# ASSIGNMENT-1

Sai Chaitanya Nandipati

M08905912

## Simulated Annealing:

Partitioning is a combinatorial problem. For combinatorial problems finding the solution is NP-Complete. To deal with such problem simulated annealing is used. It is a probabilistic function for estimating the global optimum. It mainly emphasis on accepting the inferior moves to come out of the local optimum points and to reach the global optimum.

**Flow of the Simulated Annealing Algorithm:**

The objective of the partitioning is to partition the nodes into two equal number of subsets such that the total number of connections between the two sets of nodes is minimum.

As shown in the flowchart the initial partition is randomly chosen using SRAND function in the C++ STD library. Seed for the SRAND function is the system time that is derived using time (0) function. SRAND function generates random values based on the seed value provided as an input. With different seeds it produces different numbers. Time (0) function produces the system time that keeps on changing. So, I find it as a best seed to produce the random numbers.

Initial temperature and maximum number of moves are specified to iteratively check for the local optimum point so as to reach the global optimum.

Calculating gain or the cost is the heart of the algorithm. We need to optimize it as much as possible. The time complexity of cost function that I have implemented is O (n) in the worst case. For calculating the gain heuristic from the KL algorithm is used. N is the number of nodes.

Moves are accepted based on the two gain and probability function. Move with positive gain is always accepted and if the gain is negative then a random number is generated between 0 and 1. And, if the random number is less than the Boltzmann function then move is accepted else move is not accepted.

**Random number between 0 and 1 < Exponential (Gain/ K* Initial temperature)**

It can be observed that the function accepts the inferior moves. The function seems to be random but it is not completely random. It can be observed from the function that at higher temperatures the exponential value will be close to 1 and as temperature decreases it moves close to 0. So the function accepts more number of inferior moves at the higher temperatures and as the temperature decreases probability of accepting inferior moves decreases. The value of K should be tuned such that exponential value stays within 0 to 1.

If the move is accepted, then the two nodes are swapped.

This process repeats iteratively till specified moves are completed.

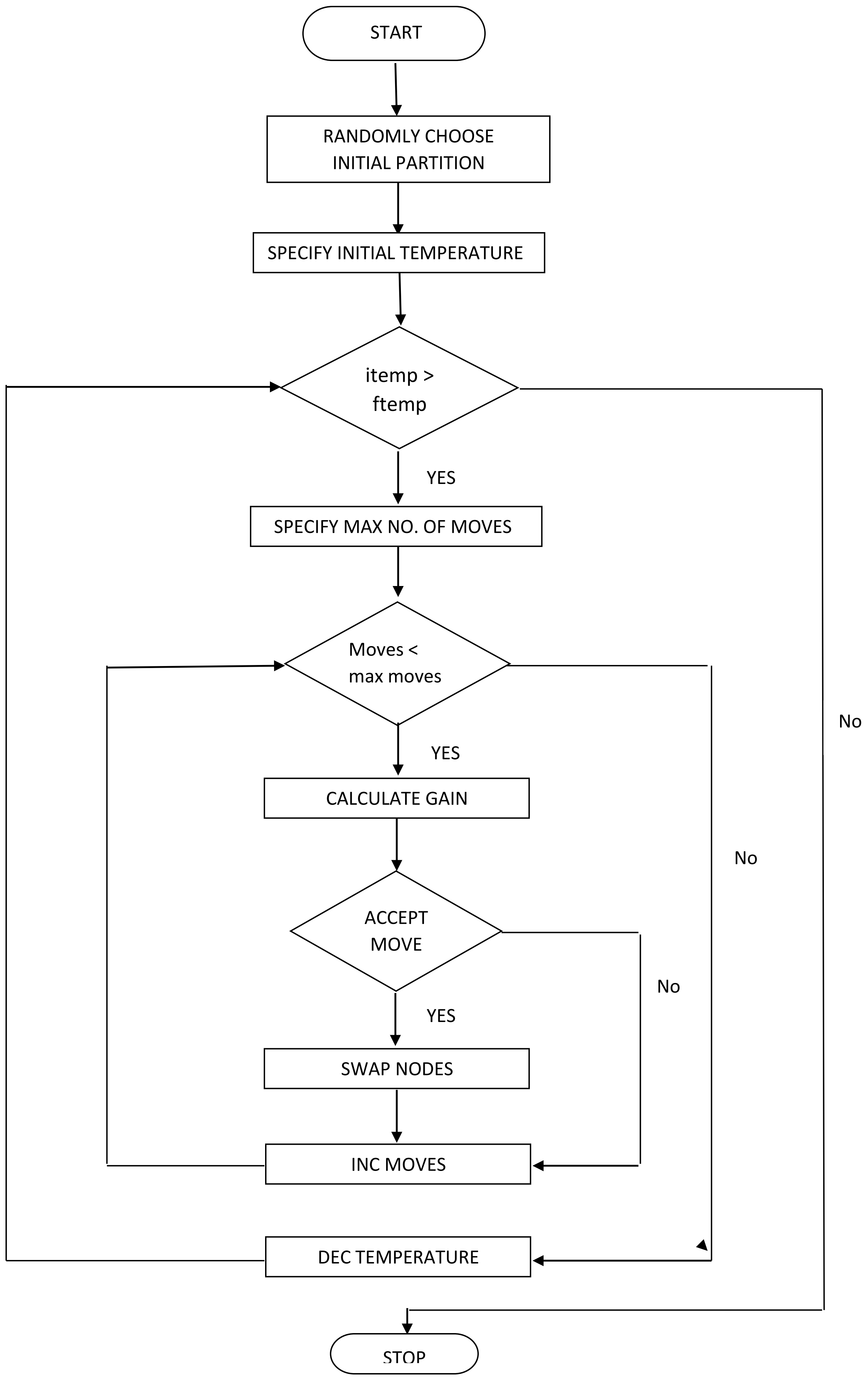
**Data structures used:**

**For storing initial partition:** 1-D vector.

vector<int> initialpartition;

**For storing adjacency matrix:** 2-D vecors.

vector<vector<int> > rowindex;


START
RANDOMLY CHOOSE INITIAL PARTITION
SPECIFY INITIAL TEMPERATURE
itemp > ftemp
YES
SPECIFY MAX NO. OF MOVES
Moves < max moves
YES
CALCULATE GAIN
ACCEPT MOVE
YES
SWAP NODES
INC MOVES
No
No
No
DEC TEMPERATURE
STOP

I feel my program is optimized and good because it can complete even the higher test benches within 20 minutes of time.

| **Benchmark** | **Final cut set** | **Seed value** | **Execution time** | **Memory requirement** |
|---|---|---|---|---|
| **B1** | 11 | 1456163130 | 209 seconds | 5248 Kb |
| **B2** | 234 | 1456163116 | 377 seconds | 5520 Kb |
| **B3** | 110 | 1456162312 | 337 seconds | 7440 Kb |
| **B4** | 1095 | 1456162256 | 604 seconds | 14720 Kb |
| **B5** | 3247 | 1456161261 | 911 seconds | 9028 Kb |
| **B6** | 3974 | 1456160565 | 560 seconds | 36128 Kb |

## Observations:

1. Increasing the number of moves results in better optimized cut set.

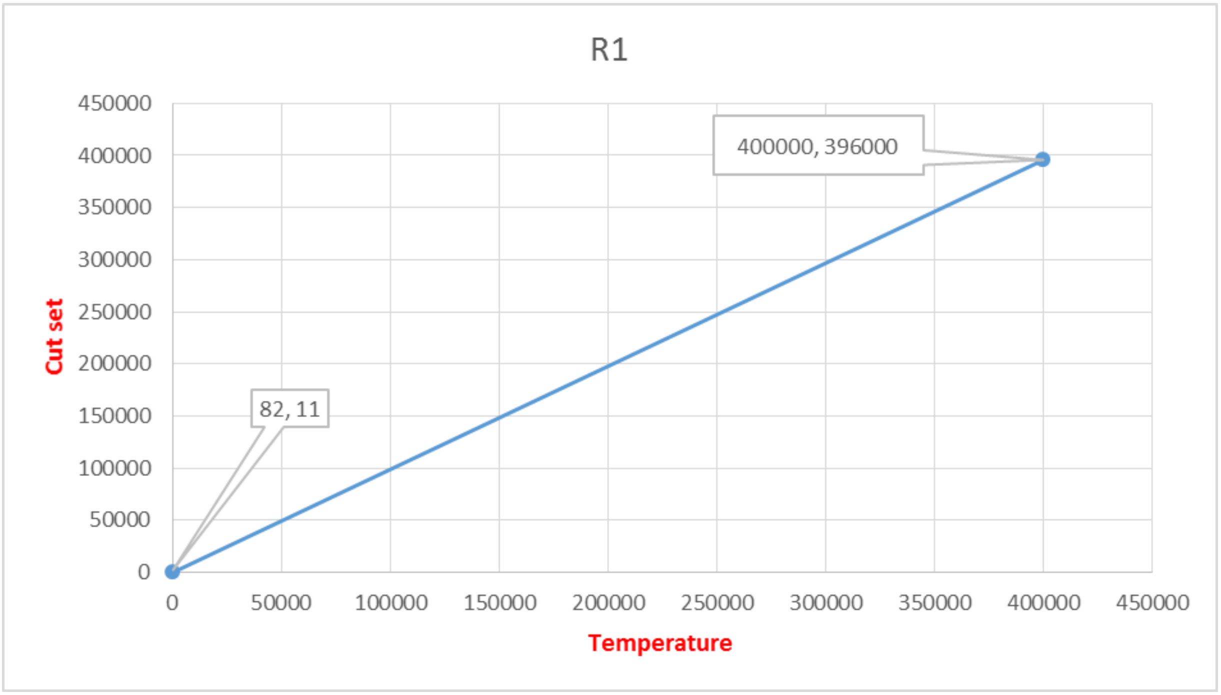
2. Even though inferior moves are not accepted it leads to global optimum for given set of test benches. It may not be true for all the cases.

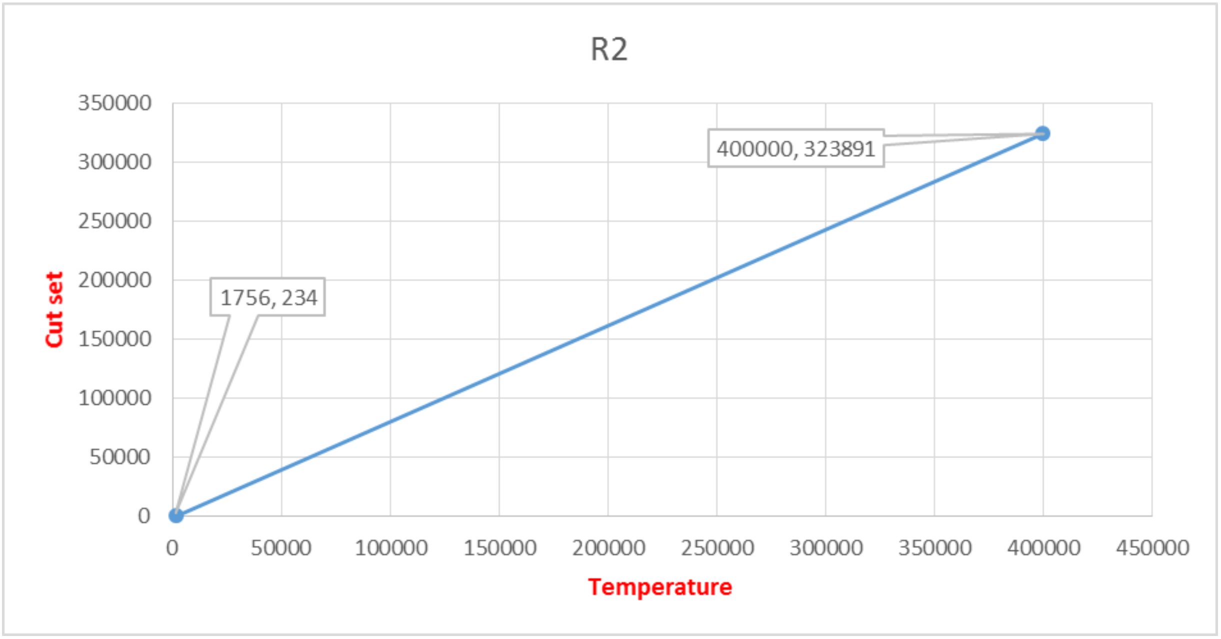
3. Cost function is the heart of the program. Optimize it as much as possible.

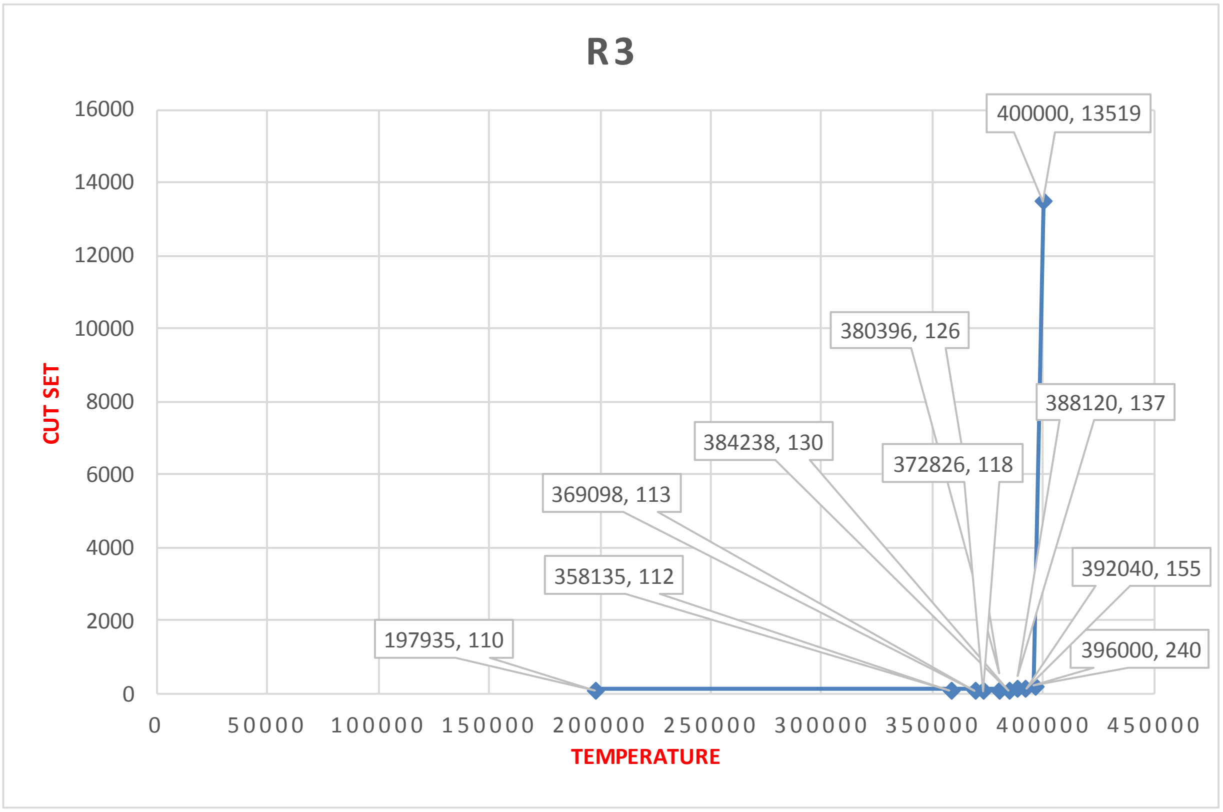
4. Bench B6 execution time is minimum compared to B5 and B4 execution time.

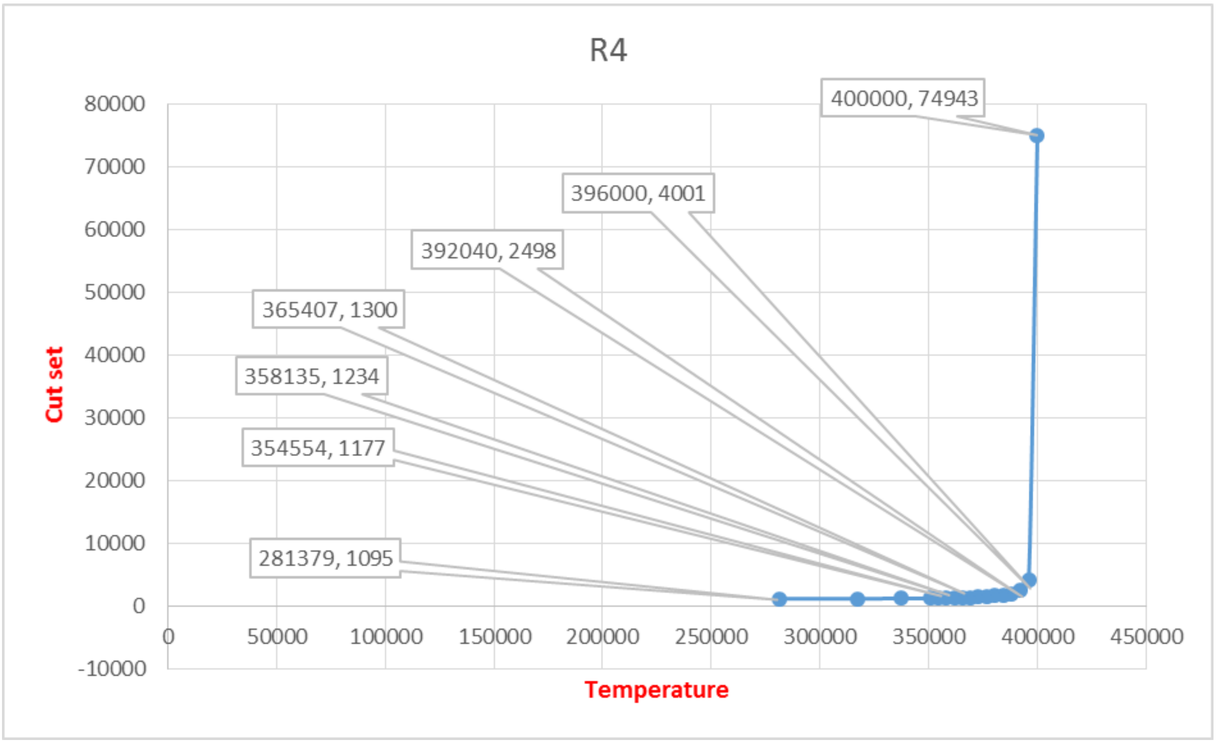
**Trajectory of Simulated Annealing:**

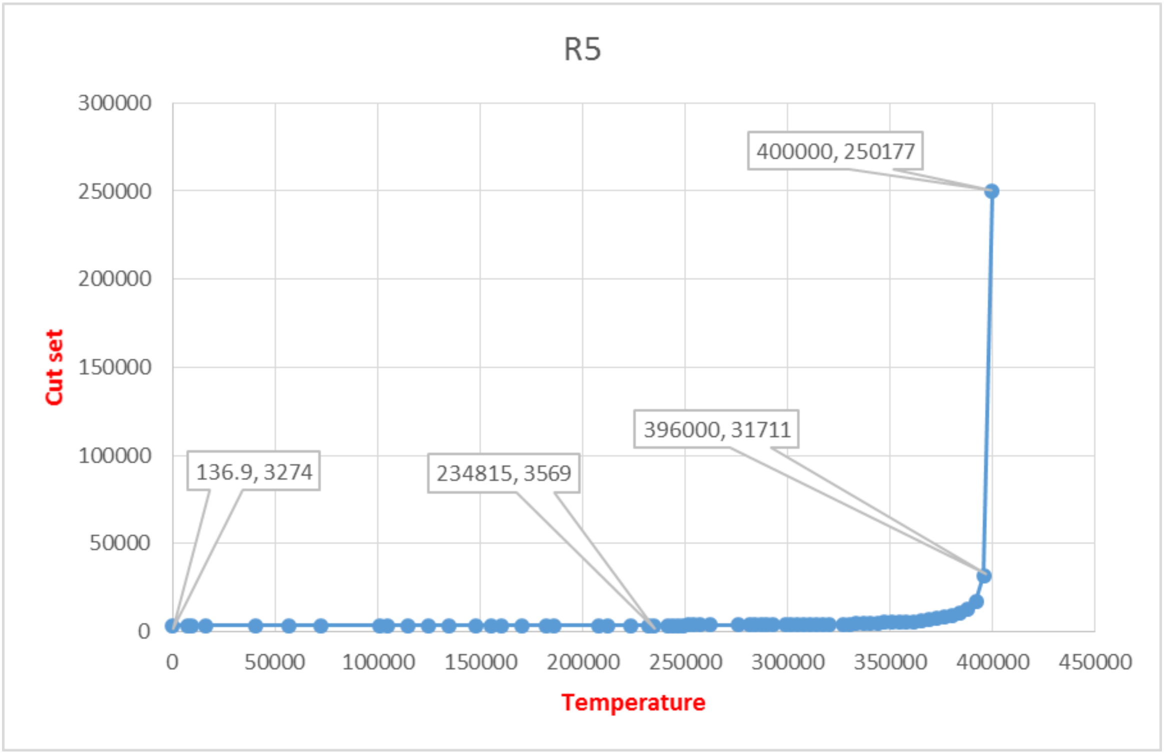
Temperature, cut set

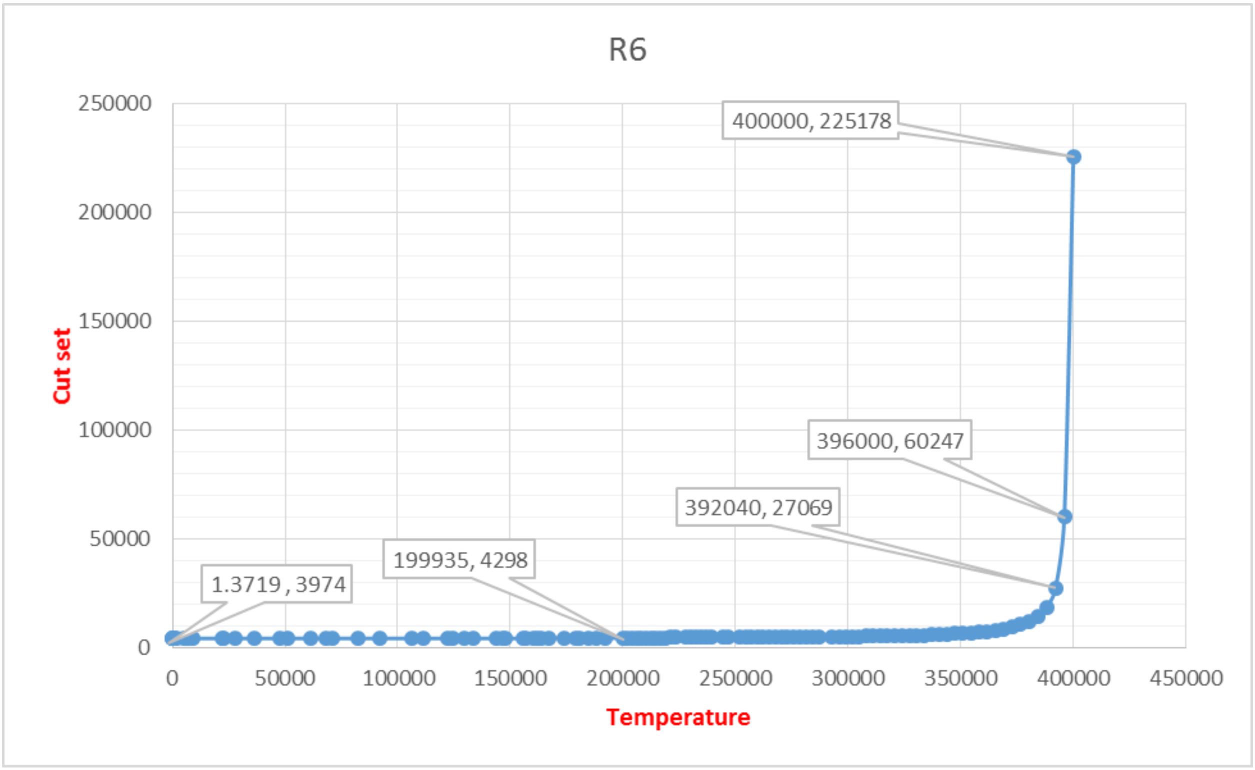





R3
CUT SET
TEMPERATURE
16000
14000
12000
10000
8000
6000
4000
2000
0
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
400000, 13519
380396, 126
388120, 137
384238, 130
372826, 118
369098, 113
358135, 112
392040, 155
396000, 240
197935, 110


R4
Cut set
Temperature
80000
70000
60000
50000
40000
30000
20000
10000
0
-10000
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
400000, 74943
396000, 4001
392040, 2498
365407, 1300
358135, 1234
354554, 1177
281379, 1095


R5
Cut set
Temperature
300000
250000
200000
150000
100000
50000
0
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
400000, 250177
396000, 31711
136.9, 3274
234815, 3569


R6
Cut set
Temperature
250000
200000
150000
100000
50000
0
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
400000, 225178
396000, 60247
392040, 27069
199935, 4298
1.3719 , 3974

## Tuning the program:

**Step 1:** Upper triangular connectivity matrix is stored in 2-D vectors. It is able to run for initial benchmarks till B3, it is unable to load higher benchmarks.

**Cost function time complexity:** O $((n/2)^2)$. n is the number of nodes
**Memory efficiency:** It is not memory efficient as non-weighted elements are stored.

**Step 2:** To overcome the problem of step1 connectivity matrix is implemented using compressed sparse row. To make accessing easier compressed sparse row is changed to co-ordinate list format which requires 3 1-D vectors to store row index, column index and weight of the edge.

**Cost function time complexity:** O $((n/2)^2)$
**Memory efficiency:** It is memory efficient compared to step 1 as it stores only weighted elements.
Able to load higher test benches but unable to meet time requirement to run higher benches.

**Step 3:** To overcome the time complexity problem of the step 2 heuristics from the KL algorithm is used. That is instead of calculating external cost for every node, internal and external connections are calculated for the randomly chosen nodes. To access the elements connected to the randomly chosen elements binary search is used. To use binary search sorting is required. To even make memory efficient 2-D vector of the type structure that stores column index and weight of the edge. Nodes are implicitly expressed as row index and nodes connected to each node and corresponding edge weights are placed as elements in each row.

**Gain=R1external+R2exteranal-R1internal-R2internal-2*Cab**

Data structures used: Two dimensional vectors of the type strut that stores corresponding column and weight. Nodes are implicitly expressed as index of each row.

**Time complexity:** This reduced the complexity to n* log (n) from quadratic complexity. Even now unable to meet the required execution time.

**Memory efficiency:** This is the memory efficient of all the data structures I have used as it is storing only weighted elements of the upper triangular connectivity matrix and node numbers are implicitly expressed as the index of each row.

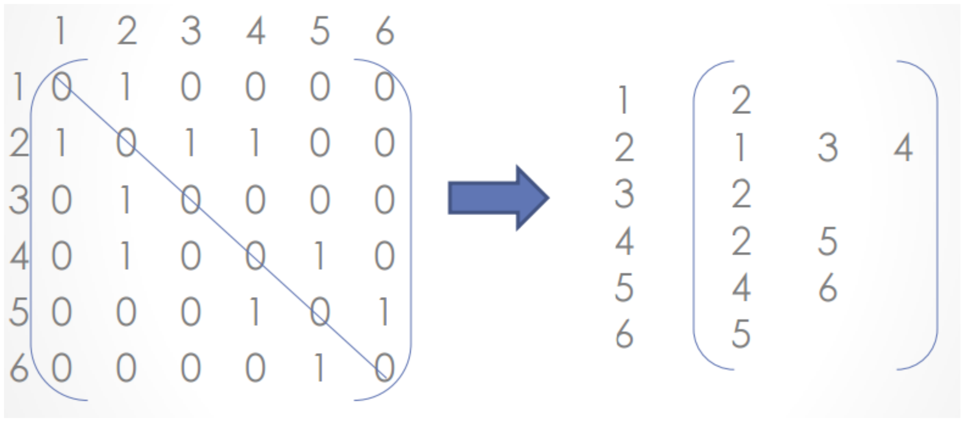
**Step 4:** Finally, to further reduce the time complexity I have used adjacency matrix which stores only connected nodes. Weights are implicitly expressed as the number of times each node appearing in the corresponding row.

**Cost function time complexity:** with this model I have seen a drastic change in the time complexity of the cost function. In the worst case time complexity is of order O (n). Able to run all the test benches in the required time.

**Memory efficiency:** It is memory efficient compared to step 1 and step 2 but as it stores the same connection twice to make accessibility easier it is not efficient compared to step3.

**Data structures used:** 2-D vectors are used to form the adjacency matrix. node number are expressed as the index of the row and corresponding connected nodes are placed as elements in the row.

vector<vector<int> > rowindex;



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 1 \\ 6 & 0 & 0 & 0 & 0 & 1 & 0 \end{array} \Rightarrow \begin{array}{c|ccc} 1 & 2 & & \\ 2 & 1 & 3 & 4 \\ 3 & 2 & & \\ 4 & 2 & 5 & \\ 5 & 4 & 6 & \\ 6 & 5 & & \end{array}$$