**Function Maximization using Simulated Annealing Algorithm**

**Elements of Artificial Intelligence**

Student: Vanessa Maria Mercea

Teacher: Andrzej Bargiela

**Table of contents**

1. Introduction on Simulated Annealing ……………………… 2
2. Presentation of the task ……………………………………... 3
3. Experimentation ……………………………………………. 4
4. Flow and Structure of the program ………………………… 7
5. Bibliography ………………………………………………. 10
6. **Introduction on Simulated Annealing**

The word annealing has its origins in metallurgy. It is a very slow and controlled process that consists of two stages: first, the material is heated to a very high temperature so that it is more malleable and workable with, as it’s state is disturbed. Then, it is slowly cooled so that the atoms can settle in the appropriate positions. Hence, they would reach such a state that is called “lowest energy state”.

In computer science, it has been observed that applying this “tendency to reach a low energy state” to many situations has concluded with pleasing results. The main applicability of this process regards optimization problems.

The annealing process in computer science works in the following way:

First, the programmer must select an initial state, and then randomly choose a succeeding state. This should happen, similarly to metallurgic annealing, while decreasing the temperature by using a function. After generating a new state, the programmer needs to choose whether to keep it or not based on the variation of “energy” between the old state and the new. Depending on the problem, this energy may have a different meaning. If the new state is a better state for the problem, it should be chosen. Until now, these previous steps are describing the Hill Climbing Algorithm. Simulated Annealing, on the other hand, differs because it also accepts bad moves – as in the chemical process of moving atoms. These bad moves are accepted or rejected based on a probability, using Boltzmann distribution that describes the current state:

In computer science, the Boltzmann constant, k, can be neglected, and E is the difference of energy between the states. The sign of the input to the exponential function must always be negative so that it remains in the subunit range. This function must be compared to a random number in the [0, 1] interval.

When the Temperature is very high, the function output will be a number very close to 1, so it will be very probable that that number is larger than that subunit random. Therefore, at high temperatures, more worsening states will be accepted. But as the temperature decreases, the function output will get closer and closer to 0, so the probability of accepting a worsening move will be very low. This way, the process will be very unstable at first, and then slowly become stable.

In a real-life situation, this process would be repeated for an infinite amount of times, and stop when the temperature would reach null-point. In computer science, a stopping condition must be found depending on the problem, and on the function chosen for decreasing the temperature.

An important factor that defines Simulated Annealing, is that it is a metaheuristic, which means that based on the problem, the meaning of the parameters and of the variables involved change, and sometimes even the algorithm itself. Such elements to consider are: the range for the next move, the temperature function, the number of iterations, the temperature starting/ending point.

1. **Presenting the task**

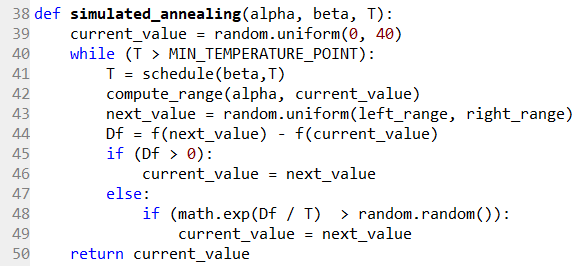
This coursework task is to perform an experimentation by tuning the parameters for the simulated annealing algorithm presented above. This should be done for an optimization problem, which is maximizing the following function, defined on the interval **x ∈ [0, 40]**:

Before proceeding into analyzing the code for this algorithm, it is best to give meaning to the variables presented in the first part of this coursework.

Thus, in this situation, the energy delta will be the difference between the output of the function for the succeeding input and the output for the current input. If the movement is made towards a higher point of the graph of the function, it should be accepted. If it is made towards a lower point, whether it will be accepted depends on the Boltzmann function probability. The changes made regarding the Simulated Annealing function consist in: the range for the next move - α, the temperature schedule, the temperature decrease coefficient - β, the starting value of the temperature - T, and the number of iterations for each temperature.

1. **Experimentation**

For getting the statistics that will be presented in this experimentation, the correct answers the algorithm was returning were counted. This was done for 100 different starting points. The tuning consisted in assigning different values of the given parameters, and in slightly modifying the algorithm. At first, there were difficulties with the floating point, as the values returned by the algorithm were very precise. That is why the comparison is done by rounding the 3rd decimal of the answer. Instead of iterating for a certain number of steps, the temperature is decreased until a minimum value is reached. In that way, changing the parameters of the algorithm would not require reconsidering the number of iterations.

Initially, Lundy’s method regarding temperature control was used: one iteration for each temperature, but a very slow decrease.

Lundy’s formula for the schedule:

**=**

was too slow for computation, regardless of the parameters. Modifying the minimum temperature point would only minimize the chances of getting a correct answer. There are no statistical results for this situation, as it would not compute in a reasonable period.

To fix this problem, the function was changed to the geometric progression:

**=**

 In this way, the speed in the decrease of temperature would become easily controllable.

At first, the neighborhood coefficient was set the maximum value, 1.0, and β to a high value, to make the decrease very slow. The first parameter that was examined was the starting temperature, T. This was exponentially reduced each time, after computing the statistics.

For α = 1.0, β = 0.99, T = 100000, in 3 runs, the results were: 90% 83% 94% correct answers. This was clearly very unstable. The temperature was lowered to see if it would make the results more stable. For α = 1.0, β = 0.99, T = 10000, in 3 runs, there were 95%, 88%, 79% correct answers.

The lowering was repeated so that the temperature reached 10. It is important to see whether this is the cause of the instability. For α = 1.0, β = 0.99, T = 10, the output was 83%, 91%, 82%. When checking the csv file to see the actual results, the answers were never stuck in the local maximum, they all were close to the correct answer. This means that the error came from the computation, so the problem might be related to the neighborhood coefficient. If the range is too big, it may generate succeeding values very far from the initial point, and cause jumps, and imprecision.

For α = 0.7, β = 0.99, T = 10, in 3 runs, the results were 93%, 93%, 90% correct. This was more stable, and the local optimum issue did not occur. The attempt to modify the temperature now, showed no considerable difference between T = 10 or T = 100. The computation errors were still there, so α became the current examined parameter. The output was examined for each gradually decreasing value of α.

For α = 0.5, β = 0.99, T = 10, output showed 95%, 97%, 96% correct answers, and when changing the temperature to 100, it showed 98%, 97%, 98% correct. This increase in percentages may be a coincidence, since the algorithm is relying on a random value comparison. This was the best situation so far, but some answers were stuck in the local optimum. By decreasing the α, the computation errors would completely disappear, and the only problem that would remain would be the occurrence of local maximum “freezing”.

This issue could be fixed using a different strategy. So, for α = 0.3, β = 0.99, T = 10, very good precision was obtained, as no wrong answers were caused by precision. α was then adjusted around the value of 0.3 until the results were satisfying: at α = 0.35, the output indicated 100%, 100%, 98% correct answers. Very good rate, but at this moment, the annealing process was getting stuck in local maximum more often. A small enough α is needed so that the next move does not jump to imprecise values, but it also must be large enough so that the local optimum problem does not intervene. β was a very large value, decreasing the temperature very slowly. Therefore, it had no impact on this “freezing” in local maximum effect.

It appears that the neighborhood cannot solve this problem, and the temperature has no significant influence. This means that we must also consider the changing the schedule or the number of iterations. The issue might be that the function is not decreasing slow enough for Lundy’s one-iteration-method, since the algorithm was using a schedule that was much faster than Lundy’s formula. Therefore, 10 iterations were added for each temperature, to allow stabilization – to let the system settle at the current temperature. Trying to run the code after this attempt took too long and it would not reach a result. The first solution for making the code run faster would be to decrease β, but because only a small number of iterations were introduced for each temperature, it was important to keep β high so that many temperature values are considered.

Thus, β was decreased to 0.95. Although this is not a large decrease, it had a significant impact on the computation time. After doing this, the results were 100%, 99%, 100% correct. Even though it rarely occurred, the local optimum problem would not be removed. The cause was the faster decrement of temperature. This means that it was still not stable enough at each temperature, so the number of iterations was increased by 5. Adding many iterations for each temperature significantly increases computation time so it’s important not to add too many.

Now that the code had 15 iterations for each temperature, and α = 0.35, β = 0.95, T = 10, the algorithm only returned 100% correct answers, no matter the number of runs.

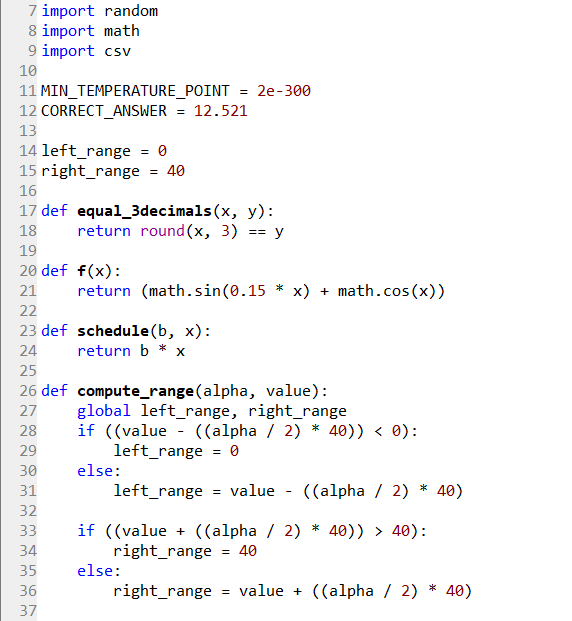
The version with α = 0.35, β = 0.99, T = 10, geometrical progression schedule and one iteration per temperature, is slightly less time-consuming than this final version, and if efficiency is the top-criteria, that is the one that should be chosen.

The final version, however, is a good compromise between effectiveness and efficiency.

1. **Flow and structure of the program**

The code presented here is the final form, after the experimentation has concluded to a satisfying result. It was written in Python 2.7. The differences between this and the initial version are visible in the experimentation point.

The program has these 4 functions that used in the simulated annealing:



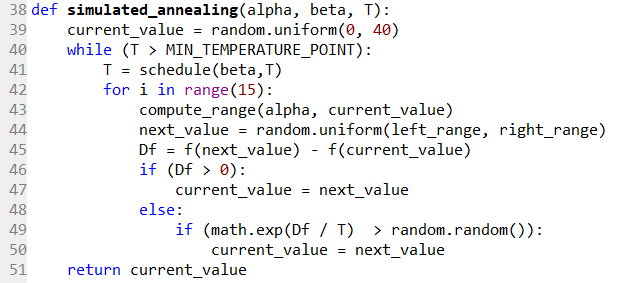
Because the algorithm uses many operations with floating point, the answers it returns should be 3-decimal-rounded before being compared to the correct one – that’s why the equal\_3decimals function is used.

For the temperature schedule, the geometric progression function was chosen:

**=**

This was done that the cooling process would be slower and slower as the temperature approached 0. Because setting the minimum temperature point to 0 would cause an overflow in the exponential function, a suitable temperature for the stopping condition was chosen. That is: 2e-300, which is a very small floating-point number: 2 \* , that the exponential function can still work with. By using this and the temperature schedule presented, the answer is predicted within 68892 temperature iterations for a starting temperature of 10. This is multiplied by 15, due to the iterations for the stabilization of temperature. It is a reasonable number for a decent computation time.

The compute\_range function sets the interval for the next move based on alpha, which is the range coefficient, and on the current value. It is needed, in order to compute the next random point located within the neighborhood. For instance, if alpha is 50%, then the next point will be generated as a random number from the interval [current point – 25%, current point + 25%], where 100% would be 40 units.

This is the simulated annealing function:

The input includes: the range coefficient – alpha, the temperature function factor – beta, and the starting temperature value, T.

Before proceeding into the iterations, a random starting value for x is chosen in the [0, 40] interval. Then, while the temperature has not reached the minimum temperature point that was set globally, a new temperature is computed with the temperature schedule.

For each temperature value, 15 iterations are used so that the system can stabilize and avoid local optimum issues. Inside those, the available range near the current point is set, so that the next value can be generated randomly in that area.

We want to accept values that lead us to a higher point in the graphic of the function, so if delta F is above 0, we accept that value as good. Otherwise, we use the Boltzmann distribution and check the probability to accept the new point. The random.random() function gives us a random number in the [0, 1] interval.

We are using , and not as the initial formula, because at that point, our Df is negative, and T is positive. We need a negative value for the exponent, to get an output within the [0, 1] interval, so it is correct to use it in this manner.

After the temperature has reached the minimum point, the current point is passed as output, and that is the x-axis value of the highest point in the graph.

1. **Bibliography**
2. The courses for Elements of Artificial Intelligence
3. <https://en.wikipedia.org/wiki/Annealing_(metallurgy)>
4. <https://en.wikipedia.org/wiki/Simulated_annealing>
5. <https://www.youtube.com/watch?v=eBmU1ONJ-os>
6. <https://www.coursera.org/learn/vlsi-cad-layout/lecture/YEcWn/simulated-annealing-placement>
7. <http://www.wolframalpha.com>