Asia-Pacific Journal of Operational Research
Vol. 23, No. 2 (2006) 273–285

A HYBRID HEURISTIC FOR THE MINIMUM WEIGHT VERTEX COVER PROBLEM

ALOK SINGH* and ASHOK KUMAR GUPTA†

J. K. Institute of Applied Physics and Technology
Faculty of Science, University of Allahabad
Allahabad – 211002, India
**alok@jkinstitute.org*
†*akgjkiapt@jkinstitute.org*

Received 18 August 2004
Revised 28 January 2005

Given an undirected graph with weights associated with its vertices, the minimum weight vertex cover problem seeks a subset of vertices with minimum sum of weights such that each edge of the graph has at least one endpoint belonging to the subset. In this paper, we propose a hybrid approach, combining a steady-state genetic algorithm and a greedy heuristic, for the minimum weight vertex cover problem. The genetic algorithm generates vertex cover, which is then reduced to minimal weight vertex cover by the heuristic. We have evaluated the performance of our algorithm on several test problems of varying sizes. Computational results show the effectiveness of our approach in solving the minimum weight vertex cover problem.

Keywords: Combinatorial optimization; greedy heuristic; minimum weight vertex cover; steady-state genetic algorithm.

1. Introduction

Let $G = (V, E)$ be an undirected graph, where V denotes the set of vertices and E denotes the set of edges. Any subset V' of V is called a vertex cover of G if $\bigcup_{v \in V'} e(v) = E$, where $e(v)$ is the set of edges incident to vertex v. A vertex cover V' is called minimal if there exists no other vertex cover V'' such that $V'' \subset V'$. Given a non-negative weight function $w : V \rightarrow \mathbb{R}^+$ associated with the vertices of G, the minimum weight vertex cover problem (MWVCP) seeks a vertex cover VC such that $\sum_{v \in VC} w(v)$ is minimized. Clearly minimum weight vertex cover is also minimal but not conversely. This is a NP-hard problem (Garey and Johnson, 1979). Due to this reason exact algorithms for this problem are guaranteed to return a solution only in time that increases exponentially with the number of vertices in the graph and this makes these algorithms infeasible even in case of moderately

*Corresponding author.

large problem instances. Moreover, MWVCP is also hard to approximate. Dinur and Safra (2001) showed that it is impossible to attain approximate solution to the unweighted case ($w(v) = 1 \forall v \in V$) of MWVCP within any factor smaller than 1.3607 unless $P = \text{NP}$. Hence any heuristic nicely approximating MWVCP in a short time in practice is a significant achievement.

Chvatal (1979) proposed a greedy heuristic that constructs the vertex cover by iteratively selecting the vertex having the smallest ratio of its weight to degree. Clarkson (1983) also proposed a heuristic that has the performance guarantee of 2. Randomized algorithm of Pitt (1985) also has the performance guarantee of 2. It randomly selects an end point u of an arbitrary edge say (u, v) to be included in MWVCP, with a probability inversely proportional to its weight, i.e., $w(u)/(w(u)+w(v))$. Shyu *et al.* (2004) proposed an ant colony optimization algorithm for the MWVCP. Khuri and Bäck (1994) proposed a genetic algorithm (Goldberg, 1989; Davis, 1991; Mitchell, 1996) for the unweighted case of MWVCP. Similarly, Evans (1998) developed a genetic algorithm for the unweighted case. However, not much work has been done on the design of heuristic algorithms for MWVCP, although a number of approximation algorithms have been proposed for MWVCP as well as its unweighted case (Bar-Yehuda and Even, 1985; Monien and Speckenmeyer, 1985; Motwani, 1992; Paschos, 1997; Balsubramaniyam *et al.*, 1998; Chen *et al.*, 1999; Hastad, 2001; Dinur and Safra, 2001).

MWVCP has many practical applications in diverse fields such as circuit design, telecommunication network design, network flow, facility location etc.

In this paper, we present a heuristic based steady-state genetic algorithm (HSSGA) for the MWVCP. The steady-state genetic algorithm generates vertex cover which is then reduced to minimal weight vertex cover by the heuristic. The heuristic first finds the set of those vertices in the vertex cover whose all edges are covered by other vertices in the vertex cover and then it quasi-randomly selects the vertex, having the maximum ratio of weight to degree, from this set for deletion from the vertex cover. Whole procedure is repeated again and again until it is impossible to delete any vertex from the vertex cover.

Our genetic algorithm differs significantly from the approaches of Khuri and Bäck (1994), and Evans (1998). Both of these approaches use generational genetic algorithm. Khuri and Bäck (1994) used a bit vector of length n to represent the chromosome, where n is the number of vertices in the graph. A 1 at the ith position indicates that vertex i is in the solution, while a 0 indicates that it is not. Their genetic algorithm uses two point crossover and a mutation rate that depends on instance size. They do not use a repair procedure to transform the infeasible solution, obtained after the application of genetic operators, into feasible solution. Instead they allow the infeasible solutions to exist in the population and use a penalty term in the fitness function to penalize them. They do not hybridize their genetic algorithm with any other heuristic.

The genetic algorithm of Evans (1998) uses opportunistically bounded inverse greedy (OBIG) heuristic (Evans, 1997) and binary decision diagram (BDD)

encoding. In BDD encoding, each bit of the chromosome corresponds not to a particular vertex but to the next decision to be made while constructing a vertex cover using OBIG. Their genetic algorithm uses binary tournament selection, uniform crossover and simple bit flip mutation. Advantage of this scheme is that only feasible solution is generated after the application of genetic operators and there is no need to use the repair procedure or penalty function. However, every feasible solution cannot be represented in this scheme, although optimal solution can always be represented. This may result in loss of diversity in population.

In contrast to these generational approaches, our genetic algorithm is steady-state. The chromosome is represented in the same fashion as in Khuri and Bäck (1994). It uses binary tournament selection, fitness based crossover (Beasley and Chu, 1996) and a variant of bit flip mutation. We have used a repair procedure to transform the infeasible solution, obtained after the application of genetic operators, into a feasible solution. The repair procedure is designed in such a way that apart from repairing the infeasible solution it also performs the job of optimization. We have also hybridized our genetic algorithm with a heuristic.

We have compared our algorithm with Shyu *et al.* (2004) ant colony optimization method. Shyu *et al.* (2004) approach to MWVCP is based on ant colony optimization (Dorigo and Stutzle, 2004), which is an evolutionary approach derived from path finding behavior of real ants. Hereafter, this approach will be referred to as ACO. They have also implemented elementary simulated annealing and tabu search algorithms for MWVCP to compare with their ACO but the ACO always gave the best results. Therefore we compare our algorithm with ACO only. We have used the same test problem instances as used in Shyu *et al.* (2004).

This paper is organized as follows: Section 2 describes our heuristic. Section 3 discusses the steady-state genetic algorithm used in our approach. In Section 4, we have compared our algorithm with ACO. Section 5 outlines some conclusions.

2. The Heuristic

The heuristic reduces a vertex cover to minimal weight vertex cover. The heuristic first finds the set S of those vertices in the vertex cover, all of whose edges are covered by other vertices in the vertex cover. Then it quasi-randomly selects the vertex, having the maximum ratio of weight to degree, from this set for deletion from the vertex cover. Whole procedure is repeated again and again until S becomes empty. The pseudo-code of our heuristic is given below, where *u01* is a uniform variate in [0, 1], VC is the current vertex cover, random(S) is a function that returns an element of S randomly and p_{sc} is the probability with which the vertex having the maximum ratio of weight to degree is selected.

$$S \leftarrow \{v : v \in VC \wedge (\forall (x, v) \in e(v), x \in VC)\}$$
$$\textit{while } (S \neq \varnothing)\{$$
$$\qquad \textit{if } (u01 \leq p_{sc})$$

$v \leftarrow \arg\max_{u \in S}(\frac{w(u)}{\deg(u)})$

else

$v \leftarrow$ *random(S)*

$VC \leftarrow VC - \{v\}$

recalculate S

}

return VC

3. The Genetic Algorithm

We have used a steady-state genetic algorithm (Davis, 1991). Steady-state genetic algorithm uses steady-state population replacement method. In this method genetic algorithm repeatedly selects two parents, performs crossover and mutation to produce a child. The child will replace a less fit member of the population. This is different from generational replacement, where a new population of children is generated and the whole parent population is replaced. The advantages of steady-state population replacement method over generational method are that the best solutions are always kept in the population and the child is immediately available for selection and reproduction. Thus, we can possibly find better solutions faster. Another advantage is the ease with which we can avoid duplicate copies of the same individual in the population. In the generational approach multiple copies of the same individual may exist in the population. Normally these individuals are among the best individuals but they can quickly dominate the whole population. In this situation, no further improvement is possible without mutation, and often, a much higher mutation rate is required to get further improvements. In the steady-state approach the child can be checked against existing population members and if it is identical to any existing individual in the population then it is discarded. In this way duplicate solutions are avoided and the problem of premature convergence is averted.

The main features of our genetic algorithm are described below:

Chromosome Representation. We have used a bit vector of length n to represent the chromosome, where n is the number of vertices in the graph. A 1 at the ith position indicates that vertex i is in the solution, while a 0 indicates that it is not.

Fitness. Fitness of a chromosome is equal to sum of weights of the vertices present in the vertex cover it represents. Here we have to minimize this fitness function.

Crossover. Fitness based crossover as proposed by Beasley and Chu (1996) is used in our genetic algorithm which produces a single child in the following way: Let p_1 and p_2 be two parents. Then the child receives bits from parent p_1 with probability $f(p_2)/(f(p_1) + f(p_2))$ and from parent p_2 with probability $f(p_1)/(f(p_1) + f(p_2))$, where $f(p_1)$ and $f(p_2)$ are respectively the fitness of p_1 and p_2. Clearly in this scheme the child probabilistically receives more bits from better of the two parents.

Crossover is applied with probability p_c, otherwise a random child is generated in which each bit is set to 1 with probability $(0.66 \times minsize/n)$, where *minsize* is the size of the smallest vertex cover (in terms of number of vertices, not weight) so far generated by the algorithm. This is done to increase diversity of the population.

Mutation. A variation of simple bit flip mutation is used, where a bit, which is 1, is set to 0 with probability p_m, and a bit, which is 0, is set to 1 with probability p_m only when ratio of its weight and degree is less than average ratio of weight and degree of all the vertices of the graph.

Selection. Binary tournament selection is used to select the two parents, where the candidate with better fitness is selected with probability p_{better}.

Repair. As the child obtained after crossover and mutation may not be a vertex cover, we have to use some sort of repair procedure to transform the child into a vertex cover. Apart from repairing the child, our repair procedure also performs the job of optimization. The repair procedure is a combination of two heuristics, but only one of them is executed during each iteration.

The first heuristic begins with computing the number of uncovered edges covered by each vertex not in the child. Next the heuristic selects one vertex among the vertices having the largest ratio of number of uncovered edges that are covered by a vertex to its weight to be included in the vertex cover. The whole procedure is repeated until the child becomes a vertex cover.

The second heuristic is inspired by step 3 of the heuristic feasibility operator proposed in Beasley and Chu (1996). It begins with randomly selecting a vertex v not in the child. Then it computes the set of vertices A, not in the child and adjacent to v. The vertex v is also included in A. Next it computes the number of uncovered edges covered by each vertex in A. Then it selects the vertex of A having the maximum ratio of number of uncovered edges which it covers to its weight to be included in the vertex cover. The whole procedure is repeated until the child becomes a vertex cover.

The repair procedure is described in detail by the pseudo-code given below where C is the child obtained after crossover and mutation, S is the set of vertices not in C and p_h is the probability of execution of the first heuristic:

$S \leftarrow V - C$
$if\ (u01 < p_h)\{$
  $while\ (C\ is\ not\ a\ vertex\ cover)\{$
    $max\ \leftarrow 0$
    $for\ (each\ t \in S)\{$
      $if\ (max < (|\{(x,t) : (x,t) \in E \wedge x \in S\}|/w(t))\ and\ u01 < 0.95)\{$
        $max\ \leftarrow |\{(x,t) : (x,t) \in E \wedge x \in S\}|/w(t)$
        $s \leftarrow t$
      $\}$
    $\}$

```
            C ← C ∪ {s}
            S ← S − {s}
        }
    }
    else {
        while (C is not a vertex cover){
            v ← random(S)
            A ← {x : x ∈ S ∧ (x, v) ∈ E} ∪ {v}
            s ← arg max_{t∈A}(|{(x, t) : (x, t) ∈ E ∧ x ∈ S}|/w(t))
            C ← C ∪ {s}
            S ← S − {s}
        }
    }
    return C
```

Clearly the first heuristic is greedier in its approach as it always selects one vertex among the vertices with large ratio and it is expected that it will always perform better than the second one. However, it is slow and leads to premature convergence as it fails to maintain diversity in the population. The second heuristic is comparatively much faster and generates diverse solutions, but produce results which on an average are of inferior quality. Therefore a combination of these two heuristics generates high quality solutions while avoiding the problem of premature convergence.

Replacement policy. The child is first tested for uniqueness among the existing population members. If it is unique, then it always replaces the worst member of the population irrespective of its own fitness, otherwise it is discarded.

Initial population generation. To generate each member of the initial population we first generate the subgraph, where each vertex of the graph can be included in the subgraph with probability 0.5. Then the subgraph is transformed into a vertex cover by the repair procedure and reduced into a minimal weight vertex cover by the heuristic. The minimal weight vertex cover is checked against already generated members for uniqueness, and if it is unique, then it is included in the initial population.

The pseudo-code of our genetic algorithm is given below where *current_best* is the smallest weight vertex cover so far found by the algorithm:

```
generate initial population
current_best ← best solution of the initial population
gen ← 0
while (gen < max_gen){
    if (u01 < p_c){
            Select two parents p_1 and p_2 using binary tournament
            selection
            C ← crossover(p_1, p_2)
            C ← mutate(C)
```

```
        }
        else
                Generate C randomly
    C ← repair(C)
    C ← heuristic(C)
    evaluate(C)
    if unique(C){
                gen ← gen + 1
                include C in the population replacing the worst member
                if f(C) < f(current_best)
                        current_best ← C
    }
}
return current_best
```


4. Experimental Results

The HSSGA has been coded in C and executed on a Pentium 4, 2.4 GHz Linux-based system with 512 MB RAM. We have used the same test problem instances as used in Shyu *et al.* (2004). In all our computational experiments we have used a population of maximum 100 individuals. Population size is determined during the initial population generation phase. While generating the ith individual ($i \leq 100$), if we fail to generate a unique individual after 10 trials, then we set the population size to $i - 1$. Such a step is actually needed for small problem instances. In all our computational experiments we have used $p_{\mathrm{c}} = 0.9$, $p_{\mathrm{m}} = 0.05$, $p_{\mathrm{h}} = 0.2$, $p_{\mathrm{sc}} = 0.5$, $p_{\mathrm{better}} = 0.8$. All these parameter values were chosen after large number of trials. These parameter values provide good results, although they are in no way optimal for all problem instances. We have executed our algorithm until either the optimal solution (if known) is found or 20,000 vertex covers are generated.

We have tested HSSGA on the same test data set as was used in Shyu *et al.* (2004). This data set consists of random graphs of various sizes and edge densities with number of vertices varying from 10 to 1000. There are two types of graphs in this set. In the first type the weights of the vertices are uniformly distributed in [20, 120], while in the second type weight of a vertex i is randomly distributed in $[1, d(i)^2]$, where $d(i)$ is the degree of vertex i. Hereafter graphs of the first type are referred as type-I graphs while graphs of the second type are referred as type-II graphs. Graph instances with number of vertices 10, 15, 20 and 25 are categorized as small problem instances, with number of vertices 50, 100, 150, 200, 250 and 300 are categorized as moderate problem instances, while those with 500, 800 and 1000 vertices are categorized as large problem instances. For a particular graph size and edge density there are 10 instances for small and moderate problem instances, while there is only a single large problem instance.

Tables 1 and 2 compare the performance of HSSGA with ACO on type I instances of small and moderate sizes. Tables 3 and 4 show the results on

Table 1. Performance of HSSGA and ACO on type I small problem instances.

n	m	HSSGA		ACO		OPT
		Avg	Time (s)	Avg	Time (s)[a]	
10	10	284.0	0.000	284.0	0.000	284.0
10	20	398.7	0.000	398.7	0.008	398.7
10	30	431.3	0.000	431.3	0.003	431.3
10	40	508.5	0.000	508.5	0.003	508.5
15	20	441.9	0.000	441.9	0.005	441.9
15	40	570.4	0.000	574.2	0.011	570.4
15	60	726.2	0.000	729.0	0.008	726.2
15	80	807.5	0.000	814.6	0.010	807.5
15	100	880.0	0.000	880.0	0.008	880.0
20	20	473.0	0.000	473.0	0.005	473.0
20	40	659.3	0.000	661.4	0.016	659.3
20	60	861.8	0.000	861.8	0.014	861.8
20	80	898.0	0.001	905.4	0.016	898.8
20	100	1026.2	0.001	1026.8	0.016	1026.2
20	120	1038.2	0.000	1041.5	0.017	1038.2
25	40	756.6	0.000	756.6	0.019	756.6
25	80	1008.1	0.000	1009.6	0.022	1008.1
25	100	1106.9	0.000	1107.4	0.025	1106.9
25	150	1264.0	0.000	1264.0	0.031	1264.0
25	200	1373.4	0.000	1377.7	0.030	1373.4

[a]Execution time on a 1.7 GHz AMD Processor.

type II instances of small and moderate sizes. Columns corresponding to n and m give, respectively, the number of vertices and number of edges in the graph. Only single run of HSSGA was performed on a particular graph instance. As the results depend on the seed used for random number generator, the same seed value of 1 was used in all the experiments. As there are 10 instances for a particular n and m, the results are average of these 10 values. For small problem instances optimal solution values are also reported. The data for ACO as well as optimal solution values for small instances are taken from Shyu *et al.* (2004). Tables 1–4 clearly show the superiority of our approach over ACO as HSSGA is able to outperform ACO on all instances, both in terms of solution quality as well as running time. For small problem instances HSSGA is able to find optimal solution value on all instances. ACO was executed on a 1.7 GHz AMD processor-based system with 256 MB RAM. As the two algorithms were executed on different machines it is not possible to exactly compare the speed of two algorithms, however even assuming that our computer is 1.5 times faster, HSSGA is several times faster than ACO on most of the instances.

Table 5 compares the performance of HSSGA and ACO on type I large instances. As there is a single instance for a particular n and m, HSSGA is executed 10 times on a particular instance each time with a different random seed. The results are average solution values of these 10 trials. The results reported for ACO are the

Table 2. Performance of HSSGA and ACO on type I moderate size problem instances.

n	m	HSSGA		ACO	
		Avg	Time (s)	Avg	Time (s)[a]
50	50	1280.0	0.002	1282.1	0.063
50	100	1735.3	0.003	1741.1	0.083
50	250	2272.3	0.003	2287.4	0.097
50	500	2661.9	0.002	2679.0	0.102
50	750	2951.0	0.003	2959.0	0.125
50	1000	3194.4	0.000	3211.2	0.117
100	100	2534.2	0.025	2552.9	0.273
100	250	3602.7	0.040	3626.4	0.367
100	500	4600.6	0.136	4692.1	0.433
100	750	5045.5	0.029	5076.4	0.502
100	1000	5508.2	0.026	5534.1	0.456
100	2000	6051.9	0.036	6095.7	0.589
150	150	3666.9	0.082	3684.9	0.691
150	250	4721.1	0.145	4769.7	0.891
150	500	6172.7	0.398	6224.0	1.194
150	750	6965.0	0.219	7014.7	1.042
150	1000	7370.4	0.223	7441.8	1.206
150	2000	8549.4	0.298	8631.2	1.103
150	3000	8899.8	0.105	8950.2	0.966
200	250	5551.9	0.316	5588.7	1.674
200	500	7202.3	0.247	7259.2	2.160
200	750	8273.3	0.304	8349.8	2.602
200	1000	9153.4	0.248	9262.2	2.221
200	2000	10836.0	0.969	10916.5	2.437
200	3000	11602.4	0.374	11689.1	2.497
250	250	6151.6	0.440	6197.8	2.273
250	500	8442.9	2.602	8538.8	4.016
250	750	9772.9	1.957	9869.4	4.047
250	1000	10760.1	1.019	10866.6	3.755
250	2000	12755.5	1.426	12917.7	3.942
250	3000	13738.0	0.559	13882.5	4.276
250	5000	14671.3	0.793	14801.8	3.842
300	300	7295.8	2.143	7342.7	4.322
300	500	9416.8	1.415	9517.4	5.178
300	750	11041.0	4.003	11166.9	6.055
300	1000	12116.3	2.055	12241.7	6.231
300	2000	14744.8	3.873	14894.9	6.488
300	3000	15846.3	1.175	16054.1	6.299
300	5000	17366.3	1.423	17545.4	6.558

[a]Execution time on a 1.7 GHz AMD Processor.

results of execution of Shyu *et al.* (2004) algorithm on our system. Actually, Shyu *et al.* (2004) did not provide solution values for large instances. It only showed the superiority of ACO over simulated annealing and tabu search by giving the relative solution quality of these metaheuristics with respect to the ACO. Therefore

Table 3. Performance of HSSGA and ACO on type II small problem instances.

n	m	HSSGA		ACO		OPT
		Avg	Time (s)	Avg	Time (s)[a]	
10	10	18.8	0.000	18.8	0.003	18.8
10	20	51.1	0.000	51.1	0.003	51.1
10	30	127.9	0.000	127.9	0.003	127.9
10	40	268.3	0.000	268.3	0.010	268.3
15	20	34.7	0.000	34.7	0.005	34.7
15	40	170.5	0.000	171.5	0.010	170.5
15	60	360.5	0.000	360.8	0.008	360.5
15	80	697.9	0.000	698.7	0.014	697.9
15	100	1130.4	0.000	1137.8	0.008	1130.4
20	20	32.9	0.001	33.0	0.011	32.9
20	40	111.6	0.000	111.8	0.017	111.6
20	60	254.1	0.000	254.4	0.016	254.1
20	80	452.2	0.000	453.1	0.016	452.2
20	100	775.2	0.000	775.2	0.016	775.2
20	120	1123.1	0.000	1125.5	0.017	1123.1
25	40	98.7	0.000	98.8	0.025	98.7
25	80	372.7	0.000	373.3	0.026	372.7
25	100	595.0	0.000	595.1	0.028	595.0
25	150	1289.9	0.000	1291.7	0.030	1289.9
25	200	2709.5	0.000	2713.1	0.030	2709.5

[a]Execution time on a 1.7 GHz AMD Processor.

Table 4. Performance of HSSGA and ACO on type II moderate size problem instances.

n	m	HSSGA		ACO	
		Avg	Time (s)	Avg	Time (s)[a]
50	50	83.7	0.002	83.9	0.072
50	100	271.2	0.003	276.2	0.097
50	250	1853.4	0.001	1886.8	0.111
50	500	7825.1	0.002	7915.9	0.120
50	750	20079.0	0.000	20134.1	0.111
100	50	67.2	0.013	67.4	0.184
100	100	166.6	0.019	169.1	0.334
100	250	886.5	0.059	901.7	0.514
100	500	3693.6	0.022	3726.7	0.481
100	750	8680.2	0.022	8754.5	0.444
150	50	65.8	0.025	65.8	0.292
150	100	144.1	0.049	144.7	0.583
150	250	616.0	0.073	625.7	1.387
150	500	2331.5	0.379	2375.0	1.908
150	750	5702.3	0.117	5799.2	1.295

Table 4. (*Continued*)

n	m	HSSGA		ACO	
		Avg	Time (s)	Avg	Time (s)[a]
200	50	59.6	0.042	59.6	0.463
200	100	134.5	0.085	134.7	0.981
200	250	483.1	0.336	488.7	2.413
200	500	1804.3	0.273	1843.6	3.423
200	750	4046.5	0.545	4112.8	3.600
250	250	419.1	0.657	423.2	3.311
250	500	1436.8	0.384	1457.4	5.781
250	750	3265.2	0.373	3315.9	5.983
250	1000	5993.3	0.457	6058.2	6.297
250	2000	25701.2	0.579	26149.1	4.859
250	5000	170306.3	0.856	171917.2	4.856
300	250	399.6	0.481	403.9	5.372
300	500	1216.9	1.559	1239.1	9.155
300	750	2644.5	1.111	2678.2	10.994
300	1000	4808.7	2.264	4895.5	9.045
300	2000	20936.1	0.788	21295.2	7.242
300	5000	141278.3	1.646	143243.5	6.553

[a]Execution time on a 1.7 GHz AMD Processor.

Table 5. Performance of HSSGA and ACO on type I large problem instances.

n	m	HSSGA		ACO	
		Avg	Time (s)	Avg	Time (s)
500	500	12621.2	14.256	12715.2	84.667
500	1000	16470.4	10.331	16592.6	104.671
500	2000	20882.6	7.223	21141.0	119.371
500	5000	27393.4	10.266	27945.5	141.306
500	10000	29627.2	8.553	30190.3	150.557
800	500	15025.0	60.296	15057.0	172.626
800	1000	22774.5	96.205	22952.0	230.080
800	2000	31433.8	91.797	31913.8	291.971
800	5000	38742.9	58.266	39120.6	337.751
800	10000	44478.3	34.753	45045.0	361.983
1000	1000	24730.2	138.868	24853.6	329.111
1000	5000	45347.2	62.514	46001.5	512.767
1000	10000	51736.0	102.958	52333.7	552.558
1000	15000	58203.9	63.946	59237.5	571.752
1000	20000	59948.0	37.700	60404.0	586.997

we borrowed executable version of ACO program from S. J. Shyu and executed it 10 times on each of the large problem instances on our system. We ran the ACO for 2000 iterations (as ACO uses 10 ants, so in 2000 iterations ACO also generates 20,000 vertex covers) with the same parameter values as reported in Shyu

et al. (2004). From Table 5 it is clear that in this case also HSSGA outperforms the ACO on all instances, both in terms of solution quality as well as running time.

5. Conclusions

We have designed and implemented a heuristic based steady-state genetic algorithm for the minimum weight vertex cover problem, which outperforms the Shyu *et al.* (2004) ACO approach. Our algorithm performs well not only in terms of solution quality but also in terms of running time.

As a future work we plan to develop a GRASP algorithm for the same problem using the repair procedure and the heuristic.

Acknowledgments

We are grateful to S. J. Shyu for making available his test problem set as well as executable version of his program. We are also thankful to Anurag Singh Baghel for many fruitful discussions.

References

Balasubramanian, R, MR Fellows and V Raman (1998). An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters*, 65, 163–168.

Bar-Yehuda, R and S Even (1985). A local-ratio theorem for approximating with the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25, 27–45.

Beasley, JE and PC Chu (1996). A genetic algorithm for the set covering problem. *European Journal of Operation Research*, 94, 394–404.

Chen, J, IA Kanj and W Jia (1999). Vertex cover: further observations and further improvements. *The 25th International Workshop on Graph-Theoretical Concepts in Computer Science*.

Chvatal, V (1979). A greedy-heuristic for the set cover problem. *Mathematics of Operations Research*, 4, 233–235.

Clarkson, KL (1983). A modification of the greedy algorithm for vertex cover. *Information Processing Letters*, 16, 23–25.

Davis, L (1991). *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.

Dinur, I and S Safra (2001). The importance of being biased. Technical Report, Department of Computer Science, Tel Aviv University, Israel.

Dorigo, M and T Stutzle (2004). *Ant Colony Optimization*. MIT Press/Bradford Books.

Downey, RG and MR Fellows (1995). Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24, 873–921.

Evans, IK (1997). Reemphasizing recombination in evolutionary search: heuristics for vertex cover. Ph.D. Thesis. University of Iowa.

Evans, IK (1998). Evolutionary algorithms for vertex cover. In *Proceedings of the 7th International Conference on Evolutionary Programming (EP98)*. Springer, 377–386.

Garey, MR and DS Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: W.H. Freeman.

Goldberg, DE (1989). *Genetic Algorithm in Search, Optimization, and Machine Learning*. Reading, MA: Adisson-Wesley.

Hastad, J (2001). Some optimal inapproximability results. *Journal of the ACM*, 48, 798–859.

Khuri, S and T Bäck (1994). An evolutionary heuristic for the minimum vertex cover problem. In: J Kunje and H Stogan (eds.) *KI-94 Workshops (Extended Abstracts)*, 83–84.

Mitchell, M (1996). *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press.

Monien, B and E Speckenmeyer (1985). Ramey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica*, 22, 115–123.

Motwani, R (1992). Lecture notes on approximation algorithms. Technical Report No. STAN-CS-92-1435, Department of Computer Science, Stanford University.

Paschos, VT (1997). A survey of approximately optimal solutions to some covering and packing problems. *ACM Computing Surveys*, 29, 171–209.

Pitt, L (1985). A simple probabilistic approximation algorithm for vertex cover. Technical Report No. YaleU/DCS/TR-404, Department of Computer Science, Yale University.

Shyu, SJ, PY Yin and BMT Lin (2004). An ant colony optimization algorithm for the minimum weight vertex cover problem. *Annals of Operations Research*, 131, 283–304.

Alok Singh received the Bachelors and Masters degrees in Computer Science from Banaras Hindu University, Varanasi, India in 1996 and 1998 respectively.

He is currently a Lecturer of Computer Science at the J. K. Institute of Applied Physics and Technology, University of Allahabad, Allahabad, India. He is also a candidate for D. Phil. degree at the same institute. His primary research interests lie in the area of combinatorial optimization with problem-specific heuristics and metaheuristics. He has authored or coauthored six publications in referred proceedings and journals.

Ashok Kumar Gupta received the B.Sc. and M.Sc.(Physics) degrees from University of Allahabad, Allahabad, India in 1960 and 1962, respectively.

He is currently a Professor of Computer Science, and Head at the J. K. Institute of Applied Physics and Technology, University of Allahabad, Allahabad, India. He is also the Director of Institute of Interdisciplinary Studies at the University of Allahabad. He has been the member of University Grants Commission, the apex body of higher education in India. His current research interests are evolutionary computation and bio-informatics. He has authored or coauthored more than 50 publications in referred proceedings and journals. He has published in several prestigious journals such as Physical Review Letters, Physical Reviews, Journal of Heuristics etc.