Fast Local Search for the Maximum Independent Set Problem*

Diogo V. Andrade[1], Mauricio G.C. Resende[2], and Renato F. Werneck[3]

[1] Google Inc., 76 Ninth Avenue, New York, NY 10011, USA
diogo@google.com
[2] AT&T Labs Research, 180 Park Ave, Florham Park, NJ 07932, USA
mgcr@research.att.com
[3] Microsoft Research Silicon Valley, 1065 La Avenida, Mtn. View, CA 94043, USA
renatow@microsoft.com

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 present a fast local search routine for this problem. Our algorithm can determine in linear time whether a maximal solution can be improved by replacing a single vertex with two others. We also show that an incremental version of this method can be useful within more elaborate heuristics. 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 problem* (MIS) 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 [2], 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 [11], for most instances one must resort to heuristics to obtain good solutions within reasonable time.

Most successful heuristics [1,7,8,9,12,14,15] maintain a single current solution that is slowly modified by very simple operations, such as individual insertions or 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, at which point a simple insertion can be performed. Grosso et al. [8] have recently obtained

* Part of this work was done while the first author was at Rutgers University and the third author at Princeton University.

C.C. McGeoch (Ed.): WEA 2008, LNCS 5038, pp. 220–234, 2008.

exceptional results in practice by performing plateau search almost exclusively. Their method (as well as several others) occasionally applies a more elaborate 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 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 briefly discuss a generalization of this method to deal with (2,3)-swaps, i.e., two removals followed by three insertions.

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 augmented 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 algorithm is described in Section 3. Section 4 illustrates how it can be applied within a more elaborate heuristic. Experimental results are presented in Section 5, and final remarks are made in Section 6.

2 Basics

The input to the MIS problem is a 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, sorted by label. One can enforce the ordering in linear time by applying radix sort to all edges.

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 solution is *maximal* if it has no free vertices.

Our algorithms represent a solution S as a permutation of all vertices, partitioned into three blocks: first the $|S|$ vertices in the solution, then the free vertices, and finally the nonfree vertices. 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 if the vertex is free) and its position in the permutation (which allows the vertex to be moved between blocks in constant time).

This structure must be updated whenever a vertex v is inserted into or removed from S. The only vertices that change are v itself and its neighbors, so each such operation takes time proportional to the degree of v, denoted by $\deg(v)$. This is more expensive than in simpler solution representations (such as lists or incidence vectors), but the following operations can be easily performed in constant time: (1) check if the solution is maximal (i.e., if the second block is empty); (2) check if a vertex is in the solution (i.e., if it belongs to the first block); (3) determine the tightness of a non-solution vertex; and (4) pick a vertex within any of the three blocks uniformly at random.

3 Local Search

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*. We use the term *move* to refer to a generic (j,k)-swap.

Our main local search algorithm is based on 2-improvements. These natural operations have been studied before (see e.g. [6]); our contribution is a faster implementation. 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. We test each solution vertex $x \in S$ in turn. In any 2-improvement that removes x, both vertices inserted must be neighbors of x (by maximality) that are 1-tight (or the new solution would not be valid) and not adjacent to each other.

To process x efficiently, we first build a list $L(x)$ consisting of the 1-tight neighbors of x, sorted by vertex identifier. 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 missing element 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 $O(m)$.

An alternative linear-time implementation is as follows. As before, process each solution vertex x in turn. First, temporarily remove x from S. Then, for each newly-free neighbor v of x, insert v into S and check if the solution becomes maximal. If it does, simply remove v and process the next neighbor of x; if it does not, inserting any free vertex will yield a valid 2-improvement.

We have also considered more powerful local search algorithms. In particular, using generalized (and more complicated) versions of the techniques above, one can detect a 3-improvement (of prove that none exists) in $O(m\Delta)$ time, where Δ is the maximum vertex degree. Similarly, 2-swaps can be implemented in linear time. Due to space constraints, we omit a full description of these algorithms.

3.1 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.

To accomplish this, 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 its removal. 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.

[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.

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, and stops when the list of candidates is empty.

3.2 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. Simply running the algorithm above on the complement of the input is not enough, since the complement may be much denser.

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) at least one of v or w is not a neighbor of x (by maximality). 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 if it has a neighbor w such that either (1) $\tau(w) = |C|$ or (2) $\tau(w) = |C| - 1$ 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 valid neighbor w.

4 Metaheuristics

4.1 Iterated Local Search

To test our local search, we use it within a heuristic based on the *iterated local search* (ILS) metaheuristic [13]. 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) set $S \leftarrow S'$ 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.

Perturbations 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 start with a random vertex, but 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.

Regarding the third step of the main loop, if the solution S' obtained after the local search is at least as good as S, S' becomes the new current solution. If $|S'| < |S|$, we have observed that always going to 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 1-swap in S (if possible).

Finally, we consider the stopping criterion. 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 [8], which we call GLP. Although it was originally formulated for the maximum clique problem, our description (as well as our implementation) refers to the MIS problem. We implemented "Algorithm 1 with restart rule 2," which seems to give the best results overall among the several variants proposed in [8]. 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. By using the number of arc scans as the stopping criterion, we ensure 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. 3.4.4 with the full optimization (`-O4`) flag. All runs were made on a 3 GHz Pentium IV with 2 GB of RAM running Windows XP Professional. 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 time to allocate, initialize and destroy the data structures specific to each algorithm.

5.1 Instances

The DIMACS family contains maximum clique instances from the 2nd DIMACS Implementation Challenge [10], 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 [8,15]).

The SAT family contains transformed satisfiability instances from the SAT'04 competition, available at [18] and tested in [8,15]. All optima are known.

The CODE family, made available by N. Sloane [17], 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. [3,4].

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. [16]. To process a triangulation efficiently in graphics hardware, their algorithm finds 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 [5]. Vertices represent intersections, and arcs represent 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.

Due to space limitations, 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 first evaluate the local search algorithm by itself, in terms of both solution quality and running time. We tested it with three different constructive algorithms. The random algorithm (R) inserts free vertices uniformly at random until the solution is maximal. The greedy algorithm (G) assigns a *cost* to each free vertex equal to the number of free neighbors it has, and in each iteration picks a free vertex with lowest cost. The randomized greedy algorithm (RG) is a variant of G that picks the vertex to insert *uniformly at random* among all minimum-cost free vertices. Both G and RG can be implemented in linear time, but there is some data structure overhead associated with RG. While G keeps the free vertices in buckets (one for each possible cost), RG maintains the vertices sorted by cost, which is more complicated.

For a representative sample of instances, we ran the constructive algorithms by themselves (R, G, and RG) and followed by local search (RL, GL, and RGL). Table 1 shows the average solutions obtained for 999 random seeds, and Table 2 the average running times. Also shown are the number of vertices (n), the average degree (DEG), and the best known solution (BEST) for each graph. 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.)

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 algorithm must visit every edge in the graph, the random algorithm only traverses

Table 1. Average solutions found by the random, greedy, and randomized greedy constructive algorithms, without (R, G, RG) or with (RL, GL, RGL) local search. The best results among these algorithms are marked in bold. The horizontal lines separate different families, in order: DIMACS, SAT, CODE, MESH, and ROAD.

GRAPH	n	DEG	BEST	R	RL	G	GL	RG	RGL
C2000.9	2000	199.5	80	51.2	59.5	66.0	**68.0**	66.5	67.4
MANN_a81	3321	3.9	1100	1082.1	1082.1	**1096.0**	**1096.0**	1095.5	1095.6
brock400_2	400	100.1	29	16.7	19.4	22.0	**23.0**	22.0	22.4
brock400_4	400	100.2	33	16.7	19.2	**22.0**	**22.0**	21.7	**22.0**
c-fat500-10	500	312.5	126	125.0	125.0	**126.0**	**126.0**	**126.0**	**126.0**
hamming10-2	1024	10.0	512	242.3	412.8	**512.0**	**512.0**	**512.0**	**512.0**
johnson32-2-4	496	60.0	16	**16.0**	**16.0**	**16.0**	**16.0**	**16.0**	**16.0**
keller6	3361	610.9	59	34.4	43.1	48.2	48.9	48.5	**49.6**
p_hat1500-1	1500	1119.1	12	6.8	8.1	10.0	10.0	9.9	**10.4**
p_hat1500-3	1500	369.3	94	42.6	77.7	86.0	**91.0**	85.9	88.3
san1000	1000	498.0	15	7.7	7.7	**10.0**	**10.0**	9.5	9.5
san400_0.7_1	400	119.7	40	19.7	20.6	21.0	21.0	**21.4**	**21.4**
san400_0.9_1	400	39.9	100	44.2	54.0	92.0	**100.0**	81.3	**100.0**
frb59-26-1	1534	165.0	59	39.4	45.8	48.0	48.0	47.6	**48.3**
1et.2048	2048	22.0	316	232.9	268.6	292.2	295.0	292.9	**295.8**
1zc.4096	4096	45.0	379	254.2	293.5	328.5	**329.5**	327.3	328.6
2dc.2048	2048	492.6	24	15.6	18.7	21.0	**22.0**	21.0	21.3
dragon	150000	3.0	66430	56332	61486	64176	64176	64024	**64247**
dragonsub	600000	3.0	282192	227004	256502	277252	**277252**	275611	276451
buddha	1087716	3.0	480664	408215	445100	463914	463914	463303	**464878**
fla	1070376	2.5	549535	476243	523237	545961	545972	545507	**546150**

the adjacency lists of the vertices that end up in the solution. Even after local search, RL is often faster than G or RG, but still finds worse solutions. On sparser instances, RL can be slower than GL or RGL, since the local search has a much worse starting point.

The local search is remarkably fast when applied to the greedy solutions. For large, sparse instances (such as fla and buddha) the local search is 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. We conjecture that the stack-like nature of buckets in G causes it to generate more "packed" solutions than RG. The higher variance of RG helps after local search: over all 999 runs, the best solution found by RGL (not shown in the table) was in most cases at least as good as the best found by any of the other algorithms. (The exceptions were san400_0.7_1, for which RL was superior, and dragonsub, for which G and GL were the best.) This suggests that RGL (or a variant) would be well-suited to multistart-based metaheuristics, such as GRASP [6].

For completeness, we briefly discuss the 3-improvement algorithm (not shown in the tables). Applied to the solutions obtained by RGL, it improved the average solutions of only six instances: 1et.2048 (296.4), 2dc.2048 (21.6), frb59-26-1 (48.6), keller6 (50.2), p_hat1500-1 (10.5) and san1000 (9.7). It also improved RL

Table 2. Constructive algorithms and local search: running times in milliseconds

GRAPH	n	DEG	R	RL	G	GL	RG	RGL
C2000.9	2000	199.5	**0.08**	0.47	8.15	8.57	16.62	16.96
MANN_a81	3321	3.9	**0.18**	0.60	0.53	0.97	0.70	1.13
brock400_2	400	100.1	**0.03**	0.11	0.77	0.85	1.49	1.56
brock400_4	400	100.2	**0.03**	0.11	0.77	0.85	1.48	1.56
c-fat500-10	500	312.5	**0.09**	0.52	2.28	2.71	5.19	5.60
hamming10-2	1024	10.0	**0.07**	0.40	0.29	0.49	0.46	0.66
johnson32-2-4	496	60.0	**0.02**	0.10	0.45	0.52	1.01	1.08
keller6	3361	610.9	**0.11**	0.88	35.32	35.96	71.73	72.43
p_hat1500-1	1500	1119.1	**0.04**	0.31	31.66	31.94	58.66	58.71
p_hat1500-3	1500	369.3	**0.07**	0.61	11.35	11.73	21.26	21.64
san1000	1000	498.0	**0.03**	1.48	8.73	8.87	16.81	17.02
san400_0.7_1	400	119.7	**0.03**	0.18	0.88	1.00	1.69	1.80
san400_0.9_1	400	39.9	**0.03**	0.14	0.35	0.46	0.67	0.79
frb59-26-1	1534	165.0	**0.06**	0.33	4.86	5.10	10.00	10.25
1et.2048	2048	22.0	**0.10**	0.46	1.17	1.46	1.97	2.26
1zc.4096	4096	45.0	**0.17**	0.85	4.06	4.63	7.86	8.44
2dc.2048	2048	492.6	**0.06**	0.45	21.25	21.59	36.15	36.41
dragon	150000	3.0	**17.80**	64.22	33.95	69.28	42.09	77.53
dragonsub	600000	3.0	**123.95**	390.90	193.38	400.14	169.59	377.23
buddha	1087716	3.0	**281.65**	795.98	448.92	854.27	447.78	859.55
fla	1070376	2.5	**299.70**	867.04	521.01	969.91	741.88	1193.48

on these instances, as well as 1et.2048 and brock400_4, but still not enough to make the random algorithm competitive with their greedy counterparts. It only improved the results obtained by GL in one case (1et.2048). On the positive side, the 3-improvement algorithm is reasonably fast. In most cases, it adds less than 20% to the time of RGL, and at most 80% (on johnson-32-2-4). Still, the minor gains and added complexity do not justify using 3-improvement within ILS.

5.3 Metaheuristics

Although local search can improve the results found by constructive heuristics, the local optima it finds are usually 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.

Tables 3, 4, and 5 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 nine runs of each algorithm (the numbers in parentheses indicate how many of these 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 averages are highlighted in bold.

Table 3. DIMACS family. For each algorithm, we show the minimum, average, and maximum solutions found over 9 runs, as well as the average running time in seconds. 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
C2000.9	2000	0.100	80	77	77.2	$78_{(2)}$	277	77	**77.9**	$79_{(2)}$	246
MANN_a45	1035	0.004	345	344	**344.7**	$345_{(6)}$	8	343	343.6	$344_{(5)}$	8
MANN_a81	3321	0.001	1100	1100	**1100.0**	$1100_{(9)}$	20	1097	1097.7	$1098_{(6)}$	27
brock400_1	400	0.252	27	25	25.0	$25_{(9)}$	26	25	**25.9**	$27_{(4)}$	27
brock400_2	400	0.251	29	25	25.4	$29_{(1)}$	26	25	**26.8**	$29_{(4)}$	26
brock400_3	400	0.252	31	25	30.3	$31_{(8)}$	27	31	**31.0**	$31_{(9)}$	23
brock400_4	400	0.251	33	25	31.2	$33_{(7)}$	27	33	**33.0**	$33_{(9)}$	20
hamming10-2	1024	0.010	512	512	**512.0**	$512_{(9)}$	24	512	**512.0**	$512_{(9)}$	13
keller6	3361	0.182	59	59	**59.0**	$59_{(9)}$	1385	59	**59.0**	$59_{(9)}$	1026
p_hat1500-1	1500	0.747	12	11	11.8	$12_{(7)}$	345	12	**12.0**	$12_{(9)}$	1207
san1000	1000	0.498	15	15	**15.0**	$15_{(9)}$	185	15	**15.0**	$15_{(9)}$	426

Table 4. Results for the CODE family with 2^{17} scans per arc

GRAPH				ILS				GLP			
NAME	n	DENS	BEST	MIN	AVG	MAX	TIME	MIN	AVG	MAX	TIME
1dc.1024	1024	0.046	94	93	**93.2**	$94_{(2)}$	31	93	93.1	$94_{(1)}$	42
1dc.2048	2048	0.028	172	170	**171.3**	$172_{(6)}$	76	170	**171.3**	$172_{(6)}$	95
1et.2048	2048	0.011	316	316	**316.0**	$316_{(9)}$	38	316	**316.0**	$316_{(9)}$	57
1tc.2048	2048	0.009	352	352	**352.0**	$352_{(9)}$	35	352	**352.0**	$352_{(9)}$	52
1zc.1024	1024	0.064	112	111	111.3	$112_{(3)}$	23	112	**112.0**	$112_{(9)}$	40
1zc.2048	2048	0.038	198	196	197.4	$198_{(6)}$	53	197	**197.7**	$198_{(6)}$	92
1zc.4096	4096	0.022	379	364	370.7	$379_{(1)}$	127	367	**373.0**	$379_{(1)}$	224
2dc.1024	1024	0.323	16	16	**16.0**	$16_{(9)}$	105	16	**16.0**	$16_{(9)}$	322
2dc.2048	2048	0.241	24	24	**24.0**	$24_{(9)}$	388	23	23.8	$24_{(7)}$	851

Together, the algorithms do rather well on these families. For almost all instances, the best known bound was found at least once. For all four exceptions (C2000.9 and the three largest frb instances) the best solution shown in the tables is within one unit of the best known [8].

The average solutions found by ILS and GLP are usually within 0.1 unit from one another. Among the exceptions, GLP found better solutions on nine (C2000.9, brock*, p_hat1500-1, and 1zc.*) and ILS on four (MANN*, 2dc.2048, and frb53-24-1). The brock instances are dense random graphs with a "hidden" larger clique. C2000.9 is also random, with larger cliques naturally hidden by the large value of n. GLP is clearly better at finding these cliques, probably because of 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 only 0.9 seconds on average.

Running times in the tables refer to full executions. When both algorithms found the same solution in every run, it makes sense to compare the average time

Table 5. Results for the SAT family with 2^{17} scans per arc

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**	$30_{(9)}$	20	30	**30.0**	$30_{(9)}$	28
frb35-17-1	595	0.158	35	35	**35.0**	$35_{(9)}$	30	35	**35.0**	$35_{(9)}$	41
frb40-19-1	760	0.143	40	40	**40.0**	$40_{(9)}$	42	40	**40.0**	$40_{(9)}$	57
frb45-21-1	945	0.133	45	44	**44.6**	$45_{(5)}$	65	44	**44.6**	$45_{(5)}$	79
frb50-23-1	1150	0.121	50	49	**49.1**	$50_{(1)}$	86	48	49.0	$50_{(1)}$	106
frb53-24-1	1272	0.117	53	51	**51.6**	$52_{(5)}$	98	51	51.1	$52_{(1)}$	121
frb56-25-1	1400	0.112	56	54	**54.1**	$55_{(1)}$	118	54	54.0	$54_{(9)}$	139
frb59-26-1	1534	0.108	59	57	**57.2**	$58_{(2)}$	137	57	57.1	$58_{(1)}$	161

to reach it (not shown in the tables). ILS is faster on 2dc.1024 (by a factor of 2), frb40-19-1 (3), and keller6 (13). The algorithms are essentially tied for 1tc.2048. GLP is faster for the remaining instances, usually by a factor of less than four. On san1000 and hamming10_2, GLP was at least 6 times faster.

Although our algorithm does well on these families, GLP is somewhat more robust on DIMACS and CODE. This is not the case for large, sparse graphs, to which we now turn our attention. Table 6 presents results for the MESH family. Because its graphs are much larger, we limit average number of arc scans to 2^{14}.

Table 6. Results for the MESH family with 2^{14} scans per arc

GRAPH		ILS				GLP			
NAME	n	MIN	AVG	MAX	TIME	MIN	AVG	MAX	TIME
dolphin	554	249	**249**	$249_{(9)}$	1	249	**249**	$249_{(9)}$	1
mannequin	1309	583	**583**	$583_{(9)}$	3	583	**583**	$583_{(9)}$	1
beethoven	4419	2000	**2002**	$2004_{(3)}$	9	1999	2001	$2004_{(1)}$	5
cow	5036	2333	2339	$2346_{(2)}$	11	2335	**2343**	$2346_{(6)}$	5
venus	5672	2668	2676	$2680_{(2)}$	11	2680	**2682**	$2684_{(4)}$	6
fandisk	8634	4057	4068	$4072_{(2)}$	18	4063	**4069**	$4073_{(1)}$	11
blob	16068	7232	7236	$7239_{(1)}$	36	7234	**7239**	$7242_{(1)}$	21
gargoyle	20000	8843	**8846**	$8849_{(1)}$	50	8841	8844	$8847_{(1)}$	32
face	22871	10203	**10206**	$10211_{(1)}$	51	10203	10205	$10207_{(1)}$	31
feline	41262	18791	18803	$18810_{(1)}$	105	18806	**18813**	$18822_{(1)}$	74
gameguy	42623	20625	20639	$20664_{(1)}$	104	20635	**20657**	$20676_{(1)}$	61
bunny	68790	32211	32228	$32260_{(1)}$	208	32221	**32246**	$32263_{(1)}$	184
dragon	150000	66399	**66417**	$66430_{(1)}$	506	66318	66331	$66343_{(1)}$	507
turtle	267534	122262	**122298**	$122354_{(1)}$	1001	122133	122195	$122294_{(1)}$	1185
dragonsub	600000	281942	281972	$282002_{(1)}$	2006	282100	**282149**	$282192_{(1)}$	2340
ecat	684496	321881	**321981**	$322040_{(1)}$	3191	321689	321742	$321906_{(1)}$	4757
buddha	1087716	480604	**480630**	$480664_{(1)}$	4773	478691	478722	$478757_{(1)}$	6795

Even though all instances come from the same application, results are remarkably diverse. 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, but 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.[2] 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. [16] computed lower bounds on the cover solutions for eleven of their original meshes (the ten largest in Table 6 plus fandisk). These can be easily translated into upper bounds for our (MIS) instances. On average, ILS is within 2.44% of these bounds (and hence of the optimum). The highest gap (3.32%) was observed for face, and the lowest for gameguy (1.22%). The gaps for GLP range from 1.13% (on gameguy) to 3.45% (on buddha), with an average of 2.48%.

Finally, Table 7 presents the results for ROAD, with the average number of scans per arc limited to 2^{12}. Here ILS has clear advantage.

Table 7. Results for the ROAD family with 2^{12} scans per arc

GRAPH			ILS				GLP			
NAME	n	DEG	MIN	AVG	MAX	TIME	MIN	AVG	MAX	TIME
ny	264346	2.8	131421	**131440**	131454$_{(1)}$	248	131144	131178	131213$_{(1)}$	293
bay	321270	2.5	166349	**166355**	166360$_{(1)}$	287	166215	166226	166250$_{(1)}$	372
col	435666	2.4	225741	**225747**	225754$_{(1)}$	395	225569	225586	225614$_{(1)}$	568
fla	1070376	2.5	549508	**549523**	549535$_{(1)}$	1046	548592	548637	548669$_{(1)}$	1505

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 1 shows the average solutions found for the two largest instances (buddha and fla) as the algorithms progress. GLP initially finds better solutions, but is soon overtaken by ILS. 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. The average solutions it found (after all 2^{12} scans per arc) were 480285 for buddha and 549422 for fla.

For comparison, we also show results for longer runs (2^{20} scans per arc, with nine different seeds) on C2000.9 (from the DIMACS family) and 1zc.4096 (from the CODE family). As before, GLP starts much better. On 1zc.4096, ILS slowly

[2] 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.

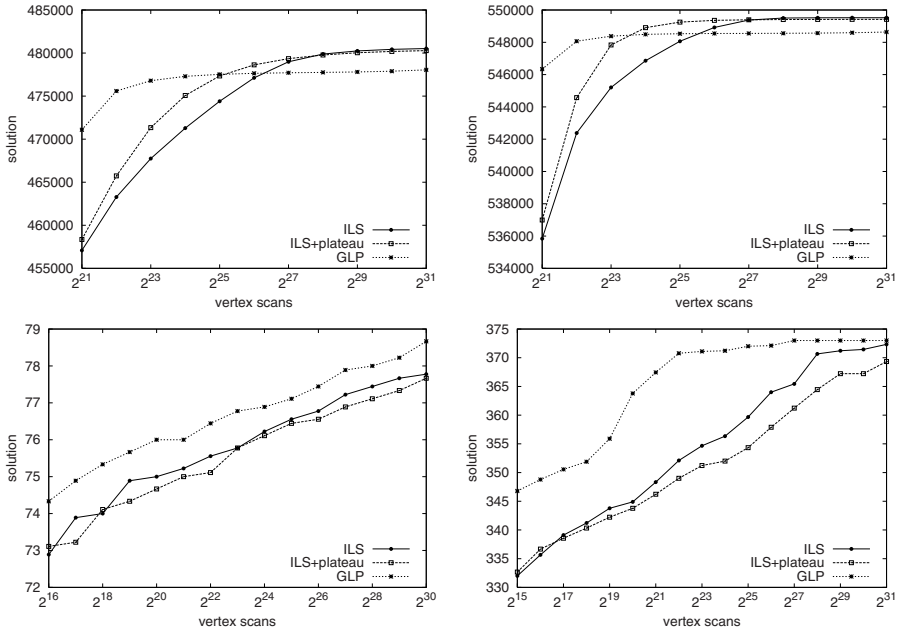

Fig. 1. Average solutions found as the number of scans per vertex increases. Results for buddha (top left), fla (top right), C2000.9 (bottom left), and 1zc.4096 (bottom right).

reduces the gap, but does not quite close it. On C2000.9, GLP is consistently better, even as the number of scans increases.

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 DIMACS, CODE, and SAT families. On large, sparse instances (meshes and road networks), its performance is consistently superior to that of GLP, particularly when the graph is irregular. For these large instances, however, we do not know exactly how far our method is from the optimal solution: there may be much 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 D. Nehab and P. Sander for sharing their paper and providing us with the MESH instances, and three anonymous referees for their helpful comments.

References

1. Battiti, R., Protasi, M.: Reactive local search for the maximum clique problem. Algorithmica 29(4), 610–637 (2001)
2. Bomze, I.M., Budinich, M., Pardalos, P.M., Pelillo, M.: The maximum clique problem. In: Du, D.Z., Pardalos, P.M. (eds.) Handbook of Combinatorial Optimization (Sup. Vol. A), pp. 1–74. Kluwer, Dordrecht (1999)
3. Butenko, S., Pardalos, P.M., Sergienko, I., Shylo, V., Stetsyuk, P.: Finding maximum independent sets in graphs arising from coding theory. In: Proceedings of the 2002 ACM Symposium on Applied Computing, pp. 542–546 (2002)
4. Butenko, S., Pardalos, P.M., Sergienko, I., Shylo, V., Stetsyuk, P.: Estimating the size of correcting codes using extremal graph problems. In: Pearce, C. (ed.) Optimization: Structure and Applications, Springer, Heidelberg (2008)
5. Demetrescu, C., Goldberg, A.V., Johnson, D.S.: 9th DIMACS Implementation Challenge: Shortest Paths (2006) (last visited on March 15, 2008), `http://www.dis.uniroma1.it/~challenge9`
6. Feo, T., Resende, M.G.C., Smith, S.: A greedy randomized adaptive search procedure for maximum independent set. Operations Research 42, 860–878 (1994)
7. Grosso, A., Locatelli, M., Della Croce, F.: Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem. J. Heuristics 10, 135–152 (2004)
8. Grosso, A., Locatelli, M., Pullan, W.: Simple ingredients leading to very efficient heuristics for the maximum clique problem. J. Heuristics (30 October, 2007), doi:10.1007/s10732-007-9055-x
9. Hansen, P., Mladenović, N., Urošević, D.: Variable neighborhood search for the maximum clique. Discrete Applied Mathematics 145(1), 117–125 (2004)
10. Johnson, D.S., Trick, M.: Cliques, Coloring and Satisfiability. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26. AMS (1996)
11. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press, New York (1972)
12. Katayama, K., Hamamoto, A., Narihisa, H.: An effective local search for the maximum clique problem. Information Processing Letters 95, 503–511 (2005)
13. Lourenço, H.R., Martin, O., Stützle, T.: Iterated local search. In: Glover, F., Kochenberger, G. (eds.) Handbook of Metaheuristics, pp. 321–353. Kluwer, Dordrecht (2003)
14. Pullan, W.J., Hoos, H.H.: Dynamic local search for the maximum clique problem. J. Artificial Intelligence Research 25, 159–185 (2006)
15. Richter, S., Helmert, M., Gretton, C.: A stochastic local search approach to vertex cover. In: Proceedings of the 30th German Conference on Artificial Intelligence (KI), pp. 412–426 (2007)
16. Sander, P.V., Nehab, D., Chlamtac, E., Hoppe, H.: Efficient traversal of mesh edges (submitted, 2008)
17. Sloane, N.J.A.: Challenge problems: Independent sets in graphs (2000) (last visited on March 15, 2008), `http://www.research.att.com/~njas/doc/graphs.html`
18. Xu, K.: BHOSLIB: Benchmarks with hidden optimum solutions for graph problems (2004) (last visited on March 15, 2008), `http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm`