CHAPTER 6
Guided Genetic Algorithm for Tree Construction [G-GAT]

6.1 Introduction

Many real world network design problems can be formulated as graphs where the possible solutions are constrained spanning trees. Such problems are more often than not, NP-hard [161]. Concentrating on a single metaheuristic like GA is not likely to yield results which are competitive with the state-of-the-art tailored metaheuristics. A carefully designed combination of GA with heuristics and/or local search procedures can provide a more efficient search behavior both in terms of quality of the solutions obtained and in terms of computational complexity. A typical genetic algorithm works on an initial population that has been generated randomly. But, drawing from the observation that a good initial population for GA accelerates the speed of its convergence to yield good solutions, and a poor initial population results in a poor search behavior of GA [74], the hybridized metaheuristic based strategy proposed in this work – Guided Genetic Algorithm for Tree construction [G-GAT] has been designed to work on good initial population which has been generated using a fast construction heuristic DRGH proposed in chapter 4.

Further, it has been observed that the strength of a GA lies in exploring the search space efficiently; but slow convergence behavior plagues its performance; whereas local search procedures have been found to be efficient at exploiting the identified regions of the search space but are poor at exploration, as a result the search might get trapped in local optima. G-GAT has therefore been designed as a comprehensive platform in which a semi greedy heuristic DRGH has been hybridized with the strength of GA and the search behavior of this combination is further enhanced by the exploitation capacity of two local search procedures Arc_Ex_Mut and level_Ex_Mut which have been proposed in chapter 5.
One of the major determinants of the success of a search procedure is the balance between intensification and diversification. It is therefore pertinent that, when further intensification is not yielding better solution, diversification is introduced. Based on this observation, a filter has been designed which imposes a cut off to suggest when the solutions should be subjected to the local search based mutation procedures. The idea of introducing two neighborhoods is that a solution which is local optima in one neighborhood structure may not be local optima in the other neighborhood structure and therefore effective switching between the different neighborhoods can help the search to escape local optima.

Finally in last part of the chapter we have proposed a new multiparent crossover operator $k$-GCO that is designed to preserve maximal parental substructures, as the parents are all above average solutions generated using $DRGH$. Strong heritability quotient of $k$-GCO along with a higher arity and generating a large number of offsprings allows for a full exploitation of the common substructures of the parents.

**6.2 Guided Genetic Algorithm for Tree construction: Methodology**

$G\text{-GAT}$, as any Standard Genetic Algorithm, can be visualized as generating a set of solutions $P(t) := \{S_1, S_2, \ldots, S_m\}$. For a generational GA with population size $\lambda$, $\lambda$ solutions are generated in the same cycle. For a steady state GA, solutions are generated one by one until $\lambda$ solutions are generated. The algorithm framework for $G\text{-GAT}$ is neither generational nor purely steady-state (refer figure 6.1). In $G\text{-GAT}$, for a population size $\text{POP\_SIZE} = \lambda$ in each iteration, at the most ‘$k$’ new candidate solutions may replace the ‘$k$’ worst chromosomes in the current population if the replacement criterion is satisfied, where $k$ is the number of parents and $k << \lambda$ but $k$ may be larger than 1 or 2 which is the case with steady state GAs. Premature convergence is one of the major issues involved in application of GAs to combinatorial optimization problems. It is tackled implicitly in the design structure of $G\text{-GAT}$ at more than one level. Both, the $\text{Diversity\_switch(\ )}$ as well as the $\text{Uniqueness(\ )}$ subroutine ensures a healthy supply of diverse genotypes to the algorithm. Keeping in view the facts that $i$) The success of a genetic algorithm, to a great extent, depends on the initial building blocks supplied to it and $ii$) The number of
generations needed by GA to reach the optimum is a function of the areas of the landscape sampled in the initial population pool, $G$-$GAT$ derives its initial population from the semi greedy heuristic $DRGH$ explained in chapter 4. Hybridization of the exploratory power of GA along with the structured yet stochastic exploitative capacity achieved by the two neighborhood-based local search procedures (explained in chapter 5) as mutation operators ensure that the most promising regions of the fitness landscape are searched, without searching those regions of the landscape, that may not hold any promise for locating optimal solutions.

![Diagram of Guided Genetic Algorithm for Tree Construction]

*Figure 6.1: Generic Flow of Guided Genetic Algorithm for Tree Construction*

The functionality of $G$-$GAT$ can be explained as consisting of different phases. These may be understood as different procedures that operate on a dynamic 2D vector representation of trees named *Dynamic Vector Representation for Tree construction* ($DVRT$). The algorithm Guided Genetic Algorithm for Tree Construction, the enhanced GA for BDMST construction has been proposed as under:-
Algorithm:  Guided Genetic Algorithm for Tree Construction (G-GAT)

Step 1: Set Counter = 0, generation = 0, Sum = 0, range = 10, pos = 1;

Step 2: Call subroutine DRGH() to generate initial population of BDMSTs such that \( P(\text{generation}) = \{T_1, T_2, \ldots, T_n\}\) where \( n \) represents the size of initial population POP_SIZE and \( T_i \in P(\text{generation}) \) if \( \text{Uniqueness}(T_i, P(\text{generation})) = \text{true} \)

Step 3: Evaluate \( F(\text{generation}) = \{f(T_1), f(T_2), \ldots, f(T_n)\}\) for all \( i, j \in \{1, 2, \ldots, N\} \)

Step 4: Find \( T^* \in P(\text{generation}) \) such that \( f(T^*) \leq f(T_i) \) for all \( T_i \in P(\text{generation}) \)

Step 5: Call subroutine k_GCO() to generate offsprings \( C \) with fitness \( f(C) \) such that \( \text{Off} = \{C_1, C_2, \ldots, C_{2k}\} \) and \( \text{fit}_{\text{Off}} = \{f(C_1), f(C_2), \ldots, f(C_{2k})\} \)

Step 6: Replacement_Check = Replacement_Strategy()

Replacement_strategy() checks whether 10% of 2k offsprings are able to replace the parents

Step 7: Calculate the average hamming distance of 2k offsprings and store the result in \( \text{offspring}_{\text{curr_gen}} \)

Step 8: \textbf{if} (generation < range)
\[
\{ \quad \text{dissimilarity_quotient}[\text{pos}] = \text{offspring}_{\text{curr_gen}} \\
\quad \text{sum} = \text{sum} + \text{offspring}_{\text{curr_gen}} \\
\}
\]
\textbf{else if} (generation = range)
\[
\text{offspring}_{\text{gen10}} = \frac{\text{sum}}{\text{range}}
\]
\textbf{else}
\[
\{ \quad \text{pos} = (\text{generation}/\text{range}) \\
\quad t = \text{dissimilarity_quotient}[\text{pos}] \\
\quad \text{dissimilarity_quotient}[\text{pos}] = \text{offspring}_{\text{curr_gen}} \\
\quad \text{offspring}_{\text{gen10}} = (\text{offspring}_{\text{gen10}} - t) + \text{dissimilarity_quotient}[\text{pos}] \\
\}
\]

Step 9: Hamming_Distance_Check(\( \text{offspring}_{\text{curr_gen}}, \text{offspring}_{\text{gen10}} \))

Step 10: \textbf{if} (generations \( \geq \) range)
\[
\text{then go to step 11} \quad \textbf{else go to step 13}
\]
6.3 Working of G-GAT

This section explains the working of G-GAT and its constituent phases.

6.3.1 Phase I: Initial Population Generation

Generating poor quality solutions using GA can be prevented by starting with an initial solution that has been obtained by running a good approximation algorithm for the given problem. Also, solutions obtained by approximation algorithms can be improved in a small number of iterations by evolutionary algorithms. Taking cue from this, the initial population for G-GAT is generated not randomly but by running DRGH subroutine, which is a fast construction heuristic that requires a probabilistic component to compute healthy individuals. The element of stochasticity in DRGH helps to generate slightly different BDMSTs in each run, ensuring that the trees created lie in the same basin of attraction or at least lie close to each other in the genotypic space. This ensures that the initial candidates
supplied to $G$-$GAT$, sample the promising regions of the search space which would not have been possible if the initial candidates were generated randomly.

**Stage 1:** In this phase the routine $DRGH$ is called to generate feasible initial candidates for $G$-$GAT$. The subroutine $Centre\_Determine(\ )$ partitions the vertex set into three subsets based on the potential of the vertices to be in the backbone of the solution tree. $Centre$ is randomly chosen from the subset which has the highest potential. The partially constructed tree is extended by adding the vertices chosen randomly from these subsets in decreasing order of their potential. These randomly chosen vertices connect to one of the vertices in the tree with the most economical edge. The process of randomization of vertices in these subsets is carried out by the routine $Reshuffle(\ )$ (explained in section 4.6) which helps to reduce the time complexity of this phase significantly.

The initial population pool may contain any number of trees. For our work the initial population pool was kept at 100 individuals as this ensured requisite amount of diversity for GA to be able to reach the optimum in small number of generations without affecting the solution quality. The number of generations required to reach the optimum and the population size are the two parameters of genetic algorithms which are inversely proportional to each other. But this is the case with traditional Genetic Algorithms, where initial population is generated randomly. By using $DRGH$ to generate the initial population for $G$-$GAT$, we have been able to obtain optimum solutions in lesser number of generations by keeping the population size as small as 100. It has been observed that population size beyond 100 does not significantly improve the solution quality of $G$-$GAT$.

**Stage 2:** To check that duplicate individuals do not enter the population pool and sufficient genotypic diversity is maintained in the population, the subroutine $Uniqueness(\ )$ is called so that the crossover operator has a greater scope to recombine the parental sub structures to create better trees. The uniqueness operator works on the principle of hamming distance where the standard binary hamming distance has been modified to characterize the distance between the two spanning trees as proposed in [188]. Uniqueness of the candidates is ensured by allowing only those candidates to be inserted into the population pool which fall above a certain threshold of $Minimum\text{-}Hamming\text{-}Distance$ i.e. $Dist(T_i,T_j)$ is
The routine DRGH is iteratively called until the criteria of unique trees for POP_SIZE is satisfied.

**Distance** \( d_{ij} \in \{0, 1, \ldots, n-2\} \) between two spanning trees \( T_i \) and \( T_j \) has been calculated as[168]:

\[
d_{ij} = \frac{1}{2} \sum_{u,v \in V} |l_{uv}^i - l_{uv}^j|
\]

Where \( l_{uv}^i \) is 1 if a link (edge) from \( u \) to \( v \) exists in \( T_i \) and 0 if it does not.

The number of edges that two trees \( T_i \) and \( T_j \) have in common can be calculated as:

\[
\text{Dist}(T_i, T_j) = n - 1 - d_{ij}.
\]

The parameter **Minimum-Hamming-Distance** is tunable and can be set to the desired number of differing edges /genes. In case of G-GAT, it has been observed that a **Minimum-Hamming-Distance** value of 3 yields optimum performance. A tree generated by DRGH has been allowed to enter in the population pool only if it differs from the already generated trees in the pool, in at least 3 edges. G-GAT has been designed as a generalized platform, and with minimum changes it can be adapted to solve a variety of optimization problems. The minimum number of genes in which the two candidates must differ can therefore be set by the user, depending upon the optimization problem at hand. This parameter \( \text{Dist}(T_i, T_j) \) has to be carefully tuned as this has a direct bearing on the diversity of the population and the regions of the search space being sampled and therefore greatly impacts the performance of G-GAT both in terms of the quality of the solution and the computational time.

### 6.3.2 Phase II: Parent Selection

In this phase the standard 2-tournament selection method has been modified to the k-2-tournament operator which is called to select \( k \) parents for recombining in the next phase. The selection pressure of highly fit individuals and the adaptive advantage [37] given to
them can be controlled by changing the tournament size. To mitigate the effect of losing the best building blocks generated, we use elitism to guarantee that the fittest individuals will be copied, unchanged, to the next generation, hence ensuring a continuously non-decreasing maximum fitness. Selection pressure gives individuals of higher quality a higher probability of being selected to create the next generation so that GAs can focus on promising regions in the search space [26]. The fitness difference between the mating pool and the population reflects the selection pressure, which is expected to improve the fitness of each succeeding generation [142]. The Algorithm for k-2-tournament operator is given as:

**Algorithm-k-2-Tournament**

**Step 1:** Initialize parent_count = 0  
**Step 2:** Rank all the parents in the Parent pool based on their fitness  
**Step 3:** Choose a preset value x, where, \(0 < x < 1\)  
**Step 4:** Randomly select two parents P1 and P2  
**Step 5:** Generate a random number RN, \(0 < RN < 1\)  
**Step 6:** if RN < x then select the higher ranked parent  
  else select the other parent  
**Step 7:** Increment parent_count by 1  
**Step 8:** if (parent_count < k) then go to step 2  
  else go to step 9  
**Step 9:** Stop

The experimental results reveal that this method gives good results and is also computationally efficient. Tests with values of \(x\) equal to 0.60, 0.75 and 0.90 give similar results. Thus this method has been used in the main experiment with \(x = 0.75\).
6.3.3 Phase III: Recombination

In this phase the multiparent crossover operator $k$-GCO is called to recombine the promising building blocks provided by DRGH. The solutions provided by DRGH shall all be lying in the basin of attraction of the same optima; hence more number of parents for $k$-GCO would mean a larger sample of building blocks to choose for recombining. Also, it has been proven that tournament selection alone cannot provide an adequate selection pressure [130], therefore, a $k$-GCO has been designed to provide a strong heritability quotient (refer chapter 7). $k$-GCO takes $k$ parents as input and tries to preserve maximum parental edges in the offsprings. Another unique feature of $k$-GCO is that it produces $2k$ offsprings from $k$ parents. As the offsprings produced by $k$-GCO differ by a very small percentage, producing $2k$ offsprings ensures that the identified region of the search space is fully exploited.

The crossover probability of $k$-GCO has been fixed at 1. This is in contrast to the literature where the different crossover probabilities of 0.5, 0.6, 0.7 [89] have been suggested. Probability of less than 1 had been suggested for crossing over because by definition, crossover operator is disruptive of the building blocks. Setting the crossover probability equal to 1 in standard uniform crossovers would completely disrupt the search process and hinder the structured evolution of the search space [200]. Despite this, crossover probability in $k$-GCO has been set to 1. This is because $k$-GCO has been designed to produce $2k$ offsprings which have a strong locality and are located close to the parents and to each other in the search space (refer chapter 7). Strong heritability in $k$-GCO is achieved by allowing feasible parental edges, in order of number of occurrences in $k$ parents to be replicated in all the $2k$ offsprings. This ensures that the crossover operator conducts an adequate amount of exploration of the search space yet does not disrupt the process of evolutionary search, thus justifying the need of setting the crossover probability equal to 1. The design details of $K$-GCO along with its functioning and significance in the overall G-GAT structure and justifying the need for a multi parent crossover has been discussed in chapter 3.
6.3.4 Phase IV: Diversity Switch

When the recent history of the search process indicates that intensification seems no longer effective, a diversification phase is needed. The solution reached after the local search phase may be accepted with 100% probability, or the probability to accept it is increased. The neighborhood changing strategy applied by Guided Local Search [27] consists of penalizing the features that occur frequently in visiting solutions and thus dynamically change the objective function. The aim therefore is, to induce a change in the landscape being searched, by changing the objective function. But, such a strategy may lead to loosing the good genes acquired over several iterations of the GA. Based on this observation, the idea of VNS (Variable neighborhood search) [101] has been modified and adapted in G-GAT, which is, to define more than one neighborhood structure, and to swap between different neighborhood structures in a strategic way during the search process. Observe that the process of shifting the neighborhoods (for example, in case of no improvements) corresponds to a diversification of the search. A solution that is locally optimal with respect to one neighborhood is probably not locally optimal with respect to another neighborhood.

For measuring the diversity, methods such as evolution history [139], distance [20] and measures in the phenotypic and genotypic space [148] have been studied in literature. We are able to demonstrate that diversity at moderate level along with adaptive neighborhood selection is able to strike a balance between global exploration and local exploitation. The adaptive algorithm that we have devised outperforms traditional fixed GA search strategy. This measure of population diversity takes into account both, phenotypic diversity as well as genotypic diversity, providing a single diversity method for use in both situations. The principle advantage of this measure of diversity is that, in comparison with traditional methods of computing pair-wise hamming distance, which is quadratic in population size, this measure is linear.

Further, it has been well established that selection pressure and population diversity are inversely related [214]. Increasing selection pressure results in a faster loss of population diversity, while maintaining population diversity, offsets the effect of increasing selection pressure. These two factors should be controlled in order to obtain their
advantages simultaneously and allowing the most promising search space regions to be reached and refined. Under these circumstances, we will say that the population has achieved useful diversity, i.e., population diversity that in some way helps to produce good solutions [132].

6.3.4.1 Design of Diversity_Switch

Diversity_switch() is a filter that is applied to impose a cutoff point for solutions to be submitted to local search. This filter is applied immediately after the explorative phase. It evaluates the diversity of the population and accordingly applies effective measures to reintroduce fresh building blocks in the population, if needed. It is a switch which decides whether the incumbent solution is a fit candidate for the more expensive exploitative phase. This switch is comprised of two checks: Replacement_Strategy() and the Hamming_Distance_Check(). Replacement_Strategy() is a routine which checks the 1st of the following two conditions and returns a boolean value. Hamming_Distance_Check() is a routine which compares the dissimilarity quotient of the offsprings of current generation with the offsprings of the previous ten generations.

\[
\text{if}(10\% \text{ of } 2k \text{ offsprings are better in fitness than the parents})
\]

OR

\[
\text{if}
\left( \left( \text{Dissimilarity_quotient}(\text{offspring}_{\text{gen, curr}}) \right) < \left( \text{Dissimilarity_quotient}(\text{offspring}_{\text{gen, 10}}) \right) \right)
\]

6.3.4.2 Dissimilarity Quotient()

Dissimilarity_quotient() is a measure of how dissimilar the offsprings are with respect to each other. It is computed by checking the number of occurrences of each of the non-parental edges in the $2k$ offsprings. It is to be noted that the offsprings differ in
\((n - p + 1)\) edges, where \(n\) represents the total number of edges in BDMST and \(p\) represents the similar number of edges in \(2k\) offsprings. The \textit{Dissimilarity\_quotient()} can be computed as given below:

\textbf{Algorithm: Dissimilarity\_Quotient()}

\begin{verbatim}
{ 
for\(a = 0; a < 2k; a = a + 1\)
{
for\(i = p + 1; i < n; i = i + 1\)
{
index = offspring[a][i]
pos = BinarySearch(*B,index)
B[pos][1] = B[pos][1] + 1
}
}
sum = 0;
for\(i = 0; i < n - 1; i = i + 1\)
{
if\(B[i][1] > 1\)
    sum = sum + B[i][1];
}
N = 2 * k * (n - p + 1);
/* where 2k represents offsprings, n represents the total number of edges in BDMST
And p represents the similar number of edges in 2k offsprings */
dissimilarity = \frac{N - sum}{(2 * k)}
}
\end{verbatim}

\textit{Diversity\_switch()} as explained above, has been so designed that it incorporates a measure of phenotypic as well as genotypic diversity. The first condition of the switch
checks the status of the phenotypic diversity of the population. It says if 10% of the offsprings have higher fitness than that of the parents, the phenotypic diversity check $p_{\text{diversity}}$ is set to 1. Again, if the quotient of dissimilarity $\text{Dissimilarity\_quotient}()$ in the offspring chromosomes of the current generation is less than the Average dissimilarity quotient of the previous ten generations, the genotypic diversity check, $g_{\text{diversity}}$ is set to 1. Four Boolean conditions may thus arise as shown in figure 6.2 below.

![Figure 6.2: Status of Diversity\_switch() as $p_{\text{diversity}}$ and $g_{\text{diversity}}$ assume different values](image)

As the phenotypic diversity parameter $p_{\text{diversity}}$ and the genotypic diversity parameter $g_{\text{diversity}}$ assume different values, the following four conditions and consequent neighborhood swapping strategies may arise:

i) $p_{\text{diversity}} = 1$ AND $g_{\text{diversity}} = 1$ : Zero mutation

ii) $p_{\text{diversity}} = 0$ AND $g_{\text{diversity}} = 1$ : Warning Stage

iii) $p_{\text{diversity}} = 1$ AND $g_{\text{diversity}} = 0$ : Warning Stage

iv) $p_{\text{diversity}} = 0$ AND $g_{\text{diversity}} = 0$ : Alarming Stage

When both $p_{\text{diversity}}$ and $g_{\text{diversity}} = 1$, it means that there is sufficient phenotypic as well as genotypic diversity in the population therefore the incumbent solution may not be submitted for further mutation. When second or third condition holds that is either phenotypic or genotypic diversity is sufficient (indicated by a value of 1) but the other one is absent (indicated by a value equal to 0), a need for introducing mutation is felt and the first NBLS method $\text{Arc\_Ex\_Mut}$ is called. This stage of diversity is known as the warning stage. $\text{Arc\_Ex\_Mut}$, as discussed in chapter 5, is a neighborhood search strategy which will exchange a randomly chosen edge at a level ‘l’ with the most eligible edge at the same level.
Fourth condition is met when both phenotypic diversity and genotypic diversity are low as indicated by a value of zero for both $p_{diversity}$ and $g_{diversity}$. This corresponds to the alarming stage, when the more extensive NBLS operator, $Level_{Ex_{Mut}}$ is called. As explained in chapter 5, this operator removes a randomly chosen edge at a level ‘l’ and connects the disconnected vertex and the subtree existing under it, with the most eligible edge at any level less than $l$. In literature we come across examples of multiple neighborhood based LS for GAs where the mutation operator randomly selects one of the neighborhoods say $N1$, $N2$, $N3$ and then make a move in the selected neighborhood structure[116]. Calling the two mutation operators in the order defined by the proposed diversity scheme, follows a natural search strategy where one starts with the least complex neighborhood and gradually includes the more expensive one.

6.3.5 Phase V: Local Search based Mutation

Genetic algorithm is good in exploring the search space but poor in convergence speed, while local search techniques have been proven to be good in exploiting the knowledge leading to a faster convergence speed but they may get entrapped in local optima. The memetic algorithms which are a hybridization of the GA with the local search have shown superior performance to both, as these can strike a balance between the two significant issues in searching techniques that is exploration and exploitation [145]. Our interest in local search(LS) based mutation stems from the fact that the LS methods may quickly and effectively search the basin of attraction of the optimal solutions, locating an optimum with a high degree of accuracy using fewer iterations. In $G_{-}GAT$ a novel, simplified and efficient hybrid is developed between GA and a new neighborhood based learning procedure that performs LS only when the diversity in the population as reflected by $p_{diversity}$ and $g_{diversity}$ becomes low. One of the challenges in designing a LS based hybrid of GA (HGA) is, the choice of a successful LS techniques. The choice of local search technique with which to hybridize GAs has been investigated in [154] and it was observed that the choice affects the search performance significantly and no single HGA always performs better on a diverse
set of functions. We have therefore considered twin neighborhoods whose frequency depends upon the status of diversity in the current population.

When the goal is to reduce the total costs associated with LS, the LS may be repeatedly performed on the same locally optimal region [195]. Randomly generating the initial population and applying crossover on the candidates may yield promising solutions, but it is likely that such candidates would lie far apart in the search landscape. So, in order to generate the points (candidate trees) that lie in the same basin of attraction, initial population has been generated by repeated application of DRGH. A mild recombination operator such as $k$-GCO would ensure minimum disruption in the genotypic makeup of the candidates. The high heritability quotient of $k$-GCO, as reflected by its high parent fitness correlation coefficient, further ensures that the offsprings lie in the same basin of attraction. Application of a neighborhood based local search method would thus ensure locating of the optima with a high degree of accuracy in fewer iterations, thus bringing down the cost associated with the LS based search of the landscape.

6.3.6 Phase VI: Replacement Strategy

Replacement strategy in GA plays a significant role in exerting suitable selection pressure that prevents the search from becoming random or directionless. A suitable replacement strategy ensures that the promising regions of the search space are favored for further exploitation over the non-promising regions. Improved performance has been seen in steady state GAs over generational GAs (where entire population is replaced every generation by the offspring population) due to their higher selection pressure and trade off in the exploration/exploitation caused by using different parent selection and replacement strategies [51]. $G$-GAT adopts a steady state model of replacement, as the overlapping nature of the steady state systems helps them to yield a higher selection pressure. But, the value of $\lambda$ in a typical ($\mu + \lambda$) steady state model [182] has not been adopted in $G$-GAT, but has been kept higher. Here, $\mu$ is the number of individuals in the population pool and $\lambda$ is the number of individuals replaced in every generation. In $G$-GAT, $k$ parents compete with the $2k$ offsprings for survival and the best $k$ offsprings replace the worst $k$ parents in the
current population pool. We replace the worst individual only if the new individual is better as it has been shown in [88], that deletion of the worst individuals induces a high selection pressure, even when the parents were selected randomly. In this way we have been able to bring down the number of generations required by $G$-$GAT$ to reach the optima.

6.3.7 Phase VII: Stopping criteria

Termination condition may be required to stop a genetic algorithm when a solution of sufficient quality is obtained or the computing time is limited. It may also be the case that the population is “trapped” in some unpromising area and it is preferable to restart the search. In $G$-$GAT$ we have used more than one termination criteria. The first termination criterion is that $G$-$GAT$ will be halted when a pre-set number of generations, decided a-priori, have been reached. The second termination condition is that the algorithm stops when the fitness value of the best string does not change over a specified number of generations.

In some of the experiments given in chapter 7, the stopping criteria have also been set to a certain Time limit. This prevents the algorithm from running too long. If the algorithm stops due to this condition, the results may be improved by increasing the value of Time limit.

6.4 Dynamic Vector for Representation of Trees: DVRT Structure

Before explaining in detail about the multiparent crossover operator $k$-$GCO$, it is imperative to discuss the $DVRT$ structure as this data structure is central to the design and execution of $k$-$GCO$. The edge-set representations of spanning trees in literature have considered the chromosomes to be linear structures, where each locus specifies an edge appearing in the tree [173]. Such chromosomal representations do not lend themselves optimally to crossover operators with higher arities. A higher number of parents as in FB-SCAN and U-SCAN crossovers [60], [62], [63] require repeated scanning of large number of parents in order to determine the number of occurrences of each feature. This increases the time complexity of such multiparent operators. Thus, the gain in performance achieved using
such operators is offset by the increased execution time. In order to draw benefit from a larger pool of ‘good’ parental edges (because of higher arity) identified by DRGH, the execution time of \(k\)-GCO needs to be minimized, so that \(k\)-GCO can be made competitive with other crossover operators. We have proposed a dynamic 2-D vector for the chromosome representation of trees [DVRT]. DVRT is a dynamic data structure which may evolve as the execution of the crossover operator proceeds. As seen from figure 6.3, the first row of the vector represents the edge indices of the edges appearing in the chromosome, the second row of the vector represents the number of occurrences of each edge in the parents selected and is initialized to -1 at the beginning of the \(k\)-GCO run. As the scanning phase of \(k\)-GCO proceeds, this row (i.e. frequency) reflects the status of the edges as being present in one, two or \(k\) parents respectively. The third row of the vector is a flag initialized to all zeros. This row will assume significance in the next phase of the \(k\)-GCO where we determine those edges which are incident on the centre. The evolution of this data structure along with its working has been explained in the subsequent phases of \(k\)-GCO.

![Table](image)

Figure 6.3: Dynamic Vector Representation for Tree structure.

### 6.4.1 Implication of Sorting and using DVRT Structure

The two prominent features of DVRT structure are 

1. at any stage it takes indices in a sorted order and
2. the use of tabular chromosome instead of a linear chromosome structure for the representation of spanning trees, as is prevalent in the literature. This saves significantly on the computational time by preventing rigorous redundant scanning of the parent chromosomes to determine the number of occurrences of each edge.

As explained below, keeping DVRT structure sorted prevents searching the complete structure every time while searching for the occurrence of a particular edge. This
further brings down the computational complexity of the crossover operator. *DVRT* structure thus plays a vital role in furthering the cause of increased arity of the crossover operator because more than two parents have been discouraged in traditional crossover operators due to high time complexity associated in searching the desirable edges (features) in the parents. Also, using additional fields in the chromosomal structure helps in different phases of *k-GCO* by either reducing the number of scans required, or by speeding up the segregation of edges into different sets. (Refer sections 6.5.2 and 6.5.3 for detailed explanation). The implication of sorting and using *DVRT* structure is explained with the help of following example.

**Case A:** The example shown in figure 6.4 considers a three parent crossover where it is required to determine the number of occurrences of each edge. The given example considers three parents *P1, P2* and *P3*, where the edge index 7 occurring in *P1* is to be searched in *P2* and *P3*. Case A considers the chromosomes to be unsorted single dimensional vector.

![Figure 6.4: Unsorted single-dimensional vector representation of trees](image)

The following observations have been made:

1. In the worst case the 1st Element of parent 1 requires $2 \times (n - 1)$ comparisons with two parents, where n-1 represents the number of edges in a spanning tree having n vertices.

2. So, (n-1) elements of parent 1 require $(n - 1) \times (2n - 2)$ comparisons.

   \[ \approx 2n^2 - 4n + 2 \]

3. Therefore for 3 parent crossover, number of comparisons required:
\[
\approx 3 \times (2n^2 - 4n + 2) \\
\approx 6n^2 - 12n + 6
\]

**Case B:** Case B considers the computations involved when the spanning tree is represented for crossover by a 2 Dimensional unsorted vector as shown in figure 6.5. The chromosome contains two fields ‘E’ representing the edge indices of the edges appearing in the tree and ‘F’ representing the frequency of occurrence of all the edges in the three parents.

The following observations have been made:

1. In the worst case the 1st Element of parent 1 requires \(2 \times (n - 1)\) comparisons with two parents.

2. So, \((n-1)\) elements of parent 1 require \((n - 1) \times (2n - 2)\) comparisons.

\[
\approx 2n^2 - 4n + 2
\]

3. But for second and third parents, the number of comparisons required would be significantly less because of the frequency parameter which is updated simultaneously in all the three parents. This avoids rechecking of the already scanned edges.

---

**Figure 6.5:** Unsorted 2-D vector representation of trees
4. Moreover, while scanning the second parent the first parent will not be traversed. So \((n - 1) \times (n - 1)\) comparisons would be required in the worst case.

5. For the third parent \((n-1)\) comparisons would be required in the worst case.

6. In totality, \(3 \times (n - 1)^2\) comparisons will be required in case B.

**Case C:** In case C, we have considered a sorted 2-D vector for chromosomal representation of spanning trees as shown in figure 6.6. The fields in the vector have been kept the same as in case B for a fair comparison. The significance of keeping the structure sorted becomes clear from the observations made during the scanning phase of \(k\)-GCO where each parent has to be scanned for the number of occurrences of each edge in all the parents.

**Note:** Figure 6.6 does not represent the full \(DVRT\) structure. Refer figure 6.3 for the exact \(DVRT\) structure along with all the fields.

![Figure 6.6: Sorted 2-D vector representation of trees](image)

Following observations have been made:

1. In the worst case the 1\(^{st}\) Element of parent 1 requires \(2 \times \log(n - 1)\) comparisons with two parents as it uses binary search for searching an edge index.

2. So, \((n-1)\) elements of parent 1 require \(2 \times (n - 1) \times \log(n - 1)\) comparisons.

3. As in case B while scanning the second parent the first parent will not be traversed. So \((n - 1) \times \log(n - 1)\) comparisons would be required in the worst case.
4. For the third parent no comparison would be required.

5. In totality $2 \times (n - 1) \times \log(n - 1)$ comparisons will be required in the worst case.

6. This time complexity is significantly less in comparison to the computational complexity of an unsorted 2-D vector, as the sorted vector does not require a complete scan of the chromosome every time an edge is being searched. For example, if edge 9 is to be searched in parent 2 and 3. Then a binary search on $P2$ reduces the number of comparisons as compared to Case B which makes a linear search. This is because in the worst case, sorted 2-D vector would make $\log(n)$ comparisons whereas an unsorted vector would make $n$ comparisons. A glance at the third parent $P3$ reveals that the edge index 9 does not appear in it. In case it is an unsorted vector as in case A or B, it would require a complete scan of $P3$ to determine that edge index 9 is not present in it. But in this case a binary search on $P3$ reveals that the edge index 9 does not feature in it.

6.5 *k*-Guided Crossover operator [*k*-GCO]

6.5.1 Phase I: Selection of *k* parents

The arity of *k*-GCO has a strong bearing on the ‘selection-time’ time complexity, particularly if tournament selection with a size of more than 2 is used. In order to compare the time complexity of *k*-GCO with other standard crossover operators, the selection-time time complexity is ignored. Entire time complexity of *k*-GCO can be taken into account for higher arities when the overall search dynamics are analyzed. Not taking ‘selection-time’ time complexity of *k*-GCO is important in static search behavior analysis because selecting *k* parents would require *k* 2-tournaments & therefore would increase the overall time complexity of *k*-GCO, making it appear more expensive and hence less competent than the standard crossover operators from the literature. When the dynamic search behavior is analyzed *k*-GCO seems to produce competent (better quality) solutions in lesser time as higher arities of *k*-GCO help in early convergence (not premature convergence) and hence may find the optimum in significantly lesser number of iterations.
6.5.2 Phase II: Gene Frequency based Scanning

The $k$ parents selected in phase 1 of $k$-GCO are taken and the DVRT structures corresponding to them are created. The structure corresponding to each parent contains three fields: Index, frequency and centre_edge. Index: it stores the indices of the edges appearing in the tree in sorted order. frequency: The Frequency counter keeps a track of the number of occurrences of the edge in all the $k$ parents. centre_edge: it is a flag which indicates whether the given edge is incident on the centre. frequency and centre_edge are initialized to -1 and 0 respectively (refer figure 6.3). The Frequency counter has been set to -1 to indicate those edges which have not yet been scanned.

The function Extract_Centre() when called, extracts the centre(s) of the selected parent BDMSTs from the Universal_centre array to be stored in the $k$ element array CentreParent as shown in figure 6.7. Universal_centre array is a global array which is populated during DRGH when the BDMSTs are initially created, and stores the centres of each of the BDMSTs. In case of odd diameter BDMSTs where the tree has two centres one of the centres is randomly chosen to be stored in the Universal_centre array. This function then generates a random number ‘$x$’ in the range 1 to $k$. The vertex at the $x^{th}$ position in the Centre_Parent array is extracted and stored in the variable ‘Centre’. This ‘Centre’ then forms the centre of the offsprings which are to be created in the next phase of $k$-GCO. It may be noted that all the offsprings will have the same centre and will thereby inherit similar backbone structure (edges). This will make the offsprings to lie close to each other in the fitness as well as the search landscape.

As the next stage i.e. the Gene_frequency_scanning phase proceeds, the frequency counter is updated to reflect the status of the edges as present in 1, 2 or $k$ parents respectively. The value of this field will thus fall in the range between 1 to $k$. The reason for representing the chromosomes as integers representing each edge rather than the standard edge set encoding of spanning trees as suggested by Raidl and Julstrom [174] becomes clear in this phase as the indices can be kept in sorted order thus facilitating the scanning of DVRT structures. Figure 6.8 illustrates three parent-BDMSTs along with their DVRT structures, the
associated Centre_Parent array and the corresponding Position_tracker when the parents are being scanned for edge 13.

The subroutine Gene_occurrence_scanning() checks for the presence of the given edge under consideration in each of the $k$ parents and assigns the same Frequency to that edge in all the $k$ parents. A temporary array Position_tracker, of size $k$ is created at this stage. It is used for tracking the position of current edge in all the $k$ parents. It helps to update DVRT to reflect the frequency of each gene in each of the $k$ parents. These $k$ positions are then updated simultaneously when the final value of the number of occurrences of that edge has been computed. The same can be understood from the example given in figure 6.8. Edge 13 is at position 2 in parent1 and 3.

![Diagram](image.png)

**Figure 6.7: Working of the Extract_Centre() function**

The value -1 for parent 2 indicates that edge 13 is not present. Storing edge indices in sorted order is a pre requisite for frequency checking routine as it saves a lot on computational time complexity. The value of position tracker is re initialized to -1 at every iteration of scanning i.e. before we begin checking the frequency of the next edge. To populate the position tracker a binary search is performed on $k$-1 parents to determine whether the given edge is present or not and, if present, at which position. The search
procedure under any given circumstances would not perform more than \( \log(n) \) comparisons as already discussed in section 6.4.1.

The function \textit{Centre Incidence()} is then invoked to check whether the given edge is incident on the centre or not. This function returns true or false depending on whether the current edge is incident on the centre or not. If one of the extracted pair of vertices from this edge is the centre, this procedure returns true. If the edge is found not to be incident on the centre, false is returned.

\begin{itemize}
  \item \textbf{a)} Three parent- BDMSTs along with their DVRT structures.
  \item \textbf{b)} Centre_Parent array for Parent1, Parent2 and Parent3.
  \item \textbf{c)} Position_tracker depicting the position of edge 13 in all the three parents.
\end{itemize}

Figure 6.8 (a) to (c): An example of a three parent crossover.
This is done by simply extracting the pair of vertices from the edge under consideration and checking if one of the vertices is centre. Here centre is the vertex which is stored in the Centre_Parent at the $i^{th}$ position corresponding to the $i^{th}$ parent. The following procedure is adopted while extracting the vertices from the edge indices.

### Retrieving Vertices from Edge Index ‘E’

![Graph Example](image)

- An example of a 5 node undirected graph where edge indices are computed only for the upper triangle.
- Formula to generate edge indices on the fly.

**Figure 6.9 (a) to (b) Assignment and generation of Edge indices.**

Considering figure 6.9 as an example BDMST, let us say that edge 14 is the edge under consideration and it is required to extract the vertices $v1$ and $v2$ corresponding to this edge E. Using the formula given above we extract $v1$ as 3 and $v2$ as 4. Here $n$ is 5 which represents the number of vertices. It is then checked in the array Centre_Parent to determine whether 3 or 4 happen to be the centre for this BDMST. If either $v1$ or $v2$ happen to be centre, the centre_edge flag for this edge will be set to 1 indicating that this edge is incident on the centre otherwise the centre edge flag will remain zero.

**Algorithm : Gene Frequency based Scanning**

**Gene Frequency based Scanning // Phase 2**

**Input: Number of nodes 'n', arity of GCO 'k', length of chromosomes 'l'

Gene_Frequency_scanning()
{

**Stage 1: DECLARING THE CHROMOSOME**

128
Struct Track
{
    int index[l];
    int frequency[l];
    int centre_edge[l];
} Parent[k];

Stage 2: INITIALIZING THE FREQUENCY COUNTER AND FLAG TO ZERO

for(i = 1; i ≤ k; i ++)
{
    for (j = 0; j < l; j ++)
    {
        Parent[i].frequency[j] = -1;
        Parent[i].centre_edge[j] = 0;
    }
}

Stage 3: EXTRACTING THE CENTRES FROM THE UNIVERSAL CENTRE ARRAY

Extract_Centre()
{
    x = random (k);
    Centre = Centre_Parent[x]
}

Stage 4: GENE OCCURRENCE SCANNING

Gene_occurrence_scanning()
{

    for(i = 1; i ≤ k; i ++)
    {
        for (j = 0; j < l; j ++)
        {
            var = Parent[i].index[j];
            val = Centre_Incidence(Parent[i].index[j]);
            Position_tracker[i] = j;
            Counter = 1;
            if(Parent[i].frequency[j] = -1) // if the edge has not been scanned already
            {
                t = 0;
                if( m = i + 1; m ≤ k; m + +) //this ensures that if number of occurrences of an edge present in parent 1 is to be ascertained, searching should not be done on parent 1 itself.
                {
                    Position_tracker[m] = -1;
                    while(Parent[m].index[t] < Parent[i].index[j])
                    {
                        t ++;
                    }
                }
            }
        }
    }
}
6.5.3 Phase III: Inheriting Parental genes

In this phase of $k$-GCO, the construction of offsprings/BDMST begins. Exploitation of promising regions (as identified by parent BDMSTs) is aided by a higher number of parents as multiparent recombination is proven to have a better exploitation capability than the binary recombination [60]. A point to note here is that $2k$ offsprings will be constructed out of $k$ parents. These $2k$ offsprings are similar to each other for a large number of edges and differ only in few edges. Thus, these offsprings tend to lie near to each other in the search
landscape. Further, it has been observed that the design process of EAs for any problem should be guided by the locality principle [175], as this ensures a better exploitation of the identified regions of the search landscape. Generating a large number of such offsprings coupled with the high heritability quotient of $k$-GCO ensures that the identified (promising) regions of the landscape are well exploited. In this phase, the subroutine $\text{Diameter\_Constraint\_Handling()}$ is called which allows for inheritance of parental edges in the offsprings while ensuring that the diameter constraint and the constraint of cycle-free structure are not violated. A High heritability in $k$-GCO is ensured in its design structure through the following procedures:

### 6.5.3.1 Stage 1: Parent\_Centre\_Add()

This routine accesses the array $PC$. $PC$ is the set of those edge indices which form a direct edge with the chosen centre. For example, from figure 6.8 vertex 6 is chosen as the centre of the offsprings. The indices of the edges which are incident on the centre vertex are populated in the array $PC$ by checking the $centre\_edge$ field of $k$ parents. Indices with $centre\_edge$ value of 1 in the $DVRT$ structure of the parents, populate $PC$. The integer labels on the arcs in figure 6.8 are not the weight of these arcs but the indices of those arcs/edges. The edges 30, 38 and 47 from parent1, 48 from parent2 and 6 from parent3 are added to the partially constructed BDMSTs which at this stage only comprise of the centre ‘Centre’. At this stage, the BDMST is a star and because all the $2k$ offsprings are supposed to being constructed simultaneously, $2k$ identical stars are constructed here. As can be seen from figure 6.10, a 3 parent $k$-GCO has yielded 6 identical offsprings till the parental edges incident on centre have been added. It may be noted that the edges are being added to the growing offsprings in sorted order.
At this stage a two dimensional array $Vertex\_depth$ is initialized. $Vertex\_depth$ is an array which stores the depth of each vertex from centre for a particular BDMST. The $i^{th}$ position of the array refers to the $i^{th}$ vertex. This array is initialized to -1 indicating that the vertices have not yet been added to the tree. As the execution of this stage proceeds, the values of this array are continuously updated. The significance of $Vertex\_depth$ becomes more evident in the next stage where it is used to check that the feasibility criteria i.e Cycle free graph and diameter constraints do not get violated. Figure 6.11 shows a $Vertex\_depth$ array which has been initialized for a 8 vertex instance graph.

For each of the indices in $PC$, the pair of vertices is extracted using the formula given in figure 6.9. Vertex which is not the centre is assigned a value of 1 in the $Vertex\_depth$ array indicating that this vertex is at a depth of 1 from the centre. Centre vertex has a depth of zero. Vertex 6 which is the centre vertex in the example has been assigned a depth of zero in the $Vertex\_depth$ array (refer figure 6.11). Vertex 2 has been shown to be at a depth of two from the centre.
6.5.3.2 Stage 2: Parent_Add( )

This routine accesses the arrays $P_1, P_2, P_3...P_k$ which contains those parental edges which do not form direct edge with centre. Here, $P_1, P_2, P_3...P_k$ denote the priority classes into which the set of parental edges have been classified. $P_k$ denotes the edges which are present in all the k parents. $P_3, P_2$ and $P_1$ denote the edges which are present in 3, 2 and 1 number of parents respectively. These sets are populated using the frequency counter field, frequency in the DVRT structure of the parents. From figure 6.10 the edges, for example edge 47, whose frequency field has a value 3 is present in all the three parents. The segregation of the edges into priority classes follows the argument that the parents are all above average structures generated using DRGH and according to the commonality hypothesis the schemata common to the above average solution candidates are above average [36], so they must be preserved in the offsprings. And more the number of parents having a particular edge, higher is the priority class of that edge. The edges are considered in the descending order of their priority class, i.e. the edges appearing in all the k parents are considered to be the most eligible (drawing again from the commonality hypothesis [36]) and are therefore considered first, then the edges belonging to the priority class $k-1$ are considered and so on. The highest priority edges are considered first so that maximum of them can be accommodated in the offspring.

For each of the indices picked up in order from the highest priority class onwards, the pair of vertices is extracted using the formula as given in figure 6.9 (b). When the vertex pair is extracted the following cases may arise:

(i) Both the vertices are not yet part of $Vertex\_depth$ vector.

(ii) One of the vertices is in $Vertex\_depth$ vector.

6.5.3.3 Stage 3: Intermediate_Component_Handler( )

This routine takes care of the above two cases. A temporary structure called Disconnected_Component_Handler is created which stores these disconnected intermediate vertices till these are added to the partially constructed BDMST. It is a linked
list based structure as shown in figure 6.12, in which a new component is initialized every time a disconnected component is detected.

**Case (i):** An edge index is considered from $P[i]$ and the corresponding pair of vertices is extracted. Now if both the extracted vertices are not yet a part of the tree then edge cannot be added to the tree till one of the vertices becomes a part of the tree. Such a pair of vertices is added to the `Disconnected_Component_Handler`, and is assigned an initial status of -1. This pair would not be processed till one of the vertices is included in the `Vertex_depth` array. Another edge index is now considered from suitable $P[i]$ and if the extracted pair of vertices is again such that neither of them is a part of the growing spanning tree then another component is created and such disconnected pair of vertices are grouped together using linked list structure as `Disconnected_Component_Handler`.

![Structure definition and Initialization of Disconnected_Component_Handler](image)

*Figure 6.12: (a) to (b) Structure definition and Initialization of Disconnected_Component_Handler*

Every time such pair of vertices is encountered:

a) The `Disconnected_Component_Handler` list is checked to see if one of the vertices is already in the list. If one of the vertices exists in the list then a new component is created and assigned the status of existing component holding the vertex. For example figure 6.13 shows the first disconnected pair of vertices as (13, 15) which is entered into the `Disconnected_Component_Handler` and is accorded the status -1. When another disconnected pair of vertices (14,16) is encountered, the `Disconnected_Component_Handler` list is scanned to see if either vertex 14 or 16 is
already present in the list. If not then the status value is decremented by one and -2 is assigned to this pair.

b) Again, it can be seen from figure 6.13 that a third disconnected vertex pair (16,17) is encountered, the list is again scanned to see if either vertex 16 or 17 is already present in the list. In this case, vertex 16 is already present in the list with a status of -2, so a new component is created and the same status -2 is assigned to the vertex pair (16,17).

<table>
<thead>
<tr>
<th>Vertex1</th>
<th>13</th>
<th>14</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex2</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>Status</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
</tr>
</tbody>
</table>

Figure 6.13: Status assignment in Disconnected_Component_Handler when condition b) holds

c) If both the vertices are present in the list having separate status. This will be the case when, let us say, a disconnected vertex pair (13,17) is encountered, refer figure 6.14 (a). The list is scanned to see if the vertices already exist in the list. In this case vertex 13 has a status of -1 and vertex 17 has a status of -2. This new component is given the higher status of the two i.e it is assigned a status of -1. The entire Disconnected_Component_Handler list is scanned and wherever vertex 17 is encountered, the status of the vertex pair is changed to -1. This starts a chain of updations, now vertex 16 is present in two different components with different status, so the higher status is given to the vertex pair (14,16) as shown in figure 6.14 (b).

<table>
<thead>
<tr>
<th>Vertex1</th>
<th>13</th>
<th>14</th>
<th>16</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex2</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Status</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex1</th>
<th>13</th>
<th>14</th>
<th>16</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex2</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Status</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Figure 6.14 (a) to (b): Status assignment in Disconnected_Component_Handler when condition c) holds.

This chain of updations is necessary because if one of the vertices from the above list, say, 13 is added to the list, all the vertices which form a direct edge with vertex 13 can now be
added to the tree, for example the edges between (13,15) and between (13,17). Vertex 15 and 17 now become a part of the growing tree and therefore vertices which form an edge with them, for example 16, can be added to the tree. Once vertex 16 becomes a part of the tree, then the edge between the vertex pair (14,16) can also be added to the tree. So it is important that if the same vertex occurs in two different components with two different status values, their status should be changed to be the same.

**Case (ii):** An edge index is considered from $P[i]$ and the corresponding pair of vertices is extracted. Now if one of the vertices from the extracted pair is already in the $Vertex_depth$ array (i.e the growing tree) as can be reflected from the depth value of that vertex from the $Vertex_depth$ array where a -1 would indicate that the vertex is not yet a part of the tree (figure 6.11). Let us assume that from the extracted vertex pair $(x1,x2)$, $x1$ is already a part of the tree and vertex $x2$ is not included in the tree. Note that such a vertex pair would not form a Disconnected Component and therefore will not be added to the $Disconnected_Component_Handler$ list. The $vertex_check( )$ routine will be called in this case that adds vertex $x2$ to vertex depth array. This will be done by computing the vertex depth for $x2$ equal to depth of $x1 + 1$. $Vertex_depth[x2]$ will be updated if $Vertex_depth[x2] \leq D/2$, i.e the diameter constraint is not violated. In case $Vertex_depth[x2] > D/2$, $Vertex_depth[x2]$ remains -1 and is not updated, which means that the edge $(x1,x2)$ is not added to the growing BDMST as adding that edge violates the given diameter constraint. In case the diameter constraint is satisfied the edge index for the vertex pair $(x1, x2)$ is computed and is Included in all the $2k$ *offsprings*. Next scan the $Disconnected_Component_Handler$ list and check for vertex $x2$ in it. If $x2$ exists, extract the second vertex $V$ and compute $Vertex_depth[V]$ equal to depth $x2 + 1$. If it does not violate the diameter constraint then update $Vertex_depth$ for vertex $V$. Now as explained above, consider all those vertices which are connected to vertex $x2$ or hold the same status as that of $x2$ in the $Disconnected_Component_Handler$ list and do not violate the diameter constraint. Update their depth in $Vertex_depth$ array if the diameter
constraint is not violated. Compute the corresponding edge indices and add them to all the 2k offsprings.

**Algorithm:** *Inheriting Parental Edges*

**Phase 3: Inheriting Parental Edges**

Call subroutine *Diameter_Constraint_handler()*

```latex
\{ \\
  \text{for} (i = 1; \ i \leq k; \ i ++) \\
  \{ \\
    \text{Call subroutine *Parent_Centre_Add(p[i]*)} \\
  \} \\
  \text{for} (i = 1; \ i \leq k; \ i ++) \\
  \{ \\
    \text{for} (j = 0; \ p[i][j] \neq 0; \ j ++) \\
    \{ \\
      \text{Call subroutine *parent_Add(p[i][j]*)} \\
    \} \\
  \}
\}
```

**Subroutine Parent_Centre_Add()**

**Parent_Add(a)**

```latex
\{
  \text{int} \ i = 0, \text{vertex, vertex1, vertex2};  \\
  \text{Static int} \ p = 0;  \\
  \text{while} (a[i] \neq 0) \\
  \{ \\
    \text{index1} = a[i]; \\
    \text{if} (\text{index1} \% n = 0) \\
    \{ \\
      \text{vertex1} = n \text{ and vertex2} = \text{index1}/n; \\
    \} \\
    \text{else} \\
    \{ \\
      \text{vertex1} = \text{index1} \% n \text{ and vertex2} = \text{index1}/n + 1 \\
      \{ \\
        \text{if} (\text{vertex1} = \text{Centre}) \\
        \{ \\
          \text{vertex} = \text{vertex2}; \\
        \} \\
        \text{else} \\
        \{ \\
          \text{vertex} = \text{vertex1}; \\
        \} \\
        \text{vertex_depth}[\text{vertex}] = 1;
    \}
\} \\
```

137
\* vertex_depth is an array storing the depth of each vertex from centre.
The \(i^{th}\) position of the array refers to the \(i^{th}\) vertex */

\(\text{for}(i = 1; \ i \leq 2k; \ i++)\)

\{
    \text{Offspring}[i][p] = index1;
\}

increment \(p\) by 1;
\} \end of \text{while}
\} \end of \text{function}

\textbf{Subroutine Parent\textunderscore Add()}

\textbf{Parent\textunderscore Add(a)}

\{
    \text{Static int} \ \text{Neg\textunderscore Counter;}
    \text{flag} = 0;
    \text{index1 = a;}
    \text{if}(\text{index1} \ % n = 0)
        \text{vertex1 = n and vertex2 = index1/n;}
    \text{else}
        \text{vertex1 = index1} \ % n \ \text{and vertex2 = index1/n + 1}
    \text{if}((\text{vertex\textunderscore depth[vertex1]} = -1) \ \text{&&} \ (\text{vertex\textunderscore depth[vertex2]} = -1))
        \text{Call subroutine} \ \textbf{Intermediate\textunderscore Component\textunderscore Handler()};
\}

\textbf{Subroutine Intermediate\textunderscore Component\textunderscore Handler()}

\textbf{Intermediate\textunderscore Component\textunderscore Handler()}

\{
\textbf{Stage 1: Declaring Disconnected Component handler}

\textbf{Struct} node
\{
    \text{int} \ v1;
    \text{int} \ v2;
\}
int status;
Struct node * link;
} component

Stage 2: Create a Disconnected Component handler

Create a temp component

temp -> v1 = vertex1;
temp -> v2 = vertex2;
temp -> link = NULL;
if (start = NULL)
{
    start = temp;
    start -> status = Neg_Counter;
}
else
    ptr = start;
Neg_Counter = Neg_Counter + 1;
while(ptr -> link != Null)
{
    if((ptr -> v1 = vertex1) OR (ptr -> v2 = vertex2))
    {
        if (flag = 0)
        {
            Negative = ptr -> status;
            flag = 1;
        }
        else
        {
            change = ptr -> status;
        }
    }
    ptr -> link = temp;
    ptr = ptr -> link;
    ptr -> status = negative;
} \end of while

ptr = start;
while (ptr -> link != Null)
{
    if(ptr -> status = Change)
    {
        ptr -> status = negative;
        ptr = ptr -> link;
    }
}
if((vertex_depth[vertex1] = -1) && (vertex_depth[vertex2] > 0))
    Call subroutine vertex_check(vertex2,vertex1);
else if((vertex_depth[vertex1] > 0) && (vertex_depth[vertex2] = -1))
    vertex_depth(vertex1,vertex2);
}

Subroutine vertex_check( )

vertex_check(x1,x2)
{
    int j = 0;
    if(Vertex_depth(x2) = -1)
    {
        t = Vertex_depth[x1] + 1;
        if(t < Diameter/2)
            Vertex_depth[x2] = t;
        if(x1 < x2)
            index1 = n * (x1 - 1) + x2;
        else
            index1 = n * (x2 - 1) + x1;
        for(i = 1; i <= k; i++)
        {
            Offspring[i][p] = index1;
        }
        increment p by 1;
    }
    set ptr1 = start;
    ptr = ptr -> link;
    while(ptr != Null)
    {
        if(ptr -> v1 = x2

140
6.5.4 Phase IV: Random Subgraph Introduction

Till this stage of $k$-GCO all the $2k$ offsprings are identical. All the $2k$ offsprings have been populated with as many parental edges as could be added keeping the constraints of diameter and cycle free graph satisfied. At this stage, all the edges from all the priority classes have been considered and we have exhausted all the parental edges that could be inherited in the offsprings. A few vertices are still left which have not yet been added to the tree. This is evident from the $Vertex_depth$ array. Such left over vertices are grouped and kept in $left_over_vertex$ array.
The 1-D vector \( Vertex\_depth \) is then converted to 2D vector \( Depth \), as this marks the point where \( Vertex\_depth \) for \( 2k \) offsprings would vary. The \( Random\_subgraph\_introduction() \) subroutine is then called, that introduces random edges in the offsprings. These edges will be randomly generated for all the \( 2k \) offsprings, and hence from this point onwards the offsprings will be different BDMSTs. The random edges are generated such that the vertices from the \( left\_over\_vertex \) pool can be connected to the tree. This is done selecting a random vertex \( x \) from the pool of vertices such that \( Vertex\_depth[x] \) is not equal to -1 and its inclusion does not violate the diameter constraint, that is,
\[
Depth[j][x] + 1 \leq \text{diameter}/2
\]
where \( j \) is the \( j^{th} \) offspring. The first unused vertex \( p \) from \( left\_over\_vertex \) array is then taken and the edge index for the vertex pair \((x, p)\) is computed. This edge is then added in \( offspring[j]\). For each \( offspring[i] \), for all \( i \) ranging from 1 to \( 2k \), such a random edge is determined anew. Depth of the vertex \( p \) is then updated in the vector \( Depth[j][p] \). This procedure is repeated for each of the vertex in the \( left\_over\_vertex \) array to form an edge with a randomly selected vertex for each of the \( k \) offsprings.

This ensures maximal preservation of parental substructures in the offsprings, ensuring a high heritability of \( K\text{-GCO} \), and the offsprings lie close to the parents in the search space. The locality of \( K\text{-GCO} \) has also been designed to be high. The offsprings lie close to each other in the search space. This is a desirable property of a recombination operator. In order to have a high locality, the partially constructed offspring trees are extended by adding the edges from \( PC[i]'s \) and then \( P[i]'s \) to all the offsprings. All the \( 2k \) \( DVRT \) structures are similar in these edges. It is only when the random edges are added that the stochastic component of \( K\text{-GCO} \) comes into play and the remaining edges are different in all the offsprings.
Algorithm: Random Subgraph Introduction

Random Subgraph Introduction // Phase 4

Random_Subgraph_Introduction() 
{
    m = 0;
    for (i = 0; i < n; i++)
    {
        for (j = 1; j ≤ k; j++)
        {
            depth[j][i] = vertex_depth[i];
        }
        if (vertex_depth[i] = -1)
        {
            left_over_vertex[m] = i;
            m = m + 1;
        }
    }
}

for (i = 0; i < m; i)
{
    for (j = 1; j ≤ k; j++)
    {
        do
        {
            x = random(n);
        }
        while ((vertex_depth[j][x] < 0) OR (vertex_depth[j][x] + 1 > Diameter/2))
        l = left_over_vertex[m];
        vertex_depth[j][l] = vertex_depth[j][x] + 1;
        if (l < x)
\[ \text{index1} = n \times (l-1) + x; \]

\text{else}

\[ \text{index1} = n \times (x-1) + l; \]

\[ \text{offspring}[j][p] = \text{index1}; \]

\} \text{ end of j loop}

\text{increment p by 1;}

\} \text{ end of i loop}

The design of \textit{k-GCO} thus ensures that minimum amount of non-parental edges are included in the offsprings. Further, the execution time of \textit{k-GCO} is greatly reduced due to the DVRT structure of its chromosomes which facilitate efficient scanning and segregation of edges into different priority classes. Furthermore, the diversity management scheme allows an effective switching between the neighborhoods, with an intelligent incremental evaluation of the neighborhoods. The above factors, along with an efficient replacement strategy contribute significantly in making \textit{G-GAT} a robust and efficient strategy to solve fairly large instances of BDMST problem in reasonable amount of time.