DRILLING OF AN ELECTRICAL BOARD - A TRAVELLING SALESMAN PROBLEM

Ava Finnegan
2072113

Methods and Models for Combinatorial Optimisation
Dipartimento di Matematica

January 25, 2023

CONTENTS

1 INTRODUCTION

This report will focus on the well-known optimisation problem of the Travelling Salesman. The problem asks the following question: "Given a list of nodes and the times between each pair of nodes, what is the shortest possible route that the drill can make so that it visits each node exactly once and returns to the origin node (node 0)?".

This will first be solved using the CPLEX API. Two heuristic methods that are both local neighbourhood searches will then be implemented. The difference between the heuristic methods is the initialisation - a random initial solution versus a greedy initial solution. The methods will be compared directly using computation time. The computation times for a number of different sizes will be investigated. It is known that the TSP (Travelling Salesman Problem) is an NP-hard problem. The solution costs of the methods will then be examined.

2 THEORY

2.1 ELECTRIC PANELS

The company's aim is to minimise the total drilling time. This is beneficial for the company as they will manufacture a substantial number of electrical panels in any given day. Thus, the optimisation of each panel will reduce the company production costs.

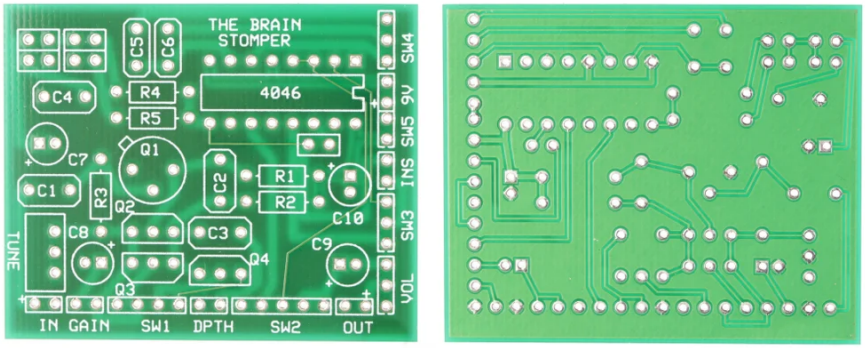


Figure 2.1: Example of a Electrical Panel board containing holes

As these electric panels are used to create various electrical circuits, the positioning of the holes are not random. There will be common attributes of these boards such as the presence of a row of holes which can be seen in 2.1

2.2 LOCAL SEARCH

A nearest neighbour local search was used as the heuristic method where the final solution will be a local minimum. Two edges are selected, removed and replaced by two other edges. The cost is calculated and then the exchange with the lowest cost is selected and executed.

2.3 INITIAL SOLUTION GENERATION

The initial solution plays a vital role of a nearest neighbour local search as it determines what local minimum the search will converge to. Using a random initial solution and restarting the search many times allows the neighbourhood to become more diverse and overcome the search becoming stuck in the same local minima.

In a greedy heuristic, a tour is constructed by growing a path in the graph and joining the end points once the path includes all the nodes in the graph. The first point can be randomly chosen or preselected (as in this case) and the node with the smallest cost is added to the path. This continues until the Hamiltonian cycle is formed.

3 METHODOLOGY

3.1 COST MATRIX

The cost matrix was an NxN matrix where the entry C_{ij} represented the time taken for the drill to move between node i and node j. It was assumed that the cost of drilling a hole from position 1 to position 2 is the same as the cost from drilling a hole from position 2 to position 1. This assumption was made as there was no apparent reason why the drill would take longer moving 'left' versus moving 'right' or moving 'up' versus moving 'down'. Thus, the cost matrices generated are symmetric.

The costs represent the time taken for the drill to move from hole_i to hole_j. Thus the time taken to move between hole_i and hole_i is taken to be 0 as the drill does not need to move. This is represented in the cost matrix as 0 values along the diagonal.

The values for the cost were randomly selected between 0 and 50. Although this is not representative of a true electrical board pattern as seen in 2.1, I believe that the random pattern will be sufficient to compare the methods.

3.2 METHOD TIME

To investigate the number of holes that the algorithm can solve in given number of seconds, a timer was used. The time was recorded at the start of the code and again at the end of the computations, this was done using both the wall clock time and the CPU time as seen in the Heuristics Lab.

3.3 REPRESENTATION OF TSP USING CPLEX

There are two decision variables in this TSP Problem. They are as follows

- x_{ij} = amount of the flow shipped from i to j, $\forall (i,j) \in A$;
- y_{ij} = 1 if arc (i,j) ships some flow, 0 otherwise, $\forall (i,j) \in A$;

3.3.1 COLUMN GENERATION

Two matrices was used to hold the variable indexes of the problem and each value was assigned a corresponding index. A matrix called 'X_matrix' held the x variable indexes. The y variable indexes were held in a matrix called 'Y_matrix'. An integer was initialised with the value 0. Thus each time a variable was added through a loop, the ij value would be added to the matrix which would equal the current index value. Once this was assigned, the index was increased by one. This was done firstly on the x variables and then was continued through the y variables.

$$min \sum_{i,j:(i,j)\in A} c_{ij} y_{ij} \tag{3.1}$$

The objective function (seen above) only contained y variables thus the x variables were entered as 0 when using 'CPXnewcols'. As the diagonal values of the matrix are 0, these were skipped in the loop and so no y_{ii} or x_{ii} variables were created.

```
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                        if(i==j || j==0) continue;
                        char xtype = 'C';
                        double lb = 0.0;
                        double ub = CPX_INFBOUND;
                        double obj = 0.0;
                        snprintf(name, NAME_SIZE, "x_%i_%i", nodes[i], nodes[j]);
                        char* xname = (&name[0]);
                        CHECKED_CPX_CALL(CPXnewcols, env, lp, 1, &obj, &lb, &ub, &xtype,
     &xname);
                        /// status =      CPXnewcols (env, lp, ccnt, obj, lb  , ub,
    xctype, colname);
                        X_matrix[i][j] = var_num;
                        var_num++;
                        //cout<<X_matrix[i][j]<<" ";
                }
```

```
16
17          }
```

3.3.2 Row Generation

The constraints were added using the CPXaddrows function. There were four constraints that needed to be added so CPXaddrows was used four times. Since all the variables/columns had already been added, the argument for the number of new columns to be added was 0 for all four constraints. One row was added each time CPXaddrows was called and a loop was used to control the number of rows added. This is illustrated in the code below which represents the following constraint:

$$\sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} = 1 \qquad \forall k \in N \setminus 0$$

This constraint was chosen as the example as this proved the most complex and encompassed the methodologies that the other three constraints used.

Three indexes were needed namely i,j and k. As the first node (0) was excluded, the k value began at the first node. The index was calculated to be '2N-3' which corresponds to the number of terms in each constraint equation. The code then looped over the indexes i and j to create the equation. Care was taken to ensure that diagonal values from the cost matrix were not included which can be seen from the condition that k cannot equal i or j at any time.

```
1       // add x constraints (forall k, sum{i} x_ik - sum{j} x_kj = 1) where j != 0
2         for ( int k = 1; k < N; k++) {
3                 std::vector<int> idx(2*N -3);
4                 std::vector<double> coef(2*N - 3);
5                 char sense = 'E';
6                 double rhs = 1.0;
7                 int index = 0;
8                 int nterms = (2*N - 3);
9                 for (int i = 0; i< N; i++) {
10                        if (i != k) {
11                                idx[index] = X_matrix[i][k];
12                                coef[index] = 1.0;
13                                index ++;
14                        }
15
16                }
17                for( int j = 1; j < N; j++) {
18                        if (j != k) {
19                                idx[index] = X_matrix[k][j];
20                                coef[index] =  -1.0;
21                                index ++;
22                         }
23                }
24               int matbeg = 0;
25               CHECKED_CPX_CALL ( CPXaddrows, env, lp, 0, 1, nterms, &rhs, &sense, &
    matbeg, &idx[0], &coef[0], NULL, NULL);
26
27        }
28
29        CHECKED_CPX_CALL( CPXwriteprob, env, lp, "tsp_exact.lp", NULL );
30        /// status =      CPXwriteprob (env, lp, "myprob"    , filetype_str);
31  }
```

3.4 Meta-heuristic Local Search Implementation

The meta-heuristic that was implemented in class was the nearest neighbourhood local search schema. The initial solution was generated randomly and then the schema iterated through the best cost of nearest neighbour until the minimum improvement was seen. This value was set to be -1e-6. and the objective function was computed at this time.

The method was edited so that instead of using an initial randomised solution, a greedy heuristic was used. The Hamiltonian cycle was created using the code below. The starting node was chosen as 0 as specified in the problem and the node with the cheapest cost was found and added to the sequence. This node was then removed from the potential candidates to preserve the cycle. This process was iterated until all the

nodes have been added in cheapest cost order. A vector named 'checker' was used to enable the tracking of used nodes by populating chosen nodes with 1. The vector was initialised with 0 except for the 0 node as this was the starting node. Thus when searching for the next node to add, the search would exclude all checker values of 1. The last node added was 0 to complete the cycle.

```
bool initRnd ( TSPSolution& sol , const TSP& tsp) {

      //initialise
      int size = sol.sequence.size() - 1;
      std::vector<int> checker(size);
      int idx_tmp;

      // populate indexes
      for (int i = 0; i < size; i++){
            if (i == 0){checker[i] = 1;}
            else {checker[i] = 0;}
      }

      // starts from row 0
      int i = 0;
      for (int k = 1; k < size; k++){
            int min_cost = 10000;
            // want to find the index with the smallest cost in row i
            for (int j = 0; j < size; j++){
                  if (tsp.cost[i][j] > min_cost || i == j || checker[j] == 1 ){
  continue; }
                  min_cost = tsp.cost[i][j];
                  idx_tmp = j;
            }

            checker[idx_tmp] = 1;

            // new loop to look in the i^th row
            i = idx_tmp;
            // overwrite the k^th
            sol.sequence[k] = idx_tmp;
      }
      std::cout << "### "; sol.print(); std::cout << " ###" << std::endl;
      return true;
}
```

As a greedy tour construction was used, if this heuristic was a bad heuristic then the search would become stuck in a bad local optimum. In the simulated annealing search algorithm, stochastic rules are added to try and escape from local optima. A random perturbation/kick was added into the search with the hope of significantly changing the start point and thus diversifying the neighbourhood. The goal was to escape from local minima to find a better solution. This perturbation was compared against the current local neighbourhood search and was either accepted or rejected. The code for this is shown below.

```

double TSPSolver::findBestNeighborDecrement ( const TSP& tsp , const TSPSolution&
    currSol , TSPMove& move )
{
  double bestDecrement = tsp.infinite;
  // idx:            0  1  2             n-2  n-1  n
  // city:           0  6  3              9    7   0
  // move 2 -- n-2:     h  i              j    1
  //city: 0 3 5 6 ...  9   8  0
  double new_neighCostVariation;
  int new_i_subs_init;
  int rand_num = (rand() % (currSol.sequence.size()-2) + 1) - 1;

  for ( uint i_subs_init = 1 ; i_subs_init < currSol.sequence.size() - 2 ; i_subs_init++
     ) {
    int h = currSol.sequence[i_subs_init-1];
    int i = currSol.sequence[i_subs_init];

    // create a random pertubation or 'kick'
    int h_rand = currSol.sequence[rand_num - 1];
    int i_rand = currSol.sequence[rand_num];

```

```
    for ( uint i_subs_end = i_subs_init + 1 ; i_subs_end < currSol.sequence.size() - 1 ;
     ++i_subs_end ) {
      int j = currSol.sequence[i_subs_end];
      int l = currSol.sequence[i_subs_end+1];
      double neighCostVariation = - tsp.cost[h][i] - tsp.cost[j][l]
                                  + tsp.cost[h][j] + tsp.cost[i][l] ;

      double perturbneighCostVariation = - tsp.cost[h_rand][i_rand] - tsp.cost[j][l]
                                  + tsp.cost[h_rand][j]  + tsp.cost[i_rand][l] ;

     if ((perturbneighCostVariation < neighCostVariation) & (i_rand != j)) {

     //if perturbation is better than nearest neighbour then update solution
        new_neighCostVariation = perturbneighCostVariation;
        new_i_subs_init = i_rand;
        }
      else {
        new_i_subs_init = i_subs_init;
        new_neighCostVariation = neighCostVariation;
      }

      if ( new_neighCostVariation < bestDecrement ) {
        bestDecrement = new_neighCostVariation;
        move.substring_init = new_i_subs_init;
        move.substring_end  = i_subs_end;

      }
    }
  }
 return bestDecrement;
}
```

4 RESULTS

4.1 CPLEX

The CPLEX method was tested on 3 different instances of each of the following sizes; 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140. Although both wall clock time and CPU time were recorded, the wall clock time was used for the results. The corresponding CPU time for each of the instances is included at the end of the report.

One of the solutions of the Travelling Salesman Problem for an N=10 instances is as follows. Given the cost matrix

Cost Matrix where N=10									
0.00	8.60	37.72	30.92	4.66	44.26	34.44	31.71	29.19	1.30
8.60	0.00	16.85	7.69	23.91	24.33	11.84	45.55	33.85	41.17
37.72	16.85	0.00	48.43	29.04	4.74	10.82	25.58	20.47	27.47
30.92	7.69	48.43	0.00	9.75	18.06	1.89	27.34	30.05	46.58
4.66	23.91	29.04	9.75	0.00	18.99	12.67	15.28	32.02	35.56
44.26	24.33	4.74	18.06	18.99	0.00	38.90	6.46	28.88	24.35
34.44	711.84	10.82	1.89	12.67	38.90	0.00	6.00	23.63	20.61
31.71	45.55	25.58	27.34	15.28	6.46	6.00	0.00	2.41	20.41
29.19	33.85	20.47	30.05	32.02	28.88	23.63	2.41	0.00	28.24
1.30	41.17	27.47	46.58	35.56	24.35	20.61	20.41	28.24	0.00

The shortest possible route that the drill can make so that it visits each node exactly once and returns to the origin node (node 0) is:

$$0 \rightarrow 9 \rightarrow 8 \rightarrow 7 \rightarrow 5 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 4 \rightarrow 0$$

The solution is a Hamiltonian Cycle which uses starting node equal to 0 (defined in the problem).

The computation time in seconds was recorded and can be viewed in the table below. It is important to note the variance in computation time in problems of the same size. Many more instances would need to be run in order to dilute outliers and obtain improved results. As 5 instances is a small number, variance is

higher and outliers are weighted heavily. This could explain the dip in the time for N=130. However, the graph does show an exponential increase in computation time as the size of the problem increases. The sharp increase in the computation time occurs around N = 100.

Computation Time (seconds)						
Number of Nodes(N)	Instance1	Instance2	Instance3	Instance4	Instance5	Average
10	0.1	0.19	0.1	0.14	0.12	0.13
20	0.52	0.32	0.18	0.27	0.29	0.32
30	0.64	0.72	0.83	0.93	0.85	0.79
40	1.5	2.22	1.89	2.17	1.45	1.85
50	3.43	2.41	4.02	3.7	1.39	2.99
60	4.29	9.78	5.13	5.26	7.75	6.44
70	7.5	7.56	16.6	15.98	12.03	11.93
80	23.84	17.4	20.75	13.02	10.24	17.05
90	18.78	21.35	20	76.1	33.66	33.98
100	32.57	88.75	35.39	66.28	58.83	56.36
110	32.16	91.97	143.2	82.94	106.83	91.42
120	222.03	187.58	85.38	139.96	281.95	183.38
130	63.13	121.51	249.43	278.69	134.85	169.52
140	536.66	490.76	454.99	445.45	565.34	498.64

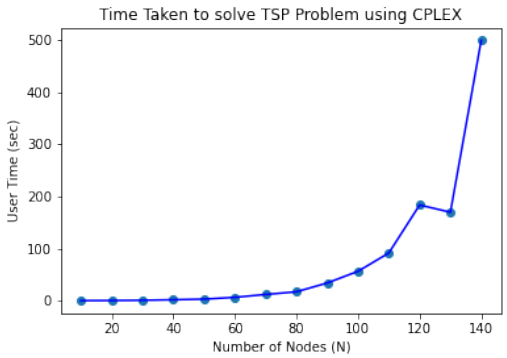


Figure 4.1: User Time against Size of Problem for CPLEX solver

To determine the model's ability to find exact solutions in a set time, a cutoff time is used. The cutoff times chosen (in seconds) were 0.5s, 1s, 10s, 30s, 60s, 300s and 600s. The number of nodes was cutoff for N=140. The model was capable of running for sizes larger than this but due to the exponential nature in computation, a cutoff point was made.

Max Size of Problem Solved in T Seconds	
Time (s)	Number of Nodes (Mean)
0.5	20
1	30
10	60
30	80
60	100
300 (5 min)	130
600 (10 min)	140

4.2 Heuristic Methodologies

The first heuristic method that was investigated was local search using nearest neighbour with an initialised solution that was a random Hamiltonian cycle starting and ending at 0. This heuristic method was completed

together as a class.

The example below shows the path taken for a cost matrix where N=10. The random solution had an initial cost of 234.88s and within 8 moves, this was reduced to 99.1s. This can be compared to the optimal solution cost which was 86.91s. The method works by reversing sub-sequences if they produce a better solution. This is illustrated below. There is a cutoff point of -1e-6 such that the algorithm stops if the solution does not improve by at least this value.

$$0 \rightarrow 9 \rightarrow 6 \rightarrow \underline{8 \rightarrow 4 \rightarrow 1} \rightarrow 7 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 0$$
$$0 \rightarrow 9 \rightarrow 6 \rightarrow 1 \rightarrow \underline{4 \rightarrow 8 \rightarrow 7 \rightarrow 3 \rightarrow 5 \rightarrow 2} \rightarrow 0$$
$$0 \rightarrow 9 \rightarrow 6 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow \underline{7 \rightarrow 8} \rightarrow 4 \rightarrow 0$$
$$0 \rightarrow 9 \rightarrow 6 \rightarrow 1 \rightarrow \underline{2 \rightarrow 5 \rightarrow 3} \rightarrow 8 \rightarrow 7 \rightarrow 4 \rightarrow 0$$
$$0 \rightarrow 9 \rightarrow 6 \rightarrow \underline{1 \rightarrow 3 \rightarrow 5} \rightarrow 2 \rightarrow 8 \rightarrow 7 \rightarrow 4 \rightarrow 0$$
$$0 \rightarrow 9 \rightarrow \underline{6 \rightarrow 3 \rightarrow 1 \rightarrow 5 \rightarrow 2 \rightarrow 8 \rightarrow 7} \rightarrow 4 \rightarrow 0$$
$$0 \rightarrow 9 \rightarrow 7 \rightarrow 8 \rightarrow 2 \rightarrow 5 \rightarrow \underline{1 \rightarrow 3 \rightarrow 6 \rightarrow 4} \rightarrow 0$$
$$0 \rightarrow 9 \rightarrow 7 \rightarrow 8 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 6 \rightarrow 3 \rightarrow 1 \rightarrow 0$$

As the initial solution is random, the algorithm should in theory be run many times so the variance can be established. Due to time constraints, each instance of size N was run 3 times and there were 5 different groups. Thus, each problem of size N was run 15 times to determine the variance of each size, in practice this should be much more. A high variance should reduce the chances of the solution becoming stuck in the same local minimum. This local minimum could be at a great distance from the global minimum. A high variance would increase the algorithms chances of finding the global minimum increasing with the number of restarts. As the local search is time efficient, running the local search algorithm multiple times is feasible.

An example of the results for a run of the same instance three times is shown in the table below. This is to demonstrate the range of the difference of the solutions for both methodologies. From the raw data, it can be seen the solutions derived from the greedy initial solution does not deviate much. The solutions that were obtained using a randomised initial search have a much larger range, this makes logical sense as it is randomly exploring the neighbourhood while the random kick that has been implemented only explores other parts of the neighbourhood if the algorithm deems it better.

Solution Costs over 3 Runs of the Same Instance			
	1	2	3
Random Initial Solution	159.61	184.89	155.96
Greedy Initial Solution and Random Kick	125.57	125.57	124.3

The local neighbourhood search was modified as discussed in the Methodology section above. As the initial solution for all runs of the same instance had the same value, a random kick was introduced into the solution to increase the diversity by artificially forcing the algorithm to consider a new area of the state space. The variance was not 0 except for the N = 10 state which is understandable as the solution found was or was very close to the optimal solution in all cases. The variance for all other sized problem was low in comparison to the variance of the random initial solution. This can also be seen from the following table where the same sized instances were run 15 times.

Variance of the Heuristic Solvers		
Number of Nodes	Random Initial	Greedy Initial and Kick
10	26.76	0
20	207.19	99.25
30	85.62	19.61
40	204.35	5.13
50	69.05	20.91
60	110.59	2.34
70	128.49	0.04
80	23.96	74.13
90	46.23	17.11
100	254.79	11.25
110	67.77	10.27
120	124.94	2.46
130	169.21	2.67
140	147.52	0.13

4.2.1 Solution Comparison

The optimal solution was known from the exact method and so it was used to test the 'accuracy' of the heuristic methods. The value taken for the solution for the heuristic method was the best value (lowest) over the three runs rather than the mean value. There would be no reason to average as the reason multiple runs were introduced was to increase the chances of finding a better solution.

The graph below compares the solution costs of the three methods. The below line represents the exact optimal solution that was found using CPLEX. This serves as a lower bound for the heuristic solutions to be compared to. It can be seen that the greedy initial search with a random perturbation outperformed the random search in nearly all cases. The optimal solution was found by the heuristic methods but this was only seen with small instance numbers (N=10 and N=20).

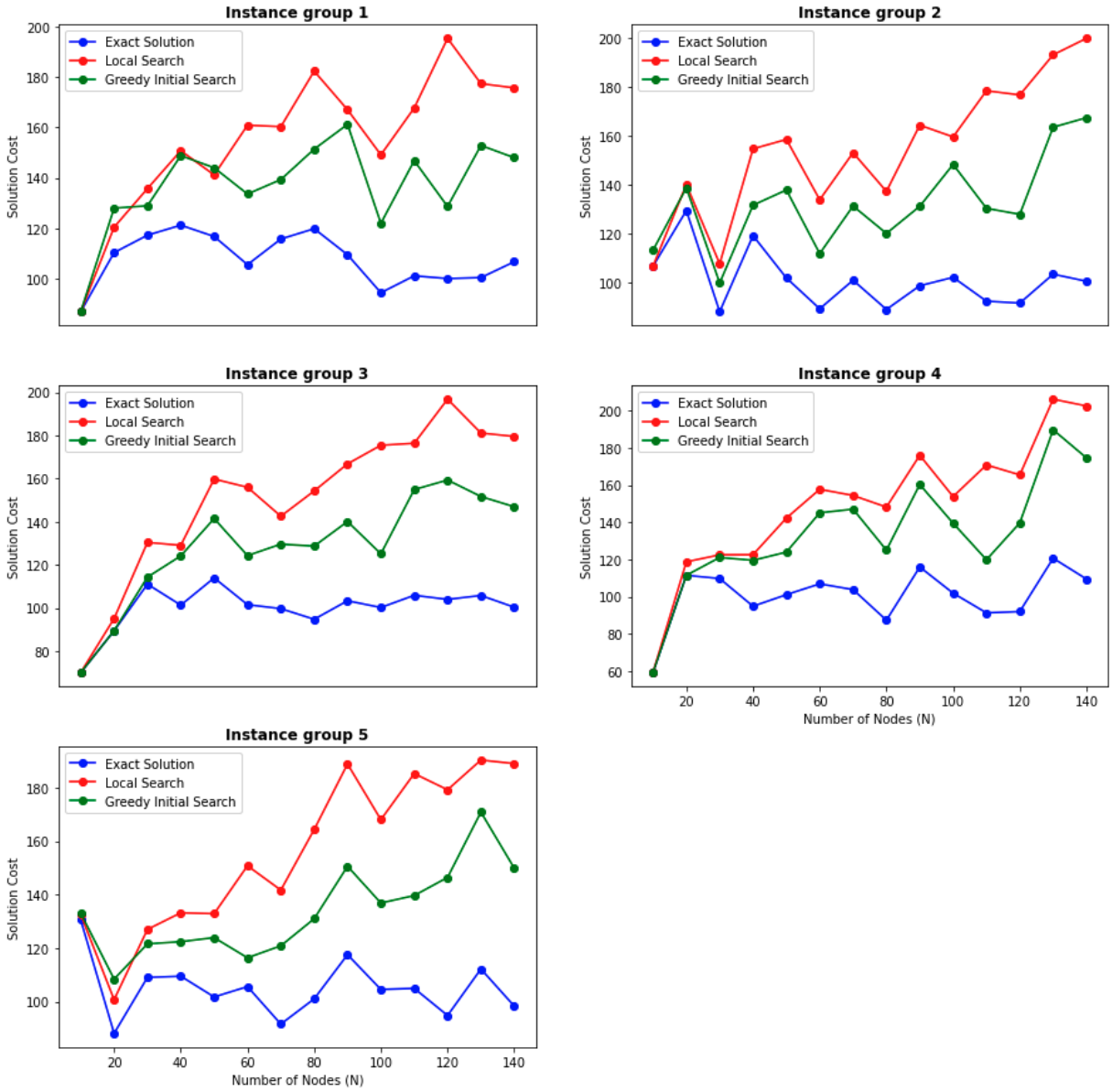


Figure 4.2: Solutions of the Solvers

As the methods for the initial solution generation differs, the costs of the initial solutions were compared. These differ greatly. The greedy method of generating an initial solution worked well for these instances where the nodes were randomly generated on a graph with values from 0-50. In contrast, the random initial

solution preformed poorly. The cost increased with the number of nodes. It can be seen from the optimal solution that the cost did not have a considerable dependence on the size of the solution. This further highlights the poor starting point for the random initialisation. The solution made larger steps in the direction of the optimal solution than the heuristic method.

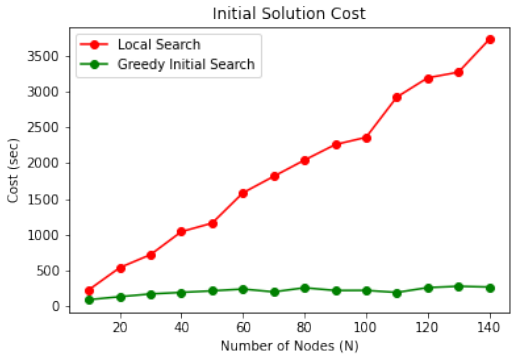
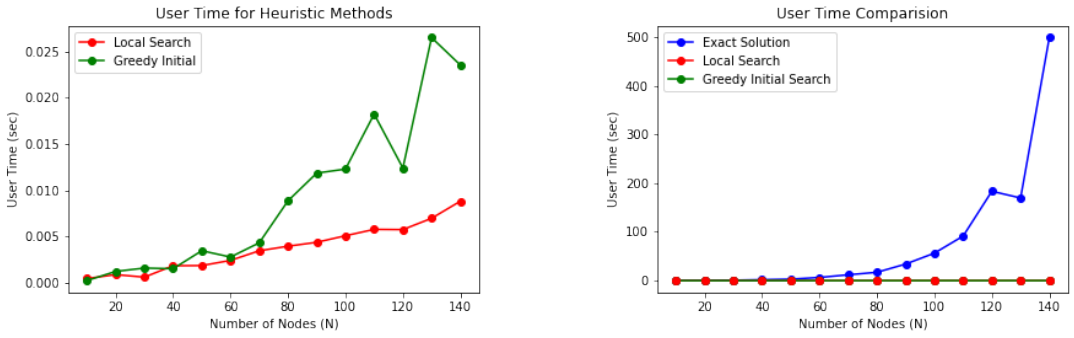


Figure 4.3: User Time against Size of Problem for Heuristic Solvers

4.3 Computation Time

The computation time for both methods can be seen the graphs below. The greedy initial approach was slower than the random initial approach. However, both methods were computed almost instantaneously and so the difference in computation time is negligible. The computation time for the exact method was then compared. This shows how efficient the heuristic methods in contrast to the exact solution in the tsp problem.



(a) (b)

Figure 4.4: User Time against Size of Problem

5 Conclusion

It can be seen that the CPLEX algorithm is growing exponentially with time - a characteristic of a NP-hard problem. The computation time of the heuristic methods are almost instantaneous and start to dominate when the number of nodes is greater than 90. The solutions for the heuristic solutions have a greater from the optimal solution as the size of the problem increases. Further analysis would need to be done for larger instances to explore this further.

Depending on the companies needs, the CPLEX solver or the heuristic solver could be used to provide a solution. The CPLEX method is clearly much more computationally costly and depends exponentially on the size of the problem. However, if the company kept the same pattern for an electrical board for a 'large' period of time and the number of nodes were reasonable, solving exactly the tsp problem would be the best approach as it would provide the maximum cost minimization possible given the constraints provided.

However the company may produce boards that can be customised and so, there may be many different patterns that are kept for a 'short' period of time. In this case, they may require a more efficient method of computing the cost and are happy to compromise on the optimality and so a heuristic method could be implemented. (Note: It is possible for an optimal solution to be found using heuristics but it is not a guarantee) Another reason to use heuristics is that the size of the problem may cause computing the exact solution infeasible.