CHAPTER 4

Discriminatory Randomized Greedy Heuristic

4.1 Introduction

Several greedy and semi greedy heuristics exist in literature such as OTTC [49], CBTC [107] and RTC [156]. One issue plaguing the greedy heuristics is that greediness often misleads the heuristic to construct shorter backbones to which the majority of vertices are connected using heavier edges. This leads to the formation of a BDMST having significantly higher weight than the optimum. The semi greedy heuristics such as RTC are able to overcome this issue to some extent, but, there is a scope for improving the solution quality if some topological information is embedded in the heuristic while choosing the centre vertex and the vertices which should form the backbone. This shall lead to longer backbone edges and lighter stars leading to generation of BDMSTs weights approximating the optimum. Further, the random choice of centre leads to significantly higher variance in the solution. Consequently, the solution has to be normalized by running the heuristic \( n \) times, each time with a new vertex as centre which leads to enormously high computational complexity.

The proposed Discriminatory Randomized Greedy Heuristic \([\text{DRGH}]\) has to overcome these issues by choosing the centre from an informed subset of the vertices. The topological information of the instance graph guides the heuristic in choosing the centre, thus, reducing the sensitivity of the final solution to the initial choice of the centre. A measure of ‘goodness’ is calculated for the vertices to act as the centre or to be in the backbone, leading to generation of BDMSTs which have longer backbone edges and lighter stars.
Discriminatory Randomized Greedy Heuristic [DRGH] is a heuristic construction strategy based upon a modification of Prim’s algorithm [150] named RTC [155]. As opposed to strategies like IR1, IR2 and CIR [2], DRGH constructs an approximate BDMST in one pass i.e. negating the need of scanning the vertex set repeatedly in order to add vertices. A Prim based algorithm [150] has been chosen as the base for enhancement since it has been experimentally shown to be the fastest algorithm for computing an MST for large dense-graphs [130]. In addition, Prim’s algorithm [150] keeps the partially-formed tree connected in every iteration during the MST-construction process. This makes it easier to keep track of the diameter, as opposed to keeping track of the diameters of the trees of a forest in Kruskal’s algorithm [111]. Furthermore, using the Kruskal’s algorithm [111], the greedy strategy will not be able to construct a spanning tree of diameter k in one pass if an intermediate step creates two non-spanning trees of diameter $k/2$. This makes it necessary to keep more information about the forest while the spanning tree is being constructed, and some cases may require backtracking.

For ease of implementation and faster processing of the underlying data structures, we have proposed to modify the direct edge-set encoding of tree and suggest the indexed-edge-set encoding. In our study, instead of encoding the BDMSTs as direct edge-sets, where an edge between a vertex $u$ and $v$ is represented by $(u, v)$, we assign a unique index to each edge. The edge between $u$ and $v$ may be indexed as 1, between $u$ and $w$ as 2 and so on. As the underlying graph is assumed to be undirected, the edge between $v$ and $u$ will also be indexed as 1. This is done in order to maintain the same edge identity for the edges $(u, v)$ and $(v, u)$ in undirected graphs.

Further, the choice of centre in such problems influences the shape of the tree and thus impacts the performance of the underlying heuristic [191], [192]. In literature, the addition of topological information has been leveraged to improve performance especially when the nodes form a non uniform cluster in which the occurrence of outliers cannot be ruled out [187]. DRGH assumes significance in light of the issues which arise in the
construction of BDMSTs using greedy & semi-greedy strategies like OTTC [2], CBTC [107] and RGH [155].

4.2.1 Significance of DRGH

Issue 1: Formation of BDMSTs having shorter backbone edges and heavy auxiliary edges.

When the heuristics such as above, are applied to problem instances whose vertices are points in Euclidean space and whose edge weights are the distances between the points, the heuristics, in general yield spanning trees whose weights are much larger than minimum. This is particularly true when the diameter bound is tight i.e., $D$ is small compared to $n$. Figure 4.1 (a) shows a spanning tree, identified by OTTC, on Euclidean instance of size $n = 250$ and diameter $D = 5$. Short edges connect only a few vertices near the centre of this tree. The remaining vertices connect via longer edges to this core, forming a star like structure. Let the backbone of tree be the subgraph induced by the tree’s non-leaf vertices. In a low weight BDMST, the backbone is longer, and the leaves get connected to it via shorter edges. A good heuristic for the BDMST problem will not prefer lower weight edges when it builds the backbone. OTTC and CBTC (refer figure 4.1 (a) and (b)) always use the lowest weight edge available, and therefore, are misled. This observation holds for almost all BDMST instances in which triangle inequality is approximately satisfied. However, due to contained greediness, RTC is able to identify slightly shorter tree (refer figure 4.1(c)) but sufficient scope for improving the solution quality exists.

![Figure 4.1](image-url) : lowest-weight BDMST found by a) OTTC, b) CBTC, and c) RTC on Euclidean instance with $n = 250$ vertices and diameter bound $D = 30$ having weights 62.38, 40.44 and 18.08 respectively [155].
**Solution strategy:** Aggregating the data points (nodes) in respect of their distance to rest of the nodes results in lighter stars. DRGH identifies the vertices that can connect to other vertices in lesser number of hops on the least weighted path thereby minimizing the probability of forming heavy stars.

**Issue 2:** Heavy dependence of the final solution on the initial choice of the centre.

Heuristics like OTTC, CBTC and RGH show a heavy dependence of the final solution quality on the initial choice of the centre. Choosing the centre randomly, as is done by these heuristics, results in higher variance in final solutions, which reflects the sensitivity of these heuristics towards the randomly chosen centre.

**Solution strategy:** DRGH is able to check the dependence of the final solution on the initial choice of the centre by choosing the centre not from the vertex set $V$ but from a small subset of the set $V$. Topological information of the instance graph has been utilized in the form of potency matrix to create an informed subset 'Potential_Centres' from the universal set $V$. The subset 'Potential_Centres' as explained in section 4.4.2 contains the vertices which are good at forming centre and the backbone; thereby resulting in lower weight trees. Choosing the centre from 'Potential_Centres' instead from $V$ also leads to significantly smaller variance in the final solutions.

**Issue 3: High Computational Complexity.**

Heuristics such as OTTC, RTC and CBTC are computationally expensive. These heuristics are sensitive to the initial choice of the centre and therefore have to be run, starting from each node as the centre. This makes the time complexity of OTTC as $O(n^4)$, and that of CBTC and RTC as $O(n^3)$, a factor of $n$ less than that of OTTC. Test demonstrates that though the randomized greedy heuristic yields substantially better solutions on Euclidean problem instances than does OTTC, but the running time of RTC is still prohibitive.

**Solution Strategy:** DRGH is less sensitive to the initial choice of the centre and therefore does not have to be run starting from each vertex. This significantly brings down the computational complexity of DRGH.
4.3 Algorithm:  DRGH

Input: Weight matrix $G[v1,v2]$ of the instance graph, Diameter bound $D$ and tolerance $\alpha$; // default value of $\alpha = 999$;

Step 1: Apply all pair shortest path algorithm to the weight matrix of the underlying graph $G(v1,v2)$ using Modified Dijkstra [65] and store the results in matrix Vertex_Potency.

Step 2: Compute the sum of hops from a vertex $i$ to all the $n$ vertices

$$S[i] = \sum_{j=1}^{n} \text{Vertex}_\text{Potency}[i,j] \text{ for } \forall i \in \{1,2 \ldots n\}$$

Step 3: Compute the sum of $S[i]$'s $\forall i \in 1 \text{ to } n$. i.e Sum = $\sum_{i=1}^{n} S[i]$  

Step 4: Sort array $S[i]$ using QuickSort() and simultaneously store the index of the vertex $i$ (the position of vertex in the array Vertex_Potency) in an array Sorted_vertex at the same position at which $S[i]$ falls in the sorted list.

Step 5: Compute the range $l$ depending upon whether $\alpha$ has been given a value. If yes then $l$ is equal to number of vertices whose value is less then $S[1] + \alpha$ // where $\alpha$ is the level of tolerance. (refer section 4.9) Else Compute the average of $S[i] : Av = \text{Sum}/n$

Step 6: Compute 'l' equal to number of vertices whose $S[i]$ value is less than equal to Av  

\ l' refers to the maximum number of vertices in the set Potential_Centres

Step 7: Generate a number $Q$ randomly in the range 2 to $l$.

Step 8: Assign first $Q$ vertices from the set Sorted_vertex to the array Potential_Centres

Step 9: Number of pendant_vertices is then computed using the formula

$$\text{pendant_vertices} = \left(\frac{n - 1}{2 \times D}\right) \text{ where } D \text{ is the diameter constraint}$$

Step 10: Calculate the number of intermediate vertices as

$$ln = (n - (\text{pendant_vertices} - 1))$$

Step 11: Assign vertices of set $V$ to the vertex subset Intermediate $V$ as

$$j = 0$$

for $i = Q + 1; i < ln; i++$

$$\text{Intermediate}_V[j] = \text{Sorted}_\text{vertex}[i]$$

$$j = j + 1$$
4.4.1 Effect of the choice of Centre on the performance of the heuristic

Selection of the centre of graph (tree) is an important topic both of practical and theoretical interest in multicast routing. The choice of the centre in such applications influences the shape of the tree and thus influences the performance of the underlying heuristic [191], [192]. The addition of topological information can be leveraged to improve performance especially when nodes form a non uniform cluster in which the occurrence of outliers cannot be ruled out [187]. An issue in this regard is, how complex the core or centre choice method should be, to ensure a reasonable performance. When there is little variation in the performance of different centres, an arbitrary choice is sufficient. When the variance is substantial, more sophisticated methods to choose the core are called for [29].

4.4.2 Need for Centre_Determine( ) subroutine

Many new multimedia applications such as videoconferences or multiplayer games on the internet ask for the selection of a meeting point in the network, where each user sends his personal data to the meeting point. The role of this particular point is to gather the information received to create a single composite data flow, which is multicast back to the users. Empirical analysis has been carried out to measure the relationship between the choice of the location of centre and the performance of the routing scheme (refer [29], [63], [113]). The performance of such a scheme is usually evaluated in terms of delay and

Step 12: Assign the left over vertices of array Sorted_vertex to array Pendant_V

Step 13: Call Subroutine Reshuffle(). This facilitates random selection of vertices from the three subsets Potential_Centres, Intermediate_V and Pendant_V.

Step 14: Call subroutine Tree_Generation()
//This routine creates the Spanning tree by selecting vertices in order from the array Potential_Centres, then the array Intermediate_V and finally from Pendant_V and accordingly updating the depth of each vertex from the randomly chosen center. A vertex is chosen to be in the tree if it does not violate the diameter constraint.
bandwidth consumption. If the objective is to minimize the average delay between every pair of nodes in the group, the optimal path is achieved on a shortest path tree rooted at the centre. In that case, each node-centre shortest path is used in both directions, and the optimal location of the centre is such as to minimize the sum of the shortest path lengths between the centre and every node. The problem of locating the centre node \( c \) under these two criteria, one being the sum of the edges between \( v \) and rest of the nodes and the other being the cost of the tree linking \( c \) to the rest of the nodes has been solved in [113] using an enumeration algorithm that evaluates the objective function at every potential core location, but the computational complexity of such an enumerative approach is extremely high. Centre_Determine( ) strategy given in this work, makes the choice of the centre under the above two criteria but manages to keep the time complexity reasonably low.

**Algorithm:** Centre_Determine( )

**Input1:** Weight matrix \( G(v_1, v_2) \) of the instance graph, Diameter bound \( D \) and tolerance \( \alpha \); // default value of \( \alpha = 999 \);

**Step 1:** Apply all pair shortest path algorithm to the weight matrix of the underlying graph \( G(v_1, v_2) \) using Modified Dijkstra [65] and store the results in matrix \( \text{Vertex Potency} \).

**Step 2:** for \( i = 1; i \leq n; i = i + 1 \)

\[
\begin{align*}
    & \text{sum_hops} = 0 \\
    & \text{for} (j = 1; j \leq n; j = j + 1) \\
    & \quad \text{sum_hops} = \text{sum_hops} + \text{Vertex Potency}[i][j] \\
    & S[i] = \text{sum_hops}
\end{align*}
\]

**Step 3:** Sort array \( S[i] \) using QuickSort( ) and simultaneously store the index of the vertex \( i \) (the position of vertex in the array \( \text{Vertex Potency} \)) in an array \( \text{Sorted vertex} \) at the same position at which \( S[i] \) falls in the sorted list.

**Step 4:** if \( (\alpha < 999) \)

\[
\begin{align*}
    & \text{limit} = S[1] + \alpha
\end{align*}
\]
\[ i = 1 \]
\[
while (S[i] \leq \text{ limit})
{
    i = i + 1
}
\]
\[
} // end of if
\]
\[
else
{
    \text{Sum} = 0
    for (j = 1; j \leq n; j = i + 1)
    \text{Sum} = \text{Sum} + S[j]
    \text{Av} = \frac{\text{Sum}}{n}
\]
\[
i = 0
\]
\[
while (S[i] \leq \text{ Av})
{
    i = i + 1
}
\]
\[
} // end of else
\]
\[
\text{// refers to the maximum number of vertices in Potential_Centres}
\]
\[
} // end of else
\]

**Step 5:** \( Q = \text{random}(l - 2) + 2 \)

**Step 6:** \( \text{for}(i = 1; i \leq Q; i = i + 1) \)
\[
\begin{align*}
\text{Potential_Centres}[i] &= \text{Sorted_vertex}[i] \\
\text{C\_length} &= Q
\end{align*}
\]
\[
\text{//where C\_length is the length of the vertex set Potential_Centres}
\]
\[
} // end of else
\]

**Step 7:** \( ln = \text{ pendant\_vertices } = (n - 1)/(2 * D) \)

**Step 8:** \( j = i \)
\[
\text{for}(i = Q + 1; i \leq ln; i = i + 1)
{
    \text{Intermediate\_V}[j] = \text{Sorted\_vertex}[i] \\
    j = j + 1
}
\]
\[
ln\_length = j
\]
As the starting vertex, significantly affects the weight of the generated trees therefore in rather than choosing randomized centre (one centre if D is even and two otherwise), randomly from the vertex set ‘V’, we propose to choose the centre from a specified subset of 'V'. The vertex set V of the instance graph is partitioned into three subsets Potential_Centres, Intermediate_V and Pendant_V (refer figure 4.2), based on their potential to form the centre, the backbone of the BDMST and the auxiliary edges. The Centre is chosen from an informed subset 'Potential_Centres' of 'V'. In addition, while extending the tree by adding vertices, the next vertex is not chosen randomly from 'V', but from these three subsets in order of merit.

In the subroutine Center_Determine() , the entries [i,j] of the matrix Vertex_Potency are initialized for the input graph G by creating an adjacency matrix of G. In [29], it is argued that the random choice of centre is better than arbitrary choice and that topologically informed choice of the centre leads to even better performance. In this case, the location of the centre is to be determined based on two criteria, i) the sum of distance between the centre and rest of the nodes is minimum and ii) minimizing the number of hops on this least weight path between centre and all other nodes. This topological information about the instance graph is computed and stored in the Vertex_Potency matrix. The entries Vertex_Potency[i][j] of this matrix represent the number of hops taken to reach from vertex i to each vertex j, on the least weighted path. Sum of the

```
//where l_length is the length of the vertex set Potential_Centres

Step 9:  j = 1
          for( ; i ≤ n; i = i + 1)
          {
                Pendant_V[j] = Sorted_vertex[i]
                j = j + 1
          }

          P_length = j
          //where P_length is the length of the vertex set Potential_Centres
```
number of hops it takes to reach from a vertex $i$ to every other vertex $j$ gives an estimate of the measure of ‘goodness’ of the vertex $i$ to be the centre. Let $S[i]$ denote the vector storing this sum for all the vertices of the graph. Further, $S[i]$ is sorted along with its vertex indices in ascending order, to reflect the relative potential of each vertex to act as the centre of the tree, or to be in the backbone of the tree.

If the objective is to minimize the average delay between every pair of users in the group, the optimal routing is achieved on a shortest path tree rooted at the centre. In that case, each shortest path from the centre to every other node is used in both directions, and the optimal location of the centre is such as to minimize the sum of the shortest path lengths between the centre node and every other node.

Let $\text{dist}(u)$ denote the depth of the shortest path rooted at $u$. (The depth of a tree rooted at $u$ is the maximum length from $u$ to any leaf in the tree). A centre of a network is any node, $c$, that has minimum value for $\text{dist}(c)$. Our method selects a core, $u$, at random from node satisfying the condition:

$$\text{dist}(u) \leq \text{dist}(c) + \alpha$$

In effect, $\alpha$ specifies a tolerance (in excess of the minimum value achieved by a node) on the depth of the tree. By varying the value of $\alpha$ we get a family of methods between the extreme of choosing a particular node as centre and choosing the centre at random. The argument for such a choice of centre is that, the nodes requiring lesser number of hops on
an average, on the shortest path should have better performance than nodes chosen at random.

In case where the value of $\alpha$ is 0 then the vertex with minimum $S[i]$ value is chosen as the centre (ties resolved randomly). In case where the value of $\alpha$ is specified as say, $x\%$, the vertices whose $S[i]$ values are within $x\%$ of the minimum $S[i]$ value can contend to be the centre. The value of tolerance has to be empirically tuned to suit the instance type and size.

For the benchmark instances taken from Beasley’s OR-Library for Euclidean Steiner tree problems, $DRGH$ has been found to give optimal results for different values of $\alpha$ ranging from $\alpha = 30\%$ to $70\%$. When $DRGH$ is deployed as an initialization procedure for any metaheuristic such as genetic algorithm amount of diversity in the initial population pool has a significant bearing on the convergence rate and run time of the metaheuristic. Under these situations a flexible tolerance level can help to control the premature convergence and maintain diversity in the genetic algorithm.

In case user cannot ascertain in advance the value of $\alpha$, the default procedure is run. Default procedure for ascertaining the set of Potential_Centres vertices is as follows. The average of the $S[i]$ values is computed and the number of vertices having $S[i]$ value less than the average are stored in the variable $'\ell'$ . The first $\ell$ vertices of the sorted vector $S[i]$ contend to populate the set Potential_Centres. The reason for determining the set Potential_Centres in this manner is that it can be safely assumed that the vertices which take less than average number of hops to reach all other nodes will form better centre and a backbone to which the rest of the vertices can be connected using lighter edges and lesser number of hops. A random number $Q$ is then determined in the range:

$$Q = \text{random} (\ell - 2) + 2$$

The first $Q$ vertices out of $\ell$ populate the subset Potential_Centres.

Procedure to calculate the sets Intermediate_V and Pendant_V remains the same irrespective of the status of the $\alpha$. We know that the number of pendant vertices in a BDMST are: \(\text{pendant vertices} = \frac{(n - 1)}{(n - 2)}\), where $n$ is the number of nodes, $D$ is the diameter constraint and Pendant vertices are the vertices which are farthest from the
centre in terms of number of number of hops on the least weighted path. The number of Intermediate vertices can therefore be calculated as the vertices following the first $Q$ vertices till $n - (\text{pendant\_vertices}) - 1$. These vertices populate the array $\text{Intermediate\_V}$. The rest of the vertices constitute the subset $\text{Pendant\_V}$.

4.5 Vertex Randomization

We have conjuncted the property of Binary Search Tree (BST) with arrays to implement the random vertex choosing procedure. The significance of deploying the underlying feature of BST is that we have been able to achieve amortized time of $O(n)$ to carry out the randomization procedure of selection of unique nodes.

4.5.1 Using BST for Vertex Randomization

Selecting a vertex randomly form one of the subsets $\text{Potential\_Centres}$, $\text{Intermediate\_V}$ or $\text{Pendant\_V}$, warrants the need of a data structure, that would require less time in accessing an element randomly and in subsequent updating of the data structure. Thus, the total processing time required by any underlying data structure for such an operation can be calculated as $\text{Processing time} = \text{Access time} + \text{Update time}$. Let us consider array as the underlying data structure. The selection of a vertex randomly from amongst the elements of an array would have a low access time but updating the array after deletion of an element is a time intensive operation. Updating an array would require time approximately of the order $n^2$ for shifting the elements. If the underlying data structure is linked list, linear access of elements would again require $O(n^2)$ time. This is because before linearly accessing linked list, it would require creation of it, and for creating a linked list we again would require $O(n^2)$ time for assuring uniqueness of nodes in the linked list. After that deletion would take $n$ time. Thus, time complexity using both array and linked list is hence prohibitive. We propose to use the BST property in conjugation with arrays to implement $\text{Reshuffle(\_\_)}$ routine. They support arbitrary accessing a node in $O(1)$ amortized time. The BST property when conjuncted with array negates the need to make
deletions and hence reduces the time complexity considerably. Using $Reshuffle(\ )$ routine we are able to improve the running time of not only DRGH abut also of RTC, originally proposed in [155].

To implement $Reshuffle(\ )$, we use the concept of BST trees. A BST tree is a rooted tree with the property that values in the left subtree are always smaller from the value of root node and all the values in the right subtree are greater than those of the root node and all the values are unique. A BST is a collection of item disjoint ordered trees, and choosing a random value $x$ divides the tree into two disjoint sets. If the range is $1$ to $r$, then choosing a random value $x$ divides the range into two disjoint subsets $(1, x - 1)$ and $(x + 1, r)$. By recursively following the same procedure we generate a randomly BST which preserves the uniqueness of nodes.

**Algorithm: Reshuffle**

**Reshuffle** *(arr, length)*

```c
int t = 1, l = 1;
r[1][t] = 1;
r[2][t] = length;
do
{ val = r[2][t] - r[1][t];
  if (val == 1)
  { r[3][t] = r[1][t];
    l = l + 1;
    r[1][l] = r[2][t];
    r[2][l] = r[2][r];
  }
if (val == 0)
{ r[3][t] = r[1][t];
}
```

4.6 Subroutine Reshuffle

Subroutine Reshuffle( ) generates a randomized list of vertices Randomized_vertex_list, from the given subset of vertices (Potential_Centres, Intermediate_V, Pendant_V). If we have 14 vertices then in the 1st instance 5 is chosen as the random vertex. So, the value 1 gets stored in r1 and 14 gets stored in r2 and the randomly chosen vertex gets stored in r3. Choosing a random vertex 5 divides the set into 2 subsets (1,4) and (6,14), which are then stored in the array r1 and r2 simultaneously at position 2 and 3. In the second instance, depending upon the range(1,4) again a random vertex say 3 is chosen which further divides the subset into two disjoint sets (1,2) and (4,4). The process of dividing the search space into disjoint sets and storing their subsequent ranges continues. During traversal, if the range has difference of 1, for example (5, 6), then r3 for that instance is assigned the
smaller value that is 5 and only (6,6) gets stored in the array r2. And if the range has a
difference of 0, then the space is not further divided. In all other cases the space gets
divided into two disjoint sets and the corresponding r3 value is chosen randomly from
within that range.

![Binary Search Tree representation of Reshuffle routine for vertex randomization](image)

a) Binary Search Tree representation of Reshuffle routine for vertex randomization

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>8</td>
<td>11</td>
<td>9</td>
<td>11</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>r2</td>
<td>14</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>8</td>
<td>13</td>
<td>9</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>r3</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>8</td>
<td>13</td>
<td>9</td>
<td>11</td>
<td>14</td>
<td>12</td>
</tr>
</tbody>
</table>

b) Populating random vertices in array r3 (r1 and r2 holding the upper and lower bounds for the random vertex selected in r3) using Reshuffle routine.

Figure 4.3 (a) to (b): Vertex randomization using Reshuffle routine

### 4.7 Tree Generation

In each iteration of DRGH, the algorithm randomly picks a vertex 'u' from the vertices not yet added to the tree. It then determines its nearest vertex 'v' from the set of vertices already in the spanning tree-'Span_Tree', whose depth is less than $\frac{D}{2}$. Edge $(u, v)$ is then added to the tree after verifying the diameter constraint. If the entry corresponding to the vertex $v$ in Vertex depth array is less than $\frac{D}{2}$, $u$ is added to the growing spanning tree. The
depth of the new vertex $u$ in the $\text{Vertex_depth}$ vector is one more than the depth of the vertex $v$, to which it is connected, thus if $\text{edge}(u, v)$ is added to the partially constructed tree then:

$$\text{Vertex_depth}[u] = \text{Vertex_depth}[v] + 1$$

This routine makes use of a vector $\text{Vertex_depth}$ to track the potential violation of the diameter constraint. This greatly simplifies the computation and complexity of $\text{DRGH}$. We are able to reduce the time complexity of $\text{DRGH}$ appreciably as compared to $\text{OTTC}$ as we make use of only one array $\text{Vertex_depth}$ instead of iteratively updating two arrays maintained by $\text{OTTC}$ the $\text{dist(.)}$ and $\text{ecc(.)}$ arrays.

**Algorithm: Tree Generation**

$$\text{Tree_generation(subset, Curr_loc, Curr_tree)}$$

```c
{
    weight = 99999
    v1 = subset[Curr_loc]
    // Curr_loc is a global variable keeping track of the current tree in array Tree_Pop
    for(i = 1; i <= tl; i = i + 1)
    // tl is a global variable keeping track of the current location in the spanning tree
    {
        v2 = Span_Tree[i]
        if((Graph[v1][v2] < weight) AND (Vertex_depth[v2] + 1 ≤ \frac{\text{Diameter}}{2}))
        {
            if(v1 < v2)
            index = n * (v1 - 1) + v2
            Else
            index = n * (v2 - 1) + v1
            d = Vertex_depth[v2] + 1
        }
        weight = Graph[v1][v2]
    } //End of for
```
The subroutine $Tree\_generation( )$ takes as parameters $\text{subset}$, $m$ and $\text{Curr\_tree}$, where $\text{subset}$ is one of the vertex subsets \{Potential\_Centres, Intermediate\_V, Pendant\_V\}, $m$ refers to the element in the vertex subset from where the partially constructed tree $Span\_Tree$ shall will be extended. In case there is one centre, the value of this parameter will be 2, in case of the odd diameter tree; the parameter $m$ will hold a value of 3. Parameter $\text{Curr\_tree}$ refers to the current tree under construction. The next vertex $v1$ is picked from the $m^{th}$ position of the Randomized_vertex_list of the vertex subset referred to by the parameter $\text{subset}$. $v1$ is checked against all the vertices already present in the spanning tree $Span\_Tree$. A vertex $v2$ from $Span\_Tree$ with which it forms the least weighted edge will be subjected to feasibility satisfaction. At this point both the constraints are checked. First it is checked to determine that the addition of this edge do not violate the diameter bound. The vector $\text{Vertex\_depth}$ is checked to confirm that the distance $\text{Vertex\_depth}[v2 + 1]$ from centre is less than $\frac{D}{2}$. And, the edge $(v1,v2)$ is the least weighted edge among all the edges considered. If the feasibility criteria are satisfied, the edge index $(v1,v2)$ generated on-the-fly is stored in the 2D-vector $Tree\_pop$ at $\text{Curr\_loc}$ for the $\text{Curr\_tree}$. $Span\_Tree$ is then updated as $Span\_Tree = Span\_Tree + v1$ and $\text{Curr\_loc} = \text{Curr\_loc} + 1.$

The task of considering the edge between a random $v1$ and all the vertices in the spanning tree is an operation that would become extremely time intensive when this operation has to be carried out before adding each vertex to the growing tree. A lookup table in the form of a $NXN$ matrix would require $\approx n^2$ lookups on an average, for addition of each edge. We instead have chosen to generate edge indices on the fly, given the pair of vertices.
4.7.1 Generating edges on the fly

When a pair of vertices, say, $v_1$ and $v_2$ is encountered, and it required to determine the edge between them more specifically the edge index of the edge connecting the two vertices, DRGH generates the edge index on the fly using the following formula:

\[
\begin{align*}
\text{if}(v_1 < v_2) & \quad \text{index}_1 = n \cdot (v_1 - 1) + v_2 \\
\text{else} & \quad \text{index}_1 = n \cdot (v_2 - 1) + v_1
\end{align*}
\]

where $n$ is the number of vertices in the instance graph. Further, it is checked whether $v_1 < v_2$ to ensure that same edge index is given to the edge between $(v_1, v_2)$ and $(v_2, v_1)$ as the instance graphs considered are undirected graphs. Distance of the newly added vertex from the centre is calculated and depth vector is updated.

![Figure 4.4: Generating Edge Indices on the fly](image)

4.8 Time complexity of DRGH

The time complexity of DRGH is the sum total of the time complexities of the Centre_Determine() subroutine, the procedure Reshuffle() and the routine Tree_generation().

a) The time complexity of Centre_Determine() routine is $O(n^2 \log(n))$. This is broadly dependent on the complexity of all-pair-shortest-path Dijkstra’s algorithm that has been incorporated.

b) The time complexity of the procedure Reshuffle() is $O(n)$ because of the BST property conjunctured with array for random retrieval of vertices.
c) The time complexity of \textit{Tree\_generation()} is $O(n^2)$. This is attributed to the basic property of tree generation procedure where in the first instance 1 comparison is made, for the second vertex two comparisons are required and so on, thus making $n$ comparisons for the $n - 1$ elements. Thus bringing overall complexity to $n^2$.

So total complexity of \textit{DRGH} is equal to
\[
\cong O(n^2(\log(n) + n + n^2))
\]
\[
\cong O(n^2(\log(n)))
\]

The time complexity of \textit{DRGH} is lesser than the time complexities of \textit{OTTC}, \textit{RTC} and \textit{CBTC}. This is significant because \textit{RTC} currently is considered the best construction heuristic for quickly generating initial solutions for metaheuristics such as \textit{GA}, \textit{ACO} and \textit{VNS}. \textit{DRGH} has not only been able to bring down the time complexity but also improve the solution quality as compared to the other popular heuristics from the literature.