MCC Summer School 2005
Genetic Algorithm Lab Solution

Gus Hart, Dept. of Physics and Astronomy, Northern Arizona University

July 5, 2005

1 The problem

The problem given in the lab was to find the configuration of particles on a 1D lattice that optimize the energy. Each site of the lattice is connected to every other site with a random coupling strength. "On-site" terms are also present (the diagonal terms below). The problem in the lab was to find the optimum configuration for N_{part} particles on a lattice with N sites. We can make the search problem harder but make the programming task easier by relaxing the restriction of a fixed number of particles and instead trying to find *both* the number and arrangement of particles on the lattice that minimizes the energy. This is the problem we'll treat in the solutions. Once you've gone through these solutions, it will be obvious how to add the restriction to a fixed number of particles.

In summary then, the problem is to minimize the following Hamiltonian:

$$E(\vec{\xi}) = (\xi_1, \xi_2, \cdots, \xi_N) \begin{pmatrix} J_{1,1} & J_{1,2} & \cdots & J_{1,N} \\ J_{2,1} & J_{2,2} & \cdots & J_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ J_{N,1} & J_{N,2} & \cdots & J_{N,N} \end{pmatrix} \begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_N \end{pmatrix} \tag{1}$$

where $\vec{\xi}$ represents the occupation of sites—a string of zeroes and ones (an entry for each site on the lattice)—and $\{J_{i,j}\}$ are the coupling constants (i.e., the strengths of the interaction between the ith and jth sites). If $i = j$, then the interaction represents, the "on-site" energy for a particle occupying that position in the lattice. I chose this Hamiltonian because it is so simple to program and can be evaluated quickly, but at the same time, the search space is enormous, 2^N.

2 Getting started

The first thing we will do in this solution is generate the $E(\vec{\xi})$ data for a case where N is large but not so large that we can't compute all possible permutations. The function m-file `enumerate.m` generates all possible permutations for an N-site lattice and computes the energy for each one.

```
function [energies,interactions] = enumerate(N)
% ENUMERATE function to enumerate all possible 2^N occupations
% on a 1-D lattice of N sites.
%
% This function returns all of the energies as well as the
% Hamiltonian matrix. This function depends on (requires)
% the "compute_energy" function. If N is large, this function is
% very slow. N=14 requires about 10 seconds on an average PC. The
% required time scales as 2^N (for obvious reasons)
```

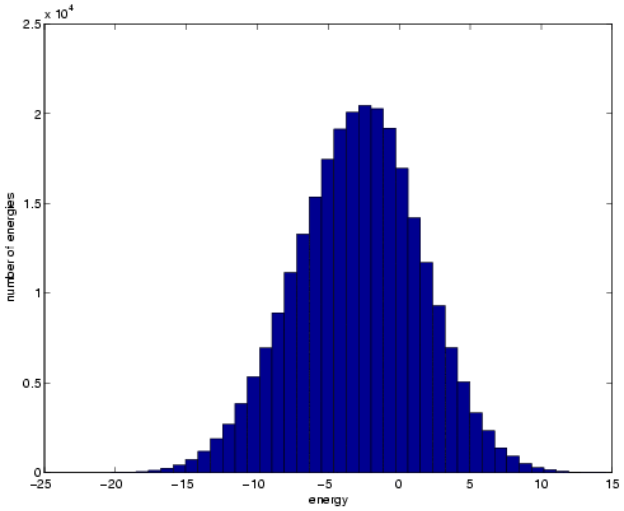

Figure 1: Histogram plot of the energies for $N = 18$

.

```
%
% Call this function from the command line and then save the variables
% to a file when experimenting with the GA so that the enumerations does
% not have to be repeated each time one runs the GA.
%

% These two lines define the Hamiltonian--the intersite coupling and
% "on-site" terms. The second line makes the Hamiltonian symmetric---
% not necessary but makes physical sense
interactions = rand(N,N)-.5;
interactions = interactions + interactions';

% This block of codes loops over each possibility and stores the
% corresponding energy.
Nperms = 2^N;
energies = zeros(1,Nperms);  % Initialize the array for energies
for i = 1:Nperms
   binstr = dec2bin(i-1);           % Convert the number to binary  (0...2^N-1)
   occupation = str2num(binstr');   % Convert it to a string
   occupation = [zeros(N-length(occupation),1); occupation]; % Pad with extra zeroes
   energies(i) = compute_energy(interactions,occupation');
end
```

Run this function at the command line `[energies,H]=enumerate(12)`. Choose a value of N that isn't going to requires hours of cpu time. On my old laptop, $N = 18$ takes about 3 minutes and gives a search space of $2^N \approx 2.6 \times 10^5$. When the function finishes, save the variables, `energies` and `H`, to disk so that you can reload them later; `save Precomputed_EandH energies H`. If you make a histogram plot of the energies, you'll see something like that in Fig. 1. Having a specific Hamiltonian and its energies precomputed will allow us to quickly test our genetic algorithm code.

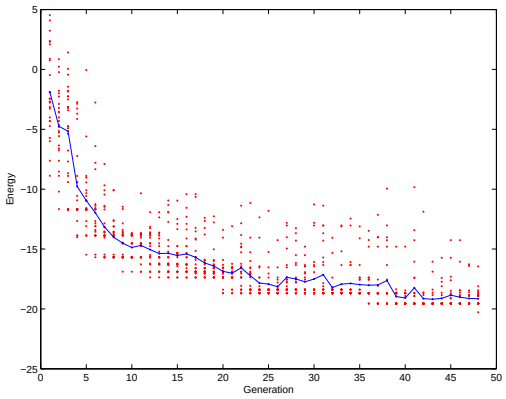

Figure 2: Sample run of the genetic algorithm for $N = 18$, $N_{\text{pop}} = 25$, $r_{\text{mut}} = 0.05$ mutations/gene. The red points indicate the energy of each individual in each generation. The blue line is the average energy for each generation. The minimization ends at 48 generations when the global minimum is found.

3 GA details

Included in the solution codes is a function `ga_minimize` that takes a specific Hamiltonian, population size, mutation rate, etc., and minimizes the given Hamiltonian using a genetic algorithm. A sample run is shown in Fig. 2.

In this particular implementation of a genetic algorithm, all the parents in each generation are replaced by new children (that is, no parents survive from one generation to the next). Parents are selecting for mating using the so-called "Roulette Wheel Selection" (see `http://cs.felk.cvut.cz/~xobitko/ga/selection.html`). In short, parents with better fitness scores are more likely to be selected.

Mating is peformed using a one-point crossover. That is, the parent genomes are sliced at a random location along the genome and then spliced together to make a new child genome. Mutations are introduced randomly (usually rarely). Each site site in a genome can be mutated. Thus, some child genomes my have several gene sites mutated while other will have none.

Here is the code for the GA minimizer function:

```
function [en_hist, it_count] = ga_minimize(H, Npop, mut_r, max_its, BFmin)
% GA_MINIMIZE attempts to find a minimum using a genetic algorithm
% H is the Hamiltonian, i.e, the coupling matrix (diagonal entries are
% on-site energies
% indv is a 2D array, each row is an individual, a string of 1's and 0's

en_hist = ones(max_its,Npop)+BFmin; % History of energies over the iteraction of the GA
```

```
                                % initialize to something larger than BFmin
it_count = 1;                   % Counter for number of generations (or iterations)

N = length(H);                  % Number of sites in the lattice
indv = ceil(rand(Npop,N)-.5);  % Initialize the array, random 0's & 1's
ga_ens = zeros(1,Npop);         % Energies of the GA's current population
fitness = zeros(Npop);          % Fitness scores of the GA's current population

while (1) % Break condition for the while loop is at the end of the loop

    % Compute energy for each individual in the population
    for i = 1:Npop
        ga_ens(i) = compute_energy(H,indv(i,:));
    end
minE = min(ga_ens);
    en_hist(it_count,:) = ga_ens;

    % Determine fitness scores for each individual
    for i = 1:Npop
        fitness(i) = 1/(1+ga_ens(i)-minE);
end

% Order the individuals and their fitness scores
[dummy, order] = sort(ga_ens);  % Somewhat obtuse but this code section
indv = indv(order,:);           % just orders the individuals and their
fitness = fitness(order);       % scores in ascending order of energy

% This block of code peforms the mating. Individuals with better
    % fitness scores are selected prefentially to be parents. Mating
    % is done using a simple one-point crossover
new_indv = indv; % Make a copy before mating
fitcum = cumsum(fitness)/sum(fitness); % This array is used in a "weighted" selection
                                        % of parents that favors those with better
                                        % fitness scores
for i = 1:Npop  % Loop over each individual, make one new child for each individual
         rnd = rand(1);
         [a,p1] = max(rnd<fitcum); % Select the first parent, p1
         rnd = rand(1);
         [a,p2] = max(rnd<fitcum); % Select the first parent, p2
         split = ceil((N-2)*rand(1)+1); % select the place to "split," i.e., the crossover
         new_indv(i,:)=[indv(p1,1:split-1) indv(p2,split:end)]; % Splice together the new
end

% Make random mutations
for i = 1:Npop
        muts = [ceil(rand(1,N)-1+mut_r)];  % Make an array of sites to mutate
        new_indv(i,:) = mod(muts+new_indv(i,:),2);  % At chosen mutation sites, switch
end                                                 % 1's to 0's and 0's to 1's

indv=new_indv;  % Copy children to current population
```

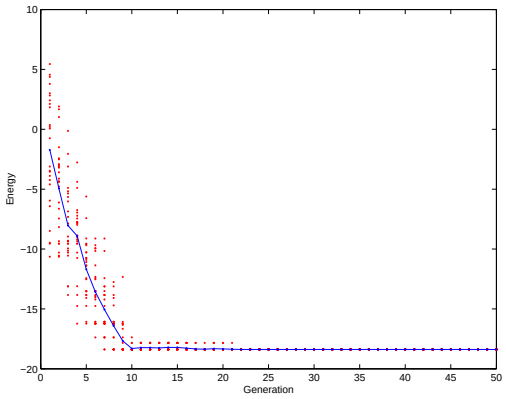

Figure 3: History of the GA minimization when the mutation rate is too small. The algorithm will get stuck in a local minimum with no way of getting out

.

```
    % Exit the function if we have found the minimum or reached the
    % maximum number of allowed iterations (generations)
    if  it_count == max_its || any(en_hist(it_count,:) == BFmin)
        break
    end
    it_count = it_count + 1;
end
```

Use the following program as a driver to play around with the effects of changing the different GA parameters. Notice that the program loads, from disk, the previous Hamiltonian and precomputed energies that were saved after running the `enumerate.m` function. When the mutation rate is too small and the population isn't large, some genetic information will be lost early in the iterations and thus some parts of the solution space will not be accessible. So the algorithm will get stuck in a local minimum. For example, see Fig. 3. On the other hand, if the mutation rate is too high, the GA degenerates to nothing more than a random search, which will be unsuccessful just because of statistics—there will be little chance of finding the correct minimum "accidentally" because of the sheer size of the search space.

```
% This program illustrates a simple implementation of a genetic algorithm.
%
% The GA optimizes a Hamiltonian that simulates the occupation
% of particles on a 1D lattice where each site is coupled (with
% a random strength) to every other site. An "on-site" term is also
```

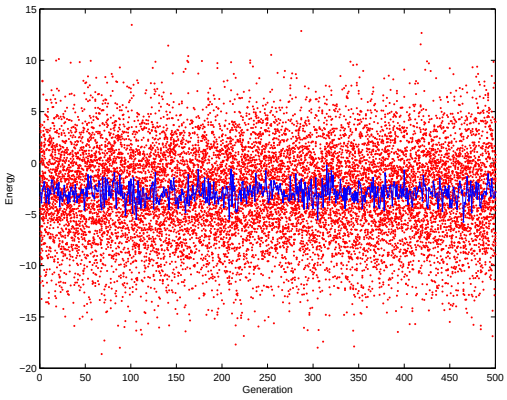

Figure 4: History of the GA minimization when the mutation rate is too small. The algorithm will get stuck in a local minimum with no way of getting out

.

```
% present for each lattice site. The objective is to find the number
% and location of particles on the lattice to minimize the energy
%
% MCC Summer School 2005
% Gus Hart Northern Arizona Univesity
% mail: gus.hart "at" nau.edu

clc   % clear the command window
clear % clear all variables
close all % close windows

load Precomputed_EandH

% Plot distributions of energies
hist(energies,40)
xlabel('Energy')
ylabel('Number of energies')


BFmin = min(energies);  % Save the minimum from the brute-force search
N=length(H(1,:));       % Number of sites in the model
Nperms = 2^N;           % Number of possible permutations of occupation variables

% Parameters for the GA
Npop = 20;         % Number of individuals in the population
max_its = 500;     % Maximum number of iterations (generations)
r_mut = .05;       % Average rate of mutations/gene

[en_hist, its] = ga_minimize(H, Npop, r_mut, max_its, BFmin);

% Calculate the average energy at each generation (iteration)
ave_en = zeros(1,its);
for i = 1:its
    ave_en(i) = sum(en_hist(i,:))/Npop;
end

% Output the results and make a plot
if BFmin == min(en_hist(its,:))
    fprintf('GA converged to correct minimum\n');
else
    fprintf('GA failed to converge\n');
end
fprintf('Brute force min: %6.3f   GA min: %6.3f\n', BFmin, min(en_hist(its,:)))
fprintf('Fraction of search space explored: %8.6f\n', Npop*its/Nperms)
fprintf('Number of iterations %6d', its)
figure
plot([1:its],en_hist(1:its,:),'r.',[1:its],ave_en(1:its),'b.-')
xlabel('Generation')
ylabel('Energy')
```

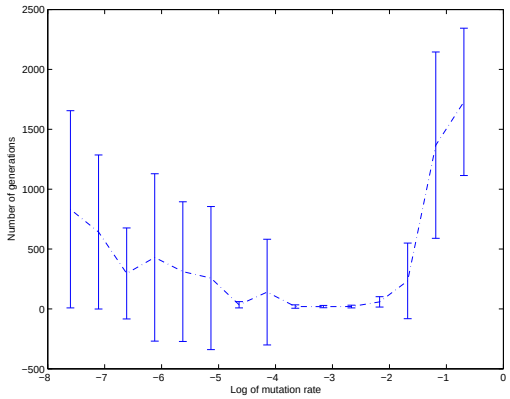

Figure 5: Study of the convergence rate and dependability of the GA as a function of mutation rate. When the mutation rate is close to the ideal value, the GA converges much more consistently, as indicated by the small variation (error bar). The error bars and graph are artificially small at each extreme of the plot. The ideal mutation rate is around 0.05 mutations/gene.

4 Optimizing the algorithm

For each GA and each search problem, the individual parameters such as populations size and mutation rate have different optimimal values. The following code sweeps through different mutation rates and measures how many generations are needed to find the solution on average. Figure 5 shows the results. The error bars indicate the variation in the number of generations to find the minimum.

We can do the same kind of test for population size. This is shown in Fig. 6. In general the population size does not have a big effect unless it is too small. In this case, the algorithm starts to behave somewhat like a simulated annealing algorithm. Note that rather that testing the convergence vs. generations, we should test the convergence vs. the total number of guesses. It's the total number of guesses, not the number of generations, that dictates the computer time required to find the solution.

The codes used for the last set of plots are included here:

```
% This program illustrates a simple implementation of a genetic algorithm.
%
% The GA optimizes a Hamiltonian that simulates the occupation
% of particles on a 1D lattice where each site is coupled (with
% a random strength) to every other site. An "on-site" term is also
% present for each lattice site. The objective is to find the number
% and location of particles on the lattice to minimize the energy
%
```

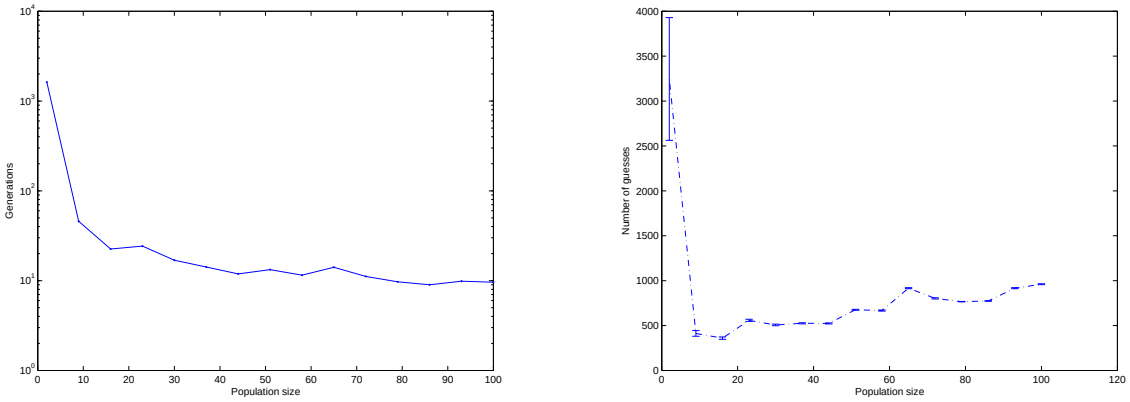

Figure 6: Convergence of the GA vs. population size. The left-hand plot shows the convergence with respect to the number of generations. The right-hand plot shows the same data but with respect to total number of guesses tried by the algorithm

.

```
% MCC Summer School 2005
% Gus Hart   Northern Arizona Univesity
% mail: gus.hart "at" nau.edu

clc  % clear the command window
clear all % clear all variables
close all % close windows

load Precomputed_EandH

BFmin = min(energies);  % Save the minimum from the brute-force search
N=length(H(1,:));       % Number of sites in the model
Nperms = 2^N;           % Number of possible permutations of occupation variables

% Parameters for the GA
Npop = 25;         % Number of individuals in the population
max_its = 2000;     % Maximum number of iterations (generations)

Ntrials = 20 ; % Number of trials to average over
Nr = 15;       % Number of different mutation rates to try
r_mut = logspace(-4,-1,Nr)*5;  % range of rates to try (logarithmically spaced)
av_gens = zeros(1,Nr);
av_err = zeros(1,Nr);
for i = 1:Nr
    av_gens(i) = 0;
    dist = zeros(1,Ntrials);
    for j = 1:Ntrials
        [en_hist, its] = ga_minimize(H, Npop, r_mut(i), max_its, BFmin);
        dist(j) = its;
        av_gens(i) = av_gens(i) + its;
    end
    av_err(i) = std(dist);
    av_gens(i) = av_gens(i)/Ntrials;
    fprintf('Finished r_mut: %6.3g\n',i)
end

% Output the results and make a plot
errorbar(log(r_mut),av_gens,av_err,'.-')
xlabel('Log of mutation rate')
ylabel('Number of generations')

% This program illustrates a simple implementation of a genetic algorithm.
%
% The GA optimizes a Hamiltonian that simulates the occupation
% of particles on a 1D lattice where each site is coupled (with
% a random strength) to every other site. An "on-site" term is also
% present for each lattice site. The objective is to find the number
% and location of particles on the lattice to minimize the energy
%
% MCC Summer School 2005
% Gus Hart   Northern Arizona Univesity
% mail: gus.hart "at" nau.edu
```

```
clc  % clear the command window
clear all % clear all variables
close all % close windows

load Precomputed_EandH

BFmin = min(energies);  % Save the minimum from the brute-force search
N=length(H(1,:));       % Number of sites in the model
Nperms = 2^N;           % Number of possible permutations of occupation variables

% Parameters for the GA
r_mut = 0.05;       % Number of individuals in the population
max_its = 2000;     % Maximum number of iterations (generations)

Ntrials = 20 ; % Number of trials to average over
Ns = 15;       % Number of different population sizes to try
Npop = linspace(2,100,Ns); % range of rates to try (logarithmically spaced)
av_gens = zeros(1,Ns);
av_err = zeros(1,Ns);
for i = 1:Ns
    av_gens(i) = 0;
    dist = zeros(1,Ntrials);
    for j = 1:Ntrials
        [en_hist, its] = ga_minimize(H, Npop(i), r_mut, max_its, BFmin);
        dist(j) = its;
        av_gens(i) = av_gens(i) + its;
    end
    av_err(i) = std(dist);
    av_gens(i) = av_gens(i)/Ntrials;
    fprintf('Finished r_mut: %6.3g\n',i)
end

% Output the results and make a plot
subplot(2,1,1)
semilogy(Npop,av_gens,'.-')
xlabel('Population size')
ylabel('Number of generations')
subplot(2,1,2)
errorbar(Npop,av_gens.*Npop,av_err,'.-')
xlabel('Population size')
ylabel('Number of guesses')
xlim([0 100])
```

5 Other things to try

The solutions given here illustrate only a very simple implementation of a genetic algorithm. Here are some things that would be instructive to experiment with.

1. The GA should converge a good deal faster if we don't throw away the best solutions in each generation—the current implementation replaces all the parents no matter how good or bad they are. Modify the

`ga_minimize.m` function so that the best of the parents survive in each generation. Add a parameter to the GA called the survival rate, r_{surv}, that dictates what percentage of the population survives from one generation to the next. Systematically test the convergence vs. the survival rate.

2. Will the GA converge faster with a 2-point crossover mating? Or with a uniform (i.e., N-point) crossover mating? Modify the current code to use a different mating scheme and see if it converges faster.

3. Try modifying the mating algorithm to mate 3 parents instead of 2.

4. Modify the selection algorithm for parents. Try a rank-based system (see `http://cs.felk.cvut.cz/~xobitko/ga/selection.html`).

5. Try using the GA for large lattices (bigger than you can enumerate directly, say $N \sim 50$). Does it get stuck in local minima? How do you know, or can you know? Can you think of a way to avoid this?

6. Included in the solution codes is a file `simulated_annealing.m` that solves the our minimization problem using a simulated annealing approach. How effecient is it relative to the GA? Does it depend on systems size? Which is more robust for large system sizes? Can you explain why?

7. Modify the model so that we that the number of particles is fixed. The search space is smaller now but constructing new guesses is more complicated since the particle number must be conserved.

If you think of other things to try, please e-mail me and let me know. Let me know if you find problems in your own research that you can apply genetic algorithms too. Thanks for all your feedback during the labs and lectures.

6 Conclusions

Please feel free to contact me if you have any questions about the solution codes. If I can help in any other way, please let me know. Any of the references mentioned in my lectures or the labs where I am an author can be downloaded from my homepage: `http://www.physics.nau.edu/~hart/vita.html`

References

[1] Gus L. W. Hart, Volker Blum, Michael J. Walorksi, and Alex Zunger "Evolutionary approach for determination of first-principles hamiltonians," Nature Materials **4**, 391–394 (01 May 2005). Also see references.

[2] Volker Blum, Gus L. W. Hart, Michael Walorski, and Alex Zunger, "Using genetic algorithms to develop model Hamiltonians: Applications to the generalized Ising model," Phys. Rev. B (to appear). Also see references.

[3] Zbigniew Michalewicz and David B. Fogel, *How to solve it: Modern Heuristics* Springer (2004, 2nd Ed.). I first encountered genetic algorithms while reading this book.

[4] I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme und Prinzipien der biologischen Evolution*, Frommann-Holzboog, Stuttgart, 1973. This work was the start of it all. I think there is an English translation.

[5] J. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, Cambridge MA (1975).

[6] John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge MA (1992).