A LOCAL SEARCH OPTIMIZATION ALGORITHM BASED ON NATURAL PRINCIPLES OF GRAVITATION

FLORIDA INSTITUTE OF TECHNOLOGY

TECHNICAL REPORT #CS-2003-10

Barry Webster
Northrop Grumman Corporation
2000 West Nasa Boulevard
M/S C07-222
Melbourne, FL 32902
barry_webster@northropgrumman.com

Philip J. Bernhard
Department of Computer Sciences
Florida Institute of Technology
150 West University Boulevard
Melbourne, FL 32901
pbernhar@cs.fit.edu

ABSTRACT

This document discusses the concept of an algorithm designed to locate the optimal solution to a problem in a (presumably) very large solution space. The algorithm attempts to locate the optimal solution to the problem by beginning a search at an arbitrary point in the solution space and then searching in the "local" area around the start point to find better solutions. The algorithm completes either when it locates what it thinks is the optimal solution or when predefined halt conditions have been met. The algorithm is repair-based, that is, it begins with a given solution and attempts to "repair" that solution by changing one or more of the components of the solution to bring the solution closer to the optimal. The algorithm uses the natural principles of gravity that act on a body in motion through space and simulates those principles to take a given solution to the problem at hand and repair it to locate an optimal solution. In this document two versions of the algorithm, called GLSA1 (based on simple gravitational force calculation) and GLSA2 (based on gravitational field calculation), are presented and the manner in which an initial evaluation of the algorithm was conducted. Then, by way of example a particular problem is given that the algorithm can be used to solve, along with a description of how the algorithm would be used to solve that problem. Finally, some conclusions and opportunities for future direction are presented.

INTRODUCTION

One of the most fundamental difficulties faced when attempting to find the optimal solution to a complex problem is that the solution space for the problem is likely to be very large. That is, the number of possible solutions to the problem is likely to be great enough to render an exhaustive search for the optimal solution impractical at best, nearly impossible at worst [1].

In cases like these, any algorithm capable of solving the problem in a reasonable amount of time must make use of one or more heuristics (or more colloquially, "rules of thumb" or educated guesses) as to how to proceed to locate the optimal solution, and often must settle for a near-optimal solution instead of guaranteeing that the optimal solution will be found [2].

There are many types of algorithms available for solving such problems. Some algorithms build a solution from scratch, while others are repair-based, taking an initial solution and repeatedly permuting the components of the solution in order to produce a more optimal solution [3]. Of interest here are the "local search" algorithms. These algorithms are repair-based, systematically and minutely permuting the current solution in such a way that the new solution produced is "in the neighborhood" of the previous solution within the solution space, as opposed to permuting the current solution unsystematically (such that the solutions generated by successive iterations "hop" around the solution space haphazardly). Each of these local search algorithms is a member of a class of algorithms that share similar techniques, or is a hybrid, combining characteristics of two or more classes [2, 4].

The algorithm to be described herein is one such hybrid. In the most basic sense, it is a variation of the "hill climbing" class of algorithms. These algorithms operate by taking the current solution and attempting to find a more optimal solution within the local search space. If one can be found, it becomes the current solution and the process repeats. If none can be found, the algorithm halts and the current solution is returned as the optimal solution. Thus, the algorithm operates as if climbing a hill, progressing from one solution to a better solution to one still better, until it reaches the "peak" of the hill it is climbing and there are no better solutions around [3].

While being based on hill climbing, the algorithm to be described herein contains elements of other classes of local search (which, for the sake of some degree of brevity,

will not be described) [3]. It also incorporates its own set of heuristics to determine its course of action. These heuristics are based on the natural principles of gravitational attraction.

Gravity is one of the four basic forces in physics. It is the weakest of the four, but it is also the most pervasive. Nothing is immune to its effects, and even the stars eventually succumb to its relentless influence. Gravitational attraction ensures that once an object is in the grip of the gravity of another object, it will never escape unless acted upon by another force [5]. By simulating the effects of gravitational attraction, the algorithm described herein attempts to locate the optimal solution to a problem by maneuvering the solution until it is in the grip of the optimal solution's "gravity".

A hurdle that is faced by all local search algorithms is that within the solution space there can be a number of localized suboptimal solutions in addition to the global optimal solution [2]. A local search algorithm may, because of how it operates, terminate at one of these local optima. Of course, if the initial solution presented to the algorithm is far away from the global optimal it may be necessary for the algorithm to terminate at a local optimum in order to complete within reasonable time constraints. However, if possible a local search algorithm should incorporate a method to attempt to bypass local optima in favor of the global optimum. The algorithm described herein contains a number of operating parameters that are intended to accomplish this function. How the algorithm attempts to maneuver the solution towards the global optimum while avoiding local optima will be explained in the next section.

DESCRIPTION OF THE ALGORITHM

The simplest way to describe how the gravitational local search algorithm (or GLSA, as it shall henceforth be called in this document) operates is to visualize a ball that is rolling around on a confined surface. On the surface are various pits, or "gravity wells", some deeper than others. As the ball rolls, it gradually loses momentum due to friction with the surface. If it begins to roll into one of the pits it will gain momentum, and conversely it will lose momentum if it begins to exit the pit. The idea is that due to the effects of gravity, eventually the ball will come to rest in one of the pits.

In this analogy, the confined surface represents the solution space for a given problem. The pits in the surface represent the various local

optima, with the deepest pit being the global optimum solution. The ball that is rolling around represents the current solution as it is being permuted. The hope is that the ball can be controlled such a way that it will have sufficient inertia to roll into and out of the more shallow pits (the local optima), settling only in the deepest pit (the global optimum). Of course, this analogy takes place in three-dimensional space, and many problems have more than three variables in their solution. However, this particular analogy was chosen solely for its ease of visualization, and GLSA can easily be extended to any number of solution variables.

We propose two basic versions of the GLSA algorithm. The first, called hereafter GLSA1, uses the Newtonian equations for gravitational force between two objects as its basis for simulation. The second, called hereafter GLSA2, uses the equations for gravitation field calculation as its basis for simulation. GLSA1 also contains more of a restriction on the movement of the solution ball, allowing it to move a distance of only one node in a given direction at a time, while GLSA2 allows for movement of distances of more than one node at a time. Aside from the differences of basis of simulation and range of movement, though, the two versions of GLSA both operate in the same fashion, using their respective bases of simulation to calculate where and how the solution ball should progress through the solution space, and then moving the solution ball until its combined momentum (the vector sum of its velocity in all dimensions) reaches zero, at which time the best solution seen to that point is returned.

There are a number of operating parameters that can be used to control the motion of the solution ball. Each of these parameters is individually adjustable, and together they comprise the "scenario" under which the algorithm will proceed. The parameters are as follows:

DENSITY (DENS) – the relative density of the medium through which the solution ball is moving. It is used to help calculate the resistive force that the solution ball experiences as it moves through the solution space. It defaults to a value of 1.2, the relative density of air [5].

DRAG (DRAG) – the drag coefficient of the solution ball. It is also used to help calculate the resistive force that the solution ball experiences as it moves through the solution space. It defaults to a value of 0.5, that of a relatively streamlined body [5].

FRICTION (FRIC) – the moving coefficient of friction of the solution ball and the surface on which it is rolling. It is also used to help calculate the resistive force that the solution ball experiences as it moves through the solution space. It defaults to a value of 0.003, the value for steel rolling on steel [5].

GRAVITY (GRAV) – the coefficient of gravity acting between two objects, and is used only in the GLSA1 version of the algorithm. It is used in the calculation of the gravitational force present between the solution ball and an adjacent solution. It defaults to the Newtonian value of 6.672 [5].

INITIAL VELOCITY (IVEL) – the maximum possible initial velocity of the solution ball for a given dimension of movement. It is used to put a bound on the relative speed of the solution ball in any one direction at the outset of the algorithm. It defaults to an arbitrary value of 10.

ITERATION LIMIT (ITER) – the maximum number of iterations of the GLSA procedure for a given problem instance. This parameter has been included to ensure termination of GLSA. In actuality, an object can settle into a stable orbit around another object. Obviously, should this happen during execution of GLSA, it would never terminate. ITER sets a limit on the number of times that the GLSA algorithm can iterate through its procedure. If the algorithm completes this number of iterations and the combined momentum of the solution ball has not yet reached zero, the algorithm terminates anyway and returns the best solution seen to that point.

MASS (MASS) – the mass of the solution ball itself, independent of the mass of the solution node at which the solution ball is currently located (as calculated by the relevance function, the function that determines the relative quality of the solution at that node). It is used in the various calculations that are dependent on the mass of an object, and defaults to an arbitrary value of 1.

RADIUS (RADI) – the distance between two objects, and is used only in the GLSA1 version of the algorithm. It is used in the calculation of the gravitational force present between the solution ball and an adjacent solution. It defaults to an arbitrary value of 2.

SILHOUETTE (SILH) – the silhouette area of the solution ball as seen from the front. It is used to help calculate the resistive force that the solution ball experiences as it moves through the solution space. It defaults to an arbitrary value of 0.1.

THRESHOLD (THRE) – the threshold at which a given velocity will be assumed to drop to zero. It is used to prevent the velocity component of the solution ball in any direction from hovering near zero without ever getting there due to round-offs and the limits of the calculations. It defaults to an arbitrary value of 2.

PSEUDOCODE OF THE ALGORITHM

Following is a high-level pseudocode representation of each of the two versions of GLSA (complete code listings for both GLSA1 and GLSA2 are included in appendices to this paper). Inputs to each of the procedures are: an encoded representation of the solution space, the node within the solution space at which the algorithm will commence, and value assignments for each of the aforementioned operational parameters.

Both procedures return the relative optimality of the best solution located by the algorithm, along with an encoded representation of that solution and the number of iterations that the algorithm took to complete.

All references to operational parameters within the pseudocode are made using the appropriate four-character abbreviations as listed above with the parameter names. The reference "RF" represents the calculated value of the relevance function for a given solution. The relevance function is an instance-specific function used by local search problems to calculate the relative optimality of a given solution. The more optimal a solution, the higher its associated RF value.

Simplified principles of gravitational forces and fields, as well as the other physical principles emulated in the algorithm, are used in the representative equations and calculations. In addition, the equations for resistive force are designed for use in a three-dimensional world, even though local search problems frequently contain more than three solution component dimensions. These steps were taken to avoid excessive computations that would have increased the complexity of GLSA without adding anything to its essence, that of using the basic principles of gravitational attraction as a basis for performing local search.

```
procedure GLSA1 is
  --Number of solution dimensions is a predefined number “n”
  integer dim;  --Dimension iteration counter
  integer cnt;  --Loop iteration counter
  begin
    for dim = 1 to n do
      --Assign a predefined starting solution component as the current solution component for
      --  dimension “dim” and as the best solution component seen thus far for that dimension
      --Randomly assign an initial velocity in the dimension “dim” within the bounds of IVEL
    end;
    cnt = 0;
    --Calculate an initial vector velocity sum, based on the random initial velocity components
    --  assigned in the previous step
    while (the velocity sum <> 0) and (cnt < ITER) do
      --Reset the velocity sum to 0
      for dim = 1 to n do
        --Calculate the solutions adjacent to the current solution and their respective RF values
        --If any of these is better than the best solution seen thus far, then make that solution
        --  the new best solution
        --Calculate the net difference in gravitational “force” between the adjacent solutions and
        --  the current solution for the current dimension “dim”, using the Newtonian equation for
        --  gravitational attraction
        --Calculate change in acceleration for the current dimension “dim”
        --Calculate change in velocity for the current dimension “dim”
        --Calculate new current solution component for the dimension “dim”, which will be the next
        --  adjacent node in the dimension “dim” in the current direction of movement as indicated
        --  by the velocity component for the dimension “dim”
      end;
      --Calculate the new velocity sum
      cnt = cnt + 1;
    end;
    return best solution found, its RF value, and the iteration count (cnt)
  end;


procedure GLSA2 is
  --Number of solution dimensions is a predefined number “n”
  integer dim;  --Dimension iteration counter
  integer cnt;  --Loop iteration counter
  begin
    for dim = 1 to n do
      --Assign a predefined starting solution component as the current solution component for
      --  dimension “dim” and as the best solution component seen thus far for that dimension
      --Randomly assign an initial velocity in the dimension “dim” within the bounds of IVEL
    end;
    cnt = 0;
    --Calculate an initial vector velocity sum, based on the random initial velocity components
    --  assigned in the previous step
    while (the velocity sum <> 0) and (cnt < ITER) do
      --Reset the velocity sum to 0
      for dim = 1 to n do
        --Calculate the solutions adjacent to the current solution, and their respective RF values
        --If any of these is better than the best solutions seen thus far, then make that solution
        --  the new best solution
        --Calculate the change in the gravitation field for the current dimension “dim”, as caused
        --  by the adjacent solutions and the current solution using the vector sum equation for
        --  gravitational field calculation
        --Calculate change in acceleration for the current dimension “dim”
        --Calculate change in velocity for the current dimension “dim”
        --Calculate new current solution component for the dimension “dim”, which will be the i^th
        --  adjacent node in the dimension “dim” in the current direction of movement as indicated
        --  by the magnitude i of the velocity component for the dimension “dim”
      end;
      --Calculate the new velocity sum
      cnt = cnt + 1;
    end;
    return best solution found, its RF value, and the iteration count (cnt)
  end;
```

EVALUATING THE ALGORITHM

In order to evaluate the efficacy of the GLSA algorithm, a test environment was devised. This environment was designed to test the ability of GLSA to find the optimal or near optimal solution within a given solution space when the optimal solution could be known with certainty.

The test environment consisted of a 10x100 matrix solution space, simulating a problem with ten variables. The matrix was populated with randomly-generated integer values ranging from zero to one hundred. These integer values represented the associated RF value of that particular node. The integers could be assigned with a pre-defined probability in order to represent problems with sparse solutions. That is, if the probability was defined to be 0.1, then only about ten percent of the nodes in the matrix would be assigned a non-zero value [1].

A complete solution to a problem instance generated in this fashion would be the sum of the integer values for a tuple consisting of one node from each of the ten dimensions, a value ranging from zero to one thousand. The higher the sum total, the better the quality of the solution. As the values were being assigned to the matrix, a record was being kept of the optimal solution seen to that point. Thus, when assignment of values to the matrix was complete, the overall optimal solution and its RF value were known.

This test environment allowed for the creation of problem instances of any degree of sparcity for the ten-dimensional problem. This provided for a solution space of up to 10^{100} possible solutions, a value sufficiently large to render an exhaustive search of the solution space impractical. Having said this, it should be readily obvious that the optimal solution to the problem could be located simply by scanning each of the ten dimensions and building the solution by selecting the largest value from the nodes in each dimension. However, this was done in order to facilitate the rapid generation of problem instances where the optimal solution was known up front.

While it is true that the definition of RF for this problem allowed for a very easy location of the optimal solution, such is very often not the case. In many cases, the RF for a problem involves a particular interrelationship between the individual variables in the solution. In these cases, the value of a given variable assignment relative to the overall RF is not known until the actual RF is calculated for a

complete solution tuple, and cannot be determined by simple examination of the value.

Thus, it can be seen that the test problem is a special case of the more general class of multi-dimensional problems, one where the RF just happens to be the sum of the values of the individual solution components. Given this fact, the problem can be approached by any solution algorithm as if it was of the more general class where a solution could not be readily obtained. Taking this approach allows for rapid generation of problem instances while maintaining a sufficiently large solution space for the solution algorithms to search.

To establish actual test scenarios using the defined test environment, a series of test instances was generated. Ten problem instances were generated for each of ten levels of sparcity, beginning at ten percent and going to one hundred percent, in increments of ten percent. Thus, ten tests were conducted with ten percent of the nodes in the solution space having non-zero values, another ten tests were conducted with twenty percent of the nodes having non-zero values, another ten tests at thirty percent, and so on, until the final ten tests were conducted where every node in the solution space had a non-zero value [1].

For each of the one hundred total test instances, the two versions of GLSA were run and the results collected and compared against the known optimal solution. To provide a further basis for comparison, the two versions of GLSA were compared against two other methods of finding a solution: random and basic hill climbing.

Each of the test scenarios progressed as follows: first, the problem instance was generated and the optimal solution recorded. Next, a random solution to the problem instance was generated (by selecting a random assignment for each of the ten dimensions) and its RF value calculated and recorded. Then, the hill climbing algorithm was run, using as its starting point the random solution, and its resultant solution and RF value was recorded. Finally, each of the two versions of GLSA was run, also using as its starting point the random solution (and using the default values for the operational parameters), and its solution and RF value was recorded.

This procedure was followed for each of the one hundred test instances. Once all the data were collected, a comparison could be made between the results obtained randomly, by hill climbing, and GLSA1/GLSA2, and the known optimal results. Having done so (and done so

by repeating the test set several times), the following items were noticed:

- The random solution, as was surmised, produced generally very poor solutions
- Hill climbing, beginning at the random solution and attempting to improve upon it, was in virtually every instance able to do so and produce a significantly better solution than the random solution
- GLSA1 and GLSA2, also beginning at the random solution and attempting to improve upon it, were also in virtually every instance able to do so and produce a significantly better solution than the random solution
- In the overwhelming majority of cases, both GLSA1 and GLSA2 were able to produce solutions better than those obtained by hill climbing, in many cases substantially better
- GLSA2 on average produced better solutions than GLSA1
- Typically, hill climbing completed in two to five iterations
- GLSA1 and GLSA2 both typically completed in twenty to twenty-five iterations, although there were a very few occasions where GLSA2 fell into a harmonic motion through the solution space and had to be terminated by maximum iteration count instead of by its normal termination conditions

These results, while far from being exhaustive, were very promising and at least served to demonstrate that the basic premise of the GLSA algorithm has some merit and is worthy of further investigation.

AN APPLICATION OF THE ALGORITHM

As previously stated, GLSA is a general purpose local search algorithm, adaptable for use with most types of problems for which a local search algorithm is appropriate. However, a particular problem will now be outlined to illustrate how GLSA can be used to solve such a problem.

The problem that will be used for this example is the File Assignment Problem, or FAP. In the FAP, a group of files needs to be assigned to a group of devices in such a manner as to optimize a predefined factor, such as file access

cost or throughput [6]. The FAP has been shown to be a difficult problem to solve efficiently, even with relatively few files and devices, so it is a good candidate for a local search solution approach [7].

There are many variations of the FAP. The particular variation that will be used in this example states that each file in the group must be assigned to one and only one device, and that each device may have zero or more files assigned to it. The goal is to minimize the total cost of accessing the files during runs of a particular set of programs (cost in this case is the time required for a device to locate and read a file, plus the latency time if a given file access request is issued but that file is on a device that is still in the process of accessing another file, plus the time required to transmit file data from a device, etc.) [6, 7].

To use GLSA with this particular variation of the FAP, a suitable RF must first be defined. In this case, RF would calculate the total file access cost for a given set of file-to-device assignments, based on a known sequence of file access requests that will be made. The set of file-to-device assignments is the solution. This solution can be encoded as an array where each subscript of the array represents one of the files in the group, and the value held at that subscript represents the device that the file is assigned to.

To initialize GLSA, a starting solution is selected, along with experimental values of the adjustable operation parameters. The run of GLSA then commences, and when it completes the solution returned gives for each file in the group the device to which it should be assigned, along with the value of RF for that solution. The solution returned can then be compared with the value of RF for solutions obtained by other algorithms to facilitate a comparison, or if an exhaustive search of the solution space is practicable, to prove whether or not GLSA located the optimal solution.

CONCLUSIONS

As previously stated, the empirical results obtained thus far with both GLSA1 and GLSA2 indicate that the algorithm holds at least some promise for being able to generate high quality solutions to a variety of local search problems such as described for the FAP.

With its available set of operational parameters, and with the variety of values that can be assigned to those parameters and their interplay, GLSA can be fine-tuned to operate

in a number of ways as its users see fit. The experimentation conducted indicated that changes in the values of the operational parameters can result in quite varied results for both versions of the algorithm (i.e. changing MASS from 1 to 10 resulted in quicker terminations of the algorithm, but with less optimal results). Thus, it is conceivable that by optimizing the values for the parameters, even better results could be obtained. However, no empirical results are currently available to verify this possibility.

With this in mind, one of the first things that can be done is to generate some results for test problems using the default parameter values, then attempt to optimize the parameter values to see if the performance of GLSA can be improved for those same test problems. An attempt could also be made to discover heuristics that can be generated to determine which combination(s) of operational parameter values are most likely to generate the best results. Other types of local search algorithms such as simulated annealing and genetic algorithms could be employed as the heuristic method, generating and testing combinations of operational parameter values against various problem instances in an attempt to discover optimal parameter settings [8].

GLSA could also be tested against these other types of algorithms to see how well its results compare. This could be done in combination with running GLSA against other different problem types, to see if GLSA performs better against some types of problems than others and/or performs better than other types of local search algorithm in general or only for certain types of problems [8].

Other modifications can also be made to GLSA. For example, it would be a simple process to parallelize GLSA by simply running an instance, with a unique set of operational parameter values, on each available processor and then returning the best solution obtained from all instances, or running both versions of the algorithm on different available processors and then returning the best solution obtained.

The possibility of other operational parameters could also be investigated, although care is required since while gravitational attraction between two objects is a relatively simple thing to calculate, the complexity increases geometrically as the number of objects for which the attraction interplays are calculated increases. If GLSA begins to account for the attractions between more than two objects, the amount of processing power required to perform the calculations could quickly become prohibitive [5].

The most basic conclusion that can be drawn at this point is that there is much that could be done with GLSA, and only time and further testing will determine whether it is an effective tool for solving difficult problems or if it just an also-ran.

REFERENCES

[1] Kondrak, Grzegorz and van Beek, Peter *A Theoretical Evaluation of Selected Backtracking Algorithms*, Proceedings of the 14th International Joint Conference on Artificial Intelligence, pgs. 541-547, 1995.

[2] Pearl, Judea *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1985.

[3] Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. *Introduction to Algorithms*, MIT Press, 1991.

[4] Prosser, Patrick *Hybrid Algorithms for the Constraint Satisfaction Problem*, Computational Intelligence, Vol. 9, No. 3, 1993.

[5] Sears, Francis W., Zemansky, Mark W., and Young, Hugh D. *University Physics*, 7th Ed., Addison-Wesley, 1987.

[6] Dowdy, Lawrence W. and Foster, Derrell V. *Comparative Models of the File Assignment Problem*, ACM Computing Surveys, Vol. 14, No. 2, June, 1982.

[7] Bernhard, Philip J. and Fox, Kevin L. *Experimental Evaluation of Techniques for Database File Assignment*, 2000.

[8] Sen, Sandip *File Placement Over a Network Using Simulated Annealing*, Association for Computing Machinery, 1994.

APPENDIX A – CODE FOR GLSA1 ALGORITHM

Note The following represents the code for the GLSA1 algorithm specific to the problem instance that was used for initial testing. While the code can be adapted for virtually any problem instance, there will be minor modifications that will be required prior to use depending on the particular nature of the problem at hand. The following is presented, and should be used, only as a reference for the general manner in which GLSA1 behaves, not as the coding for specific problem solutions.

```
/* Include the global definitions header file (this file will contain instance-
  specific definitions for the problem such as the solution space, solution
  vector, number of dimensions, RF function definition, ranges, etc.) */

#include "global.h"

/* Define the algorithmic environment */

#define DENS 1.2
#define DRAG 0.2
#define FRIC 0.57
#define GRAV 6.672
#define IVEL 10
#define ITER 100
#define MASS 1
#define RADI 2
#define SILH 0.1
#define THRE 2

int
glsa1 (solspace, strt_sol, sol, iter)
      unsigned int *iter;
      SPACE solspace;
      SOL strt_sol, sol;
{
  /* Define program variables */

  int acc, best_rf, frc, vel_sum = 0;
  unsigned int adj1, adj2, cnt = 0, res;
  SOL best, vel;

  /* Initialize the solution, directional force, and velocity vectors */

  for (dim = 0; dim < DIMS; dim++)
    {
      sol[dim] = best[dim] = strt_sol[dim];
      vel[dim] = rand () % (IVEL * 2);
      if (vel[dim] < IVEL)
        vel[dim] = 0 - vel[dim];
```

```
    else
      vel[dim] = (IVEL * 2) - vel[dim];
    if (abs (vel[dim]) <= THRE)
      {
        if (vel[dim] > 0)
          vel[dim] += THRE;
        else
          vel[dim] -= THRE;
      }
    vel_sum += pow (vel[dim], 2);
  }
vel_sum = sqrt (vel_sum);

/* Keep performing the algorithm until the combined velocity reaches zero */

while (vel_sum > 0 && cnt < ITER)
  {
    /* Reset the velocity sum */

    vel_sum = 0;

    /* Process each component of the current solution */

    for (dim = 0; dim < DIMS; dim++)
      {
        /* Calculate the gravitational attraction for the current dimension */

        adj1 = nextval (sol[dim], 'P', -1);
        adj2 = nextval (sol[dim], 'P', 1);
        if (solspace[dim][adj1] > solspace[dim][best[dim]])
          best[dim] = adj1;
        if (solspace[dim][sol[dim]] > solspace[dim][best[dim]])
          best[dim] = sol[dim];
        if (solspace[dim][adj2] > solspace[dim][best[dim]])
          best[dim] = adj2;
        frc = 0 - (GRAV * solspace[dim][adj1] *
          (solspace[dim][sol[dim]] + 1) / pow (RADI, 2));
        frc += GRAV * (solspace[dim][sol[dim]] + 1) * solspace[dim][adj2]
          / pow (RADI,2);

        /* Calculate the change in acceleration for the current dimension */

        res = (FRIC * MASS) + (0.5 * SILH * DRAG * DENS * pow (vel[dim], 2));
        if (vel[dim] > 0)
          frc -= res;
        else if (vel[dim] < 0)
          frc += res;
        acc = frc / MASS;

        /* Calculate the change in velocity for the current dimension */

        if (abs (acc) > abs (vel[dim]) / 2)
          acc = vel[dim] / 2;
        if (abs (acc) > 0)
          vel[dim] += acc;
        else
          if (vel[dim] > 0)
            vel[dim] -= MASS;
          else if (vel[dim] < 0)
            vel[dim] += MASS;
```

```
        if (abs (vel[dim]) <= THRE)
          vel[dim] = 0;
        vel_sum += pow (vel[dim], 2);

        /*  Calculate  the  new  solution  components,  accounting  for  the
          gravitational effects */

        if (vel[dim] > 0)
          sol[dim] = nextval (sol[dim], 'P', 1);
        else if (vel[dim] < 0)
          sol[dim] = nextval (sol[dim], 'P', -1);
      }

    /* Calculate the new velocity sum */

    vel_sum = sqrt (vel_sum);

    /* Increment the iteration count */

    cnt++;

   }

  /* Determine the best solution found and its RF value */

  for (dim = 0; dim < DIMS; dim++)
    sol[dim] = best[dim];
  best_rf = rfval (solspace, best);

  *iter = cnt;

  return best_rf;
}
```

APPENDIX B – CODE FOR GLSA2 ALGORITHM

Note The following represents the code for the GLSA2 algorithm specific to the problem instance that was used for initial testing. While the code can be adapted for virtually any problem instance, there will be minor modifications that will be required prior to use depending on the particular nature of the problem at hand. The following is presented, and should be used, only as a reference for the general manner in which GLSA2 behaves, not as the coding for specific problem solutions.

```
/* Include the global definitions header file (this file will contain instance-
  specific definitions for the problem such as the solution space, solution
  vector, number of dimensions, RF function definition, ranges, etc.) */

#include “global.h”

/* Define the algorithmic environment */

#define DENS 1.2
#define DRAG 0.2
#define FRIC 0.57
#define GRAV 6.672
#define IVEL 10
#define ITER 100
#define MASS 1
#define RADI 2
#define SILH 0.1
#define THRE 2

int
glsa2 (solspace, strt_sol, sol, iter)
      unsigned int *iter;
      SPACE solspace;
      SOL strt_sol, sol;
{
  /* Define program variables */

  int acc, best_rf, frc, vel_sum = 0;
  unsigned int adj1, adj2, cnt = 0, res;
  SOL best, vel;

  /* Initialize the solution, directional force, and velocity vectors */

  for (dim = 0; dim < DIMS; dim++)
    {
      sol[dim] = best[dim] = strt_sol[dim];
      vel[dim] = rand () % (IVEL * 2);
      if (vel[dim] < IVEL)
        vel[dim] = 0 - vel[dim];
      else
```

```
    vel[dim] = (IVEL * 2) - vel[dim];
   if (abs (vel[dim]) <= THRE)
    {
     if (vel[dim] > 0)
      vel[dim] += THRE;
     else
      vel[dim] -= THRE;
    }
   vel_sum += pow (vel[dim], 2);
  }
vel_sum = sqrt (vel_sum);

/* Keep performing the algorithm until the combined velocity reaches zero */

while (vel_sum > 0 && cnt < ITER)
 {
  /* Reset the velocity sum */

  vel_sum = 0;

  /* Process each component of the current solution */

  for (dim = 0; dim < DIMS; dim++)
   {
    /*  Calculate  the  gravitational  field  component  for  the  current
     dimension */

    adj1 = nextval (sol[dim], 'P', -1);
    adj2 = nextval (sol[dim], 'P', 1);
    if (solspace[dim][adj1] > solspace[dim][best[dim]])
     best[dim] = adj1;
    if (solspace[dim][sol[dim]] > solspace[dim][best[dim]])
     best[dim] = sol[dim];
    if (solspace[dim][adj2] > solspace[dim][best[dim]])
     best[dim] = adj2;
    frc = solspace[dim][adj2] - solspace[dim][adj1];

    /* Calculate the change in acceleration for the current dimension */

    res = (FRIC * MASS) + (0.5 * SILH * DRAG * DENS * pow (vel[dim], 2));
    if (vel[dim] > 0)
     frc -= res;
    else if (vel[dim] < 0)
     frc += res;
    acc = frc / MASS;

    /* Calculate the change in velocity for the current dimension */

    if (abs (acc) > abs (vel[dim]) / 2)
     acc = vel[dim] / 2;
    if (abs (acc) > 0)
     vel[dim] += acc;
    else
     if (vel[dim] > 0)
      vel[dim] -= MASS;
     else if (vel[dim] < 0)
      vel[dim] += MASS;
    if (abs (vel[dim]) <= THRE)
     vel[dim] = 0;
    vel_sum += pow (vel[dim], 2);
```

```
        /*  Calculate  the  new  solution  components,  accounting  for  the
          gravitational effects */

        sol[dim] = nextval (sol[dim], 'P', vel[dim]);
       }

     /* Calculate the new velocity sum */

     vel_sum = sqrt (vel_sum);

     /* Increment the iteration count */

     cnt++;

    }

  /* Determine the best solution found and its RF value */

  for (dim = 0; dim < DIMS; dim++)
    sol[dim] = best[dim];
  best_rf = rfval (solspace, best);

  *iter = cnt;

  return best_rf;
}
```