Lab 4:
Evolutionary Computation and Swarm Intelligence

Introduction

Goals

The goal of this assignment is to learn about two evolutionary computing methods: genetic algorithms, and particle swarm optimization. These are both population methods, that work by using differently sized populations of solutions to explore large parts of the search space in parallel. Although they can be used to solve similar or identical problems, the approach of each is quite different. You will use both GA and PSO to solve several optimization problems that are challenging for gradient descent methods.

Preparations

Read through these instructions carefully in order to understand what is expected of you. Read the relevant sections in the course book.

Report

Hand in a report according to the instructions in *General information for the lab course*. The report should contain answers to all questions, and all requested plots.

Files

You will need the files `gasolver.zip` and `psopt.zip` found on the course web page.

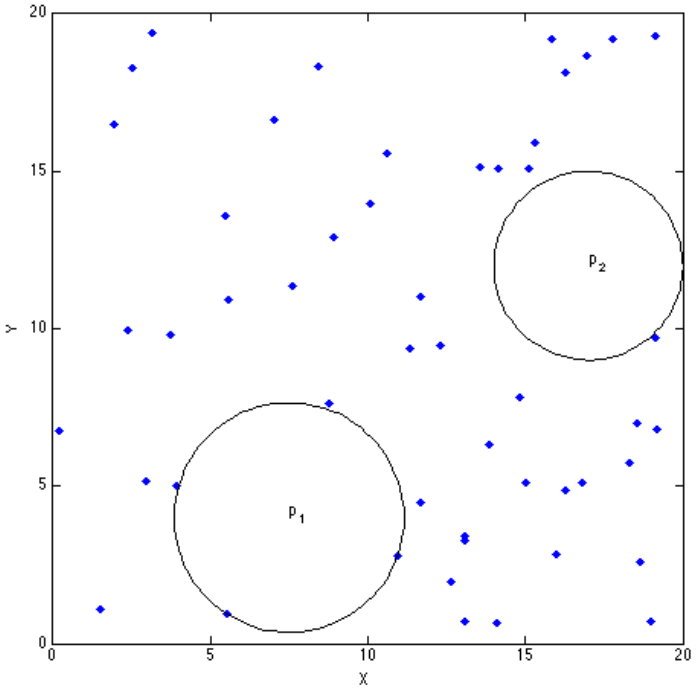

Figure 1: Two points, p_1 and p_2, with circles representing the distance to the nearest "star".

Task 1: Genetic Algorithms

For a random scattering of points, which we'll call "stars", in a bounded area, we wish to find the largest circle that can be drawn inside those same bounds that does not enclose any stars (see Figure 1). Solving this problem with a genetic algorithm requires deciding how to encode as a genome information sufficient to represent any solution. In this case, the only information we need is the center point of the circle, so our genome of point p_i will look like (x_i, y_i), representing the Cartesian coordinates.

Question 1: *Think about what each of the genetic operators means for this simplistic genome. Geometrically speaking, what would a 1-point crossover look like in this case? What about mutation?*

For the fitness of a genome, we'll use the radius of the largest circle that can be drawn at the coordinate it represents. Since the circle cannot contain any of the stars, that radius is the distance to the nearest star:

$$circle_{obj}(p_i) = \min_{s \in stars} \left\{ \sqrt{(x_i - x_s)^2 + (y_i - y_s)^2} \right\}$$

Question 2: *Would you expect more improvement in this problem to result from crossovers or mutations? Why? Is that what you would normally expect?*

Unpack the file `gasolver.zip` into a directory you can access. Locate the file `lab4.mat` in the MATLAB file browser, and double click it to add the contents of several variables to you workspace.

Now, you can set up the circle problem and run the GA solver like this:

```
star = GAparams
star.objParams.star = star1;
[best, fit, stat] = GAsolver(2, [0 20 ; 0 20], ...
    'circle', 50, 100, star);
```

GAsolver has three required parameters, and several optional parameters:

- The length of the chromosome (in this case, 2, for our two coordinates).
- An $n \times 2$ matrix of lower and upper bounds for each dimension. If there is 1 column, every position in the chromosome gets the same bounds; alternately, you may specify bounds for each position independently.
- A MATLAB function to calculate the fitness of the population. The function name is formed by prepending `ga_fit_` to the string you pass here; so, in this case, we are using the function `ga_fit_circle`, which you can find in your workspace. This function will be passed matrices holding the population (one genome per row) and the bounds of the variables, and an object of other parameters.
- The size of the population (default is 500).
- The number of generations (default is 1000).
- A structure of parameters for the solver. The `GAparams` struct has default values you can start with, and Table 1 gives brief explanations.

The function returns three variables:

- The chromosomes of the best individuals found. Each time a new global best is located, it is added to the top of this matrix, so `best(1,:)` is always the best individual found.
- The fitness values associated with the best individuals (`fit(1)` is the best fitness encountered).
- A structure of various statistics of the run. This is used to plot performance after the fact.

Table 1: Parameters for GAsolver. Default values shown in bold.

param.	values	
geneType	**'float'**	
	'binary'	$0, 1$
crossover.func	**'1point'**	
	'npoint'	set crossover.n for number of points
	'uniform'	
	'arithmetic'	(float) weighted average of n parents
	'geometric'	(float) negative values result in imaginary offspring!
	'linear'	(float) 3 offspring on a line through parents, keep best 2 (requires extra fitness evals, but more exploration)
	'none'	no crossover operator
crossover.weight	$[0-1, \dots]$	vector of parent weights (size sets # of parents)
crossover.prob	$0-1$	chance that given parents produce an offspring (default is .9)
mutate.prob	$0-1$	default is 0.005
mutate.step	$0-$	standard deviation of distribution for step size (used in some floating point mutation ops)
select.func	**'proportional'**	selection based on actual fitness values
	'rank'	selection based on ranking of fitness values
	'tournament'	pick best from pool of n parents
	'random'	ignore fitness, pick at random
select.sampling	**'roulette'**	chance of selection proportional to fitness
	'stochastic'	number of offspring proportional to fitness [1, section 8.5.3]
select.size	$\mathbf{2}-$	(tournament) size of pool (default = 2)
select.pressure	$1-\mathbf{2}$	(rank) selection pressure (default = 2)
scalingFunc	**'none'**	use raw fitness values
	'sigma'	scale fitness based on variance (per generation)
scalingCoeff	$0-\mathbf{1}$	less selection pressure at high values
replace.func	**'all'**	generational GA
	'worst'	replace worst individuals of current gen
	'random'	offspring replace random individuals
replace.elitist	**true**, false	fittest chromosome always survives
visual.active	Boolean	visualize the objective function
visual.func	string	name of the visualization function
visual.step	> 0	number of generations between updating the visualization
verbose	**true**, false	provide feedback in every generation
stop.func	**'generation'**	stop at maximum generation count
	'improvement'	stop when best fitness stops improving
	'absolute'	stop when a (known) optimum is reached
stop.window	default: 100	generations to wait for fitness improvement
stop.direction	'max','min'	whether to maximize or minimize the objective
stop.value	default: 1	known best fitness value
stop.error	default: .005	how close to get to optimum before stopping

Run GAsolver now. You should get a list of the average and maximum fitness in each generation; once all generations are complete, you will also see the genome of the individual with the highest fitness.

To get a better idea of what the solver is doing, we'll supply it with a visualization function that will plot some information from intermediate generations. We'll do this by adding the name of the visualizer function to the parameter structure:

```
star.visual.active = 1;
star.visual.func = 'circle';
[best, fit, stat] = GAsolver(2, [0 20 ; 0 20], ...
    'circle', 50, 100, star);
```

Run GAsolver again. As the solver runs, the small green circles represent the coordinates of the population of the current generation, and the large circle shows the objective value of the best individual. Run it several more times and observe how the spread of the population over the solution space changes as the population evolves. Odds are that it won't perform terribly well (a few of the default parameters are not particularly good for this problem). To this point, you've only been running for 100 generations, so one possibility is to give the solver more time to improve the solution; however, extra generations are unlikely to help in this case. To see why, we'll take a look at a plot of a measure of the diversity of the population over time.

```
ga_plot_diversity(stat);
```

Plot 1: *Include a plot showing the change in diversity of the population by generation.*

Question 3: *What are the possible sources of population diversity when using genetic algorithms? How do are those sources of diversity reflected in the shape of the diversity curve in your plot?*

Question 4: *Why is population diversity important?*

To try and maintain diversity a little longer, we'll modify some of the parameters to the algorithm. You can get an overview of the current settings like so:

```
ga_show_parameters(star);
```

One thing that can affect diversity is the choice of a selection heuristic. The default here, which you have been using up to now, is *proportional* selection, but, as you no doubt recall from lecture, *rank* based selection is often a better choice for avoiding premature convergence.

Question 5: *What's the difference between rank and proportional selection? Why does that make rank based selection better at avoiding premature convergence?*

In GAsolver, choices like selection heuristics, crossover and mutation operations, and replacement strategies are all implemented as MATLAB functions. To select a different function, you supply its name as a parameter, like this:

```
star.select.func = 'rank';
```

Rank selection also makes use of the selection pressure parameter, a number between 1 and 2 which affects the likelihood of more fit individuals being selected: at a value of 2, high fitness individuals are much more likely to be selected than low fitness individuals, while at 1 there is a much smaller difference in the probabilities.

Plot 2: *Provide one plot each of the population diversity over 100 generations using rank based selection with a pressure value of 2, 1.75, 1.5, 1.25, and 1.*

Question 6: *What selection pressure resulted in the most promising looking diversity curve? Run the algorithm a few more times using that pressure setting. What was the best fitness value and position you were able to locate?*

The replacement strategy determines which individuals from the old population and the new pool of offspring survive into the next generation. The default here is a *generational*, meaning that the offspring completely replace the old population, and *elitist*, meaning that the best individual encountered so far is always preserved unchanged.

To observe the affect elitism has on the algorithm, make 5 runs using the current settings, and make a record of which generation the best individual was created in for each trial. Also record whether the best individual was made by crossover or mutation. You can also plot the fitness of the population at each generation like this:

```
ga_plot_fitness(stat);
```

Repeat this experiment for 5 more runs, after deactivating elitism:

```
star.replace.elitist = false;
```

Plot 3: *Include fitness plots for one elitist trial, and one non-elitist trial.*

Question 7: *What affect does an elitist strategy have on the progress of fitness values?*

Question 8: *Report the generation the best individual was born in, and whether it was created by crossover or mutation, for the each of the elitist and non-elitist trials. What differences, if any, do you observe between the two methods?*

Now try the solver on two other star maps. `star2` has a global optimum in an area of low density which also has a couple of strong local optima; in `star3` the global optimum is found in the middle of an otherwise high-density section. Pass each of these maps to the solver by setting `star.objParams.star` equal to the new map, and then running the solver again, using the settings that have resulted in the best performance so far.

Question 9: *What settings did you use? How well did the solver perform on the more difficult maps? Explain any difference in performance you observed.*

Task 2: Minimizing a Function

In this task you will use genetic algorithms to minimize Ackley's function, a widely used test function for global minimization. Generalized to n dimensions, the function is:

$$f(x) = -a \cdot e^{-b\cdot\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}} - e^{\frac{1}{n}\cdot\sum_{i=1}^{n}\cos(cx_i)} + a + e^1 \quad \text{for} \quad -32.768 \leq x_i \leq 32.768, \quad i = 1, \ldots, n \tag{1}$$

We will use the standard parameters $a = 20, b = 0.2, c = 2\pi$. Ackley's function has several local minima, and a global minimum of $f(x) = 0$, obtainable at $x_i = 0$ for $i = 1, \ldots, n$. Take a moment to familiarize yourself with the shape of the two-dimensional Ackley's function, using the supplied visualizer function:

```
ack = GAparams;
ack.visual.type = 'mesh';
ga_visual_ackley([],[],[],[],[],[],ack.visual,[],[]);
```

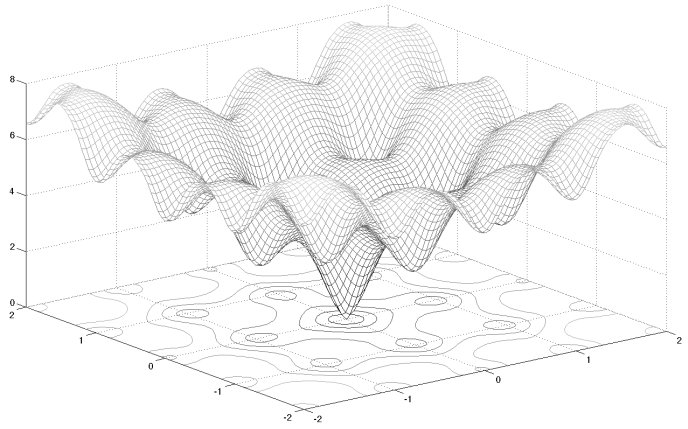

Figure 2: The area around the global minimum for Ackley's function in 2-D.

This function is challenging partly because of the strong local minima that surround the global minimum. To get a better look, you can change the bounds of the surface plot as follows:

```
ack.visual.bounds = [-2, 2];
ack.visual.interval = 0.05;
ga_visual_ackley([],[],[],[],[],[],ack.visual,[],[]);
```

You can also change the plotting function to use many of the 3-D plots supplied by MATLAB , by substituting the name of the function for '`mesh`' above. Examples include '`contour`', '`surf`', '`surfc`', etc. Just remember that any adjustments you make to `ack.visual` will also affect the visualization you get when you run the solver.

Question 10: *Why would the global minimum be difficult to find using a gradient descent method?*

Set up the problem for solving the 20 dimensional Ackley's function:

```
ack.stop.direction = 'min';
ack.visual.func = 'ackley';
ack.visual.active = true;
[best, fit, stat] = GAsolver(20, [-32, 32], ...
   'ackley', 200, 250, ack);
```

This can be a difficult function to minimize, and the initial settings may not

work well. You can try to modify the selection or replacement strategies, as we did in the last section, but this time we'll also try a few other approaches.

First, try some different crossover strategies. You can find some possibilities listed in Table 1. The 1-point, n-point, and uniform crossover operations operate by swapping the values of some genes between the parents. This is how binary encoded crossover usually works, but it's also valid for floating point encodings like we're using here. The arithmetic, geometric, and linear operations are specifically intended for floating point encodings, and all generate offspring that are some weighted linear combination of the parents. The effectiveness of the float operators depends on the `weight` parameter, a vector of length n that gives weights for to each parent (and, incidentally, determines the number of parents).

Try a number of crossover operators, and try to determine which performs best on this problem. Don't forget to pay attention to the population diversity, as well as the fitness of the best solution.

Question 11: *What effect does the choice of crossover operation appear to have on the population diversity?*

Plot 4: *Include a diversity plot for the operator that generated the best looking diversity curve. Be sure to specify which operator it was!*

The mutation operator can also have a profound effect on performance. There are only two operators provided here, uniform and inorder (the latter restricts possible mutations to a randomly selected region of the gene). However, there are a couple of different strategies for dynamically adjusting the mutation rate. The first strategy is to start with a large mutation rate, and decrease it over time. This strategy is controlled by setting the `mutate.decay` parameter to `'none'`, `'linear'`, or `'exponential'`. The second strategy is to base the probability of mutation on the fitness of the individual, where less fit individuals are more likely to mutate. This is controlled by setting `mutate.proportional` to true or false.

Try both of these strategies (you can also combine the two). You may need to increase the mutation probability, as the number given will now be the maximum probability of a mutation.

Question 12: *Which (if any) of the dynamic mutation probability strategies improved the performance of the algorithm? What difference do you observe in the results?*

Now, let's revisit the replacement strategy. In generational GA, the offspring replace the parents each generation. So far, we've been replacing all the parents except for the best individual, but this might not be the best strategy.

Another option is to only take offspring that have a better fitness. We'll call this *comparative* replacement (there doesn't seem to be a commonly accepted name for it). You can activate it by setting `replace.comparative` to true. Try comparative replacement now, using a few of the more promising combinations of parameters you've encountered up to this point.

Question 13: *Describe the effect (if any) comparative selection has on population diversity and convergence.*

Question 14: *What was the fitness value and position of the best solution you were able to locate? Report the parameter settings you used to generate that solution.*

Question 15: *Try using GAsolver to find the minimum for the 100 dimensional Ackley's function, using the same parameters as the last question. How well does it perform now? Can you improve the performance?*

Task 3: Constrained Optimization

One similarity between the last two problems is that every point in the search space, and therefore every genome, represented a solution to the problem. Fitness was used to distinguish between better or worse solutions, but the position with the worst fitness still represented a solution. For a large class of real-world problems, it is difficult to define the search space so that it contains only solutions; instead, some parts of the search space (possibly small, isolated parts) solve the problem, while other do not. These problems are called *constrained* optimization problems, and are characterized by a set of constraints that distinguish between *feasible* answers, which solve the problem, and *infeasible* answers, which do not.

There are many ways to handle constraints using genetic algorithms. We'll use what is called a *penalty* method: that is, we allow chromosomes to represent both feasible and infeasible solutions, but we modify the fitness function by adding a penalty to all infeasible solutions. The penalty is a measure of constraint violation. All infeasible solutions should receive a penalized fitness that is worse than the unpenalized fitness of the worst solution. The advantage of penalty methods is that they allow exploration of the infeasible region, which may lead to the discovery of new feasible solutions; the disadvantage is that the penalty function effectively changes the slope of the fitness function, and may introduce false optima.

For a more detailed discussion of constrained optimization, see Section 9.6.2 and Appendix A.6 of the text book.

3.1 Resource Constrained Scheduling

For this task, you will solve a Resource Constrained Scheduling Problem (RCSP). The RCSP problem is NP-hard: there is no polynomial-time algorithm to solve it, although there are many relaxations of the problem that can provide an approximate solution in polynomial time.

The problem is to schedule a set of n tasks by assigning a start time ($start_i$) to each. These tasks share one or more resources. Each resource has a finite capacity C, which limits how many tasks can use the resource at the same time. Resource capacities are renewable, but cannot be saved for use at a later time. For example, if our resource is water, then it is like a pipe that can deliver a fixed amount at any time; it is not like a water tank holding a finite amount of water, which is gradually emptied. For this problem, assume that the capacity of the resource is constant, and that each task requires a fixed amount of the resource (use_i) and lasts for a fixed period of time (dur_i).

There are two types of constraints in this problem:

- **Precedence**: some tasks may not begin until another task has completed. If task a must precede task b, then the start time of b must always be greater than or equal to the completion time of a in any feasible solution.
- **Resource**: The the sum of the resource usage of all tasks executing at the same time must never exceed the capacity of the resource.

The best solution is the schedule that requires the least amount of time for all tasks to complete. We call this time the *makespan* of the schedule.

3.2 Encoding and Fitness[1]

The simplest genome for this problem is a set of start times: for n tasks, we have a genome of length n, where each position represents the time that a particular task starts. A complete set of start times exactly represents a solution to an RCSP instance; furthermore, the resource constraints may be easily computed as:

$$\forall t: \quad \sum_{\substack{i \in tasks \\ start_i \leq t \leq start_i + dur_i}} use_i \quad \leq C \tag{2}$$

The precedence constraints, on the other hand, are more difficult to enforce, and there could be on the order of n^2 of them.

The RCSP problem in the lab files uses a different encoding. Instead of a start time for each task, the genome will encode the *delay* in starting that task; that is, the amount of time that passes from the latest completion of all

[1]This section is based on work in [3].

For each constraint i on the problem, we'll define a measure of the *violation* of that constraint $x_i \in [0, \infty]$, where 0 means there is no violation of the constraint. For a problem with n constraints, the total measure of constraint violation for a solution is given by:

$$norm\_violation = \frac{1}{n} \cdot \sum_{i=1}^{n} \frac{1}{1 + x_i} \qquad \in (0, 1] \tag{3}$$

The raw objective value is the maximum time allowed, called the horizon, minus the makespan for the current solution. This is normalized based on the minimum possible makespan ignoring the resource constraints (called the duedate here):

$$norm\_obj = \frac{horizon - makespan}{horizon - duedate} \tag{4}$$

To ensure that feasible solutions have a higher objective value than non-feasible solutions, we compute the final objective value as follows:

$$fitness = \begin{cases} \frac{norm\_violation}{2} & \text{if } norm\_violation < 1 \\ \frac{1 + norm\_obj}{2} & \text{if } norm\_violation = 1 \end{cases} \tag{5}$$

If there is *any* constraint violation, then the score is a measure of the degree of violation. If there is no violation, then the score is the maximum value for unviolated constraints plus the raw fitness.

tasks preceding the current task, until the start of the current task. In this representation, the precedence constraints are enforced for free, as long as we choose appropriate bounds for the delay times.

Question 16: *For the* delay *encoding, what should be the minimum value allowed at each position? Why? The maximum value is more difficult to define; give your best guess for the maximum value at a particular position, and motivate your guess.*

The resource constraints will be enforced using the penalty method, as described in the shaded box. The penalty function is computed in `ga_fit_rcsp.m` along with the raw fitness value (a normalized makespan). A penalized fitness of 1 represents the optimal solution; values over .5 are feasible, suboptimal solutions; values of .5 or lower are infeasible solutions.

3.3 Trying it out

The `j30rcp` subdirectory holds a suite of 480 RCSP benchmarks [2]. Each consists of 32 tasks of fixed duration, with precedence constraints. There are four resources of different capacities, and each task requires only one resource.

```
rcsp = GAparams;
rcsp.visual.func = 'rcsp';
rcsp.visual.active=true;
rcsp.visual.step = 50;
rcsp.init.func = `rcsp';
rcsp.objParams = importRCSP('J301_1');
```

The last line loads the `J301_1` benchmark instance, a very easy problem. The problem specification is stored in the `objParams` struct, including the horizon, duedate, and matrices of resource usage and task durations. There is also a large precedence matrix, `successor`, where `successor(a,b)` is 1 if task b comes after task a, and 0 otherwise. There is also a set of bounds for the tasks in `objParams`, which you will need to pass when you start GAsolver.

Also notice that there is a specialized initialization function for this problem. This is because the vector `objParams.makespan` is used to store the best makespan found in each generation, and it needs to be initialized when the program starts.

```
[best, fit, stat] = GAsolver(32, ...
    rcsp.objParams.bounds, 'rcsp', 100 , 200 , rcsp);
```

The visualization function displays a representation of the resource utilization for the best solution found so far. For each resource in the problem, there is a plot similar to Figure 3. The horizontal dimension is time, while the vertical shows the units of demand for the resource. The bar at each time is the cumulative demand on that resource of all tasks that have been scheduled to execute then. There are also two colored lines in the MATLAB plots. The red, horizontal line represents the capacity of the resource; in a feasible solution, no bar should be over this line. The green, vertical line on the right represents the horizon of the problem; any solution with a makespan greater than the horizon will be penalized in the same way that resource constraints are penalized.

Remember that the way the penalized fitness has been defined, any chromosome that represents a feasible solution will have a fitness in the range $(0.5, 1]$. There's a pretty good chance that your initial run didn't manage to find any feasible solutions. There are a couple of reasons for this; we'll consider them one at a time.

First of all, we could do a better job picking the initial population. Up to now, starting gene values have been randomly selected over the range at each gene. As you saw earlier, its difficult to define a tight maximum value in this problem, so our ranges are quite large; as a result, the algorithm is spending a lot of time looking at unpromising parts of the search space. We can get a

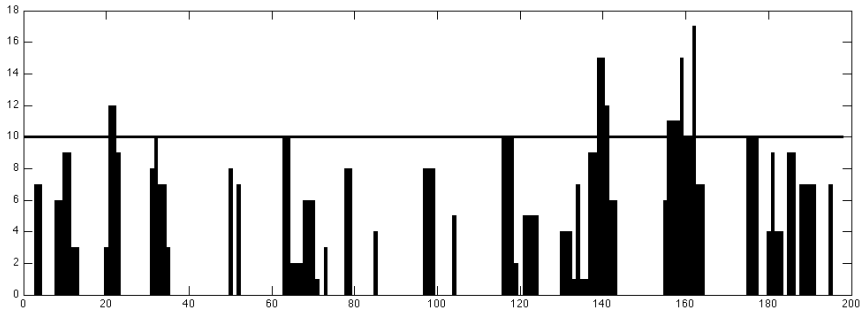

Figure 3: Resource utilization over time.

more reasonable set of starting values by changing the probability distribution we use to initialize the population to a normal distribution, like this:

```
rcsp.init.dist = 'normal';
rcsp.init.start = '???';
rcsp.init.variance = ???;
```

As you can see, there are a couple of parameters you'll need to set. `init.start` takes one of three values, `min`, `mean`, or `max`, defining where in the gene's range the peak of the normal distribution should lie; the variance determines the width of the normal curve.

Question 17: *What is the best position for the peak of the normal curve in this case? Why?*

Question 18: *What is a reasonable value for the variance here? Justify your choice. Hint: try to formulate this in terms of task length.*

Another modification you may want to try is to adjust the generation gap. So far we've been using generational GA; another approach is called *steady-state* GA: some portion of the population survives into the next generation. In this case, the survivors are selected using a replace-worst strategy, meaning that the offspring replace the older genes which have the worst fitness. Steady-state GA can be activated like so:

```
rcsp.replace.gap = 0.25;
rcsp.replace.func = 'worst';
```

The default generation gap is 0, meaning the entire population is replaced. With the above setting, only 75% of the population will be replaced.

Question 19: *What advantage could there be to using steady-state for this problem? What generation gap value seems to work best for you?*

Once again, you'll probably need to adjust several of the parameters before you can start reliably locating feasible solutions. The optimal solution is actually quite difficult to locate, but you should be able to reliably find fitnesses over .9. Once you have a set of parameters that appears to work well, try it out an a couple other easy benchmarks from the `J301_x` through `J304_x` instances.

Question 20: *List the parameters that gave you the best performance. Explain why the crossover type and selection model you wound up with are appropriate for this problem.*

Now use your tuned algorithm to find a feasible solution to a more challenging benchmark.

```
rcsp.objParams = importRCSP('J3021_5');
```

You'll probably need to let the solver run for more generations to give it time to locate a solution.

Plot 5: *Include a resource usage plot for the best solution you were able to find to the `J3021_5` instance.*

Question 21: *What was the best feasible solution you found, and what was its makespan? How does that compare with the optimal makespan for that problem?*

Task 4: Particle Swarm Optimization

There are a number of Particle Swarm toolboxes available for MATLAB , although at this time there are none that are very polished. For this task, you'll be using an open source PSO toolbox modeled after the MATLAB Optimization Toolbox. You will need the file `psopt.zip`, which you can find on the lab web page. Unpack it into a directory you can access, and add both the `psopt` directory and its sub-directories to the MATLAB path.

We'll start by exploring Ackley's function once more.

```
options = psooptimset('DemoMode','pretty',...
        'PlotFcns',{@psoplotswarmsurf,@psoplotbestf},...
        'PopInitRange',[-32;32],'InertiaWeight',1,...
        'SocialAttraction',2,'CognitiveAttraction',2);
[x, fval] = pso(@ackleysfcn,20,[],[],[],[],[],[],[],options);
```

The first line creates a structure of parameters. You can view the available parameters by typing `psooptimset` with no arguments or return values. When you create an options structure like `options` here, you only need to specify fields and values that you want to differ from the defaults. The `pso` function takes a handle to the objective function, the number of dimensions in the problem, and the options structure. The empty arguments in the middle could be used to specify linear and nonlinear constraints, but we'll ignore them.

When you run the swarm, you'll get a window with the two plots specified above. One shows the particle positions and fitness surface of the function (in the first two dimensions of the problem only). The other shows the best and average fitness scores in each iteration.

In this first trial, we've used settings that replicate what you could call "classical" PSO. The resulting swarm behavior demonstrates one of the biggest problems with PSO algorithms, a phenomenon known as *swarm explosion*: the velocities of the particles steadily increase until the swarm flies apart. There have been lots of methods proposed to deal with swarm explosion over the years, and we'll investigate a few right now.

The simplest is *velocity clamping*: we impose a maximum velocity in each dimension, and any time a particle's velocity exceeds that maximum, we reset it to the maximum and continue. To try velocity clamping, you need to set the parameter `VelocityLimit` to $\delta(\max_i - \min_i)$ in each dimension, where $\delta \in (0, 1]$. For this problem, there are no real differences between the dimensions, so you can pass a single velocity limit and it will be applied to all dimensions. Try clamping with velocity limits of 30, 15, 5, and 1. In order to better see the effect of velocity clamping, add `@psoplotvelocity` to the list of plotting functions.

Question 22: *What happens to the maximum velocity of the swarm over time when using velocity clamping? How does the maximum velocity compare with the theoretical velocity limit (the red line at the top of the velocity plot)?*

A more interesting approach to controlling swarm explosion is by introducing *inertia weights*. Inertia in a PSO refers to the amount of influence a particle's current velocity has on the particle's next velocity. To understand the inertia weight, we need to consider the velocity equation for global best, or *gbest*, PSO

(the effect on *lbest* and *pbest* is similar).

$$v_{id}(t) = wv_{id}(t-1) + c_1 r_1[p_{id} - x_{id}(t-1)] + c_2 r_2[p_{gd} - x_{id}(t-1)] \quad (6)$$

Here $v_{id}(t)$ is the velocity of particle i in dimension d at time step t, $x_{id}(t)$ is the position of that same particle, p_{id} is the particle's personal best position, and p_{gd} is the global best position. The constants c_1 and c_2 are positive acceleration factors controlling the particles' cognitive and social components: high c_1 values cause the particle to have increased confidence in its own previous best, while high c_2 values cause the particles to have increased respect for the group's best position. You can change the values of c_1 and c_2 with the `CognitiveAttraction` and `SocialAttraction` options, respectively. Frequently the values assigned to these constants are somewhere around 2.

The inertia weight is another constant, w, controlled by the option `InertiaWeight`. Up to now we've used a value of $w = 1$, which replicates the *gbest* velocity equation without inertial weight. Setting $w \geq 1$ causes the particles to accelerate constantly, while $w < 1$ should cause the particles to gradually slow down. By carefully balancing the three parameters w, c_1 and c_2, it is possible to fine tune the algorithm, balancing both the social and cognitive components, and the tension between exploration and exploitation. There are a lot of suggested combinations for these values in the literature; for example $c_1 = c_2 = 2$ and $w = 1$, or $c_1 = c_2 = 1.49$ and $w = 0.7968$. Try those combinations now, as well as a few others, and see how well behaved the resulting swarms are. Be sure to try both with and without velocity clamping turned on.

Question 23: *What values for w, c_1 and c_2 worked best on this problem? How close did this swarm come to locating the global optimum?*

Question 24: *Was velocity clamping still necessary to prevent swarm explosion, or were you able to find a combination of values that kept the swarm together?*

Another very common modification is to let the inertia weight decay over time. To cause the weight to linearly decay, set the inertia weight to a vector $[w_{\max}; w_{\min}]$. A frequently used range is $[0.9; 0.4]$. Try switching off velocity clamping, and see if you can get the inertia weight set to some combination of values that reliably comes close to the optimum, and avoids the swarm explosion.

Question 25: *What was the best combination of decaying inertia weights, and social and cognitive coefficients? What was the best fitness value you found using that combination?*

At this point, it should be clear that the right combination of inertia weight, social and cognitive constants, and velocity clamping can yield a pretty reliable particle swarm. There is one more quite interesting method for avoiding swarm

explosion that we should consider, however, which is called *constriction*. The constriction velocity equation for *gbest* is

$$v_{id}(t) = \chi[v_{id}(t-1) + \phi_1 r_1(p_{id} - x_{id}(t-1)) + \phi_2 r_2(p_{gd} - x_{id}(t-1))] \quad (7)$$

χ is called the *constriction coefficient*. Letting $\phi = \phi_1 + \phi_2$, we further require that $\phi > 4$, and then define χ in terms of ϕ.

Question 26: *What is the equation for the value of the constriction coefficient in terms of ϕ?*

The advantage of constriction is that it does not require velocity clamping. As long as the parameters are selected as described, the swarm is guaranteed to converge. Unfortunately, the toolkit we are using does not include constriction. We can, however, implement constriction using the inertia weight approach. By choosing the right values of w, c_1 and c_2, we can make equation (6) be equivalent to equation (7) for a specific ϕ_1 and ϕ_2.

Question 27: *Consider the constriction equation with $\phi_1 = 4$, $\phi_2 = 2$. What is the constriction coefficient for these values? What values of w, c_1 and c_2 would we have to use in (6) in order to implement constriction with these values?*

Try constriction now for $\phi_1 = 4$, $\phi_2 = 2$, using the appropriate parameters in the inertia weight model.

Question 28: *Describe the behavior of the swarm when using constriction. Does it locate the global optimum? How quickly does it converge?*

Task 5: Training a Neural Network

One widely used application domain for both GA and PSO is the tuning of parameterized systems. You've been dealing with a parameterized system in this course: neural networks. Training a neural network means adjusting the weights of the inputs; every weight is a parameter of the system. The objective function for a neural net is the measure of the network's error (mse, sse, etc.). For a network with a given topology, the error depends only on the weights—in other words, the set of weights is a sufficient encoding of the solution space. We'll start by training a network to solve the XOR problem from the first lab.

Question 29: *Given the network topology we used in Lab 1, and the problem definition above: how many dimensions should there be to the problem of training a neural network to solve XOR?*

Gradient descent and related learning rules for neural nets are *local* optimization algorithms: from a single starting state (i.e. random weights assigned in initialization), the algorithm searches for an optimum, employing various strategies to avoid getting stuck in local optima. Recall Lab 1, when you trained an XOR network: from some start positions, gradient descent led to a network that encoded one of the two possible solutions, while from other start positions it found a non-solution that represented a local minimum or plateau. PSO is *global* optimization: it searches larger parts of the solution space. As a result, it should be less likely to get stuck.

Just like in lab 1, we start by creating the network:

```
global psonet P T
P = [[0; 0] [0; 1] [1; 0] [1; 1]]
T = [0 1 1 0]
psonet = newff(P, T, [2], {'tansig' 'logsig'}, 'traingd', ...
            '', 'mse', {}, {}, '')
```

Make sure you name the variables exactly as given above. We're defining these variables as global so that the fitness function will be able to see the net you've created, but it only works if the names are exactly the same (sometimes programming in MATLAB is just like that). Also, we've specified a training algorithm for the network here because MATLAB expects one, but we won't actually use it to train the network.

The fitness function we'll be using is `nntrainfnc`, and you can find it in the `testfcns` directory. Take a look at the function, it's quite straightforward. For each particle, it sets the weights of the network to the values of the particle's current position. Then it calculates the error using those weights for all the test patterns. The error is the fitness value that we wish to minimize. Unfortunately, this is all implemented using fairly high-level access to the neural network's structure, so the PSO is going to run quite a bit slower. You probably will want to compensate by turning the number of particles down (something in the range of 5-10 should work well).

Here's the basic setup for training the network:

```
options = psooptimset('PopInitRange',[-1;1],...
  'PopulationSize',5,'DemoMode','pretty',...
  'PlotFcns',{@psoplotswarm,@psoplotbestf,@psoplotvelocity});
[x, fval] = pso(@nntrainfcn,9,[],[],[],[],[],[],[],options);
```

Note that we are now using `psoplotswarm` instead of `psoplotswarmsurf`.

Question 30: *We've defined the problem as having 9 dimensions, which is (probably) not the answer you gave for question 29. What possible explanation is there for the difference?*

After training, you can evaluate the best solution the PSO found like this:

```
psonet = setx(psonet, x);
plot_xor(psonet);
```

Naturally, you may need to make a few parameter adjustments before you get a good solution.

Plot 6: *Include the output from* `plot_xor` *for the best solution you were able to locate.*

Question 31: *How does the plot of the PSO solution compare with the plots you got in lab 1? Why might these plots be different?*

Task 6: Wrapping up

Question 32: *GA and PSO are both population methods, and they can be used to solve the same type of problems. What characterizes a problem that would be easier to solve using GA? What about a problem that is easier to solve with PSO?*

Question 33: *Propose a real-world problem that you consider interesting and that you believe can be solved using what you just learned. Explain in a few sentences how that would work.*

References

[1] Andries P. Engelbrecht. *Computational Intelligence.* John Wiley & Sons, 2 edition, 2007.

[2] R. Kolisch and A. Sprecher. PSPLIB - a project scheduling library. *European Journal of Operational Research*, 96:205–216, 1996.

[3] Matthew Bartschi Wall. *A Genetic Algorithm for Resource-Constrained Scheduling.* PhD thesis, MIT, 1996.