FAST LOCAL SEARCH FOR THE MAXIMUM INDEPENDENT SET PROBLEM

DIOGO V. ANDRADE, MAURICIO G.C. RESENDE, AND RENATO F. WERNECK

ABSTRACT. Given a graph $G = (V, E)$, the independent set problem is that of finding a maximum-cardinality subset S of V such that no two vertices in S are adjacent. We introduce two fast local search routines for this problem. The first can determine in linear time whether a maximal solution can be improved by replacing a single vertex with two others. The second routine can determine in $O(m\Delta)$ time (where Δ is the highest degree in the graph) whether there are two solution vertices than can be replaced by a set of three. We also present a more elaborate heuristic that successfully applies local search to find near-optimum solutions to a wide variety of instances. We test our algorithms on instances from the literature as well as on new ones proposed in this paper.

1. INTRODUCTION

The *maximum independent set* (MIS) problem takes a connected, undirected graph $G = (V, E)$ as input, and tries to find the largest subset S of V such that no two vertices in S have an edge between them. Besides having several direct applications [3], MIS is closely related to two other well-known optimization problems. To find the *maximum clique* (the largest complete subgraph) of a graph G, it suffices to find the maximum independent set of the complement of G. Similarly, to find the *minimum vertex cover* of $G = (V, E)$ (the smallest subset of vertices that contains at least one endpoint of each edge in the graph), one can find the maximum independent set S of V and return $V \setminus S$. Because these problems are NP-hard [12], for most instances one must resort to heuristics to obtain good solutions within reasonable time.

Most successful heuristics [2, 8, 9, 10, 13, 15, 16] maintain a single current solution that is steadily modified by very simple operations, such as individual insertions, individual deletions, and swaps (replacing a vertex by one of its neighbors). In particular, many algorithms use the notion of *plateau search*, which consists in performing a randomized sequence of swaps. A swap does not improve the solution value by itself, but with luck it may cause a non-solution vertex to become free, thus allowing a simple insertion to be performed. Grosso et al. [9] have recently obtained exceptional results in practice by performing plateau search almost exclusively. Their method (as well as several others) occasionally applies a more elaborate

Date: June 29, 2010.

Key words and phrases. Maximum independent set, local search, iterated local search, algorithm engineering.

Most of this work was done while D.V. Andrade was at Rutgers University and part was done while R.F. Werneck was at Princeton University.

AT&T Labs Research Technical Report.

operation for diversification purposes, but spends most of its time performing basic operations (insertions, deletions, and swaps), often chosen at random.

This paper expands the set of tools that can be used effectively within metaheuristics. We present a fast (in theory and practice) implementation of a natural local search algorithm. It is based on (1,2)-swaps, in which a single vertex is removed from the solution and replaced by two others. We show that, given any maximal solution, one can find such a move (or prove that none exists) in linear time. In practice, an incremental version runs in sublinear time. The local search is more powerful than simple swaps, but still cheap enough for effective use within more elaborate heuristics. We also discuss a generalization of this method to deal with (2,3)-swaps, which consist of two removals followed by three insertions. We can find such a move (or prove that none exists) in $O(m\Delta)$ time, where Δ is the highest degree in the graph.

Another contribution is a more elaborate heuristic that illustrates the effectiveness of our local search. Although the algorithm is particularly well-suited for large, sparse instances, it is competitive with previous algorithms on a wide range of instances from the literature. As an added contribution, we augment the standard set of instances from the literature with new (and fundamentally different) instances, never previously studied in the context of the MIS problem.

This paper is organized as follows. Section 2 establishes the notation and terminology we use. Our local search algorithms are described in Section 3. Section 4 illustrates how they can be applied within a more elaborate heuristic. We report our experimental results in Section 5. Finally, concluding remarks are presented in Section 6.

2. Basics

The input to the MIS problem is a connected graph $G = (V, E)$, with $|V| = n$ and $|E| = m$. We assume that vertices are labeled from 1 to n. We use the adjacency list representation: each vertex keeps a list of all adjacent vertices.

A solution S is simply a subset of V in which no two vertices are adjacent. The *tightness* of a vertex $v \notin S$, denoted by $\tau(v)$, is the number of neighbors of v that belong to S. We say that a vertex is *k-tight* if it has tightness k. The tightnesses of all vertices can be computed in $O(m)$ time: initialize all values to zero, then traverse the adjacency list of each solution vertex v and increment $\tau(w)$ for every arc (v, w). Vertices that are 0-tight are called *free*.

A *(j, k)-swap* consists of removing j vertices from a solution and inserting k vertices into it. For simplicity, we refer to a (k, k)-swap as a *k-swap* (or simply a *swap* when $k = 1$), and to a $(k - 1, k)$-swap as a *k-improvement*. In particular, the main algorithms described in this paper (Section 3) are fast implementations of 2-improvements (Section 3.1) and 3-improvements (Section 3.2). We use the term *move* to refer to a generic (j, k)-swap.

We say a solution is *k-maximal* if no k-improvement is possible. In particular, a 1-maximal (or simply *maximal*) solution has no free vertices.

Solution representation. Being just a subset of V, there are several ways of representing a solution S. Straighforward representations, such as lists or incidence vectors, easily allow insertions and deletions to be performed in constant time. Unfortunately, more complex operations (such as finding a free vertex or determining whether a solution is maximal) can take as much as $\Theta(m)$ time. We therefore

opt for a slightly more complicated (but more powerful) representation within our algorithms.

We represent a solution S as a permutation of all vertices that partitions them into three blocks: first the $|S|$ vertices in the solution, then the free vertices, and finally the non-solution vertices that are not free. The order among vertices within a block is irrelevant. The sizes of the first two blocks are stored explicitly. In addition, the data structure maintains, for each vertex, its tightness (which allows us to determine when a vertex becomes free) and its position in the permutation. Keeping this position allows a vertex to be moved between any two blocks in constant time. (For example, to move a vertex v from the first block to the second, we first swap v with the last vertex of the first block, then change the boundary between the blocks to make v the first vertex of the second block.)

Our data structure must be updated whenever a vertex v is inserted into or removed from S. To insert a vertex v, we must first move it to the first block of the permutation. Then, for each neighbor w of v, we increase $\tau(w)$ by one and move w to the third block (if it was previously free). To remove a vertex v, we first move it to the second block (since it becomes free). Then, for each neighbor w of v, we decrease $\tau(w)$ by one and, if its new tightness is zero, we move w from the third to the second block.

This means that inserting or deleting a vertex v takes time proportional to the degree of v, which we denote by $\deg(v)$. While this is more expensive than in simpler solution representations (such as lists or incidence vectors), this data structure allows several powerful operations to be performed in constant time. The tightness of any vertex, for example, is explicitly maintained. To determine whether the solution is maximal, it suffices to check if the second block is empty. To find out whether $v \in S$, we only need to check if it is on the first block. Finally, we can pick a vertex uniformly at random within any of the three blocks, since each block is represented as a contiguous array of known size. This operation will be especially useful for metaheuristics.

3. Local Search

We now present our two main algorithmic contributions. We start with 2-improvements (in Section 3.1), then discuss 3-improvements (in Section 3.2).

3.1. **Finding 2-Improvements.**

Our main local search algorithm is based on 2-improvements. These natural operations have been studied before (see e.g. [7]); our contribution is a faster algorithm that implements them. Given a maximal solution S, we would like to replace some vertex $x \in S$ with two vertices, v and w (both originally outside the solution), thus increasing the total number of vertices in the solution by one.

Before we get to the actual algorithm, we establish some important facts about any valid 2-improvement that removes a vertex x from the solution. First, both v and w (the vertices inserted) must be neighbors of x, otherwise the original solution would not be maximal. Moreover, both v and w must be 1-tight, or else the new solution would be valid. Finally, v and w must not be adjacent to each other.

Our algorithm processes each vertex $x \in S$ in turn, using the data structures mentioned in the previous section. First, temporarily remove x from S, creating a solution S'. If S' has less than two free vertices, stop: there is no 2-improvement involving x. Otherwise, for each neighbor v of x that is free in S', insert v into S'

and check if the new solution (S'') contains a free vertex w. If it does, inserting w leads to a 2-improvement; if it does not, remove v from S'' (thus restoring S') and process the next neighbor of x. If no improvement is found, reinsert x into the solution.

This algorithm clearly finds a 2-improvement if there is one. We can determine its running time by bounding the total number of vertex scans performed. A vertex $x \in S$ is scanned only when x itself is the candidate for removal, when it is removed from and (possibly) reinserted into the solution. A non-solution vertex v is only scanned it if is 1-tight: it is removed from the solution (and possibly reinserted) when its unique solution neighbor x is processed as a candidate for removal. This means every vertex in the graph is scanned $O(1)$ times; equivalently, every edge is visited $O(1)$ times. We have thus proven the following:

Theorem 1. *Given a maximal solution S, one can find a 2-improvement (or prove that none exists) in $O(m)$ time.*

3.1.1. *A Direct Implementation.* Although our algorithm is relatively simple as described, some of its complexity is in fact hidden by the solution representation. For example, we rely on the representation's ability to determine in constant time whether there is a free vertex, and to produce one such vertex if it exists. To answer such queries efficiently, the algorithm must temporarily modify the solution by performing insertions and deletions, which are somewhat expensive. In practice, we can save some constant factors in the running time with a more direct implementation. Although it may appear to be more complex, it does essentially the same thing.

The implementation assumes the neighbors of any vertex v are sorted by increasing order of label in v's adjacency list. This order can be enforced in linear time (for all vertices) in a preprocessing step with the application of radix sort to all edges.

As in the original algorithm, we process each solution vertex x in turn. To process x, we first build a list $L(x)$ consisting of all 1-tight neighbors of x, also sorted by label. If $L(x)$ has fewer than two elements, we are done with x: it is not involved in any 2-improvement. Otherwise, we must find, among all candidates in $L(x)$, a pair $\{v, w\}$ such that there is no edge between v and w. We do this by processing each element $v \in L(x)$ in turn. For a fixed candidate v, we check if there is a vertex $w \in L(x)$ (besides v) that does not belong to $A(v)$, the adjacency list of v. Since both $L(x)$ and $A(v)$ are sorted by vertex identifier, this can be done by traversing both lists in tandem. All elements of $L(x)$ should appear in the same order within $A(v)$; if there is a mismatch, the element of $L(x)$ missing in $A(v)$ is the vertex w we are looking for.

We claim that this algorithm finds a valid 2-improvement (or determines that none exists) in $O(m)$ time. This is clearly a valid bound on the time spent scanning all vertices (i.e., traversing their adjacency lists), since each vertex is scanned at most once. Each solution vertex x is scanned to build $L(x)$ (the list of 1-tight neighbors), and each 1-tight non-solution vertex v is scanned when its only solution neighbor is processed. (Non-solution vertices that are not 1-tight are not scanned at all.) We still need to bound the time spent traversing the $L(x)$ lists. Each list $L(x)$ may be traversed several times, but each occurs in tandem with the traversal of the adjacency list $A(v)$ of a distinct 1-tight neighbor v of x. Unless the traversal finds a valid swap (which occurs only once), traversing $L(x)$ costs no more than

$O(\deg(v))$, since each element of $L(x)$ (except v) also occurs in $A(v)$. This bounds the total cost of such traversals to $O(m)$.

3.1.2. *Incremental Version.* A typical local search procedure does not restrict itself to a single iteration. If a valid 2-improvement is found, the algorithm will try to find another in the improved solution. This can of course be accomplished in linear time, but we can do better with an *incremental* version of the local search, which uses information gathered in one iteration to speed up later ones.

The algorithm maintains a set of *candidates*, which are solution vertices that might be involved in a 2-improvement. So far, we have assumed that all solution vertices are valid candidates, and we test them one by one. After a move, we would test all vertices again. Clearly, if we establish that a candidate x cannot be involved in a 2-improvement, we should not reexamine it unless we have good reason to do so. More precisely, when we "discard" a candidate vertex x, it is because it does not have two independent 1-tight neighbors. Unless at least one other neighbor of x becomes 1-tight, there is no reason to look at x again.

With this in mind, we maintain a list of candidates that is updated whenever the solution changes. Any move (including a 2-improvement) can be expressed in terms of insertions and deletions of individual vertices. When we insert a vertex v into the solution, its neighbors are the only vertices that can become 1-tight, so we simply (and conservatively) add v to the list of candidates. When a vertex x is removed from the solution, the update is slightly more complicated. We must traverse the adjacency list of x and look for vertices that became 1-tight due to the removal of x. By definition, each such vertex v will have a single neighbor y in the solution; y must be inserted into the candidate list. We can find the solution vertex adjacent to each 1-tight neighbor v in constant time, as long as we maintain with each non-solution vertex the list of its solution neighbors.[1] Therefore, we could still update the candidate list after removing x in $O(\deg(x))$ time. For simplicity, however, we do not maintain the auxiliary data structures in our implementation, and explicitly scan each 1-tight neighbor of x.

Although we have framed our discussion in terms of 2-improvements, these updates can of course be performed for any sequence of removals and/or insertions. As we will see, this means we can easily embed the incremental local search algorithm into more elaborate heuristics.

Once invoked, the local search itself is quite simple: it processes the available candidates in random order. If a candidate x leads to a 2-improvement, we perform the move and update the list of candidates accordingly; otherwise, x is simply removed from the list of candidates. The local search stops when the list of candidates becomes empty, indicating that a local optimum has been reached.

3.1.3. *Maximum Clique.* Although our experiments focus mainly on the MIS problem, it is worth mentioning that one can also implement a linear-time 2-improvement algorithm for the maximum clique problem. Note that simply running the algorithm above on the complement of the input is not enough to ensure linear time, since the complement may be much denser than the original graph.

[1]Standard doubly-linked lists will do, but updating them is nontrivial. In particular, when removing a vertex x from the solution, we must be able to remove in constant time the entry representing x in the list of each neighbor v. This can be accomplished by storing a pointer to that entry together with the arc (x, v) in x's adjacency list.

Given a maximal clique C, we must determine if there is a vertex $x \in C$ and two vertices $v, w \notin C$ such that the removal of x and the insertion of v and w would lead to a larger clique. Such a move only exists if the following holds: (1) v and w are neighbors; (2) both v and w are adjacent to all vertices in $C \setminus \{x\}$; and (3) neither v nor w are adjacent to x (or else C would not be maximal). For a vertex v with tightness $|C| - 1$, define its *missing neighbor* $\mu(v)$ as the only solution vertex to which v is not adjacent. There is a 2-improvement involving $v \notin C$ if it has a neighbor $w \notin C$ such that $\tau(w) = |C| - 1$ and $\mu(w) = \mu(v)$. Knowing this, the local search procedure can be implemented in $O(m)$ time as follows. First, determine the tightness of all vertices, as well as the missing neighbors of those that are $(|C| - 1)$-tight. Then, for each $(|C| - 1)$-tight vertex v, determine in $O(\deg(v))$ time if it has a neighbor w satisfying the conditions above.

3.2. **Finding 3-Improvements.** We now discuss a potentially more powerful local search algorithm for the minimum independent set problem. Given a 2-maximal solution S (i.e., one in which no 2-improvement can be performed), we want to find a valid 3-improvement. In other words, we look for vertices $\{x, y\} \subset S$ and $\{u, v, w\} \subset V \setminus S$ such that we can turn S into a better (valid) solution S' by removing x and y and inserting u, v, and w. To devise an efficient algorithm for this problem, we first establish some useful facts about the vertices involved in such a move.

Lemma 1. *Edges (u, v), (u, w), and (v, w) do not belong to the graph.*

Proof. If they did, the new solution S' would not be valid. □

Lemma 2. *Each vertex in $\{u, v, w\}$ is adjacent to x, y, or both, but to no other vertex in S.*

Proof. Consider vertex u (one can reason about v and w similarly). If u is adjacent to any other vertex $z \in S$ besides x and y, the new solution S' would not be valid. If u were adjacent to neither x nor y, the original solution S would not be maximal, which contradicts our initial assumption. Therefore, u must be adjacent to either x or y (or both). □

Lemma 3. *If any two vertices in $\{u, v, w\}$ are 1-tight, they must have different neighbors in $\{x, y\}$.*

Proof. By contradiction (and without loss of generality), suppose that both v and w are 1-tight and adjacent to x. Then a 2-improvement replacing x by v and w would be valid, which contradicts our assumption that S is 2-maximal. □

Lemma 4. *At least one vertex in $\{u, v, w\}$ must be adjacent to both x and y.*

Proof. Immediate from Lemmas 2 and 3. □

Together, these lemmas imply that any valid 3-improvement involves vertices x, y, u, v, and w such that:

(1) $x, y \in S$ and $u, v, w \notin S$;
(2) u is 2-tight and adjacent to both x and y;
(3) v is adjacent to x (and maybe y) and to no other vertex in S;
(4) w adjacent to y (and maybe x) and to no other vertex in S;
(5) u, v, and w are not adjacent to one another.

To find a 3-improvement, our algorithm processes each 2-tight vertex u in turn, as follows. Let x and y be u's neighbors in the solution. Our goal is to find vertices v and w satisfying conditions 3, 4, and 5 above. To accomplish this, first temporarily remove x and y from the solution and insert u (which is now free). Let S' be the new solution. If they exist, v and w (as defined in the constraints above) must be free in S'. Therefore, if S' less than two free vertices, we are done with u; there is no pair $\{v, w\}$ leading to a valid 3-improvement. Otherwise, for each free neighbor v of x, temporarily add it to S', creating S''. If S'' is not maximal, adding any free vertex w will create a valid solution (thus leading to a 3-improvement). Otherwise, remove v and try another free neighbor of x.

Running time. It is easy to see that the algorithm above will find a valid 3-improvement if there is one. We must now bound the its total running time. When processing a 2-tight vertex u, only u, x, y, and the newly-free neighbors v of x will be scanned (no more than twice each). Note that this includes scans performed during insertions and deletions. Because these are all distinct vertices, the total time for all these scans $O(m)$. Since there are $O(n)$ 2-tight vertices u, the total time to find a valid 3-improvement (or prove that no such move exists) is $O(mn)$.

We can obtain a tighter bound in terms of the maximum degree Δ in the graph. Using the notation above, we can make the following observations. Every 2-tight vertex is scanned $O(1)$ times as u (it is first deleted, then reinserted). Furthermore, every solution vertex (x or y) is scanned $O(\Delta)$ times; more precisely, it is scanned $O(1)$ times when each 2-tight neighbor u is processed, and there are at most Δ such neighbors. Finally, every 1- or 2-tight neighbor v of x is scanned $O(\Delta)$ times: $O(1)$ times when each of its (at most two) neighbors in the solution is scanned. Together, these observations imply that every vertex can be scanned at most $O(\Delta)$ times. Equivalently, each edge is visited $O(\Delta)$ times, thus bounding the total time of the algorithm by $O(m\Delta)$.

For many of the benchmark instances we tested, Δ is a small constant, and in most it is $o(n)$. For denser instances, however, this bound on the running time may be overly pessimistic. In fact, we can use the exact same argument to get a bound of $O(mk)$, where $k \leq \Delta$ is the maximum number of 2-tight neighbors of any solution vertex x. We have thus proven the following:

Theorem 2. *Given a 2-maximal solution S, one can find a valid 3-improvement (or prove that none exists) in $O(mk)$ time, where $k \leq \Delta < n$ is the maximum number of 2-tight neighbors of any vertex $x \in S$.*

Practical considerations. Ideally, we would like to have an incremental version of the algorithm in practice, following the approach described in Section 3.1.2 for 2-improvements. There, we suggested keeping list of *candidate vertices*, which would be updated every time the solution changes. Although we could keep a similar list for 3-improvements (keeping all 2-tight vertices u that might be part of a valid move), maintaining it would be substantially more expensive because changes in the neighborhoods of any of the five relevant vertices (u, v, w, x, and y) would have to be taken into account.

Instead, we decided to use a simpler strategy in our implementation of 3-improvements. It works in *passes*. Each pass starts by running a local search based on 2-improvements to ensure the solution is 2-maximal. It then processes all vertices in turn, performing 3-improvements as they are found. If vertex u is 2-tight when it is its turn to be processed, the algorithm looks for x, y, v, and w, as explained above. Otherwise,

u is just skipped. In practice, the solution becomes 3-maximal after a very small number of passes (often 1 or 2) and almost all the improvement (relative to the starting solution) is obtained in the first pass.

4. Metaheuristics

4.1. **Iterated Local Search.** To test our local searches, we use them within a heuristic based on the *iterated local search* (ILS) metaheuristic [14]. We start from a random solution S, apply local search to it, then repeatedly execute the following steps:

(1) $S' \leftarrow \text{perturb}(S)$;
(2) $S' \leftarrow \text{localsearch}(S')$;
(3) $S \leftarrow S'$ if certain conditions are met.

Any reasonable stopping criterion can be used, and the algorithm returns the best solution found. The remainder of this section details our implementation of each step of this generic algorithm.

The perturbations in Step 1 are performed with the *force*(k) routine, which sequentially inserts k vertices into the solution (the choice of which ones will be explained shortly) and removes the neighboring vertices as necessary. (We call these *forced insertions*.) It then adds free vertices at random until the solution is maximal. We set $k = 1$ in most iterations, which means a single vertex will be inserted. With small probability $(1/(2 \cdot |S|))$, however, we pick a higher value: k is set to $i + 1$ with probability proportional to $1/2^i$, for $i \geq 1$. We must then choose *which* k vertices to insert. If $k = 1$, we pick a random non-solution vertex. If k is larger, we also start with a random non-solution vertex, and pick the j-th vertex (for $j > 1$) among the non-solution vertices within distance exactly two from the first $j - 1$ vertices. (If there is no such vertex, we simply stop inserting.)

We use two techniques for diversification. The first is *soft tabu*. We keep track of the last iteration in which each non-solution vertex was part of the solution. Whenever the *force* routine has a choice of multiple vertices to insert, it looks at κ (an input parameter) candidates uniformly at random (with replacement) and picks the "oldest" one, i.e., the one which has been outside the solution for the longest time. We set $\kappa = 4$ in our experiments. The second diversification technique is employed during local search. If v was the only vertex inserted by the *force* routine, the subsequent local search will only allow v to be removed from the solution after all other possibilities have been tried.

Step 2 of ILS is a standard local search using the 2-improvement algorithm described in Section 3.1. It stops when a local optimum is reached. We also tried incorporating the local search based on 3-improvements into ILS, but the fact that it is not incremental made it too slow to improve the results. By restricting ourselves to 2-improvements, we allow more ILS iterations to be performed.

In Step 3, if the solution S' obtained after the local search is at least as good as S (i.e., if $|S'| \geq |S|$), S' becomes the new current solution. We have observed that always going to S' (even when $|S'| < |S|$) may cause the algorithm to stray from the best known solution too fast. To avoid this, we use a technique akin to plateau search. If ILS arrives at the current solution S from a solution that was better, it is not allowed to go to a worse solution for at least $|S|$ iterations. If the current solution does not improve in this time, the algorithm is again allowed to go to a worse solution S'. It does so with probability $1/(1 + \delta \cdot \delta^*)$, where $\delta = |S| - |S'|$,

$\delta^* = |S^*| - |S'|$, and S^* is the best solution found so far. Intuitively, the farther S' is from S and S^*, the least likely the algorithm is to set $S \leftarrow S'$. If the algorithm does not go to S' (including during plateau search), we undo the insertions and deletions that led to S', then add a small perturbation by performing a random 1-swap in S, if possible. We can find a 1-swap in constant time by keeping the list of all 1-tight vertices explicitly.

Finally, we consider the stopping criterion. As already mentioned, any reasonable criterion works. In our experiments, we stop the algorithm when the average number of scans per arc exceeds a predetermined limit (which is the same for every instance within each family we tested). An arc scan is the most basic operation performed by our algorithm: in fact, the total running time is proportional to the number of such scans. By fixing the number of scans per arc (instead of the total number of scans) in each family, we make the algorithm spend more time on larger instances, which is a sensible approach in practice. To minimize the overhead of counting arc scans individually, our code converts the bound on arc scans into a corresponding bound on vertex scans (using the average vertex degree), and only keeps track of vertex scans during execution.

4.2. **The GLP Algorithm.** We now discuss the algorithm of Grosso, Locatelli, and Pullan [9], which we call GLP. Although it was originally formulated for the maximum clique problem, our description refers to the MIS problem, as does our implementation. We implemented "Algorithm 1 with restart rule 2," which seems to give the best results overall among the several variants proposed in [9]. What follows is a rough sketch of the algorithm. See the original paper for details.

The algorithm keeps a *current solution* S (initially empty), and spends most of its time performing plateau search (simple swaps). A simple tabu mechanism ensures that vertices that leave the solution during plateau search do not return during the same iteration, unless they become free and there are no alternative moves. A successful iteration ends when a non-tabu vertex becomes free: we simply insert it into the solution and start a new iteration. An iteration is considered unsuccessful if this does not happen after roughly $|S|$ moves: in this case, the solution is perturbed with the forced insertion of a single non-solution vertex (with at least four solution neighbors, if possible), and a new iteration starts. GLP does not use local search.

Unlike Grosso et al.'s implementation of GLP, ours does not stop as soon as it reaches the best solution reported in the literature. Instead, we use the same stopping criterion as the ILS algorithm, based on the number of arc scans. Although different, both ILS and GLP have scans as their main basic operation. Using the number of arc scans as the stopping criterion ensures that both algorithms have similar running times for all instances.

5. Experimental Results

All algorithms were implemented by the authors in C++ and compiled with `gcc` v. 4.3.2 with the full optimization (`-O3`) flag. All runs were made on one core of a 3.16 GHz Intel Core 2 Duo CPU with 4 GB of RAM running Windows 7 Enterprise 64-bit Edition. CPU times were measured with the `getrusage` function, which has precision of 1/60 second. Times do not include reading the graph and building the adjacency lists, since these are common to all algorithms. But they do include the allocation, initialization, and destruction of the data structures specific to each algorithm.

5.1. **Instances.** We test our algorithms on five families of instances. The DIMACS family contains instances of the maximum clique problem from the 2nd DIMACS Implementation Challenge [11], which have been frequently tested in the literature. It includes a wide variety of instances, with multiple topologies and densities. Since we deal with the MIS problem, we use the complements of the original graphs. For instances with no known optimum, we report the best results available at the time of writing (as listed in [9, 16]).

The SAT family contains transformed satisfiability instances originally from the SAT'04 competition, available at [19] and tested in [9, 16]. All optima are known.

The CODE family, made available by N. Sloane [18], consists of challenging graphs arising from coding theory. Each subfamily refers to a different error-correcting code, with vertices representing code words and edges indicating conflicts between them. The best known results for the hardest instances were found by the specialized algorithms of Butenko et al. [4, 5].

The last two families, MESH and ROAD, are novel in the context of the independent set problem. MESH is motivated by an application in Computer Graphics recently described by Sander et al. [17]. To process a triangulation efficiently in graphics hardware, their algorithm must find a small subset of triangles that covers all the edges in the mesh. This is the same as finding a small set cover on the corresponding dual graph (adjacent faces in the original mesh become adjacent vertices in the dual). The MESH family contains the duals of well-known triangular meshes. While converting the original primal meshes, we repeatedly eliminated vertices of degree one and zero from the dual, since there is always a maximum independent set that contains them. (Degree-one vertices arise when the original mesh is open, i.e., when it has edges that are adjacent to a single triangle instead of the usual two.) Almost all vertices in the resulting MIS instances (which are available upon request) have degree three.

The ROAD family contains planar graphs representing parts of the road network of the United States, originally made available for the 9th DIMACS Implementation Challenge, on shortest paths [6]. Vertices represent intersections, and edges correspond to the road segments connecting them. As in the previous family, these graphs have numerous vertices of degree one. We chose not to eliminate them explicitly, since these instances are already available in full form.

For conciseness, we only report results on a few representatives of each family, leaving out easy instances and those that behave similarly to others.

5.2. **Local Search.** We start our evaluation with an analysis of the local search algorithms by themselves, in terms of both solution quality and running time.

We first ran the local searches on solutions found by a linear-time algorithm that inserts free vertices uniformly at random into an initially empty solution until it becomes maximal. Table 1 reports the results obtained after 999 runs of this *random* algorithm (which we call R), with different random seeds. The first four columns characterize the instance: its name, number of vertices (n), average vertex degree (DEG), and best known solution (BEST). (Note that BEST is taken from the literature for CODE, DIMACS, and SAT; for ROAD and MESH, we show the best results found by the metaheuristics tested in this paper.) The next column shows the average solution obtained by the constructive algorithm (L1), followed by the average local maxima obtained by the 2-improvement (L2) and 3-improvement (L3) local searches. Finally, the last three columns show the average running time (in

TABLE 1. Performance of the local search algorithms when starting from a random solution (R). For each instance, the table shows its name, number of vertices (n), average vertex degree (DEG), and the best known solution (BEST). Average solutions are then shown for the constructive algorithm (L1), 2-improvement local search (L2), and 3-improvement local search (L3). The last three columns show the average running times of these methods in milliseconds (local search times include construction). Each block of instances corresponds to one family (DIMACS, SAT, CODE, MESH, and ROAD, respectively).

INSTANCE				AVG SOLUTION			AVG TIME (ms)		
NAME	n	DEG	BEST	L1	L2	L3	L1	L2	L3
C2000.9	2000	199.5	80	51.2	59.6	62.4	0.04	0.25	0.92
MANN_a81	3321	3.9	1100	1082.2	1082.2	1082.2	0.08	0.27	0.39
brock400_2	400	100.1	29	16.6	19.3	20.2	0.01	0.05	0.15
brock400_4	400	100.2	33	16.8	19.3	20.4	0.01	0.05	0.16
c-fat500-10	500	312.5	126	125.0	125.0	125.0	0.05	0.31	0.31
hamming10-2	1024	10.0	512	243.2	419.1	422.4	0.03	0.21	0.27
johnson32-2-4	496	60.0	16	16.0	16.0	16.0	0.01	0.04	2.28
keller6	3361	610.9	59	34.5	43.0	45.5	0.06	0.48	4.17
p_hat1500-1	1500	1119.1	12	6.9	8.1	8.9	0.02	0.16	0.93
p_hat1500-3	1500	369.3	94	42.8	77.7	84.6	0.04	0.35	1.09
san1000	1000	498.0	15	7.6	7.6	7.7	0.02	0.80	4.04
san400_0.7_1	400	119.7	40	19.6	20.5	20.5	0.01	0.09	0.45
san400_0.9_1	400	39.9	100	44.1	54.4	56.4	0.01	0.07	0.24
frb59-26-1	1534	165.0	59	39.2	45.9	48.3	0.03	0.18	1.13
frb100-40	4000	286.4	100	66.4	76.7	80.5	0.07	0.47	3.84
1et.2048	2048	22.0	316	232.4	268.4	280.9	0.05	0.24	2.17
1zc.4096	4096	45.0	379	253.8	293.2	307.4	0.08	0.45	2.93
2dc.2048	2048	492.6	24	15.6	18.7	19.9	0.03	0.23	6.46
dragon	150000	3.0	66442	56335.4	61479.2	63082.6	5.75	21.22	80.56
dragonsub	600000	3.0	282294	226999.4	256510.3	264696.9	59.97	213.42	507.79
buddha	1087716	3.0	480683	408208.3	445097.8	456406.2	151.63	482.37	1078.56
bay	321270	2.5	166363	144693.6	158677.7	162269.6	17.83	83.67	216.81
fla	1070376	2.5	549548	476260.7	523241.4	535056.1	157.88	508.17	1021.52

milliseconds) of these algorithms. Note that local search times include construction of the initial solution.

Given the somewhat low precision of our timing routine (and how fast the algorithms are in this experiment), we did not measure running times directly. Instead, we ran each subsequence of 111 seeds repeatedly until the total running time was at least 5 seconds, then took the average time per run. Before each timed run, we ran the whole subsequence of 111 once to warm up the cache and minimize fluctuations. (Single runs would be slightly slower, but would have little effect on the relative performance of the algorithms.)

For a more complete understanding of the local searches, we also ran them on solutions found by a natural greedy algorithm. It builds the solution one vertex at a time, always picking for insertion the vertex with the minimum *residual degree*, i.e., with the fewest free neighbors. Its goal is to preserve as many free vertices as possible after each insertion.

TABLE 2. Performance of the local search algorithms starting from greedy (G) solutions. L1 refers to the constructive algorithm, L2 to the 2-improvement local search, and L3 to the 3-improvement local search.

INSTANCE				AVG SOLUTION			AVG TIME (ms)		
NAME	n	DEG	BEST	L1	L2	L3	L1	L2	L3
C2000.9	2000	199.5	80	66.0	68.0	68.0	3.83	4.03	4.30
MANN_a81	3321	3.9	1100	1095.9	1095.9	1096.0	0.23	0.45	1.27
brock400_2	400	100.1	29	22.0	23.0	23.0	0.41	0.45	0.49
brock400_4	400	100.2	33	22.0	22.0	22.0	0.42	0.45	0.51
c-fat500-10	500	312.5	126	126.0	126.0	126.0	0.96	1.22	1.22
hamming10-2	1024	10.0	512	512.0	512.0	512.0	0.13	0.23	0.23
johnson32-2-4	496	60.0	16	16.0	16.0	16.0	0.25	0.28	2.60
keller6	3361	610.9	59	48.2	48.9	50.1	18.23	18.60	20.55
p_hat1500-1	1500	1119.1	12	10.0	10.0	10.0	16.63	16.78	17.20
p_hat1500-3	1500	369.3	94	86.0	91.0	92.0	6.29	6.49	6.96
san1000	1000	498.0	15	10.0	10.0	10.0	3.98	4.05	6.53
san400_0.7_1	400	119.7	40	21.0	21.0	21.0	0.43	0.48	1.22
san400_0.9_1	400	39.9	100	92.0	100.0	100.0	0.17	0.23	0.23
frb59-26-1	1534	165.0	59	48.0	48.0	48.0	2.63	2.75	3.43
frb100-40	4000	286.4	100	82.0	82.0	84.6	12.71	13.06	15.38
1et.2048	2048	22.0	316	292.4	295.1	299.2	0.66	0.81	1.98
1zc.4096	4096	45.0	379	328.5	329.5	331.0	2.24	2.56	3.42
2dc.2048	2048	492.6	24	21.0	22.0	22.0	12.21	12.38	14.83
dragon	150000	3.0	66442	64175.3	64175.3	64310.8	13.64	26.16	66.34
dragonsub	600000	3.0	282294	277255.9	277255.9	277352.2	92.87	206.55	296.35
buddha	1087716	3.0	480683	463923.1	463923.1	465028.9	225.47	476.08	911.21
bay	321270	2.5	166363	165562.0	165566.5	165638.6	49.61	93.85	178.79
fla	1070376	2.5	549548	545958.5	545969.9	546268.3	283.24	555.00	891.35

This algorithm can be easily implemented in $O(m)$ time if we keep free vertices in buckets according to the their residual degrees, which are maintained explicitly. When a new vertex is inserted into the solution, we scan its previously free neighbors to update the residual degrees of their own neighbors, which are then moved between buckets as necessary. Buckets are implemented as doubly-linked lists. We add a small degree of randomization to the algorithm by randomly permuting all vertices before adding them to their original buckets. Table 2 reports the results obtained by this implementation (which we refer to as G) by itself and when followed by local search. The columns are the same as in Table 1.

Finally, we consider a variant of the greedy algorithm that picks a vertex *uniformly at random* among all candidates with minimum residual degree. An efficient implementation of this method cannot use buckets as G does, since one cannot sample uniformly at random from linked lists in constant time. Instead, we use a technique similar to the one used by our standard solution representation (Section 2). Instead of keeping each bucket separately, we maintain a permutation of the entire list of vertices in a single array. All vertices with a given residual degree belong to a contiguous *block* in the array, in arbitrary order. Non-free vertices appear first, followed by the free vertices sorted by residual degree. By keeping the initial positions of all blocks explicilty, we can pick a random element of each block in constant time. It is easy to see that this data structure can be updated efficiently

TABLE 3. Performance of the local search algorithms starting from randomized greedy solutions (RG). L1 refers to the constructive algorithm, L2 to the 2-improvement local search, and L3 to the 3-improvement local search.

INSTANCE				AVG SOLUTION			AVG TIME (ms)		
NAME	n	DEG	BEST	L1	L2	L3	L1	L2	L3
C2000.9	2000	199.5	80	66.6	67.5	68.5	10.58	10.75	11.12
MANN_a81	3321	3.9	1100	1095.4	1095.5	1095.6	0.43	0.64	1.52
brock400_2	400	100.1	29	22.0	22.4	22.7	0.93	0.97	1.02
brock400_4	400	100.2	33	21.6	22.0	22.2	0.93	0.97	1.03
c-fat500-10	500	312.5	126	126.0	126.0	126.0	3.28	3.54	3.54
hamming10-2	1024	10.0	512	512.0	512.0	512.0	0.31	0.40	0.41
johnson32-2-4	496	60.0	16	16.0	16.0	16.0	0.69	0.72	2.97
keller6	3361	610.9	59	48.2	49.3	50.0	45.27	45.61	47.15
p_hat1500-1	1500	1119.1	12	9.8	10.5	10.6	38.71	38.85	39.20
p_hat1500-3	1500	369.3	94	85.9	88.5	89.9	14.05	14.27	14.69
san1000	1000	498.0	15	9.5	9.5	9.6	11.21	11.35	19.19
san400_0.7_1	400	119.7	40	21.3	21.3	21.4	1.04	1.09	1.80
san400_0.9_1	400	39.9	100	80.6	100.0	100.0	0.39	0.45	0.45
frb59-26-1	1534	165.0	59	47.5	48.3	49.6	8.10	8.21	9.13
frb100-40	4000	286.4	100	80.7	82.1	83.9	28.04	28.40	31.03
1et.2048	2048	22.0	316	292.7	295.7	299.5	1.28	1.43	2.64
1zc.4096	4096	45.0	379	327.1	328.6	330.4	4.87	5.19	6.11
2dc.2048	2048	492.6	24	21.0	21.2	21.6	23.37	23.50	27.62
dragon	150000	3.0	66442	64022.3	64243.7	64489.9	22.74	35.36	75.71
dragonsub	600000	3.0	282294	275614.6	276453.6	277633.6	99.60	214.87	333.25
buddha	1087716	3.0	480683	463295.8	464868.0	466632.2	221.29	477.98	854.63
bay	321270	2.5	166363	165463.4	165610.7	165753.3	75.36	120.49	200.96
fla	1070376	2.5	549548	545502.1	546146.3	546775.9	444.61	716.92	1021.07

after a vertex is inserted in the solution. Although it has higher data structure overhead than G, this *randomized greedy* algorithm (RG) still runs in $O(m)$ time. The results obtained with it are presented in Table 3.

Comparing all three tables, we observe that the greedy algorithms (G and RG) find solutions of similar quality, and are usually much better than random (R). Random is consistently faster, however, especially for very dense instances, such as p_hat1500-1. While the greedy algorithms must visit every edge in the graph, the random algorithm only traverses the adjacency lists of the vertices that end up in the solution. Applying the 2-improvement local search to R is often cheaper than running G or RG by themselves, but the local maxima reached from R are usually worse. Moreover, on very sparse instances, 2-improvement is actually faster when applied to greedy solutions than to random ones (even including construction times), since it starts much closer to a local maximum.

The 2-improvement local search is remarkably fast when applied to the greedy solutions, particularly when the solution is small compared to the total number of vertices (as in san1000, for example). Even for large, sparse instances (such as fla and buddha) the local search is about as fast as the constructive algorithm. (Recall that the local search times reported in the tables actually include construction.)

On these large, sparse instances, 2-improvements are much more effective on RG than on G. In fact, G tends to find better solutions than RG, but after local search

the opposite is true. A possible explanation is that the stack-like nature of buckets in G causes the solutions it generates to be more "packed" solutions than for RG.

The 3-exchange local search finds much better solutions than 2-exchange for all three constructive algorithms (especially G).[2] In relative terms, however, they are similar: they find their best solutions when starting from RG, and the worst when starting from R. The running times of the 3-improvement local search are often not much higher than those of the greedy algorithm, even for some dense instances (such as keller6). As anticipated in Section 3.2, this indicates that bounding the running time by $O(m\Delta)$ is sometimes too pessimistic. Of course, bad cases do happen: on johnson32-2-4, for example, the local search is 10 times slower than the standard greedy algorithm—and it does not even improve the solution.

The higher variance of RG helps after local search. Over all 999 runs, for a fixed local search, the best solution found when starting from RG is at least as good as when starting from R or G. The main exception is dragonsub: the best solution found by RG followed by 2-improvement was worse than the best solution found by G, which is actually 2-maximal. Despite this exception, these results suggest that some variant of RG could be well-suited to multistart-based metaheuristics, such as GRASP [7].

5.3. **Metaheuristics.** Although local search can improve the results found by constructive heuristics, we have seen that the local optima are usually somewhat far from the best known bounds. For near-optimal solutions, we turn to metaheuristics. We compare our iterated local search (ILS) with our implementation of Grosso et al.'s GLP algorithm. Our version of GLP deals with the maximum independent set problem directly, and its time per operation is comparable to the original implementation [9].

Tables 4, 5, and 6 present results for DIMACS, CODE, and SAT, respectively. For each instance, we first show its number of vertices, its density, and the best known solution. We then report the minimum, average, and maximum solutions found over 15 runs of each algorithm (the numbers in parentheses indicate how many of the 15 runs found the maximum). Finally, we give the average running time in seconds. Both algorithms were run until the average number of scans per arc reached 2^{17}. The best average solution found in each case is highlighted in bold.

As anticipated by our choice of stopping criterion, the relative running times of the algorithms do not fluctuate much: ILS is consistently about twice as fast on average. This gap is due to constant factors in the implementation of the algorithms. In particular, GLP maintains separate data structures to be able to pick non-tabu free vertices and non-tabu 1-tight vertices uniformly at random in constant time. Although this gap could probably be reduced with additional tuning, it is small enough for our purposes. The number of operations is similar and implementation-independent. For the remainder of this section, our discussion ignores any difference in running times between the algorithms.

Regarding solution quality, we note that, together, the algorithms do rather well on these families. For almost all instances, the best known bound was found at least once (the main exceptions are C2000.9, brock800_2, and the largest SAT graphs). Moreover, the average solutions found by ILS and GLP are usually very

[2]Due to an implementation issue, a preliminary version of this paper [1] incorrectly stated that the 3-improvement local search could seldom improve 2-maximal solutions. As the tables show, this is not true: 3-improvements are found quite often.

TABLE 4. DIMACS family. For each algorithm, we show the worst (MIN), average (AVG), and best (MAX) solution found over 15 runs (the number of runs that found the maximum is in parenthesis). We also show the average running times in seconds (TIME). Both algorithms were run until the average arc was scanned 2^{17} times.

GRAPH				ILS				GLP			
NAME	n	DENS	BEST	MIN	AVG	MAX	TIME	MIN	AVG	MAX	TIME
C1000.9	1000	0.099	68	68	**68.0**	68(15)	25	68	**68.0**	68(15)	50
C2000.5	2000	0.500	16	16	**16.0**	16(15)	436	16	**16.0**	16(15)	967
C2000.9	2000	0.100	80	76	76.9	77(14)	103	77	**77.5**	79(2)	182
C4000.5	4000	0.500	18	17	17.1	18(1)	1897	18	**18.0**	18(15)	3708
DSJC1000.5	1000	0.500	15	15	**15.0**	15(15)	103	15	**15.0**	15(15)	258
MANN_a27	378	0.010	126	126	**126.0**	126(15)	1	126	**126.0**	126(15)	2
MANN_a45	1035	0.004	345	344	**344.5**	345(8)	3	343	343.8	344(12)	5
MANN_a81	3321	0.001	1100	1100	**1100.0**	1100(15)	10	1097	1097.6	1098(9)	17
brock200_1	200	0.255	21	21	**21.0**	21(15)	3	21	**21.0**	21(15)	6
brock200_2	200	0.504	12	12	**12.0**	12(15)	4	12	**12.0**	12(15)	13
brock200_3	200	0.395	15	15	**15.0**	15(15)	4	15	**15.0**	15(15)	10
brock200_4	200	0.342	17	17	**17.0**	17(15)	4	17	**17.0**	17(15)	9
brock400_1	400	0.252	27	25	25.0	25(15)	11	25	**25.1**	27(1)	22
brock400_2	400	0.251	29	25	25.0	25(15)	11	25	**27.7**	29(10)	20
brock400_3	400	0.252	31	25	27.0	31(5)	11	31	**31.0**	31(15)	19
brock400_4	400	0.251	33	25	30.3	33(10)	11	33	**33.0**	33(15)	16
brock800_1	800	0.351	23	21	21.0	21(15)	60	21	**21.1**	23(1)	112
brock800_2	800	0.349	24	21	**21.0**	21(15)	60	21	**21.0**	21(15)	111
brock800_3	800	0.351	25	22	22.0	22(15)	60	22	**22.2**	25(1)	111
brock800_4	800	0.350	26	21	21.3	26(1)	60	21	**21.7**	26(2)	112
c-fat500-10	500	0.626	126	126	**126.0**	126(15)	30	126	**126.0**	126(15)	65
hamming10-2	1024	0.010	512	512	**512.0**	512(15)	7	512	**512.0**	512(15)	15
hamming10-4	1024	0.171	40	40	**40.0**	40(15)	39	40	**40.0**	40(15)	98
johnson32-2-4	496	0.121	16	16	**16.0**	16(15)	4	16	**16.0**	16(15)	21
keller6	3361	0.182	59	59	**59.0**	59(15)	519	59	**59.0**	59(15)	862
p_hat1500-1	1500	0.747	12	12	**12.0**	12(15)	173	12	**12.0**	12(15)	808
p_hat1500-2	1500	0.494	65	65	**65.0**	65(15)	150	65	**65.0**	65(15)	252
p_hat1500-3	1500	0.246	94	94	**94.0**	94(15)	88	94	**94.0**	94(15)	136
san1000	1000	0.498	15	15	**15.0**	15(15)	84	15	**15.0**	15(15)	234
san400_0.7_1	400	0.300	40	40	**40.0**	40(15)	15	40	**40.0**	40(15)	21
san400_0.7_2	400	0.300	30	30	**30.0**	30(15)	15	30	**30.0**	30(15)	23
san400_0.7_3	400	0.300	22	22	**22.0**	22(15)	13	22	**22.0**	22(15)	25
san400_0.9_1	400	0.100	100	100	**100.0**	100(15)	7	100	**100.0**	100(15)	8

close to one another. GLP was consistently better on the brock instances (dense random graphs with a "hidden" large clique), C2000.9 and C4000.5 (also random, with larger cliques naturally hidden by the high value of n). GLP's advantage at finding these cliques is probably due to its stronger tabu mechanism. In contrast, GLP does poorly on the MANN instances (sparse graphs with large independent sets), while ILS finds the optimal solution MANN_a81 in less than one second on average. On SAT instances, ILS found the best known solutions for all instances in at least one run, but GLP was more consistent on average, reaching these solutions more often. On SAT, ILS does better on the hardest (larger) instances, but misses the best known solution more often on smaller instances.

Although our algorithm does well on these families, GLP is somewhat more robust, especially on DIMACS and CODE. This is not the case for large, sparse

TABLE 5. Results for the CODE family with 2^{17} scans per arc (aggregated over 15 runs).

GRAPH				ILS				GLP			
NAME	n	DENS	BEST	MIN	AVG	MAX	TIME	MIN	AVG	MAX	TIME
1dc.1024	1024	0.046	94	93	**93.1**	94(2)	14	93	**93.1**	94(2)	31
1dc.2048	2048	0.028	172	170	171.1	172(8)	32	170	**171.5**	172(11)	74
1et.1024	1024	0.018	171	171	**171.0**	171(15)	8	170	170.9	171(13)	16
1et.2048	2048	0.011	316	316	**316.0**	316(15)	16	316	**316.0**	316(15)	40
1tc.1024	1024	0.015	196	196	**196.0**	196(15)	8	196	**196.0**	196(15)	18
1tc.2048	2048	0.009	352	352	**352.0**	352(15)	15	352	**352.0**	352(15)	37
1zc.1024	1024	0.032	112	111	111.1	112(2)	10	112	**112.0**	112(15)	28
1zc.2048	2048	0.019	198	196	197.3	198(6)	22	197	**197.8**	198(12)	65
1zc.4096	4096	0.011	379	358	367.7	379(1)	51	367	**374.4**	379(4)	160
2dc.1024	1024	0.323	16	16	**16.0**	16(15)	50	16	**16.0**	16(15)	198
2dc.2048	2048	0.241	24	23	23.8	24(12)	165	24	**24.0**	24(15)	527

TABLE 6. Results for the SAT family with 2^{17} scans per arc (aggregated over 15 runs).

GRAPH				ILS				GLP			
NAME	n	DENS	BEST	MIN	AVG	MAX	TIME	MIN	AVG	MAX	TIME
frb30-15-1	450	0.176	30	30	**30.0**	30(15)	9	30	**30.0**	30(15)	22
frb35-17-1	595	0.158	35	34	34.9	35(14)	13	35	**35.0**	35(15)	32
frb40-19-1	760	0.143	40	40	**40.0**	40(15)	19	40	**40.0**	40(15)	43
frb45-21-1	945	0.133	45	44	44.7	45(11)	27	44	**44.9**	45(13)	62
frb50-23-1	1150	0.121	50	48	**48.9**	50(1)	36	48	48.6	49(9)	82
frb53-24-1	1272	0.117	53	51	**51.5**	53(1)	42	51	51.3	52(4)	93
frb56-25-1	1400	0.112	56	54	**54.2**	55(3)	49	54	54.1	55(2)	111
frb59-26-1	1534	0.108	59	57	**57.3**	58(4)	57	57	57.0	57(15)	126
frb100-40	4000	0.072	100	95	**95.3**	96(5)	249	94	94.1	95(1)	495

graphs, to which we now turn our attention. Table 7 presents results for the MESH family. Because it includes much larger graphs than the previous series, we limit the average number of arc scans to 2^{15}.

Note that GLP tends to find slightly better results for smaller instances; for some larger instances, notably dragon and buddha, ILS is clearly better. The relative performance of the algorithms appears to be correlated with the regularity of the meshes: GLP is better for regular meshes, whereas ILS is superior for more irregular ones. We verified this by visual inspection—see Figure 1 for a couple of examples. Alternatively, we can use the standard deviation of the vertex degrees in the original (primal) mesh is a rough proxy for irregularity. It is relatively smaller for bunny (0.58) and dragonsub (0.63), on which GLP is the best algorithm, and bigger for buddha (1.28) and dragon (1.26), on which ILS is superior.[3] Note that dragonsub is a subdivision of dragon: a new vertex is inserted in the middle of each edge, and each triangle is divided in four. Both meshes represent the same model, but because every new vertex has degree exactly six, dragonsub is much more regular.

Although optimal solutions for the MESH family are not known, Sander et al. [17] computed lower bounds on the cover solutions for seven of their original meshes,

[3]The standard deviation is not always a good measure of regularity. Despite being highly regular, gameguy has a triangulation pattern in which roughly half the vertices have degree 4 and half have degree 8, leading to a standard deviation higher than 2.

TABLE 7. Results for the MESH family with 2^{15} scans per arc (aggregated over 15 runs).

GRAPH		ILS				GLP			
NAME	n	MIN	AVG	MAX	TIME	MIN	AVG	MAX	TIME
dolphin	554	249	**249**	249(15)	1	249	**249**	249(15)	1
mannequin	1309	583	**583**	583(15)	1	583	**583**	583(15)	2
beethoven	4419	1998	2001	2004(1)	5	2001	**2002**	2004(5)	7
cow	5036	2335	2342	2346(3)	5	2334	**2344**	2346(10)	8
venus	5672	2670	2675	2680(1)	6	2682	**2683**	2684(7)	9
fandisk	8634	4058	4066	4074(1)	9	4066	**4070**	4074(1)	17
blob	16068	7230	7237	7243(1)	17	7238	**7241**	7245(1)	33
gargoyle	20000	8843	**8846**	8849(2)	23	8843	**8846**	8849(2)	44
face	22871	10203	10207	10211(1)	25	10205	**10208**	10212(1)	48
feline	41262	18804	18812	18822(1)	48	18812	**18820**	18827(1)	104
gameguy	42623	20615	20634	20655(1)	45	20644	**20663**	20693(1)	146
bunny	68790	32218	32238	32268(1)	79	32248	**32265**	32281(1)	208
dragon	150000	66412	**66427**	66442(1)	178	66366	66381	66399(1)	494
turtle	267534	122250	**122322**	122363(1)	509	122230	122317	122404(1)	1631
dragonsub	600000	281882	281965	282002(1)	1246	282172	**282241**	282294(1)	3479
ecat	684496	321886	**321982**	322079(1)	2177	321845	321930	322116(1)	7665
buddha	1087716	480595	**480646**	480683(1)	3239	479122	479231	479285(1)	11374

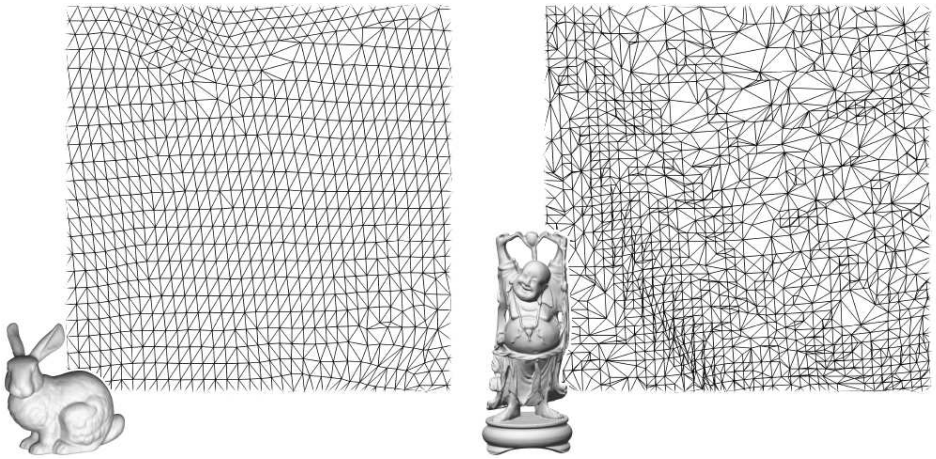

FIGURE 1. Detailed view of the meshes from which bunny (left) and buddha (right) were derived. Notice that bunny is significantly more regular. (Pictures provided by the authors of [17].)

which can be easily translated into upper bounds for our (MIS) instances. The instances (and MIS upper bounds) are: buddha (495807), bunny (32850), dragon (68480), fandisk (4168), feline (19325), gargoyle (9120), and turtle (125438). For ILS, the lowest gaps observed were on bunny (the solutions were 1.9% lower than the upper bounds), and the highest on buddha (more than 3.0%). The extremes of GLP were on the same instances: 1.8% on bunny and 3.3% on buddha.

TABLE 8. Results for the ROAD family with 2^{12} scans per arc (aggregated over 15 runs).

GRAPH			ILS				GLP			
NAME	n	DEG	MIN	AVG	MAX	TIME	MIN	AVG	MAX	TIME
ny	264346	2.8	131416	**131433**	131447(1)	63	131148	131178	131204(1)	191
bay	321270	2.5	166345	**166356**	166363(2)	84	166218	166228	166258(1)	255
col	435666	2.4	225736	**225748**	225763(1)	128	225563	225590	225615(1)	385
fla	1070376	2.5	549517	**549527**	549548(1)	374	548604	548648	548686(1)	1223

Finally, Table 8 presents the results for ROAD, with the average number of scans per arc limited to 2^{12}. Here ILS has clear advantage. On every instance, the worst result found by ILS was better than the best found by GLP.

We note that MESH and ROAD are fundamentally different from the previous families. These are large graphs with linear-sized maximum independent sets. Both ILS and GLP start from relatively bad solutions, which are then steadily improved, one vertex at a time. To illustrate this, Figure 2 shows the average solutions found for three representatitive instances from these families (buddha, turtle, and fla) as the algorithms progress. GLP initially finds better solutions, but is soon overtaken by ILS. In the case of turtle, GLP eventually catches up again. The third curve in the plots (ILS+plateau) refers to a version of our algorithm that also performs plateau search when the current solution improves (recall that ILS only performs plateau search when the solution worsens). Although faster at first, ILS+plateau is eventually surpassed by ILS in all three instances.

For comparison, Figure 2 also shows results for longer runs (2^{20} scans per arc, with 15 different seeds) on instances from the other three families: frb100-40 (SAT), C2000.9 (DIMACS), and 1zc.4096 (CODE). As before, GLP starts much better. On frb100-40, it is soon surpassed by ILS. On 1zc.4096, ILS slowly reduces the gap, but does not quite close it. On C2000.9, GLP is consistently better, even as the number of scans increases. Note that the performance of ILS+plateau is not much different from ILS itself, unlike on MESH and ROAD instances.

6. Final Remarks

We have proposed a fast implementation of a natural local search procedure for the independent set problem. Within an iterated local search (a metaheuristic), it provided results competitive with the best methods previously proposed, often matching the best known solutions (including optima) on the well-studied DIMACS, CODE, and SAT families. On some large, sparse instances (road networks and irregular meshes), its performance is consistently superior to that of GLP. For these large instances, however, we do not know exactly how far our method is from the optimal solution: there may be room for improvement. It seems reasonable, for example, to deal with these problems more locally. Instead of looking at the entire graph at once, we conjecture that one could do better by focusing at individual regions at a time.

Acknowledgements. We thank Diego Nehab and Pedro Sander for sharing their paper and providing us with the MESH instances. We are also grateful to Manuel Holtgrewe for pointing out a few minor issues with the version of our code we used in the preliminary version of this paper [1]. In particular, some vertex scans of ILS were not counted properly, which affected its stopping criterion.

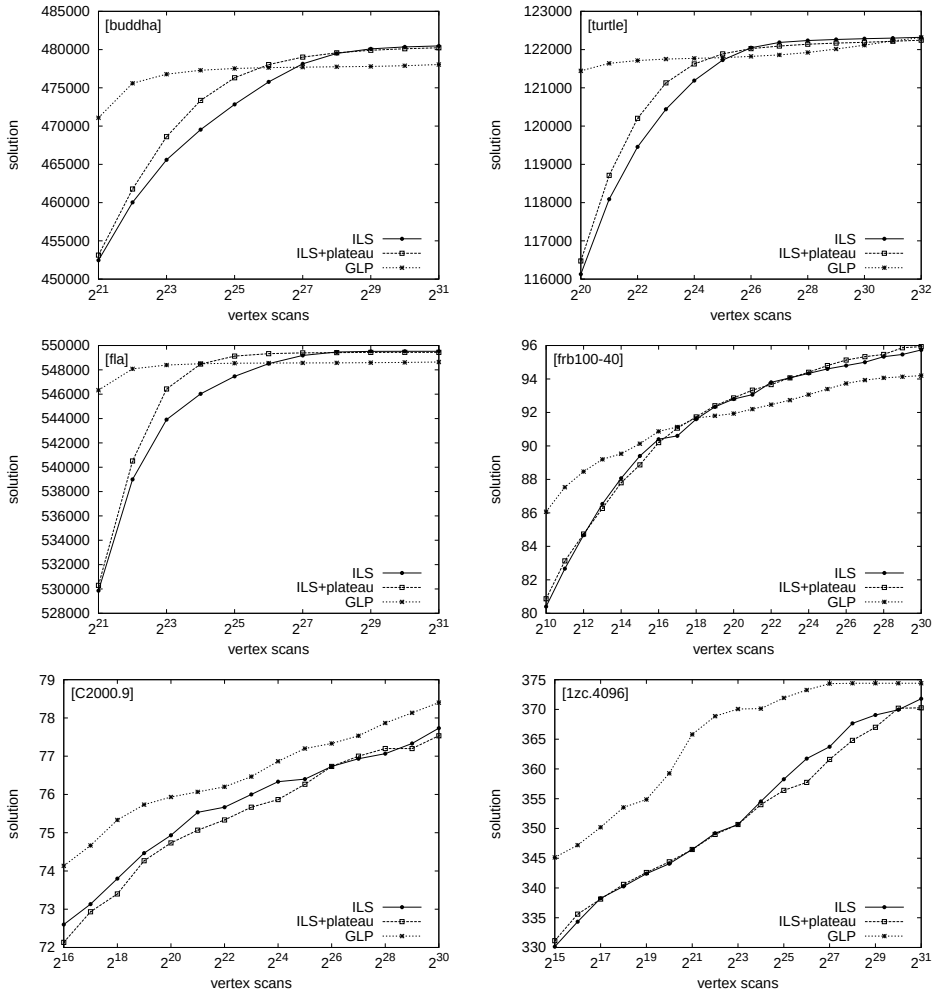

FIGURE 2. Average solutions found as the number of scans per vertex increases. Results for buddha (top left), turtle (top right), fla (middle left), frb100-40 (middle right), C2000.9 (bottom left), and 1zc.4096 (bottom right).

REFERENCES

[1] D. V. Andrade, M. G. C. Resende, and R. F. Werneck. Fast local search for the maximum independent set problem. In C.C. McGeoch, editor, *Proc. 7th International Workshop on Experimental Algorithms (WEA)*, volume 5038, pages 220–234. Springer, 2008.

[2] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4):610–637, 2001.

[3] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In D. Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization (Sup. Vol. A)*, pages 1–74. Kluwer, 1999.

[4] S. Butenko, P. M. Pardalos, I. Sergienko, V. Shylo, and P. Stetsyuk. Finding maximum independent sets in graphs arising from coding theory. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 542–546, 2002.

[5] S. Butenko, P. M. Pardalos, I. Sergienko, V. Shylo, and P. Stetsyuk. Estimating the size of correcting codes using extremal graph problems. In C. Pearce, editor, *Optimization: Structure and Applications*. Springer, 2008.
[6] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths, 2006. `http://www.dis.uniroma1.it/~challenge9`. Last visited on March 15, 2008.
[7] T. Feo, M. G. C. Resende, and S. Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42:860–878, 1994.
[8] A. Grosso, M. Locatelli, and F. Della Croce. Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem. *J. Heuristics*, 10:135–152, 2004.
[9] A. Grosso, M. Locatelli, and W. Pullan. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *J. Heuristics*, 14(6):587–612, 2008.
[10] P. Hansen, N. Mladenović, and D. Urošević. Variable neighborhood search for the maximum clique. *Discrete Applied Mathematics*, 145(1):117–125, 2004.
[11] D. S. Johnson and M. Trick, editors. *Cliques, Coloring and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS, 1996.
[12] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.
[13] K. Katayama, A. Hamamoto, and H. Narihisa. An effective local search for the maximum clique problem. *Information Processing Letters*, 95:503–511, 2005.
[14] H. R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer, 2003.
[15] W. J. Pullan and H. H. Hoos. Dynamic local search for the maximum clique problem. *J. Artificial Intelligence Research*, 25:159–185, 2006.
[16] S. Richter, M. Helmert, and C. Gretton. A stochastic local search approach to vertex cover. In *Proceedings of the 30th German Conference on Artificial Intelligence (KI)*, pages 412–426, 2007.
[17] P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Transactions on Graphics*, 27(5):144:1–144:9, 2008.
[18] N. J. A. Sloane. Challenge problems: Independent sets in graphs, 2000. `http://www.research.att.com/~njas/doc/graphs.html`. Last visited on March 15, 2008.
[19] K. Xu. BHOSLIB: Benchmarks with hidden optimum solutions for graph problems, 2004. `http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm`. Last visited on March 15, 2008.

(D.V. Andrade) Google Inc., 76 Ninth Avenue, New York, NY 10011 USA
E-mail address: `diogo@google.com`

(M.G.C. Resende) Algorithms and Optimization Research Department, AT&T Labs Research, 180 Park Avenue, Room C241, Florham Park, NJ 07932 USA.
E-mail address: `mgcr@research.att.com`

(R.F. Werneck) Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043 USA
E-mail address: `renatow@microsoft.com`