CODE ASSIGNMENTS IN CDMA NETWORKS: DISTRIBUTED ALGORITHMS AND GENETIC ALGORITHM HEURISTICS

SAMI KHURI	W. MELODY MOH	FLORA CHUNG
Dept of Math & CS	Dept of Math & CS	Advanced C3I Systems
San José State Univ.	San José State Univ.	Raytheon Systems Co.
San José, CA	San José, CA	San José, CA
khuri@cs.sjsu.edu	*melody@cs.sjsu.edu*	*fchung@ti.com*

CDMA is an important spread spectrum communication technology for wireless and mobile networks. We investigate the code assignment problem on a CDMA network with hidden terminal interference, which has been proven to be NP-hard. Our major contributions include: (1) A dynamic distributed code assignment algorithm which we believe is the first one to appear in the literature. By formal analysis, we prove that the algorithm is correct: it is deadlock-free, assigns valid codes to newly arriving nodes within fixed maximum delay, and maintains consistent topology information throughout the network. (2) New centralized heuristics based on genetic algorithms (GA). Experiments are performed on 37 test graphs. For each graph, the GA heuristics perform at least as good as, and in many instances better than, the best result obtained from 1,000 runs of the greedy algorithm.

1 Introduction

Spread spectrum communication technology, and in particular, the Code Division Multiple Access (CDMA), has attracted much attention, and been used extensively in military, industry, and academic communication [9]. It has also been adopted in 1993 by the Telecommunication Industry Association as a technology of the digital cellular system: the IS-95 CDMA standard [22]. CDMA technology requires either transmitters or receivers to communicate over a multitude of codes and a node needs to know which code to use in transmitting or receiving packets. As all the codes share the fixed channel capacity allocated to the network, the number of codes used must not exceed a given bound and should be minimized. Proper code assignment to facilitate spatial reuse of codes is thus important in CDMA networks.

CDMA code assignments can be performed in a variety of ways. When the receivers are code-agile, i.e., able to communicate over several codes, the *transmitted-oriented code assignment (TOCA)* is used [17]. If the transmitters are code agile; the *receiver-oriented code assignment (ROCA)* scheme is used [21]. If both transmitter and receivers are code agile, the *pairwise-oriented code assignment (POCA)* has been proposed [8]. The code assignment for hidden-terminal interference avoidance in a TOCA scheme has been proven to be NP-complete Bertossi and Bonuccelli [1]. They also proposed a heuristic, greedy-based, centralized algorithm, and a distributed algorithm based on the centralized version, without a formal proof of its correctness or performance. The algorithm can only run in a static environment, it does not allow nodes to join or leave the network as in real communication networks.

In the first part of this paper, we first provide a formal correctness proof of a static distributed code assignment algorithm similar to Bertossi's algorithm [1]. Based on this result, we propose a dynamic distributed code-assignment algorithm which allows network nodes to dynamically join and leave the network. It is thus suitable for practical networks, and provides mobility support. We formally analyze its correctness and delay performance. This is part of an on-going research on the design and analysis of network protocols [18,19].

Since the code assignment problem is NP-complete, only heuristic algorithms have been proposed for general network topologies [1,17]. In the second part of the paper, we propose a solution using genetic algorithms (GA). By carefully transferring the code-assignment problem into a GA instance, we design and implement a genetic algorithm to solve the problem. Experimental results of running and comparing GA solutions with those obtained by greedy-based heuristics [1] on 37 different test graphs are presented. The graphs are from DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) benchmark graphs [4] and Knuth's Stanford GraphBase [15]. This work is part of a continuous effort on applying GA to optimization problems [11,12,13,14].

2 The Code Assignment Problem

A CDMA network can be modeled as an undirected graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is the set of vertices, or network nodes. E is the set of edges; two nodes i and j in V are joined by an edge in E if nodes i and j can hear each other's transmission. In this case, i and j are said to be *adjacent* to each other. The *distance* $d_{i,j}$ between two nodes i and j is the number of hops (distinct nodes) that a packet must undergo in a communication between nodes i and j. Also, two nodes i and j can generate a hidden terminal interference if and only if they are two hops away; i.e., $d_{i,j} = 2$. Such an interference can be eliminated if i and j are assigned (or transmit using) different orthogonal codes. Thus, to eliminate hidden terminal interference, the code assignment for the TOCA CDMA network can be defined as follows:

> **Code Assignment Problem**:
> *Assign codes to all the nodes in V so that every pair of nodes that are two hops away is assigned a pair of different codes. The objective is to use a minimum number of codes.*

The above code assignment problem (also known as the 2-hop vertex coloring problem in graph theory) was first introduced by Makansi, along with some heuristic procedures[17]. Bertossi and Bonuccelli proved that the problem is NP-complete[1]. In the following, we present a centralized, greedy-based algorithm, *Algorithm CodeAssignment*, that is similar to Bertossi's[1]. It is the basis for the two distributed algorithms described in the next section. The running time of the above algorithm is $O(n^2)$.

Let $H2(v_i)$ denote the set of stations that are at distance two, i.e., two hops away, from station v_i. Let $code[v_i]$ be the code number assigned to station v_i, and *assignedcodeset* be the set of codes assigned to the stations in $H2(v_i)$. The following algorithm considers stations v_i, $i = 1, \ldots, n$, randomly, one at a time, and assigns code k (color k) to station v_i if it the smallest index code not assigned to the stations in $H2(v_i)$.

Algorithm CodeAssignment:
Randomly choose one station, say v_i, and assign it code number 1; $code[v_i] = 1$;
for $i := 2$ to n
{
    1. *assignedcodeset:=* Φ;
    2. randomly select one station, say v_j;
    3. check each member in $H2(v_j)$;
        4. add the assigned code, if any, to *assignedcodeset*;
    5. $k := 1$;
    6. **while** k is in *assignedcodeset*
        7. $k := k + 1$;
    8. $code[v_j] = k$;
}

3 Distributed Code Assignment Algorithms

Note that the centralized algorithm presented in the previous section assumes that a particular network station (network manager) knows the exact network topology, whereas in a practical network this information is actually distributed throughout the entire network. In this section, we present both static and dynamic distributed algorithms for the code assignment problem, and analyze their correctness and delay performance.

3.1 Static Distributed Code Assignment Algorithm

We denote the nodes by integers: $1 \ldots n$. $H_2(i)$ represents the set of all nodes j which are two hops away from node i and such that $j < i$. Note that the second condition is necessary in the distributed algorithm to ensure a valid code assignment (see Lemma 3). The following algorithm is a modification of Bertossi's [1]. It uses a control message $ASSIGN(i, code[i])$, which is sent by node i to its 2-hop neighbors to inform its code assignment.

Algorithm DistributedStatic (i)
(Distributed Code Assignment for Station i):

```
1.      counter := |H_2(i)|;
2.      while (counter > 0) do
   2a.  if ASSIGN(j, code[j]) is received
            then   2a(i).  if received from j
                                then broadcast ASSIGN(j, code[j])
                   2a(ii). else if j ∈ H_2(i)
                                then   counter := counter − 1;
                                       add code[j] to assignedcodeset;
                           endif
        endif
        enddo
3.      k := 1;
4.      while k ∈ assignedcodeset do
            4a. k := k + 1;
        enddo
5.      code[i] := k;
6.      broadcast ASSIGN(i, code[i]).
```

Due to the limits imposed on the length of the paper, we provide sketches of proofs pertaining to the analysis of delay and correctness. Detailed proofs can be found in the extended version of this paper, which was originally submitted to the conference committee, and is available at one of the author's web page [20].

Lemma 1
In *DistributedStatic*, if all the nodes are numbered distinctly in some specific order, then there is at least one node i for which $H_2(i)$ is empty; i.e., $H_2(i) = \Phi$. (The lemma can be easily proved from the definition of H_2.) □

Lemma 2
If the network has $n > 2$ nodes, then the *DistributedStatic* algorithm will terminate in no more than $2(n-2)$ hops of communication delay.
(The lemma can be proven by considering two cases depending on whether the network is completely connected or not.) □

Note that we assume in the above proof, and in the rest of this paper, that the internal processing delay at each node is negligible, which is often valid in real networks. We also assume that it takes a 1-hop delay for a message to go from one node to another. This delay should include proper transmission and propagation delay which are often known to individual network nodes. It may also include some nondeterministic queueing delay; quantifying queueing delay is not in the scope of this paper.

Lemma 3
The code-assignment of *DistributedStatic* is valid.
The lemma may be proven by contradiction. We assume that nodes i and j are 2-hop neighbors and that code[i] = code[j], and argue that code[j] $\neq$ code[i]. □

Theorem 4
The distributed code-assignment algorithm, *DistributedStatic*, is correct. In other words, it avoids infinite wait and deadlock (by Lemmas 1 and 2), and its code assignment is valid (by Lemma 3). □

3.2 Dynamic Distributed Code Assignment Algorithm

In this section, a dynamic distributed code assignment algorithm, *Distributed-Dynamic*, which serves as a code-assignment protocol in a wireless, mobile CDMA network, is presented. The code assignment and the topology information are dynamically updated in each network node. First, we define additional data structures that *DistributedDynamic* uses. For each node i, we denote:

> *time*[*i*]: the time when code[i] is assigned to node i
> assignedset(i)= {(j,code[j],time[j]), for all j in $H_2(j)$}
> $H_2(i) = \{j|j$ is 2-hop away from i and $time[j] < time[i]\}$
> $H_1(i) = \{j|j$ is adjacent to i}

The following additional control messages are used:

> *JOIN(i)*: Broadcast by node i to inform of its intention to join the network.
> *LEAVE(j)*: Broadcast by node i to inform of its intention to leave the network.
> *NEIGHBOR(i)*: Sent by node i to inform a new node of its neighborhood relationship. *STOP(i)*: Sent by node i to temporarily stop another node (with identity larger than i) from joining the network.

We first present the algorithm, followed by a description.

Algorithm DistributedDynamic (i)
(Dynamic Distributed Code Assignment for station i):
I. Initially,
if node i is in the network which has no code assignment for its nodes

```
        then node i executes Algorithm DistributedStatic
    else                      // Dynamic network, nodes are free to join or leave

II.   if node i wants to join the network
        then   1. time[i] = −1;   // −1 is the maximum integer for the field time[i]
               H_1(i), H_2(i), assignedset, and assignedcodeset are set to be empty;
               broadcast JOIN(i);
   2. Repeat
     a.   Receiving NEIGHBOR(j);
          if received from j
               then add j to H_1(i);
                    broadcast NEIGHBOR(j);
          else add j to H_2(i);
          endif
     b.   Receiving ASSIGN(j, code[j], time[j]);
          if received from j
               then broadcast ASSIGN(j, code[j], time[j]);
          else  add j to H_2(i);
               add (j, code[j], time[j]) to assignedset;
               add code[j] to assignedcodeset;
          endif
     c.   Receiving JOIN(j);
          if received from j;
               then add j to H_1(i);
                    broadcast JOIN(j);
          else  if j > i          // new node with a larger identity number
                    then send STOP(i) to j,
                else              // new node with a smaller id, node i should yield
                    add j to H_2(i);
                    quit & wait until received ASSIGN(j,code[j],time[j]) to re-join;
                endif
          endif
     d.   Receiving STOP(j);
          if received from j;
               then broadcast STOP(j) ;
          else
               add j to H_2(i);
               quit & wait until received ASSIGN(j,code[j],time[j]) to re-join;
          endif
     e.   Until (4-hop delay) and (received ASSIGN from all nodes in H_2(i));
   3. k := 1;
   4.  while k ∈ assignedcodeset do
```

```
            k := k + 1;
        enddo
     5. code[i] := k;
     6. time[i] := current − time;
     7. broadcast ASSIGN(i, code[i], time[i]);
        endif                                   // of Step II

III.    if node i wants to leave the network
            then  broadcast LEAVE(i);
                  clear H_2(i), H_1(i), assignedset, and assignedcodeset.
        endif

IV.     if received JOIN(j)
            then  if received from j;
                      then  add j to H_i(j);
                            send NEIGHBOR(i) to j;
                            broadcast JOIN(j);
                  else send ASSIGN(i, code[i], time[i]) to j;
                  endif
        endif
V.      if received ASSIGN(j, code[j], time[j])
            then  if received from i
                      then broadcast ASSIGN(j, code[j], time[j]);
                  else          // dynamically update topology information
                       if time[j] < time[i]
                            then  add j to H_2(i)
                                  add (j, code[j], time[j]) to assignedset;
                                  add code[j] to assignedcodeset;
                       endif
                  endif
        endif

VI. 1.  if receive LEAVE(j)
         then
            a. if receive from j
                  then  remove j from H_1(i);
                        broadcast LEAVE(j);
            b. else                     // j is a 2-hop neighbor of i
                  if j ∈ H_2(i)
                      then  remove j from H_2(i);
                            remove (j, code[j], time[j] from assignedset);
                       if code[j] not in any entry of assignedset
```

then remove *code*[j] from *assignedcodeset*;
endif
endif
endif
endif // of Step VI
endif // of Step I

We now give a brief description of the algorithm. A detailed description can be found in the extended version[20]. Step I is for the network to set up an initial code assignment in its initial phase. Step II is executed when a (newly arrived or newly activated) node wants to join the network. It first broadcasts the *JOIN* message and listens to several possible incoming messages, among which the *JOIN(j)* message is the most critical one. When the node receives a *JOIN(j)* message received from another newly arrived node j, if j is a 2-hop neighbor and if $j > i$, then it sends a *STOP(i)* message to node j, since the algorithm gives higher priority to a node with a smaller identity number. Otherwise if $j < i$, then node i quits and waits until it receives *ASSIGN(j,code[j],time[j])* to re-join. Note that the *Until* statement in Step II.2.e is, on one hand, to force a new node i to wait for at least a 4-hop delay to receive ASSIGN packets from all the old nodes in $H_2(i)$, and on the other hand, to ensure that node i will not indefinitely wait for a new node's code (while this new node is also waiting for i's code). Step III is executed by a node which wants to leave the network. It either sends the *NEIGHBOR(i)* message and updates its H_1 set, or sends the *ASSIGN* message to j. In Step V, when an $ASSIGN(j, code[j], time[j])$ message is received, it updates its topology information accordingly. Finally, in Step VI, when a *LEAVE(j)* message is received, it updates its topology information.

Lemma 5
When a new node, i, wants to join the network, and no other nodes $j < i$ join the network concurrently, it will be assigned a code in 4 hops, and it will take at most 6 hops of communication delay for all the active nodes in the network to update their topology information.
The lemma may be proven by considering three cases: Case 1 when all the 2-hop neighbors of node i are currently existing nodes; Case 2 (Case 3) when some 2-hop neighbor j of node i joins the network before (after) node i does. □

Lemma 6
When a node i wants to leave the network, it will take at most two-hop delay for all of its 1-hop and 2-hop neighbors to receive the message and to update the topology information accordingly and consistently.
The lemma may be proved by observing Steps IV.1.a and Step IV.1.b. □

Lemma 7
DistributedDynamic dynamically assigns valid codes to all new nodes.
The lemma may be proved by contradiction while considering four cases. Case 1: $time[j] < time[i]$ and $j < i$; Case 2: $time[j] < time[i]$ and $j > i$; Case 3: $time[j] > time[i]$ and $j < i$; and Case 4: $time[j] > time[i]$ and $j > i$. □

Theorem 8
The *DistributedDynamic* algorithm is deadlock-free (Lemma 5), it assigns valid codes to new nodes (Lemma 7), and maintains consistent control information for all the nodes of the network (Lemmas 5 and 6). □

4 Genetic Algorithms

Genetic algorithms (GA) are general search methods that perform best for problems with very large search spaces and for which more traditional methods fail to find optimal solutions. Though its conceptual background is genetics, GA have been applied to a wide variety of fields, with the common goal of providing improved heuristics for problems that are traditionally hard (NP-hard); i.e., their optimal solutions can not be found in polynomial time.

Starting with a randomly chosen initial population, the genetic algorithm moves from one generation to a generally better one by using the stochastic operations of reproduction, crossover, and mutation. The simple three-operator genetic algorithm [7] is represented in Figure 1 [10].

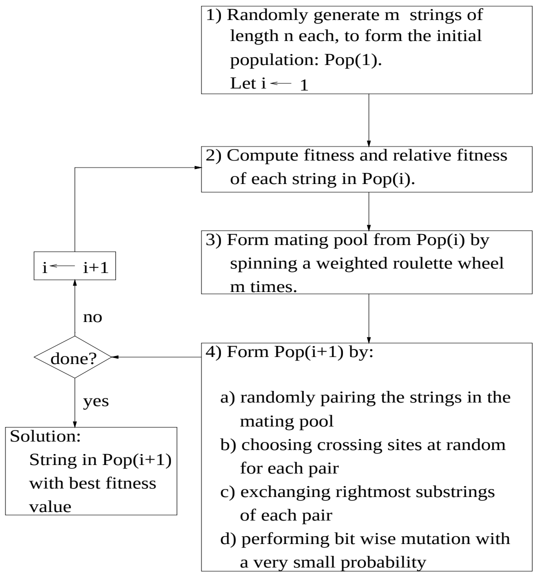

Figure 1: The Simple Genetic Algorithm

Given a problem, the first step involves finding an appropriate representation so that the genetic algorithm can be applied to it. For example, a possible encoding for the color problems is to represent a coloring of the vertices by using non-binary strings of length n, say $v_1 \; v_2 \; \ldots \; v_n$, where the value of v_j represents the color number to which the j^{th} vertex is assigned. If a graph has 11 vertices, for example, then the following string is a possible coloring, where vertex v_1 has color number 4, vertex v_2 has color number 3, and so on.

4	3	5	5	1	3	2	1	4	4	1

Figure 2: Chromosome representation for the color problems

In general, GA start with a randomly generated population, Pop(1), of size m (that is with m strings), each of length n. Each string in the population has a fitness value that measures the strength of the string.

The reproduction operator of the genetic algorithm uses the relative fitness of individuals to direct the search process by preferring better individuals to be copied into the mating pool, where they are subject to recombination and mutation to form the strings of the next generation. One of the ways of implementing reproduction is by building a weighted roulette wheel in which each string in the current population has a roulette wheel slot sized in proportion to its relative fitness. By spinning the roulette wheel m times, we create the mating pool. Thus, highly fit strings in the current population have a better chance of joining the mating pool.

The purpose of crossover consists in the combination of useful string segments from different individuals to form new, hopefully better performing offspring. One-point crossover works by randomly selecting two parent strings from the mating pool, choosing a crossover site (X-site) at random, and exchanging all bits after the X-site to form two new offspring that are placed in the next generation.

The mutation operator works on bit strings by occasionally inverting single bits of individuals. Thus, through reproduction, crossover, and mutation, we move from Pop(1) to Pop(2). The strings of Pop(2) are evaluated and the above technique is used to create Pop(3) and so on, until a desired generation is reached. Very often, the stopping criterion is an a priori determined number of generations. For a more elaborate exposition on genetic algorithms, the reader is referred to Goldberg's textbook [7].

4.1 Grouping Genetic Algorithms

For the graph problems, we implement a variant of the simple genetic algorithm discussed in the previous section: the *Grouping Genetic Algorithms*

(GGA) [5]. We use GGA to develop a heuristic algorithm for the code assignment problem described in Section 2. To further demonstrate the feasibility and strength of genetic algorithms, we also implement GGA for the traditional vertex-coloring problem. The two databases of test graphs (DIMACS bench graphs and Knuth's Stanford GraphBase) used in this work, provide optimal results for the vertex-coloring problems, but not for the code assignment problem, which is not as well-studied and tested. Note that the code assignment problem is identical to the 2-hop coloring problem.

GGA, as the name suggests, focus purely on problems with grouping properties. More precisely, both color problems consist in placing the n nodes of the network in the minimum number of colors (groups) under certain restrictions. GGA adopt all the basic notions of the GA studied in the previous section, with an alteration in the representation of strings.

The first step for the GGA implementation involves choosing a representation for the problem. In the previous section we saw that we can use non-binary strings of length n, where n is the number of vertices. Recall that the strings are of the form $v_1\ v_2\ \ldots\ v_n$, where the value of v_j represents the color number to which the j^{th} vertex is assigned. We augment the string representation with a group part. Nodes of the same color form a group. For example, the chromosome of Figure 2 with the genetic algorithm is represented by the following string:

4	3	5	5	1	3	2	1	4	4	1

A	5	8	11	B	7	C	2	6	D	1	9	10	E	3	4	

In the example, we label colors with letters so as to be able to distinguish them from nodes (that are labeled with numbers). The first half is the object part and is identical to the entire chromosome in GA, while the second half is the group part.

The fitness function of a string considers the object part of the string only. We use a penalty function to cope with constraints [12,13]. A penalty term incorporated in the fitness function is activated, thus reducing the infeasible strings strength relative to the other strings in the population. The fitness function implemented for the graph and the two-hop color problems is the sum of two terms. The first term is the objective function which calculates the minimum number of colors used in the problem instance. The second term is a penalty function used to penalize infeasible strings, where the weight of the penalty reflects the extent of violated pairs occurred in the graph.

The penalty function employed here consists of two terms. The first term of the penalty is an offset term whose sole purpose is to guarantee that infeasible vectors (or solutions) will always yield fitness values that are inferior to the ones obtained from feasible vectors. If we denote the number of colors needed to color a graph by $C(G)$, then $C(G)$ satisfies the following bound [2]:

$$C(G) \leq min\{\Delta(\Delta-1)+1, |V|\}$$

where Δ is the maximum degree of a graph, and $|V|$ is the total number of vertices in the graph. The second term of the penalty function is the square of the number of violated pairs.

More precisely, the fitness function of $x = x_1x_2\ldots x_n$, used in both color problems, is given by:

$$f(\vec{x}) = P(\vec{x}) + s[M + (T(\vec{x}))^2]$$

where $P(\vec{x})$ is the number of different values of the components of $\vec{x}$, and $s=0$ if $\vec{x}$ is feasible, and $s=1$ otherwise,

$$M = min\{\Delta(\Delta-1)+1, |V|\}$$

is the offset term, and $T(\vec{x})$ is the number of pairs of vertices that violates the color condition.

When selecting two parents for generating new offspring, we incorporate the roulette wheel selection technique: each string occupies a slot of size *inversely* proportional to the fitness value, since both color problems are minimization problems. The crossover operator, however, has been altered to work solely with the group part of the string. The details of our implementation of the crossover complies with the guidelines suggested by Faulkenauer [5] and can be found in Chung [3].

5 Experimental Results

We present the computer simulation results of the GGA implementation of the vertex coloring problem and the code assignment problem, and compare them with the greedy-based heuristic algorithm *CodeAssignment* described in Section 2.1. We consider 22 graphs for the vertex coloring problem, and a subset of these, a total 15 graphs, for the code assignment problem. The two sets of experimental results are reported in Table 1 and Table 2. Note that the number of nodes of the test graphs range from 74 to 561, and the number of edges range from 254 to 6,656.

The results of executing the greedy algorithm and the genetic algorithm for the vertex color problem with 22 problem instances are summarized in Table 1. The first eleven graphs of Table 1 are from Knuth's Stanford GraphBase [15] and the remaining eleven are from the DIMACS [4] benchmark graphs. In Table 2, we present the results obtained from running the greedy algorithm and the genetic algorithm with 15 problem instances. The results tabulated under the Greedy

Problem Instances			Optimal	Greedy Algorithm			
Name	Vertices	Edges	Solution	Best	Average	Worst	GGA
anna.col	138	493	11	11	11.20	13	11
david.col	87	406	11	11	11.76	14	11
homer.col	561	1629	13	13	14.02	16	13
huck.col	74	301	11	11	11.01	12	11
jean.col	80	254	10	10	10.31	12	10
games120.col	120	638	9	9	9.06	10	9
miles250.col	128	387	8	8	9.37	12	8
miles500.col	128	1170	20	20	22.10	25	20
queen8_8.col	64	1456	9	**11**	12.55	15	**10**
queen10_10.col	100	2940	–	**14**	15.33	18	**13**
queen13_13.col	169	6656	13	**18**	19.42	22	**17**
myciel6.col	95	755	7	7	7.32	9	7
myciel7.col	191	2360	8	8	8.65	11	8
le450_5a.col	450	5714	5	**12**	13.20	15	**11**
le450_25a.col	450	8260	25	26	28.18	31	26
mulsol.i.2.col	188	3885	31	31	31.17	33	31
mulsol.i.3.col	184	3916	31	31	31.12	33	31
mulsol.i.4.col	185	3946	31	31	31.16	33	31
mulsol.i.5.col	186	3973	31	31	31.02	32	31
zeroin.i.1.col	211	4100	49	49	49.20	50	49
zeroin.i.2.col	211	3541	30	30	30.35	32	30
zeroin.i.3.col	206	3540	30	30	30.36	36	30

Table 1: Test results for the vertex color problem

Algorithm in both tables are the best, the average, and worst results we obtained after 1000 runs on each of the 37 problem instances using the centralized algorithm *CodeAssignment*. As for the genetic algorithm, the results reported in both tables were obtained by running our GGA implementation, previously discussed, once on every problem instance. For all 22 problem instances of the vertex color problem, we use the same number of generations (100), the same population size (500), the same crossover rate (0.6), and the same mutation rate (0.075). We use the same setting for the code-assignment problem, except for the mutation rate. The mutation rates we use are 0.025, 0.125, 0.033, and 0.075, depending on the problem instance.

As can be seen in Table 1 and Table 2, the GGA perform at least as good as, and in some cases, better than the best result obtained by the greedy algorithm. With the vertex color problem, the GGA obtained the optimal solution in 17 cases (out of 21, since the optimal solution for queen10_10.col is not provided). No fine-tuning or tinkering with the stochastic operators was performed. Experimenting with the stochastic operators often leads the GGA to better results. Although we ran the genetic algorithm for 100 generations, all the results recorded in the tables were obtained before the 50^{th} and some even within 15 generations.

The results presented in Table 1 and Table 2 show that the GA are better than the greedy algorithm for the code assignment as well as the vertex color problems. One should also note that, both greedy algorithms are specially devised for the color problem while genetic algorithms are general purpose evolutionary heuristics designed for a wide range of problems.

Problem Instances			Greedy Algorithm			
Name	Vertices	Edges	Best	Average	Worst	GGA
anna.col	138	493	37	40.66	45	37
david.col	87	406	35	36.63	41	35
huck.col	74	301	16	16.70	19	16
jean.col	80	254	17	17.99	22	17
games120.col	120	638	**15**	17.36	20	**14**
queen8_8.col	64	1456	**9**	14.57	19	**8**
queen10_10.col	100	2940	**12**	19.68	26	**10**
mulsol.i.1.col	197	3925	41	41.35	43	41
mulsol.i.2.col	188	3885	75	75.69	78	75
mulsol.i.3.col	184	3916	76	76.67	79	76
mulsol.i.4.col	185	3946	76	76.57	78	76
mulsol.i.5.col	186	3973	78	78.72	81	78
zeroin.i.1.col	211	4100	**35**	35.62	38	**34**
zeroin.i.2.col	211	3541	73	73.30	76	73
zeroin.i.3.col	206	3540	74	74.25	77	74

Table 2: Test results for the code assignment problem

6 Conclusion and Future Work

We have proposed a dynamic, distributed algorithm with mobility support, and have formally analyzed its correctness and delay performance. We have also converted the centralized code assignment problem into a genetic algorithm problem, and implemented a GA solution. The new heuristic has performed at least as good as the best result among the 1,000 runs of an existing greedy heuristic for each of the 37 problem instances we examined. Future work may include detailed performance analysis and evaluation of the dynamic distributed algorithm on practical, mobile networks, applicability of GA to other communication problems, and distributed implementation of GA.

Acknowledgment

Melody Moh is supported in part by NSF (grant NCR-9714700), 3Com Technology Development Center, Lockheed Martin Missiles and Space, and Silicon Valley Research, Inc.

1. A. Bertossi and M. Bonuccelli. "Code assignment for hidden terminal interference avoidance in multihop packet radio networks," *IEEE Transactions on Networks*, 3:441–449, August, 1995.
2. J.A. Bondy and U.S.R. Murty. *Graph Theory with Applications*, McMillan, New York, 1976.
3. F. Chung. "Heuristic algorithms for graph coloring problems," Master Thesis, San José State University, San José, CA, 1997.
4. DIMACS: Center for Discrete Mathematics and Theoretical Computer Science. ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color.
5. E. Falkenauer. "A new representation and operators for genetic algorithms applied to grouping problems," *Evolutionary Computation*, pp. 123–144. The MIT Press, Massachusetts.
6. M.R. Garey and D.S. Johnson. *Computers and Intractability–A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.
7. D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison–Wesley, NY, 1989.
8. L. Hu. "Distributed code assignments for CDMA packet radio networks," *IEEE Transactions on Networks*, 6:668–677, December 1993.
9. IEEE, *CDMA Networks I - IV, IEEE Journal of Selected Areas in Communications,* Vol. 12, No. 4, May 1994, Vol. 12, No. 5, June 1994, Vol. 14, No. 8, Oct. 1996, and Vol. 14, No. 9, Dec. 1996.
10. S. Khuri and A. Batarekh. "Heuristics for the integer knapsack problem," *Proceedings of the* 10^{th} *International Conference of the Chilean Computer Science Society*, 161–172. Santiago, Chile, 1990.
11. S. Khuri and T. Bäck. "An evolutionary heuristic for the minimum vertex cover problem," *Max-Planck-Institut für Informatik*, MPI-I-94-241, pp. 86-90, August 1994.
12. S. Khuri, T. Bäck, and J. Heitkötter. "An evolutionary approach to combinatorial optimization problems," *Proceedings of the* 22^{nd} *ACM Computer Science Conference*, 66–73. ACM Press, 1994.
13. S. Khuri, T. Bäck, and J. Heitkötter. "The zero/one multiple knapsack problem and genetic algorithms," *Proceedings of the 1994 ACM Symposium on Applied Computing*, pp. 188–193. ACM Press, 1994.
14. S. Khuri and T. Chiu. "Heuristic algorithms for the terminal assignment problem," *Proceedings of the 1997 ACM Symposium on Applied Computing*, pp. 247-251, ACM Press, 1997.
15. D. Knuth. *The Stanford GraphBase*, ACM Press, NY, 1994.
16. F.T. Leighton. *Journal of Research of the National Bureau of Standards*, 84:489–505. 1979.
17. T. Makansi. "Transmitter-oriented code assignment for multihop packet radio," *IEEE Trans. Communications,* vol. 35, pp. 1370-1382, Dec. 1987.
18. C. Martel, W. M. Moh, and T.-S. Moh. "Dynamic prioritized conflict resolution on multiple access broadcast networks", *IEEE Transactions on Computers,* pp. 1074-1079, Sept. 1996.
19. W. M. Moh. "On multicasting ABR protocols for wireless ATM networks," *Proc. of IEEE International Conference on Network Protocols (ICNP),* pp.

24-31, Atlanta, GA., Oct. 1997.
20. Melody Moh's web site: http://www.mathcs.sjsu.edu/faculty/melody
21. E.S. Sousa and J.A. Silvester. "Spreading code protocols for distributed spread-spectrum packet radio networks," *IEEE Trans. Communications,* pp. 272-281, Mar. 1988.
22. A. J. Viterbi. *CDMA Principles of Spread Spectrum Communication*, Addison-Wesley, 1995.

æ