COMP 572

Project 1a

Sanjeev Shrestha

January 30, 2014

This is the first subproject of the first project. The goal of this project is to write and test a hillclimbing and simulated annealing search algorithms for two of the benchmark optimization problems.

For this subproject you only need to work on the Spherical and Schwefel functions, defined at: <http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume24/ortizboyer05a-html/node6.html#tabla:DefFunc>. (Note the first function labeled as Schwefel on this page is actually the double sum, which we are not using. We are using the Schwefel function defined immediately after the Rastigin function.)

Pay careful attention to the ranges of the functions. You will want to use those ranges both in creating initial individuals and in controlling the generation of neighbors, e.g. you don't want your GA 'wondering' out of the search space. Note that here the functions are all defined with 30 dimensions, e.g. P = 30 in the function definitions.

**Project Requirements:**

* Write a hill climbing algorithm and a simulated annealing algorithm to find the input values (x1,...,x30) that minimizes the two test function.
* **Project Write-up:**Write a short paper describing the results of your project that includes the following sections:
  + **Algorithm descriptions** - Description of the two algorithms. Be careful to include all of the details someone would need to replicate your work: how neighbors are generated in hill climbing, what the temperature schedule is for simulated annealing, etc.
  + **Results** - Table showing the results for both algorithms on both test functions.
  + **Conclusions** - If it’s not working, why not. And what are then next steps to complete the project.

1. **Function Overview**

* Sphere Function: The function is given below :-

|  |  |
| --- | --- |
| SN | Function Properties |
| 1 | $ f_{Sph}({\mathbf x}) = \sum_{i=1}^p x_i^2$ |
| 2 | $ x_i \in [-5.12, 5.12]$ |
| 3 | $ {\mathbf x}^*=(0,0,\ldots,0);\ f_{Sph}({\mathbf x}^*)=0$ |

* Schwefel Function:

|  |  |
| --- | --- |
| SN | Function Properties |
| 1 | $ f_{Sch}({\mathbf x})=418.9829 \cdot p + \sum_{i=1}^p x_i sin \left(\sqrt{\vert x_i\vert}\right)$ |
| 2 | $ x_i \in [-512.03, 511.97]$ |
| 3 | $ {\mathbf x}^*=(-420.9687,\ldots,-420.9687); \ f_{Sch}({\mathbf x}^*)=0$ |

1. **Algorithm Description**

* Hill Climbing Algorithm:

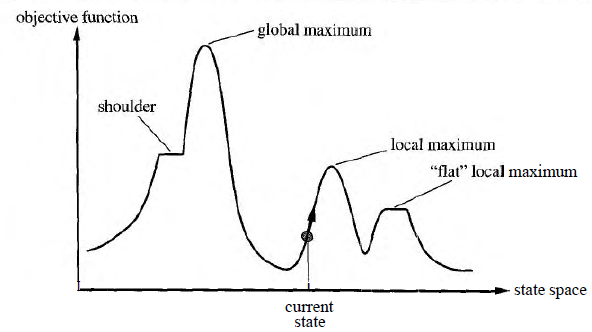


Fig 1: A one dimensional state space landscape in which elevation corresponds to the objective function. Hill Climbing search modifies the current state to try to improve it, as shown in the arrow.

For the Hill climbing algorithm we basically have a loop, we look through a set of neighbors until we find the most optimal solution or up until some number of iterations. After that, we try to move gradually up (or down) as if climbing up a hill (as shown in Fig 1). If the neighbors are not separated from each other by a considerable distance then there is a possibility of being stuck at the local optimum.

For Hill Climbing Algorithm:

/\*\*

\*

\* **@author** Sanjeev

\* **@return** solutionSet

\* **@param** probArray[] Array of Random Integers

\* **@param** functionName Either "Spherical" or "Schwefel"

\*

\*/

**public** **double**[] hillClimbing(**double** probArray[], String functionName) {

//Create an empty Neighbors Array

**double**[] neighbours = **new** **double**[*DIMENSIONS*];

//Loop Condition if good solution not found

**for** (**int** i = 0; i < 100000; i++) {

//Calculate the neighbor

neighbours = getNeighbour(probArray, functionName);

//if the current and Next Solution are same

//then move out of loop

**if** (Arrays.*equals*(probArray, neighbours)) {

**break**;

}

//check the fitness value in between current node

//and next node

**if** (checkFitness(neighbours, probArray, functionName)) {

//substitute current solution with next solution

probArray = neighbours;

}

}

//return the solution set

**return** probArray;

}

First, I have initialized the neighbors array, and then I have calculated the neighbor array with reference to the initial problem array, if the current problem array and calculated neighbor array are equal, then we have found a solution to the problem and we do not need to carry on with any further calculation. After that we check which array among the neighbor and the current array have better fitness and apply changes accordingly. Finally we return the solution array that we have calculated.

For Sphere Function:

**public** **double**[] getNeighbour4Sphere(**double**[] probArray, String functionName) {

//small change delta

**double** delta = 0.01;

//initialize the neighbor solution

**double**[] neighbour = **new** **double**[*DIMENSIONS*];

//loop and change values of neighbor

//solution based on problem set

**for** (**int** i = 0; i < probArray.length; i++) {

**if** (probArray[i] < 0.00

&& checkLimitCondition(probArray[i], functionName)) {

neighbour[i] = DoubleFormatter.*getFormattedDouble*(probArray[i]

+ delta, functionName);

} **else** **if** (probArray[i] > 0.00

&& checkLimitCondition(probArray[i], functionName)) {

neighbour[i] = DoubleFormatter.*getFormattedDouble*(probArray[i]

- delta, functionName);

} **else** {

neighbour[i] = DoubleFormatter.*getFormattedDouble*(probArray[i],

functionName);

}

}

//return changed neighbor array

**return** neighbour;

}

For calculating neighbor for a sphere function, we can see mathematically that the points converge on a global optimum when values of x tend to 0.0. Let us consider a small change delta (=0.01 in this case for 2 point precision). Let us create an empty neighbor array, and then we loop through the neighbor and populate it with small changes for initial problem array. The concept for this is given below:-

* 1. If value of x(i) is positive and lies in between the range then we need to subtract delta from it.
  2. Else if value of x(i) is negative and lies in between the range then we need to add delta to it.
  3. Else the value must be zero for x(i) and we do not need to make any changes.

Using this concept, the corresponding neighbor is created and returned.

For Schwefel Function:

**public** **double**[] getNeighbour4Schwefel(**double**[] probArray,

String functionName) {

RandomArrayGenerator newRandomNumGen = **new** RandomArrayGenerator();

**double** delta = newRandomNumGen.randomInRange(-512.03, 511.97);

delta = DoubleFormatter.*getFormattedDouble*(delta, functionName);

**double**[] neighbour = **new** **double**[*DIMENSIONS*];

**for** (**int** i = 0; i < probArray.length; i++) {

**if** (checkLimitCondition(probArray[i] - delta, functionName)

&& Schwefel.*getSingleFitnessValue*(probArray[i] - delta) < Schwefel

.*getSingleFitnessValue*(probArray[i])) {

neighbour[i] = DoubleFormatter.*getFormattedDouble*(probArray[i]

- delta, functionName);

} **else** **if** (checkLimitCondition(probArray[i] + delta, functionName)

&& Schwefel.*getSingleFitnessValue*(probArray[i] + delta) < Schwefel

.*getSingleFitnessValue*(probArray[i])) {

neighbour[i] = DoubleFormatter.*getFormattedDouble*(probArray[i]

+ delta, functionName);

} **else** **if** (Schwefel.*getSingleFitnessValue*(delta) < Schwefel

.*getSingleFitnessValue*(probArray[i])) {

neighbour[i] = DoubleFormatter.*getFormattedDouble*(delta,

functionName);

}

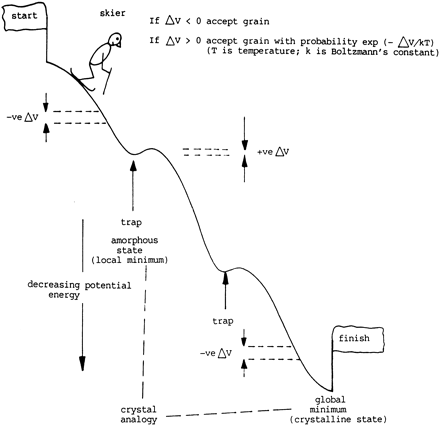
}

**return** neighbour;

}

For the Schwefel function, I tried using the approach with delta as in the Spherical Function but it did not seem to create better converging solutions; so I made some tweaks with the current delta. I generated a Random Delta in between the given range. Then I looped through the problem Array, I checked for the limiting conditions and made a SingleValueFitness function (as discussed in class the fitness values are independent of each other from one dimension to other). So if we get the better values then we substitute it else we do not make changes to neighbor. When single neighbors have been fitness then the whole fitness of the function drastically increases resulting in better and accurate solution sets.

* Simulated Annealing:



Simulated Annealing take the physical annealing process as an analogy for its search mechanism. Let us consider two states namely s1 and s2. T is the time duration for which period the annealing process occurs. As in the natural phenomenon, the time is decreased slowly to get the optimal solution for the procedure. Simulated Annealing makes a leap during times so that it may not get stuck at local optimum and move toward the Global optimum by taking leaps even though the solution may not seem favorable at first. This calculated guess is performed on the basis of the difference of energies in between the states and if the generated probability is greater than a random probability generated in between the range [ 0 , 1 ].

For Sphere Function:

**public** **double**[] simulatedAnnealing4Sphere(**double**[] s1) {

**double**[] s2 = **new** **double**[30];

**for** (**double** T = 100; T >= 0.01; T -= 0.01) {

s2 = getNeighbour(s1, Spherical.*FUNCTION\_NAME*);

**if** (checkFitness(s2, s1, Spherical.*FUNCTION\_NAME*)) {

s1 = s2;

} **else** {

**if** (*Prob*(Spherical.*getfitnessValue*(s1),

Spherical.*getfitnessValue*(s2), T,

Spherical.*FUNCTION\_NAME*)) {

s1 = s2;

}

}

}

**return** s1;

}

For the sphere function, we initialize a state s2, we loop over a Temperature that is decremented gradually by a factor of 0.01. We calculate the neighbor of the state s1 . We check for the fitness in between the states, if s2 has better fitness than s1 is replaced by s2. If that is not the case then, we calculated the Probability on the basis difference between the energies and temperature and see if the probability is greater than a random probability inn between the range [0,1]. If this is the case then s2 is replaced by s1 anyways. This goes until the steady state is found.

For Schwefel Function:

**public** **double**[] simulatedAnnealing4Schwefel(**double**[] s1) {

**double**[] s2 = **new** **double**[30];

**for** (**double** T = 100; T >= 0.01; T -= 0.01) {

s2 = getNeighbour(s1, Schwefel.*FUNCTION\_NAME*);

**if** (checkFitness(s2, s1, Schwefel.*FUNCTION\_NAME*)) {

s1 = s2;

} **else** {

**if** (*Prob*(Schwefel.*getfitnessValue*(s1),

Schwefel.*getfitnessValue*(s2), T, Schwefel.*FUNCTION\_NAME*)) {

s1 = s2;

}

}

}

**return** s1;

}

For the Schwefel function, we initialize a state s2, we loop over a Temperature that is decremented gradually by a factor of 0.01. We calculate the neighbor of the state s1 . We check for the fitness in between the states, if s2 has better fitness than s1 is replaced by s2. If that is not the case then, we calculated the Probability on the basis difference between the energies and temperature and see if the probability is greater than a random probability inn between the range [0,1]. If this is the case then s2 is replaced by s1 anyways. This goes until the steady state is found.

For Probability of exchange between values (i.e, the Energy evaluation function):

**private** **static** **boolean** Prob(**double** e1, **double** e2, **double** T,

String functionName) {

**double** P = 0.0;

**if** (functionName.equals(Spherical.*FUNCTION\_NAME*)) {

P = 1 / (Math.*exp*((e1 - e2) / T));

**return** P > 0.5;

} **else** **if** (functionName.equals(Schwefel.*FUNCTION\_NAME*)) {

P = (Math.*exp*(((e1 - e2)) / T));

**return** P > (**new** RandomArrayGenerator().randomInRange(0, 1));

}

**return** **false**;

}

I have calculated the energies of the function and calculated the necessary value for substitution if required and the probability is checked randomly in between 0 and 1.

1. **Results**

* Test Results for Hill Climbing with Sphere Function:

|  |  |  |
| --- | --- | --- |
| SN | Test Settings | Results |
| 1 | Initial Problem Set | [-4.96, 4.53, -1.36, -4.77, -3.57, 3.05, 4.93, 3.05, -0.34, -3.71, 0.27, -1.52, -2.01, -0.13, -2.72, -4.72, 4.18, -3.63, -3.63, -2.68, -2.38, -3.56, -3.14, 1.93, -1.38, 0.48, -4.63, 3.72, -2.3, 3.6] |
| 2 | Final Problem Set | [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] |
| 3 | Test Cycles | 496 |
| 4 | Time Taken | 0.39948889 seconds |

* Test Results for Hill Climbing with Schwefel Function:

|  |  |  |
| --- | --- | --- |
| SN | Test Settings | Results |
| 1 | Initial Problem Set | [-293.3018, 421.3168, -30.922, -19.6142, 51.1511, 256.0336, -243.8322, -184.7635, -493.9113, -316.8597, -140.9231, -405.0576, -339.4058, 126.9058, -413.1691, -182.9387, 117.365, 162.9681, 477.6649, -235.2159, -25.6644, -212.164, 391.8585, 126.8865, 89.9814, 141.3596, 72.8909, -226.0852, -412.1661, 488.9527] |
| 2 | Final Problem Set | [-420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753, -420.9753] |
| 3 | Test Cycles | 100000 |
| 4 | Time Taken | 25.906125718 seconds |

* Test Results for Simulated Annealing with Sphere Function:

|  |  |  |
| --- | --- | --- |
| SN | Test Settings | Results |
| 1 | Initial Problem Set | [0.05, 5.04, 0.23, 2.99, 4.65, 4.08, -0.98, 3.07, 1.16, -1.02, -2.97, 2.72, -3.95, 1.13, 4.67, 2.85, -1.96, -2.39, 1.54, -2.54, -5.09, 4.21, 0.77, -3.23, 3.06, -5.0, 1.16, -3.67, 4.87, 1.67] |
| 2 | Final Problem Set | [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] |
| 3 | Time Taken | 0.754850905 seconds |
| 4 | Fitness Value | 0.0 |

* Test Results for Simulated Annealing with Schwefel Function:

|  |  |  |
| --- | --- | --- |
| SN | Test Settings | Results |
| 1 | Initial Problem Set | [329.1616, -144.6644, -145.8046, 434.3692, -146.1963, 452.1932, -511.9206, -484.3875, -366.616, 507.0114, 46.5659, 78.0203, -357.721, 328.0272, -310.2012, 331.0047, 279.2151, -37.601, -286.8136, -262.5958, 498.55, -268.157, -108.9874, -469.9199, -245.1218, 79.9155, 507.3271, 57.2359, -431.8811, -333.6531] |
| 2 | Final Problem Set | [-420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743, -420.9743] |
| 3 | Time Taken | 3.958012082 seconds |
| 4 | Fitness Value | 5.0E-4 |

1. **Conclusion**

In conclusion, the Hill Climbing and the Simulated Annealing Algorithms were used for solving benchmark problems. Both Hill Climbing and Simulated Annealing did very well for the Spherical Function. For the Schwefel function however, Simulated Annealing did a better job than Hill Climbing Algorithm. The Hill Climbing algorithm sometimes got stuck at even the starting point when used the delta transitions. For delta transitions had to be made random for the Schwefel function just so we could make the solution better.

References

* Benchmark Problems. (n.d.). *Benchmark Problems*. Retrieved January 30, 2014, from http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume24/ortizboyer05a-html/node6.html
* Hill climbing. (2014, January 25). *Wikipedia*. Retrieved January 30, 2014, from http://en.wikipedia.org/wiki/Hill\_climbing
* How to convert nanoseconds to seconds using the TimeUnit enum?. (n.d.). *java*. Retrieved January 30, 2014, from http://stackoverflow.com/questions/924208/how-to-convert-nanoseconds-to-seconds-using-the-timeunit-enum
* Project1a. (n.d.). *Project1a*. Retrieved January 29, 2014, from http://www2.cs.uidaho.edu/~cs472\_572/s14/GAProjectB.html
* Russell, S. J., & Norvig, P. (2014). INFORMED SEARCH AND EXPLORATION. *Artificial intelligence: a modern approach* (3rd. ed., Pearson new international ed., pp. 111-116). Harlow: Pearson.
* Schwefel, H. (1995). Hill climbing Strategies. *Evolution and optimum seeking* (pp. 23-25,160). New York: Wiley.
* Simulated Annealing for beginners. (n.d.). *Simulated Annealing for beginners*. Retrieved January 30, 2014, from http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6
* Simulated annealing. (2014, January 29). *Wikipedia*. Retrieved January 30, 2014, from http://en.wikipedia.org/wiki/Simulated\_annealing
* public perrygeo / python-simulated-annealing. (n.d.). *GitHub*. Retrieved January 30, 2014, from https://github.com/perrygeo/python-simulated-annealing