Design," *International Conference on Genetic Algorithms*, June 1993, pp. 408-415.

[5] E. Falkenauer, "A Genetic Algorithm for Grouping," *Proceedings of the 5th International Symposium on Applied Stokastic Models and Data Analysis*, Granada, Spain, April 1991, pp. 23-26.

[6] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wessley Publishing Company, Inc. 1989.

[7] D. E. Goldberg, "Dynamic system control using rule learning and genetic algorithms," *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Vol 1, 1985, pp. 588-592.

[8] J. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[9] C. L. Karr and D. E. Goldberg, "Genetic Algorithm Based Design of an Air-Injected Hydrocyclone", *Control 90, Mineral and Metallurfical Processing*, 1990, pp. 265-272.

[10] R. M. Kling and P. Banerjee, "ESP: Placement by Simulated Evolution," *IEEE Transactions on CAD*, Vol 8, no. 3, March 1989, pp 245-256.

[11] R. M. Kling and P. Banerjee, "Empirical and Theoretical Studies of the Simulated Evolution Method Applied to Standard Cell Placement," *IEEE Transactions on CAD*, Vol. 10, no 10, October 1991.

[12] W. F. Punch, et al, "A genetic algorithm approach to design of advanced composite material structures", MSU Reference paper, 1993.

[13] W. Punch, E. Goodman, M. Pei, L. Chai-Shun, P. Hovland and R. Enbody, "Intelligent Clustering of High - Dimensionality Using Genetic Algorithms," *International Conference on Genetic Algorithms*, June 1993, pp. 557-564.

[14] N. N. Schraudolph, J. J. Grefenstette, "A User's Guide to GAucsd 1.4".

[15] W. Siedlecki and J. Sklansky, "A Note on Genetic Algorithms for Large-scale Feature Selection," *Pattern Recognition Letters*, October 1989, pp 335-347.

Construct C10 'molecule' with -123 eV [Top]

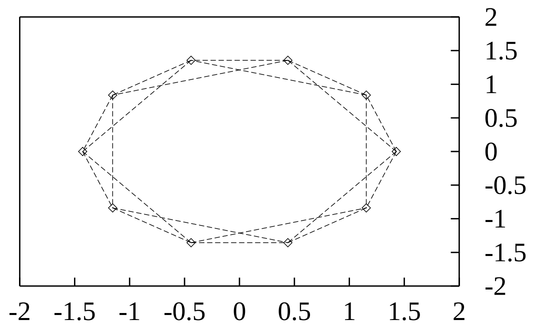

Figure 8c

coded into the genome. A branch-and-bound method was considered, but rejected as too expensive. Also, a left-to-right march was considered. Perhaps one of these approaches are actually worth attempting.

Different parallel structures could be attempted. As the project concludes, colleagues are trying their injection architecture. The results so far are very encouraging.

Another improvement involves a complete redesign of the representation scheme. Some colleagues are working with a forced symmetry representations which yielded energies in the neighborhood of the ones we received.

Bibliography

[1] E. R. Altman, V. K. Agarwal and G. R. Gao, "A Novel Methodology Using Genetic Algorithms for the Design of Caches and Cache Replacement Policy," *International Conference on Genetic Algorithms*, June 1993, pp 392-399.

[2] J. P. Cohoon, S. U. Hegde, W. N. Martin and D. S. Richards, "Distributed Genetic Algorithms for Floor Plan Design Problem," *IEEE Transactions on CAD*. Vol. 10, no. 4, April 1991, pp. 483-492.

[3] J. P. Cohoon, W. D. Paris, "Genetic Placement," *IEEE Transactions on CAD*, Vol 6, no. 6, November 1987, pp. 956-964.

[4] L. Davis, D. Orvosh, A. Cox and Y. Qiu, "A Genetic Algorithm for Survivable Network

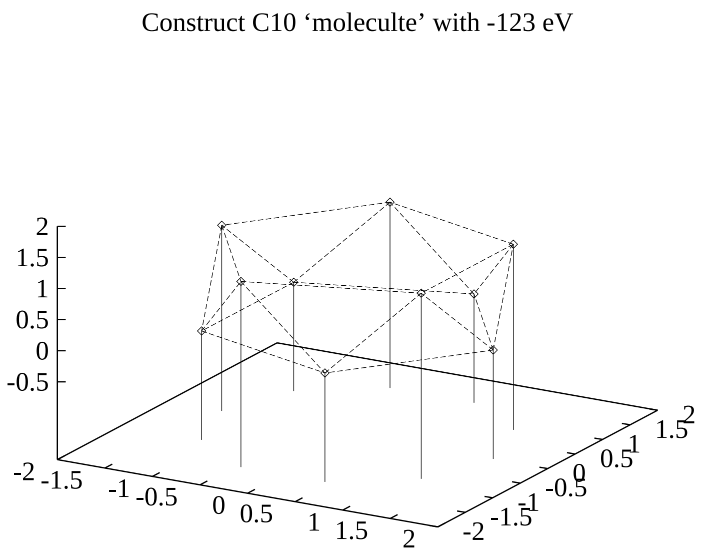

Figure 8a: a hand constructed C10 molecule

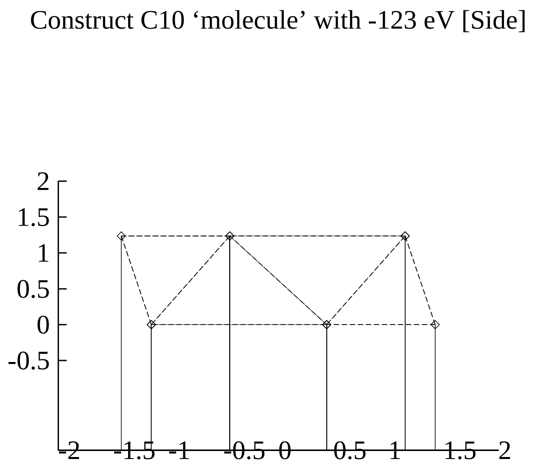

Figure 8b

and then doubled back to finish the structure. This is similar to a depth first spanning tree. Possibly a breadth first spanning tree would work better. Or perhaps a minimum spanning tree.

A reshuffling operator that works with relative coordinates seems too computationally expensive to be called often enough to help. We must make sure that the new relative encoding can be reen-

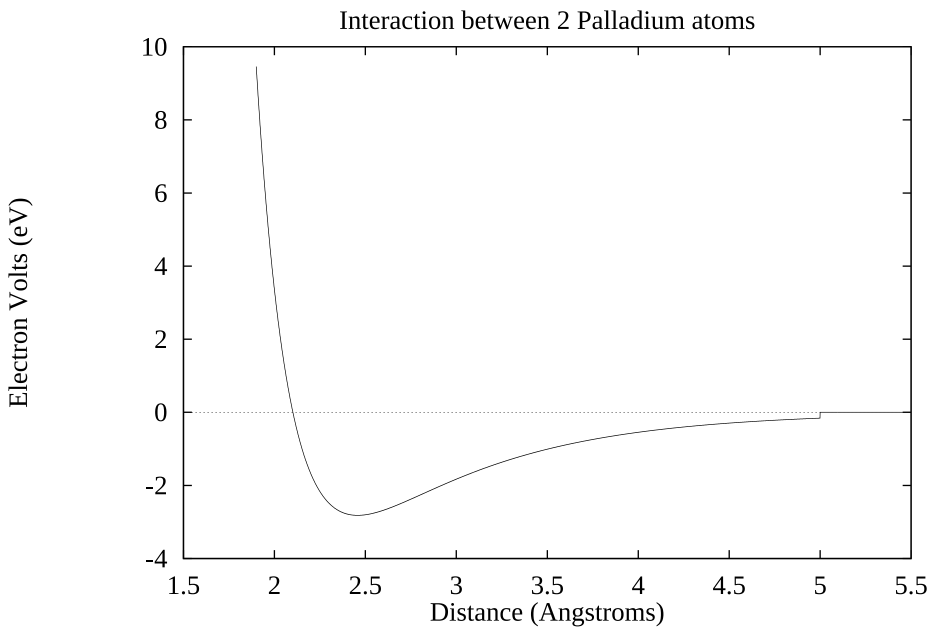

Figure 7b: 2 Palladium energy curve, zoomed in to show some detail

So far, we have tried a single population of modest size (-17 eV), a single large population (-19 eV), an island architecture with few nodes and modest populations (-21 eV), and an island architecture with many nodes and large populations (-24 eV and still improving when it terminated), all without reshuffling. The experts informed us that we should expect -34 eV, so there is plenty of room for improvement.

5. Conclusion

The work presented here was highly successful, for we found a previously unknown flaw in the carbon evaluation code. However, we did not found the lowest possible configuration; inspired by our results, we were able to hand construct a C_{10} molecule with -123 eV binding energy, pictured in Figure 8. However, the energy curve changes drastically with small variations on this molecule, so it may be that the GA would not find it. Work continues with the palladium code.

6. Improvements

We are irresistibly drawn to the following question: How can we improve on the GA? We could try using a polar coordinates system, instead of a cartesian one. Cartesian coordinates have the property of only moving atoms along the cartesian axes; polar coordinates may move an atom in planer circles or along the radius vector in 3D. This variation may help the convergence, but there is no domain advantage of one over the other.

We could also find an alternative reshuffling operator. Our reshuffling operator made long chains population, relative cartesian encoding, low mutation, and modest crossover, produced a -50.4 eV configuration.

4.2.6. Island architecture with reshuffling

The next step was to see if reshuffling will work well with the Island architecture. We set up the experiment as follows: 1st atom hardcoded at the origin, all others encoded as absolute cartesian coordinates, 10 islands with immigration in a ring fashion every 15 generations, population of 250 genomes on each island, low mutation, high crossover, and inverting every 15 generations. The answer: YES! We received energies around -92.4 eV. (Diamond energies range from -70 to -80 eV.)

4.2.7. The hammer falls

The physics experts believed this configuration should not have such a low energy, so they fed the -92.4 eV molecule into their complete evaluator (perhaps we were not using their evaluator correctly?) and receive the same energy. They concluded that the evaluation code must contain an unknown flaw, one that must be fixed before further useful experiments can be performed.

4.3. Palladium

With the flaws discovered in the carbon version of the large-molecule evaluator, we moved to another, better tested evaluation function: one for palladium. The early stages repeated the carbon efforts: extracting the evaluation code from a program designed to do more than return just the binding energy. We were quickly ready to try some experiments. The two palladium atom energy curve is less exciting than the two carbon energy curve, as shown in the figure below.

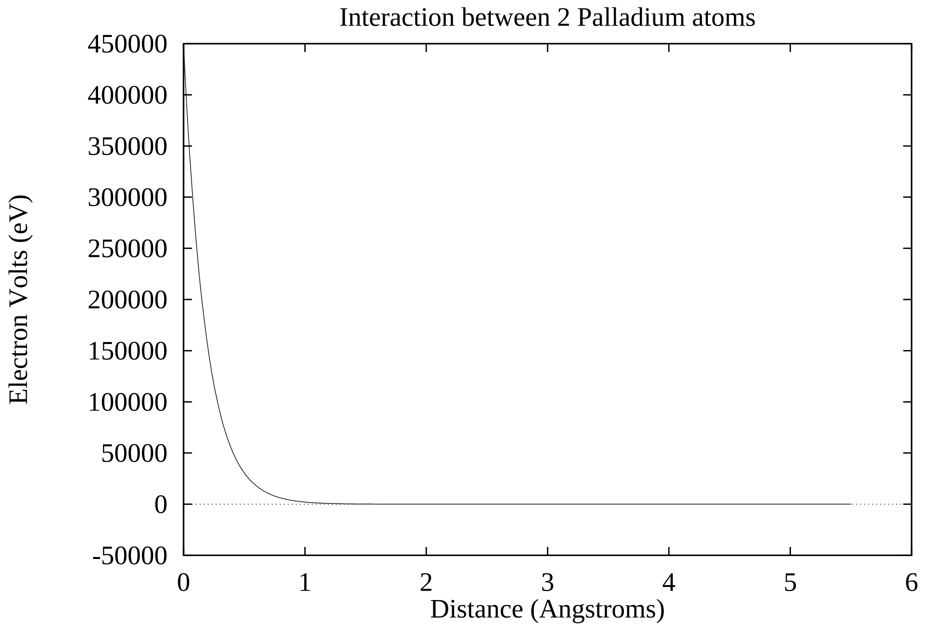

Figure 7a: 2 Palladium energy curve

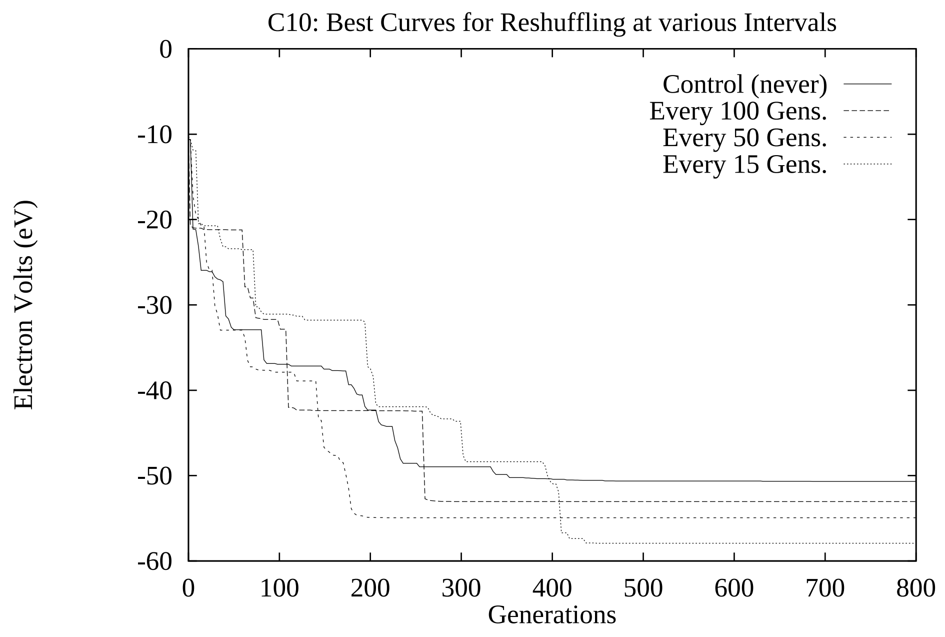

Figure 5

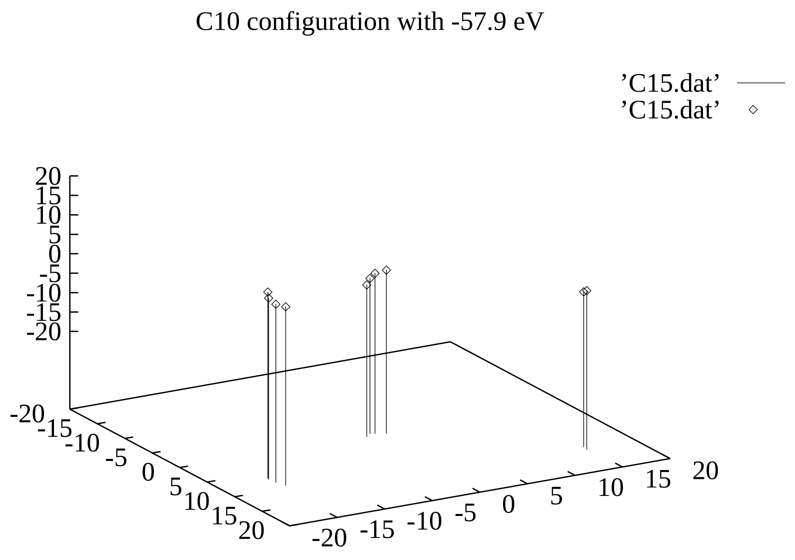

Figure 6

from one island to another [12]. We attempted such an experiment with a five node ring, a modestly sized population per node, modest crossover, low mutation, and no reshuffling. All nodes settled into the same basic configuration with -42.4 eV. Compared with a control experiment (a single large population, absolute cartesian encoding, low mutation, and modest crossover producing -54.9 eV), the results are rather disappointing. Another control experiment with a single large

```
    DEH
```

where the genome is `ABCDEFGHIJ`. When `D` moves toward `B`, the `BCIJ` and `AFG` structures are dismantled. So, while we are moving the `DEH` structure toward the `BCIJ` structure (with the hopes of forming a `BCDEHIJ` structure), we are actually destroying the other structures.

4.2.4. Reshuffle operator

As mentioned in section 4.2.2, if two atoms are swapped on the genome, the genomes are two different genomes, but the molecules are identical. That is because a genome is broken into sections that encode the positions of the different atoms. When we used absolute coordinates, it is irrelevant whether the atom is the first or last atom encoded on the genome; the only things that matter are the coordinates. Unfortunately, the GA engine doesn't know that. Therefore, we needed an operator to sort the atoms in a regular fashion. Moreover, we wanted to sort them in such a way that will keep atoms that are close to each other in the molecule near each other on the genome, thus helping to preserve their closeness during a crossover operation. We called this operator reshuffling.

Reshuffling works like this: every (parametric) number of generations, go through all genomes and place those atoms that are close together in the molecule near each other in the genome. Specifically, start at the atom which is hardcoded at the origin and find its nearest neighbor, that atom becomes atom #1. Now find atom #1's nearest neighbor, that atom becomes atom #2. And so on, picking atoms which are not already reencoded. This process produces a chain of atoms which are in the same structure. When we run into a dead end, i.e. all the neighbors of the i-th atom are already reencoded, we break the chain by back-tracking along the chain until we find an atom that has a non-reencoded neighbor, then continue the process. Eventually, we will be back to the origin atom. If there are atoms that are not reencoded, they must be part of a separate structure. A non-reencoded atom is arbitrarily selected, and that structure is reencoded.

One major advantage of the reshuffle is that it helps keep atoms that are forming a good structure together on the genome. That way, a crossover is less likely to break the bonds in the new genome, since the probability of a crossover point appearing between them is much less when they are contiguous on the genome. The other advantage is that it normalizes many of the permutations, since the old nearest neighbor may be anywhere on the genome, now it will be the next atom.

A preliminary experiment was set up, using absolute cartesian encoding. The only freedom restriction was the translational restriction: i.e. the first atom was hardcoded as the origin. The result indicated that the more often we used the reshuffle operator, the better our results. A control experiment yielded -50.6 eV; the best reshuffling, -57.9 eV, as shown in the figure 5. Reshuffling helped to form larger groups, but they were still isolated structures, as shown below.

4.2.5. Island populations (ring immigration) vs. a large single population

Improvements in GAs can appear in experiments with a number of isolated island populations, each searching individually for a solution, and at regular intervals, the elite genome is copied

(0,0,0), (1,0,0), (1,1,0), (1,1,1)

Now if we change the genome to:

(0,0,0), (1,1,0), (0,1,0), (0,0,1)

we get these absolute cartesian coordinates for the atoms:

(0,0,0), (1,1,0), (1,2,0), (1,2,1)

The rest of the genome is moved with the second atom. From these experiments, we received energies of around -52 eV, a slight increase and improving further when the GA with modest population, modest crossover, and low mutation terminated.

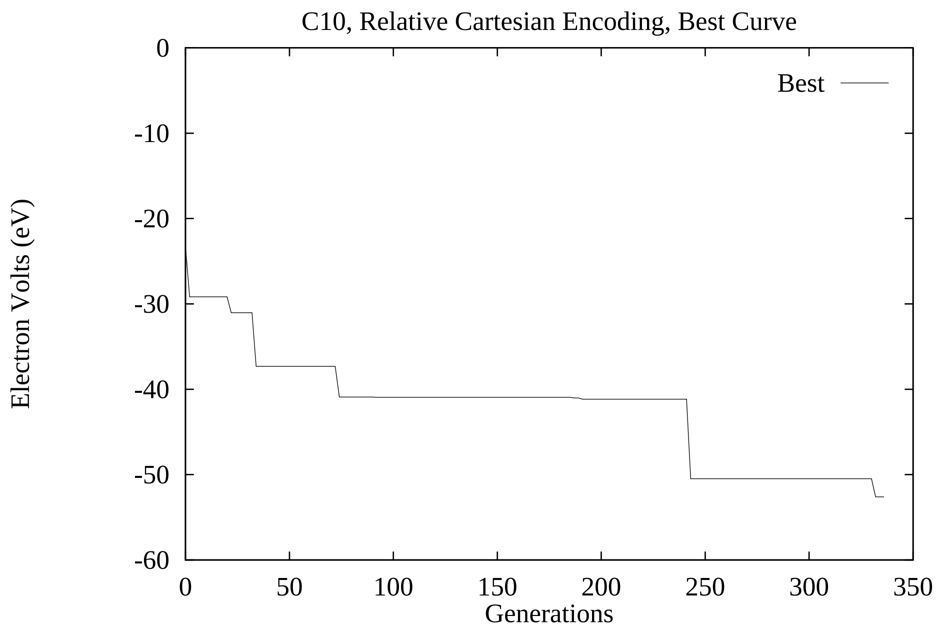

Figure 4

Even though the performance is better, the GA still converged to isolated groups. The trouble here seems to be that the atoms are not together in the genome, e.g. the first atom could bond with the last atom on the genome. This spread has two unfortunate effects. Crossover is likely to destroy the bond for the next generations, since the likelihood of a crossover point coming between the atoms on the genome is much greater when the encoding segments are distant. Further, when the first atom of a structure is moved, it may be separating other structures. For example let's consider this atom grouping:

```
AFG

BCIJ
```

Another problem is permutations of atoms in the genome. When two atoms swap their encoded positions in the genome (but the coordinates are still the same), the GA sees the genomes as two different ones, but the molecules are identical. For example, these two sequences are the same molecule with the first and second atoms permuted:

(0,0,0), (1,1,1), (0,1,0), (1,0,0), (0,0,1)

(1,1,1), (0,0,0), (0,1,0), (1,0,0), (0,0,1)

Again, the GA sees the encoded sequences as two different genomes. Therefore, permutations have the same hampering effect as translations, rotations, and mirror images.

To eliminate the freedom of translation, rotation, and mirror images from the GA, we remove some of the coordinates from the genome, hardcoding the positions. Placing the first atom on the origin normalizes the translation, but the molecule is still free to rotate. Placing the second atom on the positive X axis limits the molecule to rotations around the X axis. Placing the third atom on the positive Y half-plane, as X ranges freely and Z equals 0, eliminates all rotational freedom. Finally, placing the fourth atom on the positive Z half volume, as X and Y range freely, eliminates mirror images. We address the issue of restricting the permutations in section 4.2.4.

The best result from several runs came from an experiment with a modestly sized population, high crossover rate, low mutation rate, yielding a binding energy of -50.7 eV, yet still forming isolated groupings.

The absolute cartesian representation has the empirical downfall of forming isolated pairs, and rarely triples, of carbon atoms. That is to say, when bonds do form, they usually include only two atoms, and occasionally three atoms. (The other atoms are too distant to form bonds.) The GA then minimizes the energies of the pairs; it does not bring the molecule together. Upon reflection, we see that there is no evolutionary pressure to bring the structures together, since the energy function is the same whether the groups are a few or several angstroms apart. Moreover, a mutation changes only a single bit and therefore a single component of a single atom. Also, crossover exchanges larger portions of the genome, leaving the majority of the atoms in the same locations and changing the component of the atom which happens to be encoded at the crossover point. Essentially, if an atom moves away from a bonding structure, the resulting genome is weaker, unless a happy coincidence places it in another bonding group.

4.2.3. Absolute vs. relative cartesian coordinates

To overcome the deficiency discovered with absolute cartesian coordinates, we tried a relative cartesian representation, where each set of coordinates is relative to the previous. This has the advantage of changing a single atom and moving it and the rest of the genome. For example, if we have this relative cartesian genome:

(0,0,0), (1,0,0), (0,1,0), (0,0,1)

we would have these absolute cartesian coordinates for the atoms:

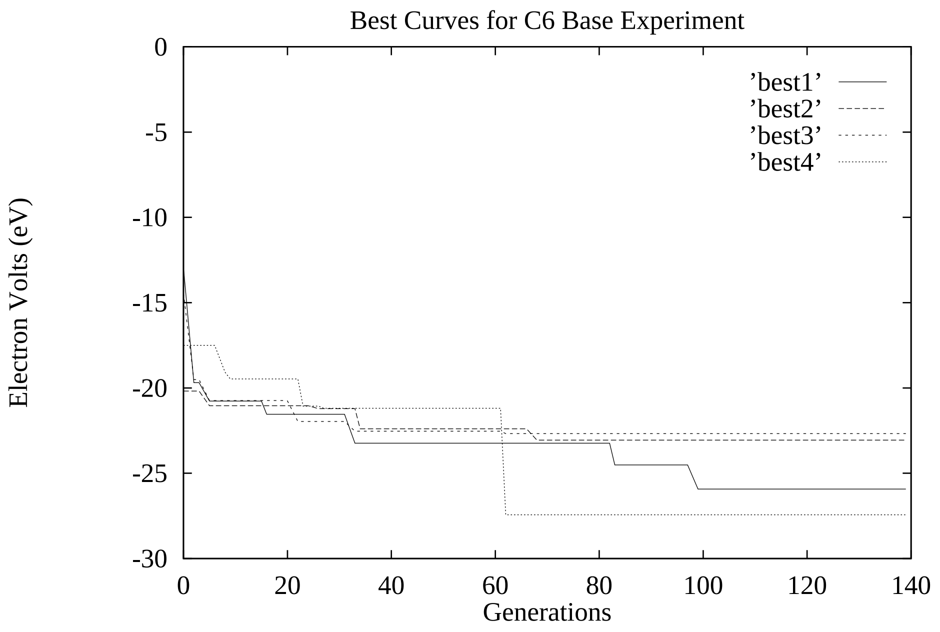

Figure 2

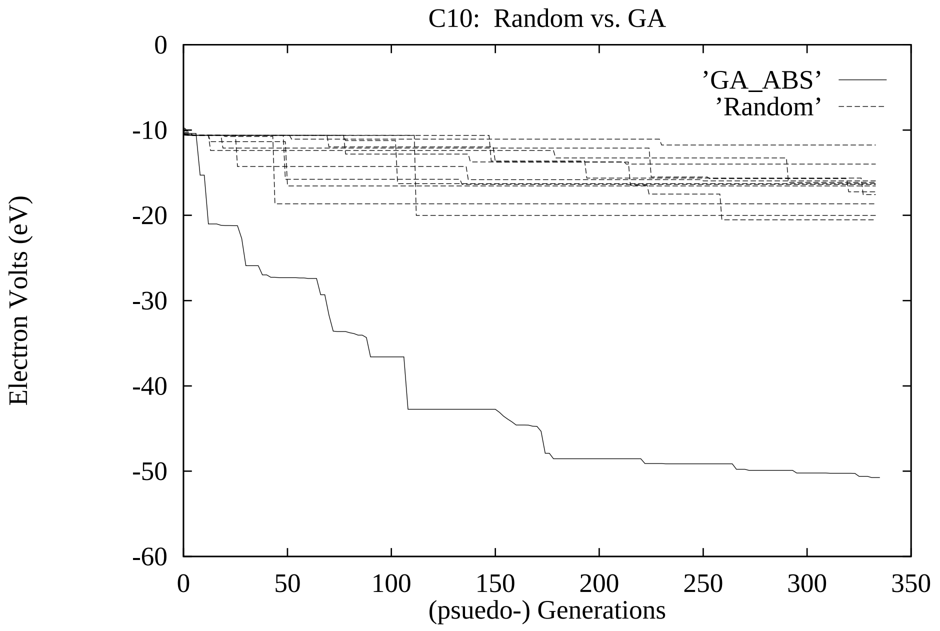

Figure 3

(1,-2,0), (1,0,0), (1,2,0), etc.

To the GA, all these identical molecules are completely different genomes. Therefore, unless the GA has a way to normalize the genome, they will fight each other for convergence.

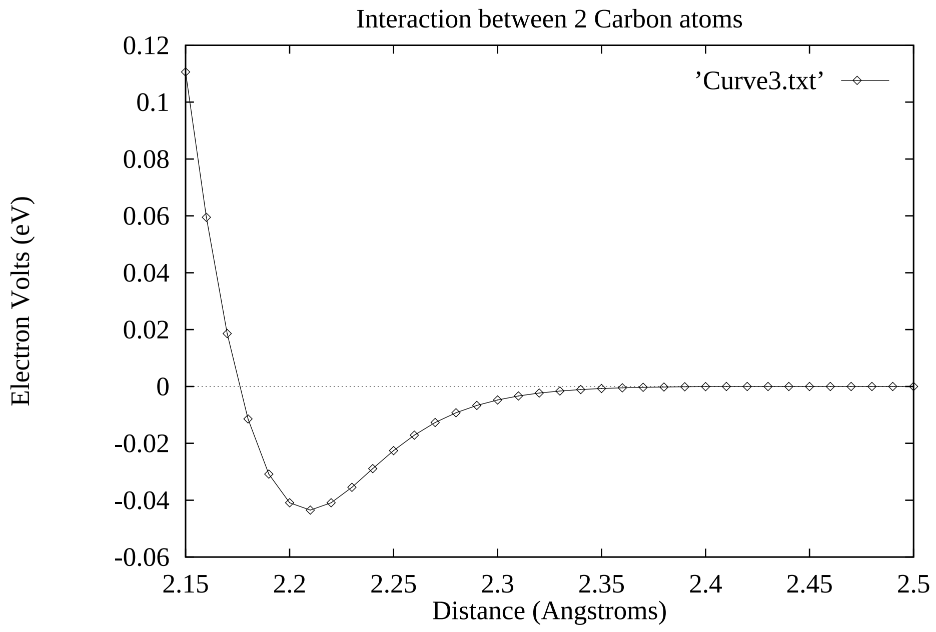

Figure 1c: Interaction between 2 Carbon atoms, zoomed in to reveal a second local minimum.

We attempted several six atom runs for some quick trials to give us a feel for the performance we can expect. The genome length was 6 x 3 x 8 bits long: 6 atoms, 3 cartesian components per atom, 8 bits per component. We received an average "Best" energy of about -25 eV, with a best "Best" of -27.4 eV (see Figure 2). This is a very encouraging result. We expected the GA to find energies similar to diamond at -7 to -8 eV per atom, so the GA should be able to find energies in the -42 to -48 eV range for the C6 case.

So how did the GA perform? To answer this question, we generated several random configurations, reporting the best random configuration found to date at regular intervals, producing a "Best" pseudo-curve. Comparing the GA's "Best" curve to this curve, one can see that the GA and the random configuration start with roughly the same energy, which is to be expected since the GA starts with random configurations. However, the GA rapidly improves the energy, a behavior the random curves do not exhibit.

4.2.2. Restricting the degrees of freedom

When all atoms are allowed to move freely, independent of each other, many duplicate molecules may be formed, differing only in rotation or translation; mirror images may also be created. For example, suppose we have a carbon molecule that forms a straight chain, starting at the origin and heading out along the x axis. That is:

(0,0,0), (2,0,0), (4,0,0), etc.

A translated and rotated, but essentially identical, molecule could be the following:

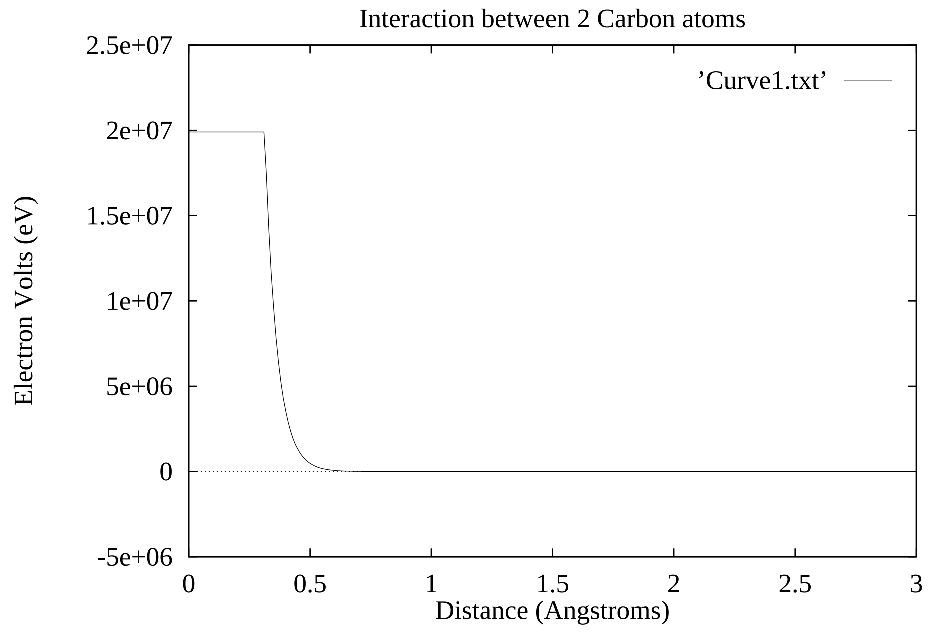

Figure 1a: Interactions between 2 Carbon atoms.

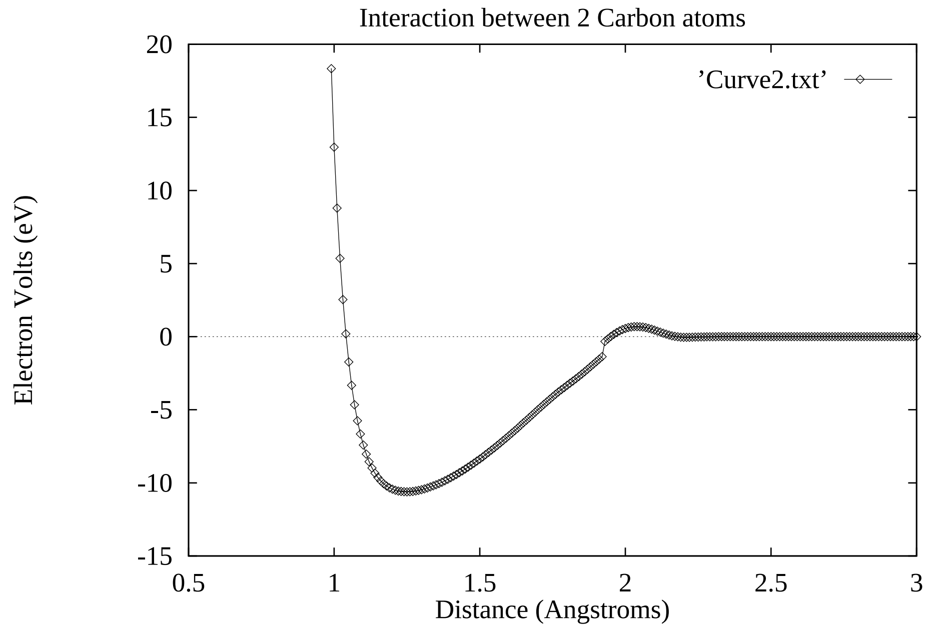

Figure 1b: Interaction between 2 Carbon atoms,
zoomed in to show some details

range arbitrarily selected as +/-40.0 angstroms, using 8 bits to represent the range. (The range fell naturally from the way the values were passed to the evaluation code. It also allows the atoms to line up in a straight line and remain within the range.) For example, the first atom has three components, and each component may have values between -40 to +40.

student, Seong Gon Kim, provided us with a Fortran function that returns the binding energy of a configuration of atoms represented by their cartesian coordinates; the smaller (or more negative) the binding energy, the stronger the bonds.

4.1 Local developments

During the course of the this project, it has been necessary to modify parts of the GA engine. Parallel architectures were added by Shyh-Chang Lin [12] that provided linear speedup and increased search space. Shyh-Chang Lin modified GAucsd to implement an island population architecture which forced the elite genome of one island to be copied to another island in a ring fashion: island #1's elite is copied to island #2; #2's to #3; etc. Each island is otherwise isolated from each other. The interval between copies is a parameterized number of generations. Also, we introduced a new operator, called reshuffling. We discuss the details and advantages of each modification below.

4.2. Carbon

We began the project by working with Carbon parameters, hoping to find a low energy state for a C_{60} molecule, the popular buckyball. However, before that could be done, we needed to find an encoding scheme with which the GA could work. Working with a C_{60} would be too cumbersome to evaluate the performance of our encoding scheme; therefore we started with C_6 and moved quickly to C_{10} molecules, for they are relatively well-studied, forming chains and rings for their low energy states. C_{10} molecules are at the threshold of becoming interesting, beginning to leave the realm of chains and rings for more three dimensional structures. We discuss the evolution of the representation for the experiments below.

4.2.1. Starting the GA

The first task at hand was to interface the C GAucsd engine with the Fortran evaluation function with as little modification as possible. The Fortran code was not originally written to accept random configuration after random configuration, rather it was written to read in a basic structure and evaluation slight perturbations of that structure. We quickly received a rewrite capable of testing many random configurations.

The energy potential between two atoms is shown in the figure below. The sudden jump in the energy at 1.92 angstroms has been attributed to an artifact in the evaluation function: we are using two atoms in a function written for a large number of atoms.

With such an curve, one would be tempted to use a table lookup for the interaction between every pair of atoms. However, such an approach would be unwise; the energy is more dependent upon the number of atoms interacting. For example, if three atoms are close enough to interact with each other, the real energy is different than the sum of the three individual table entries of a two-atom curve.

We were then ready for our first trial run, but what encoding to use? This first thing that came to mind was simply encoding a three dimensional cartesian coordinate system, i.e. an X, Y, and Z position for each atom. We allowed the components to range how ever they wished, within a

2.5.3 Elitist policy

GAucsd provides a mechanism for guaranteeing at least one exact copy of the best genome found so far, or the elite genome, will survive into the next generation. After all other operations, copying, mutation, and crossover, GAucsd checks for the existence of a copy of the elite genome. If a copy is not found, a genome is arbitrarily selected to be replaced by the elite genome. The elitist policy protects against losing the best solution and needing to re-evolve it.

3. Buckyballs, fullerenes, and other large molecules

Fullerenes are large carbon molecules named after Buckminster Fuller, inventor of the geodesic dome. A special and very popular fullerene, the C_{60} molecule, is more commonly known as the Buckyball; other popular fullerenes include the C_{70} and C_{240} molecules. Buckyballs have grown in popularity over the past years; models of fullerenes run daily in simulators to determine their various properties. One of these properties is the configuration which yields the lowest binding energy. Researchers working with fullerenes are unsure if the accepted low energy configurations are indeed the lowest possible. Verifying the configuration has been a difficult task, since there are so many degrees of freedom. Fullerenes are known to appear in many shapes: chains, rings, capes, cages, cages within cages, buckytubes, and, of course, buckyballs (the geodesic dome shape for which they all receive their association with Buckminster Fuller). We are about to use the power of a GA to find low energy configurations, and hopefully solve that problem.

3.1 Why use GA for finding a buckyball?

One of many downfalls of simulating fullerenes and other large molecules is the dimensionality of the search space. For a Buckyball, there are 60 atoms, each with 3 dimensions, giving 180 separate values. We can cut down on some of them: 3 for translation normalization, 3 more for rotation normalization, and 1 more for mirror images. However, there is still the possibility of swapping the positions individual atoms. For example, if two atoms swapped their positions in the simulation, the molecule would be identical, but the model would think it has found another configuration. (An example appears below that will explain the swapping problem in more detail.) As a result of such large dimensionality, finding a low energy configuration of such a large number of free variables perplexes many techniques. Genetic Algorithms perform better than other methods, such as gradient descent, because GAs use the heuristic of evolutionary pressures to modify a population, rather than working with a single configuration and modifying it according to stress pressures, which is the way gradient descent methods operate. A population cuts down on the search space by quickly removing impossible and unlikely configurations.

4. Experiments

GAucsd has been developed in C over many years, adding new features here and there. The evaluation function must be supplied by the experimenter, so GAucsd provides several ways to link the function with the engine, the most friendly of which uses an awk script to take a C file with a special comment line. The awk script expands that comment line into more C code that will decode a genome into the parameters that the evaluation function accepts. Dr. Tomanek and his

2.5 Some GAucsd details

As mentioned earlier, we used GAucsd, which has some special options and properties worth mentioning. First, all genomes are the same fixed length, so variable length genomes are impossible. Real and integer valued parameters to the evaluation function may be encoded with gray codes to avoid Hamming Cliffs (discussed in section 2.5.1). Moreover, if a complete initial population is not supplied, the rest are filled with truly random genomes or genomes from a “super-uniform” random generator (discussed in section 2.5.2). Further, GAucsd provides an Elist Policy (discussed in section 2.5.3) to protect the best genome from destruction.

2.5.1 Gray codes and evaluations parameter scaling

Real and integer valued parameters are encoded into evenly-sized segments, represented as integers scaled over the appropriate range. The number of segments depends on the number of bits in the genome allocated to the parameter. That is, if GAucsd is told to use six bits to encode a parameter, there are 2^6 (64) equally sized segments. However, there is a catch: Hamming Cliffs. Normally a computer stores integers in their binary versions: 0 is 000, 1 is 001, 2 is 010, 4 is 100, etc. With gray codes, all numerically adjacent numbers differ in exactly one bit: 0 is 000, 1 is 001, 2 is 011, etc. If the parameter is encoded as binary, changing from 31 to 32 is a large jump for the GA. 31 has a binary code 011111, and 32 has a binary code of 100000. That’s a difference in all six bits; that difference is a Hamming Cliff. The gray code for 31 is 010000, and the gray code for 32 is 110000. So there are no Hamming Cliffs with gray codes.

An execution-time flag allows GAucsd to add a random fractional amount to the encoded values before scaling to the parameter’s range. For example, if an encoded parameter is 123, it might become 123.456. The fraction is uniformly distributed over the interval [0,1). The encoded parameter is then linearly scaled from the encoded range to the parameter’s range.

2.5.2 Super-uniform initial populations

GAucsd includes a super-uniform random generator to seed the initial population. The super-uniform generator create genomes in such a way that “all schemata up to a certain defining length (limited by the population size) are equally represented.” [14,20] To quote GAucsd’s User’s Guide:

> In crossover-dominated GAs (i.e. those with low mutation rate) this eliminates the risk of pathological initial populations in which an important low-order schema just happens to be missing, and has to be created by an unlikely mutation event. [A start-up option] uses a reduced-variance stochastic algorithm which produces a population with no local, but large global correlations. Crossover is very effective in destroying such long-range correlations, but this option should not be used in mutation-dominated GAs with very low crossover rates. [14,20]

GAucsd can also generate “truly” random genomes, if desired.

GA is converging. By changing a single allele, a new genome is created and convergence is slowed while the GA searches the neighborhood. Each allele has a probability to mutate in the process of being copied from one generation to the next. A large probability usually kills all hope of finding and keeping good solutions. A small probability allows for variability to keep convergence at bay for little longer. While a very small probability is next to pointless. (Empirical evidence suggests a mutation rate sufficiently small to generate on average of a single mutate every two or three genomes works the best.)

2.4 Crossover

Crossover is the main evolutionary mechanism to generate new genomes from the healthy genomes. Pairs are selected at random and mated; their genomes spliced together. A point is selected at random and the alleles after that point exchanged to create two new genomes. Sometimes a genome is viewed as a circular beasty, therefore two points are selected at random and the alleles between those points are exchanged. The two parents are replaced by the new genomes. For example, suppose we have selected these two genomes for crossover:

Genome #1: `1234567`

Genome #2: `ABCDEFG`

If point 3 was the randomly selected point for the crossover, the new genomes would look like the following:

Genome #1: `123DEFG`

Genome #2: `ABC4567`

Crossover provides a way to evolve genome combinations (or schema) that were missing from a population. For example, suppose we have an extremely small population made entirely of these 2 genomes:

```
0001100

0011000
```

and they mate at point 4 to produce these:

```
0001000

0011100
```

which were not present in the original population. One of these new genomes could be the optimal. Without crossover, we would not have found it, except by a chance mutation.

limit.

2.2 Selection of genomes for copying

Each generation, the genomes are evaluated by the evaluation function and the result recorded. The genomes which performed well enough are selected to "survive" via copies of itself in the next generation; often the population of one generation represents the proportional success of the "healthy" genomes in its parent generation. That is, if a certain genome was proportionately twice as fit as another, the first will have twice as many copies (or children) as the second, on the average.

There are two common techniques for finding the proportional fitness, and they differ only in the way they select a baseline fitness. The two techniques are window scaling and sigma scaling. In window scaling, the baseline fitness is the worst fitness seen in a (parametric) window of generations, which may be all of them. The baseline for sigma scaling is calculated by adding a (parametric) multiple of the generation's fitness variance to the generation's average fitness. (GAs may maximize or minimize an evaluation function. Here we discuss minimization; maximizing followings similar rules.) The population fitness is computed as follows:

$$F_p \leftarrow \sum_{g_i \in \text{ population}} \begin{cases} 0 & : f(g_i) \geq F_b \\ F_b - f(g_i) & : \text{ otherwise} \end{cases}$$

where F_p is the fitness for the population; F_b, the baseline fitness; g_i, the i-th genome of the population; $f(g_i)$, the fitness of g_i as returned by the evaluation function. The number of copies of the "healthy" genomes ($f(g_i) < F_b$) is as follows:

$$\frac{f(g_i)}{F_p} \times \text{PopSize}$$

where PopSize is the population size, i.e. the number of genomes in the population. Unhealthy genomes have no copies. Window scaling has the unfortunate property of losing selective pressure when a fatal genome exists in the window. This property is also true for sigma scaling when the average fitness is far from the good genomes.

2.3 Mutation

Mutation can be useful to search the near neighborhood of a good solution, especially when the in the hope of producing even better candidates. Other evolutionary mechanisms, such as mutation, are thrown in to aid the GA.

Genetic Algorithms have been applied to a wide variety of applications, including floor design [2,3], feature selection [15], cache design [1], and network design [4]. In this application, we will use a GA to solve the problem of finding configurations of large molecules, like the Buckyball, with low binding energy.

2. An overview of Genetic Algorithms

As mentioned earlier, a GA is an optimizer which tries to use evolutionary pressures to form the next batch of solution candidates. The rudimentary element of a GA is the allele, or bit. Alleles are grouped in sequences called genomes; others names include gene, chromosome, and string. A genome represents a complete solution candidate, while a section of the genome encodes a parameter sent to the evaluation function, i.e. a group of alleles represent a part of the solution. Each genome is evaluated to find its "health." (The healthier the genome, the better the solution the genome encodes. Healthy could mean higher or lower values, depending on whether the optimizer is trying to maximize or minimize the function.) A batch of genomes is called a population; a population exists for a single generation. Each generation is evaluated, and the next generation formed with the knowledge gained. To form the next generation, good genomes are copied; mutations applied; and pairs mated (a.k.a. crossed-over). Then, like all good optimizers, the process repeats.

The copying process is the most important part, for it is the mechanism by which unhealthy (or bad, weak) genomes are eliminated and healthy (or good, strong) genomes grow to dominate the population.   The proportional health of a genome indicates the proportion of the next generation that the genome will occupy, i.e. healthier genomes spawn more copies than less healthy ones, and the unhealthiest genomes are not copied at all. However, crossover is crucial to forming new genomes, otherwise new genomes must wait for the chance mutation in order to improve. Mutation helps provide variability to keep the GA from growing stagnant. Other operators exist to simulate mechanisms in natural evolution, and to provide variability in the genomes.

The GA checks a few termination conditions to decide if it should stop. These termination conditions include the following: 1) a genome has passed a desired fitness, 2) the population has stabilized enough to make further searching seem pointless, or 3) the number of evaluations has passed a maximum and it's time to try a different approach.

2.1 The GAucsd GA engine

For this project, we used a GA engine available from University of California, San Diego known as GAucsd. GAucsd has a few special details, which will be addressed in section 2.5. This GA engine begins by initializing the population, either with a population supplied by the experimenter or a randomly generated one. After the initial population is generated, GAucsd moves directly to evaluating it. After that, all generations follow this pattern: Selection, Mutation, Crossover, and finally back to Evaluation. This cycle continues until one of the termination conditions is satisfied; some of the conditions may be disabled to force the GA to run to the end of the iteration

Using Genetic Algorithms to Find Low Energy Configurations of Large Molecules.

A Master's Project

Dan Redder

William Punch, Shyh Chang Lin, David Tomanek, Seong Gon Kim

Abstract

This paper reviews a genetic algorithm that shall find configurations of large molecules with low binding energies, as used for a project in the pursuit of a masters degree. We first discuss the concept of a genetic algorithm (GA) as an energy function optimizer. We'll briefly discuss fullerenes and other large molecules. Then we'll describe the experiments and their results. Finally, we'll mention a few improvements that could be done to this application of the GA.

1. Introduction

Since humans became cognitive, we have been driven to find optimal solutions to the problems that plague us. Early human problems ranked with basic survival: how best to move from one copse of trees to the next without drawing the attention of roving predators. Today's problems are motivated more by academic interest or business profit. One of the ways we represent a problem is through an evaluation function, then we try to optimize that evaluation function. The basic design of an optimizer is to generate a batch of solution candidates and test them; generate and test; generate and test; and so on; knowing how to generate the next batch from the knowledge gained from previous batches is the trick of a good optimizer. The most basic method is an exhaustive search, which is usually prohibitively large and time-consuming or even impossible, due to an infinite search space. Other methods include Trial-and-Error, Gradient Descent, Gradient Descent with Simulated Annealing, and Genetic Algorithm (a.k.a. Simulated Evolution).

The trick in a Genetic Algorithm (GA) is to simulate evolutionary pressures when generating the next batch of solution candidates. A Darwinian "survival of the fitness" concept is imposed on the generation of the next batch of solution candidates; that is to say, if a solution candidate is found to be unsuitable when compared to the others in its batch, it will not succeed into the next batch. Moreover, the better a candidate, the more likely it is that it will pass its traits to a larger portion of the next batch of candidates. Good candidates are "mated" with other good candidates