Constructing Covering Arrays using Simulated Annealing

Introduction

This report presents observations made upon applying the Simulated Annealing algorithm to construct Covering Arrays.
Covering arrays are mathematical, combinatorial entities. A covering array can be defined as CA(N;t,k,v) of strength t and order v is an N×k array over Z v with the property that every N×t subarray covers all members of Z tv at least once. [1]
Simulated Annealing is a probabilistic search algorithm that aims to find the global minimum or global maximum, hence making it a search algorithm for optimization problems. It is mainly used in problems in which there is a large search space.
This project is limited to a covering array with values t = 2, v = 2 and k =[5, 7]. Following report illustrates the experimental results obtained for these values.
On the whole, it can be observed that for the minimum possible value of N i.e., v^t, the algorithm almost never finds a solution (for the given values). But as we increase the value of N, in some iterations, the algorithm starts to find a solution. In addition, the frozen factor being achieved is almost always the stop criterion of the algorithm.

Background

Relevant background on the practical applications of covering arrays is that they are used in many fields including the field of software testing. When there are multiple components present in a software or hardware system, the tests for interactions among these components can be designed using covering arrays.
More background information on the algorithm used includes the fact that simulated annealing starts with a high temperature and gradually reduces its temperature as the algorithm proceeds forward. More importantly, it goes through roughly three modes of operation. In its initial phases i.e., when the temperature is high, it acts in the ‘Global Exploration’ mode. Hence, it is in this phase that it moves to seemingly bad states as well. As the temperature decreases, it takes on the ‘Improvement Focused’ mode where it tries to find solutions that are better than previous solutions. As temperature further cools down, it will act as a local search algorithm, narrowing down its search to find the best solution.

Proposed Approach

In this project, we try to find a solution to the problem of generating a covering array by applying the simulated annealing algorithm.
Steps of the algorithm:

- Initially, a random matrix of order Nxk is generated. Then, the randomly generated matrix is passed to the simulated annealing algorithm.

- In each iteration of the algorithm, the neighbors of the current state are calculated (using the neighborhood function), starting from the randomly generated matrix as the current state.
- Once the neighbors / possible next states are calculated, the best neighbor among those neighbors is picked by the algorithm.
- This best neighbor, which is the possible next state, is compared with the current state to decide whether or not to update the current state.
- The current state is updated if the number of missing combinations is lesser in the next best neighbor. Otherwise, the next best state is selected based on the Delta E function.
- In each iteration, current temperature is reduced by a factor of 0.99 and the final temperature is set to 0.1. (i.e., $T_{current} = 0.99 * T_{previous}$). In addition, in each iteration, if the solution found is not better than the best so far solution, then the number of temperature decrements made are kept track of until a better solution is found, so that if, even after a certain period of time, a better solution is not found, the algorithm can freeze.
- Following is the stop criteria for the algorithm: Either the solution is found (i.e., number of missing combinations equal zero), frozen factor is reached or the final temperature is achieved.

Neighborhood Function:
The neighborhood function chooses a column at random and creates a new next state by flipping each cell in that column, hence creating N neighbors.

Tiebreaker Criterion for Choosing the Next Best State:
If there are two neighbors for the current state with the same number of missing combinations, then the algorithm picks the first state it encounters as the next best state.

Proposed formula for N:
For any covering array of Nxk order, it is rarely possible to find a solution in the minimum possible N value, where N is the number of rows and k is the number of variables (or columns).
For this implementation, as we're beginning with a randomly generated initial state, we can use the concept of expectation to calculate the value of N for which the solution could be achieved.
Assuming that the expected number of missing combinations is strictly less than 1,
$Expectation(number\ of\ missing\ combination)\ <\ 1$
It can be seen that as the number of rows are increased (i.e., the value of N), the expectation value goes down.
In addition, assume that the length of domain range is equal for k variables, then as
$Exp(no.\ of\ missing\ combinations\ =\ no.\ of\ combinations\ x\ probability\ (missing\ combination)$
(where exp stands for expectation)

Then,
${}^{k}C_{t}$ x v^{t}x $(1 - (1 / v^{t}))^{N} < 1$
Solve the equation for the value of N to obtain,
log k <= N <= v^{t}log k

Proposed formula for Δ E:
Δ E is the cost difference between the next state and the current state. Here, the cost difference calculates the difference in the number of missing combinations in the current state and in the next state.

$$\Delta E = cost(current\ state) - cost(next\ best\ state)$$

Experimental Results

Column names of the table explained:

- Runs indicate the number of times the algorithm was run to construct a covering array.
- Stop criteria indicates the reason why the algorithm came to a halt.
- This can have three values:
  - Solution achieved
  - Frozen factor reached
  - Final temperature achieved
- Total iterations indicate the number of times the algorithm updated the current state within each run of it one time.

Observations for k = 5:

Runs	Stop Criteria	Total Iterations
1	Frozen Factor	49
2	Frozen Factor	53
3	Frozen Factor	51
4	Frozen Factor	44
5	Frozen Factor	91
6	Frozen Factor	54
7	Frozen Factor	47
8	Frozen Factor	53
9	Frozen Factor	60
10	Frozen Factor	80
11	Frozen Factor	63
12	Frozen Factor	45

13	Frozen Factor	49
14	Frozen Factor	47
15	Frozen Factor	43
16	Frozen Factor	46
17	Frozen Factor	45
18	Frozen Factor	61
19	Frozen Factor	69
20	Frozen Factor	54
21	Frozen Factor	63
22	Frozen Factor	98
23	Frozen Factor	60
24	Frozen Factor	65
25	Frozen Factor	67
26	Frozen Factor	56
27	Frozen Factor	48
28	Frozen Factor	43
29	Frozen Factor	102
30	Frozen Factor	58

Observations for k = 6:

Runs	Stop Criteria	Total Iterations
1	Frozen Factor	77
2	Frozen Factor	64
3	Frozen Factor	90
4	Frozen Factor	90
5	Frozen Factor	71
6	Frozen Factor	121
7	Frozen Factor	91
8	Frozen Factor	85
9	Frozen Factor	68
10	Frozen Factor	130
11	Frozen Factor	93
12	Frozen Factor	75
13	Frozen Factor	69
14	Frozen Factor	61
15	Frozen Factor	121
16	Frozen Factor	73
17	Frozen Factor	73
18	Frozen Factor	78
19	Frozen Factor	137
20	Frozen Factor	100
21	Frozen Factor	75
22	Frozen Factor	139
23	Frozen Factor	117

24	Frozen Factor	74
25	Frozen Factor	94
26	Frozen Factor	62
27	Frozen Factor	76
28	Frozen Factor	83
29	Frozen Factor	84
30	Frozen Factor	121

Observations for k = 7:

Runs	Stop Criteria	Total Iterations
1	Frozen Factor	89
2	Frozen Factor	191
3	Frozen Factor	173
4	Frozen Factor	158
5	Frozen Factor	160
6	Frozen Factor	99
7	Frozen Factor	108
8	Frozen Factor	155
9	Frozen Factor	102
10	Frozen Factor	102
11	Frozen Factor	170
12	Frozen Factor	90
13	Frozen Factor	130
14	Frozen Factor	125
15	Frozen Factor	121
16	Frozen Factor	103
17	Frozen Factor	116
18	Frozen Factor	118
19	Frozen Factor	155
20	Frozen Factor	88
21	Frozen Factor	100
22	Frozen Factor	128
23	Frozen Factor	113

24	Frozen Factor	139
25	Frozen Factor	157
26	Frozen Factor	162
27	Frozen Factor	87
28	Frozen Factor	138
29	Frozen Factor	209
30	Frozen Factor	102

Conclusions

- ☐ It can be concluded that, with N = 4, the algorithm doesn't reach the solution at all and it always stops due to the frozen factor being reached, meaning, the algorithm, at some point, starts to fail to update its best-so-far solution and freezes.
- ☐ In addition, it can be noted that when N = 6, for as few iterations as [8, 12], for k = 5, the algorithm reaches the solution.