Simulated Annealing Algorithm

By Tannu Sharma

INTRODUCTION

We are trying to solve a particular instance of block placement problem, which is an NP hard problem. Our objective is to come up with a heuristic based solution that gives us an optimal solution. One of the approaches is to use Simulated Annealing Algorithm. This algorithm begins at a particular state in solution space and then iterates several times, disturbing this state, to get to a better solution. We simulate an annealing process where heating makes molecule more mobile and allows them to jump from a steady state, followed by a cooling step. At the cooling step we expect the molecules to settle down in a more stable state. This is what we intend to do over the solution space. We do a greedy descent to find an optimal solution but we make sure we do not get trapped in a local optima, by jumping out of it randomly during annealing.

REPRESENTATION

Module Class: A module will be represented by an object which contains a label, current cell coordinates (x, y) and the list of other modules it is connected to. Keeping the list of connected modules makes the cost calculation faster and simpler.

Solution Matrix (3 x 3). This is a 2d integer array, where each element at (i, j) represents the module number, which is placed in the grid cell (i, j) of the placement grid. We fill this matrix to get our solution. There is a global Solution matrix, where the current solution is stored and there is a temporary Solution Matrix which contains a configuration we are testing in Metropolis function.

"Modules" map: A mapping of label to corresponding module object for quick access of object.

FUNCTIONS

Calculating Cost: Following is the simple iteration method to calculate cost. We calculate the Manhattan Distance which is sum of distance in x coordinate and y coordinate respectively

Input: Modules.
Cost=0

Iteration (module in Modules)
Iteration (module2 in module->ConnectedModules)

Cost+= ManhattanDistance(module,module2)

Return Cost

Push Solution: Since Cost function assumes that each module objects knows its own position (i, j) ,We have to populate that data according to some Solution Matrix. Hence before invoking Cost function we must *pushSolution* to our Module objects. This helpful when we compare 2 different solution matrix costs. This function also instantiates module objects during first run

Neighbor Function: This function is invoked from the metropolis function to find and alternate solution to the current Solution. This step is also described as disturbing the current solution. We populate a new Solution Matrix and swap some of the modules in cell. The "fixed "constraint on C9 has to be enforced.

Legalization: Only one constraint is added to enforced fixed location for C9 (3, 3) cell.

Metropolis: This is the core of simulated annealing. It searches the solution space and does greedy descent as well as jumps out of a local optima, under certain conditions. The following Algorithm explains how this works. It runs M iterations at the temperature T. It finds a new solution on every iteration, which gets accepted if it improves on the previous solution or a random number is less than a particular function of T. The second condition allows us to jump out of a local optimal solution. Since random number between 0 and 1 is compared to the value of *exp(-deltaH/T)*, where deltaH is difference in cost of the two solutions. So, higher the temperature (T), better the chance of jumping out of a local optimal solution. We gradually reduce the value of T for each call of metropolis and eventually the system

settles at an optimal solution, which has higher chance of being a global optimal solution.

Input: S (current solution matrix), M (total time at this T), T (temperature)

Do

```
  news = neighbor ( S)
  deltah = cost(news) – cost (s);
  if( deltah<0 or g < exp( random < exp(-deltah/T)) )
    Accept : Copy newS to S
  M=M-1
While( M>0)
```

Simulated Annealing: The following algorithm explains the iteration involved in simulated annealing. We go through M time steps and at each iteration, call to Metropolis is made. After every iteration Temperature T is reduced by a factor of alpha and M by a factor of beta. As time passes by Temperature is reduced (as alpha < 1). Once maxTime has passed, we terminate and the solution matrix contains our final Solution.

Input: Modules, Nets (connectivity between modules), T0, alpha beta , MaxTime.

```
Build-Module Object Model
        Solution <- initial random assignment.
        Build connectivity information from Nets
       time =0;
        Do
           Metropolis (Solution, T, M)
           time+= M
           T=T* alpha
           M=M*beta
        while (time<MaxTime)
```

RESULT

After running and testing the algorithm with the following values

alpha = 0.9
M = 10 (starting value)
T= 10 (starting value)
beta = 1;
MaxTime = 1000;

Also used as terminal condition (T>5), ie 50 % of the starting value. The minimum cost obtained is 8 and one such optimal solution is:

5 6 7
3 2 1
4 8 9

Steps to execute:
% ./ts_sim_anneal.o
Placement of cells for a given netlist:
****
6 4 7
5 3 1
2 8 9
****
Cost of total wire length is minimized to 8
Total hill climbing moves: 165

Hill climbing: the number usually varies on each execution and is in the range of 160-190