Particle Swarm Optimization and Priority Representation

by

Philip Brooks

(Under the direction of W. Don Potter)

Abstract

In this thesis I examine the poor performance of Discrete Particle Swarm Optimization when applied to forest planning, an optimization problem in which the goal is to maintain an even flow of timber from a forested area. I consider an alternative priority representation that encodes a permutation or ordering of plan elements in real numbers to improve the handling of constraints. I also examine its applications to two other constrained optimization problems, n-queens and snake-in-a-box, in order to show how it handles different kinds of problems. I find that priority representation is a useful tool for optimization within constraints.

Index words: Particle swarm optimization, Priority representation, Forest planning, N queens, Snake-in-a-box, Permutation representation

Particle Swarm Optimization and Priority Representation

by

Philip Brooks

B.A., Piedmont College, 2000

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2010

Particle Swarm Optimization and Priority Representation

by

Philip Brooks

Approved:

Major Professor: W. Don Potter

Committee: Khaled Rasheed
Peter Bettinger

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2010

Dedication

This thesis is dedicated with love to Nora Camann.

Preface

The first three chapters of this thesis and the later sections concerning the forest planning problem are expanded sections of a paper I wrote with W. D. Potter and submitted to IEA-AIE 2011.

I would like to thank my parents, Philip John Brooks and Linda Brooks, and my wife, Nora Camann, for their encouragement and support.

Thanks go to my committee, Drs. Potter, Rasheed, and Bettinger, for their assistance navigating the requirements for graduation and putting up with my last-minute requests.

I also want to acknowledge all the volunteers who maintain LaTeX and the TeXLive distribution, without which the production of this thesis would have been far more burdensome. In a similar vein I also want to acknowledge Ericsson for open sourcing Erlang and the community that has arisen around it. Erlang made writing the programs discussed in this thesis fun!

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Particle Swarm Optimization, or PSO, belongs to a class of computer programs that could be called nature-inspired heuristic optimization techniques. It is nature-inspired in that it was conceived as a model of animal flocking and herding behavior. Its inventors, James Kennedy and Russell Eberhart, recognized that it was essentially optimizing a function from the space in which the simulated animals moved. It is heuristic because it does not guarantee an optimum solution such as a formal algorithm might, but it does tend to find good solutions in potentially far less time than it would take for an algorithm to find the optimum solution, especially if the function is non-linear and the search space of a high dimensionality. Other nature-inspired heuristic optimization techniques include simulated annealing and a variety of evolution-based methods, of which Genetic Algorithms (GA) are perhaps best known.

As originally conceived, PSO optimizes functions that have as their domains multi-dimensional spaces of real numbers. Variations on PSO have been developed for functions of different domains. Among the most visible of these is Discrete PSO or DPSO, which adapts PSO to optimize functions of discrete values such as bitstrings or integers.

DPSO typically performs best when there are no constraints on the search space, which is to say that the function has a result for all possible points in the search space. Some problems enforce constraints that forbid certain points in the search space. There are a number of ways to deal with these problems, such as disqualifying solutions that violate a constraint, attempting to repair them so that they follow the constraint, or assessing a penalty depending on the severity of the violation. This thesis considers addressing constraints by changing the

Problem	Type	GA	DPSO	RO	EO
Diagnosis	Max	87%	98%	12%	100%
Configuration	Max	99.6%	99.85%	72%	100%
Planning	Min	6,506,676	35M	5,500,391	10M
Pathfinding	Max	95	86	65	74

Table 1.1: Results from [1]

representation in such a way as to reduce the number of constraint violations or allow implicit repair by assigning priorities to alternatives. Specifically, it looks at problems where these goals can be achieved by representing potential solutions as permutations.

1.2 Motivation and Outline

In [1], the authors evaluated the performance and suitability of nature-inspired optimization techniques when applied to problems chosen to exemplify the concerns faced in the areas of diagnosis, configuration, planning, and pathfinding. They found a surprising result in the planning area: DPSO performed very poorly compared to GA, a population-based optimization method that models biological evolution, and two single-element techniques, Raindrop Optimization (RO) and Extremal Optimization (EO). Their final results are reproduced in Table 1.1. For forest planning, the particular planning problem considered, a lower value is better. It is clear that their DPSO ranked last among the techniques in planning, despite comparing well in other areas (all of which were maximization problems).

The authors experimented extensively with DPSO parameters such as swarm size, inertia, and the learning constants, but none of these was sufficient to bring its performance in line with the GA or single-element methods. This suggests that if DPSO is to be improved in this area, it must be through more radical means such as hybridization with other search methods, enhancement with domain knowledge, or a reformulation of the representation. In

this thesis, I describe an attempt at the last of these. Specifically, I propose that particles encode relative priorities for each possible element of a plan rather than directly encoding a plan. This results in a significantly larger search space, but one which I argue PSO can more easily navigate when the problem features constraints.

This thesis begins by describing PSO and DPSO, then the alternative priority representation that allows a PSO to optimize functions of permutations. I then discuss the application of priority representation to the forest planning, n-queens, and snake-in-a-box problems. Forest planning is the problem that inspired the use of priority representation to order plan elements by preference. The n-queens puzzle is included to illustrate how priority representation can be applied to fitness functions that can accept a permutation as input. Snake-in-a-box admits of a novel application where particles represent ordering relations rather than permutations. Following descriptions of the problems, I discuss experimental results and their implications.

Chapter 2

Particle Swarm Optimization

Particle Swarm Optimization is a nature-inspired optimization technique that models the movement of animals in groups such as schools of fish and flocks of birds developed by James Kennedy and Russell Eberhart [2, 3]. PSO maintains a swarm of particles, each of which represents a potential solution to the problem. With each iteration of the PSO, the particles move through the search space influenced both by a memory of their previous best position and the best position attained by any particle in the population.

2.1 Motivation

PSO is inspired by nature. In particular, it is inspired by processes related to the life sciences. It is one of an increasing number of optimization techniques of similar origin, which is unsurprising considering that a great deal of the behavior of living things is optimizing the chances of surviving and reproducing. Kennedy and Eberhart cite research by Doug Hoskins [4] to show that even the simplest living things engage in optimization behavior:

> *E. coli* bacteria show what is perhaps the simplest "intelligence" behavior imaginable. . . . These single-cell organisms are capable of two kinds of locomotion behaviors, called "run" and "tumble." Running is a forward motion implemented by rotating the flagella counterclockwise. Tumbling occurs when the flagella are rotated clockwise. The change causes the bundle to fall apart, so that the next "run" begins in a random direction. The cell tumbles more frequently in the presence of an adverse chemical, and the overall change of direction is enough to

> increase the bacterium's possibility of survival, enabling it to escape from toxins. [2, pages 96–97]

In this quotation, "intelligence" is used to refer to the extent to which a living thing is able to control the distribution of its offspring over time.

Kennedy and Eberhart note that intelligence in more complex animals, as it applies to solving problems, has both a social and a cognitive aspect. An individual in isolation is severely limited in the class of problems he or she can make reasonable attempts at solving. When groups of these animals communicate and work together in groups, the class of addressable problems becomes much larger. This is especially obvious in groups of humans such as the scientific and engineering communities, which by communicating with one another have made much greater strides than if all who work in them performed their research in isolation and did not make use of others' results.

This phenomenon is not unique to humans. Other animals also exhibit the social aspect of intelligence. One class of examples has to do with the self-organized movement of groups of animals in search of food sources and seasonal habitats.

> There are two extreme ways in which groups can "decide" on a direction of movement. Either all individuals within the group contribute to a consensus, or else relatively few individuals (for convenience we will call them "leaders") have information about the group's travel direction and guide the uninformed majority. [5, page 19]

It is this sort of self-organized movement in nature that PSO models, with emphasis on the guidance of leaders. Just as every member of a herd or flock inhabits a place in a literal landscape, each particle in a PSO inhabits a point in a multi-dimensional search space. The particle has a memory of its best position in the past, which forms the cognitive influence on its direction. But the particle is also subject to a social influence from the leaders, those of its peers with the best memory of where to go.

2.2 Structure

Like Genetic Algorithms and other evolutionary computation methods, PSO maintains a population of individuals, or in PSO terminology a swarm of particles. The particles exist in a series of discrete time steps I will call "iterations" following GA terminology. These particles move through the search space influenced by their velocity in the previous iteration, a memory of their best location in the past, and the best location of their leader.

Each PSO particle i is represented as a vector of values $\vec{X}_i(t)$ and a vector of velocities $\vec{V}_i(t)$, each from $\mathbb{R}^d$ (where d is the dimensionality of the search space), at discrete time step t. The goodness or fitness of a particle is given by a problem-specific evaluation function $e(\mathbb{R}^d) \rightarrow \mathbb{R}$. The evaluation function comprises zero or more decoder functions $g_0, g_1, \ldots$ with which a fitness function f is composed:

$$e(\vec{X}_i) = f \circ \cdots \circ g_1 \circ g_0(\vec{X}_i) \tag{2.1}$$

The fitness function takes as its domain a potential solution to the problem, and returns a real number describing the goodness of the solution. In the simplest case, potential solutions take the form of vectors of real numbers, in which case there are no decoder functions so the fitness and evaluation functions are identical. Decoder functions are used when potential solutions do not take the form of vectors of real numbers. For example, the problems described in this thesis expect solutions to take the form of vectors of integers. In these cases, the decoder functions collectively transform $\vec{X}_i$ from a vector of real numbers to whatever form the fitness function requires. Decoders can also act as repair operators for problems with constraints; these decoders might make no change to a solution that does not violate any constraints, but modify solutions that do violate constraints so that they do not. Following the terminology of evolutionary computation (which in turn follows that of evolutionary biology), I call the result of applying all the decoder functions $\cdots \circ g_1 \circ g_0(\vec{X}_i)$ the phenotype[1] of $\vec{X}_i$. This is something of a mixed metaphor in the context of PSO, which models the movement of

[1]Likewise, $\vec{X}_i$ itself might be called the genotype.

groups of animals rather than the evolution of genetic material, but no more appropriate term presents itself.

At each time step, new velocities and values are calculated based on the previous ones as follows:

$$\vec{V}_i(t) = \alpha\vec{V}_i(t-1) + \phi_1(\vec{P}_i(t) - \vec{X}_i(t-1) + \phi_2(\vec{G}(t) - \vec{X}_i(t-1)) \quad (2.2)$$

$$\vec{X}_i(t) = \vec{X}_i(t-1) + \vec{V}_i(t) \quad (2.3)$$

where α is an inertia constant, ϕ_1 a random number drawn from the uniform distribution between 0 and C_1, ϕ_2 a random number drawn from the uniform distribution between 0 and C_2, $\vec{P}_i(t)$ the best location particle i has yet occupied (as measured by the fitness function) by time step t, and $\vec{G}(t)$ the global best location any particle has occupied by time t. C_1 is the cognitive constant that weights the effect of a particle's memory on its movement, and C_2 is the social constant that weights the effect of other particles on a particle's movement.

The effect is as if the particle were anchored by elastic material to its previous best position and the best position of the current leader. The longer the elastic is stretched, the greater the force of the pull. When visualized, the movements of particles resemble a cloud of swarming insects, whence stems the name "particle swarm." If left to its own devices, the particle would oscillate back and forth around its known best locations, overshooting farther each time. To control these wild oscillations, velocities are limited by the constants V_{min} and V_{max}, which are typically defined

$$V_{max} = C_1 + C_2 \quad (2.4)$$

$$V_{min} = 0 - (C_1 + C_2) \quad (2.5)$$

This restricts the oscillations of the particle around its known best locations to within V_{max} in either direction. Unlike GA populations, PSO swarms with normal inertia ($\alpha = 1$) do not tend to converge or collapse around the known best points in the search space unless encouraged to do so by an inertia constant $\alpha < 1.0$ [2, page 313]. If the fitness function

being optimized is such that exploiting good solutions means exploring the area around them within a radius much lower than V_{max}, applying low inertia may improve performance.

2.3 Implementation

The PSO implemented for this thesis begins by initializing $\vec{V}_i(0)$ to random numbers from the uniform distribution between $[V_{min}, V_{max}]$ and $\vec{X}_i(0)$ to random numbers from the uniform distribution between $[0, V_{max}]$ for discrete representation or $[-1, 1]$ for priority representation. These ranges were found to work well in preliminary tests. Once initialized, iteration 1 begins.

For each iteration $t > 0$, the fitness of each particle is found by applying the evaluation function e in Equation 2.1 to the value vector $\vec{X}_i(t-1)$ for each particle i and the results noted. If the particle's fitness exceeds its highest so far, $\vec{P}_i(t)$ is set to $\vec{X}_i(t-1)$. When this is finished, $\vec{G}(t)$ is set to the $\vec{P}_i(t)$ with the highest fitness. Once the best fitnesses have been updated, the velocities of each particle at time t are updated according to Equation 2.2. Any velocities higher than V_{max} are set to V_{max}, and any velocities lower than V_{min} are set to V_{min}. Finally, the value vectors at time t are updated according to Equation 2.3. Assuming the stopping criteria are not met, this is repeated for the next iteration $t+1$. All PSOs in this thesis use $t = 2500$ as a stopping criterion, so none ran for more than 2,500 iterations.

The fitness function is called nt times, where n is the size of the swarm and t the number of iterations. The PSOs in this thesis have swarms of 100, 500, and 1,000 particles, so they call the fitness function up to 250,000, 1,250,000, and 25,00,000 times, respectively. They may make fewer calls if a stopping criterion other than $t = 2500$ is met.

2.4 Discrete PSO

PSO as described above is intended for problems with real-valued solutions, but it is possible to apply it to problems with integer- or bit-valued solutions by discretizing the values before applying the fitness function. This is called Discrete PSO or DPSO. The PSOs tested in [1] are all of this variety.

A discretization function acts as decoder function $g_0(\mathbb{R}^d) \to \mathbb{Z}^d$ with which the fitness function is composed as in Equation 2.1. They can work in several ways. For example, to optimize fitness functions of bit representations, [2] uses the sigmoid function

$$\text{sigmoid}(x_n) = \frac{1}{1 + e^{-x_n}} \tag{2.6}$$

to squash a value x_n from the real numbers to within the interval $(0, 1)$, which is then used as a probability threshold for deciding between 0 or 1.

For the purposes of the DPSOs in this thesis, the fitness functions expect integers from the interval $[0, o-1]$, where o is the problem-specific number of possible values. For example, in the 16-queens problem, $o = 16$. All DPSO evaluation functions in this thesis use the following discretization function:

$$g_0(x_n) = |\lfloor x_n \frac{o-1}{V_{max}} \rfloor \bmod o| \tag{2.7}$$

I call Function 2.7 "staircase wave" because of its graph, shown in Figure 2.1 for o of 8 and V_{max} of 4. It has the following useful properties:

- Every possible value is accessible from every other possible value within one iteration.
- A particle traveling at maximum velocity is almost guaranteed not to arrive at the same value it left[2].

[2]I say "almost guaranteed" because the approximations involved in floating point math allow occasional exceptions, especially with high o.

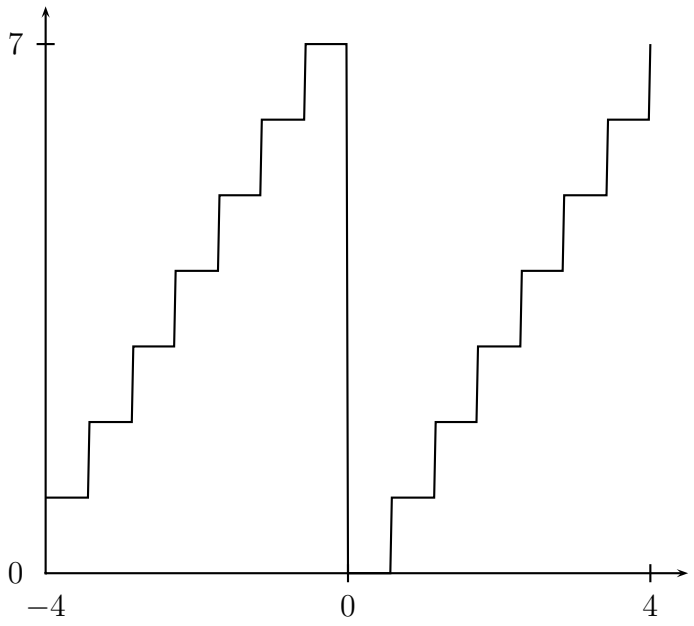

Figure 2.1: The staircase wave discretization function

Chapter 3

Priority Representation

3.1 Motivation

The purpose of priority representation is to allow PSO to encode permutations (or at least relative orderings) in real values. The ability to have particles represent permutations is useful in several situations. Some problems are most naturally expressed as questions of the order in which a series of events must happen, such as the factory scheduling problems in operations research addressed by [6]. Other problems, such as n-queens (described in Section 4.2) have constraints that disallow the same discrete value to occur twice.

Another instance in which permutations are useful is when a problem involves making a series of choices of the same o possibilities, and some choices disallow others. An example of this is the forest planning problem from [11]: when you choose to cut a stand during a particular period, you cannot cut any bordering stands in the same period. A permutation of the Cartesian product of the set of choices to be made (e.g. the stands to cut) and the possibilities for those choices (e.g. the periods) describes the order of preference for assembling a solution that violates no constraints. The application of priority representation to the forest planning problem is described in more detail in Section 4.1.2.

Using otherwise straightforward integer representations for these problems causes complications that must be carefully handled to obtain acceptable performance. When a PSO particle violates one or more constraints, the implementers have several options. One is to disqualify the particle for that iteration, so that its value may not become a personal or global best. Disqualification can be effective when constraint violations are rare, but if a

large portion of the search space violates at least one constraint, performance will suffer from the limited number of good solutions in a swarm.

In search spaces where constraint violations are common, a preferred option is to penalize the fitness of the particle depending on the number and severity of its violations. The fitness function is applied to the particle as if it violated no constraints, but the resulting value is modified by a problem-specific penalty function. This allows the PSO to choose personal and global best positions even if the entire swarm is in violation of at least one constraint, and favors positions with the fewest and least severe constraint violations. Another option is to repair what in evolutionary computation is called the phenotype of the particle—the potential solution to the problem represented by the particle's value vector—so that it does not violate any constraints. This, too, gives the PSO material with which to work before the swarm finds viable solutions.

The task of finding good penalty functions and repair operators can be a difficult one [9]. When they are used, PSO performance hinges on their appropriateness to the particular problem at hand. Inappropriate repair operators will introduce systemic bias in the phenotypes produced that may preclude the PSO from finding optimal solutions. Inappropriate penalty functions may cause the PSO to prefer impossible solutions to good ones that violate no constraints if too lax, or prevent particles from moving between clusters of good solutions (thereby trapping it on a local optimum) if too strict.

A final option is to reframe the problem by reconfiguring the search space so that impossible solutions are never found, or at least reducing the number of violations. This is the route taken in this thesis by priority representation for problems that are amenable to permutation representations.

3.2 Description

When using priority representation with PSO, each position n of the value vector $\vec{X} = x_0, x_1, \ldots$ represents a discrete item to be included in the permutation. Its place in the

Position	0	1	2	3	4	5
Vector	0.3	1.1	−0.7	1.4	−2.0	−0.1
Permutation	4	2	5	0	1	3

Table 3.1: Example of priority representation

permutation is determined by the value held by x_n in relation to those held by elements in other positions. In a typical problem using priority representation, the permutation is obtained by sorting the elements of the vector by their value while retaining their index. Table 3.1 shows an example vector and the permutation it encodes, obtained by sorting the vector in ascending order. This sorting function is typically g_0 in the evaluation function defined in Equation 2.1. If further processing of the permutation is necessary to create the sort of solution expected by the fitness function, additional problem-specific decoder functions $g_1, g_2, \ldots$ may be needed.

One case in which the vector need not be fully sorted is if the decoder functions for the problem do not require a full permutation but only a way to compare the relative priority of different plan elements. For example, in the snake-in-a-box problem described in Section 4.3.2, a snake in an n dimensional hypercube can only expand to one of at most $n-1$ different nodes[1] from any given position. Only the relative priority of those $n-1$ nodes needs to be considered.

Early work in encoding permutations in real-valued vectors was carried out by James Bean [6], who adapted GA to scheduling problems in operations research. He called the encoding "random keys," because the value of each allele served as a key indexing that allele in the permutation.

[1] One of the n is infeasible because it is the dimension the snake crossed from its last node.

3.2.1 Advantages and Disadvantages

The advantages of using priority representation in those problems where it is appropriate are several. One, in the words of [6], is "robustness to problem structure." Priority representation needs little specialization in order to be applied to the problems to which it is appropriate. In some cases, further decoding functions are necessary. This is the case in the forest planning and snake-in-a-box problems discussed here. No further decoding function is needed for the n-queens problem or the scheduling problems that concerned Bean.

Another advantage is that it uses the same method of updating particle vectors as standard PSO. No special operators are required to maintain the consistency of the permutation as in GAs with permutation operators. PSO using priority representation is no more difficult to implement than standard PSO. This is important because ease of implementation is considered one of the attractions of PSO.

Finally, there is the advantage that any permutation representation has over other discrete representations when considering problems with constraints. As mentioned earlier, this includes the generation of phenotypes that can be guaranteed not to violate certain constraints, perhaps even all of them. This is discussed further in the context of the problems in Chapter 4.

There are also some disadvantages to this representation. The search space can be larger. In problems where the constraint that motivates a permutation representation is one of order (as in factory machine scheduling problems) or one that requires that the same value not appear more than once (as in n-queens), this is not the case; the dimensionality of the search space is the same for the natural discrete representation and the priority representation. But for problems that involve making a series of choices such as forest planning and snake-in-a-box, the dimensionality of the search space does tend to increase.

It is less clear how to measure the other component of search space size, the size of each dimension. In a sense they all range over $\mathbb{R}$, but in DPSO they are discretized to o possible

values and in priority representation they are prioritized to one of d possible places in the permutation.

Another disadvantage is the additional computation time it can take to arrange the priority vector into a permutation. In many problems this is done by sorting the vector, which can add considerably to the amount of time it takes to evaluate the fitness of an individual. When sorting is not needed, as in the snake-in-a-box problem (see Section 4.3.2), there must be a decoding function to change the vector of priorities into an object of a sort that the fitness function can accept. Depending on the problem, this process may take longer than any decoding function the DPSO implementation might require.

To account for the likelihood of PSO using priority representation to take longer than DPSO for the same problem, in Section 5.1 I estimate the amount of time it takes both DPSO and PSO with priority representation to evaluate a particle in order to compare the best solutions found by the two after approximately the same amount of computation.

3.3 Alternative Permutation Representations

Priority representation is not the only way to represent permutations with PSO particles. For example, there is no need for the numbers to be real-valued. [10] uses integers to encode the relative positions of items in a permutation to apply GA to the well-known traveling salesperson problem. But there is little reason to attempt this with PSO because the discretization involved in DPSO just adds another step with no obvious gain.

Another approach is that taken by [7, 8], a hybrid of GA and PSO. There is a long history of using GA with permutation representations, and a number of mutation and crossover operations have been developed which explore the search space while maintaining permutation integrity. The hybrid presents particle values as GA-style permutations, but uses the socially- and cognitively-influenced velocity measure of PSO to decide how many times the GA permutation crossover operators are applied. [7] found that this alternative permutation representation performed better than priority representation for the resource-constrained

project scheduling problem in operations research. For the purposes of this thesis I will only evaluate priority representation because it deviates the least from standard PSO, thereby retaining more of the benefit of PSO's simplicity of implementation.

Chapter 4

Problems and Approaches

In this chapter I describe the three problems I chose to compare DPSO and PSO with priority representation.

4.1 Forest Planning

4.1.1 Description

The forest planning problem deals with managing a forested area in such a way as to achieve an even flow of harvested lumber without violating constraints intended to control erosion and maintain aesthetic appearance. The forested area is divided into plots or stands, each of which may border one or more other plots. Time is divided into discrete periods, for each of which we have estimates of how many MBF (thousand board feet, a measure of lumber volume) can be harvested from each plot.

A plan comprises decisions on when and if to cut each plot to harvest its lumber. Each plot may be cut in at most one period. There is a further unit restriction adjacency constraint that no two adjacent plots may be harvested during the same time period. The closer each period's total harvest is to a target harvest constant across periods, the better the plan. Quantitatively, the goodness of a plan is given by the formula:

$$\sum_{i=0}^{o-1}(H_i - T)^2$$

where i is the harvest period, o the number of periods, H_i the total timber harvested in period i (in MBF), and T the target harvest constant across periods (also in MBF). The

Figure 4.1: The Daniel Pickett Forest divided into 73 plots

result represents the accumulated error of the plan. The objective of the problem is to minimize this formula, which in PSO serves as the fitness function.

Following the lead of [1], we use an instance of the forest planning problem given in [11], the 73-plot Daniel Pickett Forest (see Figure 4.1) over 3 time periods covering 10 years each. Typical harvest volumes for a western US forest are assigned to each plot. The target harvest for each period is 34,467 MBF, established by a linear programming solution without the adjacency constraints.

4.1.2 Approach

To test the priority representation for the forest planning problem, I developed a real-valued PSO in Erlang, which I chose for the ease with which programs written in it can be parallelized. In order to compare the priority representation with the integer representation, I developed two decoder functions to convert particle vectors into plans that could be evaluated by the fitness function. The first, DPSO, directly encodes the plan in the particle. The

second, PPSO (for Priority PSO) encodes a permutation describing the priority assigned to harvesting each plot during each period.

DPSO

The most direct approach to applying PSO to the forest planning problem is to encode a plan in a discretized vector $g_0(\vec{X})$. For example, if discretized to integers, each element x_n of $g_0(\vec{X}) = x_0, x_1, x_2, ...$ could indicate the period in which to harvest stand n.

DPSO expects one value per plot, which it discretizes by applying Function 2.7, the staircase wave function, where o is the number of periods. The result of this function is the period during which plot n is to be harvested. This can result in plans that violate the adjacency constraint. Penalties may be applied to the fitnesses of particles that violate the constraint to discourage it without completely eliminating its influence. This was the initial approach taken in [1], but it resulted in a terrible 150M fitness. Later experiments with other parameters and a bitstring representation reduced that to 35M.

For DPSO, no penalty function is used; to ensure adherence to the adjacency constraint a simple repair operator g_1 blocks the harvesting of plots that conflict with it, a strategy used as the basis of the RO technique [11].

PPSO

PPSO treats the particle not as a plan but as a set of priorities for assembling a plan. We use a real-valued PSO where the length of $\vec{X}$ is the product of the number of plots and the number of periods. If p represents the number of plots and o the number of periods, we say that $x_n \in \vec{X}$ is the priority assigned to harvesting plot $\lfloor \frac{n}{p} \rfloor$ in period (n mod o).

To convert a particle into a plan, we use two decoder functions, g_0 and g_1, composed with the fitness function as in Equation 2.1. The first of these to be applied, g_0, sorts the values by priority, thereby forming a permutation of all possible assignments of periods to plots (ignoring for now the possibility of not cutting a plot).

Next, g_1 iterates through the permutation from highest priority to lowest priority, assigning the specified plots to be harvested at the specified periods. Should an assignment be impossible either because the plot has already been assigned to be harvested during another period or it would violate the adjacency constraint, we ignore that element and move on to the next. When we are finished, any plots that have not been assigned a period (because the harvests of its adjacent plots rule them all out) are not harvested.

This representation could be extended to include a priority for not harvesting each plot, but doing so in preliminary trials resulted in very poor performance. The target harvest T is such that nearly every plot must be harvested to achieve it because the goal of even-flow management is an *optimal* constant supply of lumber [11], so the additional dimension needlessly expands the search space.

An advantage of this priority representation is that impossible plans cannot be generated by the decoder function, so there is no need for a penalty function or repair operation to account for constraint violations. In the case of the priority representation, the decoder function g_1 essentially repairs plans based on the particle's priorities, which avoids the systemic bias that can be introduced by traditional repair operations. For example, the DPSO repair operator for this problem described other is likely biased in favor of harvesting lower-valued plots over higher-valued ones because fewer harvests have already been decided when they are being checked for adjacency violations. By assigning priorities to period-plot pairs, the only bias of PPSO is in favor of harvesting plots assigned higher priorities.

Trials

In the first set of trials, we used population sizes of 100, 500, and 1000, and the following parameters:

$$\begin{aligned} C_1 &= 2 \\ C_2 &= 2 \\ V_{max} &= 4 \end{aligned}$$

$$V_{min} = -4$$
$$\alpha = 1.0$$

We ran a set of 10 trials of 2,500 iterations each for every population size. Different initial random seeds were used for each trial in a set.

We used accepted values for the learning constants and velocity limits because we did not have an *a priori* reason to weigh the social factor more heavily than the cognitive factor or vice versa. In the case of PPSO, we noticed that individual values of the particles were straying far from the origin. It was not unusual for a particle to have multiple priorities with absolute values exceeding $4V_{max}$. In an attempt to control this, we ran another set of trials with the same parameters except $\alpha = 0.8$.

4.2 n-Queens

4.2.1 Description

The n-queens puzzle is a classic, well-understood problem often used to test students' skills in algorithm design. The goal is to arrange n queens on a chessboard (i.e. a grid of squares) of dimensions $n \times n$ in such a way that no two queens can attack one another. This puzzle is an example of a class of problems called Constraint Satisfaction Problems, commonly abbreviated to CSP, in which one is concerned solely with satisfying constraints.

In the game of chess, there are six kinds of pieces, each of which has its own method of moving and capturing other pieces [12]. The queen, the most flexible piece, can move across the board any number of squares horizontally, vertically, or diagonally. Any square to which a piece such as the queen can move, it is said to attack. Therefore, in a full solution to the n-queens puzzle no two queens can share a row, column, or diagonal. Figure 4.2 shows the squares attacked by a queen placed near the center of a standard 8×8 chessboard.

The rook is another powerful piece. Like the queen, it can move any number of squares vertically or horizontally, but that is the extent of its abilities. Unlike the queen, it cannot

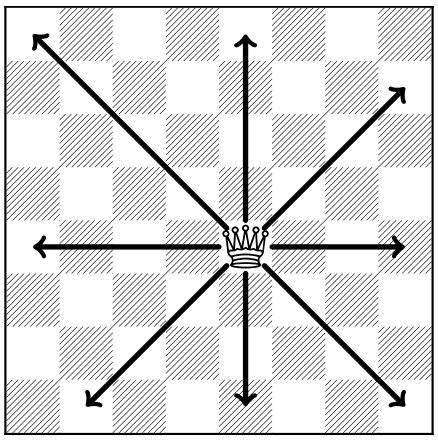

Figure 4.2: Movement of the queen

move diagonally. Figure 4.3 shows the squares attacked by a rook placed near the center of a standard 8×8 chessboard. Any solution to the n-queens puzzle is also a solution to the n-rooks puzzle, by which I mean a problem the goal of which is to place n rooks on an $n \times n$ chessboard so that no two can attack one another. The converse does not hold; there are solutions to the n-rooks puzzle where two or more rooks share a diagonal, so they are not also solutions to the n-queens puzzle.

The n-queens puzzle was chosen to compare DPSO and PPSO because, unlike forest planning, there is no need for a further decoding function g_1 to transform the permutation into a potential solution on which the fitness function can act. Instead, the problem is such that a permutation is a natural representation of the sort of solution the fitness function expects.

The n-queens puzzle is well-researched, so it is included here only to compare discrete and priority representations in a case where a permutation is the natural way to express a solution. There already exist heuristics specific to the puzzle (such those in [13]) that would find a full solution sooner than would PSO unenhanced by domain knowledge.

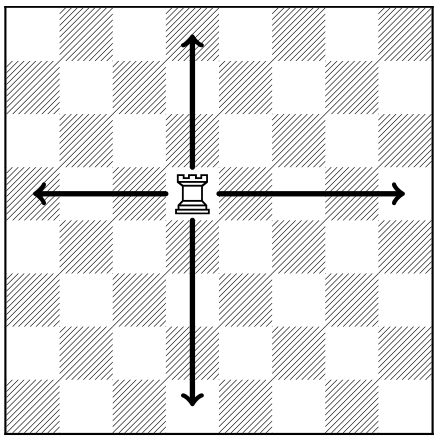

Figure 4.3: Movement of the rook

4.2.2 Approach

The PSO developed for the forest planning problem is also used for the n-queens puzzle. All trials were carried out using 16 queens on a board of dimensions 16×16. The number 16 was arbitrarily chosen as being high enough that one could expect a PSO to find full solutions but not so high that it would be guaranteed to find one.

The fitness function used by both DPSO and PPSO accepts a vector $\vec{X}$ of n integers $x_0, x_1, \ldots, x_{n-1}$ from the interval $[0, n-1]$. The fitness function interprets x_i to be the row on which the queen in column i is to be placed. We can assume there is only one queen on each column because if there were two, they would attack one another. Before the fitness function is applied to a vector, a simple repair operator removes illegal queen placements, leaving that element of the vector blank. The return value of the fitness function is the number of queens placed on the board. A full solution to the problem places all n queens.

DPSO

The DPSO implementation is straightforward. The value vector $\vec{X}$ of each particle is plugged directly into the fitness function after undergoing repair. Repair occurs from left-to-right, which is to say from column 0 to column $n-1$. If the placement at column i conflicts with a queen placed in a column $j < i$, no queen is placed in column i.

DPSO particles are guaranteed by their representation not to place two queens on the same column. They may still violate the constraint of placing two queens on the same diagonal, or placing two queens on the same row.

PPSO

The PPSO implementation is slightly less straightforward. For each i such that $0 \leq i \leq n-1$, x_i of the value vector $\vec{X}$ represents the "priority" (a term that fits less well here than in forest planning or snake-in-a-box) of placing the queen in row i. The permutation $g_0(\vec{X})$ that results in sorting the indices by their values is treated exactly the same as the value vector of DPSO.

PPSO particles are guaranteed by their representation not to place two queens on the same column or on the same row, so every PPSO particle represents a solution to the n-rooks puzzle. It is still possible for them to violate the constraint forbidding queens from being placed on the same diagonal, so a repair operator g_1 is applied to the permutation. It behaves the same as in DPSO above, starting at column 0 and removing queens that attack any other queens from columns to the left.

Trials

The same values are used for the n-queens puzzle as were used for forest planning:

$$C_1 = 2$$

$$C_2 = 2$$

$$\begin{aligned} V_{max} &= 4 \\ V_{min} &= -4 \\ \alpha &\in \{1.0, 0.8\} \end{aligned}$$

10 trials of 2,500 iterations each were run for each population size (100, 500, and 1,000) and inertia parameter. Each trial has an additional stopping criterion of finding an optimal solution, i.e. one that places all 16 queens on the board with no queen attacking any other.

4.3 Snake-in-a-Box

4.3.1 Description

Snake-in-a-box (henceforth SIB) is a problem in graph theory first described in [14]. A snake in dimension n is described as a path through an n-dimensional hypercube such that each node on the path is adjacent only to the node preceding it and the one following it. In graph-theoretic terms, it is a chordless path or induced path. It has applications in "electrical engineering, coding theory, analog-to-digital conversion, disjunctive normal form simplification, electronic combination locking, and computer network topologies." [15]

A graph is a set of vertices (vertex in the singular; also called nodes), some of which may be connected in pairs by edges. A hypercube of dimension 0 is a graph consisting of a single vertex and no edges. Hypercubes of dimension $n > 0$ are constructed by taking two hypercubes of dimension $n - 1$ and connecting each vertex in one $(n - 1)$-dimensional hypercube to its twin in the other. In equivalent graph-theoretic terms, a hypercube of dimension $n > 0$ is the Cartesian product of a hypercube of dimension $n - 1$ and the complete graph on 2 vertices.

Hypercubes of dimension 1 are lines, of dimension 2 squares, of dimension 3 cubes, and of dimension 4 tesseracts. The first 3 of these are shown in Figure 4.4.

Nodes in n-dimensional hypercubes can be labeled with bitstrings of length n in a method following [16]. When labeled in this way, two vertices are adjacent—that is, joined by an

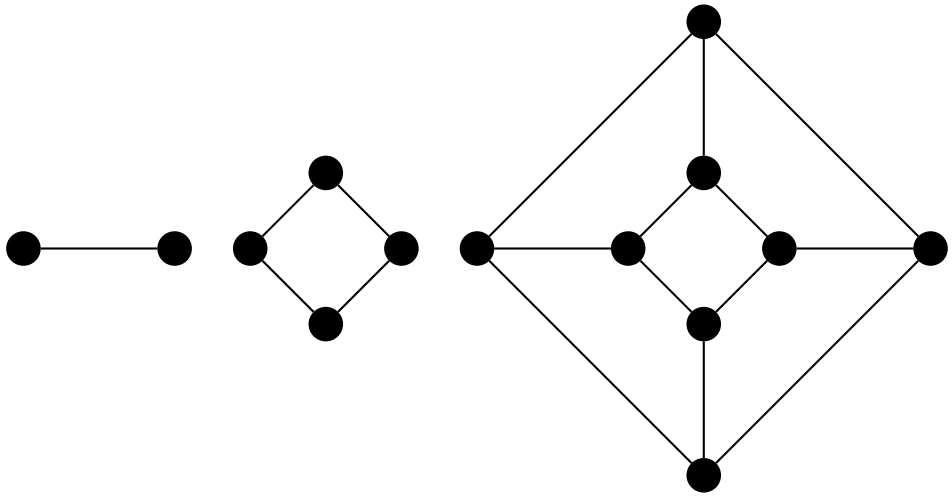

Figure 4.4: Hypercubes in dimensions 1–3

edge—if and only if the bitstrings representing them differ in exactly one bit. So, for example, in a 4-dimensional hypercube, 0000 is adjacent to 0100, but not to 0101 because they differ in two bits.

A snake is considered maximal when it cannot grow any longer. This means that the vertex at either end of the snake is only adjacent to vertices that are also adjacent to some other part of the snake. The goal of SIB is to find the longest maximal snake in an n-dimensional hypercube. This is a slightly problematic goal in that there is currently no known feasible way to discern whether a snake is the longest in any dimension $n > 7$ [17]. A verifiable goal is to find a longer snake in a dimension n than has ever been found before. The longer the snake, the more useful it is in the various applications listed above.

Hypercubes of dimension n exhibit $n!$ symmetries that leave the origin fixed. There are therefore a large number of snakes of each length that are members of the same equivalence class. It is possible to reduce the search space by only looking for canonical snakes, which start at 0 (the origin) and follow this rule from [18]:

> In order to explore exactly one path in each equivalence class, the search algorithm considers only paths which are *canonical* in the sense that the first occurrence of a 1 in each position follows the linear order of least significant to most significant. Following this rule for n = 7, the next node after 0000000 must be 0000001, rather than 1000000 or any of the other five nodes adjacent to the origin. Later on, say when 1's have appeared in each of the first four positions but not in any of the last three, the next position to contain a 1 for the first time must be the fifth.

Therefore, in all dimensions $n \geq 3$, the first nodes of a canonical snake must be the ones corresponding to the bitstring representations of the decimal integers 0, 1, 3, and 7.

SIB is the subject of ongoing research. As mentioned above, the longest maximal snakes are unknown for dimensions $n > 7$, and the search space is so large that exhaustive search is infeasible. Nature-inspired optimization methods, especially GA, feature prominently in the search for longer snakes. The current longest known snakes in dimensions up to 12 can be found at [17].

4.3.2 Approach

The natural fitness function for SIB is obvious: the length of the snake. The ongoing research on SIB has identified several other factors that might indicate the fitness of a snake, such as the size of its "skin," the set of vertices that cannot be visited as the snake continues to grow because of their adjacency to vertices the snake has already visited. The smaller the skin set, the tighter the coils of the snake and the more potential it might have to extend its length. But since the emphasis of this paper is on the comparison of integral and priority representations, only the obvious fitness function will be used to avoid complicating the matter.

In order to count as snakes, lists of nodes or vertices must satisfy the constraints of adjacency and chordlessness. The adjacency constraint requires each vertex of the snake to

be adjacent to the vertices immediately before and after it (unless it is the first or last vertex in the snake, in which case it need only be adjacent to node before it *or* the node after it). The chordlessness constraint requires each vertex of the snake to not be adjacent to any vertex in the snake except for the one before it and the one after it, with exceptions for the first and last nodes in a snake similar to those for adjacency.

I include SIB as a problem to compare DPSO and PPSO for two reasons. First, because it provides another way to apply priority representation to a problem the fitness function of which does not accept a permutation (unlike n-queens). Second, because DPSO is able to compete on more even ground in SIB than in forest planning. In forest planning, the best repair operators or penalty functions are not known, and the known ones fail to allow DPSO to cope with its constraints. In SIB, there is a known representation for DPSO—transition sequence—that handles the adjacency constraint as well as PPSO, but both must still contend with the chordlessness constraint.

DPSO

A naïve implementation of DPSO for SIB might encode a vertex path in its value vectors, but this would force the swarm to spend most of its time trying to resolve the adjacency constraint. A better representation is a snake's transition sequence, where we assume the snake starts at the origin and each vector element describes which edge we take to leave the vertex.

For SIB in dimension n, $\vec{X}$ is discretized using the staircase wave function (Function 2.7) with $o = n$ as decoder function g_0, so that all values become integers from the interval $[0, n-1]$. The snake described by $\vec{X}$ begins like all canonical snakes, so we say

$$\begin{aligned} s_0 &= 0 \\ s_1 &= 1 \\ s_2 &= 5 \\ s_3 &= 7 \end{aligned}$$

where s_i is the (decimal) representation of the ith vertex of the snake. x_i of $g_0(\vec{X})$ represents the bit of s_{i+3} flipped in order to advance to node s_{i+4}.

This guarantees that all particles will represent snakes that obey the adjacency constraint, but they must still face the chordlessness constraint. As in the previous two problems, this constraint is addressed using a simple repair operation. In this case, if flipping bit x_i of s_{i+3} would result in a violation of the chordlessness constraint, the decoder function g_1 ends the snake at s_{i+3}.

This brutal enforcement of the chordlessness constraint puts the DPSO at a disadvantage to the PPSO decoder function, which continues to extend the snake until it becomes maximal. In order to even the ground, a growth operator g_2 is applied to the result of the g_1 once it stops extending the snake. The growth operator continues extending the snake until maximal by always choosing to flip the lowest bit that does not result in a constraint violation.

In preliminary trials, g_1 also enforced a canonicity constraint, which required the snake to flip the lowest bit it had not already flipped when moving into a higher dimension. This ensured that all generated snakes were canonical, but even with the growth operator it tended to result in stagnation at low lengths. As a result, in the final trials described below canonicity was not enforced on DPSO snakes.

Another option explored during preliminary trials but discarded was to allow each DPSO to encode multiple snakes. x_i therefore encoded the bit flipped at node $i + 3$ of one snake, but also $i + 4$ of another, $i + 5$ of another, and so on. This resulted in a fitness function several orders of magnitude slower than the one for PPSO, so it was abandoned for the sake of comparison.

PPSO

In forest planning, PPSO is best applied by assigning a priority to every possible decision point/choice (stand/period) pair. While it would be possible to do something similar with

SIB, the search space would be of high dimensionality: $(l-3)n$ where l is the longest snake that could be created and n the dimensionality of the hypercube. To attempt to find a snake of length 99 (which would become the longest known) in dimension 8 would require particle vectors to be of length 768.

Another way to apply priority representation would be to assign each node in the hypercube (except those ruled out by the 4 initial nodes forced by the canonicity constraint) a priority, thereby creating a permutation of the nodes expressing the relative preference the particle has for extending a snake into that node. This is the method I chose for PPSO for SIB.

The length of the PPSO particle vector $\vec{X}$ for dimension $n > 3$ is $2^n - 3n + 2$. This accounts for the 7 nodes in the first three dimensions ruled out by the canonicity constraint, plus the additional 3 nodes ruled out in each dimension above 3. There is no need to assign a priority to those nodes because canonical snakes cannot visit them.

This priority representation is unique compared to n-queens and forest planning in that there is no need to sort $\vec{X}$ before running the decoder function that converts the permutation into something to which the fitness function can be applied. Instead, the values of $\vec{X}$ are loaded into an array indexed by the integer representation of the bitstring of each vertex in the hypercube (making the appropriate adjustments to account for the nodes excluded by the initial forced moves).

Each time we extend the snake, we do so by consulting the array for the priority of each adjacent vertex. We then choose the one granted the highest priority by the array that does not violate the canonicity or chordlessness constraints. Because of this, SIB PPSO uses priority representation less as a permutation than as an ordering relation of nodes expressing the preference of the snake for extending into them. Every path produced in this way is canonical, chordless, and maximal.

Trials

All SIB trials were carried out on the 8-dimensional hypercube. The usual 10 trials of 2,500 iterations were carried out on population sizes of 100, 500, and 1,000 and these parameters:

$$\begin{aligned} C_1 &= 2 \\ C_2 &\in \{2, 0.5\} \\ V_{max} &= C_1 + C_2 \\ V_{min} &= 0 - V_{max} \\ \alpha &= 0.8 \end{aligned}$$

I set C_2, the social influence constant, to 0.5 for half the runs following the advice of [1, page 16]:

> We wanted the particle to be influenced by its own personal history more so than the global best because the global best may be a snake that is counterproductive to the individual being evaluated. There are many different, good paths through an eight dimensional hypercube and the global best may be a completely different path from our other individuals. Furthermore if the other individuals listen too closely to the global best, they could be influenced towards snake violations (with respect to where the individual currently is) that would worsen the individual to a point where it would not be able to recover.

Chapter 5

Results and Conclusions

5.1 Approximate Evaluation Time

It is not completely fair to compare DPSO and PPSO by the results they reach after 2,500 iterations and the same population size because of the nature of priority representation. It often involves sorting the value vector or undertaking further decoding work unnecessary in DPSO, so it takes longer to evaluate the fitness of the particle, and therefore longer for the algorithm to finish. PPSO also tends to use more computing resources to calculate particle movement than DPSO because of the higher dimensionality of the search space.

In an attempt to address this disparity, I ran trials of DPSO and PPSO for each problem with a population of 1000 for 2,500 iterations and typical parameters, with all parallelism features disabled. Where $time_{dpso}$ is the time taken for DPSO and $time_{ppso}$ the time taken for PPSO, I found the iteration of PPSO by which point the PPSO had consumed CPU resources approximately equal to the DPSO at 2,500 iterations by applying the formula

$$2500 \cdot \frac{time_{dpso}}{time_{ppso}} \tag{5.1}$$

The results are tabulated in Table 5.1.

In each of the problem sections below, the best, worst, and average results and their standard deviation are given for PPSO both at iteration 2,500 and at the CPU-equivalent iteration given in Table 5.1. The latter are given in the tables labeled "Adjusted PPSO."

All trials were run on the same computer, a Velocity Micro E2250 with an Intel Core 2 Quad Q9450, with four cores running at 2.66GHz and 12MB L2 cache. The software used

Problem	Iteration
Forest Planning	1,059
n-Queens	2,292
Snake-in-a-Box	1,743

Table 5.1: PPSO iteration CPU-equivalent to iteration 2,500 of DPSO

was Erlang R14B on 64-bit Ubuntu 10.10. It took 33 hours of wall-clock time to complete all 360 trials.

5.2 Forest Planning

Results from all parameter settings are tabulated in Table 5.2. The column labeled Best contains the fitness of the best particle from each set of 10 trials. The one labeled Worst contains the fitness of the worst particle from each set. Mean contains the arithmetic mean of the fitnesses of the single best particles from each of the trials, and SD the standard deviation. Forest planning is a minimization problem, so the lower the fitness, the better the particle.

As expected from the work of [1], the DPSO configured in this way performed poorly. The PPSO performed very well, with best fitness in its peak case ($\alpha = 0.8$ and population of 500) exceeding the best performance of the RO in Table 1.1.

The reason for the disparity of performance between PPSO and DPSO appears to have to do with adjacency constraints. Table 5.3 compares the plans of the best particle from all 60 DPSO trials to the best of all 60 PPSO trials. Numerals indicate the period in which a plot is to be cut, while Xs indicate a plot is not to be cut. Because of the nature of our representations, Xs should only occur when constraints forbid the planned harvest. The DPSO particle has 10 uncut plots, while the PPSO particle has only 3. Because nearly every plot must be cut in order to meet the target harvest each period, the DPSO's way of handling

α	Pop. Size	Best	Worst	Mean	SD
		Adjusted PPSO			
1.0	100	5,717,387	23M	11M	4,953,428
1.0	500	7,080,657	12M	9,706,199	1,832,606
1.0	1000	5,821,866	23M	11M	4,703,921
0.8	100	7,445,002	18M	12M	3,741,497
0.8	500	5,554,654	14M	9,089,556	2,542,807
0.8	1000	6,543,300	15M	9,219,950	2,568,628
		PPSO			
1.0	100	5,717,387	17M	10M	3,209,376
1.0	500	6,481,785	12M	9,565,147	1,901,665
1.0	1000	5,821,866	23M	11M	4,714,044
0.8	100	7,445,002	18M	12M	3,583,356
0.8	500	5,500,330	14M	9,049,624	2,519,739
0.8	1000	6,543,300	15M	9,177,784	2,545,059
		DPSO			
1.0	100	118M	170M	145M	18M
1.0	500	114M	152M	135M	9,897,959
1.0	1000	69M	142M	112M	21M
0.8	100	47M	115M	79M	20M
0.8	500	41M	103M	67M	15M
0.8	1000	46M	78M	65M	8,145,207

Table 5.2: Forest planning results

DPSO

Plot	0	1	2	3	4	5	6	7	8	9
00	1	0	1	0	0	X	1	2	1	1
10	0	1	1	1	2	1	1	X	1	X
20	1	2	2	0	1	1	2	1	1	2
30	0	X	1	2	0	2	0	2	0	0
40	X	X	X	0	0	0	1	0	2	0
50	X	2	2	0	1	1	1	0	2	2
60	1	X	2	X	2	2	2	0	1	1
70	0	0	0							

PPSO

Plot	0	1	2	3	4	5	6	7	8	9
00	0	1	0	1	1	2	1	1	2	2
10	0	0	1	1	0	2	X	2	X	1
20	1	2	0	2	1	1	2	1	0	2
30	2	2	1	2	2	1	0	0	0	1
40	X	2	1	0	0	0	1	1	1	1
50	2	2	1	2	1	2	0	1	2	1
60	1	0	2	2	2	2	2	1	2	2
70	0	0	1							

Table 5.3: Best forest management plans

adjacency constraints puts it at a disadvantage. It may be possible to improve the DPSO's performance by finding a more flexible repair operator or penalty function that allows it to cope better with the constraints, but to do so would likely require the implementers to acquire and encode domain knowledge in a way that the PPSO does not.

Curiously, PPSO tended to perform best with a swarm size of 500, whereas Genetic Algorithms tend to improve in performance with larger population sizes. One possible explanation for this is our use of the global best particle for computing the social factor that influences the movement of each particle. A large swarm might prematurely uncover a local optimum and influence all other particles to converge on it when they should still be exploring. Some PSO implementations use the best in the neighborhood of each particle, where the neighborhood is a set of particles smaller than the whole swarm [2], which could improve the performance of large populations.

Even taking into account the increased amount of time it takes the PPSO to evaluate an individual by using the iteration CPU-equivalent to iteration 2,500 of the DPSO, PPSO does very well in this problem. The sacrifices in increased search space dimensionality and evaluation time are well worth the increased performance.

α	Pop. Size	Best	Worst	Mean	SD
		Adjusted PPSO			
1.0	100	16	15	15.1	0.3
1.0	500	16	15	15.6	0.48
1.0	1000	16	16	16	0
0.8	100	16	15	15.4	0.48
0.8	500	16	15	15.8	0.4
0.8	1000	16	16	16	0
		PPSO			
1.0	100	16	15	15.1	0.3
1.0	500	16	15	15.6	0.48
1.0	1000	16	16	16	0
0.8	100	16	15	15.4	0.48
0.8	500	16	15	15.9	0.29
0.8	1000	16	16	16	0
		DPSO			
1.0	100	14	13	13.1	0.3
1.0	500	14	13	13.3	0.45
1.0	1000	14	13	13.5	0.5
0.8	100	15	13	13.7	0.64
0.8	500	15	13	13.8	0.6
0.8	1000	15	13	14	0.44

Table 5.4: 16-queens results

5.3 n-Queens

Results from all parameter sets are shown in Table 5.4. The structure is the same as in forest planning, above. n-queens is a maximization problem, so particles with higher fitness are better. A full solution to the 16-queens puzzle places all queens, so the highest possible fitness is 16.

PPSO performed well again. It found an optimal solution in every parameter set, and did so before iteration 2,292, by which time it had consumed approximately as much CPU time as had DPSO at iteration 2,500. If we consider a trial successful if it finds a full placement of all 16 queens before its final iteration, we can define the Success Rate (SR) of each parameter

α	Pop. Size	SR	Adj. SR	Avg. Time
1.0	100	10%	10%	1,771
1.0	500	60%	60%	856
1.0	1000	100%	100%	737
0.8	100	40%	40%	1,302
0.8	500	90%	80%	1,121
0.8	1000	100%	100%	618

Table 5.5: 16-queens PPSO success rates

set as the proportion of trials that succeeded. Likewise, for PPSO we can define an Adjusted SR in terms of what proportion of trials succeeded before iteration 2,292. These rates are shown in Table 5.5. The column labeled "Avg. Time" is the arithmetic mean of the iteration in which a full solution was found. The table only includes results from PPSO, since DPSO never found a complete solution.

Again, one can likely trace the cause of the performance disparity back to constraints enforced by the problem. The DPSO's particles had to contend with both diagonal and row conflicts, while PPSO only had to contend with the diagonal constraint.

5.4 Snake-in-a-Box

The results from SIB are tabulated in Table 5.6. The columns are the same as in Tables 5.2 and 5.4. Snake-in-a-box is a maximization problem, so the higher the fitness, the better the particle.

PPSO fared well again here, but the gap between PPSO and DPSO is not nearly as dramatic as in forest planning or n-queens. This may be because unlike in previous problems, both DPSO and PPSO were able to guarantee the same constraints were satisfied except for canonicity, which was relaxed for DPSO. Much of the point of only considering canonical

C_2	Pop. Size	Best	Worst	Mean	SD
		Adjusted PPSO			
2.0	100	81	75	78	1.54
2.0	500	83	77	79.09	1.81
2.0	1000	82	76	79.09	1.86
0.5	100	78	75	76.8	1.07
0.5	500	81	77	79.2	1.46
0.5	1000	81	78	80.2	0.97
		PPSO			
2.0	100	81	75	78.09	1.51
2.0	500	83	77	79.7	1.55
2.0	1000	84	78	80.9	1.75
0.5	100	80	75	77.5	1.28
0.5	500	82	78	80.3	1.26
0.5	1000	82	78	80.3	1.09
		DPSO			
2.0	100	78	75	76.59	0.91
2.0	500	79	76	77.7	0.89
2.0	1000	80	77	78.4	1.01
0.5	100	78	75	76.8	0.97
0.5	500	81	76	77.8	1.24
0.5	1000	80	77	78.9	0.94

Table 5.6: Snake-in-a-box results

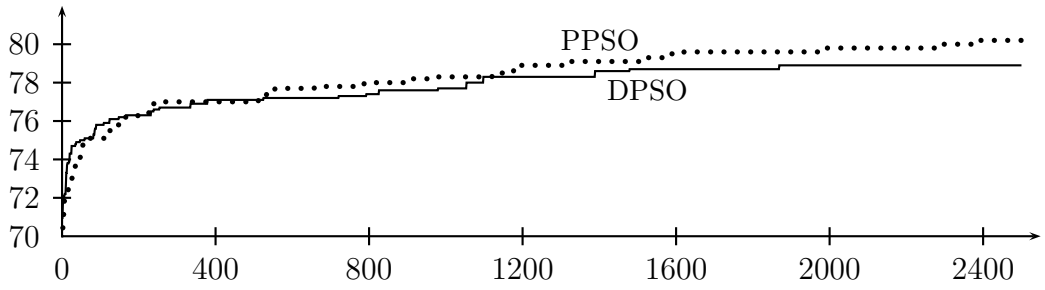

Figure 5.1: Comparison of mean adjusted fitness for SIB

snakes is to reduce the size of the search space, which is pointless if doing so actually reduces optimization performance.

It is difficult to tell from this data whether the reduced social constant $C_2 = 0.5$ improved the performance of either DPSO or PPSO compared to its normal value of 2. In some cases the quality of results did improve, but in others they fell.

Figure 5.1 compares the average best fitness (arithmetic mean of the fitness of the best individual from each trial) of PPSO and DPSO with the parameter set using which DPSO performed best on average: $C_2 = 0.5$ and population size of 1,000. The x axis represents iterations of DPSO and the y axis mean best fitness. For fair comparison, the PPSO iterations are scaled by $\frac{1743}{2500}$, the approximate ratio of how long it takes PPSO to complete an iteration compared to DPSO. Therefore the rightmost sample is of iteration 1,743 of PPSO but 2,500 of DPSO. As you can see, PPSO and DPSO kept close to one another through most of the trial, with PPSO ahead slightly by the end.

Neither PPSO nor this DPSO performed as well as DPSO did in [1] (see Table 1.1), but this is not surprising considering the performance-enhancing measures used in those tests. For instance, the populations were seeded by long trees from lower dimensions, and the growth operator was more invasive than the one used in this DPSO. Instead of simply

extending the phenotype while leaving the value vector $\vec{X}$ unchanged, it actually changed the values to those that would produce the extended snake.

5.5 Conclusions

In summary, PSO with priority representation copes well with constraints of the sort found in the forest planning problem and n-queens puzzle, while a straightforward DPSO using the natural integer representation of a plan does not. There is a price to pay in terms of time performance, since the priority representation requires the extra work of sorting the priorities before a plan can be formed, but at least in the case of the forest planning and n-queens, the greatly improved results are worth the sacrifice.

It is harder to say whether the added complexity of PPSO is worth its benefits in the case of SIB. The slight edge it appears to maintain in the results in Table 5.6 may be as much due to the particular repair and growth mechanisms used in the DPSO implementation as to any advantage PPSO has in finding good solutions. The fact that PPSO performance stuck close to that of DPSO in these trials suggests that the priority representation might be just as useful in the search for larger snakes as DPSO, although there is a potential problem in higher dimensions as the vector length for PPSO roughly doubles with each new dimension.

5.6 Future Work

There remains a future avenue of research in finding an improved repair operator, penalty function, or representation for DPSO that can better handle forest planning adjacency constraints.

As mentioned before, [7] describes a hybrid of PSO with the permutation representation and crossover operators of Genetic Algorithms. It is a more radical departure from standard PSO, but their results show that it performs better for some scheduling problems in operations research. Their GA/PSO hybrid might be worth exploring for the forest planning and snake-in-a-box problems.

It appears that prioritizing each plan element of the forest planning problem is a superior representation to a vector of integers indicating during which period to harvest each stand in the context of PSO. It stands to reason that the same may hold for Genetic Algorithms as well. GA already scored well for the forest planning problem in [1], but it may be worthwhile to try to improve on their results by applying GA with permutation representation.

Another possible permutation representation that could be used for snake-in-a-box in n dimensions is DPSO where each discretized value $d_0(x_i)$ where $x_i \in \vec{X}$ is used to index an array of all possible permutations of the integers $[0, n-1]$. The indexed permutation indicates the order of priority of flipping each bit at the $(i+4)$th vertex to expand the snake. This becomes infeasible as n exceeds 10 or so because the size of the array $(n!)$ becomes prohibitive, but may provide interesting results for lower dimensions.

Finally, PPSO shows some potential in finding new longest maximal snakes. If the SIB PPSO in this thesis were enhanced with domain knowledge using PSO analogs of the GA techniques in [1] and other current research on snake hunting with nature-inspired optimization techniques, it might be able to compete with GA in this field.

Bibliography

[1] Potter, W. D., Drucker, E., et al: Diagnosis, configuration, planning, and pathfinding: Experiments in nature-inspired optimization. *Natural Intelligence for Scheduling, Planning, and Packing Problems*, 267–294. Springer, Berlin (2009)

[2] Kennedy, J., Eberhart, R.: *Swarm Intelligence*. Morgan Kaufmann, San Francisco (2001)

[3] Kennedy, J., Eberhart, R.: Particle swarm optimization. *Proc. IEEE Intern. Conf. on Neural Networks* IEEE Service Center, New Jersey, 1942–1948 (1995)

[4] Hoskins, D.: An iterated function systems approach to emergence. *Evolutionary Programming IV: Proc. of the Fourth Ann. Conf. on Evolutionary Programming*, 673–692. MIT Press, Cambridge (1995)

[5] Beekman, M., Sword, G. A., Simpson, S. J.: Biological foundations of swarm intelligence. *Swarm Intelligence: Introduction and Applications*, 3–41. Springer, Berlin (2008)

[6] Bean, J. C.: Genetic Algorithms and Random Keys for Sequencing and Optimization. *ORSA J. on Comp.* 6, 154–160 (1994)

[7] Zhang, H., Li, X., et al: Particle swarm optimization-based schemes for resource-constrained project scheduling. *Automation in Construction* 14, 393–404 (2005)

[8] Zhang, H., Li, H., Tam, C.M.: Permutation-based particle swarm optimization for resource-constrained project scheduling. *J. of Comp. in Civil Eng.* 20, 141–149 (2006)

[9] Michaelwicz, Z., Schoenauer, M.: Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Comp.* 4, 1–32 (1996)

[10] Greffenstete, J. J., Gopal, R., et al: Genetic algorithm for the TSP. *Proc. of the 1st Int. Conf. on Genetic Algorithms and Their Applications*, 160–168. Lawrence Erlbaum, New Jersey (1985)

[11] Bettinger, P., Zhu, J.: A new heuristic for solving spatially constrained forest planning problems based on mitigation of infeasibilities radiating outward from a forced choice. *Silva Fennica* 40, 315–33 (2006)

[12] Schiller, E.: *The Official Rules of Chess: Professional, Scholastic, and Internet Chess Rules*, 2nd ed. Cardoza, New York (2003)

[13] Stone, H. S., Stone, J. M.: Efficient search techniques—An empirical study of the N-Queens Problem. *IBM J. of Research and Development* 31, 464–474 (1987)

[14] Kautz, W.H.: Unit-distance error-checking codes. *IRE Trans. on Electronic Comp.* 7, 179–180 (1958)

[15] Krafka, K.J., Potter, W.D., Horton, T.R.: The snake-in-the-box problem. *ACMSE '10*. ACM, MS (2010)

[16] Harary, F., Hayes, J. P., Wu, H. J.: A survey of the theory of hypercube graphs. *Comp. Math. Applic.* 15, 277–289 (1988)

[17] Inst. for Artificial Intelligence, U. of Georgia: *Records: Latest Records for the Snake-in-a-Box Problem (August, 2010)*. http://www.ai.uga.edu/sib/records/. Retrieved November 2, 2010

[18] Kochut, K.: Snake-in-the-box codes for dimension 7. *J. of Combinatorial Math. and Combinatorial Comp.* 20, 175–185 (1996)