Optimization Methods in HPC

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2013

Introduction

Optimization is an old, old problem:

- For many (if not most) problems of interest, an exhaustive search is not possible (or computationally feasible)
- Without exhaustive search, we need some form of directed search
- Often (have to) settle for “best,” not necessarily optimal, solution.

Classic Optimization Problems

Optimization/Search problems date back to the origins of computer science:

- Traveling salesperson
- 0/1 Knapsack problem
  - pack knapsack with selected objects to maximize value
- *n*-queens problem
  - place n queens on an $n \times n$ chessboard such that queens can not attack one another
- 15- and 8-puzzles
  - numbered tiles on $4 \times 4 (3 \times 3)$ board with one empty place, try to move tiles one-at-a-time to achieve row-major order

Sampling of Other Applications

There are many other application areas the utilize large-scale optimization:

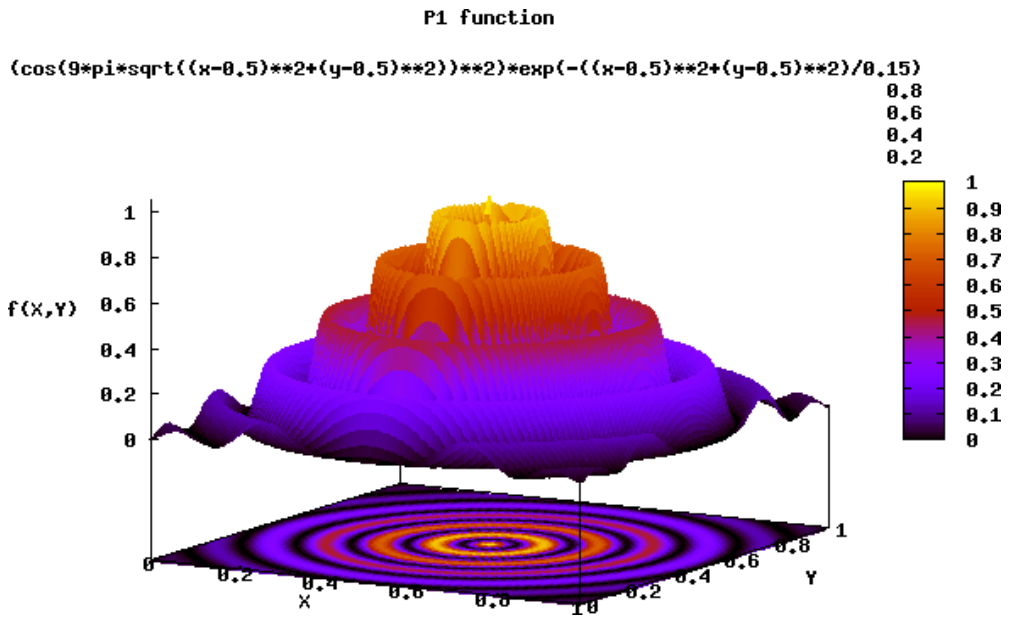
- VLSI (very large scale integration) IC design
- Financial forecasting
- Airline fleet/crew assignments
- Interatomic/Intermolecular potential determination
- Wide array of engineering design problems
- ...

Classes of Optimization Techniques

Broad classes of (global) optimization techniques:

- Newton's Method (& variations) (*covered*)
- Conjugate Gradient (discussed in context of iterative solution to PDEs)
- Branch and bound (*covered*)
- Dynamic Programming
  - makes use of overlapping subproblems, similar to B&B
- Hill Climbing (*covered*)
- Simulated Annealing
  - Similar to stochastic hill climbing and GA
- Genetic Algorithms (*covered*)

Lessons for Global Optimization



- There is no free lunch!
- Luck has nothing to do with it ...
- Global is very much more difficult than local.
- Any global optimization method can be defeated by a sufficiently clever problem.

Newton's Method For Optimization

Straightforward Newton, based on Taylor expansion of multidimensional function (to be minimized) $F(\mathbf{x})$:

$$\mathbf{x}^{n+1} = \mathbf{x}^n - \epsilon[H(F(\mathbf{x}^n))]^{-1}\nabla F(\mathbf{x}^n),$$

where H is the Hessian matrix of second derivatives (hence this is $\mathcal{O}(N^2)$ in storage) and ϵ is a (potential) adjustable parameter.
Quadratic approximation for F.
Quasi-Newton methods involve a host of shortcuts for approximating the inverse Hessian (which is often not a simple thing to calculate, let alone invert)

Gauss-Newton for Nonlinear Least Squares

Gauss-Newton is an iterative procedure for the nonlinear least squares problem, e.g. given M functions of N parameters, minimize

$$R(\mathbf{x}) = \sum_{i=1}^{M} (f_i(\mathbf{x}))^2$$

The Hessian for R is approximated using the Jacobian,

$$H(R) = 2J(\mathbf{f})^T J(\mathbf{f}),$$

such that:

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \left([J(\mathbf{f})]^T J(\mathbf{f}) \right)^{-1} J(\mathbf{f})^T \mathbf{f}(\mathbf{x}^k)$$

Levenberg-Marquardt (Nonlinear Least Squares)

In Levenberg-Marquardt, the increment is given by:

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \left([J(\mathbf{f})]^T J(\mathbf{f}) + \lambda I \right)^{-1} J(\mathbf{f})^T \mathbf{f}$$

where λ is an adjustable (non-negative) parameter (I is the identity matrix), $\lambda \to 0$ recovers Gauss-Newton, while $\lambda >> 1$ becomes gradient (steepest) descent. In practice λ is continually modified during the optimization process.
LM has become a very popular method in nonlinear least-squares (can be found in `GSL`, `MINPACK`, `Mathematica` ...).

K. Levenberg, Quart. Appl. Math. **2**, 164 (1944). D. Marquardt, SIAM J. Appl. Math. **11**, 431 (1963).

Other Quasi-Newton Variations

Basic idea is still Newton - just different approximations for the inverse Hessian. Idea is to start with a positive definite symmetric H^{-1} such that we always move in a "downhill" direction.
Currently best established is BFGS (Broyden, Fletcher, Goldfarb, Shanno, all published separately in 1970):

$$H_{k+1} - H_k = \frac{y_k y_k^T}{y_k^T (x_{k+1} - x_k)} - \frac{H_k (x_{k+1} - x_k) \left[H_k (x_{k+1} - x_k) \right]^T}{(x_{k+1} - x_k)^T H_k (x_{k+1} - x_k)^T}$$

where $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$.
Often quasi-Newton methods are called *variable metric* methods.

Newton Methods in Parallel

Parallelizing the class of Newton-based methods is straightforward, but it can be difficult to achieve a scalable approach:

- Can parallelize the calculation of the residuals in the least-squares sum, most amenable to OpenMP, but could be distributed through MPI as well with sufficient number of terms
- Term limitations impose upper limit on scalability - typically your optimization problem has a fixed number of data points and unknowns over which you are optimizing, so you can not scale the problem as you consider using more processors
- Individual terms in the least-squares sum can be cheap to calculate, leading to load balance problems

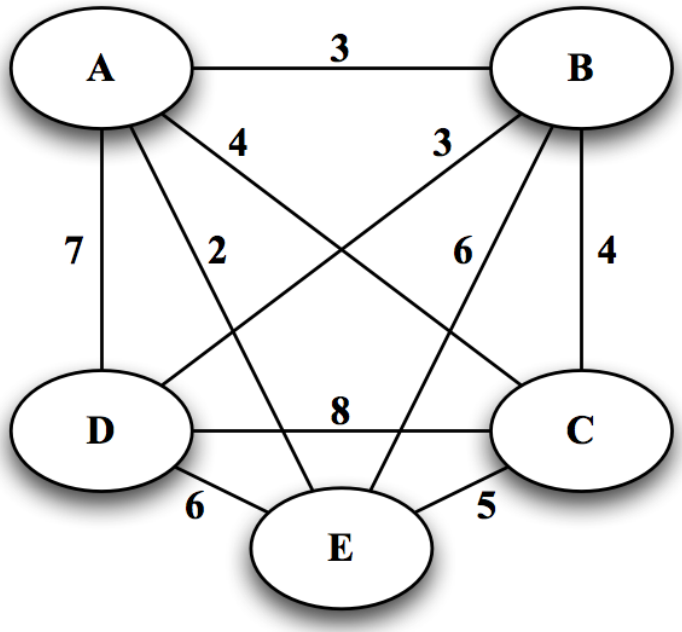
Basic Branch & Bound

The basic apporach in branch and bound (B&B) is very similar to that of *dynamic programming* - the problem is divided into subproblems:

- Subproblems typically are represented by branches in a tree
- **State-space** tree is formed, in which the root represents the starting point
- General case is that of a dynamic tree (counterexample - the knapsack problem consists of choices of only whether or not to include an item, and is a static tree)
- Generally these combinatorial optimizations problems are **NP-complete** or **NP-hard**, i.e. no polynomial expression for the running time (exponential and exhaustive)

Example: Traveling Salesperson

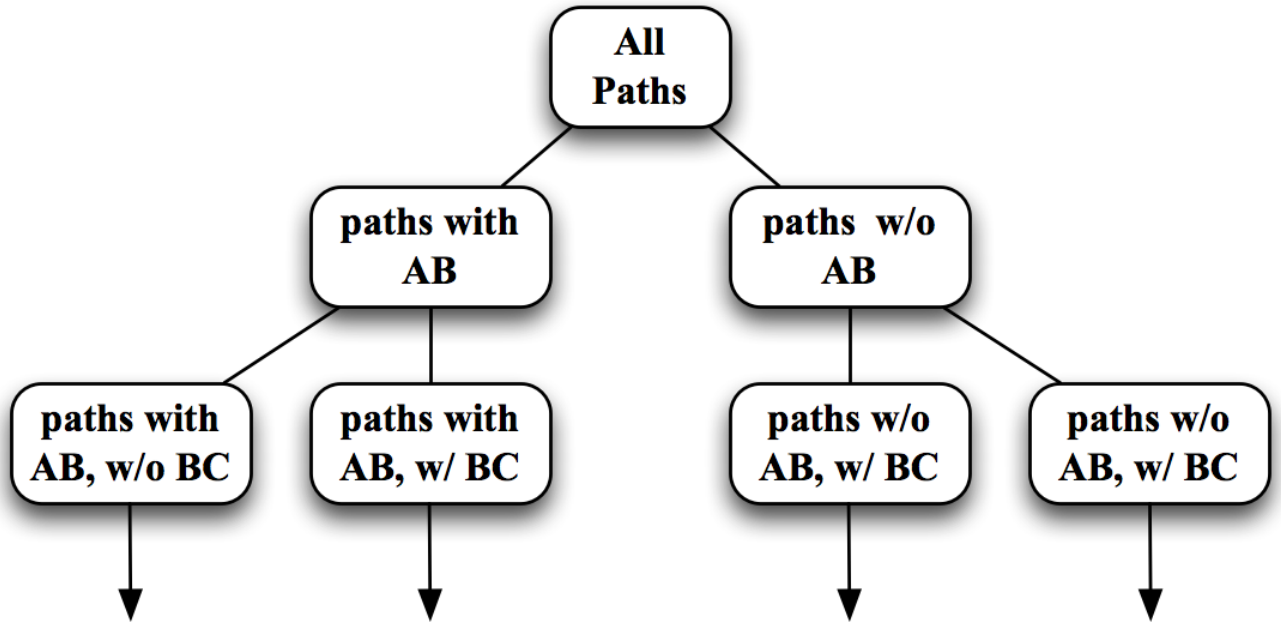
Famous combinatorial optimization problem (Hamilton) - salesperson has to visit *N* cities and return, optimizing for the shortest possible route.



Represented by a graph with vertices as the cities, pairwise distances/costs (shown for 5 destinations, A-E).

Example: TSP as a Decision Tree

We can represent our traveling salesperson (hereafter TSP) as a decision tree:



Such a tree can quickly enumerate all the potential solutions, but that does not necessarily help you search them all in a reasonable time frame.

B&B Searching

Various strategies are used in B&B searches:

- **depth-first**, start at one side of tree, first moving downward, then across
- **breadth-first**, expands each level before proceeding downward in tree
- **best-first**, directed down paths most likely to lead to better solution (does not proceed down paths than can not lead to a better solution, which are pruned)

Pruning

Pruning is a key feature of B&B searching, but it requires an ability to decide if a better solution can be found lower in the state space tree

- Needs **bounding function** ot **cutoff function** (this is the bound part of B&B)
- No general form for bounding function (depends too much on the application)
- Generally the bounding function needs to determine the upper and lower bounds for a subregion
- Key feature: in a minimization problem, if the lower bound for region A is greater than the upper bound for (previously examined region) B, then A can be safely pruned

Example: TSP Bounding Function

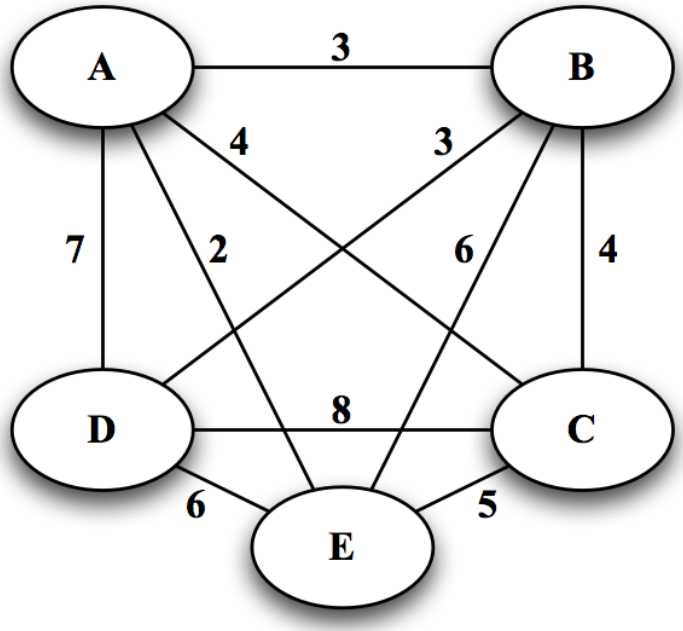
In our TSP example, we denote a subset of solutions $\{S\}$, with a lower bound $L(\{S\})$, and the best solution thus far in terms of cost, C.

- if $L \geq C$, there is no need to continue to evaluate $\{S\}$, as we are not going to beat C
- if $L < C$, we need to keep exploring $\{S\}$, as this is a candidate for a better solution
- Cost of any tour is half the sum of the costs of the 2 legs adjacent to each vertex, summed over all vertices (note the double counting)
  - Gives us an upper bound - this cost is greater than or equal to the sum of the 2 least expensive legs at each vertex
  - An overall lower bound is the sum over vertices and their 2 adjacent lowest cost legs, in our example,

$$\frac{1}{2}\left[A(2+3)+B(3+3)+C(2+6)+D(4+5)+E(2+5)\right] = 17.5$$

Example: TSP Bounding Function (cont'd)

Illustration of overall lower bound calculation:



An overall lower bound is the sum over vertices and their 2 adjacent lowest cost legs, in our example,

$$\frac{1}{2}\left[A(2+3)+B(3+3)+C(2+6)+D(4+5)+E(2+5)\right]=17.5$$

More B&B Terminology

A few more B&B implementation details (N.B., in the context of our TSP example, a node is a branch):

- **live node**, node has been reached, but not all children yet explored
- **E-node**, live node in which children are being explored
- **dead node**, all children have been explored
- **queue**, list of live nodes, maintained as search proceeds (sometimes called the **open list**)
- **best-first** strategy, queue is a priority queue
- **backtracking**, search does not proceed further down the tree, instead backs up to a higher level (best to use a last-in, first-out stack)

Example: TSP Branching & Bounding

In our TSP example, conditions for branching:

- Include path XY if it enables X or Y to have adjacent legs
- Exclude path XY if it causes more than two adjacent paths or completes a cycle with that leg already included
- Lower bound for particular branch computed, used in pruning. E.g., see decision tree, if we use path AD, but exclude BC, we then have a lower bound for that branch:

$$\frac{1}{2}\left[A(7+2)+B(3+3)+C(4+5)+D(7+3)+E(2+5)\right]=20.5$$

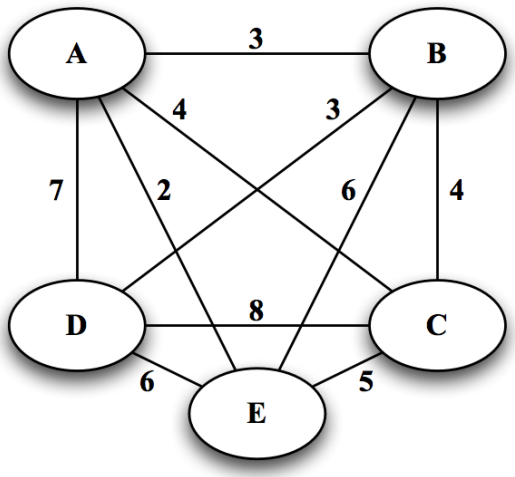
If the lower bound for a particular branch exceeds the current best solution, we can prune it and neglect its leaves

Example: TSP Branching & Bounding (cont'd)

Illustration of previous example:

- Lower bound for particular branch computed, used in pruning. E.g., see decision tree, if we use path AD, but exclude BC, we then have a lower bound for that branch:

$$\frac{1}{2}\left[A(7+2)+B(3+3)+C(4+5)+D(7+3)+E(2+5)\right]=20.5$$



A substantial amount of pruning can take place using the bounding criterion. How do you parallelize this kind of approach?

Parallelizing B&B

At first glance, **depth-first** approach appears to be most amenable to parallelization:

- Creates a **breadth-first** wavefront as processors search downward
- Drawbacks:
  - All processors need to know bounds for pruning (bounding function), which changes as better solutions are found
  - Load balancing difficult
  - Queue needs to be a shared data structure, but even in a shared-memory implementation needs to be locked to prevent simultaneous writes

- Rao and Kumar, "Concurrent Access of Priority Queues," IEEE Trans. Comp. **37**, 1657-1665 (1988).
  - Parallel Speedup limited by queue:

$$S \leq \frac{\tau_{queue} + \tau_{comp}}{\tau_{queue}},$$

where τ_{queue} is average time to access queue, and τ_{comp} is the time to compute a node. Suggests using a heap, or windowing strategies to improve potential scaling

B&B Applications

B&B has tended to be used for so-called **NP-complete** or **NP-hard** problems (polynomial time reduction is not possible, instead exponential):

- Knapsack problem
- Nonlinear Programming (objective function and all constraints are nonlinear)
- Traveling Salesperson

Generally the optimization involved is quite difficult to solve in parallel without a careful attention to load balancing issues (and the details of which tend to be unique to each application).

Successive Refinement

Successive refinement is an intuitive approach:

- Choose initial grid spacing sufficiently coarse to enable "brute force" evaluation in a reasonable amount of time
- Take K best points, and refine a domain centered on each, repeat, ...
- E.g., using $f(x, y, z)$,

$$f(x, y, z) = -x^2 + 10^6 x - y^2 - (4 \times 10^4) y - z^2,$$

where x, y, z are integers in the range of $-10^6 \leq x, y, z \leq 10^6$. choose an initial increment of 10^4, which requires $(200)^3$ evaluations:

  - Best K, subdivide 10^2 times - $K \times 10^6$ evaluations
  - Best K subdomains, $K^2 \times 10^6$ evaluations
  - For $K = 10$, converges in $\sim 10^8$ evaluations (GA can be 10-100 times faster)

- Readily parallelized - master/worker or by spatial decomposition

Hill Climbing

The name almost says it all, **hill climbing** is a method in which a "hiker" always moves uphill (maximization), i.e. in a direction of increasing function value.

- Quite susceptible to getting stuck in local extrema - how do we avoid that fate?
- Increasing the number of hill climbers:
  - Does not actually guarantee that any hiker finds the global extremum, but ...
  - Increasing number of climbers with randomized starting points can make the likelihood of missing the global extremum arbitrarily small

Hill Climbing (cont'd)

- Note that this is the essence of **Monte Carlo** search methods
- Highly (trivially!) parallelizable, either using master/worker, or static assignment

Simulated Annealing

Simulated annealing (SA) is something of a variation on hill climbing.

- Based on Boltzmann probability distribution,

$$P(E) \sim e^{-E/k_B T},$$

  where $P(E)$ is the probability that a system in thermal equilibrium at temperature T has energy E.
- Metropolis algorithm (1953), posits that the probability of a simulated system will move from state 1 to state 2 is

$$p_{12} = \exp\left[-(E_2 - E_1)/k_B T\right]$$

- Has the general property of always taking downhill step, but sometimes taking an uphill step

Summary of SA

Basics of SA:

- Description of system configurations (discrete or continuous)
- Generator of random changes
- Objective function to determine "energy," E
- Control parameter of "temperature," T
- **Annealing schedule** to gradually lower temperature to "freeze-in" solution (generally requires experimentation)
- Parallelism is similarly easy as in hill climbing

SA Example 1

Simple example of Simulated Annealing - find the global minimum of a damped sine wave:

$$f(x) = \exp\left((x-1)^2\right)\sin(8x),$$

True global minimum at 1.36313, quite a few local minima.
Example from GSL - Gnu Scientific Library.

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4  #include <gsl/gsl_test.h>
5  #include <gsl/gsl_rng.h>
6  #include <gsl/gsl_siman.h>
7  #include <gsl/gsl_ieee_utils.h>
8  #include <stdio.h>
9
10 /* set up parameters for this simulated annealing run */
11 #define N_TRIES 200             /* how many points do we try before stepping */
12 #define ITERS_FIXED_T 1000      /* how many iterations for each T? */
13 #define STEP_SIZE 1.0           /* max step size in random walk */
14 #define K 1.0                   /* Boltzmann constant */
15 #define T_INITIAL 0.008         /* initial temperature */
16 #define MU_T 1.003              /* damping factor for temperature */
17 #define T_MIN 2.0e-6
18
19 gsl_siman_params_t params = {N_TRIES, ITERS_FIXED_T, STEP_SIZE,
20                              K, T_INITIAL, MU_T, T_MIN};
21
22 double square (double x) ;
23 double square (double x) { return x * x ; }
24
25 double E1(void *xp);
26 double M1(void *xp, void *yp);
27 void S1(const gsl_rng * r, void *xp, double step_size);
28 void P1(void *xp);
```

```
29 /* now some functions to test in one dimension */
30 double E1(void *xp)
31 {
32   double x = * ((double *) xp);
33
34   return exp(-square(x-1))*sin(8*x) - exp(-square(x-1000))*0.89;
35 }
36
37 double M1(void *xp, void *yp)
38 {
39   double x = *((double *) xp);
40   double y = *((double *) yp);
41
42   return fabs(x - y);
43 }
44
45 void S1(const gsl_rng * r, void *xp, double step_size)
46 {
47   double old_x = *((double *) xp);
48   double new_x;
49
50   new_x = gsl_rng_uniform(r)*2*step_size - step_size + old_x;
51
52   memcpy(xp, &new_x, sizeof(new_x));
53 }
54
55 void P1(void *xp)
56 {
57   printf(" %12g ", *((double *) xp));
58 }
```

```
59  int main(void)
60  {
61    double x_min = 1.36312999455315182 ;
62    double x ;
63
64    gsl_rng * r = gsl_rng_alloc (gsl_rng_env_setup()) ;
65
66    gsl_ieee_env_setup ();
67    /* The function tested here has multiple mimima.
68       The global minimum is at    x = 1.36312999, (f = -0.87287)
69       There is a local minimum at x = 0.60146196, (f = -0.84893) */
70
71    x = -10.0 ;
72    gsl_siman_solve(r, &x, E1, S1, M1, NULL, NULL, NULL, NULL, sizeof(double), params);
73    gsl_test_rel(x, x_min, 1e-3, "f(x)= exp(-(x-1)^2) sin(8x), x0=-10") ;
74    x = +10.0 ;
75    gsl_siman_solve(r, &x, E1, S1, M1, NULL, NULL, NULL, NULL, sizeof(double), params);
76    gsl_test_rel(x, x_min, 1e-3, "f(x)= exp(-(x-1)^2) sin(8x), x0=10") ;
77
78    /* Start at the false minimum */
79    x = +0.6 ;
80    gsl_siman_solve(r, &x, E1, S1, M1, NULL, NULL, NULL, NULL, sizeof(double), params);
81    gsl_test_rel(x, x_min, 1e-3, "f(x)= exp(-(x-1)^2) sin(8x), x0=0.6") ;
82    x = +0.5 ;
83    gsl_siman_solve(r, &x, E1, S1, M1, NULL, NULL, NULL, NULL, sizeof(double), params);
84    gsl_test_rel(x, x_min, 1e-3, "f(x)= exp(-(x-1)^2) sin(8x), x0=0.5") ;
85    x = +0.4 ;
86    gsl_siman_solve(r, &x, E1, S1, M1, NULL, NULL, NULL, NULL, sizeof(double), params);
87    gsl_test_rel(x, x_min, 1e-3, "f(x)= exp(-(x-1)^2) sin(8x), x0=0.4") ;
88    gsl_rng_free(r);
89    exit (gsl_test_summary ());
90  }
```

```
1  [rush:~/d_gsl]$ gsl-config --libs
2  -lgsl -lgslcblas -lm
3  [rush:~/d_gsl]$ gcc -o siman_test siman_test.c -lgsl -lgslcblas -lm
4  [rush:~/d_gsl]$ export GSL_TEST_VERBOSE=1
5  [rush:~/d_gsl]$ ./siman_test
6  PASS: f(x)= exp(-(x-1)^2) sin(8x), x0=-10 (1.36313 observed vs 1.36313 expected)
7  PASS: f(x)= exp(-(x-1)^2) sin(8x), x0=10 (1.36313 observed vs 1.36313 expected)
8  PASS: f(x)= exp(-(x-1)^2) sin(8x), x0=0.6 (1.36313 observed vs 1.36313 expected)
9  PASS: f(x)= exp(-(x-1)^2) sin(8x), x0=0.5 (1.36313 observed vs 1.36313 expected)
10 PASS: f(x)= exp(-(x-1)^2) sin(8x), x0=0.4 (1.36313 observed vs 1.36313 expected)
```

SA Example 2 - TSP

Another GSL example, minimize the travel distance between 12 Southwestern cities, this time using SA.

```
1  #include <math.h>
2  #include <string.h>
3  #include <stdio.h>
4  #include <gsl/gsl_math.h>
5  #include <gsl/gsl_rng.h>
6  #include <gsl/gsl_siman.h>
7  #include <gsl/gsl_ieee_utils.h>
8
9  /* set up parameters for this simulated annealing run */
10
11 #define N_TRIES 200             /* how many points do we try before stepping */
12 #define ITERS_FIXED_T 2000      /* how many iterations for each T? */
13 #define STEP_SIZE 1.0           /* max step size in random walk */
14 #define K 1.0                   /* Boltzmann constant */
15 #define T_INITIAL 5000.0        /* initial temperature */
16 #define MU_T 1.002              /* damping factor for temperature */
17 #define T_MIN 5.0e-1
18
19 gsl_siman_params_t params = {N_TRIES, ITERS_FIXED_T, STEP_SIZE,
20                              K, T_INITIAL, MU_T, T_MIN};
21 struct s_tsp_city {
22   const char * name;
23   double lat, longitude;        /* coordinates */
24 };
25 typedef struct s_tsp_city Stsp_city;
```

```
26 void prepare_distance_matrix(void);
27 void exhaustive_search(void);
28 void print_distance_matrix(void);
29 double city_distance(Stsp_city c1, Stsp_city c2);
30 double Etsp(void *xp);
31 double Mtsp(void *xp, void *yp);
32 void Stsp(const gsl_rng * r, void *xp, double step_size);
33 void Ptsp(void *xp);
34
35 /* in this table, latitude and longitude are obtained from the US
36    Census Bureau, at http://www.census.gov/cgi-bin/gazetteer */
37
38 Stsp_city cities[] = {{"Santa Fe",     35.68,   105.95},
39                       {"Phoenix",      33.54,   112.07},
40                       {"Albuquerque",  35.12,   106.62},
41                       {"Clovis",       34.41,   103.20},
42                       {"Durango",      37.29,   107.87},
43                       {"Dallas",       32.79,    96.77},
44                       {"Tesuque",      35.77,   105.92},
45                       {"Grants",       35.15,   107.84},
46                       {"Los Alamos",   35.89,   106.28},
47                       {"Las Cruces",   32.34,   106.76},
48                       {"Cortez",       37.35,   108.58},
49                       {"Gallup",       35.52,   108.74}};
50
51 #define N_CITIES (sizeof(cities)/sizeof(Stsp_city))
52
53 double distance_matrix[N_CITIES][N_CITIES];
```

```
54  double city_distance(Stsp_city c1, Stsp_city c2)
55  {
56    const double earth_radius = 6375.000; /* 6000KM approximately */
57    /* sin and cos of lat and long; must convert to radians */
58    double sla1 = sin(c1.lat*M_PI/180), cla1 = cos(c1.lat*M_PI/180),
59      slo1 = sin(c1.longitude*M_PI/180), clo1 = cos(c1.longitude*M_PI/180);
60    double sla2 = sin(c2.lat*M_PI/180), cla2 = cos(c2.lat*M_PI/180),
61      slo2 = sin(c2.longitude*M_PI/180), clo2 = cos(c2.longitude*M_PI/180);
62
63    double x1 = cla1*clo1;
64    double x2 = cla2*clo2;
65
66    double y1 = cla1*slo1;
67    double y2 = cla2*slo2;
68
69    double z1 = sla1;
70    double z2 = sla2;
71
72    double dot_product = x1*x2 + y1*y2 + z1*z2;
73
74    double angle = acos(dot_product);
75
76    /* distance is the angle (in radians) times the earth radius */
77    return angle*earth_radius;
78  }
```

```
79  /* energy for the travelling salesman problem */
80  double Etsp(void *xp)
81  {
82    /* an array of N_CITIES integers describing the order */
83    int *route = (int *) xp;
84    double E = 0;
85    unsigned int i;
86
87    for (i = 0; i < N_CITIES; ++i) {
88      /* use the distance_matrix to optimize this calculation; it had
89         better be allocated!! */
90      E += distance_matrix[route[i]][route[(i + 1) % N_CITIES]];
91    }
92
93    return E;
94  }
95  double Mtsp(void *xp, void *yp)
96  {
97    int *route1 = (int *) xp, *route2 = (int *) yp;
98    double distance = 0;
99    unsigned int i;
100
101   for (i = 0; i < N_CITIES; ++i) {
102     distance += ((route1[i] == route2[i]) ? 0 : 1);
103   }
104
105   return distance;
106 }
```

```
107 /* take a step through the TSP space */
108 void Stsp(const gsl_rng * r, void *xp, double step_size)
109 {
110   int x1, x2, dummy;
111   int *route = (int *) xp;
112
113   step_size = 0 ; /* prevent warnings about unused parameter */
114
115   /* pick the two cities to swap in the matrix; we leave the first
116      city fixed */
117   x1 = (gsl_rng_get (r) % (N_CITIES-1)) + 1;
118   do {
119     x2 = (gsl_rng_get (r) % (N_CITIES-1)) + 1;
120   } while (x2 == x1);
121
122   dummy = route[x1];
123   route[x1] = route[x2];
124   route[x2] = dummy;
125 }
126
127 void Ptsp(void *xp)
128 {
129   unsigned int i;
130   int *route = (int *) xp;
131   printf("  [");
132   for (i = 0; i < N_CITIES; ++i) {
133     printf(" %d ", route[i]);
134   }
135   printf("]  ");
136 }
```

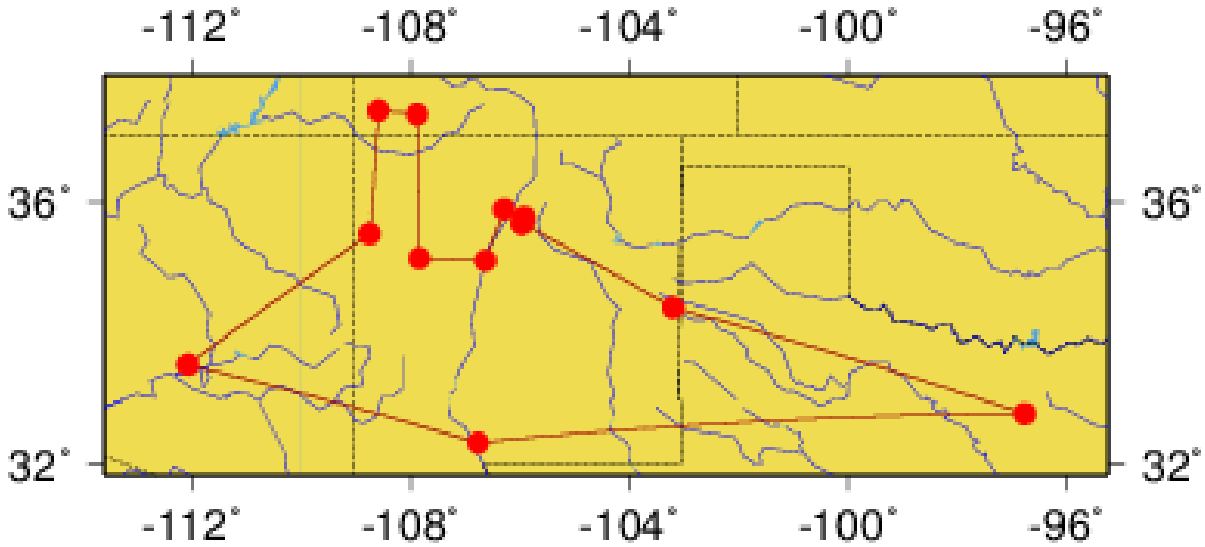
```
137 int main(void)
138 {
139   int x_initial[N_CITIES];
140   unsigned int i;
141
142   const gsl_rng * r = gsl_rng_alloc (gsl_rng_env_setup()) ;
143   gsl_ieee_env_setup ();
144   prepare_distance_matrix();
145
146   /* set up a trivial initial route */
147   printf("# initial order of cities:\n");
148   for (i = 0; i < N_CITIES; ++i) {
149     printf("# \"%s\"\n", cities[i].name);
150     x_initial[i] = i;
151   }
152
153   printf("# distance matrix is:\n");
154   print_distance_matrix();
155
156   printf("# initial coordinates of cities (longitude and latitude)\n");
157   for (i = 0; i < N_CITIES+1; ++i) {
158     printf("###initial_city_coord: %g %g \"%s\"\n",
159            -cities[x_initial[i % N_CITIES]].longitude,
160            cities[x_initial[i % N_CITIES]].lat,
161            cities[x_initial[i % N_CITIES]].name);
162   }
163
164 /*   exhaustive_search(); */
165   gsl_siman_solve(r, x_initial, Etsp, Stsp, Mtsp, Ptsp, NULL, NULL, NULL,
166                   N_CITIES*sizeof(int), params);
```

```
167   printf("# final order of cities:\n");
168   for (i = 0; i < N_CITIES; ++i) {
169     printf("# \"%s\"\n", cities[x_initial[i]].name);
170   }
171
172   printf("# final coordinates of cities (longitude and latitude)\n");
173   for (i = 0; i < N_CITIES+1; ++i) {
174     printf("###final_city_coord: %g %g %s\n",
175            -cities[x_initial[i % N_CITIES]].longitude,
176            cities[x_initial[i % N_CITIES]].lat,
177            cities[x_initial[i % N_CITIES]].name);
178   }
179   printf("# ");
180   fflush(stdout);
181   return 0;
182 }
183 void prepare_distance_matrix()
184 {
185   unsigned int i, j;
186   double dist;
187
188   for (i = 0; i < N_CITIES; ++i) {
189     for (j = 0; j < N_CITIES; ++j) {
190       if (i == j) {
191         dist = 0;
192       } else {
193         dist = city_distance(cities[i], cities[j]);
194       }
195       distance_matrix[i][j] = dist;
196     }
197   }
198 }
```

```
199 void print_distance_matrix() {
200   unsigned int i, j;
201   for (i = 0; i < N_CITIES; ++i) {
202     printf("# ");
203     for (j = 0; j < N_CITIES; ++j) {
204       printf("%15.8f   ", distance_matrix[i][j]);
205     }
206     printf("\n");
207   }
208 }
209 /* [only works for 12] search the entire space for solutions */
210 static double best_E = 1.0e100, second_E = 1.0e100, third_E = 1.0e100;
211 static int best_route[N_CITIES];
212 static int second_route[N_CITIES];
213 static int third_route[N_CITIES];
214 static void do_all_perms(int *route, int n);
215
216 void exhaustive_search() {
217   static int initial_route[N_CITIES] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
218   printf("\n# "); fflush(stdout); fflush(stdout);
219   do_all_perms(initial_route, 1);
220   printf("\n# "); fflush(stdout); fflush(stdout);
221   printf("# exhaustive best route: ");
222   Ptsp(best_route);
223   printf("\n# its energy is: %g\n", best_E);
224   printf("# exhaustive second_best route: ");
225   Ptsp(second_route);
226   printf("\n# its energy is: %g\n", second_E);
227   printf("# exhaustive third_best route: ");
228   Ptsp(third_route);
229   printf("\n# its energy is: %g\n", third_E);
230 }
```

```
231 /* James Theiler's recursive algorithm for generating all routes */
232 static void do_all_perms(int *route, int n) {
233   if (n == (N_CITIES-1)) { /* do it! calculate the energy/cost for that route */
234     double E; E = Etsp(route);  /* TSP energy function */
235     if (E < best_E) { /* now save the best 3 energies and routes */
236       third_E = second_E;
237       memcpy(third_route, second_route, N_CITIES*sizeof(*route));
238       second_E = best_E;
239       memcpy(second_route, best_route, N_CITIES*sizeof(*route));
240       best_E = E;
241       memcpy(best_route, route, N_CITIES*sizeof(*route));
242     } else if (E < second_E) {
243       third_E = second_E;
244       memcpy(third_route, second_route, N_CITIES*sizeof(*route));
245       second_E = E;
246       memcpy(second_route, route, N_CITIES*sizeof(*route));
247     } else if (E < third_E) {
248       third_E = E;
249       memcpy(route, third_route, N_CITIES*sizeof(*route));
250     }
251   } else {
252     int new_route[N_CITIES];
253     unsigned int j; int swap_tmp;
254     memcpy(new_route, route, N_CITIES*sizeof(*route));
255     for (j = n; j < N_CITIES; ++j) {
256       swap_tmp = new_route[j];
257       new_route[j] = new_route[n];
258       new_route[n] = swap_tmp;
259       do_all_perms(new_route, n+1);
260     }
261   }
262 }
```

```
[rush:~/d_gsl]$ gcc -o siman_tsp siman_tsp.c -lgsl -lgslcblas -lm ; ./siman_tsp
# initial order of cities:
# "Santa Fe"
# "Phoenix"
# "Albuquerque"
# "Clovis"
# "Durango"
# "Dallas"
# "Tesuque"
# "Grants"
# "Los Alamos"
# "Las Cruces"
# "Cortez"
# "Gallup"
...
#-iter  #-evals   temperature     position   energy
    0      2001          5000  [ 0  3  2  6  9  10  7  1  11  4  8  5 ]         4999.63
    1      4001       4990.02  [ 0  1  6  9  10  5  4  3  2  11  7  8 ]         5851.94
...
 4609   9220001      0.500774  [ 0  6  8  2  7  4  10  11  1  9  5  3 ]         3490.62
# final order of cities:
# "Santa Fe"
# "Tesuque"
# "Los Alamos"
# "Albuquerque"
# "Grants"
# "Durango"
# "Cortez"
# "Gallup"
# "Phoenix"
# "Las Cruces"
# "Dallas"
# "Clovis"
```



Plot of shortest travel path, with boundaries.

Downhill Simplex Method

Simple, (fairly) robust algorithm due to Nelder and Mead. Construct a "simplex" of $N + 1$ points in N dimensions, typically a starting point $\mathbf{P}_0$ plus

$$\mathbf{P}_i = \mathbf{P}_0 + \lambda_i \mathbf{e}_i,$$

where the $\mathbf{e}_i$ are "unit" directional vectors, and λ_i representative length scales. Common recipe:

1. Reflect point with largest value through centroid of simplex
2. If reflected point has smallest objective value, expand simplex and reflect again
3. If reflected point is neither best nor worst, repeat reflection step
4. If reflected point is now the worst, compress simplex and reflect again

Cautionary note: need to start with a reasonably large simplex, and convergence can be quite slow. Good downhill simplex animation at:

http://en.wikipedia.org/wiki/Nelder-Mead_method

Parallelization is an issue - task level parallelism is hard for simplex, unless you do something like "concurrent simplexes!" ...

J. A. Nelder and R. Mead, Comp. Journal **7**, 308 (1965).

Basic Ideas of Genetic Algorithms

The motivation behind genetic algorithms (GA) lies in biology:

- **Crossover**, the inheritance of some portion of characteristics from parents
- **Mutation**, sudden change in characteristics unrelated to parental contributions (significant, but infrequent compared to crossover)
- Eventually population tends to produce a population weighted towards characteristics that enhance reproduction
- Just as in nature, “population of solutions” tends over time to produce good solutions

Pseudo-code for Sequential GA

We can already write the basic algorithm:

```
generation = 0;
Initialize_Population(generation);
Evaluate_Population(generation);
Termination = false;
while ( ! Termination ) {
   generation ++;
   Select_Parents(generation, Population(generation-1));
   Crossover(generation,Parents(generation),Offspring(generation));
   Mutate(Offspring(generation));
   Evaluate_Population(generation);
   Update_termination(Termination);
}
```

and now we just need to fill in a few details ...

GA Details

Some of the details that we need to fill in for GA:

- How to represent individual (or potential solution) in terms of "chromosomes"
- How to produce initial population of potential solutions (individuals)
- How to evaluate fitness of each potential solution
- How to determine termination conditions
- How to select individuals to be parents and how to produce offspring

Initial Population

Let's take a concrete example - suppose that we want to maximize the function:

$$f(x, y, z) = -x^2 + 10^6 x - y^2 - (4 \times 10^4) y - z^2,$$

where x, y, z are integers in the range of $-10^6 \leq x, y, z \leq 10^6$. The above equation can be simplified to

$$f(x, y, z) = 2504 \times 10^8 - (x - 5 \times 10^5)^2 - (y + 2 \times 10^4)^2 - z^2,$$

for which we see that maximum is 2504×10^8 at $(5 \times 10^5, -2 \times 10^4, 0)$. An exhaustive search is ruled out by the number of possible coordinates, $(2 \times 10^6)^3$.

Data Representation

First, we need to choose how to represent our individuals numerically. Early GA work used strings of binary digits (0's and 1's) - and we will use it in our example as well. Floating point representation is also possible, of course, and is supported in many packages.

- Potential solution i is a 3-tuple, (x_i, y_i, z_i)
- The range of our solution is $2^{20} < 2,000,001 \leq 2^{21}$, so we need 21 bits for each coordinate, or a total of 63 bits, concatenating all 3 components.

- For example,

$x = 262,408$	00100000000100001000
$y = 16,544$	00000010000010100000
$z = -1032$	10000000010000001000

  Note that the first bit represents the sign, and our range is $\pm 1,048,575$ ($\pm 2^{20}$).
- Can use a pseudo-random number generator to produce initial population

- For **fitness**, we can simply use the value of $f(x, y, z)$ - the larger the value, the more fit
- **Constraints**, in our example a value outside the domain ($\pm 10^6$) is given a low enough fitness value that it not even evaluated (stillborn) - alternatively, one could scale the coordinates to correct such a defect and bring the value back into the domain

$$\textit{scaled coordinate} = -10^6 + (2 \times 10^6)\frac{\textit{coord}}{2^{21}}$$

- **Population size**, too few will require many generations, too many are costly to evaluate (highly dependent on problem); typically $\sim 10^2 - 10^3$.

Selection

Selective pressure means that the more fit the individual, the higher the probability of being selected to produce the next generation.

- Naively choose only the most fit individuals?
  No, that tends to produce population that gets stuck in local extrema
- Most popular technique, **tournament selection**
  - Individuals selected at random and entered into tournament
  - k individuals reduced to winner by fitness criteria
  - n such tournaments produce n individuals to be parents
  - $k = 1$, no selective pressure at all
  - larger k, more selective pressure
  - $k = 2$, “medieval joust” is a popular choice

Offspring Production

Once parents have been chosen via selection, we have to merge their information ("chromosomes") to produce **offspring**.

- **Crossover**, we combine portions of chromosome from each parent
- In our example, we will use **single-point** crossover:
  - Random cut at bit p in an m-bit chromosome of parents A and B
  - Child 1 gets 1st p bits from A, $m - p$ bits from B
  - Child 2 gets 1st p bits from B, $m - p$ bits from A
  - Resulting children are a blend of parents' characteristics
- **Multi-point** crossover, several cuts are made, smaller portions of chromosomes mingled

Mutation

Mutation in GA takes place at the level of isolated genes (analogous to disease, radiation, etc.):

- Tends to bring about major (and unpredictable) changes
- In our example, changing the first bit in each coordinate changes the sign, while the 2nd bit changes the value by $2^{19} = 524,288$. But changing the last bit only changes the value by 1.
- Generally in GA the chance of mutation is kept small to prevent shifting the solution away from convergence

Variations

Instead of just using crossover and mutation there are several popular variants:

- Carry over a few of the most-fit individuals to next generation
- Randomly create a few individuals in each generation, rather than only in the initial population
- Allow population size to vary during evolution of generations

Termination Conditions

There are a couple of alternatives for deciding when to terminate GA:

- Run a fixed number of generations (imprecise at best)
- Mimic numerical analysis and base termination on degree of improvement between successive generations (but GA can oscillate, just like more conventional numerical techniques)
- Based on degree of similarity between individuals within population (similarity increases as we converge to optimal solution)
- Can use multiple conditions as well, of course

Parallelizing GA

We have a couple of possibilities for running GA in parallel:

1. Each processor handles a sub-population, periodically sharing the most-fit individuals through **migration**
2. Each processor handles portion of each major task (selection, crossover, mutation, etc.) on the full population

Isolated Sub-populations

In the approach where each processor handles a separate sub-population, we have an additional task to manage - **migration**:

- after $k \geq 1$ generations, processors share most-fit individuals with other processors
- Tools needed:
  - Select emigrants
  - Send emigrants
  - Receive immigrants
  - Integrate immigrants
- Selection and integration most interesting - naively using most-fit individuals creates a lot of (too much!) selective pressure

Migration Models

Two of the most popular models for migration:

1. **island model**, no restrictions on which sub-population available for migration (more communication overhead)
2. **stepping-stone**, only neighboring sub-populations are available for emigration

- Very high degree of parallelism until generation k when migration occurs
- Mimics **punctuated equilibria** in which rapid changes in sub-population follow introduction of new individuals
- Isolation present in stepping-stone model tends to produce sub-populations that converge to local extrema (think Australia or other isolated island populations)

Common Population

Parallelizing over a common population is more difficult:

- Very sensitive to time required to perform various GA operations
- Often the “genetic operators” are not computationally intensive, which negatively effects this approach
- Shared memory systems can mitigate some of the above concerns, since all data is shared by default (but still synchronization is required between generations)
- Distributed memory systems can use stepping-stone like migration models to mitigate some communication, but are better suited to the sub-population approach to parallelizing GA

GA Example

Too good to pass by, the "Hello, World!" example of GA optimization:

- Author: James Matthews,
  http://www.generation5.org/content/2003/gahelloworld.asp
- Fixes string length, treats fitness by deviation from target string (so lower is better)
- Nifty example for the basic concepts

```
1  #include <iostream>                          // for cout etc.
2  #include <vector>                            // for vector class
3  #include <string>                            // for string class
4  #include <algorithm>                         // for sort algorithm
5  #include <time.h>                            // for random seed
6  #include <math.h>                            // for abs()
7
8  #define GA_POPSIZE       2048                // ga population size
9  #define GA_MAXITER       16384               // maximum iterations
10 #define GA_ELITRATE      0.10f               // elitism rate
11 #define GA_MUTATIONRATE  0.25f               // mutation rate
12 #define GA_MUTATION      RAND_MAX*GA_MUTATIONRATE
13 #define GA_TARGET        std::string("Hello world!")
14
15 using namespace std;                         // polluting global namespace, but hey...
16
17 struct ga_struct
18 {
19   string str;                          // the string
20   unsigned int fitness;                // its fitness
21 };
```

```
22  typedef vector<ga_struct> ga_vector;// for brevity
23
24  void init_population(ga_vector &population, ga_vector &buffer )
25  {
26    int tsize = GA_TARGET.size();
27
28    for (int i=0; i<GA_POPSIZE; i++) {
29      ga_struct citizen;
30
31      citizen.fitness = 0;
32      citizen.str.erase();
33      for (int j=0; j<tsize; j++) {
34        citizen.str += (rand() % 90) + 32;
35      }
36      population.push_back(citizen);
37    }
38    buffer.resize(GA_POPSIZE);
39  }
40
41  inline void print_best(ga_vector &gav)
42  { cout << "Best: " << gav[0].str << " (" << gav[0].fitness << ")" << endl; }
43
44  inline void swap(ga_vector *&population, ga_vector *&buffer)
45  { ga_vector *temp = population; population = buffer; buffer = temp; }
```

```
46  void calc_fitness(ga_vector &population)
47  {
48    string target = GA_TARGET;
49    int tsize = target.size();
50    unsigned int fitness;
51
52    for (int i=0; i<GA_POPSIZE; i++) {
53      fitness = 0;
54      for (int j=0; j<tsize; j++) {
55        fitness += abs(int(population[i].str[j] - target[j]));
56      }
57
58      population[i].fitness = fitness;
59    }
60  }
61
62  bool fitness_sort(ga_struct x, ga_struct y)
63  { return (x.fitness < y.fitness); }
64
65  inline void sort_by_fitness(ga_vector &population)
66  { sort(population.begin(), population.end(), fitness_sort); }
```

```
67  void elitism(ga_vector &population, ga_vector &buffer, int esize )
68  {
69    for (int i=0; i<esize; i++) {
70      buffer[i].str = population[i].str;
71      buffer[i].fitness = population[i].fitness;
72    }
73  }
74
75  void mutate(ga_struct &member)
76  {
77    int tsize = GA_TARGET.size();
78    int ipos = rand() % tsize;
79    int delta = (rand() % 90) + 32;
80
81    member.str[ipos] = ((member.str[ipos] + delta) % 122);
82  }
```

```
83  void mate(ga_vector &population, ga_vector &buffer)
84  {
85    int esize = GA_POPSIZE * GA_ELITRATE;
86    int tsize = GA_TARGET.size(), spos, i1, i2;
87
88    elitism(population, buffer, esize);
89
90    // Mate the rest
91    for (int i=esize; i<GA_POPSIZE; i++) {
92      i1 = rand() % (GA_POPSIZE / 2);
93      i2 = rand() % (GA_POPSIZE / 2);
94      spos = rand() % tsize;
95
96      buffer[i].str = population[i1].str.substr(0, spos) +
97        population[i2].str.substr(spos, esize - spos);
98
99      if (rand() < GA_MUTATION) mutate(buffer[i]);
100   }
101 }
```

```
102 int main()
103 {
104   srand(unsigned(time(NULL)));
105
106   ga_vector pop_alpha, pop_beta;
107   ga_vector *population, *buffer;
108
109   init_population(pop_alpha, pop_beta);
110   population = &pop_alpha;
111   buffer = &pop_beta;
112
113   for (int i=0; i<GA_MAXITER; i++) {
114       calc_fitness(*population);                  // calculate fitness
115       sort_by_fitness(*population);               // sort them
116       print_best(*population);                    // print the best one
117
118       if ((*population)[0].fitness == 0) break;
119
120       mate(*population, *buffer);                 // mate the population together
121       swap(population, buffer);                   // swap buffers
122   }
123
124   return 0;
125 }
```

```
[cash:~]$ g++ -o gahelloworld gahelloworld.cpp
[cash:~]$ ./gahelloworld
Best: LdL_dNMxidh& (184)
Best: jT``\4\okk`" (154)
Best: GnugoQrbpsw9 (143)
Best: =]nc^<yn_gb4 (123)
Best: Ffnfn+woWkK- (88)
Best: Bdinp"WpuoR! (72)
Best: Ddk]m&pls_f* (64)
Best: Bdinp"qorft' (49)
Best: Bdinn)tnsk`* (41)
Best: Bdiml"sorha# (29)
Best: Bdiml"sorha" (28)
Best: Ffnnn"rssk`" (26)
Best: Ffnfn"wosk`" (21)
Best: Gdiml"sorkd" (17)
Best: Ffnnn!work`" (15)
Best: Gdmkn!torkg" (14)
.....
Best: Gfkln workd! (5)
Best: Gdmln world! (4)
Best: Gdmln world! (4)
Best: Gdlln workd! (4)
Best: Gdmln world! (4)
Best: Gemln world! (3)
Best: Gemln world! (3)
Best: Iemln world! (3)
Best: Gemln world! (3)
Best: Iemlo world! (2)
Best: Iemlo world! (2)
Best: Gelln world! (2)
Best: Gfllo world! (2)
Best: Gello world! (1)
Best: Hello world! (0)
```

GA Packages

- **PGAPACK**, an open-source general-purpose parallel GA library: `http://ftp.mcs.anl.gov/pub/pgapack`
- **GAUL**, another: `http://gaul.sourceforge.net`
- **PIKAIA**, yet another (`IDL`, `F77`, `F90`): `http://www.hao.ucar.edu/modeling/pikaia/pikaia.php`

  (cute animation included, note P1 optimizes $f(x, y) = \cos^2(n\pi r) \exp\left(-r^2/\sigma^2\right)$, with $x, y \in [0, 1]$ and $r^2 = (x - 0.5)^2 + (y - 0.5)^2$)

Further Resources

- Hans Mittleman's Decision Tree for Optimization Software:
  http://plato.asu.edu/guide.html
  Large collection of optimization software organized by application and method - most free (including source), some commercial. Includes benchmarks and test cases.
- NEOS (Network Enabled Optimization System), another wide ranging optimization resource (and web enabled solvers) - good starting point is the wiki:
  http://www.neos-guide.org