Algorithmica (2002) 33: 3–18
DOI: 10.1007/s00453-001-0101-z

Algorithmica
© 2002 Springer-Verlag New York Inc.

Experimental Analysis of Heuristic Algorithms for the Dominating Set Problem[1]

L. A. Sanchis[2]

Abstract. We say a vertex v in a graph G *covers* a vertex w if $v = w$ or if v and w are adjacent. A subset of vertices of G is a *dominating set* if it collectively covers all vertices in the graph. The dominating set problem, which is NP-hard, consists of finding a smallest possible dominating set for a graph. The straightforward greedy strategy for finding a small dominating set in a graph consists of successively choosing vertices which cover the largest possible number of previously uncovered vertices. Several variations on this greedy heuristic are described and the results of extensive testing of these variations is presented. A more sophisticated procedure for choosing vertices, which takes into account the number of ways in which an uncovered vertex may be covered, appears to be the most successful of the algorithms which are analyzed. For our experimental testing, we used both random graphs and graphs constructed by test case generators which produce graphs with a given density and a specified size for the smallest dominating set. We found that these generators were able to produce challenging graphs for the algorithms, thus helping to discriminate among them, and allowing a greater variety of graphs to be used in the experiments.

Key Words. Dominating set, Approximation algorithms, Test cases.

1. Introduction. A dominating set for a graph $G = (V, E)$ consists of a subset of vertices $V' \subseteq V$ such that every vertex in $V - V'$ is adjacent to some vertex in V'. The dominating set problem consists of finding a smallest possible dominating set in a graph. The size of this smallest dominating set is called the domination number of the graph. The dominating set problem and its variants can be used to model many important problems for networks and communications. Unfortunately, the dominating set problem is also known to be NP-hard, and in fact it is Quasi-NP-hard to approximate within a ratio of $(\log n)/48$ [1] (where n is the number of vertices).

One common approach used to approximate dominating sets consists of using a greedy strategy. We say a vertex v covers a vertex w if $v = w$ or if v and w are adjacent. Thus a dominating set D must collectively cover all vertices in the graph. The straightforward greedy strategy for finding a small dominating set in a graph consists of choosing vertices which cover the largest possible number of previously uncovered vertices. Parekh has done some theoretical analysis of this algorithm [4]. However, the nonapproximability result stated in the previous paragraph implies that no really satisfactory worst-case bound is likely to be found by analytical means for a polynomial-time approximation algorithm for the dominating set problem.

In practice, however, it is possible that an approximation algorithm will perform well experimentally, even if only for certain types of instances. In this paper we concentrate on the experimental analysis of approximation algorithms for the dominating set problem.

[1] This research was supported in part by NSF Grant CCR-9402334.
[2] Department of Computer Science, Colgate University, Hamilton, NY 13346, USA. laura@cs.colgate.edu.

Received October 27, 1998; revised March 25, 2001. Communicated by R. Battiti and A. Bertossi.
Online publication February 20, 2002.

The algorithms we consider use various versions of the greedy strategy combined with randomization. One approach, which we found to be the most successful, involves taking into account the number of ways in which an uncovered vertex may be covered, when determining its impact on the vertices to be chosen for the dominating set. We also found that using a local search procedure to improve on the dominating set found by the greedy strategy is moderately productive, although time consuming.

One common approach taken in the experimental analysis of algorithms is to run the algorithms on instances constructed at random. Testing heuristic algorithms on random graphs alone, however, may not give a full picture of the effectiveness of the algorithm. For one thing, random graphs have an asymptotic expected domination number, and some algorithms can be shown to almost always come near this value when applied to large enough random graphs [3].

To obtain graphs with a greater variety of domination numbers and behavior, we turned to a generator which produces graphs with known dominating set size and a given density. The procedure is based on an algorithm described in [6] along with other test case generators for NP-hard problems. Being able to test an algorithm on graphs whose parameters differ from random graphs and which have a known minimum dominating set, we feel is useful in obtaining better information about the capabilities of the algorithm.

Such construction algorithms, however, must be designed with care. It can be shown that no such construction procedure is capable of producing all graphs unless NP = co-NP [6]. It is therefore important to make sure that the construction procedure does not produce a set of "easy" graphs. This topic was considered in [6] and the reader is referred there for further information.

The following section provides descriptions of the algorithms which were tested, including pseudo-code. The test case construction procedures are described in Section 3. Section 4 presents the experimental results, and Section 5 summarizes the conclusions.

2. Description of the Algorithms.

All of the algorithms described here work by constructing a set D which will at the end of the process constitute a dominating set for the graph. Usually the set D starts out being empty, and we add vertices to it until it becomes a dominating set. For one of the algorithms, the process works in reverse: initially every vertex belongs to D, and we gradually remove vertices from D.

Recall that a vertex is covered by D if it is either in D or is adjacent to a vertex in D. Thus D is a dominating set if all vertices are covered by D.

Assume that n is the number of vertices in the graph, and that the vertices are numbered $v_0, v_1, \ldots, v_{n-1}$. Let m be the number of edges in the graph.

2.1. *Greedy.*

The first algorithm, `Greedy`, is a well-known greedy heuristic for the dominating set problem. Initially D is empty. At each iteration of the algorithm, we add a vertex to D which would cover the maximum number of previously uncovered vertices. If more than one vertex satisfies this criteria, the vertex to be added is chosen uniformly at random from among the eligible vertices.

In the pseudo-code below, $weight_i$ denotes the number of previously uncovered vertices which would be covered by adding vertex v_i. The variable $covered_i$ is `true` if v_i is covered by D.

```
**ChooseVertex**(Vector ***weight***)
 1. $M = max_{1 \le i \le n} weight_i$
 2. if $M = 0$ return $-1$
 3. else
 4.       $S = \{v_i | weight_i = M\}$
 5.       Return element of $S$ chosen uniformly at random.
```

Adjust weights after adding v_i to D

```
**AdjustWeights**(Graph $G$, Vector ***weight***, Vector ***covered***, Vertex $v_i$)
 1. $weight_i$ = 0
 2. for each neighbor $v_j$ of $v_i$ such that $weight_j > 0$
 3.           if not $covered_i$ decrement $weight_j$
 4.           if not $covered_j$
 5.                 $covered_j$ = true
 6.                 decrement $weight_j$
 7.                 for each neighbor $v_k$ of $v_j$
 8.                       if $weight_k > 0$, decrement $weight_k$
 9. $covered_i$ = true
```

```
**Greedy**(Graph $G$)
 1. $D = \emptyset$
 2. For all vertices $v_i$
 3.           $weight_i$ = $1 + degree(v_i)$
 4.           $covered_i$ = false
 5. Do
 6.           $v$ = **ChooseVertex**(***weight***)
 7.           if $v \neq -1$
 8.                 add $v$ to $D$
 9.                 **AdjustWeights**($G$, ***weight***, ***covered***, $v$)
10. Until $v = -1$
11. Return $D$
```

Using an adjacency list implementation for the graph, it can be checked that each call to `ChooseVertex` takes $O(n)$ time, and that the total time spent in `AdjustWeights` throughout execution of the algorithm is $O(m)$. If d is the size of the dominating set found by the algorithm, the total complexity of the algorithm is $O(nd + m) \le O(n^2)$.

2.2. *Greedy_Rev.* The next algorithm, `Greedy˙Rev`, works in the opposite way. Initially D contains every vertex in the graph. At each iteration we remove a vertex from D which does not uniquely cover any vertex; in other words, D will still be a dominating set after removal of this vertex. In order to choose which vertex to remove, we sort the vertices in order of increasing degree, and at each step we choose the vertex of smallest degree which is eligible to be removed. If there is more than one vertex with the same smallest degree, the vertex to be removed is chosen uniformly at random.

In the following, all vertices start out in the set D. The variable $uniquely_i$ is `true` if removing v_i from D would result in D no longer being a dominating set; $coveredby_i$ is the number of vertices in D which cover v_i.

```
**ChooseVertex**(Set $D$, Vector ***uniquely***, int *start*)
 1. while ($start < n$) and (($v_{start} \notin D$) or $uniquely_{start}$)
 2.         increment *start*
 3. if $start \geq n$ return $-1$
 4. else
 5.         $min = degree(v_{start})$
 6.         $S = \{v_i | degree(v_i) = min$ and $v_i \in D$ and not $uniquely_i\}$
 7.         Return element of $S$ chosen uniformly at random.

**Adjust**(Set $D$, Vertex $v_i$, Vector ***coveredby***, Vector ***uniquely***)
 1. Decrement $coveredby_i$
 2. For each neighbor $v_j$ of $v_i$
 3.         if $coveredby_i = 1$ and $v_j \in D$
 4.                 $uniquely_j$ = true
 5.         decrement $coveredby_j$
 6.         if $coveredby_j = 1$
 7.                 if $v_j \in D$ $uniquely_j$ = true
 8.                 else for each neighbor $v_k$ of $v_j$
 9.                         if $v_k \in D$ $uniquely_k$ = true

**Greedy_Rev**(Graph $G$)
 1. Sort vertices in increasing order of their degrees
 2. $D = V$
 3. For each vertex $v_i$
 4.         $coveredby_i = degree(v_i) + 1$
 5.         if $coveredby_i = 1$ $uniquely_i$ = true
 6.         else $uniquely_i$ = false
 7. *start* = 0
 8. Do
 9.         $v$ = ChooseVertex($D$, ***uniquely***, *start*)
10.         if $v \neq -1$
11.                 remove $v$ from $D$
12.                 Adjust($D$, $v$, ***coveredby***, ***uniquely***)
13. Until $v = -1$
14. Return $D$
```

If we disregard the time taken by line 7 in `ChooseVertex`, it is not hard to see that the total time spent in the rest of `ChooseVertex` throughout the execution of the algorithm is $O(n)$. This is because $start$ is originally set to 0 and gets incremented each time `ChooseVertex` is called, until it reaches n; the total time spent in one call of `ChooseVertex` is proportional to the amount that *start* is incremented during the call. In the worst case, the time taken by line 7 could be $O(n)$ each time it is called. Thus in the worst case the time required by `ChooseVertex` throughout the algorithm is $O(nd)$, although in fact this worst case is rarely achieved.

The analysis of `Adjust` is similar to that of `AdjustWeights` in the previous algorithm. The total time spent in `Adjust` is $O(m)$. The sort procedure takes $O(n \log n)$ time. Thus the total time taken by the algorithm is $O(nd + m + n \log n) \leq O(n^2)$ in the worst case.

2.3. *Greedy_Ran*. The algorithm `Greedy˙Ran` is similar to `Greedy`, with the only difference being the way in which the vertices to be added to the dominating set are selected. The probability that any one eligible vertex is chosen at a given iteration is proportional to the number of additional vertices it would cover. Thus vertices which cover less than the maximum number of additional vertices can still be chosen, but their chance of being chosen is less than that of the vertices which would cover the greatest number of additional vertices.

```
**ChooseVertex**(Vector ***weight***)
1. Return vertex $v_i$ with probability $weight_i / \sum_j weight_j$
```

The complexity behavior is similar to that of `Greedy`.

2.4. *Greedy_Vote*. The algorithm `Greedy˙Vote` does a more sophisticated job of efficiently choosing which vertices to place into the dominating set. As in algorithms `Greedy` and `Greedy˙Ran`, we want to assign more weight to the vertices which, if added to the dominating set, would cover more vertices. However, we also pay attention to the specific vertices which would get covered, and to whether or not they could be covered in some alternate way. For example, if a vertex x would cover vertex y if x were added to the dominating set, and no other vertex (except y itself) would cover y, then x has a stronger reason for being added to the dominating set than it would for potentially covering a vertex z which has high degree and may therefore end up being covered by many other vertices.

In order to implement this idea, we define $vote(v) = 1/(1 + degree(v))$ for each vertex v. The quantity $1 + degree(v)$ represents the number of vertices which could cover v (including itself). Then for each vertex v, define $weight(v)$ to be the sum of all the quantities $vote(w)$ such that w is a vertex which has not yet been covered, and which v could cover. A vertex v with maximum value of $weight(v)$ is chosen at each iteration.

The function `ChooseVertex` for this algorithm is the same as the one for `Greedy`.

```
AdjustWeights(Graph $G$, Vector weight, Vector covered,
Vector vote, Vertex $v_i$)
 1. $weight_i$ = 0
 2. for each neighbor $v_j$ of $v_i$ such that $weight_j > 0$
 3.         if not $covered_i$ decrement $weight_j$ by $vote_i$
 4.         if not $covered_j$
 5.              $covered_j$ = true
 6.              decrement $weight_j$ by $vote_j$
 7.              for each neighbor $v_k$ of $v_j$
 8.                   if $weight_k > 0$, decrement $weight_k$ by $vote_j$
 9. $covered_i$ = true
```

```
**Greedy_Vote**(Graph $G$)
 1. $D = \emptyset$
 2. For all vertices $v_i$
 3.         $vote_i = 1/(1 + degree(v_i))$
 4.         $covered_i$ = false
 5.         $weight_i = vote_i$
 6. For all vertices $v_i$
 7.         For all neighbors $v_j$ of $v_i$
 8.                 add $vote_j$ to $weight_i$
 9. Do
10.         $v$ = ChooseVertex(**weight**)
11.         if $v \neq -1$
12.                 add $v$ to $D$
13.                 AdjustWeights($G$, **weight**, **covered**, **vote**, $v$)
14. Until $v = -1$
15. Return $D$
```

The complexity analysis is similar to that for `Greedy`, except that the initialization of the weights now takes $O(m)$ instead of $O(n)$ time. The complexity of the whole algorithm is still $O(nd + m) \leq O(n^2)$.

2.5. *Greedy_Vote_Gr.* Finally, the algorithm `Greedy˙Vote˙Gr` is a version of `Greedy˙Vote` where we also apply a local search procedure. This variant was inspired by the local search used in the GRASP program for maximum clique designed by Feo et al. [2]. Specifically, once the algorithm `Greedy˙Vote` terminates, we do an exhaustive search to determine whether it is possible to remove any two vertices from D, and replace them with either one or no vertices, while still retaining a dominating set. Whenever we succeed in decreasing the size of D in this manner, the local search procedure is called once again on the smaller set D.

In the following pseudo-code, $coveredby_i$ is set equal to the number of vertices in D that cover v_i.

```
**Local**(Graph $G$, Set $D$)
 1. For each vertex $v_i$
 2.         $coveredby_i = 0$
 3. For each vertex $w_i \in D$
 4.         increment $coveredby_i$
 5.         For each neighbor $v_j$ of $w_i$
 6.                 increment $coveredby_j$
 7. For each pair $v_i, v_j \in D$
 8.         $U = \emptyset$
 9.         For each vertex $v_k$
10.                 $covby = coveredby_k$
11.                 if $v_i$ covers $v_k$ decrement $covby$
```

```
12.                 if $v_j$ covers $v_k$ decrement *covby*
13.                 if *covby* $= 0$ add $v_k$ to $U$
14.         If $U = \emptyset$
15.                 remove $v_i$ and $v_j$ from $D$
16.                 Return Local($G$, $D$)
17.         else for each vertex $v_k$
18.                 if $v_k$ covers all vertices in $U$
19.                     remove $v_i$ and $v_j$ from $D$ and add $v_k$ to $D$
20.                     Return Local($G$, $D$)
21. Return $D$
```

The complexity analysis of this function is not straightforward because of the recursive calls to `Local` (which in practice in our experiments were limited to at most two or three). Without taking into account the recursive calls, the code in `Local` has complexity $O(d^2n^2)$ in the worst case. (Note that line 18 may require $O(n)$ time in the worst case, although the worst case is very unlikely.) In practice the calls to `Local` did slow down the algorithm considerably although they also improved the size of the dominating set returned. This is examined more fully in Section 4.

2.6. *General Remarks.* Because random choices are involved in these algorithms, each algorithm was run several times on the same graph, typically for 1000 iterations, and the best result from all iterations was chosen as the answer. In addition, if the domination number of the graph was known (as it was for the graphs produced by the test case constructors), then the run was stopped if this number was found during an iteration.

3. Test Case Construction.

As explained in the Introduction, we used a test case construction procedure to produce graphs with known domination numbers and given densities. The density of a graph is defined to be the number of edges in the graph divided by the maximum number of edges a graph with the same number of vertices can have ($n(n-1)/2$ for an undirected graph with n vertices).

The general outline of the construction algorithm is as follows. Let V be the set of vertices, and $|V| = n$. Suppose we want to construct a graph with density p and domination number d. Partition V into d nonempty subsets $V_1, V_2, \ldots, V_d$. Choose vertices $x_i, y_i \in V_i$ for each i. (x_i and y_i need not be distinct.) Add edges joining x_i to every other vertex in V_i. Finally, add additional edges to achieve the desired density, while making sure that y_i is not connected to any vertex outside of V_i, for each i. This restriction ensures that the domination number of the graph will be exactly d.

The above procedural description is very general. The actual distribution of graphs to be obtained depends on implementation choices. From previous experience with the maximum clique problem (see [5] and [7]) we suspected that certain choices would result in experimentally harder graphs. For example, in our construction we made the sizes of the V_i as uniform as possible. For ease of implementation we also decided to have $x_i \neq y_i$ for each i.

Notice that if the additional edges (which are added to achieve the desired density) are added totally at random, then the y_i vertices will tend to have lower degrees. This circumstance might make the graphs easier to solve, especially for greedy-type algorithms which look for vertices with high degrees to put in the dominating set.

In previous work with the maximum clique problem (or the equivalent minimum vertex cover problem) we found that it was possible to adjust the edge distribution so that the average degree for vertices in the maximum clique would be equal to the average degree for the rest of the vertices in the graph [5], and thus achieve "harder" graphs.

We tried a similar approach for the dominating set problem. Here we have three types of vertices. Let $X = \{x_1, x_2, \ldots, x_d\}$, $Y = \{y_1, y_2, \ldots, y_d\}$, and $Z = V - X - Y$. Unfortunately for most parameter combinations of interest it is actually not possible to make the average degrees of vertices in X, Y, and Z equal. In addition, since each vertex in Y is allowed to be adjacent to such a small set of vertices, requiring that the degrees of vertices in Y be as high as those of other vertices in the graph may actually constrain the graphs in such a way as to make them easier to solve. Because of these factors we experimented with two different construction procedures which are described as follows.

Let p be the desired density of the graph. When choosing the additional edges to achieve the desired density, not all edges are eligible to be chosen (because of the restrictions on the y_i vertices). So each eligible edge should be chosen with some probability $p' > p$. Specifically, it can be checked that the total number of eligible edges is $\mathit{MaxEdges} = (n^2+n-2dn+d^2-d)/2$ and therefore $p' = p*n(n-1)/(2*\mathit{MaxEdges})$.

With the first generator, we simply attempt to ensure that the degrees of vertices in X are not higher than those of other vertices. Let Max_1 denote the number of eligible edges that are incident on some vertex in X, and let $\mathit{Max}_2 = \mathit{MaxEdges} - \mathit{Max}_1$. Note that $n - d$ edges incident on vertices in X are always put into the graph. The first construction algorithm, which we call `Gend1`, puts additional such edges into the graph with probability $(p'\mathit{Max}_1 - (n-d))/(\mathit{Max}_1 - (n-d))$. Thus the expected number of edges in the graph which are incident on vertices in X will be $p'\mathit{Max}_1$, as desired. Other potential edges are put into the graph with probability p'.

The second construction algorithm, called `Gend2`, tries to ensure that the degree of vertices in Y is not too low. We concentrate here on the addition of edges which are not incident on vertices in X. Out of all these edges, some are incident on vertices in Y and some are not. Let Max_{2a} denote the number of potential edges which are not incident on vertices in X but are incident on some vertex in Y. Let $\mathit{Max}_{2b} = \mathit{Max}_2 - \mathit{Max}_{2a}$. We attempt to make the average degree of vertices in Y equal to the average degree of vertices in Z, but taking into account only the edges not incident on vertices in X. Let p'_a be the probability with which edges joining vertices in Y to vertices in Z will be added. Let p'_b be the probability with which edges joining vertices in Z will be added. We want

$$\frac{p'_a\mathit{Max}_{2a}}{d} = \frac{2p'_b\mathit{Max}_{2b} + p'_a\mathit{Max}_{2a}}{n-2d} = \frac{2p'\mathit{Max}_2}{n-d}.$$

Since $\mathit{Max}_{2a} = n - 2d$ and $\mathit{Max}_{2b} = \binom{n-2d}{2}$, after some algebra we derive that $p'_a = p'd(n-2d+1)/(n-d)$ and $p'_b = (n/d-3)p'_a/(n-2d-1)$. In practice, these formulas will sometimes give a value of p'_a larger than 1, which means that the average degrees cannot be equated; in these cases we set $p'_a = 1$.

Table 1. Results of running the algorithms on 400 vertex random graphs. (Each algorithm is run for 1000 iterations.)

		Greedy		Greedy_Rev		Greedy_Ran		Greedy_Vote		Greedy_Vote_Gr		
n	p	Avg.	Min.	Avg.	Min.	Avg.	Min.	Avg.	Min.	Avg.	Min.	Imp.
400	0.1	18.7	18	23.8	22	29.2	29	19.6	19	19.6	19	0
400	0.3	8	8	10.1	9	10.8	10	8	8	8	8	0
400	0.5	5	5	6.2	6	6	6	5	5	5	5	0

We found that both construction procedures were useful, but in different ways, in constructing challenging graphs for the algorithms being evaluated here. This is further discussed in the next section.

4. Experimental Results. We present in this section the results of running the different algorithms on random graphs and on graphs produced by the construction procedures described in the previous section. In the tables included in this section, each line corresponds to experiments performed on 10 different graphs having the specified parameters. Each algorithm was run for 1000 iterations on each graph.

The results of running the algorithms on random graphs are tabulated in Tables 1 and 2, for graphs with 400 and 800 vertices, respectively. Three different densities are considered: 0.1, 0.3, and 0.5. We found that higher density graphs would have very small domination numbers. For each parameter set, we give the average value returned by each algorithm over 10 different random graphs of that type, as well as the minimum value found from the 10 graphs. The last column gives the number of graphs for which the `Greedy˙Vote˙Gr` algorithm, using the local search procedure, gave an improvement over the `Greedy˙Vote` algorithm. It is clear from the results in Tables 1 and 2 that `Greedy` is the best performing algorithm, with `Greedy˙Vote` and `Greedy˙Vote˙Gr` close seconds. The algorithm `Greedy˙Vote˙Gr` does not appear to provide much advantage at all over `Greedy˙Vote`, and considering that it requires much more time than `Greedy˙Vote`, it does not appear to be a useful algorithm for random graphs.

For $1 \leq d \leq n$, the formula $E(n, p, d) = \binom{n}{d}(1-(1-p)^d)^{(n-d)}$ gives the expected number of dominating sets of size d in a graph with n vertices and density p [3]. Interestingly enough, the smallest value of d for which $E(n, p, d) \geq 1$ is equal to the size of the smallest dominating set found by `Greedy` for any of the 10 graphs with n vertices and density p, for $n = 400$ and $p = 0.1$, 0.3, and 0.5. For the 800 vertex graphs, this value is at most one less than the smallest value found by `Greedy` and

Table 2. Results of running the algorithms on 800 vertex random graphs. (Each algorithm is run for 1000 iterations.)

		Greedy		Greedy_Rev		Greedy_Ran		Greedy_Vote		Greedy_Vote_Gr		
n	p	Avg.	Min.	Avg.	Min.	Avg.	Min.	Avg.	Min.	Avg.	Min.	Imp.
800	0.1	22.2	22	29.6	29	35.3	34	23.1	22	23	22	1
800	0.3	9.1	9	12	11	12.6	12	9.9	9	9.9	9	0
800	0.5	6	6	7.2	7	6.9	6	6	6	6	6	0

Table 3. Results of running the algorithms on 400 vertex graphs generated by Gend1. (Each algorithm is run for 1000 iterations.)

			Greedy		Greedy_Rev		Greedy_Ran		Greedy_Vote		Greedy_Vote_Gr		
n	*p*	Opt.	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Imp.
400	0.1	8	14.2	0	16.1	4	30.9	0	9.2	4	8	10	6
400	0.1	11	22.1	0	25.2	0	33.2	0	16.8	0	15.6	5	7
400	0.1	14	24	0	25.6	0	35.2	0	19	1	18.6	3	2
400	0.1	18	27.3	0	27.3	0	38.5	0	21.8	0	20.4	4	5
400	0.1	23	31.3	0	28.8	0	41.8	0	24.9	0	24.8	0	1
400	0.3	3	6.8	0	9.9	0	10.9	0	6.8	0	6	4	5
400	0.3	5	9	0	10.6	0	11.7	0	8.7	0	8.7	0	0
400	0.3	8	10.9	0	11.6	0	13.3	0	9.3	0	9.2	0	1
400	0.3	11	14	0	12.6	0	15.6	0	11.1	9	11	10	1
400	0.3	14	16.1	0	14.2	8	18.2	0	14	10	14	10	0
400	0.5	3	5	0	6.3	0	6.4	0	5	0	5	0	0
400	0.5	5	6	2	6.6	0	7	0	5.6	5	5.4	6	2
400	0.5	8	8.8	3	8.2	8	9	0	8.1	9	8	10	1
400	0.5	11	11.9	3	11	10	11.6	4	11	10	11	10	0

also `Greedy˙Vote`, for each of the three densities. We surmise from these results that both `Greedy` and `Greedy˙Vote` do a very good job on random graphs at these densities. When we turn to graphs having different properties from those of random graphs, however, the situation changes.

Tables 3 and 4 present results from running the algorithms on 400 vertex graphs generated by `Gend1` and `Gend2`, respectively. We constructed graphs having again densities 0.1, 0.3, and 0.5, and domination numbers in a range surrounding and including the smallest value returned by any of the algorithms on random graphs with the same

Table 4. Results of running the algorithms on 400 vertex graphs generated by Gend2. (Each algorithm is run for 1000 iterations.)

			Greedy		Greedy_Rev		Greedy_Ran		Greedy_Vote		Greedy_Vote_Gr		
n	*p*	Opt.	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Imp.
400	0.1	8	13.3	0	13.2	6	28.7	0	15	0	13.5	5	5
400	0.1	11	18.7	0	24.3	0	29	0	19.5	0	19.4	0	1
400	0.1	14	18.9	0	23.2	0	28.9	0	19.5	0	19.4	0	1
400	0.1	18	19.6	0	25.4	0	29.1	0	20	0	19.9	0	1
400	0.1	23	23.3	8	26	0	30.9	0	23.6	6	23.4	6	2
400	0.3	3	7.8	0	10.2	0	10.4	0	8.2	0	8.1	0	1
400	0.3	5	8	0	9.9	0	10.8	0	8	0	8	0	0
400	0.3	8	8.3	7	10.6	0	11.2	0	8.6	4	8.5	5	1
400	0.3	11	11.3	7	11.6	5	12.3	0	11	10	11	10	0
400	0.3	14	14.1	9	14.2	8	14.4	6	14	10	14	10	0
400	0.5	3	5	0	6	0	6	0	5	0	5	0	0
400	0.5	5	5.5	6	6.7	1	6.3	0	5.3	7	5.2	8	1
400	0.5	8	8.1	9	8	10	8	10	8.1	9	8	10	1
400	0.5	11	11.2	8	11	10	11	10	11	10	11	10	0

density: this value is underlined in the tables. Using domination numbers farther away (in either direction) from these values appears to make the graphs increasingly easier to solve. (This is similar to the situation encountered with the maximum clique problem [5], [7].) The domination number of the graph is indicated in the `Opt.` column. For each algorithm we give the average value returned for the 10 graphs, as well as the number of graphs for which the optimum (domination number) was found.

From the results in these tables we see that these graphs are more challenging for the algorithms than the random graphs. For the graphs generated by `Gend1`, the best performing algorithms are `Greedy˙Vote` and `Greedy˙Vote˙Gr`, with `Greedy˙Vote˙Gr` often improving on `Greedy˙Vote`, especially at density 0.1. The next best algorithm is `Greedy` for smaller domination numbers and `Greedy˙Rev` for larger domination numbers.

For the graphs generated by `Gend2`, on the other hand, it is `Greedy` which performs best for most of the graphs with density 0.1 and 0.3. For density 0.5, the results returned by all algorithms are fairly close, with `Greedy˙Vote˙Gr` having a slight edge.

The two construction procedures complement each other in a way. Note that there are four types of graphs in Table 3 for which none of the algorithms ever finds the optimal dominating set. There are six such types in Table 4, and only two of these overlap with the ones from the preceding table. Thus together `Gend1` and `Gend2` are able to construct graphs which are consistently difficult for all of the algorithms, for eight out of the fourteen graph types considered. These consist of all the graphs with density 0.1 except for the ones with the smallest domination number, the three graphs with density 0.3 which have the smallest domination numbers, and the graph with density 0.5 which has the smallest domination number.

The data in Table 5, corresponding to 800 vertex graphs generated by `Gend1`, presents the same type of pattern as that observed in Table 3. The data in Table 6, corresponding

Table 5. Results of running the algorithms on 800 vertex graphs generated by Gend1. (Each algorithm is run for 1000 iterations.)

			Greedy		Greedy_Rev		Greedy_Ran		Greedy_Vote		Greedy_Vote_Gr		
n	*p*	Opt.	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Imp.
800	0.1	8	13.9	0	8	10	37.8	0	8.7	5	8	10	5
800	0.1	11	26.3	0	30.5	0	40.3	0	23.7	0	22.8	1	2
800	0.1	14	28.1	0	31.4	0	41.9	0	23.5	0	22.4	2	2
800	0.1	18	30.5	0	31.2	0	45.4	0	23.7	0	22.3	4	5
800	0.1	$\underline{22}$	34.9	0	33.1	0	48.3	0	27.3	0	27.3	0	0
800	0.1	26	37.8	0	35.7	0	51.6	0	29	0	28.8	0	2
800	0.1	30	41	0	36.5	0	54.8	0	31.4	0	31.4	0	0
800	0.3	3	8.7	0	12.1	0	12.5	0	9	0	8.8	1	1
800	0.3	5	9.7	0	12.2	0	13.3	0	9.4	0	9.4	0	0
800	0.3	$\underline{9}$	12.5	0	12.7	0	15.8	0	10.4	0	10.4	0	0
800	0.3	13	16.4	0	14.9	0	18.6	0	13	10	13	10	0
800	0.3	18	21.4	0	18.4	7	22.6	0	18	10	18	10	0
800	0.5	3	6	0	7.3	0	7	0	6	0	6	0	0
800	0.5	$\underline{6}$	7.4	2	7.8	0	8.3	0	6.3	7	6.2	8	1
800	0.5	9	10.6	1	9	10	10.3	0	9.2	8	9	10	2
800	0.5	12	13.2	2	12	10	12.8	2	12.1	9	12	10	1

Table 6. Results of running the algorithms on 800 vertex graphs generated by Gend2. (Each algorithm is run for 1000 iterations.)

			Greedy		Greedy_Rev		Greedy_Ran		Greedy_Vote		Greedy_Vote_Gr		
n	*p*	Opt.	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Avg.	Opt. found	Imp.
800	0.1	8	10.2	0	8	10	35.4	0	11.7	0	9.5	9	9
800	0.1	11	22.3	0	29.5	0	35.4	0	23	0	22.8	0	2
800	0.1	14	22.2	0	29.1	0	35.2	0	23.1	0	23.1	0	0
800	0.1	18	22.4	0	30.2	0	35.8	0	23.3	0	23.3	0	0
800	0.1	$\underline{22}$	24.2	0	29.2	0	36.5	0	24.1	0	24.1	0	0
800	0.1	26	26.9	5	31.4	0	38.1	0	26.6	4	26.6	4	0
800	0.1	30	30.4	7	32.6	0	39.7	0	30.5	5	30.2	8	3
800	0.3	3	8.7	0	10.1	2	12.1	0	9.7	0	9.5	0	2
800	0.3	5	9	0	11.8	0	12.6	0	9.4	0	9.3	0	1
800	0.3	$\underline{9}$	10	1	11.6	0	13.4	0	9.9	1	9.8	2	1
800	0.3	13	13.7	4	14	2	14.7	0	13.2	8	13	10	2
800	0.3	18	18.1	9	18	10	18.5	5	18	10	18	10	0
800	0.5	3	5.9	0	7	0	7	0	5.9	0	5.9	0	0
800	0.5	$\underline{6}$	6.3	7	7.2	0	7.5	0	6.2	8	6.1	9	1
800	0.5	9	9.4	7	9.2	8	9.2	8	9.1	9	9	10	1
800	0.5	12	12.3	7	12	10	12	10	12	10	12	10	0

to 800 vertex graphs generated by Gend2, is also similar to that in Table 4, except that for the graphs with larger domination numbers and density 0.1 or 0.3, the results of Greedy, Greedy˙Vote, and Greedy˙Vote˙Gr are all very close, and in fact the Greedy˙Vote variations have a slight edge. Again, together Gend1 and Gend2 consistently produced graphs which none of the algorithms ever solved, for density 0.1 and all domination numbers listed except the smallest one, for the second and third smallest domination numbers with density 0.3, and for the smallest domination number with density 0.5.

As has been noted, we ran each algorithm for 1000 iterations, or until the domination number was found (for graphs with known domination number), and returned for each graph the lowest value found in any iteration. In Table 7 we have calculated for each density and each type of graph, the average iteration in which the smallest domination number is encountered. Note that in spite of the randomness injected into all of the algorithms, the smallest domination number is always found at the first iteration for Greedy˙Vote and Greedy˙Vote˙Gr, indicating that it is obviously not worthwhile running the algorithm for more than one iteration. Probably the weight computation using votes as explained in the description of Greedy˙Vote introduces enough variation in the weights such that the random choices play no role. Algorithms Greedy and Greedy˙Rev tend to find the best dominating set also at a relatively early iteration, except in some cases. It would appear for these algorithms that many fewer than 1000 iterations would be sufficient. The algorithm Greedy˙Ran seems to benefit the most from an increased number of iterations, and this is not surprising since it incorporates the most randomness; on the other hand this algorithm does not have very good performance in any case.

Table 7. Average iteration in which the best dominating set is found.

		Construction	Average iteration				
n	*p*	algorithm	Greedy	Greedy_Rev	Greedy_Ran	Greedy_Vote	Greedy_Vote_Gr
400	0.1	Random	33.5	49.9	560.1	1	1
400	0.1	Gend1	168.38	5.82	325.82	1	1
400	0.1	Gend2	23.04	8.04	332.5	1	1
400	0.3	Random	1	1.8	177.8	1	1
400	0.3	Gend1	12.16	2.68	273.52	1	1
400	0.3	Gend2	2.24	1.48	268.4	1	1
400	0.5	Random	1	1.1	173.4	1	1
400	0.5	Gend1	2.4	1.68	174.35	1	1
400	0.5	Gend2	1.1	1.4	115.08	1	1
800	0.1	Random	90.1	5.4	358.8	1	1
800	0.1	Gend1	223.24	5.87	415.97	1	1
800	0.1	Gend2	58.96	8.94	386.04	1	1
800	0.3	Random	2.6	3.3	178.3	1	1
800	0.3	Gend1	6.42	1.6	367.06	1	1
800	0.3	Gend2	12.92	1.64	292.66	1	1
800	0.5	Random	1	2	154.1	1	1
800	0.5	Gend1	1.83	1.25	235.18	1	1
800	0.5	Gend2	1.85	1.15	109.63	1	1

Finally, we consider the running times of the algorithms. Since in the previous runs the algorithm was stopped if the domination number was found, and since some algorithms benefited more than others from an increased number of iterations, we determined it would be a clearer comparison to run each algorithm for exactly 100 iterations, again on each of the graphs used above. The results are tabulated in Tables 8–13. The times are reported in seconds.

As expected, the running times increase with increased density and also with higher domination numbers. The running times for all algorithms except `Greedy˙Vote˙Gr` are fairly consistent; `Greedy` has the best running time per iteration, while `Greedy˙Rev` has the worst, except for `Greedy˙Vote˙Gr` of course which is considerably slower than the other algorithms, as was expected.

Given that `Greedy˙Vote` needs only one iteration, that its time per iteration is not much higher than that for `Greedy`, and that it often gives the best results or close to the best results, we conclude that it is the most useful algorithm, at least for the graphs studied here. The use of `Greedy˙Vote˙Gr` must be considered with care because of the extra time required and the meager benefits obtained much of the time. It seems reasonable to use `Greedy˙Vote˙Gr` only when it is really important to find the very smallest dominating set which it is feasible to obtain.

Table 8. Running times for 100 iterations of each algorithm on 400 vertex random graphs.

n	*p*	Greedy	Greedy_Rev	Greedy_Ran	Greedy_Vote	Greedy_Vote_Gr
400	0.1	0.22	0.33	0.32	0.25	5.60
400	0.3	1.14	1.47	1.19	1.29	2.41
400	0.5	2.39	2.84	2.41	2.58	3.15

Table 9. Running times for 100 iterations of each algorithm on 400 vertex graphs generated by Gend1.

n	*p*	Opt.	Greedy	Greedy_Rev	Greedy_Ran	Greedy_Vote	Greedy_Vote_Gr
400	0.1	8	0.24	0.40	0.33	0.26	1.21
400	0.1	11	0.25	0.34	0.34	0.26	5.10
400	0.1	14	0.25	0.33	0.32	0.26	5.37
400	0.1	<u>18</u>	0.25	0.32	0.32	0.25	7.94
400	0.1	23	0.26	0.31	0.34	0.26	9.09
400	0.3	3	1.14	1.46	1.20	1.27	2.11
400	0.3	5	1.14	1.45	1.19	1.28	2.52
400	0.3	<u>8</u>	1.14	1.44	1.18	1.28	2.77
400	0.3	11	1.17	1.39	1.18	1.28	3.24
400	0.3	14	1.17	1.37	1.22	1.29	4.24
400	0.5	3	2.39	2.81	2.42	2.58	3.09
400	0.5	<u>5</u>	2.39	2.78	2.43	2.59	3.23
400	0.5	8	2.41	2.71	2.45	2.60	3.68
400	0.5	11	2.44	2.68	2.47	2.64	4.46

5. Conclusions.

We have presented and analyzed several greedy type approximation algorithms for the dominating set problem. For random graphs the straightforward greedy strategy performs the best, and in fact finds dominating sets whose sizes are very close to the expected sizes of the smallest dominating sets in such graphs.

When we consider the graphs created by the test case generators, however, the simple greedy strategy does not perform as well. For these graphs the more sophisticated algorithm, `Greedy˙Vote`, does a better job of choosing the vertices to be added to the dominating set. However, the test case constructors are capable of producing graphs which are challenging even for this algorithm.

Other variations, involving more randomness, or removing rather than adding vertices to the dominating set, were found not to be worthwhile.

We have also seen that performing experiments on the graphs produced by the test case generators provides much more information about the effectiveness of the algorithms

Table 10. Running times for 100 iterations of each algorithm on 400 vertex graphs generated by Gend2.

n	*p*	Opt.	Greedy	Greedy_Rev	Greedy_Ran	Greedy_Vote	Greedy_Vote_Gr
400	0.1	8	0.21	0.39	0.30	0.26	3.43
400	0.1	11	0.23	0.34	0.30	0.25	6.25
400	0.1	14	0.23	0.35	0.30	0.25	5.84
400	0.1	<u>18</u>	0.24	0.34	0.30	0.26	6.14
400	0.1	23	0.25	0.32	0.31	0.26	8.23
400	0.3	3	1.11	1.45	1.17	1.26	2.57
400	0.3	5	1.13	1.47	1.19	1.27	2.41
400	0.3	<u>8</u>	1.14	1.45	1.18	1.27	2.62
400	0.3	11	1.17	1.43	1.20	1.35	3.27
400	0.3	14	1.18	1.39	1.22	1.32	4.27
400	0.5	3	2.38	2.83	2.41	2.58	3.21
400	0.5	<u>5</u>	2.38	2.83	2.41	2.57	3.29
400	0.5	8	2.41	2.73	2.41	2.59	3.85
400	0.5	11	2.43	2.69	2.43	2.61	4.57

Table 11. Running times for 100 iterations of each algorithm on 800 vertex random graphs.

n	*p*	Greedy	Greedy_Rev	Greedy_Ran	Greedy_Vote	Greedy_Vote_Gr
800	0.1	1.87	2.43	2.06	2.01	25.90
800	0.3	7.12	8.06	7.21	7.61	11.72
800	0.5	12.06	13.68	12.18	12.90	14.54

Table 12. Running times for 100 iterations of each algorithm on 800 vertex graphs generated by Gend1.

n	*p*	Opt.	Greedy	Greedy_Rev	Greedy_Ran	Greedy_Vote	Greedy_Vote_Gr
800	0.1	8	1.79	4.01	2.08	1.91	4.58
800	0.1	11	1.89	2.42	2.08	2.01	26.62
800	0.1	14	1.89	2.41	2.09	2.01	23.30
800	0.1	18	1.90	2.41	2.11	2.00	25.86
800	0.1	22	1.94	2.38	2.13	2.03	32.35
800	0.1	26	1.95	2.37	2.14	2.05	44.08
800	0.1	30	1.96	2.39	2.17	2.06	42.83
800	0.3	3	7.10	8.11	7.23	7.61	11.25
800	0.3	5	7.12	8.12	7.24	7.62	11.33
800	0.3	9	7.15	7.97	7.28	7.64	12.21
800	0.3	13	7.21	7.88	7.32	7.67	14.82
800	0.3	18	7.29	7.75	7.36	7.72	21.27
800	0.5	3	12.09	13.63	12.19	12.94	14.83
800	0.5	6	12.14	13.65	12.23	12.97	15.04
800	0.5	9	12.20	13.38	12.32	13.07	16.73
800	0.5	12	12.22	13.28	12.35	13.09	18.93

Table 13. Running times for 100 iterations of each algorithm on 800 vertex graphs generated by Gend2.

n	*p*	Opt.	Greedy	Greedy_Rev	Greedy_Ran	Greedy_Vote	Greedy_Vote_Gr
800	0.1	8	1.82	4.09	2.10	1.97	6.55
800	0.1	11	1.86	2.47	2.05	2.01	29.49
800	0.1	14	1.86	2.46	2.05	2.02	24.25
800	0.1	18	1.84	2.42	2.08	2.01	24.24
800	0.1	22	1.87	2.42	2.06	2.01	25.34
800	0.1	26	1.88	2.40	2.07	2.03	30.98
800	0.1	30	1.93	2.42	2.09	2.06	45.58
800	0.3	3	7.13	8.94	7.26	7.62	11.91
800	0.3	5	7.15	8.16	7.25	7.61	11.51
800	0.3	9	7.15	8.16	7.26	7.62	11.86
800	0.3	13	7.17	7.98	7.28	7.65	14.74
800	0.3	18	7.27	7.83	7.30	7.71	21.01
800	0.5	3	12.05	13.73	12.32	12.88	14.41
800	0.5	6	12.46	13.60	12.69	12.91	14.68
800	0.5	9	12.51	13.34	12.48	12.96	16.33
800	0.5	12	12.30	13.23	12.29	13.01	18.82

than would be obtained from testing only on random graphs. This testing also provided the impetus for the development of the algorithm `Greedy˙Vote`.

In future work it would be interesting to attempt more variations on the test case construction procedure, to see if different graph distributions with different experimental properties can be produced.

References

[1] Sanjeev Arora and Carsten Lund, Hardness of Approximations, in D. S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, pages 399–446, PWS, Boston, MA, 1997.

[2] Thomas S. Feo, Mauricio G. C. Resende, and Stuart H. Smith, A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set, *Operations Research*, vol. 42, no. 5, September–October 1994, pages 860–978.

[3] Sotiris E. Nikoletseas and Paul G. Spirakis, Near-Optimal Dominating Sets in Dense Random Graphs in Polynomial Expected Time, in *Graph-Theoretic Concepts in Computer Science* (*Utrecht*, 1993), Lecture Notes in Computer Science, vol. 790, Springer-Verlag, Berlin, 1994.

[4] Abhay K. Parekh, Analysis of a Greedy Heuristic for Finding Small Dominating Sets in Graphs, *Information Processing Letters*, vol. 39, 1991, pages 237–240.

[5] Laura A. Sanchis, Test Case Construction for the Vertex Cover Problem, in N. Dean and G. E. Shannon, editors, *Computational Support for Discrete Mathematics*, pages 315–326, American Mathematical Society, Providence, RI, 1994.

[6] Laura A. Sanchis, Generating Hard and Diverse Test Sets for NP-Hard Graph Problems, *Discrete Applied Mathematics*, vol. 58, 1995, pages 35–66.

[7] Laura A. Sanchis and Arun Jagota, Some Experimental and Theoretical Results on Test Case Generators for the Maximum Clique Problem, *INFORMS Journal on Computing*, vol. 8, no. 2, Spring 1996, pages 87–102.