GENETIC AND EVOLUTIONARY PROTOCOLS FOR SOLVING DISTRIBUTED

ASYMMETRIC CONSTRAINT SATISFACTION PROBLEMS

Except where reference is made to the work of others, the work described in this dissertation is my own or was done in collaboration with my advisory committee. This dissertation does not include proprietary or classified information.

_____

Ser-Geon Fu

Certificate of Approval:

_____	_____
Alice E. Smith Professor Industrial and Systems Engineering	Gerry Dozier, Chair Associate Professor Computer Science and Software Engineering
_____	_____
Kai Chang Professor Computer Science and Software Engineering	Min-Te Sun Assistant Professor Computer Science and Software Engineering

_____

Joe F. Pittman
Interim Dean
Graduate School

GENETIC AND EVOLUTIONARY PROTOCOLS FOR SOLVING DISTRIBUTED ASYMMETRIC CONSTRAINT SATISFACTION PROBLEMS

Ser-Geon Fu

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fullfilment of the

Requirements for the

Degree of

Doctor of Philosophy

Auburn , Alabama
August 4, 2007

GENETIC AND EVOLUTIONARY PROTOCOL FOR SOLVING DISTRIBUTED

ASYMMETRIC CONSTRAINT SATISFACTION PROBLEM

Ser-Geon Fu

_____

Signature of Author

_____

Date of Graduation

DISSERTATION ABSTRACT

GENETIC AND EVOLUTIONARY PROTOCOLS FOR SOLVING DISTRIBUTED ASYMMETRIC CONSTRAINT SATISFACTION PROBLEMS

Ser-Geon Fu

Doctor of Philosophy, August 4, 2007
(M.Sw.E., Auburn University 2001)
(B.S., Chung-Hua University 1999)

193 Typed Pages

Directed by Gerry V. Dozier

Processor speed has been growing at an exponential rate over the past 50 years. Computers are getting smaller, cheaper and faster. Over the past 30 years, with the growth of the internet, new forms of decentralized distributed computing architectures have emerged. The emergence of distributed architectures has led to the creations of distributed computing systems and a new field of research.

Distributed computing studies the coordination of computers, processors, and/or processes that are physically distributed but work towards a common goal. Many of the

fundamental issues involved with distributed computing have been thoroughly researched in the past, for example, synchronization, point-to-point communication, deadlock issues, etc. To date, there is a growing need for the development of applications that can effectively utilize the underlying architecture to solve complex distributed optimization problems. To this end, one can either create a new algorithm specifically for the architecture or modify existing techniques to run on the new architecture. In this work, the latter approach is adopted.

Evolutionary computation (EC) has been shown to be capable of solving complex problems where traditional methods fail to yield satisfactory results. However, to date there has been no research into creating true distributed ECs with distributed genomes. This dissertation presents a set of genetic and evolutionary protocols (GEPs), which are ECs modified to solve distributed problems. To assess their performance of GEPs, we will be testing GEPs on distributed constraint satisfaction problems, where the variables and constraints are geographically distributed among various entities/agents within a distributed system. We will also apply these GEPs to the sensor network tracking problem, and the sensor network sharing problem.

Style Manual or Journal: IEEE Standard

Computer Software used: OpenOffice 2.0

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1. Distributed Architecture and Resource Allocation

Processor speed has been growing at an exponential rate over the past 50 years and computers are becoming smaller, cheaper and faster [152]. Over the past 30 years, with the growth of the Internet, new forms of decentralized, distributed computing architectures have begun to emerge [13, 22], which has led to the creation of distributed computing systems and a new field of research [13]

Distributed computing studies the coordination of computers, processors, and/or processes that are physically separated but work towards a common goal [14, 16, 19, 116, 148]. Examples of distributed systems include multi-processor systems [16], server clusters [16], multi-agent systems [19] and sensor networks [1, 11, 150, 176]. Many of the fundamental issues involved with distributed computing have been thoroughly researched in the past, for example, synchronization [148], point-to-point communication [148], deadlock issues [16], etc. For applications running on distributed systems, the main concern is how to effectively utilize the available resources to complete the task. Thus, distributed resource allocation problems (DisRAP) are central to research in distributed computing [5, 14, 16, 19, 160].

Distributed resource allocation involves the assignment of resources to separate entities/agents such that they can complete individual tasks or help the system, as a

whole, accomplish specific goals [14, 16, 19]. As there are different types of distributed systems, the resources that need to be distributed may vary from being shared among all agents [16] to being separately owned by individual agents [19]. Problems often arise due to the lack of centralized control and the existence of constraints on resources within and/or between agents [7, 30, 38, 122]. The lack of centralized control and distributed constraints mean that DisRAPs cannot be solved with traditional optimization or search techniques. Typically, DisRAPs are solved through negotiation/compromise based techniques [19, 86, 95, 160] or queueing and scheduling techniques [14, 16, 20, 116] depending on the architecture of the system. It has been shown that DisRAPs can be modelled as distributed constraint satisfaction problems (DisCSP) [54]. This makes it possible to solve DisRAPs with the solution methods that were originally developed for DisCSP [54].

1.2. Sensor Networks

A sensor network is a collection of wirelessly connected, low cost pods that contain a number of sensing devices and are deployed over a specific geographical region for any number of purposes [1, 11, 151, 176]. A sensor network can also be viewed as a multiagent system [19]. There are many problems related to the usage and set up of a sensor network [1, 11, 151, 176]. However, this research focuses on two application problems of the sensor network: the fundamental sensor network tracking problem [3, 7, 30, 87, 99, 130] and the new sensor sharing problem [38].

The sensor network tracking problem involves monitoring and following moving targets within the coverage area of a network of stationary autonomous sensing devices

[3, 7, 30, 87, 99, 130]. Each sensor pod has a Doppler radar that is only capable of detecting the relative distance and general direction of a target from itself [3, 7, 30, 87, 99, 132]. Thus, *k* sensor pods must work together and share distance and relative direction information to be able to triangulate and accurately pinpoint the actual position of the target [7]. To effectively track a target, *k* of all sensor pods that can detect the target must be assigned to follow it, but at the same time these *k* sensor pods must also be able to communicate directly with each other to share the relative position data [7]. A target is said to be *k*-trackable [7] if, out of all the pods that are able to detect it, *k* pods that are capable of directly communicating with each other can be assigned to track it.

The sensor network sharing problem involves the allocation of limited sensor resources to satisfy as many user requests for sensors as possible [38]. Each sensor pod contains *m* different sensors, and each pod is capable of turning on or off any number of the *m* sensors that it has. However, in order to reduce the power consumption of the individual pods, assume that only one of the *m* sensors can be turned on in a sensor pod at any given time. Thus, any user can request up to *n* sensors from the sensor network to collect data [38], where *n* is the number of pods in the network. Each request will also have a time value (life span) associated with it that specifies how much sensor time must be allocated to the request to completely satisfy it. When a user places a request for *x* sensors, the network would then need to assign *x* pods to have the specified sensors turned on [38]. In addition to satisfying the user's sensor needs, the network must also satisfy a series of constraints in the form of internal allocations policies for each pod [38]. As more users make requests (and old requests are completed), the network must be able

to dynamically reassign sensors among the pods so as to satisfy as many user requests as possible without violating any of the internal allocation policies.

1.3. Evolutionary Computation

Evolutionary computation (EC) is the study of algorithms and problem solving techniques inspired by the processes of natural evolution [29, 32, 49, 51, 52, 142]. ECs have been shown to find good solutions for relatively hard problems where traditional methods were not able to provide satisfactory results [49]. However, up to now ECs have mainly been used to solve centralized problems. The concept of distributed computing has generally been used as a method to speed up ECs through parallelization of the algorithm with the use of a parallel population [8, 61, 96, 137]. This study presents two new ECs, genetic and evolutionary protocols (GEPs) [41]. The GEPs were created through the addition of distributed crossover and mutation operators to the current best distributed method for solving DisCSPs, Yokoo's distributed breakout algorithm (dBA) [164, 165, 166, 170]. Unlike traditional distributed ECs [8, 61, 96, 137], GEPs use distributed candidate solutions to solve DisCSPs [164, 165, 166, 170] in a truly distributed manner. In order to compare the performance of GEPs to known methods for solving DisCSPs, the GEPs will be tested on randomly generated distributed asymmetric constraint satisfaction problems (DisACSPs) [36, 37], as well as on the sensor network tracking [3, 7, 30, 87, 99, 130] and the sensor network sharing problem[38].

1.4. Outline

The remainder of this dissertation is arranged as follows. In Chapter 2, an in-depth discussion of resource allocation problems, DisRAPs, constraint satisfaction problems, DisCSPs, and some known methods for solving them will be presented. Chapter 3 describes the creation of the GEPs from a known good solution method for DisCSPs, dBA. Chapter 4 presents and discusses the results obtained from testing the GEPs on randomly generated DisACSPs. Chapter 5 illustrates the architecture and issues involved with the sensor network, along with detailed explanations and examples of the sensor tracking and sharing problems. Chapters 6 and 7 present the results obtained from testing the GEPs on the sensor tracking and sharing problems. Finally, Chapter 8 summarizes the study, lists its conclusions, and suggest directions for possible future research.

CHAPTER 2

RESOURCE ALLOCATION AND CONSTRAINT SATISFACTION

2.1. Introduction

The applications used to demonstrate the effectiveness of the genetic and evolutionary protocols (GEPs) are both instances of the dynamic distributed resource allocation problem, namely the sensor network tracking problem [3, 7, 30, 87, 99, 130] and the sensor sharing problem [38]. To gain a better understanding of distributed resource allocation problems, centralized resource allocation problems will be presented first. Since resource allocation problems can be modeled as constraint satisfaction problems [54], this research will be focused on solving the sensor tracking [3, 7, 30, 87, 99, 130] and sharing problems [38] modeled as distributed constraint satisfaction problems [7].

2.2. Resource Allocation

Resource allocation problems (RAPs) involve the assignment or distribution of limited resources to a series of tasks, while at the same time optimizing an objective function [17, 62, 65, 70, 79, 132, 143]. Given that resources are limited, the sum of all allocated resources must not exceed the amount available. So, given an RAP with *n* variables $x_1, x_2, \ldots, x_n$, the solution must satisfy the constraint $\sum x_j \leq N$ [68]. Some RAPs may require that all available resources be allocated, which means a solution must

satisfy $\sum x_j = N$ [67, 70, 132, 143]. Also, since it is not possible to allocate negative resources, all RAPs have a non-negativity constraint such that none of the variables are assigned negative values. Additional constraints may be added to specify certain allocation patterns (policies) or guarantee a minimum amount of resources to a certain variable/task [48, 56, 57, 67, 70, 82, 85, 88, 89, 92].

There are two major types of RAP, continuous and discrete [65, 70, 132, 143]. The difference is mainly in the domain type of the variables involved. Discrete RAPs have discrete variable domains, while continuous RAPs have continuous variable domains. RAPs can further be divided based on two characteristics in the objective function, separability and convexity. An objective function is considered separable if the overall fitness is the sum of the independent local partial fitnesses [65, 70, 132, 143]. Formally, a separable function is one where

$$f(x_{1,} x_{2,} \ldots, x_n) = \sum_{i=1}^{n} f_i(x_i)$$

Convexity is the property where given any objective function *f(x)* and two values *i* and *j*, *f'(i) < f'(j)* if, and only if, *i < j*, where *f'(x)* is the first order derivative of *f(x)*. Separability and convexity are the two most exploited objective function properties in operations research [65, 70, 132, 143], for formulating solution methods for RAPs.

The field of Operations Research (OR) has developed a number of numerical methods, such as the simplex method [27, 65, 69, 77, 118, 132, 133, 135, 143, 153, 154, 158, 169], the interior point algorithm [65, 72], the branch and bound method [65, 70, 112, 157], and gradient-based methods [10, 47], that can be used to solve RAPs. To apply these methods, the RAP must first be formulated into either a linear programming

[27, 65, 70, 72, 118, 132, 135], integer programming [65, 70, 132, 134, 157], dynamic programming [9, 42, 65, 111, 159], or nonlinear programming problem [6, 10, 47, 65, 78, 158]. It should be noted that although the word "programming" is used, this has nothing to do with computer implementations, but rather, the word "programming" is used to refer to "planning" [65, 70, 111, 157]. These four problem formulation techniques are discussed in turn below, along with some techniques and strategies that are commonly used to solve the programming problems.

2.3. Linear Programming

Linear programming [65, 70, 72, 118, 135] is an indispensable tool for operations research and has been used to solve a wide variety of problems, including resource allocation problems. It should be noted that linear programming is not a problem solving method, but rather a standard model by which a problem can be formulated [65, 70, 72, 118, 135]. However, not all problems can be formulated using linear programming (LP). LP works on a specific class of problems, where the fitness/objective function is linear [65, 70, 72, 118, 135]. This linear restriction also applies to all the constraints that may exist [65, 70, 72, 118, 135, 156]. It is also assumed that all resources are separable, meaning variables must have a continuous domain [65, 70, 72, 118, 135, 156], unlike a separable objective function. Mathematically, an LP problem is modeled as follows [65, 70, 118]:

$$
\begin{aligned}
&\textit{maximize} \quad f(X) = C \cdot X \\
&\text{subject to:} \quad AX^T \leq B^T, \\
&\qquad\qquad x_i \geq 0 \quad \forall x_i \in X \\
&\textit{where :} \quad X = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n \\
&\qquad\qquad C = (c_1, c_2, \ldots, c_n) \in \mathbb{R}^n \\
&\qquad\qquad B = (b_1, b_2, \ldots, b_m) \in \mathbb{R}^n \\
&\qquad\qquad A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \quad \forall a_{ij} \in \mathbb{R}
\end{aligned}
\tag{2.3.1}
$$

where X is the vector of all the variables, C is the corresponding coefficients of each variable in the objective function f, A is a matrix of coefficients for each variable within the set of m constraint inequalities, and B is the set of upper bound values for each constraint inequality. Expanding the model, the objective function is clearly separable and can be rewritten as:

$$f(X) = \sum_{i=1}^{n} c_i x_i$$

The m linear constraint inequalities created from $AX^T \leq B^T$ can be expanded as follows:

$$
\begin{aligned}
a_{11} \cdot x_1 + \cdots + a_{1m} \cdot x_m &\leq b_1 \\
&\vdots \\
a_{m1} \cdot x_1 + \cdots + a_{n1} \cdot x_m &\leq b_m
\end{aligned}
$$

This is known as the standard LP model [65, 70, 72, 118, 135, 156] and contains m constraints and n variables.

The most common method used to solve a LP problem is the simplex method [65, 69, 70, 77, 118, 133, 135, 153, 154], which exploits the property that the optimal solution is a corner point of the feasible space. Because the feasible space is enclosed by the boundaries created by the constraints, a corner point is the point at which two or more constraint boundaries intersect. The simplex method starts from any arbitrary corner point

feasible (CPF) solution and moves the candidate solution from one CPF solution to a better neighboring one until the optimal solution is found [70, 118, 154]. The simplex method has been shown to have a worst case exponential time complexity, but an average polynomial time complexity [77, 133].

Another method for solving a LP problem is known as the interior point method [65, 72]. As with the simplex method, the interior point method is an iterative improvement method [65, 72], but rather than starting from a CPF solution, the interior point method starts from a random point within the space of feasible solutions and iteratively moves in the direction that will most improve the objective function until an optimal solution is found [65, 72]. Computationally, per iteration the interior point method is more complex than the simplex method [65]. However, it has been shown that the worst case performance of the interior point method is polynomial time [77, 133].

2.4. Non-Linear Programming

Though LP is a powerful tool for operations research and solving resource allocation problems, it is limited by the linear requirement placed on the objective function and constraints [65, 70, 118]. The formulation of problems using non-linear programming (NLP) is the same as LP with the exception that the constraints and objective function do not need to be linear [6, 10, 47, 65, 78, 158]. Though the general form of NLP is nontrivial to solve [6], solution methods have been found for specific classes of NLP problems. Examples of NLP problem classes that have known solution strategies include linearly constrained optimization, quadratic programming, convex programming, and separable programming [6, 10, 47, 65, 78, 158].

Linearly constrained optimization problems are similar to LP problems with the exception that the objective function is nonlinear [65]. These problems can usually be solved with a modified version of the simplex method used for LP [65]. The interior point method also makes an ideal solution strategy as it does not assume that the optimal solution will be a CPF solution, like the simplex method [6, 65, 72, 158].

Quadratic programming problems [65, 117] are a subclass of linearly constrained optimization problems where the objective function is quadratic (of order 2), while the constraints stay linear [6, 65, 117]. Unlike general nonlinear objective functions, a quadratic objective function is much easier to work with because the global optimal will also be the only local optimal. Quadratic programming problems are preferably solved with barrier and interior point methods [6, 65], but can also be solved with a modified simplex method [6, 65, 117].

Convex programming [63, 129, 140] covers a wide range of problem types. Problems falling under convex programming must satisfy the requirements that the objective function be convex and the constraints be concave [65, 132, 143]. A concave function is the opposite of a convex function and must satisfy the constraint that $\forall i, j \in \mathbb{R}, i < j \rightarrow f'(i) > f'(j)$, where $f'(i)$ and $f'(j)$ are first order derivatives of the function f [65, 70, 132, 143]. As with quadratic programming problems, the characteristics of the convex programming problem guarantees that there will only be one local optimal, which will also be the gobal optimal and unique solution. Convex programming problems can be solved with gradient algorithms [65, 70, 132, 143] or a modified simplex method [169].

Separable programming [62, 65, 78 132, 143] is a special type of convex programming problem with the additional property that the objective function is separable. Separable programming problems are easier to solve than convex programming problems, as the objective function can be approximated by any number of linear functions. This can only be done with separable programming problems because of the objective function's separability. Separability implies that it is possible to break the objective function into a series of single variable functions, which makes it possible to perform linear approximations on each individual variable function. Separability also guarantees that the combined optimal of the individual partial objective functions will be the global optimal. By breaking a nonlinear function into a series of linear functions, the simplex method can be used. Since a series of linear functions are used to approximate the original nonlinear objective function, the accuracy of the approximation can be increased simply by using more linear functions.

As these four examples demonstrate, solutions to NLP problems are highly dependent on the characteristics of the objective function and constraints and tend to be iterative improvement methods. With the wide range of possible problem types for NLP problems, no single solution method or strategy can be used to consistently obtain a solution.

2.5. Integer Programming

Up to this point, solution methods for continuous RAPs have been considered, but many practical problems are discrete. For such situations, integer programming (IP) [65, 70, 132, 134, 157] must be used. Integer programming can be further divided into pure

IP, where all variables require integer value assignments, and mixed IP, where only some of the variables have integer domains [70, 157]. It has been shown that all IP problems can be relaxed and/or transformed into the LP standard form [65]. The main difference is the additional constraint that all or some variables must be assigned integer values.

Pure IP problems may look simpler than LP problems due to the reduction in search space from being continuous (infinite) to discrete (finite). However, typically IP problems are harder to solve than LP problems [65, 70, 157]. The finite search space of a pure IP does not make the problem any easier to solve, as the number of feasible solutions can still grow exponentially with problem size. It should be noted that for an LP problem, the feasible search space is infinite, but only a relatively small subset of this search space is of interest, namely the corner point feasible (CPF) solutions that the simplex method targets [65, 70, 118, 154].

The most common strategy for solving an IP problem is to transform it into standard LP form [157], ignore the integer requirements, and solve it as an LP type problem with the simplex method [65, 70, 118, 132, 154, 157]. This is also known as LP relaxation [65]. However, it is not often that the solution to an LP relaxation will also be integral. If the solution found by the simplex method is not integral, then the cutting plane method [157] can be used to add a new constraint to the transformed IP problem to eliminate any non-integer optimal solution. After the constraint is added, the simplex method is reapplied to the new problem. The repeated process of finding a solution with the simplex method and adding a new constraint is performed until the optimal solution found by the simplex method also satisfies the integer requirements [70, 157]. This procedure is known to be finite [70, 154].

Another approach to solving an IP problem is the branch and bound method [65, 70, 111, 157]. The branch and bound method breaks the problem down into a number of subproblems by selecting a variable and assigning a different value from its domain for each subproblem (branch). Each subproblem is run through the simplex method to estimate the objective function's upper bound. Based on the estimates, the subproblem with the most promise is then further divided. This is repeated until an optimal set of variable assignments is found. In many ways, the branch and bound method is similar to the graph search algorithm A* [127]; where A* uses a distance heuristic to guide its search, the branch and bound method uses the estimate for the objective function's upper bound for each subproblem to guide its search.

As IP problems become larger and more complex, there is no guaranteed deterministic method for solving them [65, 70, 157]. Transforming the IP problem into the LP standard form and applying the simplex method will not guarantee an integer solution [65, 70, 157]. The cutting plane method [70, 157] works, but as the number of cutting planes increase, the problem becomes more complex due to the newly added constraints. The branch and bound method also has its faults, as it is very possible for the branching tree to grow exponentially and consume large amounts of memory [65, 70, 111, 157]. Recently, there have been developments in efficient near optimal heuristics that identify better solutions in shorter amounts of time than LP relaxation and other methods that utilize the simplex method for IP [65, 70, 111, 132, 143, 157]. Some of these new solution methods include the use of algorithms and meta-heuristics from evolutionary computation [63, 82, 90].

2.6. Dynamic Programming

Dynamic programming [9, 42, 65, 111, 159] is a solution strategy that is often used for solving decision problems. Specifically, the problem must consist of stages where a variable is assigned a value or a decision is made at each stage [9, 42, 65, 111, 159]. A common example of this type of problem is the problem of finding the shortest path between two points through a number of intermediate points [9] otherwise known as the traveling salesman problem (TSP) [65, 132, 143]. Unlike LP, where a standard form bis used to formulate the problem, dynamic programming (DP) does not have a standard form [9, 42, 65, 111, 159]. Thus, formulating a problem in DP form often requires some ingenuity [65].

DP can be further subdivided into deterministic dynamic programming and probabilistic (stochastic) dynamic programming [9, 42, 65, 111, 159]. Deterministic DP is used to solve problems where the choice at a specific stage will lead to a specific result, as with the aforementioned path finding problem [9, 111]. Probabilistic (stochastic) DP is used to solve problems where specific choices will probabilistically lead to varying outcomes [114]. In these cases, the goal would be to optimize the expected outcome [65, 114]. The branch and bound method is the primary solution algorithm for solving DP problems [9, 42, 65, 111, 159].

2.7. Resource Allocation and Evolutionary Computation

Though the deterministic methods presented above are useful and generally provide good results, they are extremely limited as to the types of problems they can address. In the case of DP, the strategy itself does not scale well to larger problems [42, 159].

Though LP has a general solution method, the simplex method, many practical application problems cannot be formulated into the LP standard form [6, 10, 47, 65, 78, 158]. In the case of NLP, there is no unique solution method for the general form, as with the simplex method for LP [6, 65, 70, 78, 132, 143, 158].

Evolutionary computation is the study of algorithms and problem solving techniques that are inspired by the processes of natural evolution [29, 49, 142]. The most widely used evolutionary computations (ECs) include genetic algorithms (GA) [29, 142, 107], evolution strategies (ES) [4, 51, 126], evolutionary programming (EP) [50-53], genetic programming (GP) [51, 80], particle swarm optimization (PSO) [74, 75], ant systems (AS) [32, 34], and ant colony optimization (ACO) [32, 33]. Recent research has shown that ECs are often able to find solutions to complex problems for which traditional methods are unsatisfactory [29, 49, 142].

2.8. Constraint Satisfaction

Resource allocation problems can be modeled as constraint satisfaction problems (CSP) [108, 109], which are composed of a set of variables, *X*, a set of domains, *D*, for each variable, and a set of constraints, *C*, limiting the assignments of values to each variable. The goal is to find a set of value assignments for *X* from *D* such that no constraints in *C* are violated [28, 39, 40, 43, 94, 106, 125, 129, 141, 161]. The constraints in *C* may come in many different forms, possibly constraining multiple variables simultaneously [28, 161]. However, all constraints can be decomposed into a set of binary constraints, where a binary constraint is one that involves only two variables [28]. With binary constraints, a CSP can be illustrated as a graph, a constraint network

[28, 161]. In a constraint network, all variables in X are vertices in the graph, while the edges between vertices represent constraints [28, 161]. Thus, for a constraint network $G(V, E)$ containing the set of vertices V and the set of edges E, $V = X$, an edge $E_{i,j}$ between V_i and V_j exists if, and only if, $\exists C_{i,j} \in C$.

There are two additional properties of a CSP, namely constraint density and constraint tightness [28, 39, 40, 92, 161]. The constraint density is the ratio between the number of constraints in the network and the total number of possible constraints. The constraint tightness represents the probability that a value assignment pair of two variables is not allowed if there exists a constraint arc (edge) between the two variables (vertices). The constraint density can be calculated by dividing the number of existing edges by the total number of possible edges, which is $n(n-1)/2$ where n is the number of vertices in the graph. Given two variables x_i and x_j and their corresponding domains d_i and d_j, the constraint tightness is found by dividing the number of no-goods between x_i and x_j by $\text{size}(d_i) \times \text{size}(d_j)$.

In the constraint network definition described so far, it should be noted that all constraints are symmetric in nature. Each arc (edge) imposes a constraint on both the vertices it is connected to. Symmetric constraints are considered public, since both variables involved know of the existence of the constraints. An example of a constraint network with symmetric constraints is given in Figure 2.1.

Constraints may also be asymmetric. A constraint network with asymmetric constraints is shown in Figure 2.2, where asymmetric constraints are represented by directional arcs (directed edges) connecting vertices within the constraint network. The directional arcs represent imposed constraints. Thus, in Figure 2.2, node B imposes

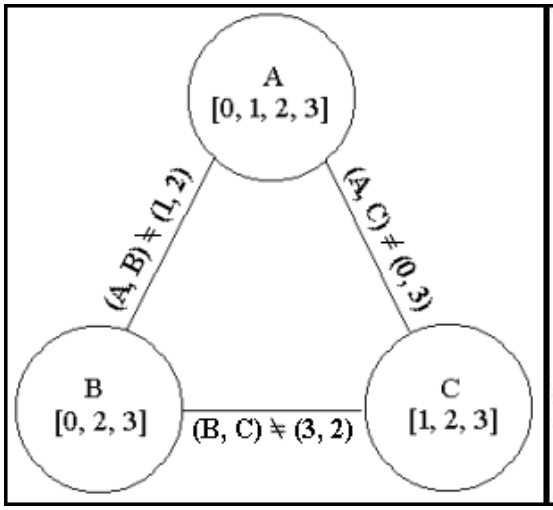

Figure 2.1. Constraint Network with Symmetric Constraints

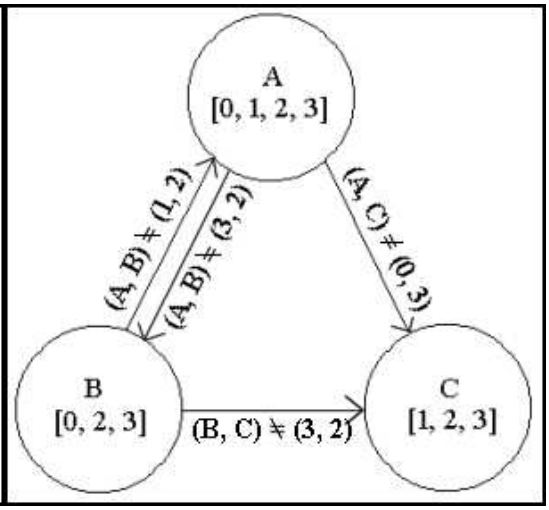

Figure 2.2. Constraint Network with Asymmetric Constraints

constraints on the values of A and C. However, since the arcs are directional, vertices A and C have no knowledge of such constraints. This type of constraint is also known as a private constraint [52], since only the variable imposing the constraints has knowledge of them. CSPs with asymmetric constraints are known as asymmetric CSPs (ACSPs). In general, ACSPs are harder to solve than CSPs because of the presence of the private constraints [36, 54].

Not all CSPs are solvable [93, 139]. It is possible that a CSP can become over constrained, causing it to have no solution [139]. For CSPs, the boundary where problems have on average one solution and beyond which problems may have no solution is called the phase transition [139]. It has been shown that the phase transition can be defined by the constraint tightness in terms of the number of variables, domain size, and constraint density [139]. The equation for the phase transition [139] is defined as follow, where *n* is the number of variables, *m* is the domain size, *p1* is the constraint density, and $p2_{crit_s}$ is the constraint tightness for the phase transition:

$$p2_{crit_S} = 1 - m^{\frac{-2}{p1(n-1)}} \tag{2.8.1}$$

This can be derived from the following equation that estimates the number of possible solutions (*S*) in a randomly generated CSP [93]:

$$S = m^n (1 - p2)^{p1 \cdot n \cdot (n-1)/2} \tag{2.8.2}$$

The phase transition for the value *p2* in terms of *n*, *m*, and *p1* is when *S* = 1. So, solving for *p2* for the case *S* = 1 will result in 2.8.1. The phase transition given in 2.8.1 is for symmetric CSPs. For asymmetric CSPs, since the total number of possible arcs between nodes for a graph with *n* nodes is $n \cdot (n-1)$ rather than $n \cdot (n-1)/2$, the equation used to estimate the number of possible solutions becomes:

$$S = m^n (1 - p2)^{p1 \cdot n \cdot (n-1)} \tag{2.8.3}$$

From this equation, it can be derived that the phase transition for ACSPs is as follow, where $p2_{crit_a}$ is the constraint tightness for the phase transition [37]:

$$p2_{crit_A} = 1 - m^{\frac{-1}{p1_a(n-1)}} \tag{2.8.4}$$

2.9. Solving Constraint Satisfaction Problems

CSPs have been shown to be NP complete [164], so solving large CSPs is nontrivial. There are a variety of algorithms that can be used to solve CSPs [39, 40, 43, 106, 125, 129, 141, 164], including two classes of deterministic methods, namely iterative improvement and backtracking algorithms [39, 40, 43, 106, 125, 129, 141, 164].

Backtracking algorithms are initiated by putting all the variables in a specific order and, starting with the first variable, iteratively assigning values to the variables while trying not to violate any constraints with variables that have already been assigned values

[164]. When a variable, X_k, cannot be assigned a value that does not violate any constraints with previously assigned variables, X_1 to X_{k-1}, backtracking is triggered. The algorithm will try to change the value of the previously assigned variable, X_{k-1}. If no new value can be found, then the algorithms rolls back again until a variable is found that can be changed. A backtracking algorithm is a depth first search of all the possibilities and is thus complete [127]. If a solution exists, it will find the solution, and if one does not exist, it will be able to determine that [164].

Backtracking searches can suffer from thrashing, where an earlier variable assignment creates a situation where there is no feasible value assignment for a variable later in the search. However, in order to identify and change the variable that is causing the conflict, a large number of backtracks and unnecessary searches must be performed. To remedy this problem, arc-revision and arc-consistency algorithms [127, 164] can be added to backtracking to eliminate the infeasible values from domains of unassigned variables and thus reduce the amount of wasted searches and backtracking [84, 164, 167]. The order by which variables are assigned values can also be changed based on how constrained each variable is [84, 164, 167].

Iterative improvement algorithms start with an arbitrary set of variable assignments (a candidate solution) that may contain multiple constraint violations. The algorithm then iteratively changes the value assignment of the variables to reduce the number of constraint violations [39, 42, 43, 106, 125, 129, 141]. These algorithms are usually incapable of determining whether a feasible solution actually exists [39, 40, 43, 106, 125, 129, 141]. One example of an iterative improvement algorithm is the min conflict Hill-climber [165].

There is also a hybrid method, the weak commitment search [164, 165 166]. As with to bracktracking algorithms, this builds solutions one variable-value assignment at a time. However, like iterative improvement algorithms all variables are given an initial random value. If the initial assignment contains no conflicts, then that is the solution. Variables are chosen based on the number of conflicts their assigned initial value has with other variable assignments. The variable with the most conflicts will be chosen first, assigned a new value that minimizes the conflicts and placed into a partial solution. Any new variable-value pair placed into the partial solution must also not conflict with variable-value pairs already in the partial solution set. When no more variables can be assigned a new value that does not conflict with those already in the partial solution, then backtracking occurs. Instead of simply removing the variable-value pair that was last placed into the partial solution set like normal backtracking, the entire partial solution is discarded and the process starts over. However, though all variable-value pairs are discarded from the partial solution set, they become the initial values for the next iteration. So, iteratively, the initial solution used to build the partial solution will improve after each backtrack. In this sense, it is very much like an iterative improvement algorithm.

ECs have also been sutilized to solve CSPs with great success [39, 40, 43, 106, 125, 129, 141]. Some of the ECs used include genetic algorithms [43 64], ant colonies [129, 141], and evolutionary/arc-consistency hybrids [40], as well as evolutionary iterative improvement hybrids [39].

2.10. Distributed Resource Allocation Problems

With the development of distributed architectures, one of the first problems that arises on the application level is how to utilize the available resources, either shared or privately owned, to accomplish the given task [14, 16, 19, 20, 95, 108, 109, 116, 121, 160]. The acquisition of resources required to complete tasks, while competing with other processes for the limited resources, is one of the main issues that affect distributed RAPs [14, 108, 109, 160]. One of the primary characteristic of the distributed RAP (DisRAP) is the lack of centralized control [14, 108, 109, 160]. A DisRAP can be formalized as a problem having a set of agents, a set of resources that are either shared or distributed among agents, and a set of tasks/requests for resources that need to be satisfied [108, 109]. The goal is to formulate an assignment of resources to specific agents or tasks to satisfy the given requests.

There are three types of DisRAP: those studied in multi-agent systems [19], those studied in distributed and parallel computing [16], and those studied in systems such as sensor networks [7]. They differ by whether the resources are owned by individual agents or shared between groups of agents and whether the tasks/processes are internal to the agent or external. A task is considered to be internal to an agent when the completion of the task is the sole responsibility of the specific agent, and to accomplish this it must acquire resources that are owned by other agents or shared with other agents. A task is external when it is one that is shared among or assigned to all agents, and the agents strive to complete this task cooperatively.

The primary focus for multi-agent systems (MAS) [19] has been generally on problems where resources are owned by the individual agents and the tasks are internal to

an agent [160]. For DisRAPs studied in distributed and parallel computing [16], the resources are shared among overlapping groups of agents, with each agent striveing to complete its own internal tasks by gathering all the necessary resources as fast as possible [116]. This problem is best summarized by the formulation of the so-called drinking philosopher problem [16]. The problems of interest here, namely the sensor network sharing [38] and tracking problems [7, 30, 108], both belong to the type where tasks are external, with the agents each possessing their own resources [108, 109]. A possible fourth type of DisRAP is where agents share resources and respond to external requests. However, this scenario cannot be considered a true DisRAP as it can be easily reformulated as a centralized problem.

2.11. Solving Distributed Resource Allocation Problems

For the first type of DisRAP with MAS, Wu et al [160] gives a thorough review of the many deterministic methods for solving these DisRAPs from a production research point of view. They divide the solutions methods into three types: (1) market, (2) compensation, and (3) coalition formation. The market based approach views the various resources as marketible commodities on the market that are valued by their importance and quantity. The processes needing the resources then bid for the resources. This approach has the specific drawback of requiring a single location to keep track of all resource costs, which creates an inherent bottleneck. The compensation approach has processes that provide some sort of compensation to the provider of the resource. This approach assumes that it is the individual agents that are seeking the resources. The coalition formation approach encourages agents to form beneficial coalitions for more

efficient and profitable usage of their resources. These approaches have all been successfully used in many real life applications [160], but are quite different from the EC based approach that will be adopted for this research.

For DisRAPs on distributed and parallel computing systems, the solution methods have generally focused on queueing methods [16, 20, 116]. The main issues that need to be resolved for any solution on such systems is the prevention of deadlock and starvation [16, 20, 116]. Deadlock is created when multiple processes obtain part of the resources they need and the must wait for other resources to become available. This creates the possiblity of a cyclic wait, where process A waits for process B to release the required resources, while B is also waiting for A to rekease the resources it needs [16, 20, 116]. Starvation occurs when a process never gets the resources it needs, either because none of the required resources is ever available or because the process is constantly being pre-empted by other processes [16, 20, 116]. Any solution method must at least prevent these two situations from acurring and, at the same time, reduce the wait time needed to gain access to all required resources to complete a task [16, 20, 116].

For the third type of DisRAP, which can be used for sensor networks, it has been shown in [7], [108] and [109] that they can be mapped to distributed constraint satisfaction problems and solved as DisCSPs. More details on DisCSPs and solution methods for DisCSPs will be presented in the next solution.

2.12. Distributed Constraint Satisfaction Problems

Just as RAPs can be reformulated as CSPs, DisRAPs can be reformulated as DisCSPs [108, 109]. DisCSPs are CSPs with the addition of a set of agents *A* among

which the variables, domains and constraints from *X*, *D* and *C* are distributed [164, 165, 166]. The distributed CSP is a type of problem and should not be confused with distributed/parallel methods that may be employed to solve CSPs. Thus, the goal, as with a standard CSP, is to create a value assignment for each agent/variable such that no constraints are violated. Due to the distributed nature of the problem, this will have to be accomplished through message passing among agents [164, 165, 166].

Typically, in a DisCSP, each agent holds exactly one variable [164, 165, 166]. However, the cases where an agent holds multiple variables can be easily handled by either finding the set of solutions to the local CSP and using that as the variable domain for the local agent or by simply creating virtual local agents to handle one variable each [164]. With the distribution of one variable per agent, the asymmetric version of the CSP can be modeled much more accurately, since the private constraints can now be stored on a per agent basis [54].

2.13. Solving Distributed Constraint Satisfaction Problems

Distributed CSPs must be solved through the use of message passing among agents. Two algorithms that we used for solving standard CSPs can be modified to solve DisCSPs, Leading to the asynchronous backtracking (ABT) algorithm [164, 167] and the asynchronous weak commitment (AWC) search [164, 167]. Unlike their centralized counterparts, which assign values to variables one at a time, these two algorithms assigned values to all variables simultaneously. After the values are assigned, they are sent to all neighbors to check for constraint violations. The assignment and reassignment process of the variables takes place asynchronously as the agents pass relevant

information among each other. The asynchronous weak commitment search has been shown to greatly outperform asynchronous backtracking [164].

Distributed breakout [164, 166] (dBA) is an iterative improvement method used for solving DisCSPs that is based on Morris' breakout method [112]. The dBA starts by assigning random values to all the variables. During, each iteration, a variable will communicate its value with its neighbors to calculate the number of conflicts and possible improvements, which is also communicated with its neighbors. Based on this information, the variable that can resolve the most conflicts will be allowed to change. In terms of computational complexity, dBA is more computationally intensive than either ABT or AWC. However, it has been shown that dBA is capable of performing better than AWC and ABT on critical problems [164]. The dBA approach will be discussed in more detail in the next Chapter, as it is the basis for the genetic and evolutionary protocols that will be presented.

The distributed stochastic algorithm (DSA) is a recently developed algorithm for solving DisCSPs [170, 171]. In some ways it is similar to the dBA, except with the distinct property that more than one variable may change value in each iteration [170, 171]. It also has a slightly lower communication overhead compared to the dBA. Once again, DSA is an iterative improvement method. Starting from a random initial set of variable assignments, the agents communicate values and calculate conflicts and improvements. Each variable that is capable of changing its value to reduce the number of local conflicts is allowed to change with a probability p. This makes it possible for multiple variables to change at the same time and then possibly escape from any local optimum. Here, genetic and evolutionary protocols based on DSA will be the main

competitor to the genetic and evolutionary protocols created from dBA. The results will show that DSA's ability to have multiple variables change at once initially gives it an advantage over dBA, but this becomes a weakness as the problems become harder.

CHAPTER 3

GENETIC AND EVOLUTIONARY PROTOCOLS

3.1. Introduction

This chapter introduces and describes the genetic and evolutionary protocols (GEPs). Yokoo's distributed breakout algorithm (dBA) [164, 165, 166, 170, 171] will be discussed in depth, along with the modifications performed to enhance its performance. The Society of Hill-Climbers (SoHC) [35, 37], which is a further modification to the dBA, will also be discussed. Afterwards, the distributed crossover and mutation operators [36, 41] used to create the GEPs from the SoHC will be described. The distributed stochastic algorithm (DSA) [171, 172] will also be presented, along with the modifications needed to create GEPs with DSA as a basis rather than dBA.

3.2. Distributed Breakout Algorithm (dBA)

The dBA was developed by Yokoo [164, 165, 166, 170, 171] to solve distributed constraint satisfaction problems (DisCSP). The dBA uses message passing to implement a distributed steepest descent hill-climber [164]. In order to prevent the hill-climber from becoming trapped at a local optimum, the breakout method [112] is used to modify the fitness space such that the search can escape from any local optimum.

Each agent begins by randomly choosing a value from the given domain for the variable it holds, which is sent to all neighbors of the agent. The agent then waits to

receive the values sent out by its neighbors to build an agent view. Based on the agent view, the agent can then calculate how many constraint violations, plus breakout violations, it is currently in, along with the most constraint violations it can resolve by changing its value, which is also known as the gain. When calculating the gain, the value from the domain that can resolve the most constraint conflicts is also found. This value is referred to here as the next best value. After calculating the local conflicts and gain, these two values are sent to all the neighbors. The agent then once more waits to receive the conflicts and gains of its neighbors in order to build its conflict and gain view. Based on the gain view, if the local agent can resolve the most conflicts, then it is allowed to change to its next best value. If none of the agents can resolve any of the existing conflicts, then it is assumed that a local minimum has been reached and breakout entries are created and/or incremented. All agents then send their current value to their neighbors again and the whole process continues until all conflicts have been resolved. Thus, each iteration of the algorithm consists of two communication cycles. The pseudo-code for the dBA can be seen in Figure 3.1. This code is executed on all agents simultaneously.

The breakout management mechanism (BMM) stores a list of breakout entries. Each entry is composed of a 4-tuple and a weight or penalty value. The 4-tuple in a breakout entry stores a no-good, which is composed of the two variables and their corresponding values that are considered a no-good. Thus, a breakout entry of $(\langle var_i, var_j, value_i, value_j \rangle, 1)$ adds an extra penalty of 1 for violating the no-good where var_i is assigned $value_i$ and var_j is assigned $value_j$. The breakout list is stored in a distributed manner, allowing each distributed agent to store the entries related to its own

1. *address* = LOCAL_ADDRESS()
2. *value* = RANDOM(*Domain*)
3. $\forall i \in neighbors$ do $SEND(value, i)$
4. WAIT()
5. $\forall i \in neighbors$ do $RECEIVE(i, \text{agent\_view}_i)$
6. $conflict = \text{CALCULATE\_CONFLICT}() + \text{BREAKOUTS\_VIOLATED}()$
7. $gain = \text{CALCULATE\_GAIN}()$
8. $\forall i \in neighbors$ do $SEND(conflict, gain, i)$
9. WAIT()
10. $\forall i \in neighbors$ do $RECEIVE(i, \text{conflict\_view}_i, \text{gain\_view}_i)$
11. $\forall i \in neighbors$ *If* $(gain \geq \text{gain\_view}_i)$ *then do*
    If $(gain > 0)$ *then* $value = \text{NEXT\_BEST\_VALUE}$
    else INCREMENT_BREAKOUT()
12. Do Step 3 – 11 Until *conflict* = 0

Figure 3.1. dBA Pseudo-code

no-goods. Thus, when an agent needs to create breakouts, it creates a breakout entry for each no-good that its value is currently violating. If an entry already exists, the weight is incremented instead. In this manner, the weight/penalty for each breakout alters the fitness space such that searches can escape or avoid any local optima.

Modifications have been made to dBA to enhance its performance [164]. One of Yokoo's modification was the use of broadcasting [164]. With broadcasting, each agent does not just send information to its direct neighbors, but also to all other agents. Broadcasting has the added benefit of letting each agent calculate the global fitness. The dBA with broadcasting (dBA+BC) has been shown to perform better than the standard dBA on critical problems [164], although on sparse problems dBA+BC performs worse than dBA [160]. A major downside of dBA+BC is the added communication overhead.

The dBA+BC algorithm was further modified by Dozier [35] to enhance its performance. Sliding was added to the dBA to improve its performance and further help it escape from a local optimum. Sliding allows a random variable to change its value

when the search is stuck at a local optimum. This improves performance by allowing variables that may not be involved in any conflicts to change their value and possibly move out of the current local optimum. A random search through the variable domain was also implemented when looking for the next best value and maximum gain in order to prevent possible cycling of the search.

3.3. Society of Hill-Climbers (SoHC)

The dBA can be considered as using a distributed candidate solution to implement a distributed hill-climber. The logical representation of the distributed candidate solution (dCS) can be seen in Figure 3.2 Each agent carries its own breakout list stored within a Breakout Management (BMM) structure. Even with this breakout mechanism, however, to gradually escape from a local optimum, dBA still suffers from the same problem as a normal hill-climber: If the initial starting point for the hill-climber is far from the actual solution, it will take more time to find the solution. The breakout mechanism only guarantees that the search will eventually escape a local optimum, but how long this takes depends on the local optimum. These two factors greatly affect the performance of the dBA.

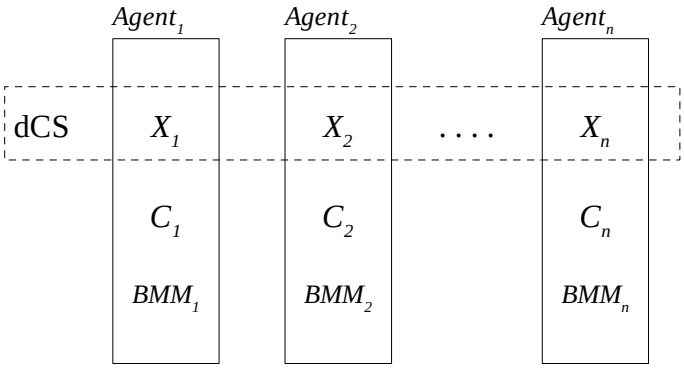

Figure 3.2. A Distributed Candidate Solution

The society of hill-climbers (SoHC) [33, 35] increases the performance of the dBA by overcoming these two problems. Instead of using a single distributed candidate solution (dCS), SoHC uses a population of dCSs. This effectively creates a distributed population, as shown in Figure 3.3. Each agent now carries a total of *k* instances of the variable they hold, where k is the population size. This is equivalent to having *k* copies of the dBA, or in this case the modified dBA+BC (mdBA) [35], running in parallel. The biggest difference here from running *k* mdBAs in parallel is that all instances share the same BMM. This is clearly seen in the figure, where although each agent holds *k* instances of the variable X_i, each agent still has only one breakout management mechanism.

The population-based approach spreads the search out and prevents the search from being trapped in a local optimum for too long. This approach also makes it possible to identify many local optima in parallel. The shared BMM facilitates the indirect

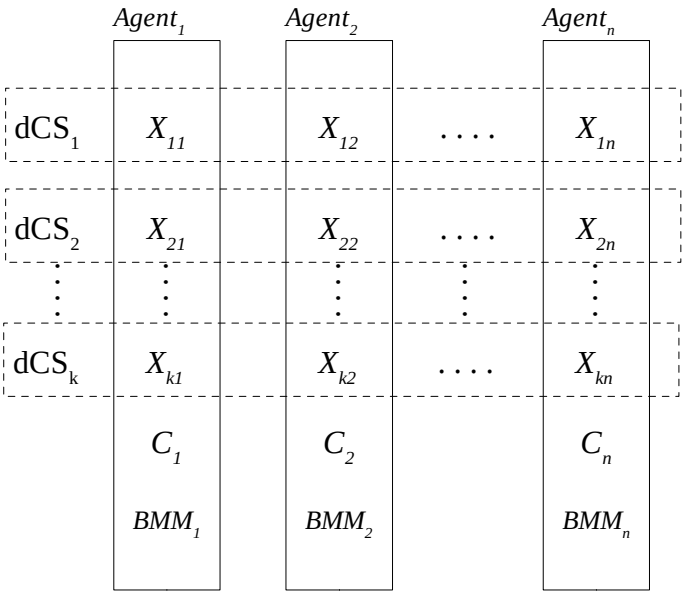

Figure 3.3. A Distributed Population

communication of known local optima, and no-goods associated with many local optima will be penalized more severely within a shorter amount of time. This helps individuals trapped in a local optimum to escape more quickly.

There are definite overheads involved with such a modification. First the computation time required locally on each agent is increased by a factor of *k*. Though the number of packets sent is still the same, more information is included in each packet; instead of sending one local value, one conflict count, and one maximum gain, the packets must now contain *k* values, *k* conflict counts and *k* gain values. Given the overall increase in performance, these drawbacks are acceptable and minimal. The increase in the amount of information per packet also increases link utilization.

3.4. Genetic and Evolutionary Protocols (GEPs)

GEPs are further modifications of the SoHC algorithm based on the addition of genetic and evolutionary operators [36, 40]. The successful application of the distributed genetic and evolutionary operators in GEPs makes two system assumptions, namely that a global view is available and that each agent has a random number generator that is seeded in exactly the same way and is used the same number of times.

The genetic operator was added to SoHC to create GSoHC, the genetic protocol. The genetic operator is a combination of distributed genetic crossover and mutation. The form of genetic crossover used is a uniform crossover. Research has also been carried out with single point, two point, and multi-point crossover [41], but the results show no significant difference in performance. Uniform crossover is therefore used as it is the simplest to implement and all crossover operations can be decided and executed locally

for an agent. For each iteration, once the conflict view has been built (after step 10 in Figure 3.1) the fitness for each individual dCS can be calculated by adding up the corresponding conflict views. Based on the calculated fitness, the distributed individuals with better than average fitness will perform the normal hill-climbing routine while the remainder execute the genetic operator and are replaced with recombinations of the best individual. Thus, only half the population effectively performs hill-climbing each iteration.

When the genetic operator is invoked, the agent will take the variable instance of the invoking dCS and replace it with the value of the dCS with the best fitness with a probability p_c. The variable instance will take on a random value from the domain with probability p_m. Finally, the value of the variable instance will stay unchanged with probability $1-(p_c+p_m)$.

The evolutionary operator is triggered in a similar way to the genetic operator. The evolutionary operator was added to SoHC to create ESoHC (evolutionary protocol), and does not use crossover, but only mutation. Thus, a dCS with below average fitness will trigger the evolutionary operator (after Step 10 in Figure 3.1). Once triggered, the agent will take the value of the variable instance of the dCS that called the evolutionary operator and replace it with the value of the dCS with the best fitness with probability 1 – p_m. With probability p_m, the value is replaced by a random value from the domain.

The GEPs greatly improve the performance of SoHC by adding elements of exploitation. Exploration and exploitation are behaviors that were added to the ant colony optimization (ACO) [32] to either diversify the search into un-searched regions or concentrate the search around regions that are known to be good. Here, the terms are

used as qualitative meassures in order to describe the search behavior of the algorithms (protocols). An algorithm that uses more exploration will have a more diverse population of candidate solutions that are more spread out in the search space, while an algorithm that uses more exploitation will have a population that is more concentrated around a region known to be good with relatively higher fitness. For example, comparing the two approaches, SoHC uses the most exploration, while ESoHC uses more exploitation. ESoHC can be seen as having half the population performing exploration and the other half exploiting the best candidate solution. It should be noted that the size of the population used also affects the level of exploration performed, as a larger population is able to diversify the search more than a smaller population.

By adding elements of exploitation to SoHC with genetic and evolutionary operators, it becomes much easier for candidate solutions to escape from a local optimum or areas of less promise. GSoHC does not perform as much exploitation as ESoHC, as each variable in the below average individuals only has a p_c chance of taking on the corresponding value of the best individual, while staying unchanged with a chance of p_c. ESoHC exploits the current best solution more than GSoHC by intensifying the search closer to the best individual, as each agent of the below average candidates takes on the value of the best with probability $1 - p_m$, where p_m is usually a relatively small value. Thus, even though GSoHC and ESoHC only have half their population doing an actual hill-climbing move, they are capable of finding solutions faster on critically hard problems.

Other modifications of SoHC have also been explored in the past [55]. The ant-like society of hill-climbers [55] (ASoHC) was one modification that specifically tried to

reduce the space complexity created by the BMM for hard problems. Apart from the possible space complexity of the BMM, the other concern was that the breakout entries may effectively partition the fitness space into portions, making it difficult for the search to jump from one region to another. The primary modification for ASoHC was the use of a decaying breakout penalty for each entry. The breakout penalty decay mimics the local and global update rules of the ant colony optimization (ACO) [32, 33]. The same decay was also added to ESoHC and GSoHC resulting in a lower space complexity and a generally shorter breakout list with no significant change in performance [55].

3.5. Distributed Stochastic Algorithm (DSA)

The distributed stochastic algorithm was developed by Weixiong Zhang [171, 172] to solve distributed resource allocation and constraint satisfaction problems. DSA's primary characteristic is its inherent parallelism, which allows multiple agents to change their value in a given iteration [171, 172]. In many ways, it is similar to dBA because it also uses communication to negotiate value changes towards a satisfactory solution. Like the original dBA, each agent running DSA only sends the value currently assigned to the agent to its neighbors. Where dBA also sends the number of constraint conflicts and the maximum gain to its neighbors, DSA keeps these values local. Thus, on the packet count level, DSA only sends half the amount of packets as dBA. The pseudo-code for DSA is given in Figure 3.4.

The pseudo-code shows that DSA is similar to dBA up to step 6. When *gain* is calculated, the next best value is also found, similar to dBA. In dBA, the *conflict* and *gain* are sent to the neighbors, while in the DSA, the *conflict* and *gain* are used, instead,

to determine whether *value* will be changed to the next best value. Whether the next best value is assigned to *value* is also determined by *p*, which specifies the degree of parallelism, or more simply, the probability that *value* will be changed. Five models were presented to show the different conditions under which *value* is changed. These models are presented in Figure 3.5. For DSA-A, *value* is changed with probability *p* if *gain* is greater than 0. For DSA-B, *value* is changed with probability *p* if either *gain* or *conflict* is greater than 0. For DSA-C, *value* is changed with probability *p* no matter what the *gain* and *conflict* are. DSA-D has *value* change if *gain* is greater than 0, or, else, *value* will change with probability *p* if *conflict* is greater than 0. Finally, DSA-E has *value* change when *gain* is greater than 0, or else, *value* will change with probability *p*. Tests on graph coloring problems, carried out by Weixiong Zhang have shown that DSA-B is the most stable and best performer of the five models [172].

What the *p* value implies is that multiple agents are capable of changing their values simultaneously. As will be shown later, this property works both to the DSA's advantage and disadvantage. The ability to have multiple agents change their values also explains why DSA does not need a mechanism similar to the breakout method used by the dBA to escape local optima. The algorithm is capable of probabilistically jumping out of a local optimum when multiple agents choose to change their values. However, this also means

1. *value* = RANDOM(*Domain*)
2. $\forall i \in neighbors \quad do \quad Send(i, value)$
3. Wait.... $\forall i \in neighbors \quad Receive(value_i, i)$
4. *conflict* = CALCULATE_CONFLICT()
5. *gain* = CALCULATE_GAIN()
6. *value* = ASSIGN_NEW_VALUE(*p*, *conflict*, *gain*)
7. Repeat steps 2 – 7 until all conflicts resolved

Figure 3.4. DSA Pseudo-Code

Model	*gain* > 0	*conflict > 0; gain = 0*	*conflict* = 0; *gain* = 0
DSA-A	*value* with *p*	--	--
DSA-B	*value* with *p*	*value* with *p*	--
DSA-C	*value* with *p*	*value* with *p*	*value* with *p*
DSA-D	*vvalue*	*value* with *p*	--
DSA-E	*vvalue*	*value* with *p*	*value* with *p*

Figure 3.5. Differing Models for DSA [172]

that the algorithm is capable of probabilistically jumping away from promising solution regions. Thus, on critically hard problems, the DSA may not converge very well. As the performance of DSA is determined by the *p* value, its performance may improve if the *p* value is varied with time. This possibility will also be explored in the testing phase of this study.

3.6. Modifications to DSA

As dBA was used in comparison to DSA in [171], DSA will be compared to the various dBA modifications created for this study, namely SoHC, GSoHC, and ESoHC. However, since SoHC, ESoHC, and GSoHC have an automatic population advantage [35, 36, 37, 39, 55], similar modifications will be made to the DSA in order to match those changes made in the dBA to create the GEPs. Since DSA-B is the best performing algorithm of the five models, based on [172], this will be used as the basis for the DSA based GEPs created for this study.

First, as for dBA, the simple DSA was given a distributed population. Unlike the SoHC, the distributed individuals in the society of DSA (SoDSA) do not share any information with each other about the search. Also, even though the original DSA did

not need a global view, a global view was added to facilitate the process of determining whether a distributed individual has found the solution. This global view also requires that at least the *conflict,* which is calculated, also be broadcast so that each agent can calculate the fitness for each distributed individual. The communication of the calculated conflict by each agent also makes it possible for DSA to support asymmetric constraints, and these constraints will be the focus of this study. To date, DSA has primarily been used to solve problems with symmetric constraints [171, 172]. Though this may be seen as a retarding factor for DSA to have to send an extra packet each iteration and create a global view, the results show that DSA also benefits from the population based approach as well.

The genetic and evolutionary operators were then added to the SoDSA to create the genetic and evolutionary SoDSA (GSoDSA and ESoDSA). The two operators were not changed and were implemented similarly to those used in GSoHC and ESoHC. The inclusion of the two operators further necessitates the need for a global view. The next chapter will compare the performance of mdBA, SoHC, GSoHC, ESoHC, DSA-B, SoDSA, GSoDSA, and ESoDSA by applying them to the problem of randomly generated asymmetric constraint satisfaction problems (DisACSPs).

CHAPTER 4

SOLVING DISTRIBUTED ASYMMETRIC CONSTRAINT SATISFACTION PROBLEMS USING GENETIC AND EVOLUTIONARY PROTOCOLS

4.1. Introduction

In order to assess the performance of the Genetic (GSoHC) and Evolutionary (ESoHC) Protocols (GEPs), they will first be tested on randomly generated distributed asymmetric constraint satisfaction problems (DisACSP). Their overall performance will be compared to those of the mDBA and Society of Hill-Climbers (SoHC) to see how much of an improvement is gained. The GEPs will also be compared to the Distributed Stochastic Algorithm (DSA) and the DSA based GEPs created for this research.

4.2. Randomly Generated DisACSP

A randomly generated DisACSP can be defined by four parameters: the number of variables/agents (n), domain size (m), constraint density (p_1), and constraint tightness (p_2) [36, 37, 164]. The problem parameters can be expressed as a 4-tuple $<n, m, p_1, p_2>$. As mentioned in Chapter 2, not all CSPs are solvable. Thus, the best indicator of problem difficulty is how close it is to the phase transition [93, 140]; the hardest problems will be at the phase transition [140]. In previous work [36, 37], we have shown that randomly generated ACSPs with parameters <30, 6, 1, 0.01> - <30, 6, 1, 0.06> are good indicators

of performance. Thus, these will be the problems primarily used to test the dBA based GEPs, mDBA, SoHC, DSA and DSA based GEPs.

In randomly generated DisACSPs, a constraint, or no-good, can be represented by a 4-tuple, $<var_1, var_2, val_1, val_2>$, composed of the two variables and their corresponding value assignments. Since the constraints are asymmetric, given two no-goods $<x, y, a, b>$ and $<y, x, b, a>$, if $x \neq y$, then the two no-goods are not equivalent and are considered to be separate constraints. It should be noted that x will never be equal to y, thus $<x, y, a, b>$ and $<y, x, b, a>$ will never be equivalent when asymmetric constraints are used. The no-good $<x, y, a, b>$ can then be interpreted as "if x is assigned the value a, then y cannot be assigned the value b."

4.3. Testing

When testing algorithms for solving randomly generated DisACSPs, a primary concern is whether the algorithm can solve the problem within a given number of iterations. The primary metric for performance is the percentage of problems an algorithm can solve. In a situation where two algorithms perform similarly on the percentage of problems solved, the algorithm that can find a solution in fewer iterations is judged to be better.

For these tests, each algorithm was given 2000 iterations to solve the given problem. If a solution was not found within 2000 iterations, then the algorithm was terminated, the problem was marked as being unsolved and 2000 was recorded as the number of iterations for that problem [37, 37, 55]. The average number of iterations needed to solve a problem will therefore also include the runs where no solution was found. Thus, if an

algorithm was unable to find a solution for any of the problems, then the average number of iterations for that algorithm would be 2000.

4.4. Results: mdBA vs SoHC

Tables 4.1 and 4.2, show the results for SoHC on a total of 600 randomly generated problems with 100 problems for each parameter set, <30, 6, 1, 0.01>, <30, 6, 1, 0.02>, <30, 6, 1, 0.03>, <30, 6 1, 0.04>, <30, 6, 1, 0.05>, and <30, 6, 1, 0.06>. A total of 30 trial runs were performed on each randomly generated problem for each parameter set. The results are therefore averaged over 3000 runs per population and tightness setting.

As described earlier, the only difference between a SoHC and mdBA is the population based approach of the SoHC. Consequently, when a SoHC has only a population size of 1, it becomes a mdBA. As the results in the tables show, the general trend is that as the problem gets harder it becomes exponentially harder to find a solution within 2000 iterations. However, the larger the population size, the more likely it is that the algorithm will find a solution and it will do so faster.

Though problems with a tightness of 0.01 are relatively easy to solve, the results show that there is still a 0.5% chance that the mdBA will fail to identify a solution within 2000 iterations. As there were 100 different problems generated, 30 runs performed on each, and a failure rate for the mdBA of 0.5%, this indicates that there were problems where the mdBA was not able to solve consistently in the 30 trial runs. Since each trial run differs by only the initial random candidate solution, this highlights the possible impact the initial starting candidate solution has on the performance of the algorithm. As the problems become harder, SoHC also begins to suffer from this consistency problem if

the population size is not large enough; for example with a SoHC of populations size 2 (SoHC-2) at a tightness of 0.02, SoHC-4 at 0.03, and SoHC-8 at 0.04.

At a tightness of 0.05, the problem is considerably harder for both SoHC and mdBA, and even SoHC-32 was not able to solve 50% of the problems. However, it still performed much better than the mdBA, which was only able to solve 5.4% of the problems. At the phase transition tightness of 0.06 where there may only be one solution, mdBA, SoHC-2, SoHC-4, and SoHC-8 were not able to find any solutions within 2000 iterations, while SoHC-16 was able to solve only 0.03% (1 out of 3000) of the problems and SoHC-32 solved 0.23% (7 out of 3000) of the problems. Based on the two-tailed *t*-test for the difference in mean with $a = 0.05$, the performance difference between SoHC-16 and SoHC-32 is not significant.

To assess the significance of the results, an ANOVA test was performed over each problem tightness to show that the change in performance, seen in Table 4.2, caused by the population increase is significant. The one-factor ANOVA test was performed and

SoHC	Population Size					
Tightness	1	2	4	8	16	32
0.01	99.50	100.00	100.00	100.00	100.00	100.00
0.02	91.50	99.20	100.00	100.00	100.00	100.00
0.03	80.53	95.97	99.77	100.00	100.00	100.00
0.04	70.93	89.50	98.23	99.77	100.00	100.00
0.05	5.40	9.40	16.17	24.80	35.47	49.73
0.06	0.00	0.00	0.00	0.00	0.03	0.23

Table 4.1. Percentage of problems solved within 2000 iterations

SoHC	Population Size					
Tightness	1	2	4	8	16	32
0.01	17.80	6.11	4.87	3.83	2.98	2.28
0.02	187.52	31.76	13.09	11.28	9.73	8.55
0.03	437.11	121.58	34.97	23.57	19.60	17.11
0.04	891.44	527.94	261.78	139.42	83.40	53.67
0.05	1945.43	1898.77	1814.74	1702.74	1542.41	1334.73
0.06	2000.00	2000.00	2000.00	2000.00	1999.49	1997.79

Table 4.2. Average number of iterations to find a solution

the results are presented in Table 4.3. Given that the F value for $df_n = 5$, $df_d = 17{,}994$ and $p = 0.01$ is 3, the results presented in Table 4.2 is clearly significant and the population size does have a significant affect on performance.

It can be argued that the direct comparison of performance between varying population sizes is inherently unfair as the larger population sizes will gain an automatic advantage, since they tend to search through more candidate solutions per iteration. One method used to equalize this performance difference is to look at the total number of fitness function evaulations, which corresponds to the number of candidate solutions searched before finding a solution. Here, instead of averaging fitness function evaulations, the average number of constraint checks made before finding a feasible solution is used for comparison. The results are presented in Table 4.4. It should be noted that in a distributed problem, the most time consuming process is message passing among the agents. This is the main reason why, in this research, iterations of an algorithm is used as a primary benchmark.

Table 4.4 shows the average number of constraint checks for instances where a feasible solution was found, which is why there are no results for population sizes of 1, 2, 4, and 8 when the constraint tightness is 0.06. At the same constraint tightness, the averages for the remaining population sizes of 16 and 32 should not be regarded as being significant either as they are averages over the very small set of runs that actually found a

Tightness	F-value
0.01	30.522196
0.02	269.509278
0.03	655.380354
0.04	1660.770074
0.05	528.006735
0.06	3.387420

Table 4.3. One Factor ANOVA test Results over the Average Iterations to find a Solution for SoHC where $df_n = 5$ and $df_d = 17{,}994$

feasible solution within 2000 iterations. In all cases, the increase of constraint checks, percentage-wise, is greater than the decrease in the average number of iterations to solve a problem. The decrease in the average number of iterations to solve a problem as the population grows does contribute to the result that the average constraint checks does not grow linearly with the population size.

As the problems get harder, it becomes harder for the many variations of SoHC to find a solution within 2000 iterations. This makes it more difficult to see the performance difference between differing population sizes. To solve this problem, the number of constraint violations at the end of 2000 iterations for those runs where a feasible solution was not found is recorded and averaged. The results are presented in Table 4.5. To put the numbers in Table 4.5 into perspective, Table 4.6 divides the numbers in Table 4.5 by the maximum number of constraints that were generated given the problem tightness to show the average percentage of the total constraints that were not resolved at the end of 2000 iterations when no solution was found.

One of the biggest difference between SoHC and a standard hill-climber is the use of the breakout list to penalize known no-goods. The purpose of the breakout list is to modify the fitness such that the search can move out of a local optimal. This should be considered when examining and interpreting the results presented in Tables 4.5 and 4.6. The tables simply show the remaining constraint conflicts after 2000 iteration when no

	Population Size					
Tightness	1	2	4	8	16	32
0.01	14,266	24,604	43,460	76,397	134,174	236,889
0.02	36,852	65,940	117,897	215,881	395,503	732,900
0.03	102,988	161,884	259,480	443,567	793,935	1,460,071
0.04	779,021	1,266,325	1,702,760	2,121,342	2,822,877	3,976,789
0.05	2,209,501	4,199,550	7,861,272	14,935,237	27,019,787	51,316,213
0.06					23,053,840	96,982,649

Table 4.4. Average number of Constraint Checks to find a feasible solution

solution is found and does not factor in the possible penalty placed on these constraints or the surrounding search space by the breakout mechanism. Thus, generally, neither the ending fitness value nor the number of remaining constraints can indicate how close a search is to finding a feasible solution as there is no real way of telling whether the search ended the 2000 iterations stuck at a local optimal or in the process of moving out of one. However, the tables do show that when the problem gets harder, with a constraint tightness of 0.05 and 0.06, where there are a large number of local optima, the increase in population size helps find relatively better sub-optimal solutions within the same 2000 iterations. This is especially seen for problems with a constraint tightness of 0.06 where less than 1% of the problems were solved within 2000 iterations.

One of the concerns about SoHC that was addressed in a previous study [55] was the issue of space complexity. The question was whether the breakout list would become too long and take up a significant amount of memory. Table 4.7 shows the average length of the breakout list at the end of each run. As expected, increasing the population size

SoHC	Population Size					
Tightness	1	2	4	8	16	32
0.01	1.07					
0.02	1.28	1.25				
0.03	1.66	1.76	3.00			
0.04	3.86	3.83	3.90	3.50		
0.05	11.22	9.87	8.72	7.47	6.80	6.22
0.06	16.91	15.06	13.79	12.81	11.90	11.18

Table 4.5. Average number of unresolved constraints when no solution was found within 2000 iterations

SoHC	Population Size					
Tightness	1	2	4	8	16	32
0.01	0.34%					
0.02	0.20%	0.20%				
0.03	0.18%	0.19%	0.32%			
0.04	0.31%	0.31%	0.31%	0.28%		
0.05	0.72%	0.63%	0.56%	0.48%	0.43%	0.40%
0.06	0.90%	0.80%	0.73%	0.68%	0.63%	0.60%

Table 4.6. Percentage of total constraints unresolved after 2000 iterations

increased the length of the breakout list, especially for harder problems. For problems with a tightness of no more than 0.03, SoHC was usually able to solve the problem before many breakouts were placed. This can be seen as the average ending length of the list decreased with population size. As the problems become harder, increasing the population size increased the chance of finding a local optimum, which in turn increased the average length of the breakout list. It is interesting to see that at the phase transition, SoHC actually placed fewer breakouts than when at a tightness of 0.05. This highlights one of the weaknesses of the breakout method, where if there is a large clustering of local optima, then the search may oscillate between the many local optima until the penalty weights have accumulated to the point of escaping the entire cluster. Thus, at the phase transition, SoHC was not able to find as many local optima as it could at a tightness of 0.05, which is why the breakout list is significantly shorter. In general, the breakout list length scales logarithmically with the population size.

Based on these results, there is no doubt that SoHC performs better than mdBA, and this performance difference can be attributed to SoHC's population based approach. However, neither was able to solve a significant number of problems at the phase transition. Next the effect on performance of adding the genetic operator to SoHC will be considered.

	Population Size					
Tightness	1	2	4	8	16	32
0.01	0.04	0.01	0.01	0.00	0.01	0.01
0.02	1.05	0.69	0.43	0.32	0.27	0.24
0.03	8.69	8.45	6.77	5.23	4.06	3.60
0.04	77.03	96.65	102.61	99.91	99.82	91.96
0.05	150.92	230.84	332.83	444.53	556.26	652.11
0.06	122.84	189.06	276.30	385.83	512.95	654.64

Table 4.7. Average Ending BreakOut List Length

4.5. Results: Genetic Protocol (GSoHC)

The distributed genetic operator has two parameters, namely the probability that the variable randomly chooses a value from the domain, the mutation rate p_m, and the probability that the variable takes on the value of the best instance, the crossover rate p_c. Since the genetic operator uses uniform crossover, the value of p_c will be 0.5. However, because mutation and crossover are performed in the same step, the parameter constraint $2 \cdot p_c + p_m = 1$ needs to be satisfied. Based on this constraint, the mutation rate was selected as the determining factor for the crossover rate. In previous works, the mutation rate was set at 6% with a crossover rate of 47%, but here the results obtained by varying the mutation rate will be explored. These results are based on the average performance over the same 600 problems used for testing SoHC. Each variation of GSoHC was run once on each of the 600 problems.

Tables 4.8 and 4.9 present the number of problems solved and the average iterations to solve a problem for each mutation and problem setting. To show the significance of the results, a one factor ANOVA test was performed to show whether changing the mutation rate has a significant affect on the average number of iterations to find a solution. The F-values for the ANOVA tests are presented in Table 4.10, while the sets of results that are significant are highlighted in Tables 4.8 and 4.9.

Table 4.8 shows that the mutation rate does not affect the performance at a tightness of 0.01. This is supported by the results in table 4.10 as the performance of varying mutation rates is not significant. At a tightness of 0.02, the GSoHC with no mutation and a population size of 2 (GsoHC-0.00-2) and 4 (GsoHC-0.00-4) are the only algorithms that were not able to solve all the problems. The performance difference starts to show at a

tightness of 0.03, where no mutation and too much mutation are both detrimental to GSoHC performance. However, an increase in population size can still make up for the performance variation due to mutation. At a tightness of 0.04, GSoHC-0.06 and GSoHC-0.12 appear to perform better than the other candidates. At a tightness of 0.05, it becomes clear that a mutation rate of 0.24 or more is too much as the two variations fall behind. GSoHC-0.12 falls slightly behind GSoHC-0.06 here, but the most interesting result is that GSoHC-0.00 is actually able to beat GSoHC-0.06 at population sizes less than 16. This is due to the higher level of exploitation in plain crossover than crossover with mutation. However, GSoHC-0.06-32 performs significantly better than GSoHC-0.00-32. At a tightness of 0.06, none of the variations perform significantly better than the others.

Table 4.9 shows the average number of iterations that were needed to solve a problem. The numbers further reinforce the performance advantage of GSoHC-0.06 as, taking into consideration the percentage of problems solved, GSoHC-0.06 was able to solve more problems faster than the other variants. These results also show that at a tightness of 0.01, where the problem is easy, the choice of mutation rate is not as important, but as the problem gets harder it becomes clear that too much mutation is not good for the search. Tables 4.11 and 4.12 shows the detailed results for GSoHC-0.06 over the same set of 600 problems, except with each problem run 30 times.

With the chosen mutation rate of 0.06 for GSoHC, Tables 4.13 and 4.14 Present the average number of remaining constraints when no solution was found within 2000 iterations, along with the percentage of overall constraints. Overall, the trend in average remaing constraints is the same as SoHC, except GSoHC averages fewer remaining

Tightness	Mutation Rate	Population Size				
		2	4	8	16	32
0.01	0.00	100.00	99.00	100.00	100.00	100.00
	0.06	100.00	100.00	100.00	100.00	100.00
	0.12	100.00	100.00	100.00	100.00	100.00
	0.24	100.00	100.00	100.00	100.00	100.00
	0.50	100.00	100.00	100.00	100.00	100.00
0.02	0.00	95.00	99.00	100.00	100.00	100.00
	0.06	100.00	100.00	100.00	100.00	100.00
	0.12	100.00	100.00	100.00	100.00	100.00
	0.24	100.00	100.00	100.00	100.00	100.00
	0.50	100.00	100.00	100.00	100.00	100.00
0.03	0.00	92.00	97.00	100.00	100.00	100.00
	0.06	100.00	100.00	100.00	100.00	100.00
	0.12	100.00	100.00	100.00	100.00	100.00
	0.24	100.00	100.00	100.00	100.00	100.00
	0.50	96.00	100.00	100.00	100.00	100.00
0.04	0.00	73.00	90.00	99.00	100.00	100.00
	0.06	92.00	100.00	100.00	100.00	100.00
	0.12	89.00	100.00	100.00	100.00	100.00
	0.24	80.00	98.00	100.00	100.00	100.00
	0.50	78.00	96.00	99.00	100.00	100.00
0.05	0.00	14.00	31.00	64.00	82.00	81.00
	0.06	11.00	29.00	53.00	87.00	95.00
	0.12	7.00	29.00	45.00	70.00	91.00
	0.24	7.00	16.00	32.00	41.00	68.00
	0.50	6.00	13.00	17.00	30.00	43.00
0.06	0.00	0.00	0.00	0.00	1.00	2.00
	0.06	0.00	0.00	0.00	0.00	0.00
	0.12	0.00	0.00	1.00	1.00	1.00
	0.24	0.00	0.00	0.00	1.00	1.00
	0.50	0.00	1.00	0.00	1.00	1.00

Table 4.8. Percentage of problems solved for GSoHC with varying mutation rates

constraints than SoHC. The reduction of remaining constraints with the increase of population size is still true.

Table 4.15 presents the average number of constraint checks to solve a problem, when a solution was found within 2000 iterations. The numbers show that GSoHC requires a lot fewer constraint checks on average to find a solution. This is mainly due to the fact that GSoHC finds solutions faster than SoHC and is capable of solving nearing twice as many problems as SoHC, in some cases. This affect is especially apparent as the

Tightness	Mutation Rate	Population Size				
		2	4	8	16	32
0.01	0.00	5.49	24.00	3.31	2.63	2.05
	0.06	5.52	4.30	3.53	2.71	1.93
	0.12	5.66	4.22	3.46	2.69	1.92
	0.24	5.56	4.02	3.39	2.56	2.00
	0.50	5.71	4.64	3.41	2.60	2.10
0.02	0.00	113.96	31.76	9.20	8.13	7.07
	0.06	16.29	12.68	9.61	8.27	6.89
	0.12	16.54	11.78	9.86	8.48	7.21
	0.24	19.47	12.95	10.19	8.93	7.83
	0.50	19.67	12.59	10.90	8.94	7.97
0.03	0.00	190.54	96.72	18.59	15.07	13.06
	0.06	52.34	25.48	19.96	16.25	13.43
	0.12	64.31	36.39	21.25	17.16	14.60
	0.24	91.07	37.74	24.73	18.23	15.20
	0.50	163.56	39.44	25.89	19.13	16.78
0.04	0.00	696.22	316.25	94.71	45.71	28.95
	0.06	521.88	157.76	86.13	46.42	32.81
	0.12	710.50	212.85	90.13	55.20	39.34
	0.24	764.71	325.97	132.28	78.06	44.84
	0.50	839.82	406.14	185.61	111.76	64.53
0.05	0.00	1861.76	1641.81	1084.19	727.98	685.31
	0.06	1874.73	1622.95	1328.67	619.98	404.85
	0.12	1945.84	1677.73	1347.86	989.11	563.74
	0.24	1928.33	1809.71	1622.62	1438.58	1054.31
	0.50	1939.65	1886.31	1828.13	1614.08	1500.40
0.06	0.00	2000.00	2000.00	2000.00	1983.69	1963.63
	0.06	2000.00	2000.00	2000.00	2000.00	2000.00
	0.12	2000.00	2000.00	1991.06	1982.00	1981.68
	0.24	2000.00	2000.00	2000.00	1988.26	1984.88
	0.50	2000.00	1981.87	2000.00	1983.22	1987.87

Table 4.9. Average number of Iterations required to solve a problem for GSoHC with varying Mutation Rates

GSoHC	Population Size				
Tightness	2	4	8	16	32
0.01	0.5310	0.9720	0.4532	0.2508	0.4152
0.02	4.8267	0.9489	7.5559	5.4176	10.1112
0.03	4.5569	3.2745	13.7172	16.6609	23.4597
0.04	4.9642	7.5421	7.5939	20.5739	19.8019
0.05	1.0338	4.2327	19.9734	43.7507	50.9842
0.06	0.0000	1.0000	1.0000	0.2777	1.0184

Table 4.10. F-values from running the one Factor ANOVA test on the results from Table 4.9 over the varying mutation rate

(df_n = 4, df_d = 495 and for p = 0.05, F = 2.39)

	Population Size				
Tightness	2	4	8	16	32
0.01	100.00	100.00	100.00	100.00	100.00
0.02	100.00	100.00	100.00	100.00	100.00
0.03	99.97	100.00	100.00	100.00	100.00
0.04	95.10	99.90	100.00	100.00	100.00
0.05	8.20	29.90	61.23	84.50	93.93
0.06	0.00	0.23	0.80	1.10	2.30

Table 4.11. Percentage of Problems Solved for GSoHC with Mutation rate of 0.06

	Population Size				
Tightness	2	4	8	16	32
0.01	5.69	4.26	3.38	2.63	2.00
0.02	16.90	11.74	9.62	8.22	7.12
0.03	52.53	28.60	20.14	16.15	13.70
0.04	493.54	167.14	77.83	46.06	32.36
0.05	1912.67	1655.58	1186.17	692.19	392.25
0.06	2000.00	1997.69	1989.41	1982.33	1966.50

Table 4.12. Average Cycles to Solve a Problems for GSoHC with Mutation rate of 0.06

GSoHC	Population Size				
Tightness	2	4	8	16	32
0.01					
0.02					
0.03	2.67				
0.04	5.23	4.67			
0.05	10.81	8.47	6.83	5.29	3.59
0.06	15.98	13.34	11.35	9.66	8.06

Table 4.13. Average number of unresolved constraints when no solution was found within 2000 iterations

GSoHC	Population Size				
Tightness	2	4	8	16	32
0.01					
0.02					
0.03	0.28%				
0.04	0.42%	0.37%			
0.05	0.69%	0.54%	0.44%	0.34%	0.23%
0.06	0.85%	0.71%	0.60%	0.51%	0.43%

Table 4.14. Percentage of total constraints unresolved after 2000 iterations

population size increases. At a population size of 32 and problems with a tightness of 0.05, GSoHC requires, on average, less than half the number of constraint checks to solve a problem as compared to SoHC.

GSoHC	Population Size				
Tightness	2	4	8	16	32
0.01	22,812	38,133	66,924	117,543	207,926
0.02	66,244	102,653	179,313	322,166	586,382
0.03	193,616	240,468	373,757	639,234	1,136,351
0.04	1,622,238	1,277,302	1,279,050	1,648,906	2,505,111
0.05	4,435,264	7,620,302	11,859,729	15,837,771	20,138,056
0.06		9,889,704	13,278,653	15,976,856	42,034,304

Table 4.15. Average number of Constraint Checks to find a feasible solution

4.6. Results: Evolutionary Protocol (ESoHC)

Unlike GSoHC, ESoHC depends on mutation only. Thus, if the mutation rate drops to 0, ESoHC becomes similar to SoHC. The only difference would be that in ESoHC with a mutation rate of 0 (EsoHC-0), the below average half of the population, fitness-wise, will become exact copies of the best individual. The impact of changing the mutation rate on the performance of ESoHC is first considered. It should be noted that given the mutation rate p_m, below average individuals in ESoHC will become variations of the best individual.

Similar to GSoHC, a one-factor ANOVA test was performed on the results to see if the variation in the average iterations to solve a problem is significant over the various mutation rates. The results of this is presented in Table 4.18, and the significant sets of results are highlighted accordingly in Tables 4.16 and 4.17.

As shown in Table 4.16 a mutation rate of 0.06 and 0.12 appears to offer the best performance, with the lower mutation rate edging the other out slightly in the number of problems solved. At a problem tightness of 0.05, where the problem gets significantly harder, it is interesting to see that a mutation rate of 1 for a population size of 2 actually performs quite well compared to lower mutation rates, although this performance does not scale with population size. Ultimately, the determining factor appears to be the

balance between exploration and exploitation; with a lower mutation, there is more exploitation, while higher mutation rates create more exploration. At low population sizes, a high level of exploration can help the algorithm, but at higher population sizes, a lower mutation rate, and greater exploitation must be used to obtain better results as the

Tightness	Mutation rate	Population size				
		2	4	8	16	32
0.01	0.06	100	100	100	100	100
	0.12	100	100	100	100	100
	0.25	100	100	100	100	100
	0.50	100	100	100	100	100
	0.75	100	100	100	100	100
	1.00	100	100	100	100	100
0.02	0.06	100	100	100	100	100
	0.12	100	100	100	100	100
	0.25	100	100	100	100	100
	0.50	100	100	100	100	100
	0.75	100	100	100	100	100
	1.00	99	100	100	100	100
0.03	0.06	100	100	100	100	100
	0.12	100	100	100	100	100
	0.25	100	100	100	100	100
	0.50	96	100	100	100	100
	0.75	98	100	100	100	100
	1.00	96	100	100	100	100
0.04	0.06	97	100	100	100	100
	0.12	91	100	100	100	100
	0.25	91	99	100	100	100
	0.50	80	98	100	100	100
	0.75	80	95	99	100	100
	1.00	75	97	100	100	100
0.05	0.06	10	33	66	88	92
	0.12	7	26	61	84	94
	0.25	6	24	36	72	88
	0.50	5	17	23	37	53
	0.75	4	8	17	31	42
	1.00	12	11	16	21	44
0.06	0.06	0	0	2	0	3
	0.12	0	0	0	1	3
	0.25	0	0	0	1	2
	0.50	0	0	0	1	0
	0.75	0	0	0	0	1
	1.00	0	0	1	0	0

Table 4.16 Percentage of Problems solved by ESoHC with varying Mutation rate

population itself contributes a certain level of exploration to the search. This is supported by the data in Table 4.17.

The results show that ESoHC performs best at a mutation rate of around 0.06 to 0.12. It is interesting to see how that at lower population sizes, ESoHC-0.06 is actually able to slightly outperform EsoHC-0.12 even though the lower mutation rate leads to higher levels of exploitation. However, as population size increases, the performance margin is reduced and ESoHC-0.12 outperforms ESoHC-0.06 at a population size of 32. To take a closer look at the performance of ESoHC-0.12 and EsoHC-0.06, we took the 100 randomly generated problems used earlier and run each ESoHC variation on each problem 30 times, producing the results in Tables 4.19 and 4.20.

The results in Tables 4.19 and 4.20 show that, for the most part, ESoHC-0.06 and ESoHC-0.12 perform the same except for when the problem tightness is at 0.05, where there is a definite discrepancy in the performance of the two ESoHC's. Based on the statistical test for the difference of means, the performance difference is significant. As mentioned earlier, ESoHC-0.06 does not scale well with population size as ESoHC-0.12 performs better at a population size of 32, while the percentage of problems solved by ESoHC-0.06 did not increase a statistically significant amount when the population size increased from 16 to 32. As this study uses a population size of 32 in the application tests in later chapters, ESoHC-0.12 will be used in this work.

Tables 4.21 and 4.22 presents the average number of remaining constraints when a solution was not found within 2000 iterations, along with the percentage of overall constraints. Compared to GSoHC, ESoHC is able to further reduce the number of remaining constraints if a solution was not found within 2000 iterations. The reduction

Tightness	Mutation rate	Population size				
		2	4	8	16	32
0.01	0.06	5.62	4.12	2.95	2.37	1.69
	0.12	5.75	4.17	2.96	2.32	1.69
	0.25	5.79	4.11	3.06	2.42	1.62
	0.50	5.97	4.20	3.31	2.57	2.05
	0.75	5.73	4.60	3.48	2.70	2.20
	1.00	6.07	4.35	3.43	2.72	1.94
0.02	0.06	15.75	11.09	8.38	6.97	5.99
	0.12	16.36	11.21	8.71	6.89	5.95
	0.25	19.54	11.97	9.93	7.78	6.30
	0.50	24.72	12.79	10.78	8.48	7.59
	0.75	20.78	13.19	10.72	8.98	8.11
	1.00	42.47	12.58	11.01	9.25	8.21
0.03	0.06	51.27	25.06	19.82	12.95	11.83
	0.12	49.09	29.81	19.63	13.96	12.01
	0.25	73.76	29.13	20.96	16.72	13.75
	0.50	154.81	35.31	23.80	17.53	15.43
	0.75	114.76	40.06	25.41	20.23	16.65
	1.00	152.33	46.98	26.84	19.14	16.85
0.04	0.06	422.44	157.72	66.63	45.53	28.30
	0.12	581.63	188.95	68.76	41.01	31.13
	0.25	738.75	220.54	87.60	55.49	33.56
	0.50	782.46	329.43	181.25	98.75	51.36
	0.75	796.24	415.04	206.88	103.54	67.40
	1.00	816.11	408.84	180.05	121.05	70.15
0.05	0.06	1869.31	1616.81	1079.15	584.67	496.54
	0.12	1944.60	1699.69	1109.38	747.35	398.95
	0.25	1948.38	1706.74	1558.01	984.08	550.96
	0.50	1939.80	1780.64	1727.08	1496.23	1303.10
	0.75	1951.03	1908.91	1844.79	1579.07	1454.74
	1.00	1881.10	1893.87	1835.21	1726.75	1530.74
0.06	0.06	2000.00	2000.00	1977.03	2000.00	1955.90
	0.12	2000.00	2000.00	2000.00	1981.89	1976.29
	0.25	2000.00	2000.00	2000.00	1982.08	1967.15
	0.50	2000.00	2000.00	2000.00	1983.00	2000.00
	0.75	2000.00	2000.00	2000.00	2000.00	1980.49
	1.00	2000.00	2000.00	1998.13	2000.00	2000.00

Table 4.17. Average Number of Cycles to Solve a Problem with varying problem Tightness and Mutation Rate

of remaining conflicts with the increase of population size is also much greater than either SoHC or GSoHC.

Table 4.23 shows the average constraint checks required to solve a problem. In comparison to SoHC and GSoHC, the average number of constraint checks is the lowest,

ESoHC	Population Size				
Tightness	2	4	8	16	32
0.01	0.3394	1.2380	3.9514	2.5037	7.1414
0.02	1.4319	4.8906	21.2557	30.8899	45.2767
0.03	3.5733	4.1276	7.3808	36.1806	42.1127
0.04	5.6627	9.2503	14.4472	14.7535	18.8872
0.05	1.4520	4.9716	30.6146	46.7359	63.3445
0.06	0.0000	0.0000	1.9364	0.6005	0.9879

Table 4.18. F-values from running the one Factor ANOVA test on the results from Table 4.14 over the varying mutation rate

(df_n = 5, df_d = 594 and for p = 0.05, F = 2.23)

	Mutation	Population Size				
Tightness	Rate	2	4	8	16	32
0.01	0.06	100.00	100.00	100.00	100.00	100.00
	0.12	100.00	100.00	100.00	100.00	100.00
0.02	0.06	100.00	100.00	100.00	100.00	100.00
	0.12	100.00	100.00	100.00	100.00	100.00
0.03	0.06	99.80	100.00	100.00	100.00	100.00
	0.12	99.83	100.00	100.00	100.00	100.00
0.04	0.06	95.87	99.87	100.00	100.00	100.00
	0.12	94.10	99.80	100.00	100.00	100.00
0.05	0.06	11.37	33.30	69.13	89.83	90.63
	0.12	7.40	28.50	64.30	87.37	95.40
0.06	0.06	0.13	0.40	0.67	1.47	3.07
	0.12	0.13	0.43	0.70	1.33	2.70

Table 4.19. Comparison of Percentage of Problems Solved for ESoHC-0.06 and ESoHC-0.12

	Mutation	Population Size				
Tightness	Rate	2	4	8	16	32
0.01	0.06	5.60	4.11	3.11	2.32	1.72
	0.12	5.50	4.11	3.07	2.32	1.68
0.02	0.06	16.41	10.93	8.48	6.95	5.75
	0.12	16.69	11.32	8.82	7.09	5.89
0.03	0.06	53.08	26.66	17.76	13.70	11.24
	0.12	61.73	28.55	18.71	14.36	11.71
0.04	0.06	449.80	156.21	69.59	40.84	28.59
	0.12	548.64	174.58	75.59	41.67	28.26
0.05	0.06	1876.80	1605.17	1071.15	596.87	486.41
	0.12	1919.95	1675.00	1131.62	634.82	349.35
0.06	0.06	1998.89	1994.99	1990.11	1980.83	1956.59
	0.12	1998.10	1995.09	1988.67	1980.96	1961.25

Table 4.20. Comparison of Average Iterations to Solve a Problem for ESoHC-0.06 and ESoHC-0.12

especially at larger populations. It is interesting to note that at a population size of 2, ESoHC performs more constraint checks to solve a problem than GSoHC. This is attributed to characteristics of the algorithm itself that will be discussed in the next section.

4.7. Results: mdBA vs SoHC vs GSoHC vs ESoHC

After determining the best parameter settings for GSoHC and ESoHC, it is now possible to compare the performance of mdBA, SoHC, GSoHC, and ESoHC to see the affect of the distributed crossover and mutation operators. Table 4.24 lists the percentage of problems solved by each algorithm. For GSoHC, the mutation rate was set to 0.06

ESoHC	Population Size				
Tightness	2	4	8	16	32
0.01					
0.02					
0.03	2.00				
0.04	6.08	2.75			
0.05	11.01	8.60	6.25	4.39	2.09
0.06	16.34	13.16	10.65	8.13	4.86

Table 4.21. Average number of unresolved constraints when no solution was found within 2000 iterations

ESoHC	Population Size				
Tightness	2	4	8	16	32
0.01					
0.02					
0.03	0.21%				
0.04	0.49%	0.22%			
0.05	0.70%	0.55%	0.40%	0.28%	0.13%
0.06	0.87%	0.70%	0.57%	0.43%	0.26%

Table 4.22. Percentage of total constraints unresolved after 2000 iterations

ESoHC	Population Size				
Tightness	2	4	8	16	32
0.01	21,732	35,576	59,737	102,979	178,961
0.02	65,024	95,987	157,701	267,324	468,051
0.03	215,580	232,026	330,249	534,828	913,065
0.04	1,819,121	1,294,928	1,175,971	1,392,709	2,017,939
0.05	4,414,477	7,630,448	10,968,302	14,227,463	17,108,391
0.06	3,295,741	8,533,629	7,079,017	20,599,088	41,263,427

Table 4.23. Average number of Constraint Checks to find a feasible solution

with a the crossover rate of 0.47. For ESoHC, the mutation rate was set at 0.12. Once again, it should be noted that SoHC with a population size of 1 is mDBA.

The mDBA quickly falls behind on the percentage of problems solved as the problem gets harder, which is expected. At a population size of 2, it is interesting to see that there is no significant performance difference between SoHC, GSoHC, and ESoHC when comparing the percentage of prob lems solved. Unlike SoHC, at a population size of 2 GSoHC and ESoHC have only one individual doing plain hill-climbing. Thus, the advantage of the distributed crossover and mutation is not as obvious. As the population size increases, the effectiveness of these two operators starts to show as the GSoHC and ESoHC are able to solve more problems even with only half the population hill-climbing. When comparing the percentage of problems solved, the performances of GSoHC and ESoHC are not significantly different, even at the phase transition.

		Population Size					
Tightness		1	2	4	8	16	32
0.01	SoHC	99.50	100.00	100.00	100.00	100.00	100.00
	GSoHC		100.00	100.00	100.00	100.00	100.00
	ESoHC		100.00	100.00	100.00	100.00	100.00
0.02	SoHC	91.50	99.20	100.00	100.00	100.00	100.00
	GSoHC		100.00	100.00	100.00	100.00	100.00
	ESoHC		100.00	100.00	100.00	100.00	100.00
0.03	SoHC	80.53	95.97	99.77	100.00	100.00	100.00
	GSoHC		99.97	100.00	100.00	100.00	100.00
	ESoHC		99.83	100.00	100.00	100.00	100.00
0.04	SoHC	70.93	89.50	98.23	99.77	100.00	100.00
	GSoHC		95.10	99.90	100.00	100.00	100.00
	ESoHC		94.10	99.80	100.00	100.00	100.00
0.05	SoHC	5.40	9.40	16.17	24.80	35.47	49.73
	GSoHC		8.20	29.90	61.23	84.50	93.93
	ESoHC		7.40	28.50	64.30	87.37	95.40
0.06	SoHC	0.00	0.00	0.00	0.00	0.03	0.23
	GSoHC		0.00	0.23	0.80	1.10	2.30
	ESoHC		0.13	0.43	0.70	1.33	2.70

Table 4.24. Comparison of Percentage of Problems Solved between mDBA, SoHC, GSoHC, and ESoHC

Table 4.25 compares the average number of cycles needed to find a solution with each of the algorithms. Based on the one factor ANOVA test with $p = 0.05$, the performance differences between the varying algorithms is significant in all cases. The importance of exploitation and exploration can clearly be seen in the results: at the same population size, SoHC has the lowest level of exploitation, followed by GSoHC, then ESoHC. Thus, at a population size of 2, SoHC is able to match GSoHC and ESoHC, which have very little exploration at this small a population size, in performance and find a solution at least as fast as GSoHC and ESoHC. As the population size increases, the population gives GSoHC and ESoHC the level of exploration that they lack and SoHC falls behind due to the lack of exploitation. Though the performance of ESoHC and GSoHC are fairly similar, ESoHC, with its higher level of exploitation, is able to perform better at larger population sizes. Ultimately, ESoHC is the better performing algorithm by a small margin, especially with a large population.As mentioned earlier, the average number of remaining conflicts when a solution was not found within 2000 iterations cannot be used as an indication as to how close the algorithm is to solving a problem, but can be used to compare, which algorithm is able to end up with the better sub-optimal solution given the same amount of run time. The results are compiled in Table 4.26. The results show the affect of the population in ending with better solutions, especially for the hard problems with a constraint tightness of 0.05 and 0.06. For problems with a tightness of 0.06, none of the algorithms could solve a significant number of the 100 test problems, however, Table 4.26 clearly shows that even though no solution could be found for the majority of test runs, ESoHC clearly ends its run on a better sub-optimal given the same 2000 iterations of run time.

Tightness		Population Size					
		1	2	4	8	16	32
0.01	SoHC	17.80	6.11	4.87	3.83	2.98	2.28
	GSoHC		5.69	4.26	3.38	2.63	2.00
	ESoHC		5.50	4.11	3.07	2.32	1.68
0.02	SoHC	187.52	31.76	13.09	11.28	9.73	8.55
	GSoHC		16.90	11.74	9.62	8.22	7.12
	ESoHC		16.69	11.32	8.82	7.09	5.89
0.03	SoHC	437.11	121.58	34.97	23.57	19.60	17.11
	GSoHC		52.53	28.60	20.14	16.15	13.70
	ESoHC		61.73	28.55	18.71	14.36	11.71
0.04	SoHC	891.44	527.94	261.78	139.42	83.40	53.67
	GSoHC		493.54	167.14	77.83	46.06	32.36
	ESoHC		548.64	174.58	75.59	41.67	28.26
0.05	SoHC	1945.43	1898.77	1814.74	1702.74	1542.41	1334.73
	GSoHC		1912.67	1655.58	1186.17	692.19	392.25
	ESoHC		1919.95	1675.00	1131.62	634.82	349.35
0.06	SoHC	2000.00	2000.00	2000.00	2000.00	1999.49	1997.79
	GSoHC		2000.00	1997.69	1989.41	1982.33	1966.50
	ESoHC		1998.10	1995.09	1988.67	1980.96	1961.25

Table 4.25. Comparison of Average number of Cycles to Solve a Problem between mDBA, SoHC, GSoHC, and EsoHC

Tightness		Population Size					
		1	2	4	8	16	32
0.01	SoHC	1.07					
	GSoHC						
	ESoHC						
0.02	SoHC	1.28	1.25				
	GSoHC						
	ESoHC						
0.03	SoHC	1.66	1.76	3.00			
	GSoHC		2.67				
	ESoHC		2.00				
0.04	SoHC	3.86	3.83	3.90	3.50		
	GSoHC		5.23	4.67			
	ESoHC		6.08	2.75			
0.05	SoHC	11.22	9.87	8.72	7.47	6.80	6.22
	GSoHC		10.81	8.47	6.83	5.29	3.59
	ESoHC		11.01	8.60	6.25	4.39	2.09
0.06	SoHC	16.91	15.06	13.79	12.81	11.90	11.18
	GSoHC		15.98	13.34	11.35	9.66	8.06
	ESoHC		16.34	13.16	10.65	8.13	4.86

Table 4.26. Average Remaining Conflicts when no solution was found within 2000 Iterations

On the other side of the average remaining conflicts, there is the number of constraint checks to solve a problem, which is presented in Table 4.27. As the problems get harder, both GSoHC and ESoHC perform fewer constraint checks to find a solution compared to SoHC. This is especially true with larger population sizes where GSoHC and ESoHC really performs better. The difference between GSoHC and ESoHC is slightly smaller than that of SoHC. This difference is most likely caused by the fewer number of iterations that ESoHC needs to find a solution as compared to GSoHC.

Table 4.28 shows a comparison of the average ending breakout list length for SoHC, GSoHC, and ESoHC. In general, ESoHC required the least number of breakouts. This is simply due to the fact that the below average half of the population will become 88% copies of the best individual, which results in a concentrated search around the best individual. Thus, fewer local optima are found, as compared to the wider searches performed by GSoHC or SoHC. The increase in space requirement is not linear to the population size, but, rather, logarithmic. Thus, space complexity is not a concern when scaling SoHC, GSoHC, and ESoHC in terms of population size.

4.8. Results: Distributed Stochastic Algorithm (DSA) and Society of DSA

In many ways, the DSA is very similar to dBA as it was also designed to solve DisCSPs. The main difference lies in that DSA is more asynchronous when it comes to deciding which agent changes its value. For the mdBA, only one agent will change its value per iteration, while for DSA, agents that are allowed to change, do so with a probability *P*. Based on the DSA-B model used in these tests, an agent is allowed to

		Population Size					
Tightness		1	2	4	8	16	32
0.01	SoHC	14,266	24,604	43,460	76,397	134,174	236,889
	GSoHC		22,812	38,133	66,924	117,543	207,926
	ESoHC		21,732	35,576	59,737	102,979	178,961
0.02	SoHC	36,852	65,940	117,897	215,881	395,503	732,900
	GSoHC		66,244	102,653	179,313	322,166	586,382
	ESoHC		65,024	95,987	157,701	267,324	468,051
0.03	SoHC	102,988	161,884	259,480	443,567	793,935	1,460,071
	GSoHC		193,616	240,468	373,757	639,234	1,136,351
	ESoHC		215,580	232,026	330,249	534,828	913,065
0.04	SoHC	779,021	1,266,325	1,702,760	2,121,342	2,822,877	3,976,789
	GSoHC		1,622,238	1,277,302	1,279,050	1,648,906	2,505,111
	ESoHC		1,819,121	1,294,928	1,175,971	1,392,709	2,017,939
0.05	SoHC	2,209,501	4,199,550	7,861,272	14,935,237	27,019,787	51,316,213
	GSoHC		4,435,264	7,620,302	11,859,729	15,837,771	20,138,056
	ESoHC		4,414,477	7,630,448	10,968,302	14,227,463	17,108,391
0.06	SoHC					23,053,840	96,982,649
	GSoHC			9,889,704	13,278,653	15,976,856	42,034,304
	ESoHC		3,295,741	8,533,629	7,079,017	20,599,088	41,263,427

Table 4.27. Average Constraint Checks to solve a problem within 2000 Iterations

		Population Size					
Tightness		1	2	4	8	16	32
0.01	SoHC	0.04	0.01	0.01	0.00	0.01	0.01
	GSoHC		0.02	0.01	0.01	0.01	0.00
	ESoHC		0.02	0.01	0.00	0.01	0.00
0.02	SoHC	1.05	0.69	0.43	0.32	0.27	0.24
	GSoHC		0.77	0.34	0.24	0.23	0.17
	ESoHC		0.70	0.35	0.24	0.20	0.17
0.03	SoHC	8.69	8.45	6.77	5.23	4.06	3.60
	GSoHC		6.28	4.39	3.31	2.71	2.46
	ESoHC		7.48	4.08	2.86	2.25	1.90
0.04	SoHC	77.03	96.65	102.61	99.91	99.82	91.96
	GSoHC		70.50	46.53	36.47	31.31	30.14
	ESoHC		75.87	45.26	31.58	24.03	20.16
0.05	SoHC	150.92	230.84	332.83	444.53	556.26	652.11
	GSoHC		162.07	254.00	295.01	267.86	227.35
	ESoHC		158.07	247.03	271.93	228.96	167.32
0.06	SoHC	122.84	189.06	276.30	385.83	512.95	654.64
	GSoHC		130.22	237.65	362.34	483.11	573.63
	ESoHC		124.49	227.68	349.87	456.55	478.81

Table 4.28. Comparison of Average Ending BreakOut List Length

change if it is able to reduce the number of conflicts or if it is currently in a conflict, but cannot reduce the number of conflicts.

Once again the DisACSPs generated to test SoHC, GSoHC, and ESoHC were used to test DSA and SoDSA. Because DSA has an extra parameter, *P*, the value of *P* was varied from 0.1 to 0.9 to test the affect on the results. Each randomly generated problem was run 30 times for each set of parameter settings. It should once again be noted that at a population size of 1, the SoDSA becomes a DSA.

Table 4.29 and Table 4.30 show the results for DSA and SoDSA on problems with a tightness of 0.01. At a problem tightness of 0.01, the problem is easy enough that SoDSA is able to solve the problems 100% of the time and the *P* value makes no difference when it comes to the percentage of problems solved. DSA is at least able to solve 99.5% of all the problems, which shows that, like dBA, the initial starting position also affects the outcome of DSA. In terms of the average number of iterations needed to find a solution, at a population size of 1 a *P* value of 0.8 seems to be the fastest. Based on a one factor ANOVA test with $p = 0.05$, the performances of DSA with $P = 0.3$ to 0.9 are not significantly different. This is highlighted in Table 4.30. The same ANOVA test performed on the results for SoDSA with population sizes 2 to 32 show that varying the *P* value has a significant affect on performance. In terms of population size, a population size of 32 is able to find solutions faster. Thus, though DSA is able to come very close to the performance of SoDSA in terms of the percentage of problems solved, SoDSA-32 is able to solve problems the fastest. For problems of tightness 0.01, a *P* value of 0.9 gives the best performance.

Tables 4.31 and 4.32 shows the results for DSA and SoDSA for problems with a tightness of 0.02. DSA quickly falls behind in performance without the advantage of a population. Comparing the percentage of problems solved for DSA, only $P = 0.6$

	Population Size					
P	1	2	4	8	16	32
0.1	99.67	100.00	100.00	100.00	100.00	100.00
0.2	99.67	100.00	100.00	100.00	100.00	100.00
0.3	99.63	100.00	100.00	100.00	100.00	100.00
0.4	99.57	100.00	100.00	100.00	100.00	100.00
0.5	99.67	100.00	100.00	100.00	100.00	100.00
0.6	99.50	100.00	100.00	100.00	100.00	100.00
0.7	99.67	100.00	100.00	100.00	100.00	100.00
0.8	99.70	100.00	100.00	100.00	100.00	100.00
0.9	99.50	100.00	100.00	100.00	100.00	100.00

Table 4.29. Percentage of Problems Solved for DSA and SoDSA on problems with Constraint Tightness of 0.01

	Population Size					
P	1	2	4	8	16	32
0.1	36.99	21.95	16.88	13.30	10.59	8.52
0.2	21.50	11.08	8.49	6.88	5.63	4.67
0.3	17.00	7.42	5.78	4.78	4.01	3.39
0.4	15.68	5.47	4.39	3.71	3.14	2.75
0.5	12.12	4.32	3.61	3.06	2.65	2.34
0.6	14.33	3.60	3.02	2.59	2.32	2.07
0.7	10.33	3.05	2.59	2.29	2.06	1.88
0.8	9.08	2.61	2.28	2.04	1.88	1.74
0.9	12.69	2.30	2.03	1.85	1.71	1.59

Table 4.30. Average Iterations to Solve a Problem for DSA and SoDSA on problems with Constraint Tightness of 0.01

performs significantly worse, but not by much. Comparing average iterations to solve a problem, $P = 0.8$ once again seem to perform the best, while the performance for $P = 0.3$ to 0.9 are not significantly different, based on the one factor ANOVA test. Looking at the performance of SoDSA-2, though not 100% of all problems can be solved, the performance variance across P values is not significant. Comparing average iterations to solve a problem, $P = 0.6$ performs the best while the performance variations between $P =$ 0.4 to 0.9 are not significant. For larger population sizes of 4 to 32, $P = 0.9$ continues to perform significantly better than all of the other values, although the problem is still not hard enough to draw a conclusion about possible performance trends for DSA and SoDSA. The only consistent conclusion seems to be that with a large enough population size, $P = 0.9$ will perform the best.

	Population Size					
P	1	2	4	8	16	32
0.1	90.60	99.30	100.00	100.00	100.00	100.00
0.2	90.80	99.30	100.00	100.00	100.00	100.00
0.3	90.80	98.87	100.00	100.00	100.00	100.00
0.4	90.63	99.00	100.00	100.00	100.00	100.00
0.5	91.50	98.87	100.00	100.00	100.00	100.00
0.6	89.80	99.27	99.93	100.00	100.00	100.00
0.7	90.37	99.10	100.00	100.00	100.00	100.00
0.8	91.73	98.97	100.00	100.00	100.00	100.00
0.9	90.33	98.97	100.00	100.00	100.00	100.00

Table 4.31. Percentage of Problems Solved for DSA and SoDSA on problems with Constraint Tightness of 0.02

	Population Size					
P	1	2	4	8	16	32
0.1	238.91	56.86	32.46	26.15	21.79	18.71
0.2	209.34	35.19	16.33	13.22	11.26	9.70
0.3	201.09	36.81	10.88	9.11	7.71	6.72
0.4	199.89	30.55	8.23	6.86	5.93	5.25
0.5	179.85	30.92	6.69	5.58	4.85	4.33
0.6	211.96	21.60	6.93	4.81	4.19	3.76
0.7	199.78	23.87	4.79	4.15	3.69	3.35
0.8	171.16	25.83	4.28	3.71	3.34	3.03
0.9	198.41	25.21	3.85	3.36	3.03	2.77

Table 4.32. Average Iterations to Solve a Problem for DSA and SoDSA on problems with Constraint Tightness of 0.02

Tables 4.33 and 4.34 shows the results for problems with a tightness of 0.03. Similar to tables 4.30 and 4.32, the sets of *P* values for each population that do not contribute significant performance variations, based on the ANOVA test, are highlighted in Table 4.34. At a tightness of 0.03, the problems are starting to become hard enough to reveal the performance difference due to the differing population sizes, and DSA falls farther behind in all respects. The *P* value still does not have a significant impact on the percentage of problems solved, although it still impacts the number of iterations needed to find a solution. For DSA, $P = 0.7$ gives the best results, though only $P = 0.1$ is significantly worse. For SoDSA-2 and SoDSA-4, the percentage of problems solved, though not 100%, is not significantly different when varying *P*. Interestingly, the best *P* value for speed is still 0.7 for SoDSA-2 and SoDSA-4. As the population size increases,

the performance variations become more statistically significant. Though the differences in performance are slight and not seemingly significant, as the problem gets harder, the *P* value needs to be smaller for smaller population sizes and increases with population size. Population size is still the most significant factor when comparing the percentage of problems solved.

The results for DSA and SoDSA on problems with a constraint tightness of 0.04 is shown in Tables 4.35 and 4.36. The problems here are considerably harder, and even SoDSA-32 is barely able to solve 100% of the problems. The effect of the *P* value is slightly more apparent and low values of *P* do not perform as well. The best performance in terms of the percentage of problems solved and the average iterations to solve a

	Population Size					
P	1	2	4	8	16	32
0.1	67.43	88.57	98.60	100.00	100.00	100.00
0.2	69.37	89.20	98.70	99.93	100.00	100.00
0.3	67.73	87.73	98.73	99.93	100.00	100.00
0.4	67.70	89.33	98.83	99.97	100.00	100.00
0.5	65.20	88.53	98.63	100.00	100.00	100.00
0.6	67.47	88.40	98.37	100.00	100.00	100.00
0.7	67.63	88.73	99.03	100.00	100.00	100.00
0.8	67.23	88.63	98.23	99.93	100.00	100.00
0.9	67.33	87.57	98.67	99.97	100.00	100.00

Table 4.33. Percentage of Problems Solved for DSA and SoDSA on problems with Constraint Tightness of 0.03

	Population Size					
P	1	2	4	8	16	32
0.1	759.46	341.25	115.45	56.38	43.22	34.99
0.2	669.43	274.89	68.40	29.49	21.71	18.20
0.3	682.04	283.53	54.77	20.74	14.85	12.45
0.4	674.72	243.75	45.21	15.40	11.33	9.55
0.5	717.31	254.15	44.86	11.90	9.30	7.84
0.6	669.47	251.40	47.33	10.15	7.89	6.79
0.7	662.99	241.98	32.44	8.57	7.00	5.98
0.8	668.46	242.53	46.73	9.24	6.31	5.47
0.9	665.17	261.93	36.79	7.92	5.86	5.06

Table 4.34. Average Iterations to Solve a Problem for DSA and SoDSA on problems with Constraint Tightness of 0.03

problem is still seen when *P* is closer to 0.9. Based on the ANOVA test results, any *P* value greater than 0.5 tend to have similar performances with insignificant variance. The general trend that the higher the population the better the performance still holds.

Closer to the phase transition, with a problem tightness of 0.05, a general performance trend appears in the results, as seen in Tables 4.37 and 4.38. The affect of the *P* value is more apparent here as the best performance is achieved with *P* = 0.4 to 0.5. The best performances are highlighted in Table 4.37 and 4.38, along with the neighboring values that are not significantly different. When comparing the average number of iterations to solve a problem, the clustering becomes tighter around 0.4 and 0.5 as the population size increases. Finally, at the phase transition tightness of 0.06, as seen in Tables 4.39 and 4.40, neither the DSA nor SoDSA are able to solve a significant number

	Population Size					
P	1	2	4	8	16	32
0.1	38.07	52.50	74.37	91.63	98.27	99.70
0.2	40.60	60.13	82.57	95.50	99.27	100.00
0.3	41.47	62.67	82.93	96.47	99.53	99.90
0.4	41.87	64.37	84.00	96.33	99.53	99.97
0.5	41.43	62.93	84.03	96.43	99.67	100.00
0.6	43.67	64.10	84.87	96.57	99.43	100.00
0.7	42.93	64.63	84.57	96.13	99.57	99.87
0.8	40.93	65.13	87.03	96.10	99.53	99.93
0.9	44.37	64.20	83.90	96.60	99.60	99.97

Table 4.35. Percentage of Problems Solved for DSA and SoDSA on problems with Constraint Tightness of 0.04

	Population Size					
P	1	2	4	8	16	32
0.1	1455.48	1263.43	894.93	544.77	298.36	171.68
0.2	1358.54	1047.84	640.29	332.87	159.59	87.30
0.3	1305.26	952.54	564.16	248.04	112.26	63.93
0.4	1275.89	873.10	502.93	216.80	92.68	45.70
0.5	1271.45	879.46	467.86	188.22	74.11	38.85
0.6	1226.70	849.75	450.86	171.91	72.19	36.78
0.7	1237.51	840.83	449.49	176.79	67.64	37.04
0.8	1265.26	828.82	406.49	185.29	74.47	36.37
0.9	1210.63	854.37	468.78	185.49	80.61	38.23

Table 4.36. Average Iterations to Solve a Problem for DSA and SoDSA on problems with Constraint Tightness of 0.04

of problems within the allotted 2000 iterations. However, SoDSA-32 is still able to solve significantly more problems than DSA.

Based on the results presented in Tables 4.29 – 4.40, several conclusions can be drawn. DSA, like dBA, gains a significant performance increase with a population based approach, like SoDSA. The logical trend of better performance with larger population size holds. The affect of *P* on the percentage of problems solved does not become significant until the tightness increases to 0.04, where the problem starts getting harder. When comparing the average number of iterations needed to solve a problem, if the problem is relatively easy, then a larger value of *P* is preferable, especially when using larger populations. As the problems get harder, a *P* value of around 0.4 to 0.5 becomes more preferable, regardless of population size. This trend can also be seen, to some extent, in the results at the phase transition, although it is not significant.

Tables 4.41 to 4.46 presents the average remaining constraint conflicts when a solution was not found within 2000 iterations. For the simpler problems with a constraint tightness of 0.01 to 0.03, there is very little dignificant variation between the average number of remaining conflicts when varying the *P* value. The difference starts to be significant starting with problems of tightness 0.04 to 0.06. Tables 4.44 to 4.46 has the *P* values with the lowest remaining conflicts and others that do not perform significantly worse highlighted. As can be seen when comparing the results in Tables 4.41 to 4.46 to those in Tables 4.29 to 4.40, having the fewest remaining conflicts when a solution was not found does not correlate to being able to solve more problems. The results, however, do reinforce certain performance characteristics of DSA and SoDSA. When the problems get very hard, with a tightness of 0.05 and 0.06, the value of *P* that yields the

	Population Size					
P	1	2	4	8	16	32
0.1	7.03	8.13	11.27	14.60	19.50	21.83
0.2	8.77	10.40	13.67	18.47	20.83	27.90
0.3	8.47	11.50	15.07	19.43	22.97	32.13
0.4	8.70	11.60	15.13	21.60	22.63	33.97
0.5	9.13	11.43	15.47	20.57	28.83	31.93
0.6	8.70	11.60	14.53	13.00	28.00	28.30
0.7	8.53	10.53	13.33	17.77	23.40	25.77
0.8	7.70	9.17	11.73	15.87	20.70	20.53
0.9	7.13	7.97	9.90	13.00	15.90	14.60

Table 4.37. Percentage of Problems Solved for DSA and SoDSA on problems with Constraint Tightness of 0.05

	Population Size					
P	1	2	4	8	16	32
0.1	1867.02	1857.65	1824.65	1787.57	1734.42	1740.55
0.2	1853.56	1829.12	1801.02	1741.12	1754.82	1649.18
0.3	1849.77	1818.69	1783.40	1724.63	1720.67	1587.02
0.4	1848.29	1816.30	1771.72	1686.63	1719.21	1548.72
0.5	1844.81	1816.42	1772.94	1697.64	1588.93	1569.27
0.6	1847.46	1817.68	1785.14	1843.27	1595.94	1615.86
0.7	1850.45	1823.16	1794.26	1741.07	1656.13	1656.69
0.8	1860.88	1844.30	1813.78	1758.31	1685.57	1726.44
0.9	1865.11	1858.80	1834.03	1797.75	1760.81	1806.90

Table 4.38 Average Iterations to Solve a Problem for DSA and SoDSA on problems with Constraint Tightness of 0.05

	Population Size					
P	1	2	4	8	16	32
0.1	0.00	0.03	0.00	0.13	0.10	0.30
0.2	0.00	0.00	0.17	0.10	0.20	0.47
0.3	0.00	0.03	0.07	0.20	0.17	0.60
0.4	0.07	0.03	0.07	0.23	0.23	0.67
0.5	0.03	0.07	0.07	0.17	0.20	0.40
0.6	0.00	0.03	0.07	0.10	0.13	0.50
0.7	0.00	0.03	0.00	0.13	0.13	0.13
0.8	0.00	0.00	0.03	0.03	0.10	0.17
0.9	0.00	0.00	0.00	0.00	0.00	0.00

Table 4.39. Percentage of Problems Solved for DSA and SoDSA on problems with Constraint Tightness of 0.06

	Population Size					
P	1	2	4	8	16	32
0.1	2000.00	1999.38	2000.00	1998.38	1999.34	1996.39
0.2	2000.00	2000.00	1998.11	1999.04	1997.80	1994.52
0.3	2000.00	1999.35	1999.06	1996.83	1998.28	1992.95
0.4	1999.48	1999.91	1998.88	1997.40	1998.49	1993.04
0.5	1999.69	1999.31	1999.01	1998.08	1997.40	1994.63
0.6	2000.00	1999.47	1999.60	1999.13	1999.34	1992.59
0.7	2000.00	1999.79	2000.00	1998.35	1998.21	1998.83
0.8	2000.00	2000.00	1999.95	1999.82	1998.25	1998.19
0.9	2000.00	2000.00	2000.00	2000.00	2000.00	2000.00

Table 4.40. Average Iterations to Solve a Problem for DSA and SoDSA on problems with Constraint Tightness of 0.06

fewest remaining conflicts when a solution was not found is actually 0.1, which is the lowest of all the tested values. In reference to Tables 4.37 and 4.39, it can be seen that a *P* value of 0.1 actually produces the worst results when it comes to percentage of problems solved. This is mainly because with a *P* value of 0.1, DSA and SoDSA are very prone to being stuck at a local optima as a minimum number of individuals in conflicts has a chance of changing their value each turn. The original motive for having multiple individuals change their values with a given probability is the hope that enough individuals will change their values simultaneously and move the search out of any local optima without the use of a breakout list like dBA. However, if too few individuals change their values each iteration, then the chance of the search moving out a local

	Population Size					
P	1	2	4	8	16	32
0.1	1.00	0.00	0.00	0.00	0.00	0.00
0.2	1.00	0.00	0.00	0.00	0.00	0.00
0.3	1.00	0.00	0.00	0.00	0.00	0.00
0.4	1.00	0.00	0.00	0.00	0.00	0.00
0.5	1.00	0.00	0.00	0.00	0.00	0.00
0.6	1.00	0.00	0.00	0.00	0.00	0.00
0.7	1.00	0.00	0.00	0.00	0.00	0.00
0.8	1.00	0.00	0.00	0.00	0.00	0.00
0.9	1.00	0.00	0.00	0.00	0.00	0.00

Table 4.41. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.01

	Population Size					
P	1	2	4	8	16	32
0.1	1.06	1.00	0.00	0.00	0.00	0.00
0.2	1.05	1.00	0.00	0.00	0.00	0.00
0.3	1.04	1.00	0.00	0.00	0.00	0.00
0.4	1.04	1.00	0.00	0.00	0.00	0.00
0.5	1.04	1.00	0.00	0.00	0.00	0.00
0.6	1.04	1.00	1.00	0.00	0.00	0.00
0.7	1.05	1.03	0.00	0.00	0.00	0.00
0.8	1.06	1.00	0.00	0.00	0.00	0.00
0.9	1.08	1.00	0.00	0.00	0.00	0.00

Table 4.42. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.02

	Population Size					
P	1	2	4	8	16	32
0.1	1.21	1.07	1.03	0.00	0.00	0.00
0.2	1.22	1.05	1.02	1.00	0.00	0.00
0.3	1.22	1.05	1.00	1.00	0.00	0.00
0.4	1.22	1.05	1.04	1.00	0.00	0.00
0.5	1.22	1.06	1.00	0.00	0.00	0.00
0.6	1.21	1.03	1.02	0.00	0.00	0.00
0.7	1.19	1.06	1.00	0.00	0.00	0.00
0.8	1.26	1.08	1.06	1.00	0.00	0.00
0.9	1.24	1.10	1.03	1.00	0.00	0.00

Table 4.43. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.03

	Population Size					
P	1	2	4	8	16	32
0.1	3.09	1.80	1.24	1.08	1.02	1.00
0.2	2.16	1.40	1.10	1.01	1.00	0.00
0.3	1.74	1.24	1.06	1.02	1.13	1.00
0.4	1.72	1.22	1.05	1.02	1.00	1.00
0.5	1.61	1.22	1.08	1.05	1.00	0.00
0.6	1.62	1.22	1.09	1.08	1.08	0.00
0.7	1.63	1.21	1.06	1.03	1.00	1.00
0.8	1.72	1.30	1.10	1.06	1.00	1.33
0.9	1.98	1.31	1.10	1.05	1.00	1.00

Table 4.44. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.04

	Population Size					
P	1	2	4	8	16	32
0.1	14.21	11.13	9.05	7.08	5.39	4.24
0.2	14.25	11.59	9.10	7.15	5.45	4.11
0.3	14.77	11.76	9.42	7.41	5.69	4.43
0.4	15.35	12.67	10.10	8.02	6.25	4.79
0.5	16.12	13.40	11.08	8.94	7.26	5.42
0.6	17.50	14.67	12.38	10.29	8.22	6.60
0.7	19.23	16.72	14.17	12.00	9.86	8.31
0.8	21.36	18.64	16.32	14.26	12.30	10.49
0.9	23.96	21.51	19.18	17.24	15.44	13.56

Table 4.45. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.05

optima becomes none. More on this will be discussed in the next section as the performance of DSA and SoDSA is analyzed.

Lastly, the average constraint checks required to find a solution within 2000 iterations is presented in Tables 4.47 to 4.52. The results show that being able to find a

	Population Size					
P	1	2	4	8	16	32
0.1	22.89	19.93	17.71	15.60	14.04	12.68
0.2	23.70	21.24	18.80	16.84	15.39	13.87
0.3	24.70	22.26	20.00	18.23	16.57	15.04
0.4	25.97	23.58	21.31	19.61	18.01	16.54
0.5	27.25	25.08	22.97	21.31	19.51	18.23
0.6	28.92	26.91	25.00	23.27	21.59	20.11
0.7	31.02	29.02	27.15	25.64	23.82	22.56
0.8	33.60	31.39	29.87	28.17	26.73	25.33
0.9	36.66	34.72	33.15	31.52	30.25	28.69

Table 4.46. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.06

solution faster does not guarantee that more of the problems can be solved especially when matched up with results from Tables 4.29 to 4.40. However, the results do support the trend that for easy problems, a *P* of 0.9 is capable of finding a solution faster. As the problems get harder, the value of *P* that can find a solution faster gradually decreases. The increase in population size has little to no affect on the optimal *P* value. More about the behavior of DSA and SoDSA based on these results will be discussed in the following section.

	Population Size					
P	1	2	4	8	16	32
0.1	41,685	65,740	105,760	176,341	297,063	508,054
0.2	20,931	33,235	56,464	95,962	165,764	293,084
0.3	14,193	23,341	40,131	69,510	122,623	219,751
0.4	10,815	17,905	31,386	55,917	101,298	185,324
0.5	8,748	14,896	26,478	47,639	87,791	164,322
0.6	7,266	12,740	23,192	42,503	78,813	148,412
0.7	6,389	11,178	20,399	38,247	72,145	137,219
0.8	5,527	10,089	18,635	35,424	67,138	128,769
0.9	4,989	9,212	17,275	32,927	63,140	122,411

Table 4.47. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.01

	Population Size					
P	1	2	4	8	16	32
0.1	86,919	139,098	226,117	391,216	689,928	1,228,311
0.2	43,364	70,891	118,292	203,861	365,618	660,315
0.3	29,074	47,885	80,408	142,589	257,036	470,370
0.4	21,834	36,676	63,142	112,275	203,059	378,377
0.5	17,647	30,236	51,638	93,783	172,951	322,183
0.6	15,340	25,793	45,109	81,694	152,425	285,120
0.7	12,952	23,167	39,888	73,333	137,512	260,195
0.8	11,697	20,661	36,498	67,440	126,992	242,672
0.9	10,377	18,906	33,958	63,224	119,856	229,029

Table 4.48. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.02

	Population Size					
P	1	2	4	8	16	32
0.1	256,549	406,027	600,091	863,054	1,420,850	2,465,145
0.2	133,667	210,828	304,376	450,904	730,207	1,289,255
0.3	89,595	148,621	207,276	310,585	514,120	908,315
0.4	66,529	108,046	160,070	240,069	400,827	717,769
0.5	58,510	88,966	129,819	204,270	336,525	602,268
0.6	45,160	74,597	111,059	174,661	295,864	532,329
0.7	40,168	65,244	98,844	153,526	269,789	484,582
0.8	35,660	57,667	87,048	143,971	249,641	456,520
0.9	33,251	59,110	84,299	137,809	238,890	436,341

Table 4.49. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.03

	Population Size					
P	1	2	4	8	16	32
0.1	1,185,962	2,159,166	3,814,223	6,058,160	8,158,710	11,095,129
0.2	885,914	1,536,605	2,461,745	3,776,954	4,767,772	5,825,573
0.3	687,533	1,202,676	1,972,869	2,669,788	3,270,514	4,109,543
0.4	580,197	1,009,777	1,621,180	2,167,323	2,616,436	3,386,825
0.5	542,240	881,336	1,420,254	1,932,303	2,397,269	2,819,954
0.6	534,785	870,588	1,343,411	1,777,263	2,140,710	2,640,954
0.7	501,859	783,004	1,238,076	1,736,596	2,145,547	2,526,749
0.8	497,755	850,363	1,391,140	1,916,964	2,206,104	2,621,437
0.9	577,536	992,885	1,596,807	2,229,644	2,701,790	3,065,091

Table 4.50. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.04

4.9. Performance Analysis of DSA and SoDSA

As mentioned earlier, the primary difference between the dBA and DSA is in that exactly one agent changes its value per iteration for dBA while a number of agents may probabilistically change their values in DSA. The *P* value determines this probability of

	Population Size					
P	1	2	4	8	16	32
0.1	2,325,052	4,328,498	9,170,672	18,590,982	34,692,857	66,021,711
0.2	2,310,771	4,580,092	9,468,491	17,515,021	31,836,127	58,624,391
0.3	2,332,829	4,708,799	8,377,293	16,451,018	30,736,035	59,254,702
0.4	2,291,876	4,406,440	8,711,331	16,039,103	31,136,762	58,211,324
0.5	2,605,810	4,779,735	8,281,816	16,271,251	30,799,112	56,490,934
0.6	2,216,820	4,839,562	8,226,317	16,664,668	29,440,260	59,474,756
0.7	2,190,048	4,749,221	9,381,499	17,467,788	30,556,004	61,914,500
0.8	2,622,188	4,936,391	9,937,838	18,615,795	33,850,608	63,897,159
0.9	2,509,306	5,038,406	10,823,231	19,877,491	36,606,913	70,720,779

Table 4.51. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.05

	Population Size					
P	1	2	4	8	16	32
0.1		733,555		17,773,339	60,698,015	74,809,270
0.2			9,825,664	24,759,523	42,431,858	77,215,742
0.3		269,555	7,277,405	10,086,828	45,439,114	78,835,952
0.4	3,840,760	10,766,250	4,034,625	21,628,304	65,141,250	92,230,027
0.5	3,499,285	6,234,638	6,270,380	21,777,927	36,011,523	65,147,014
0.6		2,652,630	18,402,965	30,049,993	76,339,491	54,381,032
0.7		8,711,890		19,080,695	35,744,929	119,338,806
0.8			26,412,040	41,709,105	14,515,950	103,706,929
0.9						

Table 4.52. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.06

change. Examining the rules by which an agent is allowed to change for DSA-B in Figure 3.5, it can be concluded that as long as an agent is involved in a conflict, then that agent will be allowed to change its value with probability *P*. The other rule, where agents are allowed a chance to change if they can reduce the number of conflicts, is ignored because it will never be triggered unless the agent is currently in a constraint conflict. This is due to the nature of asymmetric constraints.

It is tempting to assume that the number of agents that change their value each iteration is equal to $\text{total\_agents} \times P$, however this is not true. Based on the rules, only agents that are involved in a conflict can change with a probability. When the search first starts, most of the agents are likely to be involved in a constraint conflict, making it

highly likely that many agents will change their value each iteration. However, as the conflicts are resolved the number of agents that are expected to change their values drops. The worst case comes when only one agent is involved in a constraint conflict, which is very likely for DisACSPs. For normal DisCSPs, since the constraints are symmetric, at least 2 agents will be in a constraint conflict near the end of the search. With only one agent in a constraint conflict, if *P* is a small value, like 0.1, then it becomes likely that the search will stay in the same location for multiple iterations. With only one agent in a conflict and $P = 0.1$, the search may stay at the same position for up to 10 iterations before the agent in the conflict will change its value. This shows that a relatively small *P* value will significantly slow down the search when there are only a few conflicts left to resolve. Consequently, for easier problems a higher *P* value will actually help the search converge faster. This is supported by the results in Tables 4.41 to 4.43 and 4.47 to 4.49.

From an exploration versus exploitation point of view, the *P* value determines the amount of exploration at the beginning of the search and the amount of exploitation as the number of conflicts is reduced. This supports the results for the easier problems. Since there is a larger number of feasible solutions, extensive exploration at the beginning of the search can easily lead to or near a feasible solution. With high exploitation, once close to a solution, the search will converge to it fairly quickly. This is why the results for the easier problems favor a *P* of 0.7 to 0.9. As the problems increase in difficulty, high levels of exploration at the beginning of the search may not guarantee that the candidate solutions will improve, which is why a *P* of 0.4 to 0.5 become more favorable. This *P* value is low enough that the expected number of agents changing each iteration is small, and as the search nears a feasible solution and the number of conflicts

decreases, the *P* value is high enough that the search does not get stuck in one position for too long.

4.10. Results of the Genetic Operator on SoDSA

This section examines the results of the genetic SoDSA (GSoDSA) created from the addition of the distributed genetic operator to the SoDSA. The crossover and mutation rates for the distributed genetic operator is kept the same as GSoHC. The results are presented in Tables 4.53 – 4.64.

The results for problems with a constraint tightness of between 0.01 and 0.03 are not far from expectations as the *P* value's effect on performance is not very apparent when it comes to percentage of problems solved, and the differences are not statistically significant. When looking at the average number of cycles to solve a problem, the maximum tested *P* value still performs the best overall, along with the scaling of performance with population size.

As the problems get harder and closer to the phase transition, a very distinct shift of performance begins to appear, as shown in Tables 4.59 to 4.62, and the value of *P* that produces better performance starts to get smaller At a tightness of 0.04, a *P* value of around 0.6 and 0.7 produces the best results, though the performance of *P* between 0.5 and 0.8 are statistically similar. At a problem tightness of 0.05, the range of *P* that produces better performance is reduced to 0.2 to 0.4. This distinct drop of the optimal range of *P* can be attributed to the genetic operator, which produces a larger amount of exploration than the original SoDSA. Consequently, in order to compensate for the added level of exploration by the genetic operator, the amount of exploration SoDSA

	Population Size				
P	2	4	8	16	32
0.1	100.00	100.00	100.00	100.00	100.00
0.2	100.00	100.00	100.00	100.00	100.00
0.3	100.00	100.00	100.00	100.00	100.00
0.4	100.00	100.00	100.00	100.00	100.00
0.5	100.00	100.00	100.00	100.00	100.00
0.6	100.00	100.00	100.00	100.00	100.00
0.7	100.00	100.00	100.00	100.00	100.00
0.8	100.00	100.00	100.00	100.00	100.00
0.9	100.00	100.00	100.00	100.00	100.00

Table 4.53. Percentage of Problems Solved for GSoDSA on problems with Constraint Tightness of 0.01

	Population Size				
P	2	4	8	16	32
0.1	15.09	9.16	6.59	5.22	4.21
0.2	8.02	5.64	4.42	3.52	2.91
0.3	5.53	4.13	3.20	2.65	2.15
0.4	4.10	3.15	2.52	2.07	1.69
0.5	3.20	2.53	2.00	1.65	1.39
0.6	2.62	2.07	1.68	1.40	1.20
0.7	2.14	1.68	1.38	1.17	1.00
0.8	1.78	1.39	1.17	0.99	0.85
0.9	1.41	1.16	0.99	0.83	0.71

Table 4.54. Average Iterations to Solve a Problem for GSoDSA on problems with Constraint Tightness of 0.01

	Population Size				
P	2	4	8	16	32
0.1	98.57	99.97	100.00	100.00	100.00
0.2	98.90	99.97	100.00	100.00	100.00
0.3	99.40	100.00	100.00	100.00	100.00
0.4	99.00	100.00	100.00	100.00	100.00
0.5	99.03	100.00	100.00	100.00	100.00
0.6	98.90	100.00	100.00	100.00	100.00
0.7	98.97	100.00	100.00	100.00	100.00
0.8	98.80	99.93	100.00	100.00	100.00
0.9	98.97	100.00	100.00	100.00	100.00

Table 4.55. Percentage of Problems Solved for GSoDSA on problems with Constraint Tightness of 0.02

	Population Size				
P	2	4	8	16	32
0.1	61.93	22.29	16.59	13.44	11.40
0.2	40.85	13.27	10.07	8.41	7.15
0.3	24.60	8.85	7.24	6.11	5.30
0.4	29.02	6.95	5.65	4.83	4.14
0.5	26.59	5.64	4.59	3.96	3.42
0.6	28.07	4.73	3.90	3.32	2.87
0.7	25.65	3.97	3.30	2.86	2.50
0.8	28.78	4.77	2.89	2.51	2.21
0.9	24.93	3.09	2.58	2.23	1.98

Table 4.56. Average Iterations to Solve a Problem for GSoDSA on problems with Constraint Tightness of 0.02

	Population Size				
P	2	4	8	16	32
0.1	86.47	97.63	100.00	100.00	100.00
0.2	88.07	98.63	99.97	100.00	100.00
0.3	89.77	98.87	99.93	100.00	100.00
0.4	89.87	98.60	99.90	100.00	100.00
0.5	89.97	98.57	99.93	100.00	100.00
0.6	89.77	98.57	100.00	100.00	100.00
0.7	89.57	98.27	99.97	100.00	100.00
0.8	89.23	98.40	99.93	100.00	100.00
0.9	88.57	98.60	99.97	100.00	100.00

Table 4.57. Percentage of Problems Solved for GSoDSA on problems with Constraint Tightness of 0.03

	Population Size				
P	2	4	8	16	32
0.1	384.69	117.11	43.94	31.21	25.14
0.2	301.88	67.94	24.32	18.46	15.10
0.3	249.44	50.53	18.84	13.10	10.77
0.4	242.00	49.58	15.48	10.35	8.57
0.5	236.67	46.95	12.41	8.49	7.06
0.6	233.09	44.82	9.46	7.29	6.01
0.7	235.20	50.49	8.76	6.23	5.34
0.8	237.88	44.05	8.86	5.67	4.77
0.9	250.86	40.59	7.34	5.28	4.43

Table 4.58. Average Iterations to Solve a Problem for GSoDSA on problems with Constraint Tightness of 0.03

performs needs to be reduced for better performance. With problems at the phase transition, as shown in Tables 4.63 and 4.64, the performance of GSoDSA is not very impressive, though GSoDSA-32 is able to solve significantly more problems than the other population sizes, but, when $p = 0.9$, GSoDSA-32 cannot find a solution to any of the given problems.

Next, the average number of remaining constraint conflicts when a solution was not found within 2000 iterations is shown in Tables 4.65 to 4.70. The tables show that the average number of remaining conflicts for GSoDSA follow the same trend as those for SoDSA. Population size is one major factor that the remaining conflicts are lower, while the *p* value's contribution is similar to that of SoDSA. Overall, GSoDSA found better

	Population Size				
P	2	4	8	16	32
0.1	50.77	73.67	87.97	96.40	98.97
0.2	61.10	81.50	93.53	98.00	99.47
0.3	66.57	85.33	95.63	98.90	99.77
0.4	68.30	87.23	96.47	99.37	99.90
0.5	68.53	89.80	97.30	99.50	99.90
0.6	71.77	89.73	97.93	99.73	99.93
0.7	71.87	90.33	97.90	99.60	99.87
0.8	70.83	89.30	97.77	99.47	99.97
0.9	69.50	88.53	97.03	99.60	100.00

Table 4.59. Percentage of Problems Solved for GSoDSA on problems with Constraint Tightness of 0.04

	Population Size				
P	2	4	8	16	32
0.1	1291.11	851.48	499.38	246.40	127.38
0.2	1053.66	639.52	330.04	159.63	78.30
0.3	920.85	520.93	246.09	111.89	56.61
0.4	851.21	462.32	212.11	91.16	44.88
0.5	835.67	406.77	186.52	80.06	40.30
0.6	774.90	391.58	166.74	75.73	39.82
0.7	775.91	388.67	168.60	77.66	39.60
0.8	815.64	412.31	180.48	82.13	42.11
0.9	849.86	441.89	213.80	91.29	45.79

Table 4.60. Average Iterations to Solve a Problem for GSoDSA on problems with Constraint Tightness of 0.04

	Population Size				
P	2	4	8	16	32
0.1	4.03	9.27	17.67	26.67	37.07
0.2	6.70	13.70	22.13	34.53	44.90
0.3	7.10	13.60	22.47	35.07	45.93
0.4	6.53	11.90	21.27	32.87	42.83
0.5	5.80	11.63	17.63	27.17	38.63
0.6	5.67	8.70	15.07	22.97	32.63
0.7	3.70	7.37	12.10	18.20	26.17
0.8	3.67	5.93	8.73	12.87	19.17
0.9	2.40	3.80	6.23	9.33	12.00

Table 4.61. Percentage of Problems Solved for GSoDSA on problems with Constraint Tightness of 0.05

	Population Size				
P	2	4	8	16	32
0.1	1959.01	1900.23	1779.13	1640.21	1472.41
0.2	1927.32	1843.01	1721.30	1525.79	1340.06
0.3	1923.60	1839.05	1727.47	1515.99	1325.24
0.4	1926.64	1857.62	1735.02	1574.87	1387.45
0.5	1932.82	1862.20	1778.57	1647.19	1476.29
0.6	1938.00	1894.38	1813.77	1701.82	1557.05
0.7	1955.71	1914.69	1845.71	1763.74	1648.41
0.8	1959.65	1928.04	1894.71	1833.70	1739.07
0.9	1974.90	1959.06	1928.72	1882.94	1840.73

Table 4.62. Average Iterations to Solve a Problem for GSoDSA on problems with Constraint Tightness of 0.05

	Population Size				
P	2	4	8	16	32
0.1	0.00	0.00	0.03	0.17	0.43
0.2	0.00	0.00	0.03	0.00	0.40
0.3	0.00	0.00	0.03	0.03	0.30
0.4	0.00	0.00	0.00	0.00	0.50
0.5	0.00	0.00	0.00	0.00	0.47
0.6	0.00	0.00	0.00	0.00	0.63
0.7	0.00	0.00	0.00	0.00	0.37
0.8	0.00	0.00	0.00	0.00	0.20
0.9	0.00	0.00	0.00	0.00	0.00

Table 4.63. Percentage of Problems Solved for GSoDSA on problems with Constraint Tightness of 0.06

	Population Size				
P	2	4	8	16	32
0.1	2000.00	2000.00	1999.85	1998.96	1993.96
0.2	2000.00	2000.00	1999.58	2000.00	1994.71
0.3	2000.00	2000.00	1999.54	1999.59	1994.63
0.4	2000.00	2000.00	2000.00	2000.00	1992.80
0.5	2000.00	2000.00	2000.00	2000.00	1992.17
0.6	2000.00	2000.00	2000.00	2000.00	1993.84
0.7	2000.00	2000.00	2000.00	2000.00	1997.05
0.8	2000.00	2000.00	2000.00	2000.00	1998.17
0.9	2000.00	2000.00	2000.00	2000.00	2000.00

Table 4.64. Average Iterations to Solve a Problem for GSoDSA on problems with Constraint Tightness of 0.06

sub-optimal solutions than SoDSA when a feasible solution was not found within 2000 iterations.

Lastly, Tables 4.71 to 4.76 shows the average number of constraint checks performed when a solution was found within the 2000 iterations. The first noticeable difference between the results for GSoDSA and SoDSA is how fast the *p* value that requires the fewest constraint checks to solve a problem decreases for GSoDSA. At a problem tightness of 0.05, the *p* value that was able to find the solution with the fewest constraint checks is around 0.5 to 0.6 for SoDSA and around 0.1 to 0.2 for GSoDSA. This is due to the affect that the genetic operator has on the search, which is give it more exploration. Thus, the optimal *P* value decreases to increase exploitation. More about this will be discussed when comparing the results for SoDSA, GSoDSA and ESoDSA.

	Population Size				
P	2	4	8	16	32
0.1	0.00	0.00	0.00	0.00	0.00
0.2	0.00	0.00	0.00	0.00	0.00
0.3	0.00	0.00	0.00	0.00	0.00
0.4	0.00	0.00	0.00	0.00	0.00
0.5	0.00	0.00	0.00	0.00	0.00
0.6	0.00	0.00	0.00	0.00	0.00
0.7	0.00	0.00	0.00	0.00	0.00
0.8	0.00	0.00	0.00	0.00	0.00
0.9	0.00	0.00	0.00	0.00	0.00

Table 4.65. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.01

	Population Size				
P	2	4	8	16	32
0.1	1.00	1.00	0.00	0.00	0.00
0.2	1.02	1.00	0.00	0.00	0.00
0.3	1.00	0.00	0.00	0.00	0.00
0.4	1.00	0.00	0.00	0.00	0.00
0.5	1.03	0.00	0.00	0.00	0.00
0.6	1.00	0.00	0.00	0.00	0.00
0.7	1.03	0.00	0.00	0.00	0.00
0.8	1.00	1.00	0.00	0.00	0.00
0.9	1.06	0.00	0.00	0.00	0.00

Table 4.66. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.02

	Population Size				
P	2	4	8	16	32
0.1	1.05	1.04	0.00	0.00	0.00
0.2	1.03	1.00	1.00	0.00	0.00
0.3	1.03	1.00	1.00	0.00	0.00
0.4	1.03	1.00	1.00	0.00	0.00
0.5	1.03	1.00	1.00	0.00	0.00
0.6	1.03	1.00	0.00	0.00	0.00
0.7	1.03	1.00	1.00	0.00	0.00
0.8	1.03	1.08	1.00	0.00	0.00
0.9	1.04	1.00	1.00	0.00	0.00

Table 4.67. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.03

	Population Size				
P	2	4	8	16	32
0.1	1.60	1.21	1.07	1.03	1.00
0.2	1.30	1.10	1.08	1.02	1.00
0.3	1.19	1.08	1.02	1.06	1.00
0.4	1.20	1.06	1.08	1.00	1.00
0.5	1.15	1.05	1.04	1.00	1.00
0.6	1.11	1.05	1.02	1.00	1.00
0.7	1.19	1.05	1.01	1.00	1.00
0.8	1.18	1.06	1.01	1.00	1.00
0.9	1.36	1.05	1.04	1.00	0.00

Table 4.68. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.04

	Population Size				
P	2	4	8	16	32
0.1	8.49	4.84	3.12	2.18	1.74
0.2	9.70	6.30	3.91	2.81	2.00
0.3	11.01	7.69	5.43	3.73	2.51
0.4	12.22	9.24	6.93	5.27	3.80
0.5	13.28	10.51	8.52	6.67	4.88
0.6	15.01	12.13	9.96	8.25	6.30
0.7	16.65	14.30	11.87	9.78	8.00
0.8	19.01	16.77	14.33	12.35	10.06
0.9	21.52	19.41	17.08	15.30	13.32

Table 4.69. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.05

	Population Size				
P	2	4	8	16	32
0.1	17.89	13.56	10.25	7.97	6.37
0.2	20.18	16.71	14.01	11.57	9.76
0.3	21.91	19.06	16.59	14.46	12.61
0.4	23.54	20.93	18.70	16.69	15.14
0.5	25.25	22.70	20.82	19.18	17.36
0.6	27.20	24.97	22.99	21.16	19.83
0.7	29.16	27.40	25.50	23.78	22.34
0.8	31.81	29.91	28.21	26.64	25.24
0.9	34.96	33.34	31.53	29.99	28.71

Table 4.70. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.06

4.11. Results of the Evolutionary Operator on SoDSA (ESoDSA)

The distributed mutation operator can now be applied to SoDSA to reveal its possible impact on performance. The mutation rate for the operator is kept the same as ESoHC, $p_m = 0.12$, and the same test suit as before will be used. Because of the lack of a crossover operator, the mutation alone has a high level of exploitation, which impacts the performance of SoDSA in a slightly different way, especially in the optimal range of *P*.

Tables 4.77 – 4.82 present the results for ESoDSA on problems with a constraint tightness of 0.01 to 0.03. These results once again fall into similar performance expectations to those seen earlier, where the problems are simply not hard enough to show any affect of the mutation operator.

	Population Size				
P	2	4	8	16	32
0.1	42,546	57,967	92,888	160,227	284,336
0.2	25,915	40,503	69,775	124,310	225,624
0.3	19,619	32,654	57,941	104,430	191,212
0.4	16,035	27,803	50,095	91,551	170,471
0.5	13,791	24,358	44,438	82,380	154,854
0.6	12,166	21,956	40,513	76,110	144,178
0.7	10,877	19,981	37,652	70,637	135,233
0.8	9,970	18,490	35,188	66,519	128,457
0.9	9,172	17,238	32,929	63,150	122,033

Table 4.71. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.01

	Population Size				
P	2	4	8	16	32
0.1	102,566	145,857	237,461	411,906	733,971
0.2	57,211	90,503	157,333	281,920	514,864
0.3	44,094	69,843	122,615	222,657	407,743
0.4	33,976	57,130	102,065	186,849	347,583
0.5	29,047	48,892	87,895	164,146	306,926
0.6	24,836	43,376	79,084	147,910	278,000
0.7	23,549	39,398	72,251	136,086	256,060
0.8	20,527	36,577	67,019	127,032	241,345
0.9	19,313	33,756	63,246	120,028	228,468

Table 4.72. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.02

	Population Size				
P	2	4	8	16	32
0.1	363,380	427,246	589,690	933,469	1,604,005
0.2	203,002	251,855	361,039	599,891	1,049,869
0.3	147,786	191,279	275,402	457,794	814,457
0.4	121,919	154,437	225,647	376,971	674,967
0.5	108,915	131,849	191,576	326,989	585,819
0.6	95,033	115,437	168,313	291,768	524,849
0.7	81,784	110,627	156,852	269,447	486,328
0.8	83,470	97,092	145,060	251,344	455,404
0.9	75,314	92,664	140,255	239,688	436,742

Table 4.73. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.03

	Population Size				
P	2	4	8	16	32
0.1	1,745,151	2,517,435	3,620,975	4,674,241	6,146,909
0.2	1,319,736	1,925,526	2,561,595	3,236,842	4,146,459
0.3	1,170,564	1,661,341	2,094,692	2,483,850	3,297,897
0.4	1,029,465	1,472,869	1,946,699	2,288,040	2,809,025
0.5	991,445	1,458,829	1,830,966	2,050,615	2,625,208
0.6	986,464	1,464,432	1,833,424	2,130,577	2,427,032
0.7	1,067,161	1,492,505	1,948,996	2,188,829	2,583,201
0.8	1,117,592	1,633,978	2,110,944	2,349,707	2,789,906
0.9	1,239,523	1,926,397	2,520,491	2,726,721	3,234,453

Table 4.74. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.04

	Population Size				
P	2	4	8	16	32
0.1	3,593,778	6,329,153	10,751,783	18,053,987	29,752,111
0.2	3,663,114	6,231,130	10,513,397	18,371,181	30,798,587
0.3	4,106,944	6,738,343	11,346,353	19,685,627	35,854,586
0.4	4,176,077	6,646,806	12,844,431	22,844,991	40,812,985
0.5	4,423,049	8,130,322	12,908,288	24,962,242	45,062,051
0.6	4,458,580	8,331,941	14,357,599	25,769,490	51,494,768
0.7	5,156,076	8,474,265	16,745,830	27,800,473	53,428,556
0.8	5,186,621	10,006,428	18,197,907	30,699,647	60,136,035
0.9	5,934,815	11,362,293	20,762,531	39,277,046	66,188,174

Table 4.75. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.05

	Population Size				
P	2	4	8	16	32
0.1			12,369,032	34,312,158	62,741,138
0.2			6,109,140	16,283,334	39,000,375
0.3			2,737,999	14,966,864	23,235,259
0.4					16,560,794
0.5					44,527,776
0.6					68,852,839
0.7					103,312,926
0.8					105,794,349
0.9					

Table 4.76. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.06

As the problems get harder, the range of *P* that produces better performance begins to appear, as seen earlier for GSoDSA. However, the difference lies in the optimal value range of *P*. For GSoDSA, at a problem tightness of 0.04, the optimal range was around 0.6 to 0.7, while for ESoDSA this range is slightly higher at around 0.7 to 0.9. This is also true for problems with a tightness of 0.05. Where GSoDSA had an optimal range of around 0.2 to 0.3, ESoDSA's range is slightly higher at around 0.3 to 0.4. Once again, the performance of ESoDSA at the phase transition is still not very impressive, though ESoDSA-32 does perform significantly better than the others.

Tables 4.89 to 4.94 present the average number of remaining constraint conflicts for ESoDSA when a solution was not found within 2000 iterations. The results can best be described as extreme. In the worst case, ESoDSA performs worse than SoDSA, but much better than GSoDSA in the best cases. This is especially seen when the problem

	Population Size				
P	2	4	8	16	32
0.1	100.00	100.00	100.00	100.00	100.00
0.2	100.00	100.00	100.00	100.00	100.00
0.3	100.00	100.00	100.00	100.00	100.00
0.4	100.00	100.00	100.00	100.00	100.00
0.5	99.97	100.00	100.00	100.00	100.00
0.6	100.00	100.00	100.00	100.00	100.00
0.7	100.00	100.00	100.00	100.00	100.00
0.8	100.00	100.00	100.00	100.00	100.00
0.9	100.00	100.00	100.00	100.00	100.00

Table 4.77. Percentage of Problems Solved for ESoDSA on problems with Constraint Tightness of 0.01

	Population Size				
P	2	4	8	16	32
0.1	12.47	6.16	4.01	2.97	2.30
0.2	6.65	4.23	3.07	2.44	1.91
0.3	4.69	3.25	2.51	2.03	1.64
0.4	3.69	2.71	2.08	1.73	1.46
0.5	3.59	2.24	1.81	1.50	1.27
0.6	2.44	1.86	1.55	1.30	1.10
0.7	2.01	1.59	1.31	1.11	0.94
0.8	1.70	1.34	1.13	0.95	0.81
0.9	1.42	1.17	0.96	0.81	0.70

Table 4.78. Average Iterations to Solve a Problem for ESoDSA on problems with Constraint Tightness of 0.01

	Population Size				
P	2	4	8	16	32
0.1	98.37	99.97	100.00	100.00	100.00
0.2	98.30	100.00	100.00	100.00	100.00
0.3	98.60	100.00	100.00	100.00	100.00
0.4	98.90	100.00	100.00	100.00	100.00
0.5	98.77	100.00	100.00	100.00	100.00
0.6	99.10	100.00	100.00	100.00	100.00
0.7	99.13	99.97	100.00	100.00	100.00
0.8	98.97	100.00	100.00	100.00	100.00
0.9	98.80	99.97	100.00	100.00	100.00

Table 4.79. Percentage of Problems Solved for ESoDSA on problems with Constraint Tightness of 0.02

	Population Size				
P	2	4	8	16	32
0.1	62.95	18.53	12.47	9.59	7.82
0.2	50.35	10.96	8.28	6.83	5.69
0.3	39.39	8.04	6.43	5.36	4.60
0.4	30.50	6.42	5.16	4.38	3.81
0.5	31.51	5.30	4.34	3.71	3.26
0.6	23.83	4.52	3.72	3.21	2.79
0.7	22.29	4.54	3.23	2.81	2.45
0.8	25.59	3.41	2.86	2.46	2.20
0.9	27.80	3.71	2.55	2.23	1.98

Table 4.80. Average Iterations to Solve a Problem for ESoDSA on problems with Constraint Tightness of 0.02

	Population Size				
P	2	4	8	16	32
0.1	83.10	97.47	99.87	100.00	100.00
0.2	85.77	97.70	99.97	100.00	100.00
0.3	87.70	98.10	99.90	100.00	100.00
0.4	88.67	98.03	99.93	100.00	100.00
0.5	88.77	98.50	99.97	100.00	100.00
0.6	88.53	98.73	99.97	100.00	100.00
0.7	89.27	98.57	100.00	100.00	100.00
0.8	89.17	98.50	99.90	100.00	100.00
0.9	89.17	98.50	100.00	100.00	100.00

Table 4.81. Percentage of Problems Solved for ESoDSA on problems with Constraint Tightness of 0.03

	Population Size				
P	2	4	8	16	32
0.1	447.01	116.47	38.81	24.82	19.18
0.2	352.40	83.00	21.98	15.65	12.79
0.3	292.74	64.71	17.81	12.01	9.94
0.4	263.45	59.07	13.91	9.81	8.09
0.5	255.24	47.30	11.24	8.18	6.84
0.6	255.91	40.87	10.06	7.10	5.97
0.7	239.19	43.10	8.28	6.34	5.25
0.8	241.61	42.57	9.40	5.79	4.80
0.9	239.97	43.22	7.05	5.27	4.41

Table 4.82. Average Iterations to Solve a Problem for ESoDSA on problems with Constraint Tightness of 0.03

	Population Size				
P	2	4	8	16	32
0.1	47.07	66.90	84.50	94.17	98.27
0.2	57.50	74.13	91.37	97.20	98.83
0.3	62.47	83.00	94.00	97.67	98.93
0.4	65.53	83.10	94.17	98.47	99.70
0.5	68.50	86.87	96.00	98.83	99.83
0.6	68.60	85.63	96.67	98.97	99.90
0.7	68.30	89.00	96.43	99.40	99.83
0.8	70.60	90.27	97.73	99.43	99.90
0.9	70.23	88.77	96.60	99.47	99.90

Table 4.83. Percentage of Problems Solved for ESoDSA on problems with Constraint Tightness of 0.04

	Population Size				
P	2	4	8	16	32
0.1	1312.89	944.16	540.41	278.99	133.41
0.2	1091.57	718.80	338.06	156.15	77.69
0.3	960.71	523.94	258.34	123.99	68.17
0.4	879.19	507.35	232.04	99.11	47.33
0.5	814.16	430.57	196.95	87.31	42.24
0.6	813.62	450.75	180.20	83.95	36.25
0.7	824.90	398.08	188.74	78.04	39.71
0.8	801.59	393.25	168.26	79.56	42.31
0.9	836.74	450.20	219.56	93.44	47.80

Table 4.83. Average Iterations to Solve a Problem for ESoDSA on problems with Constraint Tightness of 0.04

	Population Size				
P	2	4	8	16	32
0.1	4.63	10.57	16.97	23.60	33.23
0.2	6.77	14.93	22.90	32.57	41.73
0.3	7.47	16.63	25.77	35.83	45.17
0.4	7.00	15.03	24.97	36.90	46.17
0.5	6.20	12.10	22.07	34.40	45.10
0.6	5.73	10.53	18.43	27.47	39.07
0.7	3.87	8.20	13.43	20.50	28.43
0.8	3.00	6.07	9.57	13.63	19.43
0.9	2.20	3.47	5.67	9.50	13.63

Table 4.84. Percentage of Problems Solved for ESoDSA on problems with Constraint Tightness of 0.05

	Population Size				
P	2	4	8	16	32
0.1	1949.09	1877.26	1783.94	1665.63	1490.03
0.2	1928.13	1815.48	1684.95	1507.86	1325.57
0.3	1911.12	1787.13	1651.71	1466.52	1270.28
0.4	1921.66	1812.52	1674.90	1466.29	1269.07
0.5	1926.06	1853.92	1711.94	1521.42	1321.96
0.6	1934.86	1876.34	1767.20	1640.10	1458.31
0.7	1955.72	1902.67	1831.69	1733.80	1606.67
0.8	1969.13	1930.05	1874.81	1822.35	1739.93
0.9	1976.11	1963.11	1929.04	1881.21	1815.97

Table 4.86. Average Iterations to Solve a Problem for ESoDSA on problems with Constraint Tightness of 0.05

	Population Size				
P	2	4	8	16	32
0.1	0.00	0.00	0.03	0.07	0.37
0.2	0.00	0.03	0.03	0.23	0.63
0.3	0.00	0.00	0.03	0.10	0.50
0.4	0.00	0.00	0.00	0.00	0.13
0.5	0.00	0.00	0.00	0.00	0.20
0.6	0.00	0.00	0.00	0.00	0.57
0.7	0.00	0.00	0.00	0.00	0.63
0.8	0.00	0.00	0.00	0.00	0.17
0.9	0.00	0.00	0.00	0.00	0.17

Table 4.87. Percentage of Problems Solved for ESoDSA on problems with Constraint Tightness of 0.06

	Population Size				
P	2	4	8	16	32
0.1	2000.00	2000.00	1999.70	1999.16	1994.98
0.2	2000.00	1999.80	1999.65	1997.20	1992.61
0.3	2000.00	2000.00	1999.83	1999.18	1992.98
0.4	2000.00	2000.00	2000.00	2000.00	1997.60
0.5	2000.00	2000.00	2000.00	2000.00	1996.54
0.6	2000.00	2000.00	2000.00	2000.00	1991.60
0.7	2000.00	2000.00	2000.00	2000.00	1991.75
0.8	2000.00	2000.00	2000.00	2000.00	1998.29
0.9	2000.00	2000.00	2000.00	2000.00	1997.96

Table 4.88. Average Iterations to Solve a Problem for ESoDSA on problems with Constraint Tightness of 0.06

tightness is 0.05 and 0.06. Another aspect of note is that ESoDSA performs worse than GSoDSA especially at low population sizes of 2 and 4.

Finally, Tables 4.95 to 4.99 shows the average number of constraint checks when a solution was found within 2000 iterations. The results for when the tightness is 0.06 is omitted, since ESoDSA was not able to solve a significant number of problems at the phase transition. Again, like GSoDSA, the advantage of the mutation operator does not show until the population size increases to a certain level. And when a solution is found, ESoDSA is capable of finding it faster than GSoDSA or SoDSA in the best case.

	Population Size				
P	2	4	8	16	32
0.1	1.00	0.00	0.00	0.00	0.00
0.2	1.00	1.00	0.00	0.00	0.00
0.3	1.00	0.00	0.00	0.00	0.00
0.4	1.00	0.00	0.00	0.00	0.00
0.5	1.00	0.00	0.00	0.00	0.00
0.6	1.00	1.00	0.00	0.00	0.00
0.7	1.00	0.00	0.00	0.00	0.00
0.8	1.00	0.00	0.00	0.00	0.00
0.9	1.00	0.00	0.00	0.00	0.00

Table 4.89. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.01

	Population Size				
P	2	4	8	16	32
0.1	1.00	1.00	1.00	1.00	0.00
0.2	1.02	1.00	1.00	0.00	0.00
0.3	1.01	1.00	1.00	0.00	0.00
0.4	1.00	1.00	0.00	0.00	0.00
0.5	1.01	1.00	1.00	0.00	0.00
0.6	1.00	1.00	1.00	0.00	0.00
0.7	1.00	1.00	1.00	0.00	0.00
0.8	1.01	1.00	1.00	0.00	0.00
0.9	1.00	1.00	1.00	0.00	0.00

Table 4.90. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.02

	Population Size				
P	2	4	8	16	32
0.1	1.07	1.03	1.01	1.03	1.00
0.2	1.06	1.03	1.02	1.00	1.00
0.3	1.06	1.03	1.01	1.00	1.00
0.4	1.05	1.03	1.00	1.00	1.00
0.5	1.06	1.02	1.00	1.00	1.00
0.6	1.06	1.02	1.00	1.00	1.00
0.7	1.06	1.03	1.01	1.00	1.00
0.8	1.06	1.02	1.02	1.00	1.00
0.9	1.08	1.04	1.00	1.00	1.00

Table 4.91. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.03

	Population Size				
P	2	4	8	16	32
0.1	2.11	1.22	1.10	1.07	1.02
0.2	1.74	1.21	1.10	1.05	1.03
0.3	1.56	1.20	1.11	1.07	1.02
0.4	1.52	1.21	1.12	1.07	1.04
0.5	1.39	1.21	1.13	1.08	1.05
0.6	1.46	1.21	1.15	1.07	1.03
0.7	1.51	1.22	1.15	1.11	1.04
0.8	1.65	1.24	1.15	1.10	1.04
0.9	2.20	1.23	1.18	1.13	1.07

Table 4.92. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.04

	Population Size				
P	2	4	8	16	32
0.1	11.96	5.76	2.56	1.87	1.76
0.2	12.81	6.60	2.75	1.93	1.74
0.3	13.64	7.35	2.97	1.98	1.79
0.4	14.99	8.41	3.53	2.20	1.84
0.5	16.73	9.71	4.09	2.29	1.95
0.6	19.04	12.07	5.48	2.71	2.09
0.7	22.46	15.09	7.82	3.24	2.21
0.8	29.43	20.20	11.53	4.93	2.60
0.9	40.60	28.10	18.17	9.11	3.86

Table 4.93. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.05

	Population Size				
P	2	4	8	16	32
0.1	20.67	15.01	9.89	6.42	5.16
0.2	22.31	17.31	12.54	8.24	5.91
0.3	23.89	19.55	14.97	10.42	7.08
0.4	26.18	21.99	17.67	12.94	9.03
0.5	29.32	24.94	20.63	15.90	11.49
0.6	34.31	28.93	24.41	19.56	14.62
0.7	42.29	34.50	29.39	24.34	18.82
0.8	53.10	43.88	36.93	30.15	23.77
0.9	68.14	56.46	45.49	36.81	29.09

Table 4.94. Average Remaining Conflicts when a solution was not found within 2000 iterations for problems with tightness of 0.06

	Population Size				
P	2	4	8	16	32
0.1	51,892	71,469	103,708	153,509	242,228
0.2	32,709	48,484	75,745	122,887	210,058
0.3	24,435	38,446	62,464	106,211	186,485
0.4	19,836	31,761	53,900	93,528	169,369
0.5	16,687	27,748	47,418	85,681	156,289
0.6	14,532	24,495	43,032	78,489	145,851
0.7	12,826	21,985	39,766	73,395	137,261
0.8	11,535	20,290	37,077	69,033	129,935
0.9	10,432	18,747	34,943	65,267	123,964

Table 4.95. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.01

	Population Size				
P	2	4	8	16	32
0.1	140,478	188,066	269,129	407,524	647,854
0.2	82,259	114,680	175,409	282,221	481,523
0.3	58,183	85,008	133,677	226,519	397,381
0.4	47,736	68,732	111,406	193,585	344,046
0.5	39,849	58,930	97,230	169,804	309,668
0.6	34,038	51,291	86,431	153,448	281,672
0.7	29,715	46,326	78,529	141,860	261,447
0.8	26,880	42,165	73,529	132,349	246,278
0.9	24,379	39,452	69,080	125,677	234,688

Table 4.96. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.02

	Population Size				
P	2	4	8	16	32
0.1	490,941	576,642	721,104	971,414	1,436,113
0.2	319,558	340,934	410,327	598,006	965,864
0.3	226,022	234,933	309,589	463,902	751,986
0.4	183,932	180,563	252,899	383,209	636,965
0.5	154,179	152,739	212,260	336,571	564,299
0.6	130,332	136,513	193,864	304,470	516,776
0.7	127,100	125,044	176,449	280,225	484,164
0.8	107,225	115,884	166,426	264,076	458,330
0.9	100,447	107,446	154,786	255,330	444,390

Table 4.97. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.03

	Population Size				
P	2	4	8	16	32
0.1	2,355,099	2,726,528	3,520,206	4,773,925	7,073,629
0.2	1,889,058	1,824,803	2,139,780	2,829,185	3,992,412
0.3	1,563,163	1,405,081	1,520,078	1,911,209	2,842,228
0.4	1,435,519	1,131,554	1,175,373	1,606,677	2,142,501
0.5	1,291,574	978,213	1,027,035	1,355,630	1,993,144
0.6	1,189,639	877,986	1,014,372	1,236,056	1,706,210
0.7	1,157,601	825,882	854,445	1,111,234	1,618,174
0.8	1,257,572	856,753	810,371	1,107,536	1,533,351
0.9	1,478,347	978,806	876,916	1,072,485	1,572,602

Table 4.98. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.04

	Population Size				
P	2	4	8	16	32
0.1	4,629,642	7,770,018	11,647,125	17,903,560	27,920,933
0.2	5,247,980	6,750,838	9,547,754	14,850,556	23,674,069
0.3	4,457,469	6,952,846	9,199,942	14,126,160	21,407,914
0.4	4,259,277	6,601,637	8,730,343	12,138,217	18,882,487
0.5	5,486,227	7,277,284	8,258,080	12,042,307	19,197,504
0.6	5,716,990	6,917,400	9,197,221	11,749,542	19,888,730
0.7	6,397,717	7,836,249	10,206,255	12,972,664	19,968,623
0.8	5,852,732	9,904,828	10,942,930	15,644,613	21,342,322
0.9	5,476,070	12,228,213	14,668,554	16,358,605	27,548,492

Table 4.99. Average Number of Constraint Checks to solve a problem when a solution was found within 2000 Iterations for problems with tightness 0.05

4.12. Performance Comparison of DSA, SoDSA, GSoDSA, ESoDSA

Table 4.100 shows the best possible results for SoDSA, GSoDSA, and ESoDSA over the percentage of problems solved for any given *P*. Not considering the possible affects of *P* on the results makes it possible to collect the best results in order to compare the impact of the genetic and evolutionary operators. For the relatively easier problems with constraint tightnesses of 0.01 to 0.03, there is no distinct difference between the performance of the three algorithms, SoDSA, GSoDSA, and ESoDSA. At a tightness of 0.04, SoDSA-2 falls slightly behind the performance of GSoDSA-2 and ESoDSA-2, but is able to catch up as the population size increases.

At a tightness of 0.05 and population sizes 2 and 4, SoDSA performs better than either GSoDSA and ESoDSA. This performance difference may be attributed to the fact that GSoDSA and ESoDSA have only half the population performing the DSA step, while the other half is applying either the genetic or evolutionary operator. At a population size of 8, GSoDSA and ESoDSA catches up to the performance of SoDSA as the benefits of the genetic and evolutionary operators become more apparent. At a population size of 32, both GSoDSA and ESoDSA perform significantly better than SoDSA.

At the phase transition, the performance of SoDSA seems to be slightly ahead, though not by a significant margin. The lead is taken over by GSoDSA and ESoDSA at a population size of 32, though still not by a significant amount.

Table 4.101 shows the best average number of iterations to solve a problem for SoDSA, GSoDSA, and ESoDSA. For the easier problems with tightnesses of 0.01 to

		Population Size					
Tightness		1	2	4	8	16	32
0.01	SoDSA	99.70	100.00	100.00	100.00	100.00	100.00
	GSoDSA		100.00	100.00	100.00	100.00	100.00
	ESoDSA		100.00	100.00	100.00	100.00	100.00
0.02	SoDSA	91.73	99.30	100.00	100.00	100.00	100.00
	GSoDSA		99.40	100.00	100.00	100.00	100.00
	ESoDSA		99.13	100.00	100.00	100.00	100.00
0.03	SoDSA	67.73	89.33	99.03	100.00	100.00	100.00
	GSoDSA		89.97	98.87	100.00	100.00	100.00
	ESoDSA		89.27	98.93	100.00	100.00	100.00
0.04	SoDSA	44.37	65.13	87.03	96.60	99.67	100.00
	GSoDSA		71.87	90.33	97.93	99.73	100.00
	ESoDSA		70.60	90.27	97.73	99.47	99.90
0.05	SoDSA	9.13	11.60	15.47	21.60	28.83	33.97
	GSoDSA		7.10	13.70	22.47	35.07	45.93
	ESoDSA		7.47	16.63	25.77	36.90	46.17
0.06	SoDSA	0.07	0.13	0.10	0.13	0.51	0.53
	GSoDSA		0.00	0.00	0.03	0.17	0.63
	ESoDSA		0.00	0.03	0.03	0.23	0.63

Table 4.100. Best Possible results for SoDSA, GSoDSA, and ESoDSA given any value of *P* (Percentage of Problems Solved)

0.03, the performances of the three are fairly similar. As would be expected, both GSoDSA and ESoDSA were able to find solutions faster on average than SoDSA. Interestingly, at a tightness of 0.04, GSoDSA is slightly slower than SoDSA and ESoDSA at finding a solutions with a population size of 32. At a tightness of 0.05, the performance differences fall back in line with those of the percentage of problems solved, where SoDSA performs better at lower populations and GSoDSA and ESoDSA perform better at higher populations.

The key factor to consider when comparing the results of SoDSA, GSoDSA, and ESoDSA is the balance of exploration and exploitation. As with the ant colony optimization [31] and all other forms of search algorithms, the proper balance of exploration and exploitation can have a major impact on the overall performance. Too much exploration and the search may not converge towards a solution, while too much exploitation and the search may get stuck at a local optimum.

		Population Size					
Tightness		1	2	4	8	16	32
0.01	SoDSA	9.08	2.30	2.03	1.85	1.71	1.59
	GSoDSA		1.41	1.16	0.99	0.83	0.71
	ESoDSA		1.42	1.17	0.96	0.81	0.70
0.02	SoDSA	171.16	21.60	3.85	3.36	3.03	2.77
	GSoDSA		24.60	3.09	2.58	2.23	1.98
	ESoDSA		22.29	3.41	2.55	2.23	1.98
0.03	SoDSA	662.99	241.98	32.44	7.92	5.86	5.06
	GSoDSA		233.09	40.59	7.34	5.28	4.43
	ESoDSA		239.97	40.87	7.05	5.27	4.41
0.04	SoDSA	1210.63	828.82	406.49	171.91	67.64	36.37
	GSoDSA		774.90	388.67	166.74	75.73	39.60
	ESoDSA		801.59	393.25	168.26	78.04	36.25
0.05	SoDSA	1844.81	1816.30	1771.72	1686.63	1588.93	1548.72
	GSoDSA		1923.60	1839.05	1721.30	1515.99	1325.24
	ESoDSA		1911.12	1787.13	1651.71	1466.29	1269.07
0.06	SoDSA	1999.48	1998.48	1998.65	1998.49	1992.35	1994.56
	GSoDSA		2000.00	2000.00	1999.54	1998.96	1992.17
	ESoDSA		2000.00	1999.80	1999.65	1997.20	1991.60

Table 4.101. Best Possible results for SoDSA, GSoDSA, and ESoDSA given any value of *p* (Average Number of Iterations to Solve a Problem)

DSA/SoDSA is special in that the amount of exploration performed by the search decreases over time. At the beginning of the search, when many agents are in constraint conflicts, a high value of *P* gives the search a decent amount of exploration and it is capable of possibly resolving multiple conflicts in one iteration as multiple agents change their values. A relatively small *P* in the beginning will slow exploration, but can turn out to be a conservative way of resolving conflicts. This is because as the problems get harder, having multiple agents change their values simultaneously is likely to create even more constraint conflicts. However, as the number of agents in constraint conflicts is reduced, the amount of exploration performed by DSA is also reduced, since fewer agents are expected to change their value each iteration. No matter how hard the problem is, when the number of agents in conflicts drops to a certain level, it becomes favorable to have a higher *P*, in order to maintain a certain level of exploration, because if *P* is too small the search will actually stagnate. Thus, for best results, a value of *P* must be chosen such that the exploration in the very beginning of the search is not too wide, and at the same time, the search is prevented from stagnating too quickly near the end. As discussed earlier, the optimal range of *P* for SoDSA near the phase transition, at a problem tightness of 0.05, is 0.4 to 0.5.

The addition of the genetic and evolutionary operators created a shift in this optimal *P* range by adding more exploration and some exploitation to SoDSA. The evolutionary operator, which uses mutation only, contributes a higher level of exploitation to SoDSA at the beginning of the search, but when the number of agents in conflicts drops below a certain point, the mutation operator actually helps exploration. For example, given that *P* = 0.5, when the number of agents in conflicts drops to 4, then for every iteration, the

number of agents expected to change their value is 2. However, a below average individual that invokes the mutation operator will become an 88% copy of the best individual, which in the current case with 30 agents, means that it is equivalent to changing 3 to 4 agents' values in one iteration. This behavior helps ESoDSA find a solution faster than SoDSA as it is able to search much more effectively around an area of promise than SoDSA. Thus, by denoting the number of agents in constraint conflicts as $agent_C$ and the total number of agents as $agent_T$, then the point beyond which the evolutionary operator starts contributing exploration to the search is when $agent_T \cdot p_m > agent_C \cdot P$. The genetic operator adds even more exploration to SoDSA, as the below average individuals effectively become 47% copies of the best individual. This exploration is much wider than that used by the evolutionary operator, which though still good at finding a solution, does not work as fast.

4.13. Final Comparison

Finally, it is useful to combine all the results for mdBA, SoHC, GSoHC, ESoHC, DSA, SoDSA, GSoDSA, and ESoDSA and compare their performance. Table 4.102 presents the percentage of problems solved for dBA and dBA based GEPs alongside the best of the DSA and DSA based GEPs. For problems with constraint tightnesses of 0.01 and 0.02, the problems are easy enough that the difference between the 6 algorithms are minimal. The most obvious result is how soon the performance of dBA and DSA starts to lag behind. The effect of population size is very obvious; at a problem tightness of 0.03, the DSA based GEPs are already lagging behind at a population size of 2. The performance gap widens as the problems' constraint tightness increases to 0.04.

However, as the population size increases, ESoHC and GSoHC are able to solve about twice as many problems as SoHC, SoDSA, GSoDSA, and ESoDSA. The performance advantage continues to show at the phase transition with population sizes of 16 and 32, where ESoHC and GSoHC solve at least twice as many problems as the other 4 algorithms.

		Population Size					
Tightness		1	2	4	8	16	32
0.01	SoHC	99.50	100.00	100.00	100.00	100.00	100.00
	GSoHC		100.00	100.00	100.00	100.00	100.00
	ESoHC		100.00	100.00	100.00	100.00	100.00
	SoDSA	99.70	100.00	100.00	100.00	100.00	100.00
	GSoDSA		100.00	100.00	100.00	100.00	100.00
	ESoDSA		100.00	100.00	100.00	100.00	100.00
0.02	SoHC	91.50	99.20	100.00	100.00	100.00	100.00
	GSoHC		100.00	100.00	100.00	100.00	100.00
	ESoHC		100.00	100.00	100.00	100.00	100.00
	SoDSA	91.73	99.30	100.00	100.00	100.00	100.00
	GSoDSA		99.40	100.00	100.00	100.00	100.00
	ESoDSA		99.13	100.00	100.00	100.00	100.00
0.03	SoHC	80.53	95.97	99.77	100.00	100.00	100.00
	GSoHC		99.97	100.00	100.00	100.00	100.00
	ESoHC		99.83	100.00	100.00	100.00	100.00
	SoDSA	67.73	89.33	99.03	100.00	100.00	100.00
	GSoDSA		89.97	98.87	100.00	100.00	100.00
	ESoDSA		89.27	98.93	100.00	100.00	100.00
0.04	SoHC	70.93	89.50	98.23	99.77	100.00	100.00
	GSoHC		95.10	99.90	100.00	100.00	100.00
	ESoHC		94.10	99.80	100.00	100.00	100.00
	SoDSA	44.37	65.13	87.03	96.60	99.67	100.00
	GSoDSA		71.87	90.33	97.93	99.73	100.00
	ESoDSA		70.60	90.27	97.73	99.47	99.90
0.05	SoHC	5.40	9.40	16.17	24.80	35.47	49.73
	GSoHC		8.20	29.90	61.23	84.50	93.93
	ESoHC		7.40	28.50	64.30	87.37	95.40
	SoDSA	9.13	11.60	15.47	21.60	28.83	33.97
	GSoDSA		7.10	13.70	22.47	35.07	45.93
	ESoDSA		7.47	16.63	25.77	36.90	46.17
0.06	SoHC	0.00	0.00	0.00	0.00	0.03	0.23
	GSoHC		0.00	0.23	0.80	1.10	2.30
	ESoHC		0.13	0.43	0.70	1.33	2.70
	SoDSA	0.07	0.13	0.10	0.13	0.51	0.53
	GSoDSA		0.00	0.00	0.03	0.17	0.63
	ESoDSA		0.00	0.03	0.03	0.23	0.63

Table 4.102. Comparison of Percentage of Problems Solved between the dBA and DSA variations

Table 4.103 presents the average number of iterations needed to solve a problem. Here, the DSA and DSA based GEPs have the distinct advantage of speed for problems with constraint tightnesses of 0.03 or less. The ability of DSA and DSA based GEPs to change more than one variable per iteration works to their advantage, making them up to twice as fast as the dBA and dBA based GEPs when finding a solution. However, this speed becomes a handicap as the problems get harder. With harder problems, it becomes less desirable to simultaneously change more than one variable each iteration, as this can create more new conflicts than it resolves.

Another possible reason for the poor performance of the DSA and DSA based GEPs when the problem gets harder is the algorithm's inability to slide. DSA is solely dependent on the chance that more than one agent will change its value to escape from a local optimum. However, for DisACSPs, it is very likely that the search will reach a point where only one agent is in a constraint conflict. Thus, based on the rules of DSA-B, only that agent is allowed the option of changing. However, if no value in that agent's domain causes another agent to be in a constraint conflict or resolve the one it is in, then the search becomes trapped in a local optimum. In these situations, the mdBA will simply “slide” and have a random variable change its value while laying down breakouts. The added population approach, used by SoDSA, helps resolve this situation somewhat, and the addition of the genetic and evolutionary operators further help to reduce the possibility of such a situation occurring.

Though the genetic and evolutionary operators are able to enhance the performance of DSA on harder problems, it is still insufficient to beat the consistency of SoHC,

		Population Size					
Tightness		1	2	4	8	16	32
0.01	SoHC	17.80	6.11	4.87	3.83	2.98	2.28
	GSoHC		5.69	4.26	3.38	2.63	2.00
	ESoHC		5.50	4.11	3.07	2.32	1.68
	SoDSA	9.08	2.30	2.03	1.85	1.71	1.59
	GSoDSA		1.41	1.16	0.99	0.83	0.71
	ESoDSA		1.42	1.17	0.96	0.81	0.70
0.02	SoHC	187.52	31.76	13.09	11.28	9.73	8.55
	GSoHC		16.90	11.74	9.62	8.22	7.12
	ESoHC		16.69	11.32	8.82	7.09	5.89
	SoDSA	171.16	21.60	3.85	3.36	3.03	2.77
	GSoDSA		24.60	3.09	2.58	2.23	1.98
	ESoDSA		22.29	3.41	2.55	2.23	1.98
0.03	SoHC	437.11	121.58	34.97	23.57	19.60	17.11
	GSoHC		52.53	28.60	20.14	16.15	13.70
	ESoHC		61.73	28.55	18.71	14.36	11.71
	SoDSA	662.99	241.98	32.44	7.92	5.86	5.06
	GSoDSA		233.09	40.59	7.34	5.28	4.43
	ESoDSA		239.97	40.87	7.05	5.27	4.41
0.04	SoHC	891.44	527.94	261.78	139.42	83.40	53.67
	GSoHC		493.54	167.14	77.83	46.06	32.36
	ESoHC		548.64	174.58	75.59	41.67	28.26
	SoDSA	1210.63	828.82	406.49	171.91	67.64	36.37
	GSoDSA		774.90	388.67	166.74	75.73	39.60
	ESoDSA		801.59	393.25	168.26	78.04	36.25
0.05	SoHC	1945.43	1898.77	1814.74	1702.74	1542.41	1334.73
	GSoHC		1912.67	1655.58	1186.17	692.19	392.25
	ESoHC		1919.95	1675.00	1131.62	634.82	349.35
	SoDSA	1844.81	1816.30	1771.72	1686.63	1588.93	1548.72
	GSoDSA		1923.60	1839.05	1721.30	1515.99	1325.24
	ESoDSA		1911.12	1787.13	1651.71	1466.29	1269.07
0.06	SoHC	2000.00	2000.00	2000.00	2000.00	1999.49	1997.79
	GSoHC		2000.00	1997.69	1989.41	1982.33	1966.50
	ESoHC		1998.10	1995.09	1988.67	1980.96	1961.25
	SoDSA	1999.48	1998.48	1998.65	1998.49	1992.35	1994.56
	GSoDSA		2000.00	2000.00	1999.54	1998.96	1992.17
	ESoDSA		2000.00	1999.80	1999.65	1997.20	1991.60

Table 4.103. Comparison of Average Number of Iterations to Solve a Problem between dBA and DSA variations

ESoHC, and GSoHC. However, when the problems are relatively simple, the DSA and DSA based GEPs offer a faster alternative to SoHC, GSoHC, and ESoHC.

4.14. An Adaptive SoDSA and the BreakOut List

As mentioned earlier, varying the value of *P* for the DSA and DSA based GEPs may possibly increase the performance and remove one parameter from consideration when implementing the algorithm. It is interesting to briefly look at possible ways of adapting the *p* value throughout the search process to see if the performance can be increased. The possibility and impact of adding the breakout management mechanism from SoHC into SoDSA is also considered.

To see if performance can be increased for the SoDSA by having an adaptive *P* value, a version of SoDSA can be created with an adaptive *p* value based on a simple rule and then tested on the same set of randomly generated DisACSPs. The main objective of implementing this adaptive *P* value is to keep the search from stagnating or remaining in one location too long due to the low probability of change and to try and mimic the behavior of dBA and SoHC, to a certain extent, they are known to perform well. So, for this test, simply let *P* equal the inverse of the number of agents currently in a constraint conflict. This guarantees that, on average, at least one agent will change its value every iteration, even when only one agent is in a constraint conflict. This effectively makes DSA behave similarly to dBA. The *P* value is updated every iteration after the number of conflicts are communicated. For a population, every distributed individual possesses its own *P* value. For comparison, the results from SoDSA with a fixed *P* value of 0.5 were arbitrarily selected. The percentage of problems solved can be seen in Table 4.104.

The results show that adaptive SoDSA (ASoDSA) can only, at best, match the performance of the SoDSA with a fixed *P* value. At a problem tightness of 0.05, the SoDSA with fixed *P* value performs significantly better than ASoDSA at all population

sizes. Table 4.105 shows the average number of iterations needed to find a solution and further reinforces the performance edge of the fixed SoDSA. With the exception of when the problem tightness is 0.01 and the population size is greater than 2, SoDSA is significantly faster than ASoDSA when it comes to finding a solution.

The adaptive *P* value becomes both an advantage and a disadvantage to ASoDSA. Near the end of the search, when the number of agents in conflict is relatively small, ASoDSA will converge or move around much faster than the standard SoDSA, since at least 1 agent will change its value each iteration. However, the way selected to adapt the

		Population Size					
Tightness		1	2	4	8	16	32
0.01	Adaptive	99.78	100.00	100.00	100.00	100.00	100.00
	Fixed	99.67	100.00	100.00	100.00	100.00	100.00
0.02	Adaptive	89.67	99.11	100.00	100.00	100.00	100.00
	Fixed	91.50	98.87	100.00	100.00	100.00	100.00
0.03	Adaptive	67.00	88.22	98.33	100.00	100.00	100.00
	Fixed	65.20	88.53	98.63	100.00	100.00	100.00
0.04	Adaptive	36.78	55.11	80.00	94.67	99.44	99.78
	Fixed	41.43	62.93	84.03	96.43	99.67	100.00
0.05	Adaptive	2.22	4.00	8.56	11.78	20.56	30.22
	Fixed	9.13	11.43	15.47	20.57	28.83	31.93
0.06	Adaptive	0.00	0.11	0.00	0.22	0.44	0.56
	Fixed	0.07	0.07	0.10	0.13	0.30	0.53

Table 4.104. Comparison of Adaptive SoDSA and Fixed SoDSA (*p*=0.5)

		Population Size					
Tightness		1	2	4	8	16	32
0.01	Adaptive	11.14	4.90	3.56	2.74	2.07	1.58
	Fixed	12.12	4.32	3.61	3.06	2.65	2.34
0.02	Adaptive	222.85	32.30	11.87	9.61	8.10	6.89
	Fixed	179.85	30.92	6.69	5.58	4.85	4.33
0.03	Adaptive	700.13	275.42	67.04	25.08	20.48	17.36
	Fixed	717.31	254.15	44.86	11.90	9.30	7.84
0.04	Adaptive	1450.11	1160.72	706.15	379.65	172.75	108.18
	Fixed	1271.45	879.46	467.86	188.22	74.11	38.85
0.05	Adaptive	1977.89	1962.34	1909.23	1871.73	1767.32	1629.25
	Fixed	1844.81	1816.42	1772.94	1697.64	1588.93	1569.27
0.06	Adaptive	2000.00	1998.71	2000.00	1997.84	1995.67	1994.79
	Fixed	1999.61	1999.23	1998.65	1998.49	1998.12	1996.00

Table 4.105. Comparison of Average Number of Iterations to Solve a problem between Adaptive-SoDSA and fixed SoDSA (*p*=0.5)

P value also effectively impacts the amount of exploration that can be performed compared to the fixed *P* value implementation. The inability for multiple agents to change their values slows down the search at the beginning, which is why for easier problems the standard SoDSA is faster at finding a solution. For slightly harder problems (tightness of 0.04), the adaptive *P* value creates a situation that is the opposite of expectations.

The way the *P* value is adapted effectively turns DSA into a simple hill-climber in the average case. This means that it will lock onto a gradient towards the closest local optimum and go straight towards it. Then, once the search progresses beyond the point where the number of agents in conflict reduces to a certain level, there is no way for the ASoDSA to escape the local optimum. Consequently, for an ASoDSA, each dCS will shoot for the closest local optimum and effectively get stuck there. For the standard SoDSA, the ability for multiple agents to change their values simultaneously means that it has a higher probability of avoiding this result, although ultimately the search will end in the same way. The result is that the standard SoDSA has a higher probability of finding a local optimum that is the global optimum, when compared to ASoDSA.

The next question therefore becomes whether a breakout list will help improve the performance of SoDSA and ASoDSA, since this implementation of ASoDSA seems to have run into the same problem many iterative improvement algorithms suffer from of being trapped at a local optimum. The addition of a breakout list would also mean that the distributed individuals will actually share some information about the search space instead of searching independently. The testing is primarily focused on problems of

tightness 0.04 to 0.06, as these problems are hard enough for a breakout list to make a difference.

Tables 4.106 and 4.107 show the results from the test runs on standard SoDSA with and without a breakout list. Statistical analysis was performed on the results to determine if the differences were significant, but the differences between SoDSA with and without a breakout list were not statistically significant except for when the problem tightness was 0.06 and the population size 16 and 32. In the two exception cases, SoDSA with breakout performed better. However, in general, the performance remained unchanged.

Moving on to the performance differences between ASoDSA with and without a breakout list, the results are shown in Tables 4.108 and 4.109. The differences between the performances are even closer than the differences between SoDSA with and without breakout. Statistically, adding a breakout list does not change the performance of ASoDSA.

		Population Size					
Tightness	BreakOut	1	2	4	8	16	32
0.04	With	37.33	60.11	82.33	93.00	98.67	99.78
	Without	41.43	62.93	84.03	96.43	99.67	100.00
0.05	With	4.22	9.22	15.33	22.89	31.22	35.78
	Without	9.13	11.43	15.47	20.57	28.83	31.93
0.06	With	0.11	0.00	0.00	0.33	1.44	1.89
	Without	0.07	0.07	0.10	0.13	0.30	0.53

Table 4.106. Comparison of SoDSA, with and without a breakout list, over the percentage of problems solved within 2000 iterations with *p*=0.5

		Population Size					
Tightness	BreakOut	1	2	4	8	16	32
0.04	With	1346.12	939.17	497.04	254.20	96.18	46.99
	Without	1271.45	879.46	467.86	188.22	74.11	38.85
0.05	With	1954.93	1898.81	1822.47	1711.27	1579.58	1450.77
	Without	1844.81	1816.42	1772.94	1697.64	1588.93	1569.27
0.06	With	1999.22	2000.00	2000.00	1995.77	1983.01	1976.06
	Without	1999.61	1999.23	1998.65	1998.49	1998.12	1996.00

Table 4.107. Comparison of SoDSA, with and without a breakout list, over the average number of cycles to solve a problem with *p*=0.5

		Population Size					
Tightness	BreakOut	1	2	4	8	16	32
0.04	With	40.67	64.89	81.67	92.11	96.78	99.11
	Without	36.78	55.11	80.00	94.67	99.44	99.78
0.05	With	2.22	2.78	8.22	11.44	17.44	25.00
	Without	2.22	4.00	8.56	11.78	20.56	30.22
0.06	With	0.11	0.00	0.00	0.11	0.56	0.78
	Without	0.00	0.11	0.00	0.22	0.44	0.56

Table 4.108. Comparison of ASoDSA, with and without a breakout list, over the percentage of problems solved within 2000 iterations with *p*=0.5

		Population Size					
Tightness	BreakOut	1	2	4	8	16	32
0.04	With	1387.54	1004.09	674.11	387.92	220.09	119.33
	Without	1450.11	1160.72	706.15	379.65	172.75	108.18
0.05	With	1979.11	1971.63	1908.31	1874.20	1788.28	1684.05
	Without	1977.89	1962.34	1909.23	1871.73	1767.32	1629.25
0.06	With	1998.89	2000.00	2000.00	1998.70	1992.04	1995.38
	Without	2000.00	1998.71	2000.00	1997.84	1995.67	1994.79

Table 4.109. Comparison of ASoDSA, with and without a breakout list, over the average number of cycles to solve a problem with *p*=0.5

The reason a breakout list does not significantly improve the performance of DSA (SoDSA) is due to the way DSA chooses the agent to change its value, especially at local optima. Unlike mdBA, which supports "sliding," the DSA only allows agents in conflicts to change. Because DSA is designed to actively reduce and resolve conflicts, a behavior similar to thrashing in backtrack algorithms can occur, especially when solving DisACSPs. Because an agent in a constraint conflict can only actively try to change its own value to resolve it, a situation may occur where the value assignment of agent X_i is creating conflicts with other agents, but locally it sees no conflict as the constraints are asymmetric. Thus, the other agents try to resolve the conflicts among themselves until either they find a solution or a value assignment in another agent triggers a constraint conflict in X_i, which finally causes it to change its value and resolve other outstanding constraint conflicts. This is less likely to happen for a mdBA, as agent X_i has a chance of changing its value whenever the search hits a local optimum. Because of the difference

in behavior at a local optimum, the breakout list has less of an affect or no affect on the performance of DSA/SoDSA than it does for mdBA/SoHC.

Though there are many other ways of modifying and utilizing the breakout list and the adaptive *P* value that could potentially improve the performance of DSA and SoDSA, it is beyond the scope of this research, which is more concerned with the implementation of genetic and evolutionary operators.

CHAPTER 5

THE SENSOR NETWORK

5.1. Introduction

This chapter, presents the sensor network and its architecture. Some issues related to the usage and implementation of a sensor network, like energy efficiency [1, 11, 25, 44, 71, 81, 104, 128, 138, 149, 163, 178] and packet routing [1, 11, 12, 68, 71, 138, 150, 151, 175, 178], will also be discussed. The sensor network tracking [3, 7, 30, 46, 87, 99, 132] and sharing problems [38] will then be stated in greater detail.

5.2. A Sensor Network

A sensor network can best be described as a collection of wirelessly connected, low cost pods containing various sensing devices that can be deployed over a specific geographical region for any number of purposes [1, 11, 151, 176]. Examples of sensor network applications include monitoring small localized changes in the environment [1, 11, 151, 176], target tracking [3, 7, 30, 46, 87, 99, 132, 176], and general data collection [1, 11, 151, 176]. A sensor network, as a tool, is scalable, robust, and can be highly efficient [1, 11, 151, 176].

Each sensor pod within the sensor network usually contains a wireless transceiver, multiple sensors, and some sort of power supply [1, 11, 151, 176]. This simplicity of

construction makes them small and low in cost. However, this also limits the tasks they can perform. Individually, each pod cannot collect much data, but as a collective network, the sensor network can collect large amounts of fine grain data as sensor pods are usually densely deployed [1, 11, 151, 176]. Pods communicate through the use of wireless communication, primarily broadcasting [1, 11, 151, 176]. All data that is collected is forwarded from one pod to the next until it reaches the sink or base station, set up to collect and analyze all the information.

The sensor network's scalability comes from the fact that pods can be added indefinitely to further increase the granularity of the data collected. The density of pods and the use of data forwarding makes the network robust and tolerant to pod failures, which usually occur frequently due to possible mechanical or power failure. The hopping of data from one pod to the next instead of sending information directly to the base station is much more power efficient [1, 11, 25, 68, 151, 176], as the power requirement for sending data directly from one node to another, without going through any intermediate nodes, increases exponentially in relation to the physical distance between the two nodes.

The study of sensor networks is currently a promising field of research because of the potential it offers as a tool and the various issues that arise from its applications [176]. The problems range from the implementation of software to the construction of the hardware. There are power issues [1, 11, 25, 44, 71, 81, 104, 128, 138, 149, 163, 178], networking issues [1, 11, 12, 100, 119, 128, 140, 155, 161, 176, 177, 178], and, more practically, issues involved with adapting it for specific applications [176]. This research focuses on problems that arise at the application layer pertaining to the software that runs

on the sensor network. Software solutions to the sensor network sharing [38] and tracking [3, 7, 30, 46, 87, 99, 130, 176] problems will be explored using the new genetic and evolutionary protocols.

5.3. Sensor Network Issues

Power is the primary constraint limiting a sensor network [1, 11, 151, 176]. Since each sensor pod usually carries nothing more than a battery with a fixed life span, the problem has been how to conserve energy while maintaining an acceptable performance. The single most power consuming task a sensor pod needs to perform is communication [1, 11, 151, 176], and topology and connectivity are the two primary factors affecting power consumption for communication [100, 119, 128, 155, 160, 178].

The problem of minimizing power use for communication can be solved by reducing the effective communication radius [155] of the pod and/or reducing the amount of traffic passing through the pod. Though reducing the communication radius is an effective way of reducing power consumption, there is the concern of losing network connectivity. The fear is that one pod failure may sever communications between a group of pods and the base station. Also, a pod whose failure will cause the network to lose connectivity will also be inherently more prone to failure, as more traffic will pass through it to reach the base station. These concerns make the problem of finding an energy efficient topology, that minimizes the power requirement for sending a packet between any two given nodes in the network nontrivial. It has also been shown that such a topology will be a minimum spanning tree and thus is an NP-hard problem [68].

Broadcasting is not the most energy efficient way of communicating data [1, 11, 151, 176] as the amount of packet traffic generated is $O(n^2)$, which is why various other routing methods have been devised for more energy efficient routing. These methods include direct diffusion [1, 11, 151, 176] and rumor routing [1, 11, 151, 176]. The two main strategies either call for a pod to ignore a packet completely if the pod is not on the packet's direct/established path towards the base station, or require it to aggregate information from multiple incoming packets into a single outgoing packet to reduce the number of active packets while ensuring the same amount of information is propagated.

5.4. The Sensor Tracking Problem

The tracking of mobile objects with a sensor network is a fundamental application [176]. Though this is often approached as a sensor network problem [3, 7, 30, 46, 87, 99, 130, 176], here it will be examined from a distributed CSP point of view, as formalized by Bejar et al [7], who referred to it as SensorCSP.

The sensor tracking problem involves the monitoring and following of moving targets within the coverage area by a network of stationary autonomous sensing devices [7, 108, 176]. Each sensor pod has a Doppler radar that is only capable of detecting relative distance and the general direction of a target from itself [108]. Thus, *k* sensor pods must work together and share distance and relative direction information to be able to triangulate and accurately pinpoint the actual position of the target. To effectively track a target, *k* of all sensor pods that can detect the target must be assigned to follow the target, but at the same time, these *k* sensor pods must also be able to directly communicate with each other to share the relative position data [7, 108]. A target is said

to be *k*-trackable [7], if, out of all the pods that are able to detect it, *k* pods that are capable of direct communication can be assigned to track it.

The sensor tracking problem is a distributed resource allocation problem, since the data that is collected about the target is distributed among multiple sensor pods. Along with resource requirements to accurately track an object, there are also communication constraints among sensors. Bejar et al [7] showed that the SensorCSP formulation of the sensor tracking problem is NP-complete and suggested the use of distributed CSP solvers such as asynchronous backtracking or distributed backtracking [7] to solve it.

To further illustrate the sensor tracking problem, Figure 5.1 presents a tracking scenario. The figure shows a total of 9 sensor pods and lines are used to represent the communication links between the pods. Note that not all pods can communicate directly with each other. Assuming that the sensing distance of a pod is equivalent to its communication distance, target A can then be detected by pods 4, 5, 7, and 8, while target B can be detected by pods 1, 2, 4, and 5. If 4 sensor pods must be able to detect a target and directly communicate with each other in order to properly track a target (a 4-trackable configuration), then a problem arises where targets A and B cannot be tracked simultaneously. However, if a 3-trackable solution is all that is needed, then a possible solution would have target A tracked by pods 5, 7, and 8 and target B tracked by pods 1, 2, and 4.

This research focuses on solving a specific version of the SensorCSP problem where targets need to be 3-trackable using the new evolutionary and genetic protocols. For testing purposes, assume that all sensors have perfect visibility and can detect anything within their area of deployment. Whether two pods can directly communicate is

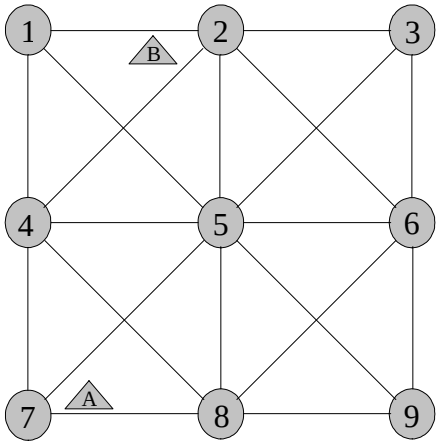

Figure 5.1. Sample Sensor Tracking Scenario

randomly determined based on a set communication density. The actual implementation and the results obtained will be presented in the next chapter.

5.5. The Sensor Sharing Problem

In general, there are three models for sensor network operations [150], namely continuous, event-driven, and user-initiated. A sensor network under the continuous model will start collecting data as soon as it is deployed and continue until all pods fail. An event-driven sensor network will idle until a specific phenomenon occurs or condition is met. At this point, the network starts collecting information until the event ends (sensor tracking). A user-initiated sensor network will idle until a user manually starts the data collection process and will continue collecting data until the user tells it to stop. The sensor network sharing problem arises from a user-initiated model.

The sensor network sharing problem [38] involves the allocation of limited sensor resources to satisfy as many user requests for sensors as possible. Each sensor pod contains *m* different sensors and is capable of turning on or off any one of the *m* sensors.

However, to reduce power consumption of the individual pods, assume that only one of the *m* sensors can be turned on in a sensor pod at any given time. Thus, users can request up to *n* sensors from the sensor network to collect data, where *n* is the number of pods in the network [25]. Each request will also have a time value associated with it that specifies how much sensor time must be allocated to the request to completely satisfy it. When a user places a request for *x* sensors, the network would then need to assign *x* pods to have the specified sensors turned on [38]. In addition to satisfying the user's sensor needs, the network must also satisfy sensor constraints in the form of internal allocation policies of each pod [38]. As more users make requests (and old requests are completed), the network will need to dynamically reassign sensors among pods so as to satisfy as many user requests as possible while not violating any of the internal allocation policies of the sensor pods.

For example, assume a sensor network with nine sensor pods, where each pod carries the same three types of sensor. When there are no outstanding requests for sensors, all pods are inactive and all the sensors are turned off. When a user makes a request for sensors, the request is broadcast from the sink to all the pods in the network. Upon reception of the request, the pods will “wake up” and negotiate a set of sensor assignments to satisfy the requests. Since each pod can only have one sensor active at any given time, with a sensor network of nine pods, a maximum of nine sensors can be requested by a user. A request for sensors made by a user can be represented by a sensor request vector, where the number of components of the vector is equivalent to the number of types of sensors that are available, *m*. Each component of the vector will then represent the number of sensors of that specific type required. So, a vector of <1, 3, 0>

represents a request for 1 sensor of type 1, 3 sensors of type 2 and none of type 3. To satisfy this sensor request vector, a total of four sensor pods must become active and turn on the proper sensor. Along with the sensor request vector, each request also contains a duration value, *t*, which specifies how long the sensor(s) requested must stay turned on. Assuming that $t = 10$, the full user request here would then be {<1, 3, 0>, 10}. Once the sensor and duration requirements are met, the request completes and expires. If there are no more requests active, then all sensor pods return to their original inactive state. Since there are only nine sensors pods, a sensor request like <5, 2, 3> cannot be accepted.

One of the benefits of a user-initiated sensor network is its inherently longer life span. When there are no user requests for resources, the entire network remains dormant, thus requiring each pod to expend little to no power. Once a request is issued, the network wakes up and the pods negotiate a means of satisfying the request. When a set of sensor assignments that satisfies the request is found, the rest of the podswho are not assigned to have an active sensor can once again idle. This is true mainly for those not on the forwarding path of data from the active pods to the sink.

The sensor sharing problem can be seen as a general form of the sensor tracking problem. Instead of tracking specific targets, the sharing problem involves the allocation of sensor resources for more general purposes. Similarly, the sensor sharing problem can also be modelled as a DisCSP.

CHAPTER 6

THE SENSOR TRACKING PROBLEM

6.1. Introduction

This chapter presents the implementation and testing of the sensor tracking problem. Specifically, the genetic and evolutionary protocols are tested by using them to solve the 3-trackable problem. For comparison, the modified DSA and the DSA based GEPs from Chapter 4 will also be used. Before the results are presented, it is first necessary to discuss the details of the implementation, the mapping of genetic and evolutionary protocols to the problem, and the testing methodology.

6.2. Problem Implementation

The details of the sensor tracking problem have been discussed several times in previous chapters. Here, the focus is on the details of the implementation, along with the assumptions made for the testing of the GEPs.

As mentioned in the previous chapter, the sensor tracking problem is treated as a DisCSP, as formulated by Bejar [7]. The first assumption is that all the pods are capable of detecting everything within their region of deployment. However, though pods have perfect visibility, direct communication between pods is not guaranteed. As with the asymmetric constraints used in Chapter 4, the communication between pods will also be asymmetric. Given two pods *A* and *B*, they can directly communicate with each other if,

and only if, *A* can communicate with *B* and *B* can communicate with *A*. The communication arcs used to determine whether pods can communicate will be generated based on an arc density, which is equivalent to the constraint density used in Chapter 4.

When a pod is not tracking a target, its radar will be switched to the off state to conserve energy. To accommodate this, the value 0 is reserved in the variable domain to represent the off state. The values in a variable's domain will thus represent the ID number of the target currently being tracked. So, with a domain size of 6, excluding 0, the network as a whole can track up to 6 targets simultaneously.

6.3. The Targets

Since all the sensor pods are assumed to have perfect visibility, where the virtual target is placed physically is not considered. It is assumed that the target is inserted within the region of deployment of the sensor network. The targets will appear at fixed intervals and will remain until they have been tracked for a specified amount of time. The arrival time between targets and the tracking time required for each target are varied. Note that a target is being tracked if, and only if, exactly 3 pods that can directly communicate with each other are assigned to monitor it.

6.4. Theoretical Analysis

Since the question of whether two pods can communicate with each other is determined probabilistically by a communication density, there exists a communication density at which there is no possible configuration by which a target can be successfully tracked. Similar to the way the phase transition is calculated for the general randomly

generated DisACSP, the process starts by estimating the number of possible solutions, *S*. For only one target that must be 3-trackable, the number of possible ways of tracking this target with *n* sensor pods, assuming all pods can directly communicate with each other, would be:

$$S = C_{n,3} = \frac{n!}{3!(n-3)!} \tag{6.3.1}$$

This, however, does not take into consideration the probability that two pods may not be able to communicate with each other. Assuming that the probability that any pod *A* can communicate with pod *B* is α and assuming that this communication is asymmetric, the probability of having 3 pods that can directly communicate with each other is $\alpha^{3\cdot(3-1)}$. Note that if the communication links between pods were to be symmetric, then the probability would instead be $\alpha^{3\cdot(3-1)/2}$. Combining this with 6.3.1 gives the expected number of ways to track a target with exactly 3 sensor pods, with the probability of communication between two pods being α.

$$S = C_{n,3}\alpha^{3(3-1)} = \frac{n!}{3!(n-3)!}\cdot\alpha^{6} \tag{6.3.2}$$

Based on 6.3.2, this can then be extended to cover the case where instead of one target, two targets need to be tracked. Once again, the communication density is ignored. If all pods can directly communicate with each other, then the number of ways to track two targets in a 3-trackable configuration with *n* sensor pods is:

$$S = C_{n,3}\cdot C_{n-3,3} \tag{6.3.3}$$

Next, again consider the communication density. The formulation of 6.3.2 reveals that the probability of any set of three randomly chosen pods being able to communicate directly with each other is α^{6}, where α is the probability that pod *A* can directly

communicate with pod *B*. So, to obtain the average number of configurations, *S*, for tracking two targets in a 3-trackable manner, α^6 must be factored in for every three pods chosen, giving the following:

$$S = C_{n,3}\alpha^6 \cdot C_{n-3,3}\alpha^6 = \frac{n!}{3!\,3!\,(n-6)!}\alpha^{12} \tag{6.3.4}$$

Based on 6.3.2 and 6.3.4, it is then possible to deduce that the expected number of solutions, *S*, for tracking *x* targets with *n* sensors pods and a communication density of α, in a *k*-trackable configuration is as follows:

$$S = \frac{n!}{(k!)^x\left[n-(x\cdot k)\right]!}\alpha^{x\cdot k\cdot(k-1)} \tag{6.3.5}$$

Based on 6.3.5, the communication density at which there is only one expected feasible solution would be:

$$\alpha = \left[\frac{n!}{(k!)!\left[n-(x\cdot k)\right]!}\right]^{\frac{-1}{x\cdot k\cdot(k-1)}} \tag{6.3.6}$$

Equation 6.3.6 predicts that the communication density at which the problem is toughest for tracking 10 targets in a sensor network of 30 sensor pods and in a 3-trackable configuration is around 0.388. This will be supported by the test results presented next.

6.5. Test Method

A total of 30 sensor pods were used to test the sensor tracking problem. Since the focus here is on the 3-trackable problem and 30 sensors pods can track a maximum of 10 targets, the domain size was therefore increased from the previous size of 6 to 10. The communication arc density varied between [0.4, 1.0] and the arc was generated using the same method as the random DisACPs in Chapter 4, with the exception that the constraint tightness was 0 and the constraint density was equal to the arc density.

The virtual targets were inserted into the network at fixed intervals and each had a specific life span. The time unit used was the cycles/iterations of the algorithm being executed. The target drop interval used for testing was {1, 2, 5}, so, a new target appeared every 1, 2, or 5 iterations. The amount of time that a target needed to be tracked also varied and was set to either {10, 50, 200}, which specifies the number of iterations that a target needed to be successfully tracked before disappearing. The protocols being tested ran a total of 1000 iterations.

Based on the differing combinations of arrival times and life spans, it was possible for more than 10 targets to be inserted into the network. In such cases, it was assumed that the first 10 targets in the queue would be active while the rest were on standby, in the queue. Thus, queuing time was also being meassured, and was intended to be as short as possible. Given the presence of queuing, the insertion of targets into a network becomes similar to placing a explicit tracking request into the network. Thus, the number of target tracking requests that could be satisfied during the 1000 iteration execution time also serves as a very important metric.

The thrid metric that was measured and collected was the number of stable cycles within the 1000 iterations. A stable cycle is defined as one where all active targets (up to 10) are being tracked successfully. This also implies that any two sensor pods that are assigned to track the same target will be able to directly communicate with each other. A stable cycle has the benefit of saving energy as no reconfiguration or reassignment of sensor pods needs to be performed. Ideally, a reassignment of tracking tasks should only occur when either a new target becomes active or when an active target has been satisfied.

The protocols and algorithms that were tested on the sensor tracking problem included the dBA based GEPs, SoHC, SoDSA, and DSA based GEPs. The population size was set to 32, as this gave the best results. For the genetic and evolutionary operators used, the settings for crossover and mutation rates were the same as those used in Chapter 4. For the DSA and DSA based GEPs, p was set to 0.1 as this permits no more than 3 agents to change their value each iteration.

6.6. Results

The results from using SoHC on the sensor tracking problem are presented in Table 6.1. As expected, when the communication density drops, the number of stable cycles decreases, the average queue time for a request increases, and the number of satisfied tracking requests decreases. The ideal number (maximum number) of stable cycles, average queue time and number of requests satisfied are also included as reference metrics for the performance. The maximum number of stable cycles is based on the ideal assumption that each new tracking request can be resolved in a single iteration. Based on the fact that the sensors are target agnostic, as long as the number of targets that need tracking stays the same, the network requires no reconfiguration. Consequently, the ideal number of stable cycles is simply $1000-\left[MIN\left(10, T_L/T_A\right)\right]$, where T_L represents the life span of a target and T_A is the arrival interval for the requests. The minimum average queue time is simply the interval between each new request, while the number of requests satisfied is also a function of T_L.

For the parameter settings that eventually have at least 10 (the maximum) tracking tasks in the queue, the number of stable cycles is fairly similar. Since any set of sensors

can track any target, once a stable configuration is found swapping out one target for another will not affect the stability of the network as this does not require a reassignment of tracking tasks. As the results show, however, this is only true for parameters that have a life span to arrival time ratio of 10 or above. The number of total iterations subtracted by the maximum number of possible stable cycles gives the average number of iterations needed to set up the initial configuration for tracking the maximum 10 targets with the network. So, for SoHC, at a communication density of 1.0 it takes about 20 iterations to find a stable configuration to track 10 targets. At a communication density of 0.8, the number of iterations increases to around 60, while at 0.6, the number of iterations varies from 300 – 400 iterations. Approaching the point where only one feasible solution is expected at a communication density of around 0.388, SoHC is unable to maintain a stable cycle except for when the average number of active targets in the queue drops below 10. This supports the previous prediction that the hard problems will be when the communication density gets close to 0.388. It should be noted that, based on the definition of a stable cycle, not being able to maintain a stable cycle does not mean that no 3-trackable configuration for a target exists. Rather, it implies that no 3-trackable configuration can be found such that the current set of targets can be tracked simultaneously.

The parameter setting with the worst performance, in terms of stable cycles, is when the arrival intervals of the targets is 2 iterations and the targets have a life span of 10 iterations. The low number of stable cycles is due to the constant need to reconfigure sensor assignments. Since SoHC is only able to change the assignment of one agent in any given iteration, this implies that it would take at least 3 iterations to find a 3-trackable

configuration for any target. This, in combination with the arrival interval of the targets and their life spans, creates a situation where for the majority of the 1000 iterations, the number of targets that must be tracked fluctuates between 5 to 8. The constant need to reassign a sensor creates the large number of unstable cycles. In contrast, for the parameter setting with arrival intervals of 5 and life spans of 10, the number of targets being tracked at any given time is much fewer and more constant, which makes it possible to maintain a much larger number of stable cycles.

The results of applying the GEPs (GSoHC and ESoHC) to the tracking problem using the same parameters are shown in Tables 6.2 and 6.3. As seen in Chapter 4, on randomly generated DisACSPs, ESoHC is able to find solutions faster than GSoHC, which in turn is faster than SoHC. The same trend is seen here for the number of stable cycles. ESoHC is able to maintain a higher number of stable cycles since it is able to find

Arrival interval	1	1	1	2	2	2	5	5	5
Life Span	10	50	200	10	50	200	10	50	200
Density	Stable Cycles								
0.4	0.00	0.00	0.00	0.06	0.00	0.00	280.85	1.23	1.23
0.6	640.38	606.18	623.83	561.52	673.30	597.71	933.68	643.32	640.75
0.8	931.00	932.51	932.65	645.14	931.85	932.09	977.89	929.44	930.52
1.0	970.97	970.96	970.96	724.63	970.96	970.97	989.00	970.97	970.96
Ideal	990.00	990.00	990.00	995.00	990.00	990.00	998.00	990.00	990.00
	Average Queue Time								
0.4	42.96	221.68	759.35	44.22	227.04	760.21	42.69	217.95	727.70
0.6	11.44	58.23	231.49	11.98	56.25	232.83	10.35	56.34	226.83
0.8	10.28	51.41	206.62	10.63	51.10	205.19	10.07	50.63	202.92
1.0	10.15	50.78	203.72	10.23	50.45	202.12	10.04	50.15	200.72
Ideal	10.00	50.00	200.00	10.00	50.00	200.00	10.00	50.00	200.00
	Requests Satisfied								
0.4	247.68	42.95	7.04	242.09	41.96	6.62	177.25	45.15	7.78
0.6	869.75	167.86	39.02	495.84	172.18	38.68	199.00	169.88	39.20
0.8	963.83	189.44	40.00	495.85	189.47	40.00	198.99	187.68	40.00
1.0	975.76	190.00	40.00	495.86	190.00	40.00	199.00	190.00	40.00
Ideal	990.00	190.00	40.00	499.00	190.00	40.00	199.00	190.00	40.00

Table 6.1. Results of SoHC on the Sensor Tracking Problem over all parameter settings

a solution faster. However, the results for a communication density of 0.4 reveal that SoHC actually performs better in terms of the average queue time and number of requests satisfied. The reason for this lies in the issue of population diversity. SoHC has a more diverse population than either ESoHC or GSoHC. The use of genetic and evolutionary operators, due to their emphasis on exploitation, causes half the population to be largely focused around a single solution. This becomes a problem when a different solution needs to be found. For the GEPs, their population is not as diverse, especially when a solution is found, which makes it difficult for GEPs to move from the previous solution to a new one as fast when the requirements change. SoHC, however, with its more diverse population is not as prone to this weakness.

A closer look at the performance of SoHC and the GEPs at a communication density of 0.4 in Table 6.4, reveals some interesting results. The differences in the number of stable cycles for the three algorithms are either non-existant or not statistically significant, with the exception of the parameter setting for arrival interval and life span of (5, 10), which is the simplest of all the parameter settings. At a setting of (5, 10) ESoHC has an edge due to its ability to quickly find a solution. GSoHC does not perform significantly worse than ESoHC and is much more stable than SoHC.

When comparing the average queuing time for tracking tasks, GSoHC is only able to perform significantly better on the easiest problem setting. In general, the performance of GSoHC improves as the life span of a target increases and its performance when targets have a life span of 200 iterations is statistically similar to the performance of SoHC. Overall, because ESoHC is still able to quickly find a solution when one is needed, it is able to effectively reduce queuing time for new targets. The performance of

Arrival interval	1	1	1	2	2	2	5	5	5
Life Span	10	50	200	10	50	200	10	50	200
Density	Stable Cycles								
0.4	0.00	0.00	0.00	2.76	0.00	0.00	451.29	0.77	0.75
0.6	661.87	699.16	697.00	635.58	698.59	694.63	943.46	672.62	682.82
0.8	944.64	944.13	945.28	746.10	942.43	942.65	980.91	936.02	932.93
1.0	979.85	980.03	979.92	641.67	974.96	974.95	989.85	972.24	972.21
Ideal	990.00	990.00	990.00	995.00	990.00	990.00	998.00	990.00	990.00
	Average Queue Time								
0.4	50.62	240.74	756.35	49.90	252.97	742.24	33.74	233.73	692.59
0.6	11.45	56.37	228.16	12.09	56.26	226.85	10.38	55.82	224.50
0.8	10.21	51.09	205.07	10.52	50.94	204.43	10.06	50.56	202.85
1.0	10.08	50.40	201.91	10.24	50.27	201.28	10.04	50.15	200.70
Ideal	10.00	50.00	200.00	10.00	50.00	200.00	10.00	50.00	200.00
	Requests Satisfied								
0.4	209.82	40.33	6.36	223.36	37.62	6.52	183.07	40.97	7.39
0.6	870.22	172.51	39.61	495.95	171.98	39.65	199.00	171.16	39.35
0.8	970.33	189.90	40.00	495.93	189.84	40.00	199.00	187.97	40.00
1.0	983.44	190.00	40.00	495.78	190.00	40.00	199.00	190.00	40.00
Ideal	990.00	190.00	40.00	499.00	190.00	40.00	199.00	190.00	40.00

Table 6.2. Results of GSoHC on the Sensor Tracking Problem over all parameter settings

Arrival interval	1	1	1	2	2	2	5	5	5
Life Span	10	50	200	10	50	200	10	50	200
Density	Stable Cycles								
0.4	0.00	0.00	0.00	0.00	0.00	0.00	476.36	0.65	0.58
0.6	710.03	703.98	694.51	605.59	720.63	729.83	948.84	698.44	727.41
0.8	951.95	951.51	952.42	679.28	948.74	947.86	984.89	936.80	937.48
1.0	983.19	983.05	983.16	611.53	976.38	976.39	989.14	972.27	972.23
Ideal	990.00	990.00	990.00	995.00	990.00	990.00	998.00	990.00	990.00
	Average Queue Time								
0.4	46.06	217.59	684.06	42.61	218.80	659.24	46.06	204.99	629.64
0.6	11.20	56.20	227.52	11.78	55.63	223.28	11.20	55.19	219.89
0.8	10.18	50.92	204.31	10.48	50.81	203.86	10.18	50.54	202.50
1.0	10.05	50.29	201.34	10.25	50.21	201.03	10.05	50.15	200.69
Ideal	10.00	50.00	200.00	10.00	50.00	200.00	10.00	50.00	200.00
	Requests Satisfied								
0.4	230.51	41.54	6.66	237.71	40.57	7.48	185.89	44.50	7.23
0.6	886.19	173.12	39.52	495.88	173.96	39.86	199.00	173.09	39.86
0.8	973.65	189.96	40.00	495.85	189.95	40.00	199.00	188.12	40.00
1.0	985.42	190.00	40.00	495.78	190.00	40.00	199.00	190.00	40.00
Ideal	990.00	190.00	40.00	499.00	190.00	40.00	199.00	190.00	40.00

Table 6.3. Results of ESoHC on the Sensor Tracking Problem over all parameter settings

ESoHC is not statistically different from that of SoHC when a small life span is assigned to the targets, but when the life span increases to 200 iterations ESoHC becomes significantly better than either SoHC or GSoHC.

When comparing the number of requests satisfied (targets successfully tracked for a fixed number of iterations), SoHC performs slightly better than the other two methods, with the exception, again, of when the problem is the easiest. ESoHC is closely behind SoHC, as their performances are statistically similar most of the time. Consequently, factoring in the number of stable cycles, the average queuing time, and the number of requests satisfied in 1000 iterations, ESoHC and SoHC are both good choices as their performances are statistically similar in many cases.

SoDSA, GSoDSA, and ESoDSA were also implemented to solve the tracking problem. The results are shown in Tables 6.5 to 6.7. The value of *p* was chosen to be 0.1, since this would guarantee that no more than 3 agents will change their values at the same time. It is very obvious that this problem is not easy for the DSA variations. SoDSA, GSoDSA, and ESoDSA barely keep up with the performance of SoHC, and the GEPs at a communication density of 1.0. As the communication density is reduced, the

Arrival interval	1	1	1	2	2	2	5	5	5
Life Span	10	50	200	10	50	200	10	50	200
	Stable Cycles								
SoHC	0.00	0.00	0.00	0.06	0.00	0.00	280.85	1.23	1.23
GSoHC	0.00	0.00	0.00	2.76	0.00	0.00	451.29	0.77	0.75
ESoHC	0.00	0.00	0.00	0.00	0.00	0.00	476.36	0.65	0.58
	Average Queue Time								
SoHC	42.96	221.68	759.35	44.22	227.04	760.21	42.69	217.95	727.70
GSoHC	50.62	240.74	756.35	49.90	252.97	742.24	33.74	233.73	692.59
ESoHC	46.06	217.59	684.06	42.61	218.80	659.24	46.06	204.99	629.64
	Requests Satisfied								
SoHC	247.68	42.95	7.04	242.09	41.96	6.62	177.25	45.15	7.78
GSoHC	209.82	40.33	6.36	223.36	37.62	6.52	183.07	40.97	7.39
ESoHC	230.51	41.54	6.66	237.71	40.57	7.48	185.89	44.50	7.23

Table 6.4. Comparison of results for SoHC, GSoHC, and ESoHC at a communication density of 0.4

performance of the DSA variations all drop drastically. Based on these results, it seems that the DSA variations have trouble holding onto even a semi-stable configuration, especially when the communication density is less than 1.

Table 6.8 shows a comparison of the 3 DSA and DSA based GEPs at a communication density of 0.4. The number of stable cycles is virtually identifical as none of the algorithms could maintain a stable cycle. For the average queue time, ESoDSA gives the lowest average queue time for problem settings that result in at least 10 simultaneous target tracking tasks for the majority of the 1000 iteration test run. For the not-so-hard problem of targets arriving every 2 iterations and staying for 10 iterations, GSoDSA achieves the lowest average queue time for the requests, while for the easiest problem setting, SoDSA performs the best. The results for the number of requests satisfied follows the same performance trend as the average queue time. These results

Arrival interval	1	1	1	2	2	2	5	5	5
Life Span	10	50	200	10	50	200	10	50	200
Density	Stable Cycles								
0.4	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01
0.6	0.00	0.00	0.00	0.01	0.00	0.00	0.70	0.15	0.20
0.8	14.94	14.53	22.95	0.87	12.68	19.37	824.74	21.51	24.45
1.0	957.09	956.53	955.29	664.93	954.16	954.59	963.56	949.28	950.04
Ideal	990.00	990.00	990.00	995.00	990.00	990.00	998.00	990.00	990.00
	Average Queue Time								
0.4	207.95	798.13	1000.00	215.66	821.16	1000.00	197.77	803.18	1000.00
0.6	51.13	264.64	904.09	54.73	263.96	896.94	52.65	258.53	891.98
0.8	20.30	100.32	400.53	20.52	101.44	403.10	11.24	100.59	398.10
1.0	10.17	50.86	204.21	10.25	50.77	203.63	10.05	50.71	203.27
Ideal	10.00	50.00	200.00	10.00	50.00	200.00	10.00	50.00	200.00
	Requests Satisfied								
0.4	44.97	5.84	0.00	44.60	4.49	0.00	49.00	4.91	0.00
0.6	190.98	32.76	3.70	177.98	32.60	3.89	177.09	32.90	3.50
0.8	489.50	94.70	20.37	473.93	93.19	20.17	199.00	93.07	20.21
1.0	974.85	189.99	40.00	495.86	189.99	40.00	199.00	188.33	40.00
Ideal	990.00	190.00	40.00	499.00	190.00	40.00	199.00	190.00	40.00

Table 6.5. Results of SoDSA (p=0.1) on the Sensor Tracking Problem over all parameter settings

Arrival interval	1	1	1	2	2	2	5	5	5
Life Span	10	50	200	10	50	200	10	50	200
Density	Stable Cycles								
0.4	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01
0.6	0.00	0.00	0.00	0.01	0.01	0.01	4.21	0.19	0.16
0.8	18.05	27.29	20.51	0.84	20.00	21.71	826.76	14.01	21.63
1.0	956.02	956.30	956.73	660.22	953.93	953.33	964.54	949.87	948.82
Ideal	990.00	990.00	990.00	995.00	990.00	990.00	998.00	990.00	990.00
	Average Queue Time								
0.4	211.37	791.97	1000.00	195.70	827.33	1000.00	200.30	805.15	1000.00
0.6	51.44	263.63	899.26	52.54	268.30	896.85	52.48	261.53	888.21
0.8	20.29	100.56	400.63	20.53	100.08	400.26	11.20	100.21	405.92
1.0	10.17	50.87	204.06	10.25	50.78	203.74	10.05	50.70	203.37
Ideal	10.00	50.00	200.00	10.00	50.00	200.00	10.00	50.00	200.00
	Requests Satisfied								
0.4	45.78	5.40	0.00	49.23	4.69	0.00	46.83	5.06	0.00
0.6	189.27	32.92	3.96	186.64	32.21	4.02	176.62	32.47	4.07
0.8	490.29	95.14	20.35	474.09	94.83	20.30	199.00	93.01	19.81
1.0	974.41	189.99	40.00	495.85	189.98	40.00	199.00	188.38	40.00
Ideal	990.00	190.00	40.00	499.00	190.00	40.00	199.00	190.00	40.00

Table 6.6. Results of GSoDSA (*p*=0.1) on the Sensor Tracking Problem over all parameter settings

Arrival interval	1	1	1	2	2	2	5	5	5
Life Span	10	50	200	10	50	200	10	50	200
Density	Stable Cycles								
0.4	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.01	0.01
0.6	0.00	0.00	0.00	0.01	0.01	0.01	22.41	0.26	0.23
0.8	20.98	22.96	28.56	2.63	12.11	21.58	904.81	17.87	25.25
1.0	961.53	962.31	961.79	807.63	955.44	954.64	985.84	945.85	946.23
Ideal	990.00	990.00	990.00	995.00	990.00	990.00	998.00	990.00	990.00
	Average Queue Time								
0.4	196.72	639.69	996.25	235.87	646.76	996.18	202.94	646.10	997.21
0.6	58.33	289.33	861.19	58.73	289.61	870.83	56.25	284.36	867.64
0.8	22.73	112.15	445.02	23.01	113.45	444.83	10.70	113.13	444.21
1.0	10.14	50.72	203.47	10.22	50.73	203.56	10.03	50.78	203.64
Ideal	10.00	50.00	200.00	10.00	50.00	200.00	10.00	50.00	200.00
	Requests Satisfied								
0.4	46.28	5.72	0.03	38.40	4.97	0.02	44.75	4.67	0.02
0.6	166.82	29.26	2.99	164.44	29.33	2.55	164.53	29.44	2.41
0.8	439.44	84.96	17.93	425.82	83.01	17.85	199.00	82.27	17.48
1.0	977.03	189.97	40.00	495.99	189.95	40.00	199.00	188.05	40.00
Ideal	990.00	190.00	40.00	499.00	190.00	40.00	199.00	190.00	40.00

Table 6.7. Results of ESoDSA (*p*=0.1) on the Sensor Tracking Problem over all parameter settings

show that the more dynamic the problem, the tougher it is for ESoDSA and the better it is for GSoDSA. They also show that SoDSA is still the better choice for easy problems.

A quick look at Table 6.9 which provides a direct comparison between the mdBA, DSA, and the various GEPs shows that ESoHC and SoHC are still the best choices for the problem. As the table shows, the mdBA and dBA based GEPs are many times more effective than the DSA and DSA based GEPs.

6.7. Conclusion

These results show that ESoHC performs the best when it comes to maintaining stable cycles, as it is able to find solutions faster than any of the other algorithms. However, SoHC may be a better choice when the communication density nears the critical density, as it is able to satisfy more requests and reduces queue times better than either ESoHC or GSoHC. The DSA and DSA based GEPs were not able to perform as well as either the SoHC or dBA based GEPs as they have difficulty maintaining a stable solution when multiple agents are changing their values simultaneously.

Arrival interval	1	1	1	2	2	2	5	5	5
Life Span	10	50	200	10	50	200	10	50	200
Density	Stable Cycles								
SoDSA	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01
GSoDSA	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01
ESoDSA	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.01	0.01
	Average Queue Time								
SoDSA	207.95	798.13	1000.00	215.66	821.16	1000.00	197.77	803.18	1000.00
GSoDSA	211.37	791.97	1000.00	195.70	827.33	1000.00	200.30	805.15	1000.00
ESoDSA	196.72	639.69	996.25	235.87	646.76	996.18	202.94	646.10	997.21
	Requests Satisfied								
SoDSA	44.97	5.84	0.00	44.60	4.49	0.00	49.00	4.91	0.00
GSoDSA	45.78	5.40	0.00	49.23	4.69	0.00	46.83	5.06	0.00
ESoDSA	46.28	5.72	0.03	38.40	4.97	0.02	44.75	4.67	0.02

Table 6.8. Comparison of SoDSA, GSoDSA, and ESoDSA at communication density 0.4

Arrival interval	1	1	1	2	2	2	5	5	5
Life Span	10	50	200	10	50	200	10	50	200
Density	Stable Cycles								
SoHC	0.00	0.00	0.00	0.06	0.00	0.00	280.85	1.23	1.23
GSoHC	0.00	0.00	0.00	2.76	0.00	0.00	451.29	0.77	0.75
ESoHC	0.00	0.00	0.00	0.00	0.00	0.00	476.36	0.65	0.58
SoDSA	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01
GSoDSA	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01
ESoDSA	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.01	0.01
	Average Queue Time								
SoHC	42.96	221.68	759.35	44.22	227.04	760.21	42.69	217.95	727.70
GSoHC	50.62	240.74	756.35	49.90	252.97	742.24	33.74	233.73	692.59
ESoHC	46.06	217.59	684.06	42.61	218.80	659.24	46.06	204.99	629.64
SoDSA	207.95	798.13	1000.00	215.66	821.16	1000.00	197.77	803.18	1000.00
GSoDSA	211.37	791.97	1000.00	195.70	827.33	1000.00	200.30	805.15	1000.00
ESoDSA	196.72	639.69	996.25	235.87	646.76	996.18	202.94	646.10	997.21
	Requests Satisfied								
SoHC	247.68	42.95	7.04	242.09	41.96	6.62	177.25	45.15	7.78
GSoHC	209.82	40.33	6.36	223.36	37.62	6.52	183.07	40.97	7.39
ESoHC	230.51	41.54	6.66	237.71	40.57	7.48	185.89	44.50	7.23
SoDSA	44.97	5.84	0.00	44.60	4.49	0.00	49.00	4.91	0.00
GSoDSA	45.78	5.40	0.00	49.23	4.69	0.00	46.83	5.06	0.00
ESoDSA	46.28	5.72	0.03	38.40	4.97	0.02	44.75	4.67	0.02

Table 6.9. The Comparison of the six algorithms/protocols at communication density 0.4

CHAPTER 7

THE SENSOR SHARING PROBLEM

7.1. Introduction

This chapter, discusses the implementation, testing, and results for the sensor sharing problem. As before, the dBA based GEPs will be compared to the DSA and DSA based GEPs. Along with the testing of the sensor sharing problem, some theoretical analysis will also be presented for the sensor sharing problem as well.

7.2. Problem Implementation

For the tracking problem, the domain space of each sensor pod is the corresponding target it is tracking. In the sensor sharing problem, however, instead of each pod possessing one sensing device, each pod now holds a total of *m* separate sensors. The domain space here for each pod is the sensor that is currently turned on (activated). In the tracking problem, a target must be tracked by exactly 3 sensor pods. Thus, the tracking of a target can be seen as a request for 3 sensor pods that satisfy the conditions that they can directly communicate with each other and are able to detect the target. In contrast, a request for sensors in the sensor sharing problem can vary based on the number of sensors that are needed. The sensor sharing problem, instead of being concerned with the allocation of pods, is concerned with the allocation of sensors to specific tasks.

Each of the pods in the sensor network used for the sharing problem has an internal allocation policy that limits how sensor resource can be allocated, and they are modeled as DisACSPs. Since the variable domains here now represent the specific sensor that is currently active, a value of *s* indicates that the sensor of type *s* is currently turned on in the pod. The value of 0 is reserved to represent the situation where all the sensors in the pod have been turned off. Pods with all sensors turned off cannot conflict with another pod's internal allocation policy.

7.3. The Requests

For testing purposes, it will be assumed that all requests are in unit form, meaning that each request will be for exactly one sensor. As defined in the problem statement, since only one sensor can be turned on for each pod, the network can service at most *n* unit requests at any given time, where *n* is the number of pods. In addition to arriving at specific intervals, each request has a specific life span, that represents the amount of time the request must be satisfied for it to be considered completed. Thus, if a request for sensor *s* requires 100 iterations worth of sensor time, this means that at least one pod must be assigned to have sensor *s* turned on for exactly 100 iterations. Note that though a request may require 100 iterations of sensor data, there is no constraint on the fact that the 100 iterations of data must be collected in consecutive iterations. Thus, it will be assumed that a request for 100 iterations of data will need 100 iterations worth of data, no matter where and when the data is collected.

As with the tracking problem, requests are automatically placed in a queue as they arrive. Since all pods send information towards a base station, it is assumed that the

complete request queue is stored on a base station and broadcast to the pods as needed. It is also assumed that a sensor starts collecting data as soon as it is turned on. This collected data is forwarded to the base station.

The pods in the sensor network only know about the first *n* requests queued at the base station, where *n* is the number of pods in the network. The network reassigns active sensors in its pods until it can satisfy both these requests and its internal allocation policies. As sensors start collecting and forwarding data as soon as they are turned on, it is very possible that the base station will be sent sensor data that exceeds the requirements of the first *n* queued request(s). In these cases, the sensor data is passed down to the next request in the queue that needs it, on a first-come-first-serve basis. It is therefore possible for a queued request to be satisfied before the sensor network actively tries to satisfy it, based solely on the extra sensor data that is generated.

7.4. Testing

The metrics used to test the GEPs on the sensor sharing problem are very similar to those used for the sensor tracking problem. A total of 30 pods were used and each pod carried 6 distinct sensors for a domain size of 6. The constraint density for the allocation policy was kept constant at 1, while the constraint tightness was varied from 0.01 to 0.06 as for the random DisACSPs, and the policies were randomly generated accordingly. The intervals by which the requests arrives was set to {1, 10, 50}, while the life span of the requests was {10, 50, 200}. As before, a population size of 32 was used for the dBA based GEPs, SoHC, SoDSA, and DSA based GEPs. The sensor network again ran for 1000 iterations.

7.5. Theoretical Discussion

As discussed in Chapter 2, not all CSPs and ACSPs are solvable, and at the phase transition any randomly generated CSP is only expected to have one solution. The question then arises as to whether such a phase transition exists for the sensor sharing problem that can be used to predict whether or not there is a stable solution that satisfies both internal allocation policies and external requests.

First, examine once again the equation used to calculate the average number of feasible solutions for a randomly generated DisACSP, first given in Chapter 2:

$$S = m^n (1 - p2)^{p1 \cdot n \cdot (n-1)} \tag{2.8.3}$$

The primary characteristic of this equation is that it is actually made up of two parts. The first part, m^n, represents the number of possible value assignments for the n variables, which are referred to as the number of candidate solutions. The second part of the equation is the probability that any given candidate solution will satisfy all random constraints. Hence, the number of candidate solutions multiplied by the probability that a given candidate solution satisfies all constraints will give the expected number of feasible solutions that will satisfy all constraints. Using this as a basis, a similar equation for the sensor network sharing problem can be constructed.

One of the defining differences between the randomly generated DisACSP and the sensor sharing problems is the solution needed. For the randomly generated DisACSP, the requirement is to simply satisfy the constraints, and the focus is therefore on the second part of the equation, and any solution that does not violate any constraints will be acceptable. For the sensor sharing problem, however, a solution must also satisfy any external requests. Thus, a very specific solution is needed, which will limit the number

of candidate solutions in the first part of the equation. So, given a problem with n variables and a domain size of m, the number of candidate solutions for randomly generated DisACSPs that have no constraints is simply m^n. Given the same number of variables and domain size, a sensor sharing problem, with no allocation restrictions, dealing with an external request for k sensors of type i will only have

$$C_{n,k} = \frac{n!}{k!(n-k)!} \tag{7.4.1}$$

candidate solutions to work with. This assumes that the request is for sensors of the same type, which turns the problem into a one target sensor tracking problem where the target needs to be k-trackable. If the request is for k_i sensors of type i and k_j sensors of type j, then the number of feasible solutions is:

$$C_{n,k_i} C_{n-k_i,k_j} = \frac{n!}{k_i!(n-k_i)!} \frac{(n-k_i)!}{k_j!(n-k_i-k_j)!} = \frac{n!}{k_i!\,k_j!(n-k_i-k_j)!} \tag{7.4.2}$$

From this example, it can be seen that how hard a problem setting is for the sensor sharing problem is highly dependent on the type of external request that needs to be satisfied. This is because the external requests are the primary limiting factor that reduces the number of candidate solutions. The worst case scenario is when a request asks for n sensors of the same type, where n is the number of sensor pods in the network. Lacking any allocation restrictions, there is only one candidate solution for such a request, which means that there will be a low probability of a feasible solution if there are any allocation policies or restrictions.

The examples above, 7.4.1 and 7.4.2, calculate the number of candidate solutions for the sensor sharing problem and are not complete without a consideration of internal allocation policies, which will give the probability that a given candidate solution is

feasible. Given that each pod carries *m* sensor types, there are a total of *n* pods and the number of sensors requested of each type is denoted by k_1 to k_m, the complete equation for finding the number of candidate solutions, *S*, is as follows:

$$S = C_{n,X} \prod_{i=1}^{m} C_{(X - \sum_{j=1}^{i-1} k_j), k_i} \; ; X = \sum_{i=1}^{m} k_i \tag{7.4.3}$$

This can be expanded to:

$$S = \frac{n!}{X!(n-X)!} \frac{\dot{X}!}{\prod_{i=1}^{m} k_i!} = \frac{n!}{\prod_{i=1}^{m} (k_i!) \left[(n - \sum_{i=1}^{m} k_i)! \right]} \tag{7.4.4}$$

In the case where $\sum_{i=1}^{m} k_i = n$, the equation will be reduced to $S = (n!) / \left[\prod_{i=1}^{m} k_i! \right]$. In the worst case scenario where *n* sensors of one type are requested, then *S = 1*.

Before considering the other half of the equation, which deals with the allocation policies, it should be noted once again that sensor pods that are not assigned a specific sensor to turn on will stay off and thus not be in any constraint conflicts with the other pods. This means that when calculating the probability that a candidate solution is also feasible, only the pods that have been assigned sensors (turned on) will be considered. Taking into consideration the existing internal constraints in the form of a DisACSP with a constraint density, p_1, and a constraint tightness, p_2, the equation used to approximate the number of feasible solutions for the sensor sharing problem becomes as follows:

$$S = \frac{n!}{\prod_{i=1}^{m} k_i! (n - \sum_{i=1}^{m} k_i)!} (1 - p_2)^{p_1 \cdot \sum_{i=1}^{m} k_i \cdot \left[\left(\sum_{i=1}^{m} k_i \right) - 1 \right]} \tag{7.4.5}$$

The second part of this equation is slightly different from the one used to estimate the number of feasible solutions for randomly generated DisACSPs, 2.8.3. Instead of using

$\left(1-p_2\right)^{p_1 \cdot n \cdot (n-1)}$, n is replaced with $\sum_{i=1}^{m} k_i$, which is the total number of sensors being requested, because the only concern is constraint satisfaction for those pods that need to be assigned a sensor. Then solving for the constraint tightness p_2 when $S = 1$, the single solution point, gives the following:

$$p_2 = 1 - \left[\frac{\prod_{i=1}^{m} k_i ! \left(n - \sum_{i=1}^{m} k_i \right) !}{n!} \right]^{\frac{1}{p_1 \left(\sum_{i=1}^{m} k_i \right) \left(\sum_{i=1}^{m} k_i - 1 \right)}} \tag{7.4.6}$$

Thus, 7.4.5 reveals several things about the sensor sharing problem. Firstly, the smaller the number of sensors requested, the higher the constraint density and tightness can be, while at the same time maintaining the presence of feasible solutions. As the number of sensors requested approaches *n*, the number of sensors of each type requested becomes more of a factor. Based on 7.4.5, the hardest problems are those where all the sensors requested are of one type, while if the requests are spread evenly among all the available sensor types, the problem becomes easier. It should be noted that since the requests for sensors are random, 7.4.5 does not really give us a general idea of where the phase transition for the problem is, as this is highly dependent on the number and distribution of sensor requests among the available sensor types. However, it can give us a best and worst case upper and lower bound of the constraint tightness value, where the problem is expected to have only one solution (Figure 7.1).

For testing, both uniform and non-uniform distributions of resquests among the sensor types will be tested. For the first part, a uniform random number generator will be used to generate the sensor requests, which should evenly spread the requests among all available sensor types. For the second part a Gaussian random number generator is used

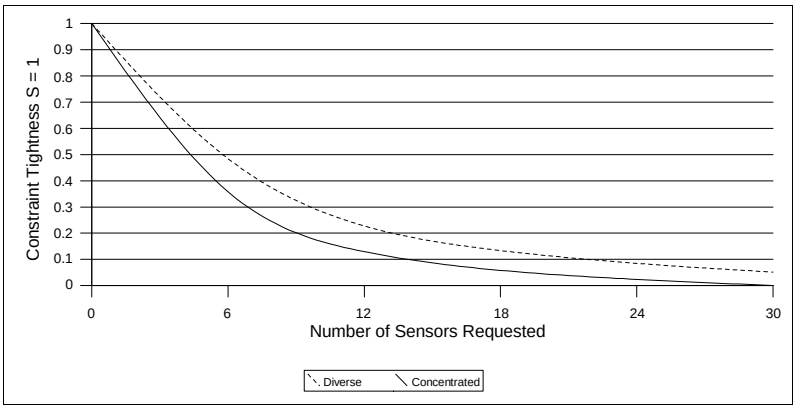

Figure 7.1. Upper and Lower bound for constraint tightness for the Sensor Sharing Problem

to generate senosr requests from a normal distribution with mean 3.5 and standard deviations of 1.0 and 0.5. All numbers generated from the Gaussian random number generator is rounded to the nearest integer in the set [1, 6]. Thus, the probability of requesting a specific sensor type is listed in Table 7.1.

As the results reveal, since the requests are queued and have a fixed life span, the performance is primarily affected by the number of active requests at any given time. So, for example, when the requests arrive at an interval of 1 iteration and have a life span of 10 iterations, then the expected maximum number of requests in the network at any given time will be 10, which is much lower than the 30 sensor pods, which will be used for the test. As the number of active requests at any given time nears the number of pods and becomes greater than the number of pods, the problem becomes very hard and it becomes

	Standard Deviation	
Values	1.0	0.5
1	0.022750	0.000032
2	0.135905	0.022718
3	0.341345	0.477250
4	0.341345	0.477250
5	0.135905	0.022718
6	0.022750	0.000032

Table 7.1. Probability of a requesting a sensor of a specific type given a Gaussian random number generator with mean 3.5

almost impossible to establish stable cycles, as discussed further in the next section. The results also show that the clustering of sensor requests further make problems more difficult.

7.6. Results (Uniform Distribution)

Table 7.2 gives the results for SoHC, GSoHC, ESoHC, SoDSA, GSoDSA, and ESoDSA for the sensor sharing problem where the requests arrive every iteration and have a life span of 10 iterations. The setting for the arrival interval and life span of the unit requests implies that there are effectively 10 requests for sensors in the network at any given time. Based on 7.4.6, with 10 unit requests for sensors, the tightness at which the problem is expected to have one solution is around 0.174 to 0.28 depending on how spread out the requests are among the sensor types. Consequently, this problem setting is not very hard. All the algorithms tested achieved similar average queue times and were able to satisfy the same amount of requests. The differences between the algorithms therefore lie solely in the number of stable cycles, where SoDSA and DSA based GEPs once again fall behind by a large margin. Due to the more dynamic nature of the problem (requiring more active reassigning of tasks) as opposed to the relatively static tracking problem used in the previous chapter, SoHC is able to outperform GSoHC and ESoHC when it comes to maintaining a stable solution. With only 10 out of 30 possible pods allocated at any given time, the problem never really gets very hard for SoHC, though the performance does decrease with increasing constraint tightness, as expected.

Table 7.3 shows the results for all the algorithms on the sensor sharing problem with external requests arriving every iteration and having a life span of 50. The longer life

span of the requests implies two things. Firstly, the maximum of 30 sensor requests will constantly be active, thus making the problem harder. Secondly, the longer life span of the requests implies that fewer reconfiguration steps will need to be taken, which is why the number of stable cycles are greater than those shown in Table 7.2. The DSA based GEPs are still incapable of maintaining stable cycles, though they do not fall far behind when considering the average queue time for requests and the number of requests satisfied. However, this is due to allowing the sensors to send out data even when they are in a constraint conflict.

With the maximum number of requests for sensors active at all times, the problems become hard very fast, as neither SoHC, GSoHC, or ESoHC can keep a stable cycle when the constraint tightness increases beyond 0.03. Based on 7.4.6, the single solution

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	115.80	100.90	87.74	75.44	63.60	51.84
GSoHC	111.95	97.37	82.74	69.10	55.33	43.13
ESoHC	111.24	95.97	82.13	59.88	44.23	31.58
SoDSA	9.44	5.42	3.24	1.85	1.13	0.63
GSoDSA	6.14	3.44	2.01	1.14	0.72	0.45
ESoDSA	6.30	3.49	2.07	1.22	0.73	0.48
	Requests Satisfied					
SoHC	995.00	995.00	995.00	994.95	994.95	994.89
GSoHC	990.00	989.99	989.97	989.96	989.96	989.94
ESoHC	990.00	990.00	990.00	989.99	989.97	989.92
SoDSA	990.28	990.25	990.17	990.13	990.07	990.04
GSoDSA	990.35	990.29	990.24	990.18	990.14	990.06
ESoDSA	990.33	990.28	990.24	990.22	990.14	990.08
	Average Queue Time					
SoHC	10.83	10.85	10.87	10.90	10.93	10.97
GSoHC	10.83	10.84	10.86	10.89	10.92	10.97
ESoHC	10.83	10.85	10.86	10.90	10.95	11.01
SoDSA	10.77	10.81	10.84	10.88	10.92	10.95
GSoDSA	10.75	10.79	10.82	10.86	10.90	10.94
ESoDSA	10.75	10.79	10.83	10.86	10.90	10.94

Table 7.2. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 1 iteration and life span of 10 iterations

point for this problem lies between 0 and 0.0514, depending on how spread out the requests are among the sensor types. Since the requests were generated with a uniform random number generator, the assumption is that the requests would be fairly evenly spread out among the sensor types, and the tightness where the problem really gets hard should be around 0.05, which is supported by the results. The biggest surprise here is that GSoHC actually performs slightly better than either SoHC or ESoHC. ESoHC's performance is expected, as it uses the highest level of exploitation, which keeps the population concentrated around the region where the current solution is. SoHC's performance is affected by the harder problem, while GSoHC, which falls between ESoHC and SoHC in terms of exploration and exploitation, performs slightly better as it is able to explore more possibilities.

Table 7.4 shows the results for the test with requests arriving every iteration and a life span of 200 iterations. The longer life span of the request automatically translates to a higher number of stable cycles, as seen earlier. The longer life span also helps the DSA based GEPs lock onto a stable solution for a while. One point to note is that SoDSA is actually able to maintain almost 4 times more stable cycles than ESoDSA or GSoDSA. This is mainly due to the higher level of exploitation performed by the DSA when the number of remaining constraint violations drop to a certain level, as seen in chapter 4. Here, there is also a significant drop in performance by ESoHC as the constraint tightness increases, while SoHC clearly performs the best in terms of requests satisfied, average queue time and stable cycles. Once again, the maximum 30 sensor requests that are active at all times makes problems with a constraint tightness of 0.4 and greater much harder.

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	474.12	324.09	36.02	0.64	0.14	0.05
GSoHC	514.97	331.90	50.53	0.80	0.17	0.15
ESoHC	469.11	206.07	18.82	0.32	0.17	0.19
SoDSA	3.91	0.37	0.27	0.25	0.22	0.19
GSoDSA	1.44	0.29	0.23	0.21	0.21	0.17
ESoDSA	1.34	0.27	0.22	0.23	0.17	0.17
	Requests Satisfied					
SoHC	584.52	579.49	562.27	537.35	513.60	490.80
GSoHC	570.91	568.81	554.78	539.79	531.54	522.61
ESoHC	569.98	557.65	536.52	510.35	492.57	485.01
SoDSA	560.88	550.95	541.21	529.50	517.86	505.48
GSoDSA	555.03	546.48	537.49	526.54	515.76	504.41
ESoDSA	555.02	546.47	537.59	526.53	515.75	504.51
	Average Queue Time					
SoHC	218.59	221.12	228.52	239.39	249.73	258.86
GSoHC	232.57	236.05	243.03	249.62	253.58	258.23
ESoHC	234.32	240.27	251.50	262.80	270.05	275.69
SoDSA	241.28	246.47	251.15	256.31	262.24	268.59
GSoDSA	244.21	248.29	252.91	257.84	263.38	269.04
ESoDSA	244.23	248.27	252.90	257.82	263.41	269.06

Table 7.3. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 1 iteration and life span of 50 iterations

Table 7.5 shows the results for the test with requests arriving every 10 iterations and a life span of 10 iterations. The long inter-arrival time and short life span means that only 1 sensor request for 1 sensor will be active at all times. This makes the problem relatively easy and the constraint tightness for the allocation policies has a minimal affect on performance. The performances for SoHC, GSoHC, and ESoHC are virtually identical. The DSA based GEPs still lag behind, but not by far as the problem is very easy. It should be noted that the highly convergent ESoDSA performs significantly better than SoDSA or GSoDSA as it is able to find a solution very quickly, thus maintaining more stable cycles.

Tables 7.6 to 7.10 show the results for testing requests that arrive every 10 and 50 iterations with life spans of 10, 50, and 200. In comparison to the above scenarios, these

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	873.17	841.55	516.62	11.74	0.05	0.15
GSoHC	872.34	821.16	409.12	11.52	0.18	0.17
ESoHC	861.62	581.76	139.19	1.88	0.17	0.15
SoDSA	120.34	1.13	0.28	0.24	0.19	0.20
GSoDSA	33.07	0.42	0.25	0.21	0.18	0.15
ESoDSA	36.03	0.53	0.23	0.20	0.20	0.13
	Requests Satisfied					
SoHC	135.47	135.10	135.00	130.85	123.90	117.70
GSoHC	120.35	120.26	120.10	119.22	117.84	116.14
ESoHC	120.18	119.89	115.15	109.14	106.83	105.34
SoDSA	125.96	125.12	122.39	120.61	118.67	114.88
GSoDSA	126.71	124.54	122.11	120.25	117.82	114.55
ESoDSA	126.74	124.53	122.10	120.29	117.81	114.50
	Average Queue Time					
SoHC	442.63	442.47	449.53	465.00	469.00	470.23
GSoHC	457.57	458.50	468.29	486.08	491.89	494.44
ESoHC	457.47	468.01	481.27	488.08	499.04	503.49
SoDSA	486.52	493.94	492.97	496.93	501.50	500.37
GSoDSA	494.07	495.53	495.20	498.72	501.43	501.44
ESoDSA	494.08	495.50	495.23	498.80	501.43	501.65

Table 7.4. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 1 iteration and life span of 200 iterations

problems are much easier and the results reflect this. As the number of active requests at any given time does not exceed 20, these problems do not pose a challenge for the SoHC, GSoHC and ESoHC, whose performances are identical in all but one problem. The hardest problem among these is shown in Table 7.7 where requests arrive every 10 iterations with a life span of 200, where the tightness for only one feasible solution is around 0.11. The performance of ESoHC quickly drops as the tightness increases. SoHC still performs better than either GSoHC or ESoHC in maintaining stable cycles. As described in the previous chapter, a dynamic environment favors methods with greater population diversity, which is why SoHC performs slightly better. The DSA based GEPs are able to keep up somewhat with these easier problems, but among them, ESoDSA performs the best and this can also be seen in the results given in Table 7.7. ESoDSA's performance advantage lies in its ability to converge towards a solution faster than either

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	902.12	901.32	900.27	898.62	897.55	895.99
GSoHC	900.85	899.47	898.22	897.06	895.92	894.59
ESoHC	901.14	900.28	899.45	898.74	898.01	897.30
SoDSA	713.60	711.80	710.29	709.22	706.96	705.60
GSoDSA	650.53	649.69	645.93	643.53	641.71	638.76
ESoDSA	755.36	754.49	754.22	753.25	752.49	751.87
	Requests Satisfied					
SoHC	99.00	99.00	99.00	99.00	99.00	99.00
GSoHC	99.00	99.00	99.00	99.00	99.00	99.00
ESoHC	99.00	99.00	99.00	99.00	99.00	99.00
SoDSA	99.00	99.00	99.00	99.00	99.00	99.00
GSoDSA	99.00	99.00	99.00	99.00	99.00	99.00
ESoDSA	99.00	99.00	99.00	99.00	99.00	99.00
	Average Queue Time					
SoHC	10.84	10.84	10.84	10.84	10.84	10.84
GSoHC	10.86	10.86	10.86	10.86	10.86	10.86
ESoHC	10.86	10.86	10.86	10.86	10.86	10.86
SoDSA	10.88	10.88	10.89	10.89	10.89	10.89
GSoDSA	10.89	10.89	10.89	10.89	10.90	10.90
ESoDSA	10.90	10.90	10.90	10.90	10.90	10.90

Table 7.5. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 10 iteration and life span of 10 iterations

SoDSA and GSoDSA, but because of the properties of DSA this convergence is actually better for ESoDSA in dynamic environments, especially when the problem gets progressively harder.

7.7. Results (Normal Distribution)

Tables 7.11 to 7.19 present the results of the sensor sharing problem when the requests for sensors are generated based on a normal distribution and a standard deviation of 1. The probability of requesting a specific sensor type is shown in Table 7.1. With a standard deviation of 1, the probability of requesting sensors of type 1 and 6 are fairly small, while the majority of requests should be for sensors of type 3 and 4.

As mentioned in section 7.5, the number of feasible solutions for the sensor sharing problem drops as the sensor requests become more clustered around specific sensor

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	896.49	891.22	886.74	882.32	878.20	873.79
GSoHC	896.63	891.75	886.20	880.47	874.38	867.53
ESoHC	896.15	891.22	886.40	882.12	877.97	874.20
SoDSA	737.38	722.57	704.64	683.69	660.05	634.92
GSoDSA	672.23	648.25	620.34	589.94	558.30	522.00
ESoDSA	760.12	756.39	749.98	742.49	733.02	720.58
	Requests Satisfied					
SoHC	95.00	95.00	95.00	95.00	95.00	95.00
GSoHC	95.00	95.00	95.00	95.00	95.00	95.00
ESoHC	95.00	95.00	95.00	95.00	95.00	95.00
SoDSA	95.00	95.00	95.00	95.00	95.00	95.00
GSoDSA	95.00	95.00	95.00	95.00	95.00	95.00
ESoDSA	95.00	95.00	95.00	95.00	95.00	95.00
	Average Queue Time					
SoHC	50.88	50.88	50.88	50.88	50.88	50.88
GSoHC	50.87	50.88	50.88	50.89	50.91	50.93
ESoHC	50.87	50.87	50.87	50.87	50.88	50.88
SoDSA	51.04	51.05	51.07	51.10	51.13	51.16
GSoDSA	51.09	51.12	51.15	51.19	51.23	51.28
ESoDSA	51.14	51.14	51.14	51.14	51.15	51.16

Table 7.6. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 10 iteration and life span of 50 iterations

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	889.64	877.25	863.29	852.44	841.14	744.30
GSoHC	890.44	872.89	846.93	793.60	676.16	457.79
ESoHC	890.70	876.58	862.80	767.06	493.87	242.96
SoDSA	574.83	245.34	127.62	93.75	80.49	72.67
GSoDSA	400.35	157.72	97.24	77.67	67.82	61.10
ESoDSA	773.44	717.60	623.82	464.86	299.22	181.12
	Requests Satisfied					
SoHC	80.00	80.00	80.00	80.00	80.00	80.00
GSoHC	80.00	80.00	80.00	80.00	80.00	79.94
ESoHC	80.00	80.00	80.00	79.99	79.73	79.05
SoDSA	80.00	80.00	80.00	80.00	80.00	79.99
GSoDSA	80.00	80.00	80.00	80.00	79.99	79.96
ESoDSA	80.00	80.00	80.00	80.00	80.00	80.00
	Average Queue Time					
SoHC	200.89	200.90	200.95	201.04	201.13	202.01
GSoHC	200.92	200.96	201.06	201.37	202.30	204.51
ESoHC	200.92	200.94	200.99	201.60	205.18	211.68
SoDSA	201.99	202.90	204.10	205.45	206.86	208.27
GSoDSA	202.70	204.02	205.36	206.72	208.12	209.52
ESoDSA	201.80	201.77	201.81	202.01	202.38	202.92

Table 7.7. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 10 iteration and life span of 200 iterations

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	981.00	981.00	981.00	981.00	981.00	981.00
GSoHC	981.00	981.00	981.00	981.00	981.00	981.00
ESoHC	981.00	981.00	981.00	981.00	981.00	981.00
SoDSA	936.24	936.19	936.40	936.17	936.20	936.42
GSoDSA	925.06	924.92	924.75	924.87	924.62	924.61
ESoDSA	944.49	944.76	944.60	944.66	944.69	944.80
	Requests Satisfied					
SoHC	20.00	20.00	20.00	20.00	20.00	20.00
GSoHC	20.00	20.00	20.00	20.00	20.00	20.00
ESoHC	20.00	20.00	20.00	20.00	20.00	20.00
SoDSA	20.00	20.00	20.00	20.00	20.00	20.00
GSoDSA	20.00	20.00	20.00	20.00	20.00	20.00
ESoDSA	20.00	20.00	20.00	20.00	20.00	20.00
	Average Queue Time					
SoHC	10.95	10.95	10.95	10.95	10.95	10.95
GSoHC	10.95	10.95	10.95	10.95	10.95	10.95
ESoHC	10.95	10.95	10.95	10.95	10.95	10.95
SoDSA	10.99	10.99	10.99	10.99	10.99	11.00
GSoDSA	11.00	11.00	11.00	11.00	11.00	11.00
ESoDSA	10.99	11.00	10.99	10.99	10.99	11.00

Table 7.8. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 50 iteration and life span of 10 iterations

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	980.30	980.10	979.89	979.62	979.55	979.15
GSoHC	980.81	980.69	980.53	980.34	980.23	979.99
ESoHC	981.00	981.00	981.00	981.00	981.00	981.00
SoDSA	948.47	948.73	948.72	948.51	948.14	948.05
GSoDSA	940.49	940.45	940.15	940.03	939.30	939.71
ESoDSA	956.37	955.55	955.66	954.88	954.04	953.81
	Requests Satisfied					
SoHC	19.00	19.00	19.00	19.00	19.00	19.00
GSoHC	19.00	19.00	19.00	19.00	19.00	19.00
ESoHC	19.00	19.00	19.00	19.00	19.00	19.00
SoDSA	19.00	19.00	19.00	19.00	19.00	19.00
GSoDSA	19.00	19.00	19.00	19.00	19.00	19.00
ESoDSA	19.00	19.00	19.00	19.00	19.00	19.00
	Average Queue Time					
SoHC	50.83	50.83	50.83	50.83	50.83	50.83
GSoHC	50.74	50.74	50.74	50.74	50.74	50.74
ESoHC	50.74	50.74	50.74	50.74	50.74	50.74
SoDSA	50.77	50.78	50.78	50.78	50.78	50.78
GSoDSA	50.78	50.78	50.78	50.78	50.78	50.78
ESoDSA	50.78	50.78	50.78	50.78	50.78	50.78

Table 7.9. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 50 iteration and life span of 50 iterations

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	980.00	979.20	978.67	977.65	977.25	976.80
GSoHC	981.35	980.58	979.90	979.21	978.51	977.85
ESoHC	981.36	980.72	980.17	979.63	978.99	978.46
SoDSA	954.14	953.34	952.77	951.90	950.91	950.22
GSoDSA	946.28	944.97	943.11	941.59	940.43	938.49
ESoDSA	956.34	954.30	950.92	946.93	945.34	938.57
	Requests Satisfied					
SoHC	17.00	17.00	17.00	17.00	17.00	17.00
GSoHC	17.00	17.00	17.00	17.00	17.00	17.00
ESoHC	17.00	17.00	17.00	17.00	17.00	17.00
SoDSA	16.88	16.90	16.91	16.90	16.91	16.90
GSoDSA	16.90	16.89	16.91	16.90	16.91	16.91
ESoDSA	16.90	16.88	16.90	16.91	16.91	16.92
	Average Queue Time					
SoHC	200.78	200.78	200.78	200.78	200.78	200.78
GSoHC	200.80	200.80	200.80	200.80	200.80	200.81
ESoHC	200.80	200.80	200.80	200.80	200.80	200.80
SoDSA	200.99	200.98	200.98	200.98	200.98	200.99
GSoDSA	201.00	201.00	201.00	201.01	201.01	201.02
ESoDSA	200.98	200.99	201.00	201.02	201.02	201.04

Table 7.10. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 50 iteration and life span of 200 iterations

types. This implies that as more sensor requests are made, it becomes harder to find a feasible solution as compared to when sensor requests were uniformly distributed among all available sensor types. This is seen in Table 7.11 as the number of stable cycles drop slightly as compared to the results in Table 7.2.

The primary point of interest of the results in Table 7.11 is that ESoHC actually performs better than GSoHC when it comes to the number of stable cycles, while SoHC performs slightly better than ESoHC. This result is due to the use of sensor requests that are normally distributed among the various sensor types. Based on the pobabilities in Table 7.1, a sensor request for a sensor of type 3 or 4 occurs at least once every 3 requests. This implies that even though it is harder to find a feasible solution with normally distributed sensor requests, once a feasible solution is found, there is a higher

probability that a reassignment of sensor tasks is not necessary when a request is complete and a new one arrives. With sensor requests arriving once every iteration, there is a very high probability that a feasible solution at any given iteration will be very similar to a feasible solution at a fairly recent previous iteration. Thus, ESoHC, with its high level of exploitation around the current best solution, performs better because it takes longer to move from one area of promise to another. In the previous section, this was ESoHC's weakness, while here, it becomes ESoHC's strength. SoHC, with its higher level of exploration, is still able to out perform ESoHC slightly, since its members will separately lock onto different areas of promise, which is the exactly the opposite of what ESoHC does for this problem. GSoHC falls behind because it does not have the level of exploitation to keep its search in one region, while also not having enough exploration to move quickly to other regions of promise.

The general performance difference between the dBA based GEPs and DSA based GEPs are still present. The DSA based GEPs just are not able to stay on a feasible solution long enough to have any stable cycles. The number of requests satisfied are fairly similar for the two different types of GEPs, while the DSA based GEPs have significantly longer queuing times for requests.

Table 7.12 presents the results for when sensor requests arrive every iteration and have a life span of 50 iterations. Here, the sensor network is pushed to its maximum load of trying to satisfy an average of 30 sensor requests at any given time. As mentioned earlier, the hardest problems are when the requests are concentrated around a specific sensor type. Based on the probability distribution in Table 7.1 and that the current problem deals with a maximum of 30 sensor requests at one time, the hardest set of

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	103.71	90.34	78.99	68.07	56.81	45.68
GSoHC	102.16	87.17	74.20	62.04	49.73	38.05
ESoHC	103.28	89.52	77.01	65.28	53.37	42.19
SoDSA	3.60	1.50	0.71	0.35	0.22	0.12
GSoDSA	2.55	1.00	0.48	0.26	0.15	0.09
ESoDSA	2.48	1.02	0.43	0.23	0.13	0.09
	Requests Satisfied					
SoHC	989.98	989.95	989.93	989.90	989.86	989.79
GSoHC	989.98	989.94	989.90	989.86	989.80	989.75
ESoHC	989.97	989.95	989.91	989.88	989.83	989.79
SoDSA	988.07	988.10	988.16	988.19	988.22	988.20
GSoDSA	988.14	988.17	988.24	988.25	988.23	988.20
ESoDSA	988.13	988.20	988.25	988.26	988.22	988.22
	Average Queue Time					
SoHC	10.83	10.85	10.87	10.89	10.92	10.96
GSoHC	10.83	10.86	10.88	10.91	10.95	10.99
ESoHC	10.83	10.85	10.87	10.89	10.93	10.97
SoDSA	12.96	12.89	12.85	12.82	12.81	12.83
GSoDSA	12.86	12.79	12.75	12.74	12.76	12.80
ESoDSA	12.86	12.79	12.75	12.74	12.76	12.80

Table 7.11. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 1 iteration and life span of 10 iterations and normal distribution with standard deviation of 1

requests that is likely to appear is one for 4 sensors of type 2 and 5, 11 sensors of type 3 and 4, and none of type 1 and 6. Plugging this into 7.4.6 and solving for the single solution point will give 0.0375 for constraint tightness. Then looking at the most diverse set of feasible requests, which is a set of requests for 1 sensor of type 1 and 6, 4 sensors of type 2 and 5, and 10 sensors of type 3 and 4, and plugging it into 7.4.6 to solve for the single solution point gives a constraint tightness of 0.0428. So, given the probability distribution of sensor requests among sensor types and combining it with the equation 7.4.6, it can be estimated that the constraint tightness for which there is only one expected feasible solution for the sharing problem is between 0.0375 and 0.0428. This is supported in the results.

Here, the performance advantage of ESoHC becomes more clear as the problems get harder. At full load, the performance of SoHC drops quickly as the constraint tightness increases and closes in on the single solution point. And with its higher level of exploitation than SoHC, GSoHC is able to come second in performance. The DSA based GEPs are not even able to hold 1 stable cycle. The number of requests satisfied and the average queue time further favor the dBA based GEPs. In comparison with the results in Table 7.3, dBA based GEPs are able to keep more stable cycles simply because of the slightly lower probability of needing to reassign sensor duty when an old request completes and a new one arrives.

Table 7.13 shows that as the life span of the request increases, GSoHC gains a significant advantage over ESoHC when nearing the single solution point. The results show ESoHC having a slight advantage until the constraint tightness increases to 0.03 where GSoHC performs slightly better. At a tightness of 0.04, GSoHC widens the margin. SoHC, on the other hand, quickly drops behind as the tightness nears the single solution point. The results clearly show that, even though ESoHC performs better when the problems are easy and request life spans are shorter, when the problem does get harder and the life span of the request increases, GSoHC has a slight edge.

Tables 7.14 to 7.19 show the remaining results. The longer interval between request arrivals keep the problems simple and the results in line with those previously obtained from using uniformly distributed requests. With the easier problems, the dBA based GEPs perform nearly identically. On the DSA based GEPs side, SoDSA does slightly better when the arrival interval is at 10 iterations with its higher level of exploitation near the solution. GSoDSA and ESoDSA do slightly better when the arrival interval is at 50

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	497.27	348.84	42.50	0.52	0.19	0.19
GSoHC	502.49	349.24	76.33	1.58	0.18	0.17
ESoHC	505.05	373.20	94.39	1.39	0.16	0.18
SoDSA	0.26	0.15	0.12	0.14	0.12	0.11
GSoDSA	0.17	0.14	0.09	0.08	0.10	0.12
ESoDSA	0.18	0.14	0.12	0.10	0.09	0.10
	Request Satisfied					
SoHC	570.00	568.21	551.60	524.81	500.71	478.68
GSoHC	570.00	568.39	555.85	542.44	534.26	527.21
ESoHC	570.00	569.08	557.73	546.18	540.75	535.70
SoDSA	548.05	533.21	518.45	502.16	486.06	469.46
GSoDSA	548.84	533.78	518.67	502.08	485.63	469.13
ESoDSA	548.80	533.81	518.64	502.10	485.70	469.19
	Average Queue Time					
SoHC	233.96	237.81	245.31	256.66	266.83	277.35
GSoHC	233.75	237.43	242.87	249.30	253.01	256.62
ESoHC	233.67	236.96	242.23	247.84	250.38	252.81
SoDSA	258.10	262.79	268.46	274.96	281.95	290.11
GSoDSA	258.05	262.95	268.73	275.38	282.26	290.26
ESoDSA	258.03	262.90	268.71	275.35	282.28	290.17

Table 7.12. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 1 iteration and life span of 50 iterations and normal distribution with standard deviation of 1

iterations where the problems are less dynamic because of the larger interval between the arrival of sensor requests. However, SoDSA is slowly able to regain the performance lead as the problem tightness increases when the life span increases to 200 iterations.

Next, Tables 7.20 to 7.28 present the results for the sensor sharing problem when sensor requests are normally distributed around 3.5 with a standard deviation of 0.5. The lower standard deviation implies that the requests for sensors will be further clustered around types 3 and 4, as seen in Table 7.1. Based on the table, the requests for sensor types 1 and 6 will almost never show up. Using the same method as before, based on 7.4.6, the constraint tightness for the single solution point is most likely within the range 0.0214 to 0.0275. This is supported by the results in 7.21 and 7.22.

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	873.22	836.29	427.32	5.47	0.22	0.17
GSoHC	874.17	834.70	614.88	54.55	0.88	0.17
ESoHC	874.25	839.10	613.49	46.24	0.78	0.16
SoDSA	0.68	0.18	0.14	0.13	0.11	0.09
GSoDSA	0.27	0.12	0.11	0.09	0.08	0.07
ESoDSA	0.82	0.13	0.12	0.09	0.09	0.09
	Request Satisfied					
SoHC	120.24	120.10	119.91	117.40	111.97	107.00
GSoHC	120.23	120.10	120.01	119.30	118.83	118.17
ESoHC	120.22	120.12	120.01	119.48	119.25	119.08
SoDSA	120.45	118.65	115.91	112.70	109.20	105.30
GSoDSA	120.31	118.46	115.70	112.42	109.20	105.22
ESoDSA	120.33	118.49	115.79	112.42	109.22	105.20
	Average Queue Time					
SoHC	457.44	458.02	467.43	489.04	495.11	501.33
GSoHC	457.39	458.01	462.74	477.33	486.61	490.97
ESoHC	457.33	457.97	462.56	476.31	481.89	485.10
SoDSA	524.46	520.77	518.42	517.19	516.43	516.70
GSoDSA	523.72	520.60	518.69	517.00	516.77	516.39
ESoDSA	523.78	520.77	518.81	516.96	516.89	516.34

Table 7.13. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 1 iteration and life span of 200 iterations and normal distribution with standard deviation of 1

The further clustering of sensor requests around 2 specific sensor types increases the probability that no sensor reassignment is needed when a new request is made. As seen in Table 7.11, an increase in clustering of sensor requests of a specific type increases the number of stable cycles as there are times when no sensor reassignment is needed to satisfy the new request. This is also seen in Table 7.20 to a larger degree as the standard deviation is dropped by half and increasing the clustering. One point of interest is that the DSA based GEPs also benefit greatly from this tighter clustering as the number of stable cycles for the DSA based GEPs jump significantly when comparing Table 7.11 to Table 7.20. Similar to Table 7.11, SoHC still performs slightly better, followed by

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	902.91	901.72	900.36	899.60	898.20	897.10
GSoHC	902.85	901.67	900.51	899.60	898.64	897.39
ESoHC	902.90	901.62	900.46	899.59	898.45	897.17
SoDSA	701.30	699.83	698.44	697.00	695.43	694.74
GSoDSA	684.58	682.48	681.08	680.87	679.67	680.73
ESoDSA	684.55	683.01	681.98	680.40	679.81	680.82
	Request Satisfied					
SoHC	99.00	99.00	99.00	99.00	99.00	99.00
GSoHC	99.00	99.00	99.00	99.00	99.00	99.00
ESoHC	99.00	99.00	99.00	99.00	99.00	99.00
SoDSA	99.00	99.00	99.00	99.00	99.00	99.00
GSoDSA	99.00	99.00	99.00	99.00	99.00	99.00
ESoDSA	99.00	99.00	99.00	99.00	99.00	99.00
	Average Queue Time					
SoHC	10.84	10.84	10.84	10.84	10.84	10.84
GSoHC	10.84	10.84	10.84	10.84	10.84	10.84
ESoHC	10.84	10.84	10.84	10.84	10.84	10.84
SoDSA	10.97	10.98	10.98	10.98	10.99	10.99
GSoDSA	10.96	10.97	10.97	10.98	10.99	10.99
ESoDSA	10.96	10.96	10.97	10.98	10.99	10.99

Table 7.14. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 10 iteration and life span of 10 iterations and normal distribution with standard deviation of 1

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	898.51	895.17	892.09	889.07	885.82	883.05
GSoHC	898.94	895.47	892.47	889.05	885.64	882.60
ESoHC	898.92	895.40	892.04	888.60	885.59	882.64
SoDSA	726.99	710.06	689.62	661.90	629.21	586.55
GSoDSA	688.87	662.90	627.62	581.90	528.38	464.15
ESoDSA	688.44	662.44	627.12	582.49	526.92	464.28
	Request Satisfied					
SoHC	95.00	95.00	95.00	95.00	95.00	95.00
GSoHC	95.00	95.00	95.00	95.00	95.00	95.00
ESoHC	95.00	95.00	95.00	95.00	95.00	95.00
SoDSA	95.00	95.00	95.00	95.00	95.00	95.00
GSoDSA	95.00	95.00	95.00	95.00	95.00	94.99
ESoDSA	95.00	95.00	95.00	95.00	94.99	95.00
	Average Queue Time					
SoHC	50.76	50.76	50.76	50.76	50.76	50.76
GSoHC	50.76	50.77	50.77	50.78	50.78	50.79
ESoHC	50.76	50.77	50.77	50.77	50.78	50.78
SoDSA	52.59	52.72	52.87	53.08	53.34	53.68
GSoDSA	53.16	53.35	53.62	53.99	54.47	55.06
ESoDSA	53.18	53.36	53.63	54.00	54.46	55.06

Table 7.15. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 10 iteration and life span of 50 iterations and normal distribution with standard deviation of 1

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	891.97	879.95	868.00	858.45	846.31	720.00
GSoHC	888.85	875.58	864.80	854.62	835.14	735.99
ESoHC	889.57	876.46	865.77	855.00	831.88	727.09
SoDSA	89.73	69.77	58.91	51.94	46.61	42.76
GSoDSA	74.61	60.19	51.20	45.51	41.90	38.74
ESoDSA	74.67	59.64	51.73	45.66	41.86	38.09
	Request Satisfied					
SoHC	80.00	80.00	80.00	80.00	80.00	79.99
GSoHC	80.00	80.00	80.00	80.00	80.00	79.99
ESoHC	80.00	80.00	80.00	80.00	80.00	79.98
SoDSA	74.98	74.98	74.99	74.98	74.95	74.83
GSoDSA	74.84	74.94	74.95	74.91	74.88	74.69
ESoDSA	74.84	74.93	74.94	74.91	74.87	74.71
	Average Queue Time					
SoHC	200.89	200.90	200.94	201.00	201.10	202.31
GSoHC	200.92	200.95	200.98	201.04	201.21	202.16
ESoHC	200.91	200.93	200.97	201.03	201.22	202.20
SoDSA	247.73	248.01	248.46	248.65	249.30	250.49
GSoDSA	250.19	249.44	249.61	249.92	250.60	251.97
ESoDSA	250.17	249.45	249.58	249.89	250.59	251.94

Table 7.16. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 10 iteration and life span of 200 iterations and normal distribution with standard deviation of 1

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	986.00	986.00	986.00	986.00	986.00	986.00
GSoHC	986.00	986.00	986.00	986.00	986.00	986.00
ESoHC	986.00	986.00	986.00	986.00	986.00	986.00
SoDSA	946.13	946.42	946.52	946.48	946.56	946.50
GSoDSA	951.21	951.26	951.27	951.28	951.45	951.45
ESoDSA	951.28	951.44	951.25	951.26	951.28	951.51
	Request Satisfied					
SoHC	20.00	20.00	20.00	20.00	20.00	20.00
GSoHC	20.00	20.00	20.00	20.00	20.00	20.00
ESoHC	20.00	20.00	20.00	20.00	20.00	20.00
SoDSA	20.00	20.00	20.00	20.00	20.00	20.00
GSoDSA	20.00	20.00	20.00	20.00	20.00	20.00
ESoDSA	20.00	20.00	20.00	20.00	20.00	20.00
	Average Queue Time					
SoHC	10.70	10.70	10.70	10.70	10.70	10.70
GSoHC	10.70	10.70	10.70	10.70	10.70	10.70
ESoHC	10.70	10.70	10.70	10.70	10.70	10.70
SoDSA	10.79	10.79	10.79	10.79	10.80	10.79
GSoDSA	10.78	10.78	10.78	10.78	10.78	10.78
ESoDSA	10.78	10.78	10.78	10.78	10.79	10.78

Table 7.17. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 50 iteration and life span of 10 iterations and normal distribution with standard deviation of 1

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	981.83	981.62	981.42	981.28	981.11	980.98
GSoHC	981.84	981.63	981.44	981.31	981.15	980.95
ESoHC	981.84	981.61	981.46	981.29	981.14	980.92
SoDSA	940.29	940.30	940.11	939.78	939.54	939.20
GSoDSA	945.38	945.03	944.85	944.62	944.23	944.11
ESoDSA	945.35	945.19	944.79	944.52	944.34	944.20
	Request Satisfied					
SoHC	20.00	20.00	20.00	20.00	20.00	20.00
GSoHC	20.00	20.00	20.00	20.00	20.00	20.00
ESoHC	20.00	20.00	20.00	20.00	20.00	20.00
SoDSA	20.00	20.00	20.00	20.00	20.00	20.00
GSoDSA	20.00	20.00	20.00	20.00	20.00	20.00
ESoDSA	20.00	20.00	20.00	20.00	20.00	20.00
	Average Queue Time					
SoHC	50.75	50.75	50.75	50.75	50.75	50.75
GSoHC	50.75	50.75	50.75	50.75	50.75	50.75
ESoHC	50.75	50.75	50.75	50.75	50.75	50.75
SoDSA	50.89	50.89	50.88	50.90	50.89	50.90
GSoDSA	50.86	50.87	50.87	50.88	50.89	50.89
ESoDSA	50.86	50.87	50.87	50.87	50.89	50.89

Table 7.18. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 50 iteration and life span of 50 iterations and normal distribution with standard deviation of 1

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	979.32	978.59	977.85	977.24	976.56	976.03
GSoHC	979.30	978.61	977.85	977.20	976.59	975.97
ESoHC	979.33	978.63	977.82	977.18	976.53	975.93
SoDSA	926.09	924.69	922.89	921.34	919.09	916.92
GSoDSA	927.06	924.03	920.29	914.79	909.08	900.64
ESoDSA	927.15	923.69	920.32	914.99	908.20	901.31
	Request Satisfied					
SoHC	16.00	16.00	16.00	16.00	16.00	16.00
GSoHC	16.00	16.00	16.00	16.00	16.00	16.00
ESoHC	16.00	16.00	16.00	16.00	16.00	16.00
SoDSA	16.00	16.00	16.00	16.00	16.00	16.00
GSoDSA	16.00	16.00	16.00	16.00	16.00	16.00
ESoDSA	16.00	16.00	16.00	16.00	16.00	16.00
	Average Queue Time					
SoHC	200.88	200.88	200.88	200.88	200.88	200.88
GSoHC	200.88	200.88	200.88	200.88	200.88	200.88
ESoHC	200.88	200.88	200.88	200.88	200.89	200.89
SoDSA	202.53	202.61	202.71	202.82	202.95	203.04
GSoDSA	202.71	202.84	202.97	203.15	203.36	203.60
ESoDSA	202.71	202.86	202.98	203.18	203.37	203.58

Table 7.19. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 50 iteration and life span of 200 iterations and normal distribution with standard deviation of 1

ESoHC and GSoHC. For the DSA based GEPs, SoDSA is able to maintain slightly more stable cycles than GSoDSA and ESoDSA.

With requests arriving every iteration and having a life span of 50 iterations, Table 7.21 shows the results where the network is at full load constantly. Like before, the increase in clustering of the requests makes it possible to maintain more stable cycles, but at the same time, a sudden drop in stable cycles is seen between problems with constraint tightness of 0.02 and 0.03. As predicted by equation 7.4.5, this is the region where the problem goes from having multiple feasible solutions to having less than 1 feasible solution. With the relatively small value of the single solution point, the performance of the DSA based GEPs are abysmal as even problems with a constraint tightness of 0.01 is slightly harder than before due to how close it is now from the single solution point. Table 7.21 also shows that apart from being able to maintain more stable cycles, ESoHC is also able to keep queue time down and the number of requests satisfied up even beyond the single solution point. Overall, ESoHC and GSoHC perform much better than SoHC with ESoHC being the best.

The results for a request arrival time of 1 iteration and life span of 200 iterations shown in Table 7.22 follow the same performance trend as the results in Table 7.13. Both ESoHC and GSoHC perform better than SoHC and are neck to neck when comparing stable cycles and average queue time. However, ESoHC performs slightly better in the number of requests satisfied. The extended life span and clustering of sensor types requested help out the DSA based GEPs slightly as they are able to maintain quite a few more stable cycles.

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	160.99	146.54	133.79	123.55	113.36	103.53
GSoHC	154.93	140.49	127.55	115.76	104.37	93.24
ESoHC	157.24	144.12	131.39	120.66	109.24	98.64
SoDSA	14.17	6.52	2.91	1.26	0.67	0.34
GSoDSA	10.10	4.60	1.98	0.81	0.36	0.19
ESoDSA	10.25	4.53	1.91	0.89	0.39	0.20
	Requests Satisfied					
SoHC	990.00	990.00	990.00	990.00	990.00	990.00
GSoHC	990.00	990.00	990.00	990.00	990.00	989.97
ESoHC	990.00	990.00	990.00	990.00	989.99	989.96
SoDSA	988.63	988.67	988.66	988.70	988.70	988.59
GSoDSA	988.64	988.67	988.68	988.63	988.51	988.51
ESoDSA	988.65	988.71	988.67	988.61	988.56	988.46
	Average Queue Time					
SoHC	10.69	10.71	10.73	10.75	10.77	10.79
GSoHC	10.71	10.73	10.76	10.78	10.81	10.84
ESoHC	10.70	10.72	10.74	10.76	10.79	10.82
SoDSA	12.43	12.41	12.40	12.40	12.41	12.48
GSoDSA	12.41	12.38	12.36	12.38	12.43	12.51
ESoDSA	12.41	12.37	12.36	12.39	12.43	12.51

Table 7.20. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 1 iteration and life span of 10 iterations and normal distribution with standard deviation of 0.5

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	532.79	183.87	2.69	0.28	0.23	0.23
GSoHC	551.56	285.26	14.57	0.22	0.26	0.28
ESoHC	554.34	293.90	14.35	0.20	0.26	0.25
SoDSA	0.49	0.14	0.13	0.14	0.09	0.09
GSoDSA	0.61	0.11	0.09	0.10	0.08	0.04
ESoDSA	0.61	0.13	0.08	0.09	0.06	0.04
	Requests Satisfied					
SoHC	570.50	556.99	519.59	472.14	439.05	409.22
GSoHC	570.86	562.77	538.81	515.34	501.43	489.58
ESoHC	570.89	563.17	541.95	526.53	518.84	512.20
SoDSA	531.67	516.59	498.19	475.00	453.67	430.33
GSoDSA	524.58	512.82	497.11	476.51	457.40	457.40
ESoDSA	524.54	512.80	497.03	476.45	457.30	435.90
	Average Queue Time					
SoHC	234.46	242.66	261.45	283.66	297.47	309.59
GSoHC	233.98	239.55	251.89	261.62	267.58	272.80
ESoHC	233.86	239.28	250.17	256.33	259.96	263.53
SoDSA	263.70	268.53	276.07	286.42	296.22	306.92
GSoDSA	268.23	271.01	277.19	285.67	294.64	304.82
ESoDSA	268.19	271.06	277.16	285.73	294.75	304.80

Table 7.21. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 1 iteration and life span of 50 iterations and normal distribution with standard deviation of 0.5

Tables 7.23 and 7.24 present the results for when requests arrive once every 10 iterations and have life spans of 10 and 50 iterations. The performance of the dBA based GEPs are as expected and not too surprising. For the DSA based GEPs, SoDSA is able to perform better than GSoDSA and ESoDSA because of its high level of exploitation near the solution. These results follow a performance trend similar to those in Table 7.14, but slightly different from those in Table 7.5. The primary difference lies in that SoDSA actually performs better than ESoDSA here where in Table 7.4, it was the opposite. This is attributed to the change from generating requests from a uniform distribution to that of a normal distribution which increased the amount of clustering in sensor types requested. So, two elements changed for the problem. Firstly, the problems become harder with clustering as predicted with equation 7.4.6. Secondly, the increase in clustering affects

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	876.56	497.53	15.62	0.26	0.29	0.33
GSoHC	880.39	597.02	69.41	0.44	0.26	0.24
ESoHC	880.48	595.60	71.16	0.35	0.27	0.24
SoDSA	4.90	0.16	0.13	0.13	0.09	0.13
GSoDSA	3.65	0.12	0.09	0.08	0.06	0.07
ESoDSA	3.00	0.11	0.07	0.08	0.06	0.07
	Requests Satisfied					
SoHC	121.00	119.20	111.88	103.85	98.44	93.23
GSoHC	121.00	119.97	116.51	112.58	110.21	108.62
ESoHC	121.00	119.97	117.44	115.41	114.16	113.19
SoDSA	119.88	117.36	112.19	106.65	101.46	96.95
GSoDSA	119.39	117.13	112.66	106.85	102.22	102.22
ESoDSA	119.41	117.12	112.71	106.81	102.26	97.57
	Average Queue Time					
SoHC	460.13	463.71	473.52	484.12	492.26	499.07
GSoHC	460.08	462.51	472.81	479.73	482.50	485.93
ESoHC	460.07	462.52	473.18	479.30	481.16	483.28
SoDSA	523.42	518.53	512.66	509.90	507.42	509.47
GSoDSA	528.71	521.77	516.19	509.92	507.69	507.79
ESoDSA	528.70	521.84	516.29	509.74	507.83	507.87

Table 7.22. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 1 iteration and life span of 200 iterations and normal distribution with standard deviation of 0.5

the probability that a new sensor request will cause a reassignment of sensor tasks. These two factors together contribute to the gain in performance for SoDSA that makes it perform slightly better here than ESoDSA.

Table 7.25 contains the results for the third hardest problem among the ones tested here. With an request arrival time of 10 iterations and a life span od 200 iterations, there are on average 20 sensor requests to satisfy at any given time. Based on the previous method for predicting where the hard problems are, the constraint tightness here for problems that are only expected to have one solution is around 0.07432 to 0.08547. This accounts for the drop in performance seen starting from a constraint tightness of 0.05. SoDSA still is able to perform better than the other DSA based GEPs.

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	917.41	916.71	916.15	915.36	914.74	914.19
GSoHC	917.42	916.75	916.22	915.48	914.95	914.36
ESoHC	917.43	916.74	916.17	915.48	914.95	914.30
SoDSA	773.15	771.91	771.45	770.54	769.87	768.96
GSoDSA	743.60	742.57	741.52	741.03	740.60	740.76
ESoDSA	744.21	742.57	741.09	740.54	740.41	740.91
	Requests Satisfied					
SoHC	99.00	99.00	99.00	99.00	99.00	99.00
GSoHC	99.00	99.00	99.00	99.00	99.00	99.00
ESoHC	99.00	99.00	99.00	99.00	99.00	99.00
SoDSA	99.00	99.00	99.00	99.00	99.00	99.00
GSoDSA	99.00	99.00	99.00	99.00	99.00	99.00
ESoDSA	99.00	99.00	99.00	99.00	99.00	99.00
	Average Queue Time					
SoHC	10.66	10.66	10.66	10.66	10.66	10.66
GSoHC	10.66	10.66	10.66	10.66	10.66	10.66
ESoHC	10.66	10.66	10.66	10.66	10.66	10.66
SoDSA	10.75	10.75	10.75	10.75	10.75	10.76
GSoDSA	10.74	10.75	10.75	10.75	10.75	10.76
ESoDSA	10.74	10.74	10.75	10.75	10.75	10.76

Table 7.23. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 10 iteration and life span of 10 iterations and normal distribution with standard deviation of 0.5

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	906.30	903.70	900.88	898.21	895.90	893.40
GSoHC	906.49	903.86	901.07	898.42	895.73	892.90
ESoHC	906.43	903.78	900.94	898.02	895.44	892.80
SoDSA	789.97	778.78	762.70	742.51	718.05	685.97
GSoDSA	742.97	722.75	693.50	659.18	614.77	563.72
ESoDSA	742.92	723.23	692.66	657.46	614.80	562.78
	Requests Satisfied					
SoHC	95.00	95.00	95.00	95.00	95.00	95.00
GSoHC	95.00	95.00	95.00	95.00	95.00	95.00
ESoHC	95.00	95.00	95.00	95.00	95.00	95.00
SoDSA	95.00	95.00	95.00	95.00	95.00	95.00
GSoDSA	95.00	95.00	95.00	95.00	95.00	95.00
ESoDSA	95.00	95.00	95.00	95.00	95.00	95.00
	Average Queue Time					
SoHC	50.67	50.67	50.67	50.67	50.67	50.67
GSoHC	50.67	50.67	50.68	50.68	50.69	50.70
ESoHC	50.67	50.68	50.68	50.68	50.69	50.69
SoDSA	51.76	51.82	51.90	52.02	52.15	52.35
GSoDSA	52.24	52.32	52.47	52.65	52.93	53.26
ESoDSA	52.23	52.32	52.48	52.68	52.93	53.28

Table 7.24. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 10 iteration and life span of 50 iterations and normal distribution with standard deviation of 0.5

Tables 7.26 to 7.28 contain the last batch of results. The longer time between arrival of requests help ESoDSA and GSoDSA perform slightly better than SoDSA for request life spans of 10 and 50 iterations, which are relatively short compared to the time between requests. However, as the problem gets harder with the longer life span of 200 iterations, SoDSA is able to regain its performance lead as the problem's constraint tightness increased.

7.8. Conclusions

The results show that as expected, the difficulty of a problem is determined by the ratio between the arrival interval of requests and the life span of the constraints, which defines how many active requests will be present at any given time, and the distribution

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	897.47	887.53	876.53	838.16	705.16	415.43
GSoHC	895.04	883.41	871.56	845.54	734.10	507.79
ESoHC	895.30	884.35	872.29	843.16	724.75	482.91
SoDSA	117.48	78.18	65.00	57.33	51.56	47.60
GSoDSA	89.91	64.02	54.40	47.96	42.90	39.98
ESoDSA	89.65	64.76	54.30	47.38	43.33	39.83
	Requests Satisfied					
SoHC	80.00	80.00	80.00	79.88	79.47	78.50
GSoHC	80.00	80.00	80.00	79.97	79.71	79.31
ESoHC	80.00	80.00	80.00	79.95	79.69	79.31
SoDSA	74.97	74.99	75.02	74.94	74.67	74.02
GSoDSA	74.35	74.48	74.49	74.42	74.06	74.06
ESoDSA	74.33	74.46	74.51	74.43	74.05	73.30
	Average Queue Time					
SoHC	200.84	200.84	200.89	201.10	202.14	206.29
GSoHC	200.86	200.89	200.94	201.11	201.92	204.06
ESoHC	200.86	200.88	200.92	201.10	201.96	204.03
SoDSA	240.63	242.17	242.28	242.72	244.09	247.24
GSoDSA	244.90	245.39	245.30	245.66	246.95	250.25
ESoDSA	244.92	245.44	245.25	245.69	246.94	250.23

Table 7.25. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 10 iteration and life span of 200 iterations and normal distribution with standard deviation of 0.5

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	988.00	988.00	988.00	988.00	988.00	988.00
GSoHC	988.00	988.00	988.00	988.00	988.00	988.00
ESoHC	988.00	988.00	988.00	988.00	988.00	988.00
SoDSA	953.94	954.05	953.86	954.01	954.04	953.99
GSoDSA	958.23	958.10	958.24	958.28	958.40	958.47
ESoDSA	958.19	958.17	958.20	958.20	958.31	958.26
	Requests Satisfied					
SoHC	20.00	20.00	20.00	20.00	20.00	20.00
GSoHC	20.00	20.00	20.00	20.00	20.00	20.00
ESoHC	20.00	20.00	20.00	20.00	20.00	20.00
SoDSA	20.00	20.00	20.00	20.00	20.00	20.00
GSoDSA	20.00	20.00	20.00	20.00	20.00	20.00
ESoDSA	20.00	20.00	20.00	20.00	20.00	20.00
	Average Queue Time					
SoHC	10.60	10.60	10.60	10.60	10.60	10.60
GSoHC	10.60	10.60	10.60	10.60	10.60	10.60
ESoHC	10.60	10.60	10.60	10.60	10.60	10.60
SoDSA	10.68	10.68	10.68	10.68	10.68	10.68
GSoDSA	10.67	10.67	10.67	10.67	10.67	10.67
ESoDSA	10.67	10.67	10.67	10.67	10.67	10.67

Table 7.26. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 50 iteration and life span of 10 iterations and normal distribution with standard deviation of 0.5

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	984.90	984.80	984.70	984.59	984.50	984.40
GSoHC	984.90	984.80	984.70	984.61	984.52	984.41
ESoHC	984.90	984.80	984.69	984.60	984.50	984.42
SoDSA	949.58	949.45	949.36	949.26	949.04	948.92
GSoDSA	953.46	953.47	953.23	953.11	952.93	952.83
ESoDSA	953.52	953.36	953.24	953.13	952.90	952.79
	Requests Satisfied					
SoHC	20.00	20.00	20.00	20.00	20.00	20.00
GSoHC	20.00	20.00	20.00	20.00	20.00	20.00
ESoHC	20.00	20.00	20.00	20.00	20.00	20.00
SoDSA	20.00	20.00	20.00	20.00	20.00	20.00
GSoDSA	20.00	20.00	20.00	20.00	20.00	20.00
ESoDSA	20.00	20.00	20.00	20.00	20.00	20.00
	Average Queue Time					
SoHC	50.60	50.60	50.60	50.60	50.60	50.60
GSoHC	50.60	50.60	50.60	50.60	50.60	50.60
ESoHC	50.60	50.60	50.60	50.60	50.60	50.60
SoDSA	50.70	50.70	50.69	50.69	50.70	50.71
GSoDSA	50.68	50.68	50.69	50.69	50.69	50.69
ESoDSA	50.68	50.69	50.68	50.69	50.69	50.69

Table 7.27. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 50 iteration and life span of 50 iterations and normal distribution with standard deviation of 0.5

	Constraint Tightness					
	0.01	0.02	0.03	0.04	0.05	0.06
	Stable Cycles					
SoHC	979.45	978.86	978.35	977.82	977.40	976.85
GSoHC	979.43	978.89	978.36	977.76	977.35	976.86
ESoHC	979.43	978.89	978.27	977.75	977.25	880.48
SoDSA	932.50	931.94	930.77	930.02	928.34	926.35
GSoDSA	932.71	930.18	925.90	921.51	914.95	907.06
ESoDSA	932.72	929.92	926.08	921.05	915.23	907.24
	Requests Satisfied					
SoHC	16.00	16.00	16.00	16.00	16.00	16.00
GSoHC	16.00	16.00	16.00	16.00	16.00	16.00
ESoHC	16.00	16.00	16.00	16.00	16.00	16.00
SoDSA	16.00	16.00	16.00	16.00	16.00	16.00
GSoDSA	16.00	16.00	16.00	16.00	16.00	16.00
ESoDSA	16.00	16.00	16.00	16.00	16.00	16.00
	Average Queue Time					
SoHC	200.88	200.88	200.88	200.88	200.88	200.88
GSoHC	200.88	200.88	200.88	200.88	200.88	200.88
ESoHC	200.88	200.88	200.88	200.88	200.88	200.89
SoDSA	202.03	202.06	202.13	202.21	202.26	202.35
GSoDSA	202.14	202.20	202.31	202.41	202.59	202.73
ESoDSA	202.14	202.23	202.30	202.42	202.58	202.74

Table 7.28. Results of all algorithms on the Sensor Sharing Problem with arrival intervals at 50 iteration and life span of 200 iterations and normal distribution with standard deviation of 0.5

of sensor requests among the differing sensor types. The closer the number of active requests approaches the maximum of 30, the harder the problem becomes, especially when the constraint tightness on the allocation policies is increased. As for the problems with requests arriving every interation with life spans of 50 and 200, the performance drops dramatically around the constraint tightness where there is only one feasible solution. The results reflect the prediction of where the hard problems would be based on equation 7.4.6.

For the first portion of the test, assuming that the uniform random number generator used to randomly generate the unit requests is truly uniform, it can be expected that the 30 randomly generated unit requests that are active at any given time will be evenly spread out among the different sensor types. If this is the case, then based on equation 7.4.6 with 30 external requests evenly distributed among the domain values, the constraint tightness at which the expected number of feasible solutions drops to 1 is 0.05. This explains why it is so hard to maintain a stable cycle when the tightness increases above 0.03.

For the second portion of the test, the Gaussian random number generator used to generate requests for sensors created a situation where the sensor types being requested were clustered. This made the problem harder, as predicted with 7.4.6, because of the reduction in the number of feasible solutions. However, the clustering also made the sensor sharing problem less dynamic, requiring less frequent sensor reassignment, which helped boost the performance of ESoHC and GSoHC along with SoDSA.

Overall, depending on the distribution of requests among sensor type, the length of time between the arrival of request and the life span of requests, a few conclusions can be

drawn. SoHC, with its greater diversity in population, performs better than either GSoHC or ESoHC when the requests are more evenly distributed among sensor types. When the requests are not uniformly distributed among the sensor types, the performance of SoHC drops while ESoHC and GSoHC performs much better. Generally, SoHC will perform better when the problem is more dynamic. For the DSA based GEPs, the results are slightly different, as ESoDSA has a slight edge when the distribution of requests is more uniform, while SoDSA performs slightly better with non-uniform requests. ESoDSA performs the best for easier problems, while SoDSA performs better on the harder problems, especially with non-uniform requests. However, none of the DSA based GEPs is able to perform acceptably on harder problems.

CHAPTER 8

CONCLUSIONS AND FURTHER RESEARCH

With the growing popularity and usage of distributed architectures in computing, distributed resource allocation and distributed constraint satisfaction problems will become much more common. As noted earlier, a great deal of work has been devoted to solving the traditional centralized resource allocation and constraint satisfaction problems and good solution methods and strategies have been found. However, a distributed architecture poses a new challenge for traditional methods with unique requirements, specifically decentralized control. Even when adapted to solve distributed problems, the methods only produce good results for fairly easy problems, for example, distributed asynchronous backtracking or asynchronous weak commitment searches.

This research demonstrated the use of genetic and evolutionary protocols (GEPs) that utilize a truly distributed genome for candidate solutions, along with distributed genetic and evolutionary operators, to solve distributed constraint satisfaction and resource allocation problems. The tests on randomly generated distributed asymmetric constraint satisfaction problems (DisACSPs) showed that GEPs are superior to the current best known algorithms, the distributed breakout algorithm (dBA) and the distributed stochastic algorithm (DSA).

The results also showed that dBA and DSA can both be modified with a population based approach (SoHC and SoDSA) to increase performance with only a linear increase

in overhead. The genetic and evolutionary operators used to increase the performance of SoHC can also be used to improve performance for other similar algorithms. However, this direct implementation of the genetic and evolutionary operators to create DSA based GEPs still has room for improvement.

The sensor network tracking and sharing problems presented interesting application test beds for GEPs. GEPs performed well for the sensor tracking problem, since the problem is fairly static and reassignment of sensor duty does not have to take place very often. However, for the sensor sharing problem, where the external requests are more varied and there is a much more dynamic scenario with frequent reassignments of sensor duty, GEPs did not perform as well as plain SoHC. As mentioned earlier, this is completely due to the lack of population diversity caused by the use of the crossover and mutation operators, which help GEPs find solutions faster than SoHC, but increase the time they need to adapt to changes in the problem.

In summary, this dissertation presented and demonstrated the performance and advantages of a new type of distributed EC utilizing genetic and evolutionary protocols. The genetic and evolutionary protocols easily outperformed the best known algorithms in solving distributed constraint satisfaction and resource allocation problems and show great potential. The distributed genetic and evolutionary operators used here can also be applied to other distributed methods and are likely to improve their performances.

For further research, the possibility of creating a DSA with an adaptive p value that can increase its performance is an avenue to explore. Preliminary trials in this work did not yield good results, but were generally inconclusive. There is also the open question

of whether a breakout list will help DSA. It could very well be that other DSA models are more suitable for such modifications.

Also, the implementation of the sensor tracking problem used for testing GEPs was fairly simplistic, as the focus was to show that GEPs can be utilized to solve such problems and get preliminary results to assess the performance. The next step would be to move towards a more realistic model of the sensor tracking problem. For example, the removal of the assumption of perfect visibility would change the entire problem and make it harder. Topology considerations can also be added to create more realistic ommunication models.

Further more, in this work, unit requests were used for the sensor sharing problem to simplify the testing and to help discover more about the general behavior of GEPs when there is a need to satisfy both external and internal constraints. However, more realistically, more tests need to be performed using block requests of varying sizes to further explore the nature of the sensor sharing problem.

Lastly, the approach towards the sensor tracking and sharing problems in this work is at a very high level, which opens up the possibility of performing lower level analysis to determine the impact of GEPs on the power consumption of each individual pod and the general communication latency required to create global views.

BIBLIOGRAPHY

[1] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. "Wireless Sensor Networks: a Survey." *Computer Networks* Vol. 38, pg. 393-422, 2002.

[2] Angeline, P. "Adaptive and Self-Adaptive Evolutionary Computations." *Computational Intelligence: A Dynamic Systems Perspective*. IEEE Press, pp152-163, 1995.

[3] Aslam, J., Butler, Z., Constantin, F., Crespi, V., Cybenko, G., and Rus, D. "*Tracking a Moving Object with a Binary Sensor Network.*" *Proceedings of SenSys '03*.

[4] Bäck, T., Hoffmeister, F. and Schwefel, H.-P. "A Survey of Evolution Strategies." In *Proceedings of the Fourth International Conference on Genetic Algorithms*, 2-9. San Mateo, CA 1991.

[5] BarNoy, A. et al. "On Chromatic Sums and Distributed Resource Allocation." *Information and Computation*, Vol. 140, No. 2, pp183-202, 1998.

[6] Bazaraa, M. S., Sherali, H. D., and Shetty, C. M.. *Nonlinear Programming: Theory and Algorithms*. John Wiley & Sons, Inc., 1993.

[7] Bejar, R., Krishnamachari, B., Gomes, C., and Selman, B. "Distributed Constraint Satisfaction in a Wireless Sensor Tracking System." *Workshop on Distributed Constraint Reasoning, International Conference on Artificial Intelligence*, 2001.

[8] Belding T. C. "The Distributed Genetic Algorithm Revisited." *Proceedings of the Sixth Intl. Conf. on Genetic Algorithms*, pages 114--121, San Mateo, CA, 1995.

[9] Bellman, R. "Some Applications of the Theory of Dynamic Programming-A Review." *Journal of the Operations Research Society of America***,** Vol. 2, No. 3 (Aug., 1954), pp. 275-288.

[10] Bertsekas, D. P.. *Nonlinear Programming*. Athena Scientific. Massachusetts, 1995.

[11] Bharathidasan, A., Ponduru, V. A. S. "Sensor Networks: An Overview." IEEE INFOCOM '04.

[12] Bhuvaneswaran, R. S., Bordim, J. L., Cui, J. T., and Nakano, K.. "Fundamental Protocols for Wireless Sensor Networks." IPDPS, April 2001.

[13] Carter, R. L., Louis, D. St., and Andert, E. P. Jr. "Resource Allocation in a Distributed Computing Environment." *Proceedings of Digital Avionics Systems Conference*, 1998.

[14] Burghart, T.. "Distributed Computing Overview." QUOIN Inc., June, 1998.

[15] Cassandras, C. G. and Julka, V. "Descent Algorithms for Discrete Resource Allocation Problems." *Proceedings of the 33rd Conference on Decision and Control*, 1994.

[16] Chandy, K. M. and Misra, J. "The Drinking Philosophers Problem." *ACM Transactions on Programming Languages and Systems*, 1984.

[17] Charnes, A. and Cooper, W. W. "The Theory of Search: Optimum Distribution of Search Effort." *Management Science*, Vol. 5, No. 1 (Oct., 1958) , pp. 44-50.

[18] Chellapilla, K. and Fogel, D. B. "Exploring Self-Adaptive Methods to Improve the Efficiency of Generating Approximate Solutions to Traveling Salesman Problems using Evolutionary Programming." *Evolutionary Programming VI*, Springer, Berlin (1997) 361-371.

[19] Chevaleyre, Y. et la. "Issues in Multiagent Resource Allocation." *Informatica*, 30:3-31, 2006

[20] Choy, M. and Singh, A. K.. "Efficient Fault-Tolerant Algorithms for Distributed Resource Allocation." *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 3, pp535-559, 1995.

[21] Commander, C. W. "A Survey of the Quadratic Assignment Problem, with Applications." *Morehead Electronic Journal of Applicable Mathematics*, Issue 4, 2005.

[22] Coffman, K. G. and Odlyzko, A. M.. "Growth of the Internet." *Optical Fiber Telecommunications IV B: Systems and Impairments*, I. P. Kaminowand T. Li, eds., pp. 17-56, Academic Press, 2002.

[23] Coit, D. W. and Smith, A. E.. "Solving the Redundancy Allocation Problem Using a Combined Neural Network/Genetic Algorithm Approach." *Computers & Operations Research*, July 1995.

[24] Coit, D. W. and Liu, J. "System Reliability Optimization with k-out-of-n Subsystems." *International Journal of Reliability, Quality, and Safety Engineering*, Vol. 7, No. 2, pp129-142, 2000.

[25] Coleri, S., Puri, A., and Varaiya, P. "Power Efficient System for Sensor Networks." *Eighth IEEE International Symposium on Computers and Communication Proceedings. (ISCC 2003).*, July 2003.

[26] Day, R. O., Kleeman, M. P., and Lamont, G. B. "Solving the Multi-Objective Quadratic Assignment Problem Using a Fast Messy Genetic Algorithm." *Proceedings of CEC'03*.

[27] Debeau, D. E. "Linear Programming Isn't Always the Answer." *Operations Research*, Vol. 5, No. 3 (Jun., 1957) , pp. 429-433.

[28] Dechter, R. "Constraint Networks." *Encyclopedia of Artificial Intelligence*, second edition, pp276-295, Wiley and Sons, 1992.

[29] DeJong, K. and Spears, W. (1993). "On the State of Evolutionary Computation." *Proceedings of the Fifth ICGA*, 618-623. Kaufmann, San Mateo, CA.

[30] Dodin, P., Verliac, J., and Nimier, V. "Analysis of the Multisensor Multitarget Tracking Resource Allocation Problem," in: *Proceedings, 3rd International Conference on Information Fusion*, 2000, pp. WeC1—17—22.

[31] Dorigo, M and Di Caro, G. "Ant Algorithms for Discrete Optimization." *Artificial Life*, Vol 5, No. 3, pp137-172, 1999.

[32] Dorigo, M., Bonabeau, E., and Theraulaz, G.. (1999). *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press.

[33] Dorigo, M. and Gambardella, L. M. "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem." *IEEE Transactions on Evolutionary Computation*, Vol 1., No. 1, 1997.

[34] Dorigo, M., Maniezzo, V., and Colorni, A. "Ant System: Optimization by a Colony of Cooperating Agents." *IEEE Transactions on Systems, Man and Cybernetics*, Part B, Volume: 26 , Issue: 1, Pages:29 – 41, Feb. 1996.

[35] Dozier, G. "Distributed Constraint Satisfaction via a Society of Hill-Climbers." In *Proceedings of the 2002 World Automation Conference(International Symposium on Soft Computing with Industrial Applications)*, Orlando Florida, June 9-13.

[36] Dozier, G. "Solving Distributed Asymmetric Constraint Satisfaction Problems Using an Evolutionary Society of Hill-Climbers." *Proceedings of Genetic and Evolutionary Computation Conference (GECCO-2003).*

[37] Dozier, G. and Rupela, V. "Solving Distributed Asymmetric CSPs via a Society of Hill-Climbers." *Proc. Of IC-AI'02*, pp. 949-953, CSREA Press.

[38] Dozier , G. "Sharing the Sensor Web via Recurrent Distributed Meta-Evolutionary Constraint Satisfaction." *Proceedings of the 2003 Conference on Space Mission Challenges for Information Technology (SMC-IT 2003)* , pp. 153-160 , July 13-16 , Pasadena , CA.

[39] Dozier, G., Bowen, J., and Bahler, D. "Solving Randomly Generated Constraint Satisfaction Problems Using a Micro-Evolutionary Hybrid that Evolves a Population of Hill-Climbers." *Proceedings of the 2nd IEEE Conference on Evolutionary Computation*, pages 614-619, 1995.

[40] Dozier, G., Bowen, J., and Homaifar, A. "Solving Constraint Satisfaction Problems Using Hybrid Evolutionary Search." *IEEE Transactions on Evolutionary Computation*, Vol. 2, No. 1, April 1998.

[41] Dozier, G. V., Cunningham, H., Britt, W., and Zhang, F. "Distributed Constraint Satisfaction, Restricted Recombination, and Hybrid Genetic Search." *GECCO 2004*: 1078-1087.

[42] Dreyfus, S. E. "Computational Aspects of Dynamic Programming." *Operations Research*, Vol. 5, No. 3 (Jun., 1957) , pp. 409-415.

[43] Eiben, A. E., Raue, P-E., and Ruttkay, Zs. "Solving Constraint Satisfaction Problems Using Genetic Algorithms." *The 1st IEEE Conference on Evolutionary Computation*, pp. 542-547. 1994.

[44] V. Ekanayake, C Kelly, IV, and R Manohar. *"An Ultra-Low Power Processor for Sensor Networks*." Proceedings of ASPLOS '04.

[45] Fabiunke, M. "Parallel Distributed Constraint Satisfaction." In *Proc. Intern. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA-99)*, pages 1585–1591, 1999.

[46] Fang, Q., Zhao, F., and Guibas, L. "Counting Targets: Building and Managing Aggregates in Wireless Sensor Networks." Xerox Palo Alto Research Center (PARC) Technical Report, June 2002.

[47] Fiacco, A. V. and McCormick, G. P.. *Nonlinear Programming: Sequential Unconstrained Minimization Techniques.* Society for Industrial and Applied Mathematics, Philadelphia, 1990.
[48] Fitzpatrick, S. and Meertens, L. "An Experimental Assessment of a Stochastic, Anytime, Decentralized, Soft Colourer for Sparse Graphs." In *Proc. 1st Symp. on Stochastic Algorithms: Foundations and Applications*, pages 49–64, 2001.
[49] Fogel, D. B. "The Advantages of Evolutionary Computation." *Proceedings of Biocomputing and Emergent Computation*, 1997.
[50] Fogel, D. B. and Chellapilla, K. "Revisiting Evolutionary Programming." *SPIE Aerosense98, Applications and Science of Computational Intelligence*, Orlando, FL, pp. 2-11, 1998.
[51] Fogel, D. B. *Evolutionary Computation: Towards a New Philosophy of Machine Intelligence.* IEEE Press, New York, 2000.
[52] Fogel, L. J., Owens, A. J., & Walsh, M. J. *Artificial Intelligence Through Simulated Evolution*. New York: Wiley Publishing, 1966.
[53] Fogel, L. J., Angeline, P. J., and Fogel, D. B. (1995). "An Evolutionary Programming Approach to Self-Adaptation on Finite State Machines." *Proceedings of the Fourth International Conference on Evolutionary Programming*, 355—365.
[54] Freuder, E. C., Minca, M., and Wallace, R. J. "Privacy/Efficiency Tradeoffs in Distributed Meeting Scheduling by Constraint-Based Agents." *Distributed Constraint Reasoning*, pp. 63-70, 2001.
[55] Fu, S. and Dozier, G. V. "Solving Distributed Constraint Satisfaction Problems with an Ant-Like Society of Hill-Climbers." IC-AI 2003: 263-269.
[56] Galinier, P. and Hao, J-K.. "Hybrid Evolutionary Aglorithms for Graph Coloring." *Journal of Combinatorial Optimization 3*, 379–397 (1999)
[57] Galstyan, A., Krishnamachari, B., Lerman, K. "Resource Allocation and Emergent Coordination in Wireless Sensor Networks." *American Association of Artificial Intelligence*, 2004.
[58] Glover, F. and Laguna, M. *Tabu Search*. Springer, 1998.
[59] Glover, F., Taillard, E., and D. de Werra. "A User's Guide to Tabu Search." *Annals of Operation Research* vol. 41, pp. 3-28, 1993.
[60] Glover, F., Laguna, M. and Marti, R. "Fundamentals of Scatter Search and Path Relinking." *Control and Cybernetics*, 29/3 (2000), 653-684.
[61] Gorges-Schleuter, M. "ASPARAGOS An Asynchronous Parallel Genetic Optimization Strategy." *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 422-427, 1989.
[62] de Guenin, J. "Optimum Distribution of Effort: An Extension of the Koopman Basic Theory." *Operations Research***,** Vol. 9, No. 1 (Jan. - Feb., 1961) , pp. 1-7**.**
[63] Hadj-Alouane, A. B., and Bean, J. C. "A Genetic Algorithm for the Multiple-Choice Integer Program." *Operations Research*, Vol. 45, No. 1 (Jan. - Feb., 1997) , pp. 92-101.
[64] Handa, H., Katai, O., Baba, N., and Sawaragi, T. "Solving Constraint Satisfaction Problems by Using Coevolutionary Genetic Algorithms." *Proceedings of 1998 IEEE Internation Conference on Evolutionary Computation*, pp. 21-26.

[65] Hillier, F. S. and Lieberman, G. J. *Introduction to Operations Research*. McGraw-Hill Inc. New York, 1995.
[66] Hinterding, R., Michalewicz, Z. and Eiben, A. E. "Adaptation in Evolutionary Computation: A Survey." *Proceedings of the IEEE Conference on Evolutionary Computation*, 1997.
[67] Horng, J-T. et la. "Resolution of Quadratic Assignment Problems Using an Evolutionary Algorithm." *Proceedings of CEC'00*.
[68] Hubaux, J.P. and Enz, C. "Minimum Energy Broadcast in All-Wireless Networks: NP-Completeness and Distribution Issues." *Proceedings of MOBICOM '02*, September, 2002.
[69] Hung, M. S. "A Polynomial Simplex Method for the Assignment Problem." *Operations Research***,** Vol. 31, No. 3 (May - Jun., 1983) , pp. 595-600**.**
[70] Ibaraki, T. and Katoh, N. *Resource Allocation Problems: Algorithmic Approach*. The MIT Press, Massachusetts, 1988.
[71] Kang, I. and Poovendran, R. "A Novel Power-Efficient Broadcast Routing Algorithm Exploiting Broadcast Efficiency." *Proceedings of IEEE Vehicular Technology Conference*, pp. 2926-2930, Orlando, FL, Oct. 6-9, 2003.
[72] Karmarkar, N. "A New Polynomial-time Algorithm for Linear Programming." *Combinatorica*, vol. 4, issue 4, pp. 373-395, 1994.
[73] Karush, W. "A General Algorithm for the Optimal Distribution of Effort." *Management Science*, Vol. 9, No. 1 (Oct., 1962) , pp. 50-72.
[74] Kennedy, J. "The Behavior of Particles." *Proceedings of the 7th International Conference on Evolutionary Programming VII*, pp. 581-589, 1998.
[75] Kennedy, J. and Eberhart, R. C. "Particle Swarm Optimization." *Proc. IEEE int'l conf. on neural networks* Vol. IV, pp. 1942-1948. IEEE service center, Piscataway, NJ, 1995.
[76] Kirkpatrick, S., Gelatt, C. D. Jr., and Vecchi, M. P. "Optimization by Simulated Annealing" *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*, pp.606 – 615, Morgan Kaufmann Publishers Inc., 1987.
[77] Klee, V. and Minty, G. J. "How Good is the Simplex Algorithm?" In O. Shisha, editor, *Inequalities III*, pages 159-175. Academic Press, New York, NY, 1972
[78] Kodialam, M. S. and Luss, H. "Algorithms for Separable Nonlinear Resource Allocation Problems." *Operations Research*, Vol. 46, No. 2 (Mar. - Apr., 1998) , pp. 272-284.
[79] Koopman, B. O. "The Optimum Distribution of Effort." *Journal of the Operations Research Society of America*, Vol. 1, No. 2 (Feb., 1953) , pp. 52-63.
[80] Koza, J. R. *Genetic Programming*. MIT Press, 1992.
[81] Kubisch, M., Karl, H., Wolisz, A., Zhong, L. C., and Rabaey, J. "Distributed Algorithms for Transmission Power Control in Wireless Sensor Networks." *Wireless Communications and Networking (WCNC'03)*, March 2003.
[82] Kulturel-Konak, S., Smith, A. E., and Coit, D. W. "Efficiently Solving the Redundancy Allocation Problem Using Tabu Search." *IIE Transactions*, 2003.
[83] Kulturel-Konak, S., Norman, B. A., Coit, D. W., and Smith, A. E. "Exploiting Tabu Search Memory in Constrained Problems." *INFORMS Journal of Computing*, Vol. 16, No. 3, pp.241-254, Summer 2004.

[84] Kumar, V. "Algorithms for Constraint Satisfaction Problems: A Survey." *AI Magazine*, 1992.

[85] Kuo, W. and Prasad, V. R. "An Annotated Overview of System-Reliability Optimization." *IEEE Transactions on Reliability*, 49/2 (2000), 176-187.

[86] Kuwabara, K., Ishida, T., Nishibe, Y., and Suda, T. "An Equilibratory Market-Based Approach for Distributed Resource Allocation and Its Application to Communication Network Control." From *Market-Based Control: A Paradigm for Distributed Resource Allocation*, World-Scientific, Singapore, 1995.

[87] Li, D., Wong, K. D., Hu, Y. H., and Sayeed, A. M. "Detection, Classification, and Tracking of Targets." *IEEE Signal Processing Magazine*, March 2002.

[88] Liang, Y-C., Kulturel-Konak, S., and Smith, A. E. "Meta Heuristics for the Orienteering Problem." *Proceedings of CEC'02*, 2002.

[89] Liang, Y-C. and Smith, A. E. "An Ant System Approach to Redundancy Allocation." *Proceedings of CEC'99*, 1999.

[90] Liang, Y-C. and Smith, A. E. "Ant Colony Optimization for Constrained Combinatorial Problems." *Proceedings of 5th International Conference on Industrial Engineering*, 2000.

[91] Liang, Y-C. and Smith, A. E. "An Ant Colony Optimization Algorithm for the Redundancy Allocation Problem (RAP)." *IEEE Transactions on Reliability*, Vol. 53, No. 3, 2004.

[92] Eliane M. L., et al. "An Analytical Survey for the Quadratic Assignment Problem." To appear in *European Journal of Operations Research*.

[93] MacIntyre, E., Prosser, P., Smith, B., and Walsh, T. "Random Constraint Satisfaction: Theory Meets Practice." *The Proc. Of CP-98*, pp. 325-339, Springer-Verlag.

[94] Mackworth, A. K. (1977). "Consistency in the Networks of Relations". *Artificial Intelligence, 8 (1)*, pp. 99-118.

[95] Mailer, R. and Lesser, V. "Cooperative Negotiation for Optimized Distributed Resource Allocation in Soft-Real-Time." *Proceedings of Second International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2003)*, ACM Press, pp. 576-583. July 2003.

[96] Manderick, B., and Spiessens, P. "Fine-grained parallel genetic algorithms." *Proceedings of the Third International Conference on Genetic Algorithms*, pages 428-433, 1989.

[97] Maniezzo, V. and Colorni, A. "The Ant System Applied to the Quadratic Assignment Problem." *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 5, 1999.

[98] Maruyama, T., and Hirose, T., and Konagaya, A. "A Fine-Grained Parallel Genetic Algorithm for Distributed Parallel Systems." *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 184-190, 1993.

[99] McErlean, D. and Narayanan, S. "Distributed Detection and Tracking in Sensor Networks." *36th Asilomar Conf. Signals, Systems and Computers*, 2002.

[100] Meguerdichian, S., Koushanfar, F., Potkonjak, M., and Srivastava M. B. "Coverage Problems in Wireless Ad-hoc Sensor Networks." *Proceedings of IEEE Infocom*, Vol 3, pg. 139-150, April 2001.

[101] Men, P. and Freisleben, B. "A Comparison of Memetic Algorithms, Tabu Search, and Ant Colonies for the Quadratic Assignment Problem." *Proceedings of CEC'99*.

[102] Mendelson, H., Pliskin, J. S., and Yechiali, U. "A Stochastic Allocation Problem." *Operations Research*, Vol. 28, No. 3, Part 2 (May - Jun., 1980) , pp. 687-693.

[103] Miehle, W. "Numerical Solution of the Problem of Optimum Distribution of Effort." *Journal of the Operations Research Society of America*, Vol. 2, No. 4 (Nov., 1954) , pp. 433-440.

[104] Min, R., Bhardwaj, M., Cho, S-H., Shih, E., Sinha, A., Wang, A., and Chandrakasan, A. "Low-Power Wireless Sensor Networks." In *Proceedings of Fourteenth International Conference on VLSI Design*, Bangalore, India, January 2001.

[105] Min, R. and Chandrakasan, A. "Top Five Myths about the Energy Consumption of Wireless Communication." *ACM Sigmobile Mobile Computing and Communications Review (MC2R)*, January 2003

[106] Minton, S., Johnston, M. T., Philips, A. B., and Laird, P. "Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems." *Arti.cial Intelligence*, 58:161-205, 1992.

[107] Mitchell, M. *An Introduction to Genetic Algorithms*. The MIT Press, Cambridge, MA, 1996.

[108] Modi, P. J., Shen, W-M., and Tambe, M. "Distributed Resource Allocation: Formalization, Complexity Results and Mapping to Distributed CSPs." *Principles and Practice of Constraint Programming*, 2001.

[109] Modi, P. J. et la. "Dynamic Distributed Resource Allocation: A Distributed Constraint Satisfaction Approach." *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages*, pp. 181-193, 2001.

[110] Modi, P. J., Shen, W-M., Tambe, M., and Yokoo, M. "An Asynchronous Complete Method for Distributed Constraint Optimization." In *Proc of Autonomous Agents and Multi-Agent Systems*, 2003."

[111] Morin, T. L. and Marsten, R. E. "Branch-and-Bound Strategies for Dynamic Programming." *Operations Research*, Vol. 24, No. 4 (Jul. - Aug., 1976) , pp. 611-627.

[112] Morris, P. 1993. "The Breakout Method for Escaping from Local Minima." In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 40-45.

[113] Nguyen, T. and Deville, Y. "A Distributed Arc Consistency Algorithm." *Science and Computer Programming*, Vol. 30, no. 1-2, pp227-250, 1998.

[114] Norman, J. M. and White, D. J. "A Method for Approximate Solutions to Stochastic Dynamic Programming Problems Using Expectations." *Operations Research*, Vol. 16, No. 2 (Mar. - Apr., 1968) , pp. 296-306.

[115] Owechko, Y. and Shams, S. "Comparison of Neural Network and Genetic Algorithms for a Resource Allocation Problem." *IEEE World Congress on Computational Intelligence*, vol. 7, pp. 4655-4660, 1994.

[116] Page, I., Jacob, T., and Chern, E. "Fast Algorithms for Distributed Resource Allocation." *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, 1993.

[117] van de Panne, C., Whinston, A., and Beale, E. M. L. "A Comparison of Two Methods for Quadratic Programming." *Operations Research*, Vol. 14, No. 3 (May - Jun., 1966) , pp. 422-443.
[118] Pannell, D. J.. *Introduction to Practical Linear Programming*. John Wiley & Sons, Inc., New York, 1997.
[119] Pishro-Nik, H., Chan, K., Fekri, F. "On Connectivity Properties of Large-Scale Sensor Networks." *IEEE Sensor and Ad Hoc Communications and Networks* 2004.
[120] Prosser, P., Conway, C., and Muller, C. "A Constraint Maintenance System for the Distributed Constraint Satisfaction Problem." *Intelligent Systems Engineering, v*.1 n.1, p. 76-83, Autumn 1992.
[121] Prosser, P., Conway, C., and Muller, C. "A Constraint Maintenance System for the Distributed Resource Allocation Problem." *Intelligent Systems Engineering 1*, 76—83.
[122] Rabbat, M. and Nowak, R. "Distributed Optimization in Sensor Networks." *3rd International Symposium on Information Processing in Sensor Networks*, April 2004.
[123] Randall, M. "A General Meta-Heuristic Based Solver for Combinatorial Optimization Problems." *Computational Optimization and Applications*, 2001.
[124] Raynal, M. "A Distributed Solution to the k-out of-m Resources Allocation Problem." *Proceedings of the International Conference on Computing and Information*, 1991.
[125] Riff, M-C. "Evolutionary Algorithms for Constraint Satisfaction Problems." *Proceedings of the XVIII International Conference of the Chilean Computer Science Society*, pp158, 1998.
[126] Rudolph, G. "Global Optimization by Means of Distributed Evolution Strategies" *Proceedings of the First Conference on Parallel Problem Solving from Nature*, pp. 209—213.
[127] Russell, S. J. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.
[128] Salhieh, A., Weinmann, J., Kochhal, M., and Schwiebert, L. "Power Efficient Topologies for Wireless Networks." *Proceedings of International Conference on Parallel Processing*, 2001.
[129] Schaofs, L. and Naudts, B. "Ant Colonies are Good at Solving Constraint Satisfaction Problems." In *Proc. of the 2000 Congress on Evolutionary Computation*, pages 1190-195.
[130] Schiavone, G., Wahid, P., Van Doorn, E., Palaniappan, R., and Tracy, J. "Target Detection and Tracking Using a UWB Sensor Web" Antennas and Propagation Society Symposium, 2004. EEE , Volume: 2 , 20-25 June 2004 Pages:1287 - 1290 Vol.2.
[131] Sebag, M. and Shoemauer, M. "A Society of Hill-Climbers." *The Proc. Of ICEC-97*, pp. 319-324, IEEE Press, 1997.
[132] Shamblin, J. E. and Stevens, G. T. Jr. *Operations Research: A Fundamental Approach*. McGraw-Hill Inc., New York, 1974.
[133] Shamir, R. "The Efficiency of the Simplex Method: A Survey." *Management Science*, Vol. 33, No. 3 (Mar., 1987) , pp. 301-334.

[134] Shaprio, J. "Dynamic Programming Algorithm for the Integer Programming Problem I: The Integer Programming Problem Viewed as a Knapsack Type Problem." *Operations Research*, **16**, 103-121.
[135] Shenoy, G. V. *Linear Programming: Methods and Applications.* John Wiley & Sons, Inc., New York, 1989.
[136] Shin, J., Guibas, L. J., and Zhao, F. "A Distributed Algorithm for Managing Multi-Target Identities in Wireless Ad-hoc Sensor Networks." *2nd Workshop on Information Processing in Sensor Networks (IPSN '03)*, April 2003.
[137] Shonkwiler, R. "*Parallel Genetic Algorithms*." ICGA 1993: 199-205.
[138] Srinivasan, V., Nuggehalli, P., and Rao, R. "Design of Optimal Energy Aware Protocols for Wireless Sensor Networks." *Vehicular Technology Conference (VTC'01). IEEE VTS 53rd*, May 2001
[139] Smith, B. (1994). "Phase Transition and the Mushy Region in Constraint Satisfaction Problems," *Proc. Of ECAI-94*, pp. 100-104, John Wiley & Sons Ltd.
[140] Sohrabi, K., Gao, J., Ailawadhi, V., and Pottie, G. J. "Protocols for Self-Organization of a Wireless Sensor Network." *IEEE Personal Communications*, 7, October 2000.
[141] Solmon, C. (2002). "Ants can Solve Constraint Satisfaction Problems", to appear in: *IEEE Transactions on Evolutionary Computation,* IEEE Press.
[142] Spears, W. M., De Jong, K. A., Bäck, T., Fogel, D. B., and de Garis, H. (1993). "An Overview of Evolutionary Computation," *The Proceedings of the European Conference on Machine Learning*, v667, pp. 442-459.
[143] Taha, H. A. *Operations Research: An Introduction*. MacMillan Publishing Co., Inc. New York, 1971.
[144] Tanese, R. "Distributed Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms*, Pages: 434 - 439, 1989.
[145] Tardos, E. "A Strongly Polynomial Algorithm to Solve Combinatorial Linear Programs." *Operations Research*, Vol. 34, No. 2 (Mar. - Apr., 1986) , pp. 250-256.
[146] Tate, D. M. and Smith, A. E. "A Genetic Approach to the Quadratic Assignment Problem." *Computers Ops Res.*, Vol. 22, No. 1, pp73-83, 1995.
[147] Tasgetiren, M. F. and Smith, A. E. "A Genetic Algorithm for the Orienteering Problem." *Proceedings of CEC'00*, 2000.
[148] Tel, Gerard. *Introduction to Distributed Algorithms*. Cambridge University Press,1994.
[149] Tian, D. and Georganas, N. D. "Energy Efficient Routing with Guaranteed Delivery in Wireless Sensor Networks." *WCNC 2003*, March 2003.
[150] Tilak, S., Abu-Ghazaleh, N. B., Heinzelman, W. "A Taxonomy of Wireless Micro-Sensor Network Model." *ACM SIGMOBILE Mobile Computing and Communications Review*, 2002.
[151] M. Tubaishat and S. Madria. "Sensor Networks: An Overview." *IEEE Potentials*, 22, 2, 20-23, April 2003.
[152] Tuomi, I. "The Lives and Death of Moore's Law." *First Monday*, volume 7, number 11 (November 2002).
[153] Wagner, H. M. "A Comparison of the Original and Revised Simplex Methods." *Operations Research*, Vol. 5, No. 3 (Jun., 1957) , pp. 361-369.

[154] Wagner, H. M. "The Simplex Method for Beginners." *Operations Research*, Vol. 6, No. 2 (Mar. - Apr., 1958) , pp. 190-199.

[155] Wan, P-J. and Yi, C-W. "Asymptotic Critical Transmission Radius and Critical Neighbor Number for k-Connectivity in Wireless Ad Hoc Networks." *MobiHoc*, May 2004

[156] Wattenhofer, R., Li, L., Bahl, P., Wang, Y. M. "Distributed Topology Control for Power Efficient Operation in Multihop Wireless Ad hoc Networks." *Proc. IEEE Infocom 2001*.

[157] Walukiewicz, S. *Integer Programming*. Kluwer Academic Publishers. Boston, 1991.

[158] Wolfe, P. "Some Simplex-Like Nonlinear Programming Procedures." *Operations Research*, Vol. 10, No. 4 (Jul. - Aug., 1962) , pp. 438-447.

[159] Wong, P. J. and Luenberger, D. G. "Reducing the Memory Requirements of Dynamic Programming." *Operations Research*, Vol. 16, No. 6 (Nov. - Dec., 1968), pp. 1115-1125.

[160] Wu, T., Ye, N., and Zhang, D. "Comparison of Distributed Methods for Resource Allocation." *International Journal of Production Research*, Vol. 43, No. 3, pp515-536, 2005.

[161] Xue, F. and Kumar, P. R. "The Number of Neighbors Needed for Connectivity of Wireless Networks." *Wireless Networks 10*, 169–181, 2004.

[162] Yangt, C. C. and Yang, M-H. "Constraint Networks: A Survey." *IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 2, pp1930-1935, 1997.

[163] Ye, W., Heidemann, J., Estrin, D. "An Energy-Efficient MAC Protocol for Wireless Sensor Networks." *Proceedings of INFOCOM, 2002*.

[164] Yokoo, M. *Distributed Constraint Satisfaction*, Springer-Verlag.

[165] Yokoo, M., Ishida, T., Durfee, E., and Kuwahara, K. "Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving." *Proceedings of 12th IEEE International Conference on Distributed Computing Systems '92*, pp. 614-621.

[166] Yokoo, M., Durfee, E.,Ishida, T., and Kuwahara, K. "The Distributed Constraint Satisfaction Problem: Formalization and Algorithms." *IEEE Transaction on Knowledge and DATA Engineering*, vol 10, No. 5, September 1998.

[167] Yokoo, M and Hirayama, K. "Algorithms for Distributed Constraint Satisfaction: A Review." *Autonomous Agents and Multi-Agent Systems*, vol. 3, no. 2, pp. 198-212, 2000.

[168] Yu, Y., Krishnamachari, B., and Prasanna, V. K. "Energy-Latency Tradeoffs for Data Gathering in Wireless Sensor Networks." *IEEE Infocom'04, 2004.*

[169] Zangwill, W. I. "The Convex Simplex Method." *Management Science*, Vol. 14, No. 3, Theory Series (Nov., 1967) , pp. 221-238.

[170] Zhang, W. and Wittenburg, L. "Distributed Breakout Revisited." In *AAAI-2002*, Edmonton Alberta Canada, 2002.

[171] Zhang, W and Xing, Z. "Distributed Breakout vs. Distributed Stochastic: A Comparative Evaluation on Scan Scheduling." *AAMAS-02 Workshop on Distributed Constraint Reasoning*.

[172] Zhang, W, Wang, G. and Wittenburg, L. "Distributed Stochastic Search for Constraint Satisfaction and Optimization: Parallelism, Phase Transitions and Performance". In *Workshop on Probabilistic Approaches in Search AAAI-2002*, pages 53 – 59, Edmonton Alberta Canada, July 2002.

[173] Zhang, W., Xing, Z., Wang, G., and Wittenburg, L. "An Analysis and Application of Distributed Constraint Satisfaction and Optimization Algorithms in Sensor Networks." In *Proc. AAMAS-2003*, pages 185 – 192, Melbourne Australia, July 2003.

[174] Zhang, W., Deng, Z., Wang, G., Wittenburg, L., and Xing, Z. "Distributed Problem Solving in Sensor Networks." In *Proc. AAMAS-02*.

[175] Zhao, J. and Govindan, R. "Understanding Packet Delivery Performance In Dense Wireless Sensor Networks." *The First ACM Conference on Embedded Networked Sensor Systems (Sensys'03)*, November 2003

[176] Zhao, F. and Guibas, L. *Wireless Sensor Networks: An Information Processing Approach*. Morgan Kaufmann Publishers, 2004.

[177] Zhu, J. and Papavassiliou, S. "On the Connectivity Modeling and the Tradeoffs Between Reliability and Energy Efficiency in Large Scale Wireless Sensor Networks." *WCNC*, March 2003.

[178] Zuniga, M. and Krishnamachari, B. "Optimal Transmission Radius for Flooding in Large Scale Sensor Networks." *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, 2003.