                     chromo_2->gdp->genotype);

               if (dist < sigma_share)
               {
                  chromo_1->share += (1.0 - (dist / sigma_share));

               }
            }
         }

         chromo_1->trial_share = chromo_1->share;

         total_abs_fitness[pop] += fabs(chromo_1->fitness / chromo_1->share);
      }
   }

   return;
}

/*
 * Function:  compchromo_sharing
 *
 * Sharing sort function
 */

int compchromo_sharing(CHROMOSOME *chromo1,
                       CHROMOSOME *chromo2)
{
   if((chromo2->fitness / chromo2->share) >
      (chromo1->fitness / chromo1->share))
   {
      return 1;
   }
```

```
   else if((chromo2->fitness / chromo2->share) ==
           (chromo1->fitness / chromo1->share))
   {
      return 0;
   }
   else
   {
      return -1;
   }
}

--------------------------------------------------------------------------------
/* reprod.c
 * Copyright (C) 1993, 1994 Peter Ross and Geoff Ballinger.
 * Modified by Alasdair Turner (copyright (C) 1994 Alasdair Turner) Sept 1994
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see the file COPYING.
 *
 * Reproduction choices. Each takes a CHROMOSOME * pointing to the array
 * of populations, and two ints giving base index and size of target pop
 * in that array. Returns nothing.
 */

/*
 * This program has been modified for use with MDS solutions code
 *
 * Overview of changes   Alasdair Turner   September 1994
 *
 * MDS code header file is included for MDS functions and variables.
 *
 * ss_select:
 *   Uses the shared fitness rather than standard fitness if sharing
 *   is selected.
 *
 * ss_one_reproduction:
 *   Modified to include crowding, sharing and tribes methods.
 *   If crowding, a new location is chosen for the chromosome to be replaced.
 *   If sharing, the new shared fitness for the population if the child
 *     were inserted is calculated
 *   If tribes the fitness is only evaluated if the child is sufficiently
 *     close in terms of Hamming distance to the chromosome to be replaced.
 *
 *   Uses the `mds_insert' function to insert into the population
 */

#define TRUE 1
#define FALSE 0

#include <stdio.h>
#include "pga.h"

#include "mds.h"

extern CHROMOSOME (*selection)();
extern CHROMOPAIR (*crossover)();
extern CHROMOSOME (*mutate)();
extern double (*mrate)();
extern double (*evaluate)();
extern double drandom();
extern int compchromo();
extern void insert();

extern double mute_rate;
extern double cross_rate;
extern double *total_abs_fitness;
extern int numchromo;
extern int twins;
extern int ss_xmax, ss_ymax, ss_n;


extern char *malloc();
extern double fabs();
```

```
/************************************************************************/
/* This function performs one-at-a-time reproduction on the specified   */
/* subpopulation. Adaptive mutation is used if activated.               */
/************************************************************************/

void one_reproduction(pops,base,range)
CHROMOSOME *pops;
int base,range;
{
  CHROMOSOME parent1,parent2,child;
  CHROMOPAIR children;

  parent1=selection(pops,base,range);        /* Select two parents. */
  parent2=selection(pops,base,range);

  children=crossover(parent1,parent2);       /* Cross them */
  child=mutate(children.child1,mrate(parent1,parent2)); /*  mutate (per bit) */
  child.fitness=evaluate(child.gdp->genotype);  /* Evaluate the child. */
  insert(pops,child,base,range);          /* Insert the child into */
                                          /* the population.      */

  if(twins) {
    child=mutate(children.child2,mrate(parent1,parent2)); /*  mutate */
    child.fitness=evaluate(child.gdp->genotype);  /* Evaluate the child. */
    insert(pops,child,base,range);        /* Insert the child into */
                                          /* the population.      */
  }
}

/************************************************************************/
/* This function performs generation based reproduction. A new          */
/* population is built by selecting members of the old one, possibly   */
/* crossing them with another, and possibly mutating them. This new     */
/* population is sorted and then used to replace the old population.   */
/************************************************************************/

void gen_reproduction(pops,base,range)
CHROMOSOME *pops;
int base,range;
{
  CHROMOSOME *newpop,parent1,parent2;
  CHROMOPAIR children;
  int loop,reeval,pop;

  pop = base/numchromo;  /* which population we're reconstructing */

  newpop=(CHROMOSOME *)malloc(sizeof(CHROMOSOME)*range);

  for(loop=0; loop<range;) {
    reeval=FALSE;
    parent1=selection(pops,base,range);
    parent2=selection(pops,base,range);
    if (drandom()<=cross_rate) {
      reeval=TRUE;
      children=crossover(parent1,parent2);
      children.child1=mutate(children.child1, mrate(parent1,parent2));
      if(twins)
        children.child2=mutate(children.child2, mrate(parent1,parent2));
    }
    else {
      children.child1=parent1;
      if(twins)
        children.child2=parent2;
    }

    /* By this point, child may have refcount 0 if product of crossover */
    /* and/or of mutation; or may have refcount > 0 otherwise.          */
    /* Increment child's refcount, so it is 1 if completely new, >1     */
    /* if in old population too.                                        */
    children.child1.gdp->refcount++;
    if(twins)
      children.child2.gdp->refcount++;
```

```
    if (reeval) {
      children.child1.fitness=evaluate(children.child1.gdp->genotype);
      if(twins)
        children.child2.fitness=evaluate(children.child2.gdp->genotype);
    }
    newpop[loop++]=children.child1;
    if(twins)
      newpop[loop++]=children.child2;
  }

  qsort((char *)newpop,range,sizeof(CHROMOSOME),compchromo);

  total_abs_fitness[pop] = 0.0;
  for(loop=0; loop<range; loop+=1) {
    /* If old member is not wanted in new population or elsewhere, its */
    /* refcount will be 1. Decrement ref counts in old pop, ditch the  */
    /* unwanted. */
    if(pops[base+loop].gdp->refcount == 1) {
       free(pops[base+loop].gdp->genotype);
       free(pops[base+loop].gdp);
    } else
       pops[base+loop].gdp->refcount--;

    pops[base+loop]=newpop[loop];
    total_abs_fitness[pop] += fabs(newpop[loop].fitness);
  }

  free(newpop);
}

/* This does 'spatial selection' reproduction, distantly based on Wright's
 * shifting-balance model of evolution. The much more common panmictic model
 * is Fisher's in which the entire population competes in selection - eg
 * in roulette-wheel or rank-based selection, the fitness of all members
 * of the population is implicitly used in each selection. In Wright's
 * model, selection need only consider chromosomes in some (2-D) neighbourhood
 * each time. It seems to make sense to use a 2-D grid rather than 3-D or
 * higher; given sufficiently high dimension, every point would be an
 * immediate neighbour of every other so it would be a form of panmictic
 * evolution again.
 *
 * In this implementation, the chromosomes are regarded as being arranged
 * in a 2-D toroidal grid; so first the population size is first automatically
 * adjusted so that the grid can be rectangular and roughly square.
 *
 * Reproduction is generational and proceeds as follows. Each chromosome
 * is replaced by the result of crossover and mutation applied to two parents
 * chosen as the fittest met on a random walk of length ss_n starting from
 * the place where the child will be installed. If you use the -t (twins)
 * flag location (x,y) and location (x,y+1) (if it exists) get used to
 * store the children.
 *
 * This spatial selection reproduction utilises specific selection and
 * insertion algorithms, so you cannot use the -s option with it.
 * /


/*****************************************************************/
/* This chooses a location in the neighbourhood of given x,y using */
/* a random walk of length ss_n. Choose the fittest found, ties    */
/* are broken by choosing last.                                    */
/*****************************************************************/

#define LOCATION(x,y) (base+(y)*ss_xmax+(x))

/* The helps cut down on the number of drandom() calls. */
/* Returns random integer in range [0,7].              */
int rand_dir_bits = 0;
int rand_dir_store = 0;
int rand_dir()
{
  int direction;

  if(rand_dir_bits == 0) {
```

```
    rand_dir_store = (int) (drandom() * 16777216.0);
    rand_dir_bits = 24;
  }
  direction = rand_dir_store & 0x07;
  rand_dir_store = rand_dir_store >> 3;
  rand_dir_bits -= 3;
  return(direction);
}

CHROMOSOME ss_select(pops,base,x,y)
CHROMOSOME *pops;
int base,x,y;
{
  int i;
  CHROMOSOME chosen;
  double f;

  chosen = pops[LOCATION(x,y)];
  f = chosen.fitness / (sharing ? chosen.share : 1.0);
  for(i=0;i<ss_n;i++) {
    switch(rand_dir()) {
        case 0:
          y -= 1;
          break;
        case 1:
          x += 1;
          y -= 1;
          break;
        case 2:
          x +=1;
          break;
        case 3:
          x +=1;
          y +=1;
          break;
        case 4:
          y += 1;
          break;
        case 5:
          x -=1;
          y +=1;
          break;
        case 6:
          x -=1;
          break;
        case 7:
          x -=1;
          y -=1;
          break;
    }

    /* Note that it is probably cheaper to do the next two lines */
    /* than to use conditional tests of whether x or y is out of */
    /* bounds.                                                   */
    x = (x+ss_xmax) % ss_xmax;
    y = (y+ss_ymax) % ss_ymax;

    if(pops[LOCATION(x,y)].fitness /
       (sharing ? pops[LOCATION(x,y)].share : 1.0) >= f) {
       chosen = pops[LOCATION(x,y)];
       f = chosen.fitness / (sharing ? chosen.share : 1.0);
    }
  }
  return(chosen);
}

/*****************************************************************/
/* Spatial selection. Uses specific selection and insertion, so  */
/* you cannot set them independently. Generational version.      */
/*****************************************************************/

void ss_gen_reproduction(pops,base,range)
CHROMOSOME *pops;
int base, range;
```

```
{
  int x,y;
  CHROMOSOME parent1, parent2;
  CHROMOPAIR children;
  CHROMOSOME *newpop;

  /* Somewhere to keep the new population, so that all selection is */
  /* from the old population.                                       */
  newpop = (CHROMOSOME *)malloc(range*sizeof(CHROMOSOME));

  /* Scan the grid, doing selection and insertion */
  for(x=0;x<ss_xmax;x++) {
    for(y=0;y<ss_ymax; ) {
      parent1 = ss_select(pops,base,x,y);
      parent2 = ss_select(pops,base,x,y);
      children = crossover(parent1, parent2);
      children.child1 = mutate(children.child1, mrate(parent1,parent2));
      children.child1.fitness=evaluate(children.child1.gdp->genotype);
      children.child1.gdp->refcount++;
      newpop[y*ss_xmax+x] = children.child1;
      y++;
      /* If twins, put the second chromosome just below the first if */
      /* there is a place to put it. Don't wrap; top is already new. */
      if(twins && y < ss_ymax) {
        children.child2 = mutate(children.child2, mrate(parent1,parent2));
        children.child2.fitness=evaluate(children.child2.gdp->genotype);
        children.child2.gdp->refcount++;
        newpop[y*ss_xmax+x] = children.child2;
        y++;
      }
    }
  }

  /* Copy new that is better into old, freeing any unwanted old stuff. */
  for(x=0;x<numchromo;x++) {
    if(pops[base+x].fitness < newpop[x].fitness) {
      if(pops[base+x].gdp->refcount == 1) {
          free(pops[base+x].gdp->genotype);
          free(pops[base+x].gdp);
      } else
         pops[base+x].gdp->refcount--;
      pops[base+x] = newpop[x];
    } else {
      free(newpop[x].gdp->genotype);
      free(newpop[x].gdp);
    }
  }

  free(newpop);
}

/*****************************************************************/
/* Spatial selection. Uses specific selection and insertion, so    */
/* you cannot set them independently. One (Genitor-ish) version.   */
/*****************************************************************/

void ss_one_reproduction(pops,base,range)
CHROMOSOME *pops;
int base, range;
{
  int x,y;
  CHROMOSOME parent1, parent2;
  CHROMOPAIR children;
  CHROMOSOME *repl_chromo;

  x = (int)(drandom() * (double)ss_xmax);
  y = (int)(drandom() * (double)ss_ymax);

  if(twins) { /* Twins occupy adjacent sites vertically */
    y = (y >> 1) << 1;
  }

  parent1 = ss_select(pops,base,x,y);
  parent2 = ss_select(pops,base,x,y);
```

```
  children = crossover(parent1, parent2);
  children.child1 = mutate(children.child1, mrate(parent1,parent2));

  if(crowding)
    repl_chromo = mds_ss_crowded_chromo(pops+base, x, y, &children.child1);
  else
    repl_chromo = &pops[LOCATION(x,y)];

  if (landowner)
  {
    if (mds_n_distance(children.child1.gdp->genotype,
                       repl_chromo->gdp->genotype) > dist_cutoff)
    {
      children.child1.fitness=-1000000.0;
    }
    else
    {
      children.child1.fitness=evaluate(children.child1.gdp->genotype);
    }
  }

  if (sharing == 1)
    mds_trial_share(pops+base, range, &children.child1, repl_chromo);
  else if (sharing == 2)
    mds_ss_trial_share(pops+base, x, y, &children.child1, repl_chromo);
  mds_insert(pops+base, range, &children.child1, repl_chromo, 0);

  /* If twins, put the second chromosome just below the first. But don't */
  /* wrap, top may be valuable and unrelated.                            */
  y++;
  if(twins && y < ss_ymax) {
    children.child2 = mutate(children.child2, mrate(parent1,parent2));
    children.child2.fitness=evaluate(children.child2.gdp->genotype);
    if (crowding)
      repl_chromo = mds_ss_crowded_chromo(pops+base, x, y, &children.child2);
    else
      repl_chromo = &pops[LOCATION(x,y)];
    if (sharing == 1)
      mds_trial_share(pops+base, range, &children.child2, repl_chromo);
    else if (sharing == 2)
      mds_ss_trial_share(pops+base, x, y, &children.child2, repl_chromo);
    mds_insert(pops+base, range, &children.child2, repl_chromo, 0);
  }
}



--------------------------------------------------------------------------------
/* select.c
 * Copyright (C) 1993, 1994 Peter Ross and Geoff Ballinger.
 * Modified by Alasdair Turner (copyright (C) 1994 Alasdair Turner) Sept 1994
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see the file COPYING.
 *
 * Selection choices, Each takes a CHROMOSOME * pointing to array of all
 * pops, and two integers giving base index and size of target pop; returns
 * the selected CHROMOSOME.
 */

/*
 * This program has been modified for use with MDS solutions code
 *
 * Overview of changes   Alasdair Turner   September 1994
 *
 * fitprop_select:
 *   Uses the shared fitness rather than standard fitness if sharing
 *   is selected.
 */

#include "pga.h"

extern double bias;
extern double drandom();
```

```
extern double *total_abs_fitness;
extern int numchromo;
extern int tournament_size;

extern int sharing;

extern double sqrt();
extern double fabs();


/***********************************************************************/
/* This function performs rank based selection and is lifted from      */
/* Whitley's paper on GENITOR in icga 89. It selects from the specified */
/* subpopulation within 'pops'.                                         */
/***********************************************************************/

CHROMOSOME rank_select(pops,base,range)
CHROMOSOME *pops;
int base,range;
{
  double value;

  if(bias<1.05)
     value=drandom()/bias;
  else {
     value=(bias-sqrt(bias*bias-4.0*(bias-1.0)*drandom()))/(2.0*(bias-1.0));
  }
  value=value*(double)range;

  return(pops[base+(int)value]);
}


/***********************************************************************/
/* This function implements fitness proportionate selection. It treats  */
/* the fitness of the population as a "roulette wheel", rolls a ball on */
/* the wheel, and picks the chromosome within whose sector the ball     */
/* stops.                                                               */
/***********************************************************************/

CHROMOSOME fitprop_select(pops,base,range)
CHROMOSOME *pops;
int base,range;
{
  int pop,loop;
  double where;

  pop = base/numchromo;  /* which pop we're working on */

  where=drandom()*total_abs_fitness[pop];  /* Random float in 0..total */

  for(loop=base; where>fabs(pops[loop].fitness /
                           (sharing ? pops[loop].share : 1.0)); loop+=1)
    where-=fabs(pops[loop].fitness / (sharing ? pops[loop].share : 1.0));

  return(pops[loop]);
}

/***********************************************************************/
/* This fitness function performs a form of tournament selection as in  */
/* the classic marriage problem in dynamic programming. Select one at   */
/* uniform random, then continue for up to tournament_size tries making */
/* uniform random selections: choose the first that exceeds the fitness */
/* of that first-chosen. After tournament_size failures, choose first.  */
/***********************************************************************/

#define ANY (base + (int)(drandom() * (double)range))

CHROMOSOME tm_select(pops,base,range)
CHROMOSOME *pops;
int base, range;
{
  int i, choice;
  double f;
```

```
  int reject = ANY;

  f = pops[reject].fitness / (sharing ? pops[reject].share : 1.0);
  for(i=0;i<tournament_size;i++) {
    choice = ANY;
    if(pops[choice].fitness / (sharing ? pops[choice].share : 1.0) > f)
      return(pops[choice]);
  }
  return(pops[reject]);
}

/**********************************************************************/
/* This fitness function performs straight tournament selection as in   */
/* Brindle, Alberta TR 81-2. Pick tournament_size at uniform random,    */
/* return fittest of those.                                             */
/**********************************************************************/

CHROMOSOME tn_select(pops,base,range)
CHROMOSOME *pops;
int base, range;
{
  int i, choice;
  double f;
  int one = ANY;

  f = pops[one].fitness / (sharing ? pops[one].share : 1.0);
  for(i=0;i<tournament_size-1;i++) {
    choice = ANY;
    if(pops[choice].fitness / (sharing ? pops[choice].share : 1.0) > f) {
       one = choice;
       f = pops[one].fitness / (sharing ? pops[one].share : 1.0);
    }
  }
  return(pops[one]);
}


------------------------------------------------------------------------------
/* init.c

   Copyright (C) 1993, 1994 Peter Ross
   Modified by Alasdair Turner (copyright (C) 1994 Alasdair Turner) Sept 1994

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/

/*
 * This program has been modified for use with MDS solutions code
 *
 * Overview of changes   Alasdair Turner   September 1994
 *
 * MDS code header file is included for MDS functions and variables.
 *
 * init_pops_sorted:
 *   If sharing, then after calculating fitness of population, uses
 *   `mds_init_sharing_pop' of `mds_init_ss_sharing_pop' to calculate
 *   either the standard shared fitness of the population or the
 *   the spatial shared fitness of the population respectively.
 *
 *   When sorting the population, if sharing then uses the
 *   `compchromo_sharing' function with qsort (takes into account the
 *   shared fitness) rather than `compchromo'.
```

```
 *
 * init_pops_unsorted:
 *   If sharing, then after calculating fitness of population, uses
 *   'mds_init_sharing_pop' of 'mds_init_ss_sharing_pop' to calculate
 *   either the standard shared fitness of the population or the
 *   the spatial shared fitness of the population respectively.
 */

#include <stdio.h>           /* Include the IO and maths header files. */
#include <math.h>

#include "pga.h"             /* defines genodata and chromosome structs */

/* PGA-specific stuff */

extern int compchromo();
extern int chromo_id;
extern int ss_task;
extern int geno_size;
extern int max_allele;
extern int mcb_range;
extern double (*evaluate)();
extern double *total_abs_fitness;

/* MDS stuff */

#include "mds.h"

/* C stuff */

extern char *malloc();
extern long random();

/************************************************************************/
/* This initialises a chromosome of random 0s and 1s and .. up to range */
/* INCLUSIVE. So range = 1 for 0s and 1s, range=2 for 0/1/2 etc         */
/************************************************************************/

void init_genotype(gdp,range)
GENODATA *gdp;
int range;
{
  int i;

  gdp->refcount = 1;
  gdp->id = chromo_id++;
  gdp->parent1 = gdp->parent2 = 0;
  gdp->genotype = (char *)malloc(geno_size+1);
  gdp->genotype[geno_size] = '\0';  /* for printing */
  for(i=0;i<geno_size;i++)
     gdp->genotype[i] = '0'+(random() % (range+1));
}

/************************************************************************/
/* This creates and randomly initialises the chromosome populations     */
/************************************************************************/

CHROMOSOME *init_pops_sorted(numpops,numchromo)
{
  int chromo,pop;
  CHROMOSOME *pops;

  pops=(CHROMOSOME *)malloc(numpops*numchromo*sizeof(CHROMOSOME));
  if(pops == (CHROMOSOME *)NULL) {
        fprintf(stderr, "Cannot get enough memory to initialise - goodbye\n");
        exit(1);
  }

  for(pop=0;pop<numpops;pop++) {
    total_abs_fitness[pop] = 0.0;
  }

  for(chromo=0; chromo<numpops*numchromo; chromo+=1) {  /* Initialise the */
    pops[chromo].gdp = (GENODATA *)malloc(sizeof(GENODATA)); /* pops.     */
```

```
    init_genotype(pops[chromo].gdp,max_allele);
    pops[chromo].fitness=evaluate(pops[chromo].gdp->genotype);
    /* total_abs_fitness[n] is total fitness of population n */
    total_abs_fitness[chromo/numchromo] += fabs(pops[chromo].fitness);
  }

  /* If sharing, we calculate the share for each chromosome  */
  /* Note that total_abs_fitness is recalculated.            */

  if (sharing == 1) {
    for(pop=0;pop<numpops;pop++) {
      mds_init_sharing_pop(pops+(pop*numchromo),numchromo,pop);
    }
  } else if (sharing == 2 && ss_task) {
    for(pop=0;pop<numpops;pop++) {
      mds_init_ss_sharing_pop(pops+(pop*numchromo),numchromo,pop);
    }
  }

  if(!ss_task) { /* Don't bother sorting if it's ssgen or ssone */
    for(pop=0; pop<numpops; pop+=1) {
      if (sharing)
        qsort((char *)((int)pops+(pop*numchromo*sizeof(CHROMOSOME))),
            numchromo, sizeof(CHROMOSOME), compchromo_sharing);
      else
        qsort((char *)((int)pops+(pop*numchromo*sizeof(CHROMOSOME))),
            numchromo, sizeof(CHROMOSOME), compchromo);
    }
  }
  return(pops);
}

CHROMOSOME *init_pops_unsorted(numpops,numchromo)
{
  int chromo,pop;
  CHROMOSOME *pops;

  pops=(CHROMOSOME *)malloc(numpops*numchromo*sizeof(CHROMOSOME));
  if(pops == (CHROMOSOME *)NULL) {
        fprintf(stderr, "Cannot get enough memory to initialise - goodbye\n");
        exit(1);
  }

  for(pop=0;pop<numpops;pop++) {
    total_abs_fitness[pop] = 0.0;
  }

  for(chromo=0; chromo<numpops*numchromo; chromo+=1) {  /* Initialise the */
    pops[chromo].gdp = (GENODATA *)malloc(sizeof(GENODATA)); /* pops.     */
    init_genotype(pops[chromo].gdp,max_allele);
    pops[chromo].fitness=evaluate(pops[chromo].gdp->genotype);
    /* total_abs_fitness[n] is total fitness of population n */
    total_abs_fitness[chromo/numchromo] += fabs(pops[chromo].fitness);
  }

  /* If sharing, we calculate the share for each chromosome  */
  /* Note that total_abs_fitness is recalculated.            */

  if (sharing == 1) {
    for(pop=0;pop<numpops;pop++) {
      mds_init_sharing_pop(pops+(pop*numchromo),numchromo,pop);
    }
  } else if (sharing == 2 && ss_task) {
    for(pop=0;pop<numpops;pop++) {
      mds_init_ss_sharing_pop(pops+(pop*numchromo),numchromo,pop);
    }
  }

  return(pops);
}

------------------------------------------------------------------------------
/* help.c
 * Copyright (C) 1993, 1994 Peter Ross and Geoff Ballinger.
```

```
 * Modified by Alasdair Turner (copyright (C) 1994 Alasdair Turner) Sept 1994
 *
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see the file COPYING.
 *
 * Help invoked by -h lag or by command-line foulup.
 */

/*
 * This program has been modified for use with MDS solutions code
 *
 * Overview of changes   Alasdair Turner   September 1994
 *
 * Now includes help for MDS methods:
 *   -D (crowding), -E (sharing), -Y (tribes)
 *
 * Also non-interactive mode has been changed to work like interactive mode,
 * both incorporating new termination condition: help changed for -N and
 * new -X (terminate after <n> static generations) included.
 */

#include <stdio.h>
#include "version.h"

/************************************************************************/
/* Help for the user                                                    */
/************************************************************************/
void pga_help()
{
fprintf(stderr,"%s, version %s\n",TITLE,VERSION);
fprintf(stderr,"   -P<n>    Set number of populations. (5)\n");
fprintf(stderr,"   -p<n>    Set number of chromosomes per population. (50)\n");
fprintf(stderr,"   -n<n>    Set chromosome length. (32)\n");
fprintf(stderr,"   -l<n>    Set # of generations per stage. (100)\n");
fprintf(stderr,"   -i<n>    Set reporting interval in generations. (10)\n");
fprintf(stderr,"   -M<n>    Interval between migrations. (10)\n");
fprintf(stderr,"   -m<n>    Set bit mutation rate. (0.02)\n");
fprintf(stderr,"   -c<n>    Set crossover rate (only for `gen'). (0.6)\n");
fprintf(stderr,"   -b<n>    Set selection bias. (1.5)\n");
fprintf(stderr,"   -a       Adaptive mutation flag (FALSE)\n");
fprintf(stderr,"   -t       Twins: crossover produces pairs (FALSE)\n");
fprintf(stderr,"   -g       In function optimisation, when decoding treat\n");
fprintf(stderr,"              bit pattern as Gray code (FALSE)\n");
fprintf(stderr,"   -C<op>   Set crossover operator. (two)\n");
fprintf(stderr,"   -s<op>   Set selection operator. (rank)\n");
fprintf(stderr,"   -r<op>   Set reproduction operator. (one)\n");
fprintf(stderr,"   -e<fn>   Set evaluation function. (max)\n");
fprintf(stderr,"   -S<n>    Seed the random number generator. (from clock)\n");
fprintf(stderr,"   -N       Non-interactive mode\n");
fprintf(stderr,"   -F<file> Use problem datafile <file> instead of default.\n");
fprintf(stderr,"   -D<op><n> Crowding with crowding factor <n>.\n");
fprintf(stderr,"   -E<op><n> Sharing with niche separation distance <n>.\n");
fprintf(stderr,"   -Y<n>    Tribes with niche separation distance <n>.\n");
fprintf(stderr,"   -X<n>    Stop after <n> generations with no improvement\n");
fprintf(stderr,"   -F<file> Use problem datafile <file> instead of default.\n");
fprintf(stderr,"   -h       Display this information.\n");
fprintf(stderr,"   <file>   Also log output in <file>. (none)\n\n");
fprintf(stderr,"   Crossover operators ... one, two, uniform, none.\n");
fprintf(stderr,"   Selection operators ... rank, fitprop,\n");
fprintf(stderr,"                           tnK (K=integer > 1),\n");
fprintf(stderr,"                           tmK (K=integer > 1).\n");
fprintf(stderr,"   Reproduction operators ... one, gen,\n");
fprintf(stderr,"                              ssoneN (N=integer > 0),\n");
fprintf(stderr,"                              ssgenN (N=integer > 0).\n");
fprintf(stderr,"   Evaluation functions ... max, dj1, dj2, dj3,\n");
fprintf(stderr,"                            dj5, bf6, knap, tt,\n");
fprintf(stderr,"                            tt:E+S (E=r/w/tM; S=r/f/tN),\n");
fprintf(stderr,"                            mcbK (K in 1..9), rr.\n");
}
```

Appendix D

Timetabling Problem Specification

This appendix includes the problem constraints files used to define the `tt10`, `tt30`, `tt60` and `tt120` problems. These form the 10 solution, 30 solution, 60 solution and 120 solution 20 event problems.

The files may be loaded into the PGA program using the `-ett` flag to set the timetabling evaluation function and `-F<file-name>` flag to set the problem constraint file `<file-name>` required.

A full specification is only included for the `tt120` problem. The other problems are all defined the same constraints as in the `tt120` problem *plus* the extra constraints shown in their respective section.

`tt120` definition

```
# ttdata.120
#
# 20 event problem with 5 time slots and 120 solutions

clash-type 2  # don't gripe, but don't allow duplicate constraints

events 20
slots   5
days    1

p-clash   =   1.0
p-order   =   0.0
p-near    =   0.0
p-exclude =   0.0

separate  0 and 4 and  8 and 12 and 16
separate  0 and 5 and  9 and 13 and 17
separate  0 and 6 and 10 and 14 and 18
separate  0 and 7 and 11 and 15 and 19

separate  1 and 4 and  8 and 12 and 16
separate  1 and 5 and  9 and 13 and 17
separate  1 and 6 and 10 and 14 and 18
separate  1 and 7 and 11 and 15 and 19

separate  2 and 4 and  8 and 12 and 16
separate  2 and 5 and  9 and 13 and 17
separate  2 and 6 and 10 and 14 and 18
separate  2 and 7 and 11 and 15 and 19

separate  3 and 4 and  8 and 12 and 16
```

```
separate  3 and 5 and  9 and 13 and 17
separate  3 and 6 and 10 and 14 and 18
separate  3 and 7 and 11 and 15 and 19

separate  4 and  8 and 12 and 16
separate  4 and  9 and 13 and 17
separate  4 and 10 and 14 and 18
separate  4 and 11 and 15 and 19

separate  5 and  8 and 12 and 16
separate  5 and  9 and 13 and 17
separate  5 and 10 and 14 and 18
separate  5 and 11 and 15 and 19

separate  6 and  8 and 12 and 16
separate  6 and  9 and 13 and 17
separate  6 and 10 and 14 and 18
separate  6 and 11 and 15 and 19

separate  7 and  8 and 12 and 16
separate  7 and  9 and 13 and 17
separate  7 and 10 and 14 and 18
separate  7 and 11 and 15 and 19

separate  8 and 12 and 16
separate  8 and 13 and 17
separate  8 and 14 and 18
separate  8 and 15 and 19

separate  9 and 12 and 16
separate  9 and 13 and 17
separate  9 and 14 and 18
separate  9 and 15 and 19

separate  10 and 12 and 16
separate  10 and 13 and 17
separate  10 and 14 and 18
separate  10 and 15 and 19

separate  11 and 12 and 16
separate  11 and 13 and 17
separate  11 and 14 and 18
separate  11 and 15 and 19

separate  12 and 16
separate  12 and 17
separate  12 and 18
separate  12 and 19

separate  13 and 16
separate  13 and 17
separate  13 and 18
separate  13 and 19

separate  14 and 16
separate  14 and 17
separate  14 and 18
separate  14 and 19

separate  15 and 16
separate  15 and 17
separate  15 and 18
separate  15 and 19
```

`tt60` definition

This is comprised of the constraints for the `tt120` problem plus the following constraints.

```
order 0 before 4, 5, 6 and 7
order 1 before 4, 5, 6 and 7
order 2 before 4, 5, 6 and 7
order 3 before 4, 5, 6 and 7
```

`tt30` definition

This is comprised of the constraints for the `tt120` problem plus the following constraints.

```
order 0 before 4, 5, 6 and 7
order 1 before 4, 5, 6 and 7
order 2 before 4, 5, 6 and 7
order 3 before 4, 5, 6 and 7

order 12 before 16, 17, 18 and 19
order 13 before 16, 17, 18 and 19
order 14 before 16, 17, 18 and 19
order 15 before 16, 17, 18 and 19
```

`tt10` definition

This is comprised of the constraints for the `tt120` problem plus the following constraints.

```
order 0 before 4, 5, 6 and 7
order 1 before 4, 5, 6 and 7
order 2 before 4, 5, 6 and 7
order 3 before 4, 5, 6 and 7

order 4 before 8, 9, 10 and 11
order 5 before 8, 9, 10 and 11
order 6 before 8, 9, 10 and 11
order 7 before 8, 9, 10 and 11

order 12 before 16, 17, 18 and 19
order 13 before 16, 17, 18 and 19
order 14 before 16, 17, 18 and 19
order 15 before 16, 17, 18 and 19
```