CHAPTER 4

Heuristic Algorithms for Graph Problems

Abstract

This chapter discusses new heuristic and meta-heuristic algorithms for Orienteering and Constrained Shortest Path problems. An algorithm has been proposed for the orienteering problem that can handle complete as well as incomplete graphs. This algorithm gives superior performance in comparison to other algorithms reported in the literature. The experimental analysis of this algorithm shows that the roulette wheel selection heuristic performs better than the other selection methods. Having established roulette wheel heuristic as the best strategy, we compare its performance against Ostrowski's algorithm. We found that the new algorithm RWS_OP easily outperforms Ostrowski's algorithm. In fact, RWS_OP performs better both in terms of score and time i.e., helps in achieving a better total collected score and utilizes the given time budget more effectively. Further, for complete graphs a meta-heuristic based on flower pollination (FPA_OP) is shown to perform better than other reported algorithms, especially for the larger values of time budget. A heuristic has also been proposed for the constrained shortest path problem which uses bidirectional search. This algorithm improves the results (for the average case) of the recently reported path delay discretization algorithm suggested by Chen et al.

In section 4.1, an introduction to heuristics has been presented. Section 4.2 presents a comparison of various selections methods for the orienteering problem. The results of the roulette wheel selection method are compared with the results known for incomplete graphs in section 4.3. Section 4.4 presents the implementation of the flower pollination algorithm for the orienteering problem. In section 4.5, the bidirectional search heuristic for the constrained

shortest path problem has been proposed and finally the conclusion of the chapter is stated in section 4.6.

4.1 Introduction

Most of the optimization problems are NP-Hard and to solve these problems, we need efficient algorithms that can generate optimal solutions in polynomial time. However, it is difficult to obtain an algorithm that simultaneously possesses three properties: (1) computes optimal solutions, (2) for any instance and (3) in polynomial time (Williamson & Shmoys, 2010). To tackle the NP-Hard optimization problems, the decision maker needs to compromise with at least one of the above stated requirements. A category of algorithms that compromises with the polynomial-time solvability and generates an optimal solution for an instance by exploiting the entire search space is called an exact algorithm. But in case of exact algorithms, it can never be predicted that for the next input instance, how much time is required for determining the solution. It can take seconds, minutes, hours or even days. So exact algorithms might require immensely large amount of time to deal with the formidable challenges and hence are not very appropriate for the NP-Hard problems. Another category of algorithms are the heuristic algorithms that relaxes the requirement of determining the optimal solution and settles with a near to optimal solution. These algorithms generate a solution that is good enough, in polynomial time for an input instance of any size. Approximation algorithms forms the third category where again the compromise is with the optimal solution but the advantage of these algorithms is that the relaxation is minimized to the extent where an upper bound can be set on the quality of the solution.

Therefore, the problems that have large inputs and which cannot be solved in polynomial time can be dealt with using heuristic algorithms. The word heuristic was derived from the Greek word “heuriskein” which means to discover and a heuristic algorithm is basically used to gain some knowledge or compute some desired result by using the rule-of-thumb, trial and error method, intelligent guesswork etc. instead of implementing some pre-established formula. Heuristic algorithms have the advantage of computing a solution for NP-Hard or NP-Complete problems with tolerable time and space complexity at the cost of optimality of the solution i.e., one needs to compromise with the quality of the solution to generate one with acceptable time and space complexity. However, the solution computed using a heuristic is most of the

times a near to optimal solution and for real life applications it is sufficient to have an approximate or partial solution. Most of the practical problems can be presented as an optimization problem. The set of possible solutions for these optimization problems is often referred to as the search space and the algorithms used to explore the search space are regarded as the search algorithms (Kokash, 2005).

In this chapter, we concentrate on heuristic algorithms and propose few heuristics to deal with the considered research problems viz. orienteering problem (OP) and constrained shortest path problem (CSPP).

4.2 Comparison of Selection Methods for Orienteering Problem

Orienteering problem (OP) is an NP-Hard graph problem. In OP the aim is to determine a Hamiltonian Path P that connects the stated source (v_1) and target (v_N), includes a subset (V') of the vertex set V such that the total collected score can be maximized within the stated time budget (T_{max}). OP can be represented by an undirected weighted graph (complete or incomplete) $G(V, E)$ where V is the set of vertices and E is the set of edges. We associate two functions, that is, a time function and a score function to the edges and vertices respectively. Let $t: E \rightarrow \Re^+$ denote the time function and $S: V \rightarrow \Re^+$ signify the score function. Consequently, for a subset $V' of\ V$ and $E' of\ E$, we have $S(V') = \sum_{v \in V'} S_v$ and $t(E') = \sum_{e \in E'} t(e)$ respectively (Vansteenwegen, Souffriau, & Oudheusden, 2011).

OP finds application in several fields like logistics, transportation networks, tourism industry, etc. Most of the existing algorithms for OP can only be applied on complete graphs that satisfy the triangle inequality. Real-life scenario does not guarantee that there exists a direct link between all control point pairs or that the triangle inequality is satisfied. To provide a more practical solution, we propose a stochastic greedy algorithm (SEL_OP) that uses the various selection methods, does not require that the triangle inequality condition be satisfied and is capable of handling both complete as well as incomplete graphs.

The OP can be stated as an integer programming problem which is as follows (Vansteenwegen, Souffriau, & Oudheusden, 2011):

$$Max \sum_{i=1}^{N-1} \sum_{j=2}^{N} S_i x_{ij} \quad (4.1)$$

$$\sum_{j=2}^{N} x_{1j} = 1 \quad , \quad \sum_{i=1}^{N-1} x_{iN} = 1 \quad (4.2)$$

$$\sum_{i=1}^{N-1} x_{ik} \leq 1 \quad \forall\, k = 2, \ldots\ldots, N-1 \quad (4.3)$$

$$\sum_{j=2}^{N} x_{kj} \leq 1 \quad \forall\, k = 2, \ldots\ldots, N-1 \quad (4.4)$$

$$\sum_{i=1}^{N-1} \sum_{j=2}^{N} t_{ij} x_{ij} \;\leq\; T_{max} \quad (4.5)$$

$$2 \leq u_i \leq N \quad \forall\, i = 2, \ldots\ldots, N \quad (4.6)$$

$$u_i - u_j + 1 \leq (N-1)\left(1 - x_{ij}\right) \quad \forall\, i, j = 2, \ldots\ldots\ldots, N \quad (4.7)$$

$$x_{ij} \,\epsilon\, \{0,1\} \quad \forall\, i, j = 1, \ldots\ldots, N \quad (4.8)$$

Eq. 4.1 represents the objective function of OP i.e., maximization of the total collected score. Few other constraints like a path should have v_1 as its starting and v_N as its ending node is taken care by Eq. 4.2 and no vertex is visited more than once and the path remains connected is ensured by Eqs. 4.3 - 4.4. The important condition that the path satisfies the time bound (T_{max}) is represented by Eq. 4.5. The need to eliminate sub tours is executed by Eq. 4.6 and 4.7. Variable u_i signifies the position of vertex v_i in the path and x_{ij} = 1 if v_j is visited after v_i otherwise, x_{ij} = 0.

4.2.1 Selection Methods

(I) Tournament Selection

In this selection technique, one individual is chosen from a population of individuals. Several tournaments are run among the randomly selected individuals from a population and the best ones are copied to the next generation (those that win the tournament i.e., have the best fitness value). The process is repeatedly performed to saturate the population. The tournament size is denoted by q and the most common tournament size is q=2. By altering the tournament size, the selection pressure can be easily adjusted. In case the tournament size is large, then weak individuals have a lesser chance of getting selected. Some advantages of this method are (1) it does not require any sorting mechanism and therefore can be implemented in O(n) time and (2) it supports

parallel architecture (Back, 1994; Razali, & Geraghty, 2011; Sivaraj, & Ravichandran, 2011).

(II) (μ, λ) Selection

This selection technique was formulated with the idea of reducing the offspring population generated as a result of recombination and mutation. In this case, the offspring population of size $\lambda \geq \mu$ is reduced by selecting μ best offspring individuals as new parents for the next generation. It is an easy to implement technique, takes less time and practically follows the greedy approach therefore generates good results and helps in most of the cases (Back, 1994; Razali, & Geraghty, 2011; Sivaraj, & Ravichandran, 2011).

(III) Roulette Wheel Selection

This selection method is also called proportional selection method and uses the fitness proportionate approach. It selects an element randomly from a list on the basis of the fitness function. The elements having a higher fitness value have a greater probability of getting selected, however, even the elements with a lesser fitness value has a non-zero probability of getting selected (Back, 1994; Razali, & Geraghty, 2011; Sivaraj, & Ravichandran, 2011).

(IV) Random Selection

It is the simplest selection method. In this technique, any node is randomly selected from the set of available candidate nodes i.e., a node is selected only because it is readily available and can be conveniently used for further operation without performing any other processing. This method can explore a large search space and provides a wide range of answers. Also, if the technique is allowed to run large number of times, it may generate the best possible solution (Back, 1994; Razali, & Geraghty, 2011; Sivaraj, & Ravichandran, 2011).

4.2.2 Algorithm for Incomplete and Complete Graphs

Input: A graph $G(V, E)$ with t_{ij} (time taken to traverse) value and S_i (score) value of each edge and each vertex respectively. A constant $PathListSize$ value which denotes the maximum number of paths that are considered at each level.

Output: A Hamiltonian path with the best possible total collected score such that the total travel time is within the stated time bound.

$SEL\_OP(G, PathListSize, T_{max})$

1. $\boldsymbol{Create}\ PathList;$
   // Array of paths which is initially empty.
2. $PathList \leftarrow \emptyset;$
3. $Path\ \mathrm{P} \leftarrow Dijkstra(v_1, v_n);$
   // Shortest path between source (v_1) and target (v_N).
4. $\boldsymbol{Insert}\ P\ to\ PathList;$
5. $\boldsymbol{return}\ Generation(PathList);$

$Generation(PathList)$

1. $\boldsymbol{Create}\ NewPathList\ , ChildpathList;$ // Queues storing paths.
   $NewPathList \leftarrow \emptyset;$
   $ChildPathList \leftarrow \emptyset;$
2. $\boldsymbol{for}\ i \leftarrow 0\ to\ |PathList|$
   $a.Selection(PathList[i], ChildPathList);$
   //$ChildPathList$ will contain children generated from each path in $PathList$.
3. $\boldsymbol{for}\ i \leftarrow 0\ to\ PathListSize$
   // Selecting best $PathListSize$ children for next generation.
   $a.ChildPath = BestPath(ChildPathList);$
   // The path with the maximum total score.
   $\boldsymbol{Remove}\ ChildPath\ from\ ChildPathList;$
   $\boldsymbol{Insert}\ ChildPath\ to\ NewPathList;$
   $b.\boldsymbol{if}\ |ChildPathList| == 0$
   $break;$

4. $\boldsymbol{if}\ NewPathList == PathList$

// Terminate if no new child is generated and return the $BestPath$.

$\boldsymbol{return}\ BestPath(PathList);$

5. $\boldsymbol{return}\ Generation(NewPathList);$

$Selection\ (Path\ P, ChildPathList)$

//$Path$ is an array of nodes that forms a path.

1. $q_{max} \leftarrow\ 0;$
2. $\boldsymbol{for}\ i \leftarrow 0\ to\ |V|$
3. $a.\boldsymbol{if}\ i \in P$

// If a node is already present in the path then ignore it.

$\boldsymbol{continue};$

$b.\boldsymbol{Calculate}\Delta t_i\ ;$

// The time increment due to insertion of v_i at its best position.

$$c.\ q_i = \begin{cases} S_i/|\Delta t_i|\,, & \Delta t_i\ \geq 1 \\ S_i, & -1 \leq \Delta t_i < 1 \\ S_i * |\Delta t_i|, & \Delta t_i < -1 \end{cases}$$

$d.\boldsymbol{if}\ q_{max} \leq\ q_i$

$q_{max} \leftarrow q_i$

4. $Create\ CandidateList\ ;$

// Array of candidate nodes used for selection.

5. $CandidateList \leftarrow \emptyset;$
6. $\boldsymbol{for}\ i \leftarrow 0\ to\ |V|$
7. $a.\boldsymbol{if}\ i \in P$

$\boldsymbol{continue};$

$b.\boldsymbol{if}\ (q_i\ \geq\ \alpha\ q_{max})\&\&\ (t_{parent} + \Delta t_i \leq T_{max})$

// α is the greediness parameter that decides which node should participate in selection process.

$c.\boldsymbol{Insert}\ v_i\ to\ CandidateList\ ;$

8. $\boldsymbol{if}|CandidateList| == 0$

$\boldsymbol{Insert}\ P\ to\ ChildPathList;$

// If no new nodes are added then insert the parent path P.

$\boldsymbol{Return};$

9. $\boldsymbol{for}\ i \leftarrow 0\ to\ PathListSize$

// Generates $PathListSize$ number of children paths from path P.

10. $a.\ ChildNode \leftarrow Compute(CandidateList)$

// Compute 1 using tournament selection.

// Compute 2 using (μ, λ) *selection.*

// Compute 3 using roulette wheel selection.

// Compute 4 using random selection.

$\boldsymbol{Remove}\ ChildNode\ from\ CandidateList;$

$\boldsymbol{Insert}\ ChildNode\ to\ Path\ P\ and\ \boldsymbol{Insert}\ Path\ P\ to\ ChildPathList.$

$\boldsymbol{Remove}\ ChildNode\ From\ Path\ P;$

$b.\ \boldsymbol{if}\ |CandidateList| = 0$

$\boldsymbol{break};$

$BestPath(PathList)$

1. $max \leftarrow 0; bestpath \leftarrow 0;$
2. $\boldsymbol{for}\ i \leftarrow 0\ to\ |PathList|$

a. $\boldsymbol{if}\ \big(Score\big(PathList(i)\big)\big) > maxScore$

// *Score* (*Path P*) is the sum of the rewards associated with each node of *Path P* .

$b.\ maxScore \leftarrow Score\big(PathList(i)\big);$

$bestpath \leftarrow i;$

3. $return\ PathList(bestpath);$

The aim of above stated algorithm is to determine a path that connects the source (v_1) and target (v_N) and a subset of vertices that maximize the total collected score and obeys the time limit. This algorithm can be implemented on both complete as well as incomplete graphs. To ensure that in a path, source and target is connected we implement the Dijkstra's algorithm as shown in step 3 of $SEL\_OP()$. To take care of the other constraint that no vertex is visited more than once, the explored nodes are removed from the set of available nodes that forms the candidate list for the selection process.

In lines 1-5 of $SEL\_OP()$, Dijkstra's algorithm is applied to find the shortest path connecting the source and target, and the path obtained is stored in

$Path$. A queue of $Path$s, viz. $PathList$ is maintained that contains the list of $Paths$ obtained after each iteration and is initialized with the shortest path P.

Lines 1-5 of $Generation()$ denotes that in each iteration (i.e., in each run), $Generation$ function inserts a node into the $Path$ obtained after the previous iteration. Thus, an iteration of $Generation$ function accepts a list of $Path$s in the form of $PathList$ and for each $Path$ in the $PathList$, it calls the $Selection$ function. Another queue of $Path$s, namely $ChildPathList$ is maintained, which is initially empty. It stores the child paths generated using the $Selection$ function for each $Path$ of $PathList$. After each iteration, we consider only $PathListSize$ number of $Path$s at a time. Thus, $PathListSize$ number of $BestPath$s ($BestPath$ is the one which has the highest value for total collected score) from the $ChildPathList$ are inserted into another queue of $Path$s, viz. $NewPathList$. Then the termination condition is checked to determine whether a new child is generated by the $Generation$ function or not. If no new child is generated (i.e., all paths in the queue, $NewPathList$ is the same as that of $PathList$), then the $Generation$ function will return the $BestPath$ from the $PathList$ else, the $Generation$ function is recursively called for $NewPathList$. In the $Selection$ (), four selection techniques were implemented one by one. A comparison of their results has been presented in the next section. A detailed explanation of lines 1-10 of $Selection$ () has been stated in section 4.3.1 after the RWS_OP algorithm. The time complexity of SEL_OP is $O(|V|^3)$.

4.2.3 Experimental Analysis

Our code was implemented in C++ and compiled using CodeBlocks on an Intel Core i5 650 running at 2.20 GHz. SEL_OP has the capability to tackle both complete as well as incomplete graphs and here the results for incomplete graphs are reported by running the code on some real data. Two instances were considered viz. a real road network data of 160 and 306 cities of Poland. Each instance is associated with two files viz. cities.txt and distances.txt. The names of the various cities and their scores are specified in cities.txt. The score of each city is calculated on the basis of the number of inhabitants using the formula $score = inhabitants/10000$. The other file distances.txt represents for each

city, its adjacent city and their respective edge lengths. These edges of the graph correspond to the roads of the real map of Poland.

The algorithm SEL_OP was executed using four different selection methods defined in section 4.2.1. In Table 4.1, the maximum and mean score value achieved by each selection technique for different T_{max} values is presented and in Figure 4.1, these values have been plotted for the first instance with 160 cities. It was observed that roulette wheel selection method performs better than the other three techniques in most of the cases and helps in obtaining a better total collected score when compared to the other techniques.

It was also seen that when the same algorithm was executed for the 306 cities instance, only the random selection method suffers and rest of the selection methods achieves the same maximum total collected score most of the times as shown in Table 4.2 and Figure 4.2. The possible reason for this observation might be the existence of big clusters in the 306 cities instance which is absent in case of the 160 cities instance as shown in Figure 4.3 (a) and 4.3 (b). Therefore, the selection techniques have more possible options available to be explored in case of 160 cities instance and do not get stuck within a cluster. On this basis, it can be stated here that roulette wheel selection procedure performs better than the other methods when executed by SEL_OP algorithm.

Table 4.1: Comparison of the mean and maximum value of the total collected score obtained by SEL_OP when executed with four different selection procedures for 160 cities

T_{max}	(μ,λ)		Random		Tournament		Roulette Wheel	
	Max	**Mean**	**Max**	**Mean**	**Max**	**Mean**	**Max**	**Mean**
500	49	49	49	48	49	48	49	48
750	67	67	67	64	67	66	67	65
1000	83	83	88	77	88	84	88	79
1250	103	103	103	99	103	101	102	100
1500	118	118	116	114	119	116	116	114
1750	135	135	130	128	135	128	130	128
2000	145	145	145	143	145	143	148	143
2250	157	157	160	154	157	155	162	155
2500	179	179	184	175	179	176	187	174
2750	190	190	203	193	191	189	203	193
3000	206	206	214	205	206	202	214	206
3250	234	234	233	223	234	228	233	224
3500	248	248	249	242	249	245	251	244
3750	255	255	261	255	257	255	263	256
4000	269	269	271	264	272	268	270	267

Table 4.2: Comparison of the mean and maximum value of the total collected score obtained by *SEL_OP* when executed with four different selection procedures for 306 cities

T_{max}	(μ,λ)		Random		Tournament		Roulette Wheel	
	Max	**Mean**	**Max**	**Mean**	**Max**	**Mean**	**Max**	**Mean**
500	127	121	127	117	127	121	127	118
750	155	149	155	147	155	148	155	147
1000	178	173	177	168	178	172	178	169
1250	211	173	208	190	211	172	211	193
1500	236	213	234	218	236	219	236	221
1750	263	241	263	253	263	254	263	251
2000	282	280	279	273	282	277	282	273
2250	307	300	299	291	307	292	307	281
2500	325	323	323	307	325	309	325	306
2750	349	336	333	325	349	327	349	329
3000	366	355	366	346	366	345	366	344
3250	396	339	386	366	396	371	396	369
3500	420	394	414	384	420	393	420	390
3750	435	408	428	410	435	411	435	400
4000	459	430	448	429	459	437	459	440

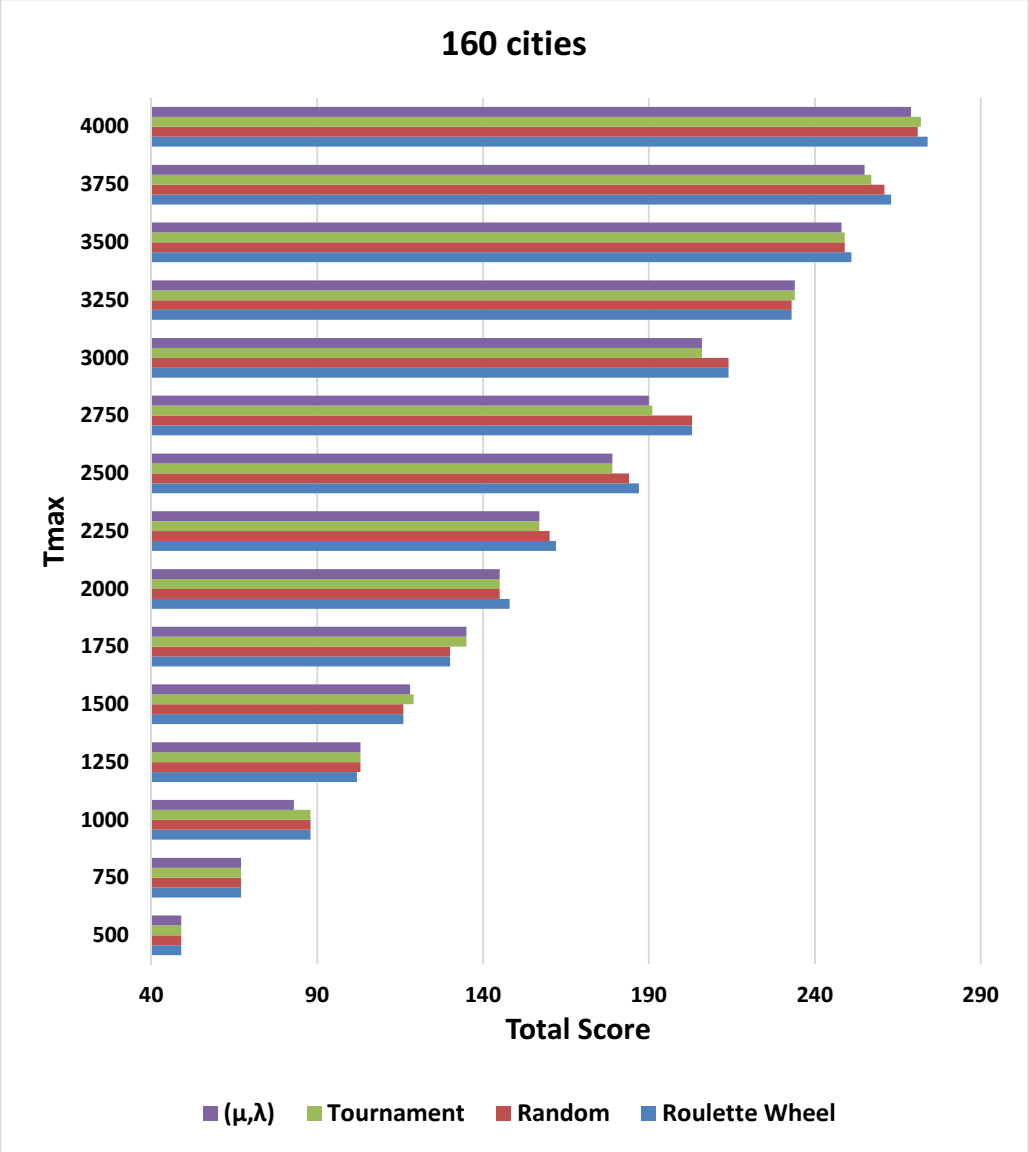


Fig. 4.1: Comparison of the maximum value of the total collected score obtained by four different selection methods for different T_{max} values (160 cities)

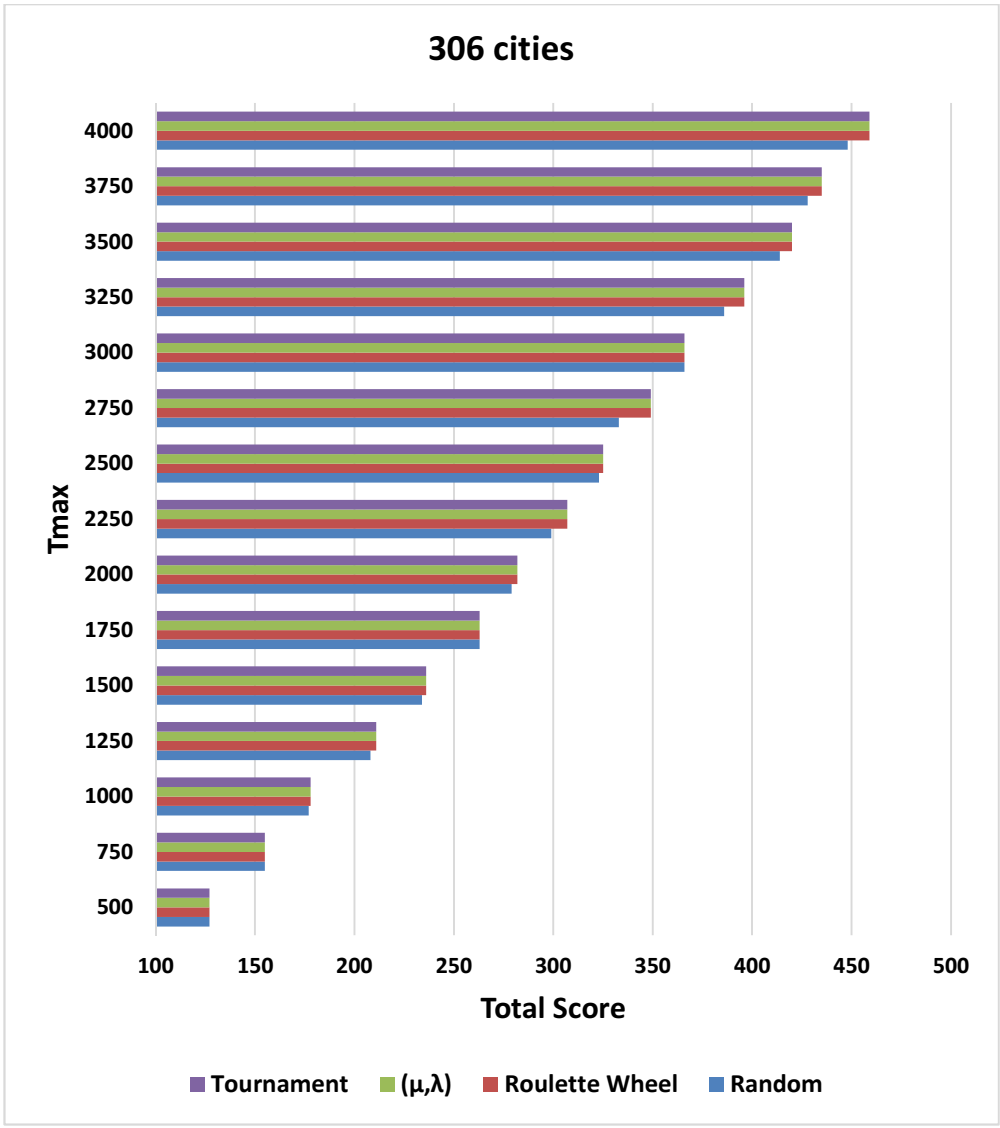
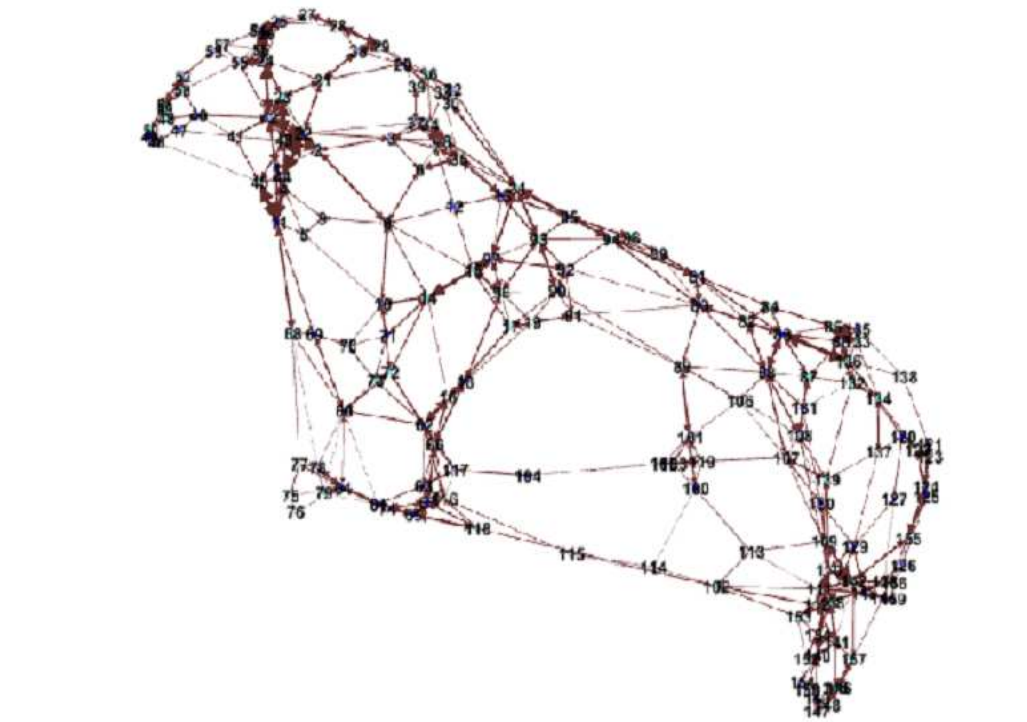
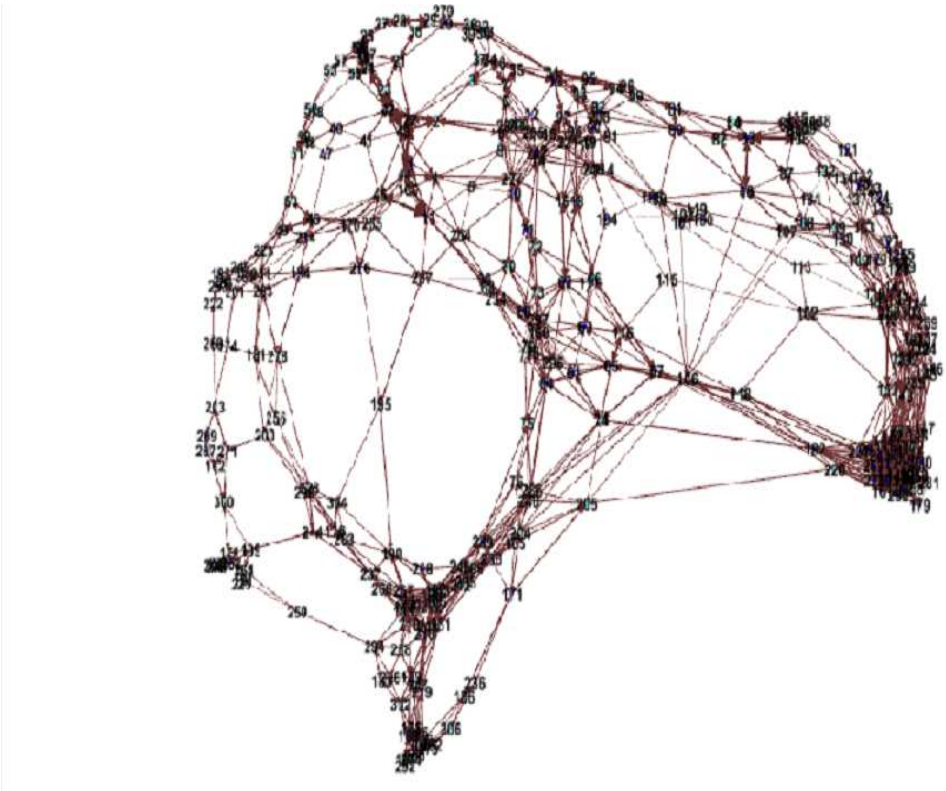


Fig. 4.2: Comparison of the maximum value of the total collected score obtained by four different selection methods for different T_{max} values (306 cities)



(a)



(b)

Fig. 4.3: Graph for (a) 160 cities and (b) 306 cities instance

4.3 Roulette Wheel Selection based Heuristic Algorithm for the Orienteering Problem

In section 4.2, it was shown through experiments that the roulette wheel selection method performs better than the other selection methods. Here, based on several experiments on standard benchmark data, we show that RWS_OP is faster, more efficient in terms of time budget utilization and achieves a better performance in terms of the total collected score as compared to a recently reported algorithm for incomplete graphs.

Here, we have proposed a stochastic greedy heuristic for OP (RWS_OP) that uses the roulette wheel selection method for determining the path that maximizes the total collected score within the specified time frame (T_{max}). The algorithm is guaranteed to reach the destination node (v_N) since the starting node (v_1) and the final node (v_N) are always the end points for all the generated paths. One necessary condition of OP is to ensure that a node is visited at most once, which in our algorithm is implemented by removing the explored nodes from the set of available nodes. This algorithm can be applied on both complete as well as incomplete graphs. We also compare our results with those reported by Ostrowski et al for large instances to show that RWS_OP executes more efficiently. Ostrowski *et al.* (2011) proposed two algorithms one for incomplete graphs (IG) and the other that requires conversion of IG to complete graphs (CG) before OP can be solved. Few drawbacks of their paper are: (1) The paper does not categorically provide conclusive evidence about which of these algorithms is better. (2) Further, the authors allow each vertex to be visited more than once, but the reward is collected only on the first visit. This strategy is not only disadvantageous from the time budget point of view, but also produces invalid results (i.e., Non-Hamiltonian path). (3) In the second version of their algorithm, virtual edges need to be added using Dijkstra's algorithm, which results in unnecessary complication. (4) Their strategy requires application-specific complex crossover and mutation operations that often produce infeasible partial solutions that require additional correction / repair operations. A stochastic greedy algorithm that uses roulette wheel selection method has been proposed here to avoid the search getting trapped in local maxima and removes all the disadvantages of Ostrowski's method mentioned above. It also

outperforms Ostrowski's algorithm by improving the score and utilizing the time budget up to almost 99%. The objective function used in RWS_OP is based on the greedy adaptive search procedure and path relinking (GRASP) technique suggested by Campos *et al.* (2013). Also, it has been shown that roulette wheel selection with the new instance coding technique outperforms full GA implementation by Ostrowski *et al*. in terms of both quality of solution and search time and space. In our method of candidate representation, we start with the shortest path between v_1 and v_N and using the roulette wheel proportionate selection scheme, keep on adding the nodes until T_{max} is reached. At each step, the probability of selecting a node is proportionate to its fitness defined in Eq. 4.9. Thus, without using crossover and mutation and without the need to maintain a population of possible solution, this algorithm is able to outperform GA based technique reported by Ostrowski *et al*. (2011).

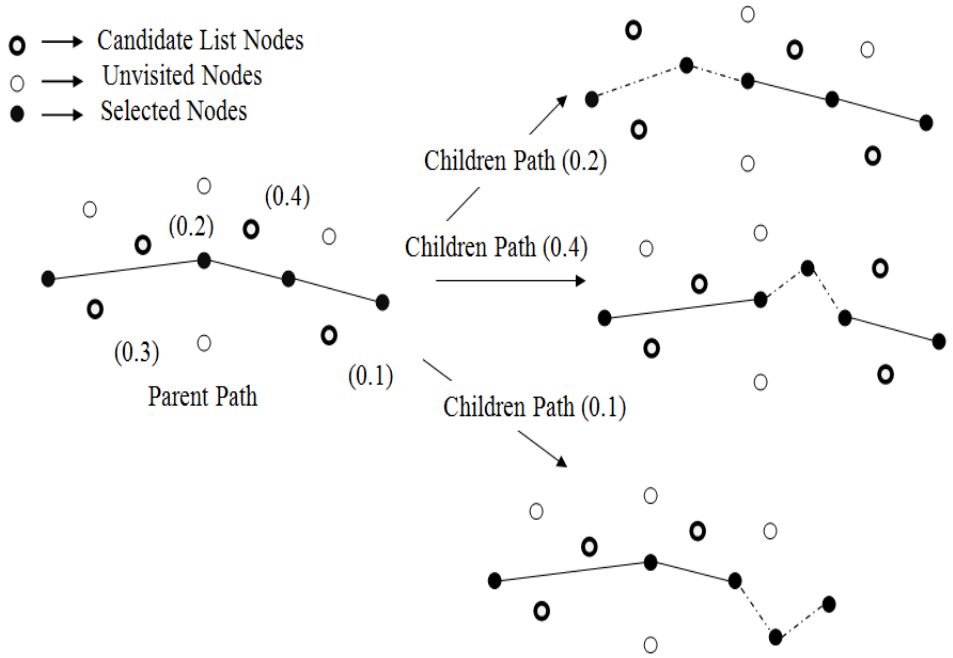


Fig. 4.4: The process of selecting a path using roulette wheel selection function where the number in () denotes the probability of node selection

4.3.1 Algorithm RWS_OP

Input: A graph $G(V, E)$ with t_{ij} (time taken to traverse) value of each edge (e_{ij}) connecting vertex v_i and $v_j \in V$, S_i (score) value of each vertex $v_i \in V$ and the $PathListSize$ (maximum number of paths considered at each level).

Output: A Hamiltonian path with the highest possible collected score such that total travel time is within the specified time budget.

$Path$ is an array of nodes or vertices, which is a sequence connecting the source and the target.

$PathList$, $NewPathList$ and $ChildPathList$, are all queues and each of its element is a $Path$. $PathListSize$ is a constant whose value defines the maximum number of paths to be considered after each iteration.

$RWS\_OP(G, PathListSize, T_{max})$

1. ***Create*** $PathList$;
   // Array of paths which is initially empty.
2. $PathList \leftarrow \emptyset$;
3. $Path$ P $\leftarrow Dijkstra(v_1, v_n)$;
   // Shortest path between source (v_1) and target (v_N).
4. ***Insert*** P *to* $PathList$;
5. ***return*** $Generation(PathList)$;

$Generation(PathList)$

1. ***Create*** $NewPathList$, $ChildpathList$; // Queues storing paths.
   $NewPathList \leftarrow \emptyset$;
   $ChildPathList \leftarrow \emptyset$;
2. ***for*** $i \leftarrow 0$ *to* $|PathList|$
   $a.Selection(PathList[i], ChildPathList)$;

// $ChildPathList$ will contain children generated from each path in $PathList$.

3. ***for*** $i \leftarrow 0$ *to* $PathListSize$
   // Selecting best $PathListSize$ children for next generation.
   $a.ChildPath = BestPath(ChildPathList)$;

// The path with the maximum total score.

$\boldsymbol{Remove}\ ChildPath\ from\ ChildPathList;$

$\boldsymbol{Insert}\ ChildPath\ to\ NewPathList;$

$b.\boldsymbol{if}\ |ChildPathList| == 0$

$break;$

4. $\boldsymbol{if}\ NewPathList == PathList$

// Terminate if no new child is generated and return the $BestPath$.

$\boldsymbol{return}\ BestPath(PathList);$

5. $\boldsymbol{return}\ Generation(NewPathList);$

$Selection(Path\ P, ChildPathList)$

//$Path$ is an array of nodes that forms a path.

1. $q_{max} \leftarrow 0;$
2. $\boldsymbol{for}\ i \leftarrow 0\ to\ |V|$
3. $a.\boldsymbol{if}\ i \in P$ // If a node is already present in the path then ignore it.

$\boldsymbol{continue};$

$b.\boldsymbol{Calculate}\Delta t_i\ ;$

// The time increment due to insertion of v_i at its best position.

$$c.\ q_i = \begin{cases} S_i/|\Delta t_i|, & \Delta t_i \geq 1 \\ S_i, & -1 \leq \Delta t_i < 1 \\ S_i * |\Delta t_i|, & \Delta t_i < -1 \end{cases};$$

$d.\boldsymbol{if}\ q_{max} \leq q_i$

$q_{max} \leftarrow q_i$

4. $Create\ CandidateList\ ;$

// Array of candidate nodes used for roulette wheel selection.

5. $CandidateList \leftarrow \emptyset;$
6. $\boldsymbol{for}\ i \leftarrow 0\ to\ |V|$
7. $a.\boldsymbol{if}\ i \in P$

continue;

$b.\boldsymbol{if}\ (q_i \geq \alpha\, q_{max})\&\&\ (t_{parent} + \Delta t_i \leq T_{max})$

// α is the greediness parameter that decides which node should participate in roulette wheel selection.

$c.\boldsymbol{Insert}\ v_i\ to\ CandidateList$;

8. $\boldsymbol{if}|CandidateList| == 0$

$\boldsymbol{Insert}\ P\ to\ ChildPathList$;

// If no new nodes are added then insert the parent path P.

Return;

9. $\boldsymbol{for}\ i \leftarrow 0\ to\ i \leftarrow PathListSize$

// Generates $PathListSize$ number of children paths from path P.

10. $a.ChildNode \leftarrow RouleteWheel(CandidateList)$

$\boldsymbol{Remove}\ ChildNode\ from\ CandidateList$;

$\boldsymbol{Insert}\ ChildNode\ to\ Path\ P\ and\ \boldsymbol{Insert}\ Path\ P\ to\ ChildPathList.$

$\boldsymbol{Remove}\ ChildNode\ From\ Path\ P$;

$b.\boldsymbol{if}|CandidateList| \leq 0$

break;

$RouleteWheel(CandidateList)$

1. $sum \leftarrow 0$;

2. $\boldsymbol{for}\ i \leftarrow 0\ to\ |CandidateList|$

$sum+= fitness(i)$;

$//fitness(i) \leftarrow q_i$, where $q_i = \begin{cases} S_i/|\Delta t_i|, & \Delta t_i \geq 1 \\ S_i, & -1 \leq \Delta t_i < 1 \\ S_i * |\Delta t_i|, & \Delta t_i < -1 \end{cases}$

3. $RandomNumber \leftarrow Rand()\%(sum + 1)$

4. $PartialSum \leftarrow 0$;

5. $Selected \leftarrow 0$;

6. $\boldsymbol{while}(RandomNumber > PartialSum)$

$a.Selected++$;

$b.PartialSum+= fitness(selected)$;.

7. $\boldsymbol{return}\ selected$;

$BestPath(PathList)$

1. $max \leftarrow 0; bestpath \leftarrow 0;$
2. $\boldsymbol{for}\ i \leftarrow 0\ to\ |PathList|$

   a. $\boldsymbol{if}\ \Big(Score\big(PathList(i)\big)\Big) > maxScore$

   // *Score* ($Path\ P$) is the sum of the rewards associated with each node of $Path\ P$.

   $b. maxScore \leftarrow Score\big(PathList(i)\big); bestpath\ \leftarrow i;$
3. $return\ PathList(bestpath);$

The $RWS\_OP()$ and $Generation()$ functions work in a similar manner as explained in section 4.2.2 above. Here, the $Selection()$ and the $RouletteWheel()$ functions have been explained in detail.

The $Selection$ function is used for generating $ChildPaths$ from $Path\ P$. This function generates a $ChildPath$ which contains one node more than that already present in $Path$ P. This function decides the node to be inserted and its location in $Path\ P$. As shown in line 3 of $Selection()$, for each vertex that has not yet been inserted in $Path\ P$, its best position (one that leads to smallest increment in time Δt_i) is evaluated and then the ratio q_i is computed. The ratio q_{max} is calculated in the following way:

$$(q_{max}) = max_{i \in (V \setminus P)}(q_i)$$

In lines 4-7 of $Selection()$, another list of nodes, named $CandidateList$ is created which is initially empty. All those nodes which satisfy the following inequality are inserted in the $CandidateList$:

$$CandidateList = \{i \in (V \setminus P) : (q_i \geq \alpha\, q_{\max})\ \&\&\ (t_{parent} + \Delta t_i \leq T_{max})\}$$

Here α is the greediness parameter. Greater value of α denotes a more greedy solution and smaller value of α denotes a more random solution. As the value of α decreases, more nodes will be selected for the $CandidateList$. As stated in lines 8-10 of $Selection()$, if the $CandidateList$ is empty, $Path\ P$ is inserted in the queue $ChildPathList$, else $RouleteWheel$ selection method is used to choose a node from the $CandidateList$. The chosen node is removed from the $CandidateList$ and inserted at its best position in $Path\ P$ to generate a $ChildPath$ which is then en-queued into $ChildPathList$. This process of extracting nodes from the $CandidateList$ is repeated until a constant

($PathListSize$) number of $ChildPaths$ are generated or $CandidateList$ becomes empty.

Roulette wheel selection is a population-based selection method used in genetic algorithms that stochastically picks out a node based on their fitness value i.e., a node having greater fitness value has more chances of getting selected, although nodes with lower fitness also have a nonzero probability of selection. This assists the search in escaping local maxima. It is conceptually equal to giving each individual option, a portion of a circular roulette wheel proportional in area to the individual's fitness value (Zhang et al., 2012). As it can be seen, lines 1-7 of the $RouleteWheel()$ selects a node from the $CandidateList$ using $FitnessProportionateSelection$ approach where an element is randomly chosen, but the probability of choosing an element with higher fitness is greater than choosing an element with lower fitness value. In our selection process, the fitness of an element of $CandidateList$ is computed using the following equation:

$$fitness(i) \leftarrow q_i, \text{ where } q_i = \begin{cases} S_i/|\Delta t_i|, & \Delta t_i \geq 1 \\ S_i, & -1 \leq \Delta t_i < 1 \\ S_i * |\Delta t_i|, & \Delta t_i < -1 \end{cases} \quad (4.9)$$

Therefore, nodes having a higher value of S_i and lower value of time increment (Δt), have a greater probability of getting selected and also the nodes having low S_i and high (Δt) values can be selected but with a lesser probability (> 0). The time complexity of $Dijkstra's\ (v_1, v_N)$ algorithm is $O(|V|^2)$ where $|V|$ is the number of vertices in the graph. In the $RouleteWheel$ function, there exists an iterative loop that runs $|CandidateList|$ times and because the $CandidateList$ can contain at most $(|V| - 2)$ nodes, therefore the time complexity of $RouleteWheel$ will be $O(|V|)$.

In the selection function, line 2 is an iterative loop running for all the vertices of G i.e., $|V|$ times and in each iteration, the best position of node i in the current $Path\ P$ is found which takes $O(|V|)$ time. Line 6 represents another iterative loop, which takes $O(|V|)$ time as it also runs for all the vertices of G. $Insert$ operation in the $CandidateList$ is similar to simple insertion in an array and takes $O(1)$ time. $Remove$ operation in $CandidateList$ will take $O(|CandidateList|)$ or $O(|V|)$ time. $Remove$ operation in $ChildPathList$ takes $O(|V|)$ time as $O(|V|)$ time is required to bring the element to the front of

the $ChildPathList$ queue and $O(1)$ to dequeue it. $Insert$ operation in $Path\ P$ takes $O(|V|)$ time. The iteration of line 9 takes $O(PathListSize * |V|)$ time as it runs for $PathListSize$ times, and each iteration takes $O(|V|)$ time. Hence, the overall time complexity of $Selection$ function is $\big(O(|V|^2) + O(|V|) + O(PathListSize * |V|)\big)$, which is equivalent to $O(|V|^2)$ as $PathListSize \leq |V|$.

For the $Generation$ function, time complexity of line 2 which is an iterative loop is $O(PathListSize * |V|^2)$ as it runs for all the paths in $PathList$ and there can be at most $PathListSize\ Paths$ in this queue. In each iteration, it calls the $Selection$ function that takes $O(|V|^2)$ time. The $BestPath$ function will take $O(PathListSize)$ time. Since it is a recursive function, after each recursion, the $Paths$ contained in $PathList$ will increase by 1. The function will terminate when no new node is available that can be inserted in the $Path$. Therefore, the recurrence relation can be written as:

$$T(M) = T(M-1) + PathListSize * |V|^2$$

Here, $T(M)$ is the time complexity for the $Generation$ function where, M is the number of nodes that can be added to the $Paths$ contained in $PathList$ and after one iteration, the function calls itself with $M-1$ number of nodes that can be added to the $Paths$ of $NewPathList$. Solving the above recurrence, we get the overall complexity as $O(|V|^3)$.

The main function, which is RWS_OP, uses the Dijkstra's algorithm ($O(|V|^2)$ time), $Selection$ function ($O(|V|^2)$ time) and $Generation$ function ($O(|V|^3)$ time). Therefore, the overall time complexity of RWS_OP is $O(|V|^3)$. The memory consumption of RWS_OP is $|PathListSize|^2 * N$ as opposed to Ostrowski_CG and Ostrowski_IG methods that occupies $PopulationSize * N$ memory where $N = number\ of\ nodes\ in\ the\ graph$.

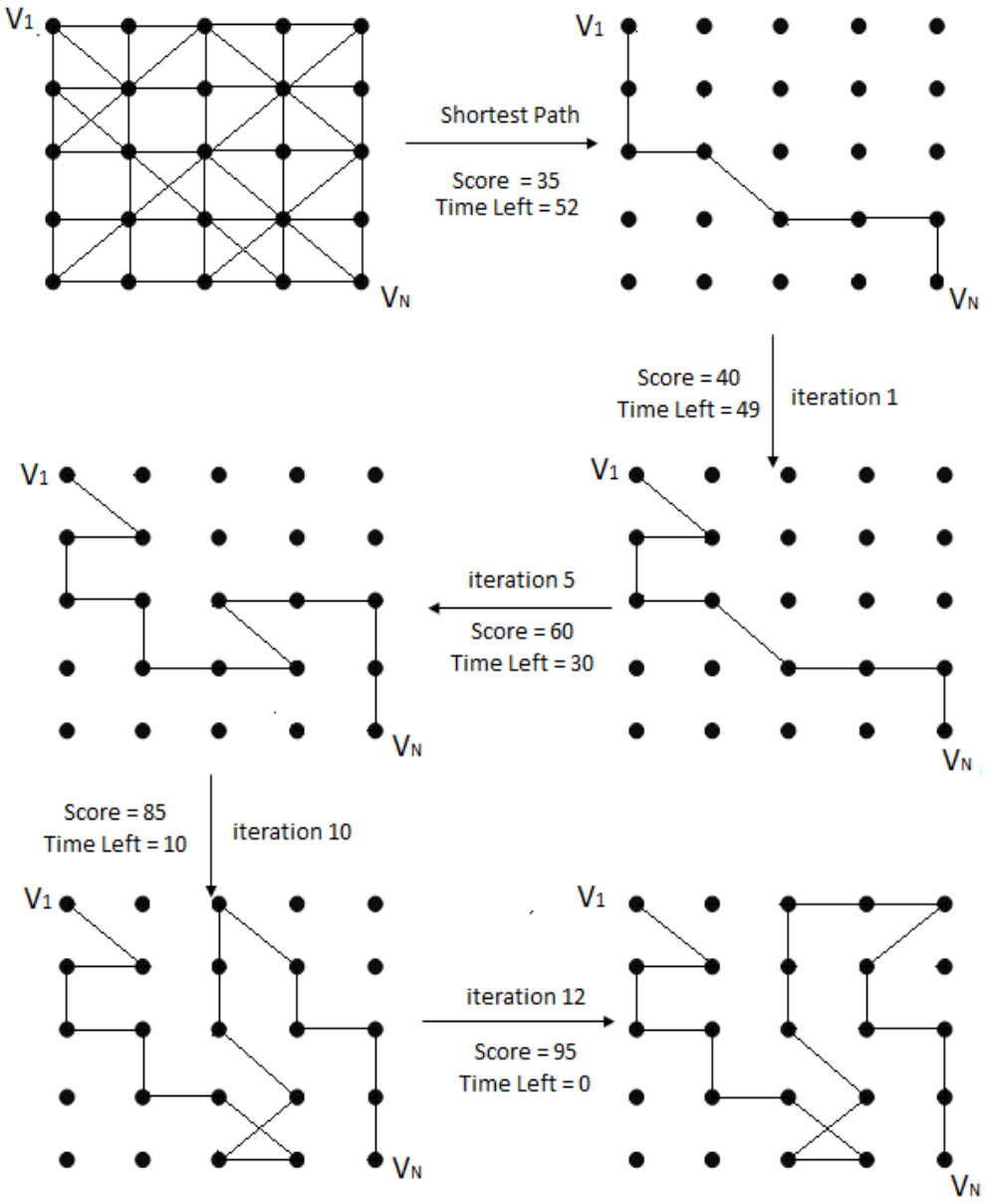


Fig. 4.5: Progression of RWS_OP algorithm for a graph with 25 nodes with source $(V_1) = 1$, destination $(V_N) = 25$ and $T_{max} = 70$

4.3.2 Experimental Analysis

Most of the standard benchmark instances available for the orienteering problem include Set 1, Set 2 and Set 3 given by Tsiligirides, Set 64 and Set 66 given by Chao etc. (www.mech.kuleuven.be/en/cib/op/). In each of the available instances, only scores and coordinates of each vertex are specified leading to the

formation of complete graphs that satisfy the triangle inequality. However, in the real world, there is no guarantee that two nodes will be connected through a direct path. To deal with such cases, graphs are usually complemented with fictional edges by running the Dijkstra's algorithm for every node, before applying the classic OP algorithms available for complete graphs. This results in a considerable increase in the search space size. Our method works on both, complete as well as incomplete graphs without the need to insert fictional edges.

We have implemented our code in C++ using CodeBlocks on an Intel Core i5 650 at 2.20 GHz. As stated before, RWS_OP is capable of handling both complete as well as incomplete graphs, and here we present a few observations by applying RWS_OP on a real road network data (same as SEL_OP) based on 160 and 306 cities of Poland (http://piwonska.pl/p/research/).

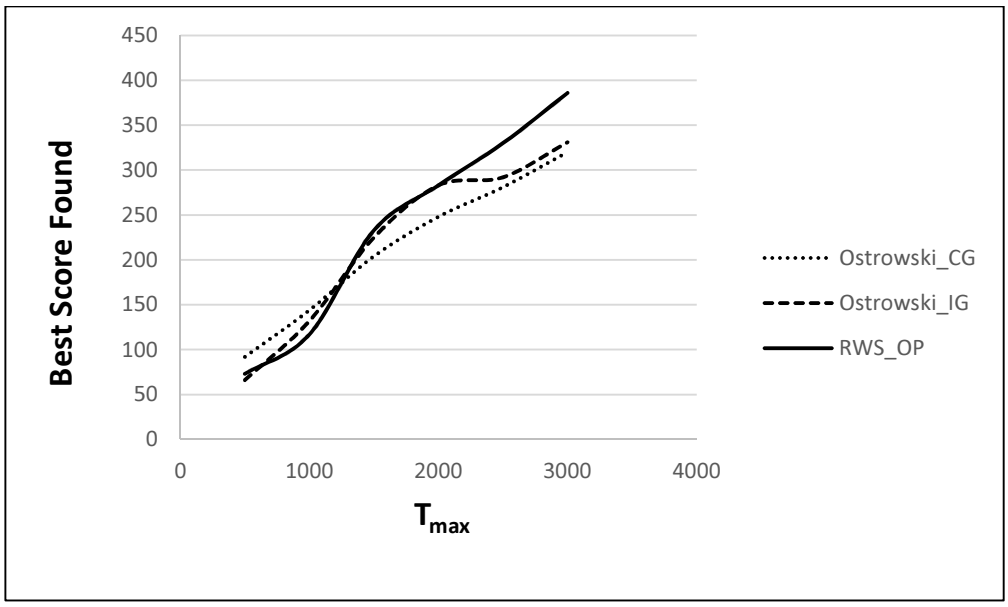
The following tables and plots show a comparison of our results with those obtained by applying the genetic algorithm suggested by Ostrowski et al on the same instances (Ostrowski and Koszelew, 2011). As can be seen in Table 4.3 and Fig. 4.6, it has been observed that RWS_OP performs better for larger T_{max} values (taking the same first and last node i.e., source = destination) than the one proposed by Ostrowski *et al*. because the highest total collected score attained for a graph with 306 nodes is greater in our case. Further, the mean score and 95% confidence interval (CI) of mean for 30 runs $\left(i.e\ CI = \frac{1.96 * standard\ deviation\ of\ the\ number\ of\ runs}{\sqrt{number\ of\ runs}}\right)$ are higher in our case when compared to the values obtained by the genetic algorithm. Fig. 4.7 shows that for the same T_{max} value, there is a significant decrease in the execution time of RWS_OP as compared to the CG (Complete Graph version) and IG (Incomplete Graph version) of the genetic algorithm proposed by Ostrowski *et al*. (2011). It has also been observed, that the execution time increases linearly with the increase in T_{max} because increase in T_{max} leads to the exploration of more number of nodes. Fig. 4.8 presents the effect on scores, by varying the value of the greediness parameter (α), which balances the degree of randomness and greediness. Increasing the value of the greediness parameter (α), makes the algorithm more greedy. Therefore, even after several executions of the code, most of the times, the same value is obtained for score, i.e., lesser variation in

the score. However, decreasing the value of (α) increases the randomness, i.e., more variations are observed in the scores obtained, as shown through smaller and larger boxes respectively. Also, it has been observed that the maximum score achieved increases with the increase in (α). However, for large number of executions, a greater maximum score can be obtained even for smaller values of (α). RWS_OP is also efficient in terms of the time utilization (as shown in Table 4.4 and Table 4.5) as almost 99% of the specified time budget (T_{max}) is utilized. This helps in determining a better path that covers almost 70% of the cities, thus leading to a greater total collected score. As the algorithm progresses, one node is added to the final path after each iteration. This leads to an increase in the total collected score and decrease in the given time budget as shown in Fig. 4.9. Most of the experiments are performed at $\alpha = 0.6$ because at 0.6, both randomness and greediness come into play and as stated before, decreasing the value of α makes the algorithm more random and increasing it makes RWS_OP greedier. RWS_OP uses roulette wheel selection for choosing a node to be added in the path. Thus, different runs of the same algorithm, with the same input parameters may result in selection of different nodes for being added in the final path. The trend in the utilization of time budget and an increase in the total collected score, for three different runs at $\alpha = 0.2$ and $T_{max} = 1500$ (for 160 cities instance) has been shown in Fig. 4.10. Fig. 4.11 shows how the time budget is utilized, and the total collected score increases, with each iteration of RWS_OP (for different values of α) at $T_{max} = 1500$. The proposed heuristic is capable of exploring almost 70% of the search space as shown in Fig. 4.12 (a). In Fig. 4.12 (b), a box plot has been presented that shows the percentage of nodes explored and unexplored at $T_{max} = 7000$ for different values of α. As can be observed in 4.12 (b), the percentage of nodes explored (NE) is higher than the percentage of nodes unexplored (NU). Further, the percentage of score collected for the explored nodes is higher than the percentage of score left out (score that could not be collected) for lower values of α. This happens because, at lower values of α more randomness is induced into RWS_OP. RWS_OP is efficient both in terms of time and space complexity when compared to Ostrowski_CG and Ostrowski_IG methods. Therefore, RWS_OP can be implemented for larger values of T_{max} which helps in

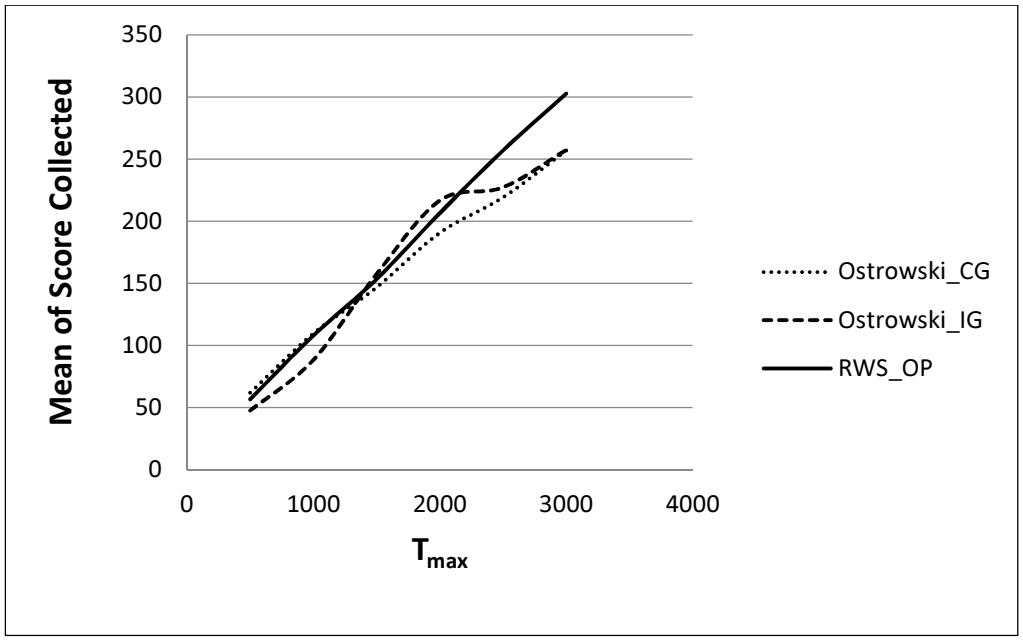
achieving a higher total collected score for the considered instances as shown in Table 4.6 and Fig. 4.13.

Table 4.3: Comparison of maximum, mean and confidence Interval (CI) for mean of scores obtained by RWS_OP (keeping $v_1 = v_N$ i.e., $v_1 = v_N = 1$) with those obtained by executing the Ostrowski's algorithm (Please refer (Ostrowski & Koszelew, 2011), their Table 5 for Ostrowski_CG and Table 7 for Ostrowski_IG) on Real Road Network database with 306 cities of Poland

T_{max}	RWS_OP(α=0.6)			Ostrowski_CG (highest fitness/travelTime)			Ostrowski_IG (fitnessGain^2/travelTimeIncrease)		
	Mean	CI for Mean	Maximum	Mean	CI for Mean	Maximum	Mean	CI for Mean	Maximum
500	56.56	±3.5	73	61.9	±3.5	92	47.5	±3.3	66
1000	107.9	±2.64	117	109.5	±5.4	144	88.3	±6.0	132
1500	153.3	±9.29	233	146.7	±8.8	204	157.4	±13.5	225
2000	206.5	±13.1	283	190.6	±9.7	248	216.6	±12.8	283
2500	256.9	±13.07	330	219.1	±10.7	281	227.2	±13.9	292
3000	302.7	±13.4	386	256.9	±11.4	320	257.3	±15.2	331



(a)



(b)

Fig. 4.6: Comparison of (a) maximum score and (b) mean score of each method with respect to time budget (T_{max})

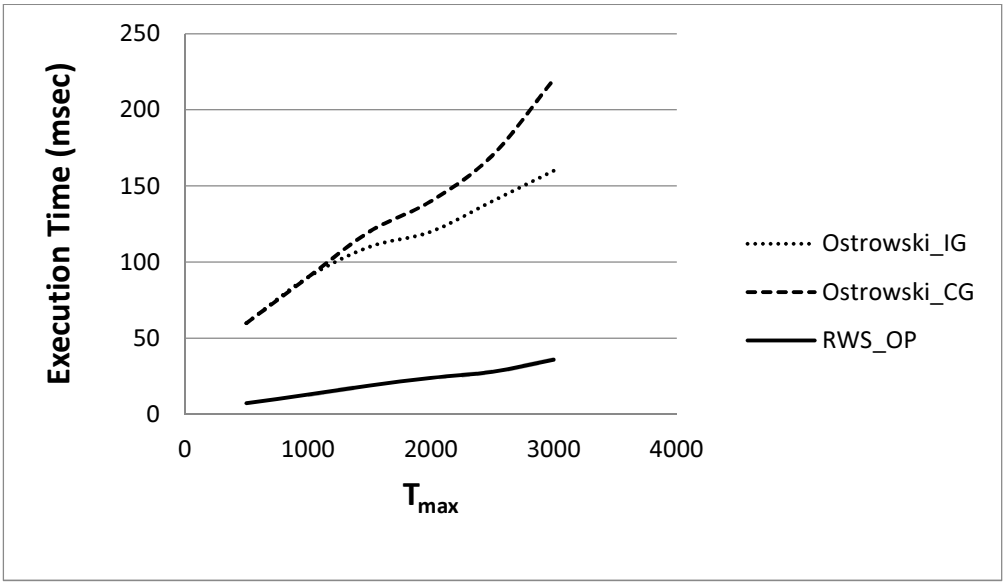
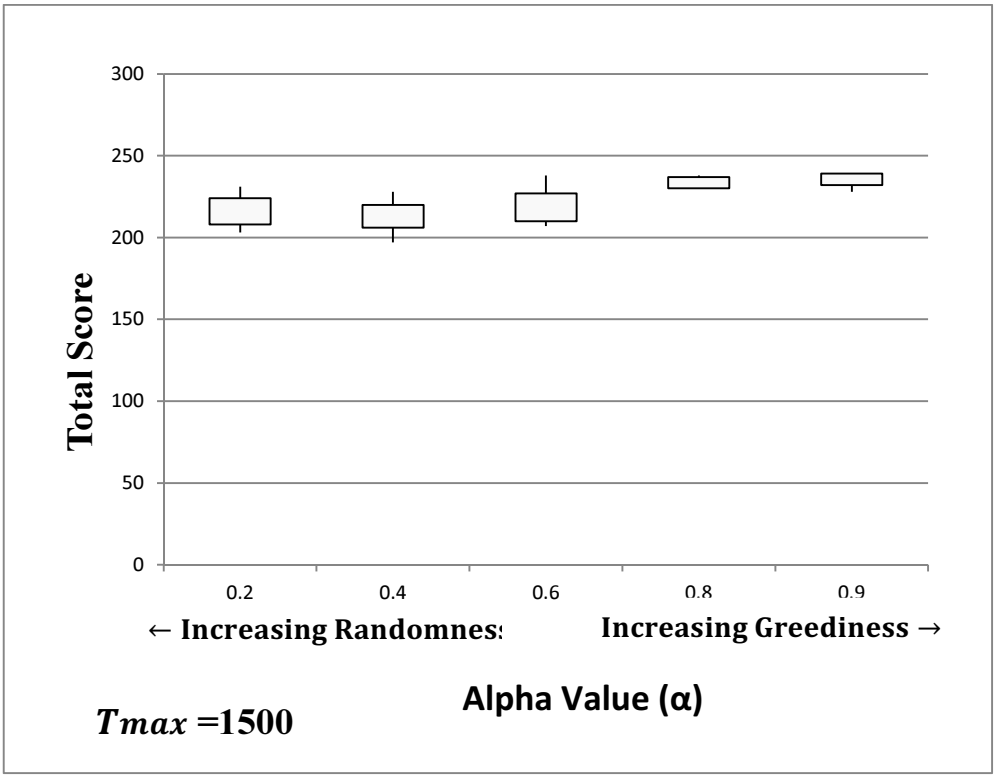
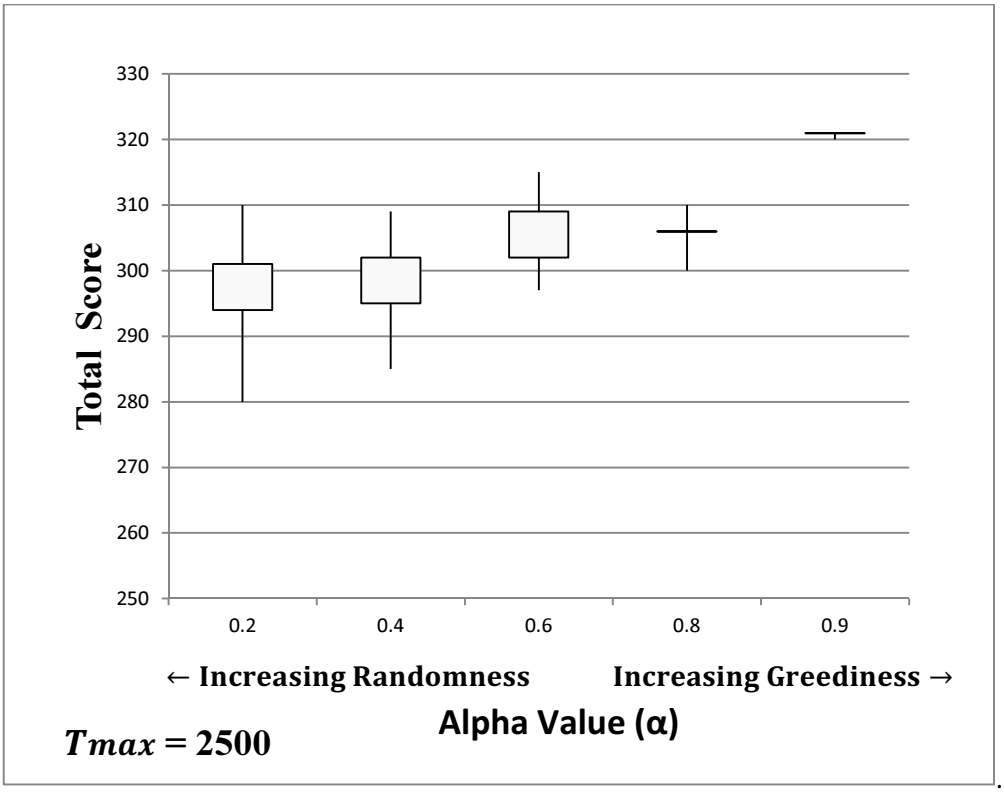


Fig. 4.7: Comparison of execution time of each method with respect to time budget (T_{max}) based on 30 runs at $\alpha = 0.6$ for Real Road Network database with 306 cities of Poland



(a)



(b)

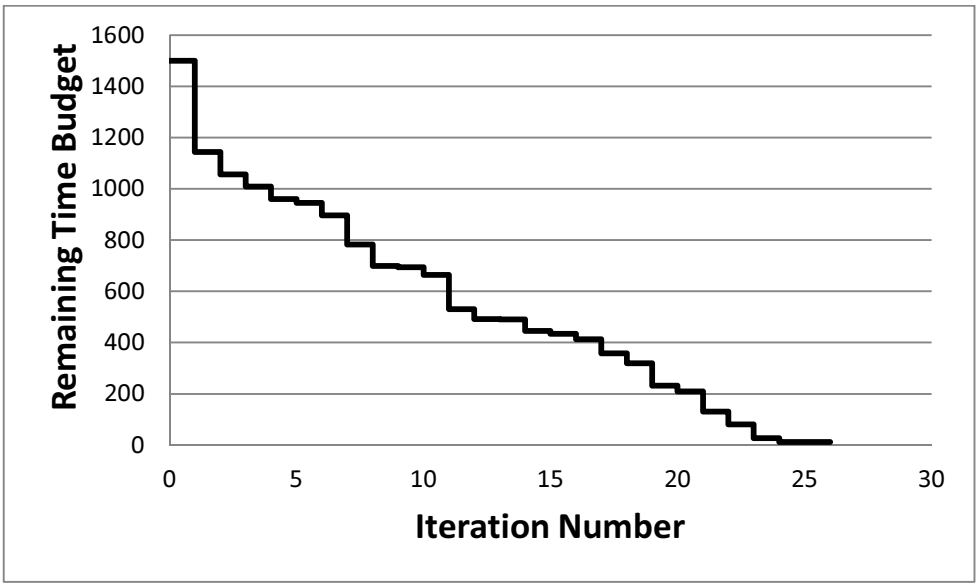
Fig. 4.8: Comparison of score with respect to α for (a) $T_{max} = 1500$ and (b) $T_{max} = 2500$ for a Real Road Network database with 306 cities of Poland

Table 4.4: The Highest Score Collected, Mean of Score Collected, Mean Time to Traverse the Path and % of Time Budget Utilized values obtained by RWS_OP at $\alpha = 0.6$ (keeping $v_1 \neq v_N$ i.e., $v_1 = 1$ and $v_N = 306$) when implemented on a Real Road Network database with 306 cities of Poland

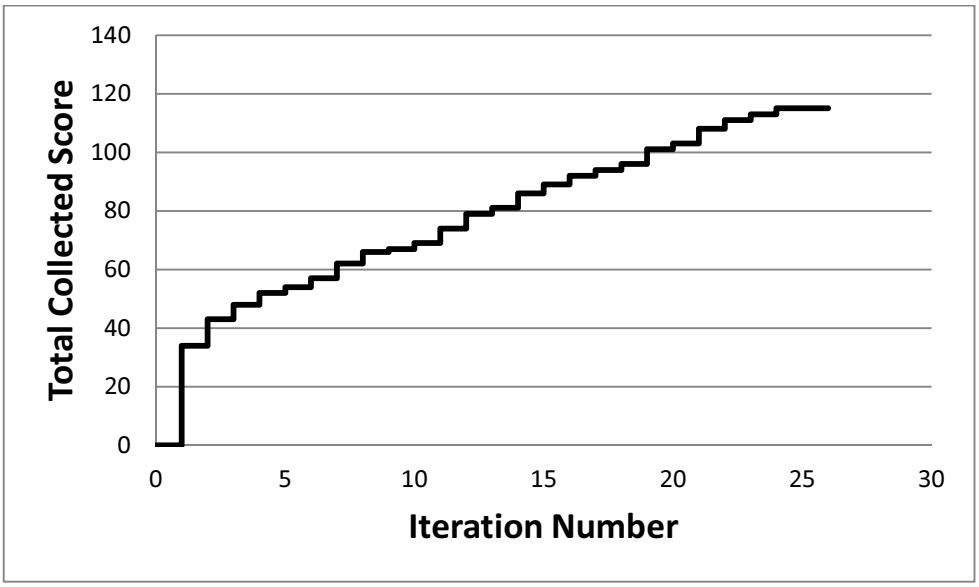
T_{max}	306 cities (α=0.6)			
	Highest Score Collected	Mean of Score Collected	Mean Time to Traverse the Path	% of Time Budget Utilized
500	122	120.1	495.1	99.02
750	154	150.3	743.4	99.12
1000	177	174.3	995.1	99.51
1250	210	195.1	1246.53	99.72
1500	243	220.4	1495.33	99.69
1750	261	243.6	1742.72	99.58
2000	286	270	1993.4	99.67
2250	310	291	2244.8	99.77
2500	324	312.4	2493.76	99.75
2750	345	332.5	2744.8	99.81
3000	371	349.6	2993.1	99.77
3500	411	392.1	3495.47	99.87
4000	451	436.9	3995	99.88
4500	502	487.1	4494.3	99.87
5000	537	524.5	4995.2	99.90
5500	574	561	5491.7	99.85
6000	620	593.3	5990.7	99.85

Table 4.5: The Highest Score Collected, Mean of Score Collected, Mean Time to Traverse the Path and % of Time Budget Utilized values obtained by RWS_OP at $\alpha = 0.6$ (keeping $v_1 \neq v_N$ i.e., $v_1 = 1 \; and \; v_N = 160$) when implemented on a Real Road Network database with 160 cities of Poland

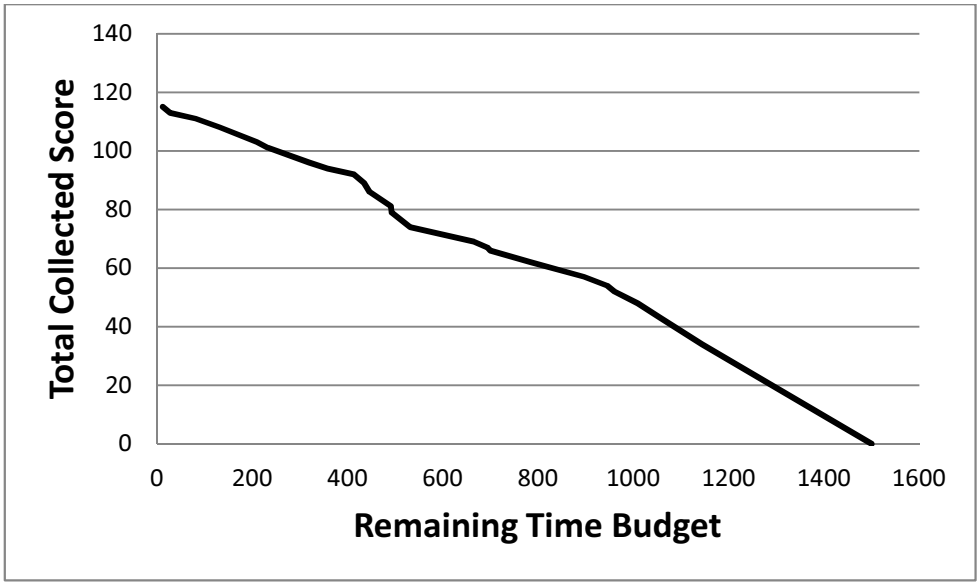
T_{max}	160 cities (α=0.6)			
	Highest Score Collected	Mean of Score Collected	Mean Time to Traverse the Path	% of Time Budget Utilized
500	49	48.83	496.67	99.34
750	67	65.1	747.07	99.6
1000	88	76.8	990.8	99.08
1250	104	102.3	1230.3	98.42
1500	117	115	1488.37	99.25
1750	129	127.9	1739.1	99.38
2000	145	142.6	1993.63	99.68
2250	162	156.1	2242.4	99.66
2500	185	176.4	2485.8	99.43
2750	202	193.2	2736.4	99.5
3000	214	205.1	2980.3	99.34
3500	250	243.6	3488.9	99.68
4000	271	265.1	3982.7	99.57
4500	306	290.23	4479.4	99.54
5000	314	307.53	4960.9	99.22
5500	322	316.9	5319.9	96.73
6000	322	314.7	5336	88.93


Remaining Time Budget
1600
1400
1200
1000
800
600
400
200
0
0
5
10
15
20
25
30
Iteration Number

(a)

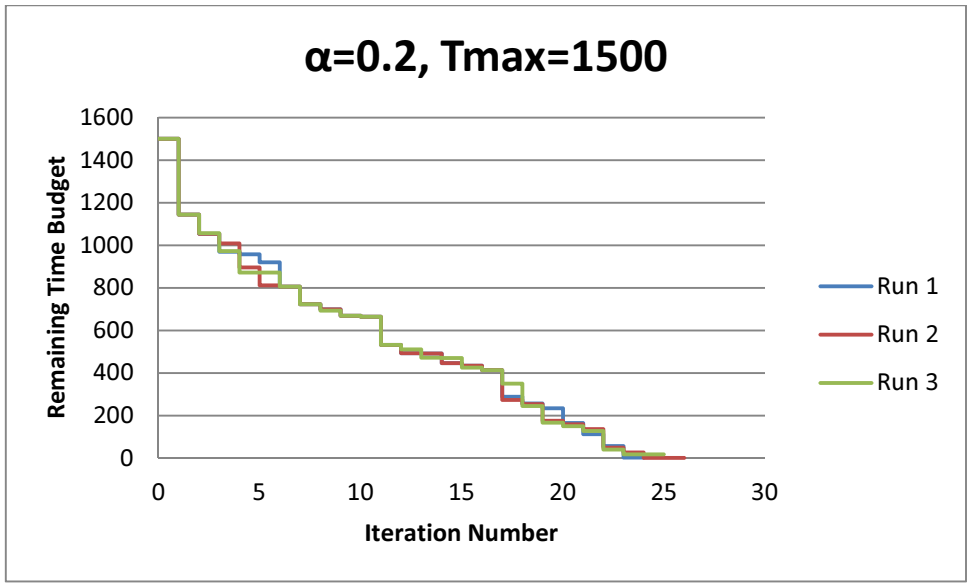

Total Collected Score
140
120
100
80
60
40
20
0
0
5
10
15
20
25
30
Iteration Number

(b)

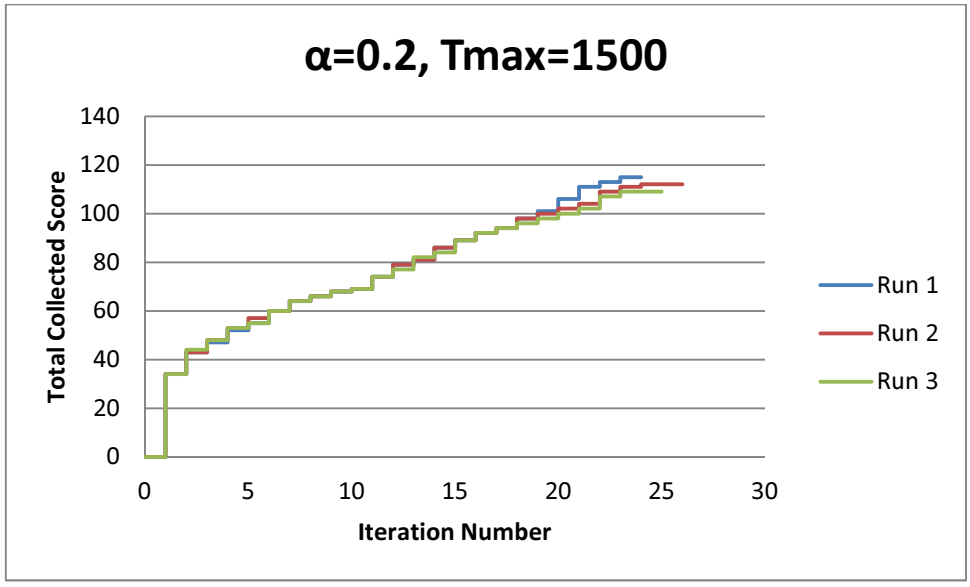


(c)

Fig. 4.9: Plots showing (a) utilization of the time budget and (b) increase in the total collected score at $\alpha = 0.6$ and $T_{max} = 1500$ for a Real Road Network database with 160 cities of Poland (c) Progression of RWS_OP algorithm

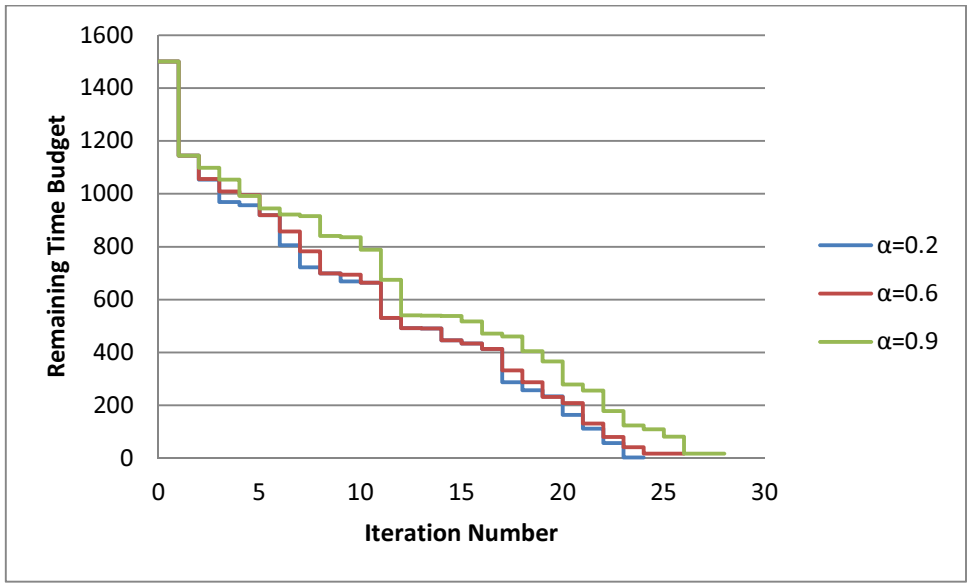


(a)

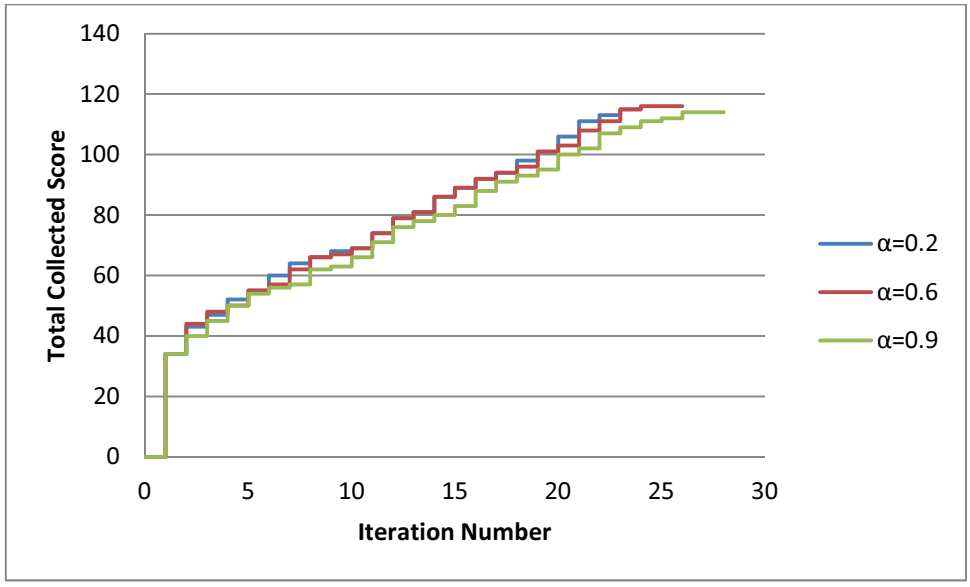


(b)

Fig. 4.10: Plots showing the observation of three different runs of RWS_OP with $\alpha = 0.2$ and $T_{max} = 1500$ for a Real Road Network database with 160 cities of Poland. As the algorithm progresses, it results in (a) decrease in the time budget and (b) increase in the total collected score as shown above

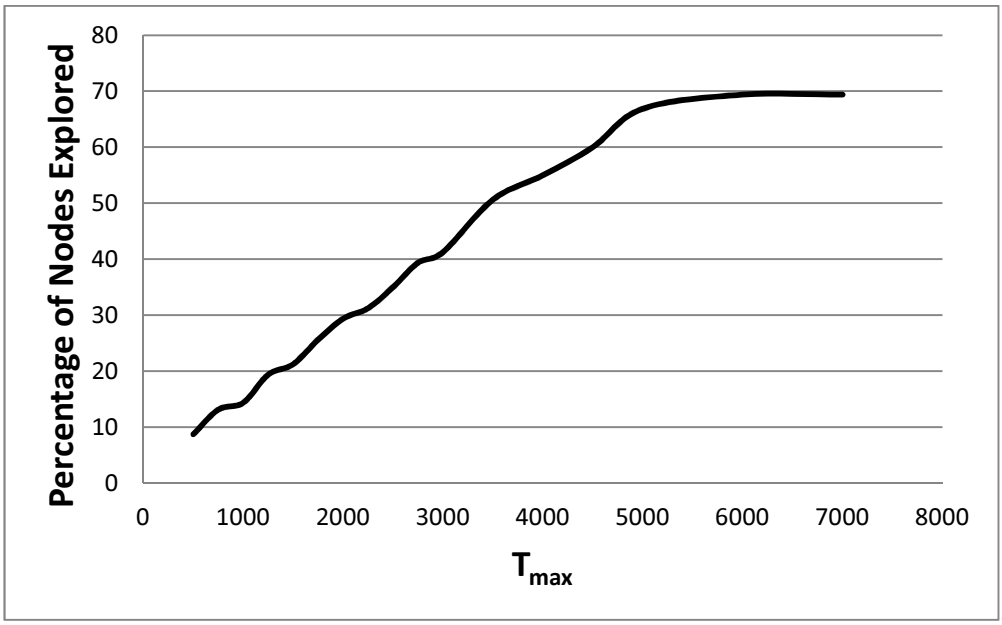


(a)

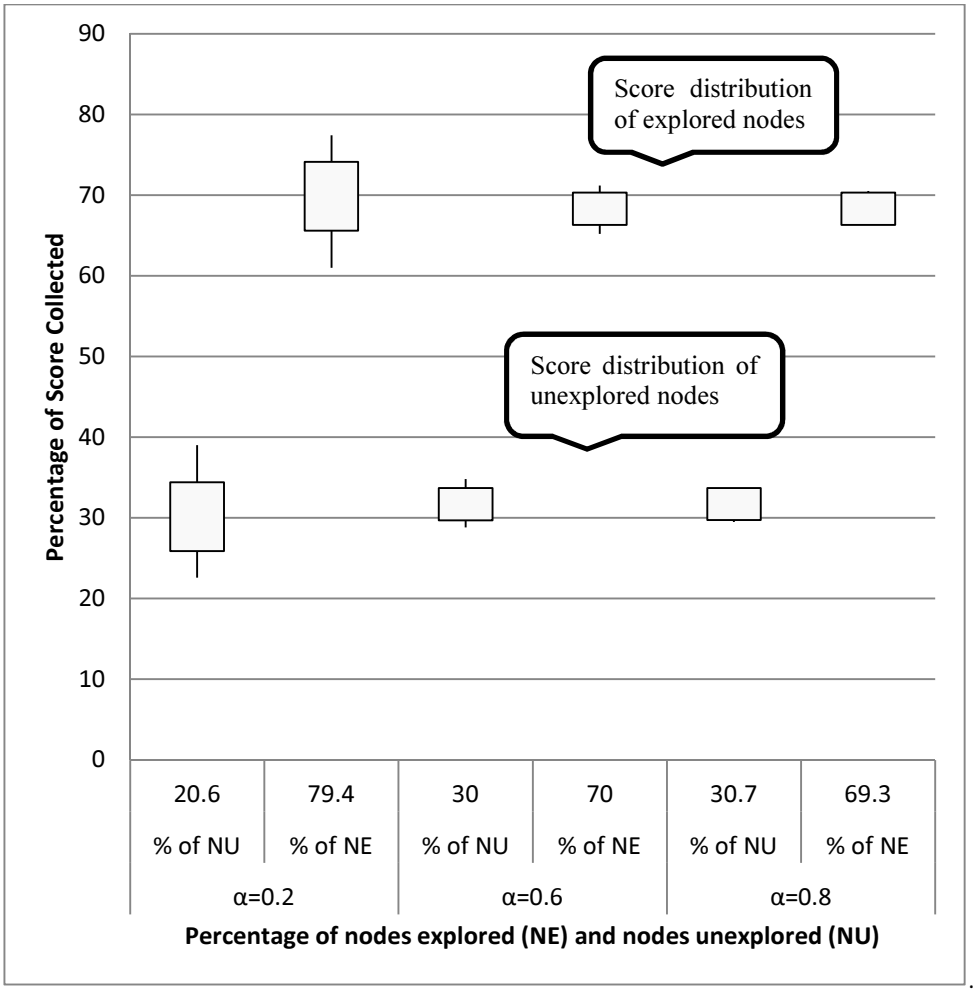


(b)

Fig. 4.11: Plots showing (a) utilization of the time budget and (b) increase in the total collected score for three different α values at $T_{max} = 1500$ for a Real Road Network database with 160 cities of Poland



(a)



(b)

Fig. 4.12: Plots showing (a) the percentage of nodes explored with the increase in T_{max} values at $\alpha = 0.6$ and (b) percentage of nodes explored and unexplored for different values of α at $T_{max} = 7000$ for a Real Road Network database with 160 cities of Poland for 30 runs. Experiments clearly show the effectiveness of the RWS_OP algorithm

Table 4.6: The Highest Score Collected, Mean of Score Collected and confidence interval (CI) for Mean of Score Collected obtained by RWS_OP when implemented on a Real Road Network database with 306 cities of Poland for different T_{max} values at $\alpha = 0.6$ (keeping $v_1 = v_N$ i.e., $v_1 = v_N = 1$)

T_{max}	RWS_OP(α=0.6)		
	Mean of Score Collected	CI for Mean of Score Collected	Highest Score Collected
500	56.56	±3.5	73
1000	107.9	±2.64	117
1500	153.3	±9.29	233
2000	206.5	±13.1	283
2500	256.9	±13.07	330
3000	302.7	±13.4	386
3500	353.16	±14.9	430
4000	427.8	±12.54	460
4500	466.7	±13.5	508
5000	506.1	±12.2	548
5500	553.2	±11.12	593
6000	595.2	±6.4	645
7000	653.1	±6.64	686
8000	718.2	±4.23	743
9000	767.8	±4.75	784
10000	769	±8.04	792
11000	769	±7.33	793

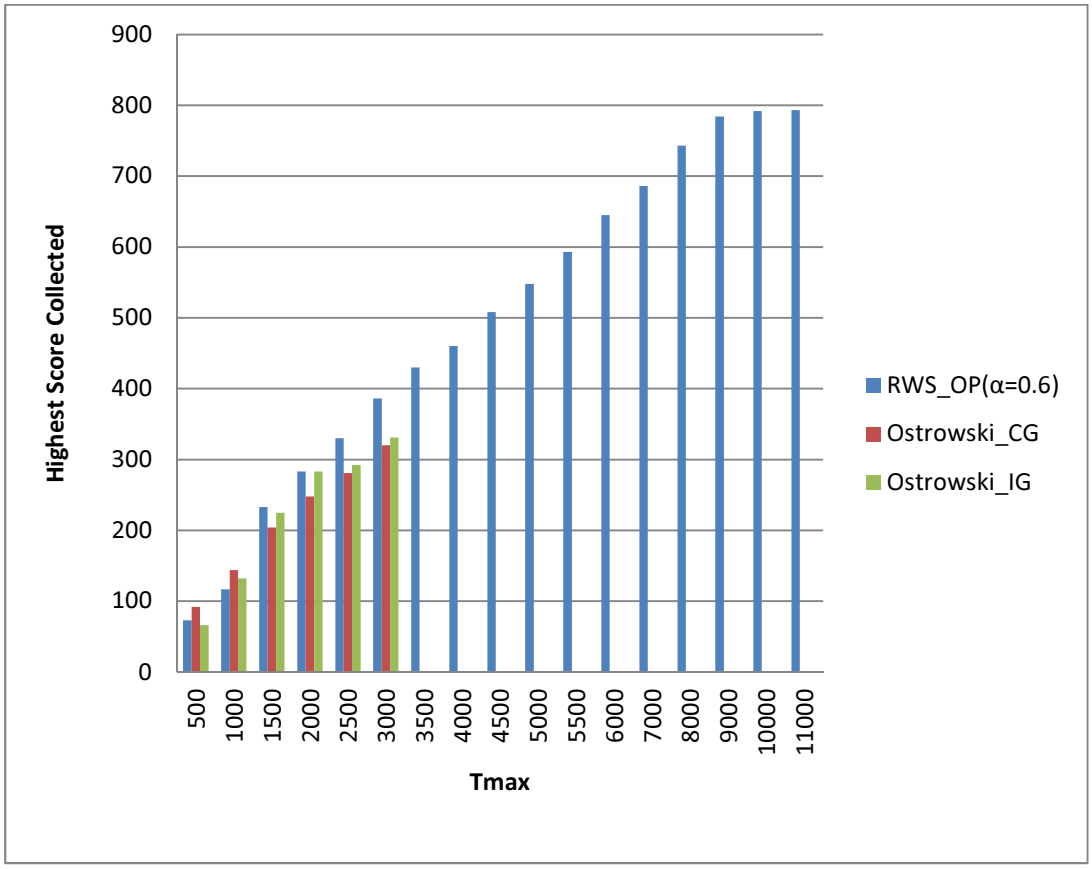


Fig. 4.13: Plot showing that RWS_OP can achieve higher total collected score for larger T_{max} values as compared to Ostrowski_CG and Ostrowski_IG methods when implemented on a Real Road Network database with 306 cities of Poland at $\alpha = 0.6$

4.4 Flower Pollination Algorithm for Orienteering Problem

A metaheuristic is a combination of some local improvement methods and higher level techniques or strategies. It is basically an iterative generation process that helps in searching, generating or finding a heuristic (partial search algorithm) that explores and exploits the search space efficiently, avoids getting trapped in the local optima and performs a robust search to determine the near to optimal solution for the optimization problem at hand from the solution space (Yang, 2012; Yang, Karamanoglu, & He, 2013). The advantage of metaheuristics is that it can also tackle the optimization problems with

nonlinearity and multimodality. These days in industry and engineering applications, the problem under consideration is extremely complex and to generate an optimal solution for such problems is a challenging task. It has been found by several researchers that metaheuristic algorithms are quite efficient for solving such problems and the latest trend is to apply nature-inspired metaheuristics for solving the critical optimization problems. Several nature-inspired metaheuristics have been developed by the researchers after studying the complex biological systems. These metaheuristics include the genetic algorithm, bat algorithm, firefly algorithm, ant colony optimization algorithm, particle swarm optimization algorithm etc. (Fister Jr., Yang, Fister, Brest, & Fister, 2013).

In this chapter, we present an application of the nature-inspired metaheuristic called the flower pollination algorithm (FPA) introduced by Xin-She Yang in 2012 and use it to solve the NP-Hard orienteering problem. FPA is inspired from the pollination process of the flowers. Pollination is the reproduction process of the flowers which is carried out through agents like bees, bats, birds, insects and other animals. These agents are called pollinators. Pollination can be either abiotic or biotic and can take place in two ways, namely self-pollination and cross-pollination. The pollination that takes place when pollens are carried through some pollinators like insects then it is called biotic and about 90% of the pollination activity is biotic. In 10% of the cases, pollens are carried through natural carriers like wind, water etc. and are termed as abiotic. If the pollination takes place with pollen from a flower of a different plant then it is called cross-pollination and if the fertilization happens due to the pollen from the same flower or a different flower of the same plant then it is termed as self-pollination. Another term that can be associated with this process of pollination is the flower constancy. Honeybee is a pollinator that helps in implementing this phenomenon called flower constancy where the pollinators tend to visit only a specific species of flowers and ignore the other species that exist. Pollinators like birds, bees, insects etc. can fly to long distances. Therefore, biotic, cross-pollination over long distances can be termed as global pollination. Also, the behaviour of the biotic pollinators i.e., their jumps and flying distances etc. follows the Levy distribution as stated by Xin-She Yang (2012).

4.4.1 Algorithm FPA_OP

Input: A graph $G(V, E)$ with t_{ij} (time taken to traverse) value of each edge (e_{ij}) connecting vertex v_i and $v_j \in V$, S_i (score) value of each vertex $v_i \in V$.

Output: A Hamiltonian path with the highest possible collected score such that total travel time is within the specified time budget T_{max}.

Initialize a population of n flowers
/pollen gametes with random solutions
Find the best solution R^* in the initial population
Define a switch probability sp $\in [0,1]$
while $(t < No\_of\_Iterations)$
  for $i = 1 : n$ *(all n flowers in the population)*
    if $rand < sp$
      Do global pollination via
      $index2 = index1 + L(index(R^*) - index1)$
    else
      Randomly choose j and k among all the solutions
      Do local pollination via $index2 = index1 + \epsilon(j - k)$
    end if
    Evaluate new solutions
    If new solutions are better, update them in the population
  end for
  Find the current best solution R^*
end while

In the above stated algorithm, initially n pollens are generated randomly where each pollen represents a possible solution i.e., a path satisfying the constraint that the total time taken by the path is less than the upper bound T_{max}. Then for the valid paths three values are evaluated: (1) the total time taken by the path, (2) the total score collected by the path and (3) the ratio of $score/time(S_i/\ t_{ij})$ for each of the valid paths. Then these paths are stored in a priority queue on the

basis of the (S_i / t_{ij}) ratio. To determine the best solution R^* in the initial solution, any of the selection methods stated in section 2.1 can be implemented. The switch probability $sp \in [0,1]$ controls the type of pollination to be performed i.e., global pollination or local pollination. $rand$ is a randomly generated number and if it is less than sp then global pollination is performed else local pollination is performed. In case of global pollination, a function is used that randomly generates the value for $index1$. Then using $index1$ and Levy distribution L , the value for $index2$ is computed. The value for L is calculated using the following equation:

$$L \sim \frac{\lambda \Gamma(\lambda) sin(\Pi \lambda/2)}{\Pi} \frac{1}{s^{1+\lambda}} \tag{4.10}$$

Where, $\lambda = 1.5$ and s denotes the step size. To make the step size appropriate, so that neither it is too large nor too small, its value is calculated as $size\ of\ the\ priority\ queue/5$. Similarly, in local pollination again the value of $index1$ is considered and two randomly generated solutions j and k are used to calculate the value for $index2$. The value for ϵ is randomly chosen from the interval $[0,1]$. After the two parents are determined (i.e., the paths at $index1$ and $index2$), a crossover operation is performed to compute the child paths. Two points are found randomly in the parent2 (at $index2$) and the nodes lying between the two points of parent2 are inserted in parent1, one by one at the best possible location (which leads to minimum increment in the total time taken). This way child1 is formulated. In a similar manner, two points are selected in parent1 and the nodes lying between the two points are added one by one to parent2 at its best location and this way another path i.e., child2 is created. Then it is checked whether these new paths (child1 and child2) are valid or not i.e., their total time taken is less than T_{max} or not. If the new path is valid then it is stored in the queue else it is discarded. If the new path that has been generated has a better score than the previous best solution, then it is updated in the queue. At the end, when the algorithm terminates, the path obtained with the highest value of total collected score forms the final solution.

4.4.2 Experimental Analysis

The code for FPA_OP was developed in C++ and compiled using CodeBlocks on an Intel Core i5 650 at 2.20 GHz. The code was implemented on instances with 32, 33, 64, 66, 102, 150 etc. nodes. Each instance represents a complete graph. The results obtained for FPA_OP were compared against the best known algorithm in the literature for OP viz. the GRASP algorithm suggested in (Campos, Marti, Sanchez-Oro, & Duarte, 2013). The results obtained (average of 10 runs) by FPA_OP were compared with the C3 method of GRASP and it was found that FPA_OP helps in obtaining higher total score collected value for larger T_{max}. However, at lower T_{max} values the results obtained through GRASP were better than FPA_OP. Fig. 4.14 is a plot for a complete graph with 102 nodes, source=1 and destination=102. The results for the total collected score by GRASP and FPA_OP algorithms are compared for different T_{max} values and the observation stated above can be clearly seen in the plot. A similar behavior was observed on comparing the results (average of 10 runs) of RWS_OP and FPA_OP i.e. FPA_OP performs better for higher T_{max} values. Therefore, the FPA_OP algorithm can be preferred in the cases were achieving a higher total collected score is the priority and the delay in time can be tolerated.

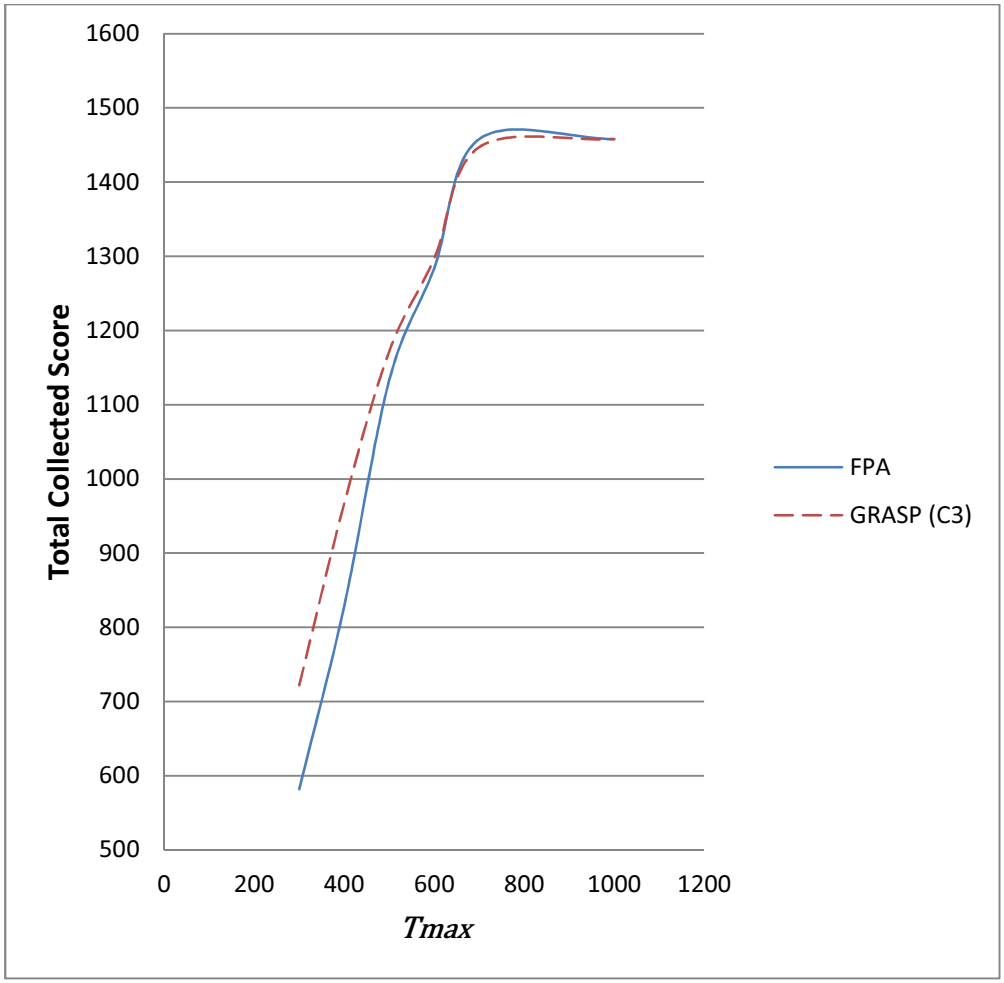


Fig. 4.14: Comparison of the total collected score value achieved by GRASP and FPA algorithms for different T_{max} values when applied on a graph with 102 nodes, source=1, destination=102

4.5 Bidirectional Shortest Path Algorithm for the Constrained Shortest Path Problem with Good Average-Case Behavior

As stated in the earlier chapters, constrained shortest path problem (CSPP) is an NP-Complete problem and several heuristics and some approximation algorithms are already present in the literature to tackle the problem. In 2008, Chen *et al*. proposed a discretization technique to deal with the problem and convert it to a polynomial time solvable one. They introduced two types of discretization, namely randomized discretization and path delay discretization. We prefer the path delay discretization (PDA) here for implementing the

bidirectional shortest path algorithm with the aim to improve the average-case complexity of CSPP. The mathematical formulation of CSPP and path delay discretization technique has been explained in detail in section 3.2.1 (III) and 3.2.1 (IV) of chapter 3.

The traditional shortest path algorithm searches in a forward direction from the source and the search terminates when the goal vertex is reached. Bidirectional search proceeds in two directions. It executes both forward directional search from the source node and backward directional search from the target node. Thus two trees are formed, one the source tree and the other one is the destination tree. Both these trees are iteratively expanded, till a cross edge is found that crosses from one tree to another. Next all such cross edges are considered to find the shortest path between the source and destination pairs. With the following assumptions, the bidirectional search presents a significant improvement in the runtime of the shortest path problem: (1) For each node in the graph $G(V, E)$ all input edges are provided in an order sorted by their edge costs and (2) The edges are chosen independently from the exponential distribution. Bidirectional search has an expected running time of $\Theta(\sqrt{n}\log(n))$, as compared to the runtime of unidirectional search $\Theta(n \log n)$ (Luby, & Ragde, 1989).

Here a technique has been proposed to solve CSPP that chooses the edge costs from an exponential distribution and then uses a sorted adjacency lists to implement a priority queue based algorithm which makes the average running time of the algorithm faster when compared to similar counterparts.

4.5.1 Algorithm

$bidirectional(G, s, d, \lambda)$

1. **if** $s = d$ **then**
2. **return**
3. **for** $i = 0$ to n **do**
4. $zs[i][\lambda] := zd[i][\lambda] := ts[i] := td[i] := \infty$
5. $ununsed[i] := 0$
6. $flag[i] := \boldsymbol{UNVISITED}$

7. $ts[s] := td[d] := 0$
8. $flag[s] :=$ ***S_VISITED***
9. $flag[d] :=$ ***D_VISITED***
10. $ss := s$
11. $sd := d$
12. **if** $degree[s] > 0$ **then**
13. $push\ (qs,\ s,\ to[s][0],\ dist[s][0])$
14. $unused[s] := 1$
15. **if** $degree[d] > 0$ **then**
16. $push\ (qd,\ d,\ to[d][0],\ dist[d][0])$
17. $unused[d] := 1$
18. $minpath := \infty$
19. **while** $\neg empty(qs) \vee \neg empty(qd)$ **do**
20. **if** $\neg empty(qs)$ **do**
21. $(u,\ v,\ l) := Extract\_min(qs)$
22. **if** $flag[v] =$ ***D_VISITED*** **do**
23. $minpath := l + td[v]$
24. $break$
25. **if** $flag[v] =$ ***UNVISITED*** **do**
26. $ts[v] := l$
27. $\lambda' := \left\lfloor \left(\frac{zs[u][\lambda] + delay[u][v]}{r} \right) * \lambda \right\rfloor$
28. $zs[v][\lambda'] := zs[u][\lambda] + delay[u][v]$
29. $flag[v] :=$ ***S_VISITED***
30. $push(ss, v)$
31. **if** $degree[v] > 0$ **then**
32. $push\ (qs,\ v,\ to[v][0],\ dist[v][0])$
33. $unused[v] := 1$
34. **if** $degree[u] > unused[u]$ **then**
35. $push\ (qs,\ u,\ to[u]\left[unused[u]\right],\ dist[u][unused[u]])$
36. $unused[u] := unused[u] + 1$
37. **if** $\neg empty(qd)$ **do**
38. $(u,\ v,\ l) := Extract\_min(qd)$

39. **if** $flag[v] = \boldsymbol{S\_VISITED}$ **do**
40. $minpath := l + ts[v]$
41. $break$
42. **if** $flag[v] = \boldsymbol{UNVISITED}$ **do**
43. $td[v] := l$
44. $\lambda' := \left\lfloor \left(\frac{zd[u][\lambda] + delay[u][v]}{r} \right) * \lambda \right\rfloor$
45. $zd[v][\lambda'] := zd[u][\lambda] + delay[u][v]$
46. $flag[v] := \boldsymbol{D\_VISITED}$
47. $push(sd, v)$
48. **if** $degree[v] > 0$ **then**
49. $push\ (qd,\ v,\ to[v][0],\ dist[v][0])$
50. $unused[v] := 1$
51. **if** $degree[u] > unused[u]$ **then**
52. $push\ (qd,\ u,\ to[u][unused[u]],\ dist[u][unused[u]])$
53. $unused[u] := unused[u] + 1$
54. **while** $(\neg empty(qs) \wedge \neg empty(sd)) \vee (\neg empty(qd) \wedge \neg empty(ss))$ **do**
55. **if** $(\neg empty(qs) \vee \neg empty(sd))$ **then**
56. $(u,\ v,\ l) := Extract\_\min(qs)$
57. **if** $degree[u] > unused[u]$ **then**
58. $push\ (qs,\ u,\ to[u][unused[u]],\ dist[u][unused[u]])$
59. $unused[u] := unused[u] + 1$
60. **if** $(flag[v] = \boldsymbol{D\_VISITED} \wedge (l + td[v] < minpath))$ **then**
61. $minpath := l + td[v]$
62. **if** $(l + td[top(sd)] \geq minpath)$ **then**
63. **while** $(\neg empty(sd) \wedge (l + td[top(sd)] \geq minpath))$ **do**
64. $flag[top(sd)] := \boldsymbol{D\_ELIMINATED}$
65. $pop(sd)$
66. **break**
67. **if** $(\neg empty(qd) \vee \neg empty(ss))$ **then**
68. $(u,\ v,\ l) := Extract\_\min(qd)$
69. **if** $degree[u] > unused[u]$ **then**

70. $push\ (qd\ u,\ to[u][unused[u]],\ dist[u][unused[u]])$
71. $unused[u] := unused[u] + 1$
72. **if** $(flag[v] =$ ***S_VISITED*** $\wedge (l + ts[v] < minpath))$ **then**
73. $minpath := l + ts[v]$
74. **if** $(l + ts[\ top(ss)\ ] \geq minpath)$ **then**
75. **while** $(\neg empty(ss) \wedge l + ts[\ top(ss)\ ] \geq minpath)$ **do**
76. $flag[\ top(ss)\ ] :=$ ***S_ELIMINATED***
77. $pop(ss)$
78. **break**
79. ***return*** $minpath$

To deal with CSPP using our algorithm, firstly two shortest path trees are initialized i.e. one from source and other from the destination. In the above algorithm, lines 1-2 check whether the source and destination are same. Lines 3-18 constitute the initialization steps. Here, arrays td and ts which store the distances of a given node in a tree from its source node are initialized to ∞. zs and zd are the 2D arrays which store the discretized delays to a given node. These are also initialized with ∞. Flags viz. **S_VISITED**, **D_VISITED**, **S_ELIMINATED**, **D_ELIMINATED** and **UNVISITED** are used to mark the presence / absence of nodes in one of the trees. Initially all the flags are marked as **UNVISITED.** Two stacks ss and sd store the order in which the nodes have been covered in their respective trees. These are initialized to the source nodes of both the trees. $unused$ is an array which has been initialized with 0. This acts as a counter that determines the number of nodes explored from the adjacency list of a given node. Two queues qs and qd are used to simulate the bidirectional algorithm on the graph. Source nodes of both the trees are pushed in their respective stacks. $minpath$ that holds the final output is initialized to ∞. Lines 19-53 constitutes the Phase 1 of the bidirectional search. In this phase, the goal is to find out the cross edge between the trees expanding from their respective source nodes. Line 21 takes the minimum value node from the queue qs. Lines 22-24 are used to determine the cross edge. If the next node to be explored is present in the tree growing from the destination node, then we have encountered a cross edge. Lines 25-36 cover the case of a node being

UNVISITED. If this is the case then, using the discretized delay which has been stored in zs matrix and the delay matrix, the new discretized delay parameter λ' is computed. Then the new discretized delay value is updated in the zs matrix for the node under consideration. Thereafter, this node is pushed into the stack and also inserted in the queue. This node is marked **S_VISITED**. Lines 37-53 work on similar lines and repeat the above steps for the tree growing from the destination node. Lines 54-79 mark the phase 2 of the algorithm. This loop executes till either of the queue-stack pair becomes empty. In lines 55-66, queue qs and stack sd are considered. Minimum element from the remaining elements of the queue is extracted out in line 56. If some undiscovered edges from this node are left, then this node is inserted back into the queue as shown in lines 57-59. In lines 60-61, we try to make pairs of this node with the elements in the stack sd to find whether there exists such a node pair which gives a better cost than the one already found. If such a pair is found, our answer is updated. If the above case does not exist then in lines 62-66, the stack is popped till the stack becomes empty or a path cost which is lower than the path cost that has been discovered so far, is found. Similar process is repeated for the queue qd and stack ss in lines 67-79. This algorithm provides the most cost efficient path under the given delay constraints. This is clear from the fact that in phase 1, we find a path with tentatively the minimum cost. If this is the minimum possible one, then phase two is futile. But if this is not, then phase 2 tries out each combination of the undiscovered nodes from either of the queues, by pairing them with the nodes in the stack in order to find out a path with a lower cost. Hence, it is definite that the resultant minimum cost path found is the best one available.

4.5.2 Experimental Analysis

The test cases were generated by the random graph generator *gengraph-win* and weights were assigned using a random number generator that chose the random numbers independently from an exponential distribution. The goal is to perform average case analysis of the bidirectional algorithm. Thus, hundred test cases were generated for each problem size instance and then the average of these results was calculated to determine the average case behaviour of the algorithm.

The numbers generated from the exponential distribution represents the cost of the edges in the graph. The delay values of the edges in the graph were chosen randomly from a uniform distribution. The corresponding edges for each node were then sorted according to their edge weights. The index of the terminating node for each edge was maintained separately and provided with the input.

As stated in the earlier section, the bidirectional shortest path algorithm for CSPP has the following two assumptions and the reason for each of them has been explained in detail in the following subsections:

(1) For each node in the graph $G(V, E)$ all input edges are provided in an order sorted by their edge costs.

(2) The edges are chosen independently from the exponential distribution.

(1) Benefit of sorted edge list representation of a graph

Suppose there are two nodes, one with a high branching factor and other with a lesser branching factor. This branching factor affects the cost of the algorithm. For the normal unsorted edge list representation, all the outgoing edges from these nodes need to be added to the list which will take $O(b)$ time for visiting each node (b is the branching factor), whereas for the sorted edge list representation, only the smallest edge from these nodes needs to be added. This will take only $O(1)$ time (independent of branching factor of the node). Thus for sublinear time complexity either we should have sorted edge list representation or b should be sublinear.

(2) Benefit of exponential distribution

Consider a graph on which a shortest path algorithm is run. The nodes are considered based on their distance from the source node. In case the edge costs are drawn from a uniform distribution then on an average most edges are approximately equal to the average of the interval in which the distribution lies. A path of length 5 will most probably be 5 times longer that the path of length 1. Consequently, the search tends to branch out rather than proceed away from the source (first considering smaller paths before moving on to longer paths).

In exponential distribution, the cost of a whole path is approximately equal. Most importantly the cost of the path is independent of its path length. Thus there is an equal probability of exploring both a shorter and a longer path. Since there are many longer paths than there are shorter paths, the search gets pushed away from the source.

(I) Functions for generating the exponential distribution

Both the following classes require C++11 and –std=c++11 compiler flag must be passed to compile them.

(a) default_random_engine generator

This is a random number generator that returns numbers in uniform distribution. It does not accept any parameters in its constructor. These are passed to other classes, which just transform its output to the desired distribution.

(b) exponential_distribution <double> distribution(lambda)

It is a template of a class. The template parameter accepts the type of random values to be generated (int, float, double etc.). The parameter lambda is the parameter for the exponential distribution. Distribution is the name of the object that we have constructed. To generate the random values we call distribution (generator).

(II) Results

Constraint shortest path algorithm is implemented using two different methods viz. using the Dijkstra's based approach (suggested by Chen *et al*. (2008)) and the bidirectional search based algorithm. Theoretically, the bidirectional algorithm should be faster than PDA algorithm suggested by Chen *et al*. (2008). This theoretical analysis was established by the following practical results which clearly shows that the bidirectional approach in the average case is indeed faster i.e. the bidirectional search algorithm for CSPP has an average case complexity of $O\left(\left(\sqrt{n}\log n\right)L/\varepsilon\right)$. The graph in Fig. 4.15 shows a comparison of the bidirectional search method with the Dijkstra's based method for CSPP

and as can be observed from the plot, there is an improvement in the average execution time of the bidirectional search method when compared to the Dijkstra's based method of Chen *et al*. (2008).

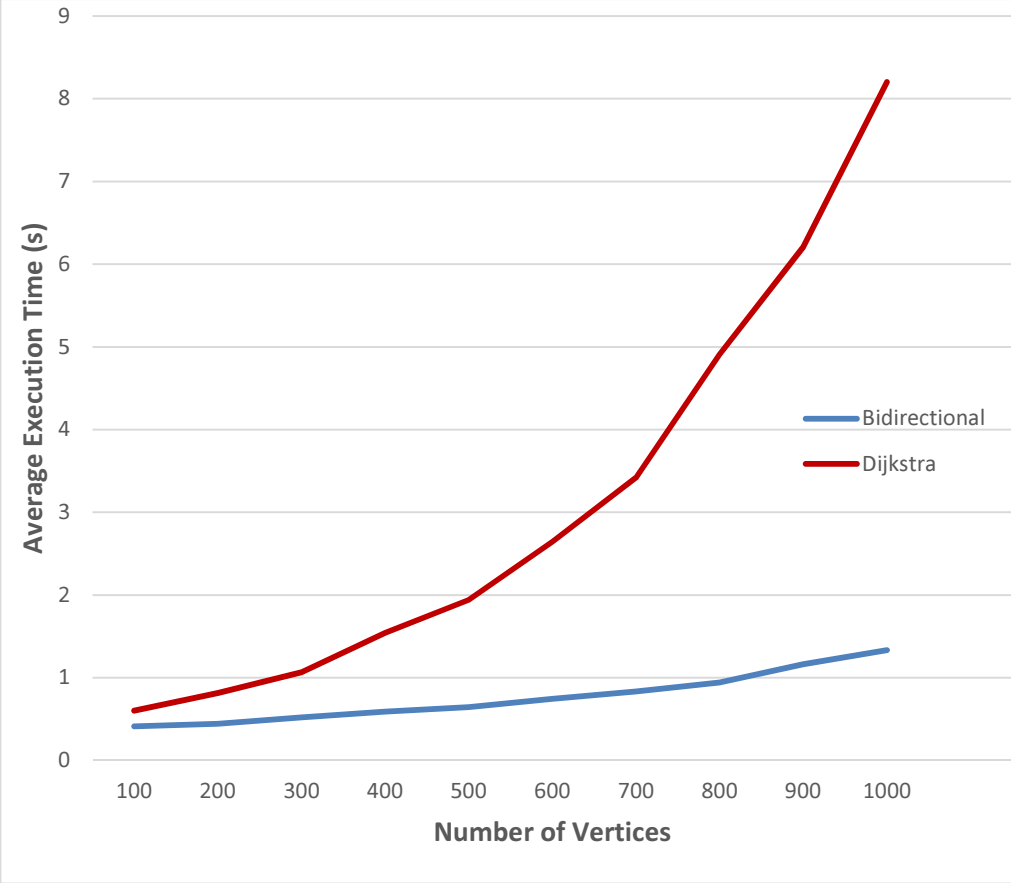


Fig. 4.15: Comparison of the average execution time (s) of the CSPP algorithm suggested by Chen *et al.* (2008) with the bidirectional search algorithm for different network sizes

4.6 Conclusion

In this chapter, a new stochastic greedy heuristic approach (SEL_OP) for the orienteering problem has been introduced which can be applied on both complete as well as incomplete graphs. It was implemented using four different selection methods and the results thus obtained were compared on standard

benchmarks. It was found that roulette wheel selection method performs better than the other three selection techniques and helps in achieving a better total collected score for the specified time budget. Then through experimental analysis, it has been shown that RWS_OP (the algorithm with the roulette wheel selection method) is more efficient than the previously suggested method by Ostrowski *et al*. (2011) for incomplete graphs in terms of execution time. For a particular time bound, the proposed heuristic (RWS_OP) achieves a higher total collected score than the genetic algorithm of Ostrowski *et al*. (2011), utilizes almost 99% of the given time budget and is capable of exploring 70% of the considered search space.

Another meta-heuristic called the flower pollination algorithm suggested by Yang (2012) has been implemented for OP (FPA_OP) and the results thus obtained for instances with different number of nodes has been compared with those obtained by running the GRASP algorithm (Campos, Marti, Sanchez-Oro, & Duarte, 2013) and RWS_OP. It was found that in situations where achieving a better score is the priority at the cost of time delay, FPA_OP algorithm can be preferred as it helps in obtaining a higher total collected score than GRASP or RWS_OP for larger values of T_{max} .

A bidirectional shortest path heuristic was implemented for the constrained shortest path problem. The results shown in the above section proves that an improvement has been achieved (for the average case) over the results present in the literature. The parameter λ in the algorithm plays an important role. The larger the value of λ, greater will be the accuracy of the result as the discretization errors introduced will be lesser.