A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set*

Thomas A. Feo §, *Mauricio G.C. Resende* † *and Stuart H. Smith* ‡

November 1989

(revised December 1992)

Abstract

An efficient randomized heuristic for maximum independent set is presented. The procedure is tested on randomly generated graphs having from 400 to 3500 vertices and edge probabilities from 0.2 to 0.9. The heuristic can be trivially implemented in parallel and is tested on an MIMD computer with 1, 2, 4 and 8 processors. Computational results indicate that the heuristic frequently finds the optimal or expected optimal solution in a fraction of the time required by implementations of simulated annealing, tabu search, and an exact partial enumeration method.

Key words: Randomized heuristic, maximum independent set, GRASP, parallel algorithm, simulated annealing, tabu search, random graphs.

Consider a graph $G = (V, E)$ with vertex set V and edge set E and its complement, $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(i, j) \notin E;\ i, j \in V, i \neq j\}$. An *independent set* (or vertex packing or stable set) is a vertex set whose elements are pair wise nonadjacent, i.e. a subset $S \subset V$ is independent if $\forall\ i, j \in S$, the edge $(i, j) \notin E$. A maximum independent set is an independent set of maximum cardinality. Conversely, $S \subset V$ is a *clique* if $\forall\ i, j \in S$, $(i, j) \in E$, and a *vertex cover* is a subset $S \subset V$ such that all edges of G have at least one endpoint in S. Finding a maximum independent set, a maximum clique or a minimum vertex cover of a graph are equivalent, and all three problems are known to be NP-hard (Garey & Johnson, 1979).

---

* To appear in *Operations Research*.

§ The University of Texas, Austin, TX 78712 USA. Partially supported by the Air Force Office of Scientific Research and the Office of Naval Research contract # F49620-90-C-0033.

† AT&T Bell Laboratories, Murray Hill, NJ 07974 USA.

‡ Purdue University, W. Lafayette, IN 47907 USA.

Operations Research, vol. 42, pp. 860-878, 1994

Practical applications of these optimization problems are abundant. They appear in information retrieval, signal transmission analysis, classification theory, economics, scheduling, experimental design, and computer vision. See (Avondo-Bodeno, 1962; Balas & Yu, 1986; Berge, 1962; Deo, 1974; Pardalos & Xue, 1992; Trotter Jr., 1973) for details. In addition, a method that finds large independent sets is often the essential subroutine in practical graph coloring heuristics (Bollobás & Thompson, 1985; Johnson, Aragon, McGeoch, & Schevon, 1991).

This paper presents a greedy randomized adaptive search procedure (GRASP) for maximum independent set. GRASP™ is a Trademark of Optimization Alternatives. A GRASP possesses two phases, a construction phase and a local search phase. Procedurally, the method is run many times with each run employing a different random number stream. The best overall solution is kept as the result.

In the first phase, a feasible solution is iteratively constructed, one element at a time. In the case of independent set problems, the elements are the vertices of the graph. At each construction iteration, the choice of the next element to be added is determined by ordering all elements in a candidate list with respect to a greedy function. This function assesses the benefit of selecting each element. The heuristic is considered adaptive if the benefits associated with every element are updated at each iteration to reflect the changes brought on by the selection of the previous element. The probabilistic component of a GRASP is implemented by randomly choosing one of the best candidates in the list, but not necessarily the top candidate. This choice technique allows for different solutions to be obtained each time the heuristic is executed, but does not necessarily compromise the power of the adaptive greedy component of the method.

As is the case for many deterministic methods, the solutions generated by a GRASP construction are not guaranteed to be locally optimal with respect to simple neighborhood definitions. Hence, it is often beneficial to apply a local search to attempt to improve each initial solution. Normally, a local optimization procedure such as a two-exchange is employed. While such procedures can require exponential time from an arbitrary starting point, it is observed that their efficiency significantly improves as the initial solutions improve. This effect is conjectured to be a principle factor accounting for the success of GRASP. Through the use of customized data structures and intelligent implementation, an efficient construction phase is created which proposes good initial solutions for quick local search. The result is that many GRASP solutions are generated in the same amount of time required for the local optimization procedure to converge from a single random start. Furthermore, the best of these GRASP solutions is observed to be significantly better than the solution obtained from a random starting point.

An especially appealing characteristic of GRASP is the ease with which it can be imple-

mented on a parallel processor in an MIMD environment. Each processor can be initialized with its own copy of the procedure, the instance data, and an independent random number sequence. The GRASP iterations are then performed in parallel with only a single global variable required to store the best solution found over all processors. As the availability and use of parallel computers increases, the effectiveness of GRASP implementations will improve, especially in comparison to inherently sequential approaches such as simulated annealing, tabu search, and genetic heuristics

GRASP has been applied successfully to several combinatorial optimization problems. These include set covering problems arising from the incidence matrix of Steiner triple systems (Feo & Resende, 1989), corporate acquisition of flexible manufacturing equipment (Bard & Feo, 1991), computer aided process planning (Bard & Feo, 1989), airline flight scheduling and maintenance base planning (Feo & Bard, 1989), scheduling of parallel machines (Laguna & González-Velarde, 1991) and p-hub location problems (Klincewicz, 1989).

Consider the family of undirected random graphs $G_{|V|,p}$ possessing $|V|$ vertices. Each possible edge from the edge set of the complete graph on $|V|$ vertices appears, independently of the inclusion of any other edge, with probability p. This family of graphs has been studied extensively (Bollobás, 1985). For empirical testing of the heuristic, this paper considers $|V| = \{400, 600, 1000, 1500, 2000, 3500\}$ and $p = \{0.2, 0.5, 0.6, 0.83, 0.9\}$ with special attention given to $G_{1000,.5}$. Let X_k be a stochastic variable denoting the number of independent sets of size k in an instance of $G_{1000,.5}$, then:

$$
\begin{array}{lcllcl}
E(X_{14}) & = & 4.23 \times 10^{3} & P(X_{14} = 0) & \leq & 0.02 \\
E(X_{15}) & = & 1.70 \times 10^{1} & P(X_{15} = 0) & \leq & 0.18 \\
E(X_{16}) & = & 3.19 \times 10^{-2} & P(X_{16} = 0) & \leq & 1.00 \\
E(X_{17}) & = & 2.18 \times 10^{-5} & P(X_{17} = 0) & \leq & 1.00
\end{array}
$$

It is highly probable, therefore, that independent sets of size 15, but not 16, exist in these graphs. However, finding independent sets of size 15 has challenged researchers (Bollobás & Thompson, 1985).

In Section 1, the heuristic is described in detail. This section includes the heuristic in pseudo-code, a description of the data structures used, and a complexity analysis of the method. Parallelization of the algorithm is discussed. In Section 2, the empirical results are summarized. The solution value and running time performance of serial and parallel implementations of GRASP are presented. An implementation of simulated annealing is given and its performance is reported. To have an idea of how our implementation compares with

other approximate and exact codes for maximum independent set, we have compared our GRASP code with the tabu search implementation STABULUS (Friden, Hertz, & de Werra, 1989) and an exact partial enumeration procedure (Carraghan & Pardalos, 1990) on the same serial computer. Concluding remarks are made in Section 3.

1. The Heuristic

As outlined in the introduction, a greedy randomized adaptive search procedure possesses four basic components: a greedy function, an adaptive search strategy, a probabilistic selection procedure, and a local search technique. These components are linked together into an iterative method that constructs a feasible solution one element at a time and then feeds the solution to the local search procedure. When applied to the maximum independent set problem, the independent sets are formed one vertex per construction iteration. The greedy function chosen in this implementation orders admissible vertices with respect to minimum admissible vertex degree. The term *admissible* refers to a vertex that is not adjacent to any vertex in the current independent set. Minimum degree is an intuitively effective criterion in constructing large independent sets. Selecting a vertex with the fewest admissible neighbors yields the largest set of admissible vertices the next construction iteration. The adaptive strategy is implemented each iteration by updating the degrees of admissible vertices. These updates reflect reductions in the set of admissible vertices due to the latest vertex added to the independent set. A candidate list of admissible vertices is constructed at each iteration. This list is ordered with respect to the minimum updated degree (adaptive greedy function) of each vertex. The list is then restricted to those vertices of lowest or very low degree. The probabilistic component of the heuristic randomly selects one of the candidates in the list, but not necessarily the candidate of least degree. It is important to note that this randomization leads to very different solutions each time the GRASP completes the construction of a maximal independent set. Furthermore, one can keep from compromising the adaptive greedy objective by the way in which the candidate list is restricted. For example, if two nodes of lowest degree are admissible during the first iteration, then selecting either one satisfies the greedy objective. However, each will likely lead to very different independent sets.

As discussed above, after a vertex is added to the current independent set, the degrees of the remaining admissible vertices are updated. Let V_i be the set of admissible vertices remaining at the i-th stage, i.e. after the i-th independent vertex has been selected. The number of degree updates performed at the i-th stage is proportional to $|V_i|^2$. If $|V_{i+1}| = p \times |V_i|$, with $0 < p < 1$, the amount of work required at the $(i+1)$-st stage is p^2 times that at the i-th stage. For example, if $p = \frac{1}{2}$, then after two nodes have been

selected, the remaining graph is approximately $\frac{1}{4}$ the size of the original, and the number of updates performed is on the order of $\frac{1}{16}$ as many as required during the first stage. Thus, after the first few vertices have been chosen, the number of degree updates becomes insignificant. To further exploit this property, our implementation fixes a small number of independent vertices before applying the construction phase. Let V_r be the set of admissible vertices remaining after the fixed vertices are added to the independent set. The GRASP is executed on the graph induced by V_r. From the previous discussion, it is evident that this conditioning results in significantly fewer computations per independent set, and thus, permits more GRASP iterations to be executed in a fixed amount of time.

In the following discussion, the number of fixed nodes is set to two. The strategy that is described can easily be generalized to allow conditioning on any number of vertices. However, the success of the strategy depends on conditioning on vertices belonging to at least one large cardinality independent set. Our approach selects pairs of vertices which leave a large number of vertices in the set V_r. This is accomplished by procedure `gentup` which is illustrated in Figure 1. This procedure takes as input a graph $G = (V, E)$ and two integers, `nlow` and `ntup`, and returns a set

$$\Lambda = \Big\{ \{v_1^1, v_2^1\}, \{v_1^2, v_2^2\}, \ldots, \{v_1^{\mathtt{ntup}}, v_2^{\mathtt{ntup}}\} \Big\}$$

of `ntup` independent pairs of vertices taken from the set of `nlow` vertices of lowest degree in G. The value of `ntup` may be restrictive in `gentup` since there may not exist `ntup` independent tuples among the vertices of `nlow` lowest degrees. Likewise, by choosing `ntup` sufficiently large, all possible independent pairs of vertices in the set V_{low} of `nlow` vertices of lowest degree in G can be considered. Let

$$P = \{i_1, i_2, \ldots, i_{|V|}\} \tag{1}$$

be a permutation of $\{1, 2, \ldots, |V|\}$ such that

$$\deg(v_{i_1}) \leq \deg(v_{i_2}) \leq \cdots \leq \deg(v_{i_{|V|}}).$$

The permutation P is computed in Line 1 of Figure 1. In Line 2, the set

$$V_{low} = \{v_{i_1}, v_{i_2}, \ldots, v_{i_{\mathtt{nlow}}}\}$$

of `nlow` vertices of lowest degree in G is defined. The set $\mathcal{P}$ of all pairs of independent sets of vertices in V_{low} is set empty and the counter of pairs, k, is initialized in Line 3. Loop 4-9 is repeated for every independent pair of vertices in V_{low}. In Lines 5-7, the set $\mathcal{P}$ of independent pairs is augmented. The *freedom* σ, of an independent pair of vertices $\{v_i, v_j\}$

```
procedure gentup(V, E, nlow, ntup, Λ)
1     Sort V by degree and set permutation P = {i_1, i_2, ..., i_|V|} of Equation (1);
2     V_low := {v_{i_1}, v_{i_2}, ..., v_{i_nlow}};
3     𝒫 := ∅; k := 0;
4     for v_i, v_j ∈ V_low and (v_i, v_j) ∉ E →
5         k := k + 1;
6         v_1^k := v_i; v_2^k := v_j;
7         𝒫 := 𝒫 ∪ {(v_1^k, v_2^k)};
8         σ({v_1^k, v_2^k}) := | {v ∈ V | (v, v_1^k) ∉ E and (v, v_2^k) ∉ E} |;
9     rof;
10    Sort 𝒫 by σ value and set permutation Q = {j_1, j_2, ..., j_|𝒫|} of Equation (2);
11    Λ := ∅;
12    ntup := min{ntup, |𝒫|};
13    for k = 1, ..., ntup →
14        Λ := Λ ∪ {(v_1^{j_k}, v_2^{j_k})};
15    rof
end gentup;
```

Figure 1: Pseudo-Code - Tuple generation procedure

is defined to be the number of vertices in the original graph not adjacent to either v_i or v_j, i.e.

$$\sigma(\{v_i, v_j\}) := |\ \{v \in V \mid (v, v_i) \notin E \text{ and } (v, v_j) \notin E\}\ | .$$

The freedom of each pair $\{v_i, v_j\}$ is computed in Line 8. Let

$$Q = \{j_1, j_2, \ldots, j_{|\mathcal{P}|}\} \tag{2}$$

be a permutation of $\{1, 2, \ldots, |\mathcal{P}|\}$ such that

$$\sigma(\{v_1^{j_1}, v_2^{j_1}\}) \geq \sigma(\{v_1^{j_2}, v_2^{j_2}\}) \geq \cdots \geq \sigma(\{v_1^{j_{|\mathcal{P}|}}, v_2^{j_{|\mathcal{P}|}}\}).$$

In Line 10, $\mathcal{P}$ is sorted by freedom and the permutation Q is constructed. Finally, in Lines 11-15, the set Λ of the `ntup` independent pairs of vertices of largest freedom is constructed. The GRASP for maximum independent set, denoted by procedure `mis` illustrated in Figure 2, will now be described. In addition to the graph $G = (V, E)$ and parameters `nlow` and `ntup`, procedure `mis` takes as input `alpha`, a candidate list percentage, `niter`, the number of GRASP iterations, `iscoff`, the local search cutoff, and `k`, the local search exchange

parameter. The local search and its parameters will be discussed shortly. The number `alpha` is used to construct the restricted candidate list. At each stage of the independent set construction, the candidate list is comprised of all admissible vertices whose degrees are within $1 +$ `alpha` times the degree of the minimum degree admissible vertex. The output from `mis` is $\mathcal{S}^*$, a *maximal* (not necessarily maximum) independent set.
In Line 1 of Figure 2, the vertex degrees are initialized, and in Line 2, the best solution ($\mathcal{S}^*$) is assigned the empty set. Procedure `gentup` is called in Line 3 to generate the independent vertex pairs for conditioning. The **for** loop 4-20 performs the conditioning for the current pair.

The **for** loop 5-19 performs the GRASP iterations. The current independent set, $\mathcal{S}$, is initialized to the fixed vertex pair in Line 6, and the admissible list, $\mathcal{A}$, is set up in Line 7. In Line 8, the updated degrees for vertices in $\mathcal{A}$ are calculated. To build the next independent set $\mathcal{S}$, loop 9-16 is repeated until there are no more admissible vertices. In Line 10, the smallest degree (`mindeg`) is computed from the set of admissible vertices. The candidate list $\mathcal{C}$ is built in Line 11. As mentioned, it is the subset of vertices of $\mathcal{A}$ having degree less than or equal to $(1 +$ `alpha`$) \times$ `mindeg`. The random choice of a vertex from $\mathcal{C}$ is made in Line 12, and this vertex is added to the current independent set $\mathcal{S}$ in Line 13. The set of admissible vertices is updated in Line 14, and their degrees are recomputed in Line 15. Once a maximal independent set is found, the local search `local` is applied in Line 17 if the cardinality of $\mathcal{S}$ is larger than `iscoff`. If the set $\mathcal{S}$ returned by `local` is the best found so far, it is saved as the incumbent $\mathcal{S}^*$ in Line 18.

A description of the local search used in Line 17 of `mis` will now be given. The purpose of this procedure is to explore different solutions that neighbor $\mathcal{S}$ and to converge to a local optimum. The neighborhood definition used here is `k`-exchange. The method attempts to find `k` vertices whose removal from the current independent set will allow at least `k` $+ 1$ vertices to be added back to the set.

The local search routine, `local`, is illustrated in Figure 3. The input is the graph $G = (V, E)$, an independent set $\mathcal{S}$, and the exchange parameter `k`. The output is an independent set $\mathcal{S}$ (perhaps unchanged). The procedure attempts to find a `k`-exchange which will improve the current independent set as follows:

1. Remove `k` nodes from $\mathcal{S}$, thus defining a reduced set $\mathcal{S}'$.

2. Let $\mathcal{A}$ be the set of admissible vertices with respect to $\mathcal{S}'$. Note that $\mathcal{A}$ will contain the `k` vertices that are removed.

3. Apply an exhaustive search procedure to find the *maximum* independent set $\mathcal{N}$, in the graph induced by $\mathcal{A}$.

```
procedure mis(V,E,nlow,ntup,alpha,niter,iscoff,k,S*)
1    for i = 1,...,|V| → deg(i) := 0 rof;
2    S* := ∅;
3    gentup(E,V,nlow,ntup,Λ);
4    for k = 1,...,ntup →
5        for i = 1,...,niter →
6            S := {v_1^k} ∪ {v_2^k};
7            A := V \ {{v_1^k} ∪ {v_2^k}} \ {w ∈ V | (w,v_1^k) ∈ E or (w,v_2^k) ∈ E};
8            Update deg(v), ∀v ∈ A;
9            do A ≠ ∅ →
10               mindeg := min{deg(v)|v ∈ A};
11               C := {v ∈ A | deg(v) ≤ (1 + alpha) × mindeg};
12               Select v* ∈ C at random;
13               S := S ∪ {v*};
14               A := A \ {v*} \ {w ∈ A | (w,v*) ∈ E};
15               Update deg(v), ∀ v ∈ A
16           od;
17           if |S| > iscoff → local(E,V,S,k) fi;
18           if |S| > |S*| → S* := S fi
19       rof
20   rof
end mis;
```

Figure 2: Pseudo-Code - GRASP for maximum independent set

```
procedure local(V, E, S, k)
1    for each k-tuple {v_{i_1}, ..., v_{i_k}} ∈ S →
2        S' := S \ {v_{i_1}, ..., v_{i_k}};
3        A := {w ∈ V | (w, v_i) ∉ E, ∀v_i ∈ S'};
4        Apply exhaustive search to the graph induced by A to find N;
5        if | N |> k →
6            S := S' ∪ N;
7            local(V, E, S, k);
8        fi;
9    rof;
end local;
```

Figure 3: Pseudo-Code - Independent set local search routine

4. If $\mid \mathcal{N} \mid > \mathtt{k}$, i.e. a larger set has been found, update $\mathcal{S} = \mathcal{S}' \cup \mathcal{N}$ and restart the procedure. Procedure `local` terminates when all possible `k`-exchanges have been considered for a particular $\mathcal{S}$ and no improvement is possible.

The computation time required by `local` increases exponentially with `k`, due to the exhaustive search procedure in the third step. In practice, `k` is set small, usually two or three. For this research, $\mathtt{k} = 2$ produced good results. The parameter `iscoff` also directly affects the time spent in the local search. Procedure `local` is applied only to those sets whose size is greater than `iscoff`. Setting $\mathtt{iscoff} = 0$ allows the local search to be applied to all independent sets. However, in practice, for $G_{1000,.5}$ it is very time consuming to apply `local` to sets of size 10 or less. Furthermore, improving the cardinality of such sets to 15 or 16 is unlikely.

1.1 Computational Complexity

The burdensome computational steps in procedure `mis` are revising the admissible list $\mathcal{A}$ and updating the degrees of vertices in $\mathcal{A}$. We focus our analysis of time and space complexity of this method to the case of dense graphs. Note that the set $\mathcal{A}$ is always decreasing in freedom. To efficiently implement these operations for dense graphs, we store $G = (V, E)$ as a $|V| \times |V|$ adjacency matrix and the list $\mathcal{A}$ as a one-dimensional array. This allows direct access to the edge structure of G. Furthermore, $\mathcal{A}$ can be compressed each iteration with a single pass. All other sets are stored as simple arrays.

Let $n = |V|$ and $m = |E|$. The initialization in Lines 1 and 2 as well as the call to `gentup` in Line 3 are performed only once and take $\mathcal{O}(n^2)$ time. The conditioning in Lines 6-8 is performed once per fixed pair and requires $\mathcal{O}(n^2)$ operations. Now, consider the construction of one independent set in loop 9-16. The **do** loop is executed $\mathcal{O}(n)$ times. Forming the candidate list and updating $\mathcal{S}$ and $\mathcal{A}$ can be performed in $\mathcal{O}(n)$ steps per iteration. Thus, a total of $\mathcal{O}(n^2)$ operations are required per independent set. In order to count the number of degree updates performed, it suffices to note that each edge is considered only once per independent set. This occurs when either of its incident vertices is removed from $\mathcal{A}$. Hence, the total number of updates performed is $\mathcal{O}(m)$, which for dense graphs is $\mathcal{O}(n^2)$. The total operation count per independent set constructed is therefore $\mathcal{O}(n^2)$, which for dense graphs is linear in the input size. The storage requirement is also $\mathcal{O}(n^2)$.

Although we have only implemented this heuristic for dense graphs, it can easily be applied to sparse instances. An efficient implementation would employ linked lists to store the adjacency structure, and a heap to store and update the vertex degrees. The total number of vertex updates per independent set remains $\mathcal{O}(m)$ (equal to $\mathcal{O}(n)$ for sparse graphs). However, by using a heap, each update can be executed in $\mathcal{O}(\log n)$ time. Thus, the total running time per independent set becomes $\mathcal{O}(n \log n)$.

The above complexity analysis omits the time spent in the local search routine. An analysis of the operation count of `local` requires `numimp`, the maximum number of local search improvements, and $f_{\mathtt{k}}$, the maximum number of vertices that become admissible when `k` nodes are removed from $\mathcal{S}$. The exhaustive search routine requires at most $2^{f_{\mathtt{k}}}$ operations per call. The total number of calls will be $\mathcal{O}\left(\mathtt{numimp}\binom{|\mathcal{S}|+\mathtt{numimp}}{\mathtt{k}}\right)$. Thus, the total time per call is $\mathcal{O}\left(\mathtt{numimp}\binom{|\mathcal{S}|+\mathtt{numimp}}{\mathtt{k}}2^{f_{\mathtt{k}}}\right)$. The running time of the local search in the worst case is exponential with respect to its input size. Although this appears ominous, the procedure, like many exponential time local search routines, is observed to be well behaved in practice. For $G_{1000,.5}$ random graphs and $\mathtt{k} = 2$, we found that $\mathtt{numimp} \leq 3$, $|\mathcal{S}| + \mathtt{numimp} \leq 16$ and $f_{\mathtt{k}} \leq 12$.

1.2. Parallel Implementation

GRASP can be easily and efficiently implemented on a MIMD parallel architecture with `nproc` processors. Two approaches to execute procedure `mis` in parallel are considered. One applies to the GRASP iterations (loop 5-19) and the other to the tuple conditioning (loop 4-20).

The parallel implementation of the GRASP loop is straightforward. The `niter` GRASP iterations are partitioned among the `nproc` processors along with independent random number sequences. Care is needed with this approach to assure that there is no intersection of

the `nproc` sequences.

The second strategy, parallelization of loop 4-20 of `mis`, avoids the issue of independent random number sequences. Here, the set of `ntup` tuples is partitioned among the `nproc` processors. All processors are given identical random seeds. Tuples are interleaved among processors according to tuple freedom, i.e. tuple $(\texttt{nproc} * k) + i$ is allocated to processor i, $i = 1, \ldots, \texttt{nproc}$, $k = 0, \ldots, \lfloor \texttt{ntup}/\texttt{nproc} \rfloor$, where $\lfloor x \rfloor$ denotes the largest integer less than or equal to x.

The second approach was implemented on an Alliant FX/80 parallel computer with 1, 2, 4 and 8 processors. The computational results are presented in the next section.

2. Experimental Results

The class of random graphs $G_{1000,.5}$ is the primary test bed for the computational experiments. One can expect a large portion of these problems to have maximum independent sets of size 15 and very few to have sets of size 16. The heuristic is tested with different parameters for 200 instances of these graphs on 1, 2, 4 and 8 processors. The procedure using eight processors is also tested on randomly generated graphs having from 400 to 2000 vertices and edge probabilities from 0.2 to 0.9. An implementation of simulated annealing is given later in this section and its performance is reported. For comparison with other approaches, we compare the GRASP code with a tabu search implementation (Friden et al., 1989) and an exact partial enumeration procedure (Carraghan & Pardalos, 1990) on the same computer. These results conclude the section.

Let A be the adjacency matrix of $G = (V, E)$, i.e. $a_{ij} = 1$ if $(i, j) \in E$ and $a_{ij} = 0$, otherwise. The test problems were generated by selecting each $a_{ij} = 1$ if and only if `rand(`s`)` evaluates to less than or equal to p, the probability of the existence of an edge. The function `rand(`s`)` is the portable FORTRAN random number generator given in (Schrage, 1979). It is passed an integer seed s in the interval $[0, 2^{31} - 1]$ by parameter and returns by value a real number in the interval (0,1) and by parameter a new integer seed. The use of this function allows easy reproduction of our test problems.

Most runs were carried out on an Alliant FX/80 parallel/vector computer. The Alliant uses a variant of the UNIX® Operating System. UNIX® is a Registered Trademark of UNIX Systems Laboratories, Inc. It is configured with 8 parallel processors, used for numerically intensive computations and 6 microprocessors used for other less intensive tasks, 256 Mbytes of main memory, 512 Kbytes of cache memory, and 3.1 Gbytes of disk storage. Each parallel processor has 8 vector registers, each capable of operating on 32 double precision numbers simultaneously. The Alliant machine operates in scalar mode at approximately 1 MFlops. The GRASP code was written in FORTRAN and compiled to run on the Alliant with the

Table I: Expected number of independent sets of size k for $G_{|V|,p}$.

k	$G_{400,.9}$	$G_{1000,.83}$	$G_{400,.6}$	$G_{600,.5}$	$G_{1000,.5}$	$G_{1500,.5}$	$G_{2000,.5}$	$G_{3500,.5}$	$G_{1000,.2}$
3	$1.06 \cdot 10^{4}$								
4	$1.05 \cdot 10^{3}$								
5	$8.32 \cdot 10^{0}$	$1.66 \cdot 10^{5}$							
6	$5.48 \cdot 10^{-3}$	$3.92 \cdot 10^{3}$							
7	$3.08 \cdot 10^{-7}$	$1.34 \cdot 10^{1}$							
8		$6.84 \cdot 10^{-3}$	$1.09 \cdot 10^{5}$						
9		$5.26 \cdot 10^{-7}$	$3.12 \cdot 10^{3}$						
10			$3.19 \cdot 10^{1}$						
11			$1.18 \cdot 10^{-1}$						
12			$1.61 \cdot 10^{-4}$	$5.51 \cdot 10^{4}$					
13				$6.09 \cdot 10^{2}$	$4.91 \cdot 10^{5}$				
14				$3.12 \cdot 10^{0}$	$4.23 \cdot 10^{3}$	$1.27 \cdot 10^{6}$			
15				$7.43 \cdot 10^{-3}$	$1.70 \cdot 10^{1}$	$7.70 \cdot 10^{3}$			
16				$8.29 \cdot 10^{-5}$	$3.19 \cdot 10^{-2}$	$2.18 \cdot 10^{1}$	$2.22 \cdot 10^{4}$		
17					$2.81 \cdot 10^{-5}$	$2.90 \cdot 10^{-2}$	$3.95 \cdot 10^{0}$	$5.50 \cdot 10^{4}$	
18						$1.82 \cdot 10^{-5}$	$3.32 \cdot 10^{-3}$	$8.10 \cdot 10^{1}$	
19							$1.32 \cdot 10^{-6}$	$5.68 \cdot 10^{-2}$	
20								$1.89 \cdot 10^{-5}$	
36									$1.25 \cdot 10^{5}$
37									$1.06 \cdot 10^{3}$
38									$6.99 \cdot 10^{0}$
39									$3.58 \cdot 10^{-2}$
40									$1.43 \cdot 10^{-4}$

Alliant FORTRAN compiler with flags `-O -DAS`. No special care was taken to vectorize the code or to implement it in parallel other than the concurrent calls to `mis` using different tuple data. All times reported are user times given by the system call `times()`.

The computational study where we compare GRASP with other approaches was conducted on serial Silicon Graphics machines. We describe those machines later in the paper.

Tables I–II display, for all problem classes tested in this study, the expected number of independent sets of size k and an upper bound on the probability that there are no independent sets of size k.

2.1. Varying `nlow` and `niter` on $G_{1000,.5}$

For the class $G_{1000,.5}$ we considered 200 instances, corresponding to the random seeds $s = 1, 2, \ldots, 200$. Recall that X_k denotes a stochastic variable representing the number of independent sets of size k. The expectation of X_k, $E(X_k)$, and an upper bound, $U(X_k)$, on the probability that $X_k = 0$ are given in Tables I-II (Bollobás, 1985).

From Tables I-II one can see that independent sets of size 14 are abundant in $G_{1000,.5}$, while sets of size 16 are rare. Therefore, in the computational experiment, we search for sets of size 15 or 16, halting the program when such a set is found.

Table II: Upper bound on probability that the number of independent sets of size k equals 0, for $G_{|V|,p}$.

k	$G_{400,.9}$	$G_{1000,.83}$	$G_{400,.6}$	$G_{600,.5}$	$G_{1000,.5}$	$G_{1500,.5}$	$G_{2000,.5}$	$G_{3500,.5}$	$G_{1000,.2}$
3	.001								
4	.006								
5	.164	.001							
6	1.000	.003							
7		.096							
8		1.000	.019						
9			.039						
10			.165						
11			1.000						
12				.033					
13				.062	0.15				
14				.967	.023	.009			
15				1.000	.178	.012			
16					1.000	.111	.010		
17						1.000	.434	.003	
18							1.000	.022	
19								1.000	
36									.374
37									.999
38									1.000

We ran four combinations of the parameters `nlow` and `niter`,

$$(\texttt{nlow}, \texttt{niter}) = \{(20, 50), (20, 100), (50, 50), (50, 100)\}.$$

The remaining parameters were set as follows: tuple size limit `ntup` = 400, candidate list parameter `alpha` = 0.1, local search cutoff parameter `lscoff` = 11, local search exchange parameter `k` = 2 and `nproc` = 8 processors. All runs with `nlow` = 50 had the number of tuples limited by the parameter `ntup` = 400.

Table III summarizes the results of running our code on 8 parallel processors. The four columns correspond to different settings of `nlow` and `niter`. The first three rows display the number of instances out of the 200 tested where the code found $\mathcal{S}^*$ to be of size 14, 15 and 16, respectively. The remaining rows provide mean, minimum and maximum values for preprocessing, total time, local search and number of tuples considered until the best solution was found. All times are given in user seconds. These rows exclude runs in which the algorithm failed to find independent sets of size greater than or equal to 15. Figure 4 plots cumulative instances for which a solution of 15 or 16 was found, i.e. an (x, y) entry implies that solutions of size 15 or 16 have been found for y of the 200 instances by x CPU seconds. Entries for instances in which no 15 or 16 was found are excluded from the plots.

The main observation is that the number of successful searches (i.e. searches that find an independent set of size 15 or 16) increases as the values of `nlow` and `niter` increase. The average time to run `mis` also increases as these parameters increase. However, this increase is due to the fact that more instances are solved with the larger parameters (and the harder

Table III: Sensitivity to `nlow` and `niter` on 8 processors. Computed for runs that found a set of size 15 or 16.

(`nlow`,`niter`)		(20,50)	(20,100)	(50,50)	(50,100)
Sets found with:	$\lvert\mathcal{S}^*\rvert = 14$	42	27	3	0
	$\lvert\mathcal{S}^*\rvert = 15$	157	172	194	198
	$\lvert\mathcal{S}^*\rvert = 16$	1	1	3	2
Construction time:	average	58.24	91.79	108.89	128.34
	minimum	2.84	1.57	2.53	2.49
	maximum	179.23	343.06	653.26	1143.53
Local search time:	average	46.98	81.02	88.11	113.03
	minimum	0.61	1.09	0.35	0.36
	maximum	143.78	322.94	541.44	998.65
Tuples considered:	average	38.42	33.97	69.27	46.71
	minimum	8	8	8	8
	maximum	113	114	397	392

instances tend to take longer to solve).

2.2. Parallel Algorithms

Now we illustrate the speedup obtained by using parallel processors. Again, we consider the class $G_{1000,.5}$ and generate the same 200 instances as in Table III. We ran the code, distributing the tuples to 1, 2, 4 and 8 processors and using the parameter settings `nlow` = 50 and `niter` = 100. The remaining parameters were kept the same as in the runs summarized in Table III.

Table IV summarizes the results of running our code on a single processor and on 2, 4 and 8 parallel processors. On all runs, the code found independent sets of size 15 or 16. Even though `gentup` was not implemented in parallel, we have included entries for preprocessing in the table. The rows are similar to those of Table III. Again, all times are given in user seconds. Figure 4 plots cumulative instances by time for the runs in Table IV. GRASP implements in parallel quite naturally. However, since random number sequences change when the number of processors is changed, an individual instance may experience speedups well below or above the ratio of processors. We have, therefore, computed averages over the 200 instances generated with seeds $s = 1, 2, \ldots, 200$. Figure 5 plots the average speedup for the runs, going from 1 to 2 to 4 to 8 processors. The efficiency (speedup divided by the

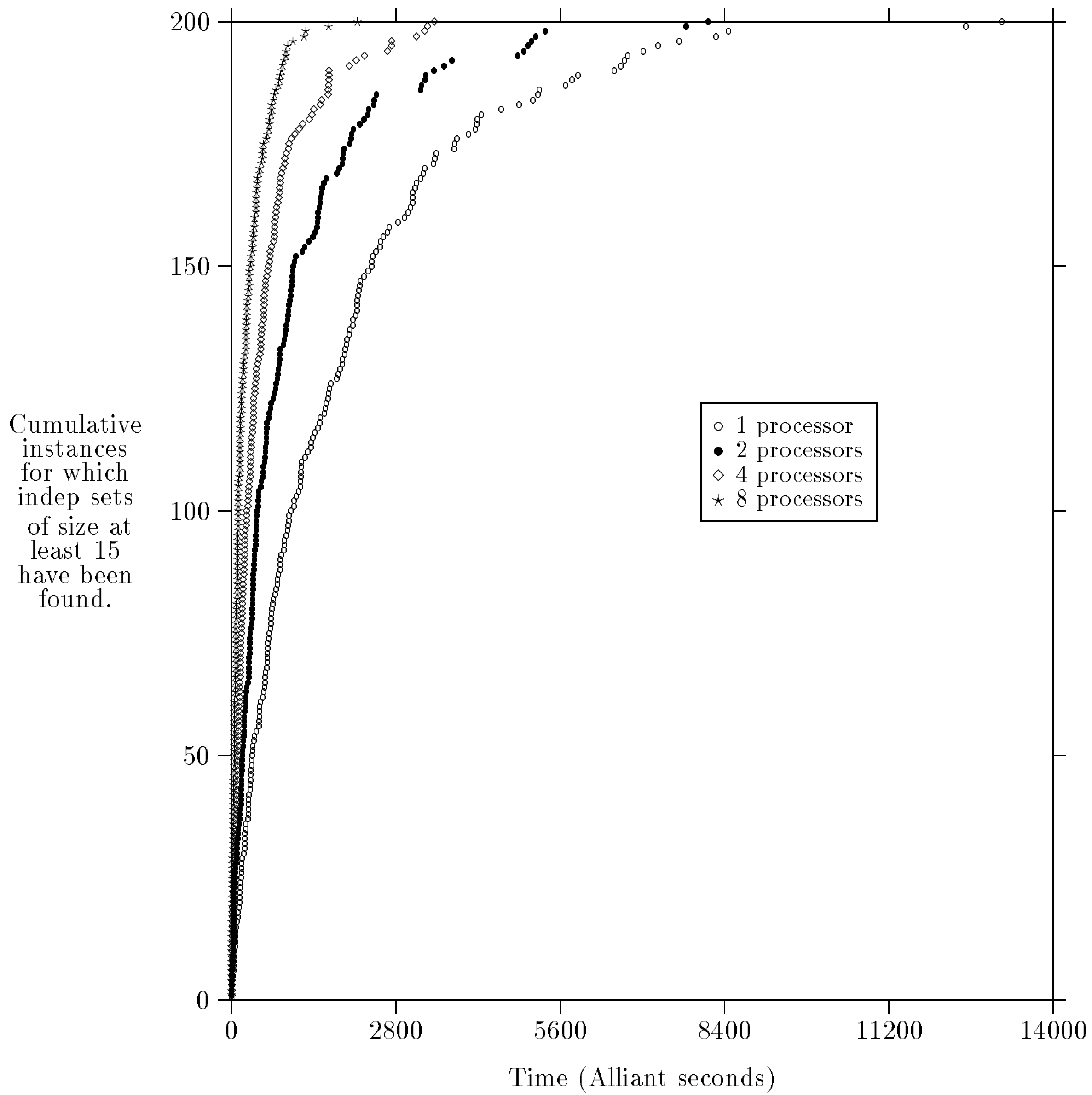

Figure 4: Running times ($|V| = 1000, p = 0.5,$ `nlow` $= 50,$ `niter` $= 100,$ `nproc` $= 1, 2, 4, 8$)

Table IV: 1, 2, 4 and 8 parallel processors

nproc		1	2	4	8
Construction time:	average	1190.22	511.90	270.49	128.34
	minimum	3.40	2.82	2.68	2.49
	maximum	8775.05	4340.14	1886.19	1143.53
Local search time:	average	617.55	439.44	234.41	113.03
	minimum	0.99	1.15	0.84	0.36
	maximum	4360.06	3787.99	1569.93	998.65
Tuples considered:	average	43.11	46.17	49.17	46.71
	minimum	1	2	4	8
	maximum	324	386	339	392

ratio of processors) of going from one to eight processors was over 93.6%.

2.3. Other Random Graphs

In this Section, we consider other classes of random graphs. We begin with the class $G_{400,.6}$ that has an independent set distribution described in Tables I-II. Note that an independent set of size 11 is rare. We generated 200 instances, with random seeds $s = 1, 2, \ldots, 200$ and searched for independent sets of size greater than or equal to 10. The parameters were set as they were for the 8 processor runs of Section 2.1 with the exception of the local search cutoff parameter `iscoff`, that was set to 9. Instead of conditioning on two vertices being in the independent set, we condition on a single vertex.

Table V and Figure 6 summarize these runs. In 197 of the 200 instances an independent set of size 10 was found. In 3 instances the best set found had size 11.

We now consider the class of dense graphs $G_{400,.9}$ with independent set distribution described in Tables I-II. We generated 200 instances, with random seeds $s = 1, 2, \ldots, 200$ and searched for independent sets of size greater than or equal to 5. The parameter settings were similar to those of the previous class of problems with the exception of the local search cutoff parameter `iscoff`, that was set to 4.

Table V and Figure 7 summarize these runs. In 196 of the 200 instances an independent set of size 5 was found. In 4 instances the best set found had size 4.

We now consider the class of random graphs $G_{1000,.83}$. Edge probability $p = 0.83$ was chosen instead of $p = 0.8$ because the former class of graphs has a more accentuated cutoff in expected number of independent sets (Tables I-II) than does the latter. Note that an

Table V: Summary of GRASP runs on $G_{400,.6}$, $G_{400,.9}$ and $G_{1000,.83}$.

		$G_{400,.6}$	$G_{400,.9}$	$G_{1000,.83}$
Sets found with:	$\lvert\mathcal{S}^*\rvert = 11$	3		
	$\lvert\mathcal{S}^*\rvert = 10$	197		
	$\lvert\mathcal{S}^*\rvert = 7$			190
	$\lvert\mathcal{S}^*\rvert = 6$			10
	$\lvert\mathcal{S}^*\rvert = 5$		196	
	$\lvert\mathcal{S}^*\rvert = 4$		4	
Construction time:	average	0.53	0.39	13.31
	minimum	0.31	0.11	1.10
	maximum	5.93	2.77	53.31
Local search time:	average	0.12	0.02	13.64
	minimum	0.00	0.00	0.01
	maximum	1.59	0.03	52.36
Tuples considered:	average	8.02	11.52	66.05
	minimum	8	8	8
	maximum	12	46	251

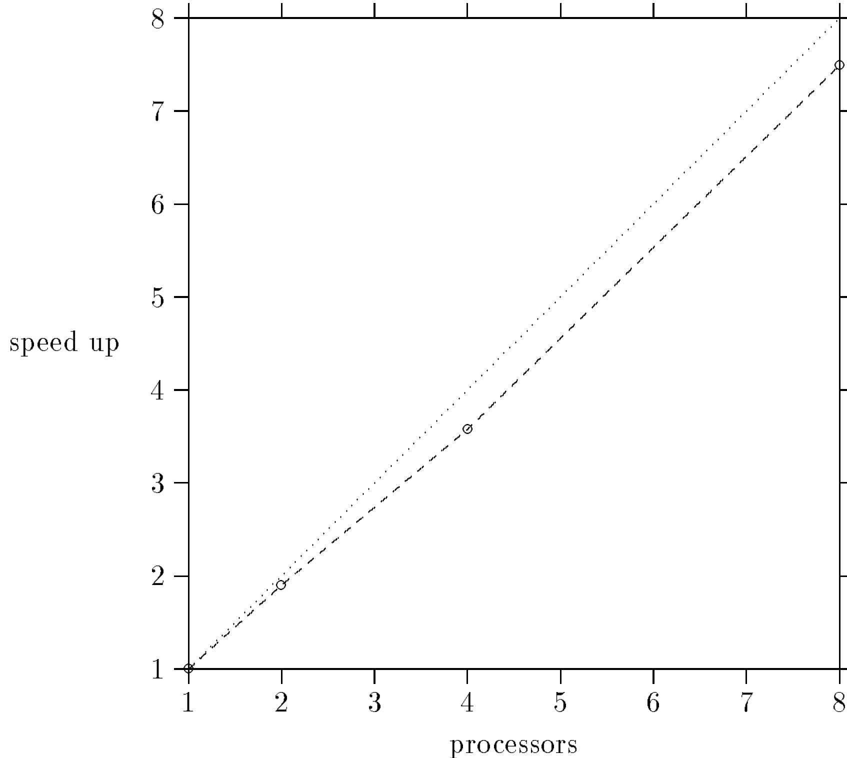

Figure 5: Average Speedup

independent set of size 8 is very rare. We generated 200 instances, with random seeds $s = 1, 2, \ldots, 200$ and searched for independent sets of size greater than or equal to 7. The parameters were set as they were for the 8 processor runs of Section 2.1 with the exception of the local search cutoff parameter `iscoff`, that was set to 5.

Table V and Figure 8 summarize these runs. In 190 of the 200 instances an independent set of size 7 was found. No set of size 8 or greater was found. In the 10 instances where no set of size 7 was found, independent sets of size 6 were encountered.

Next, we consider the class of random graphs $G_{1000,.2}$ which is characterized in Tables I-II. Note that independent sets of size 39 are very rare. We generated 200 instances, with random seeds $s = 1, 2, \ldots, 200$ and searched for independent sets of size greater than or equal to 37. The parameters were set as they were for the 8 processor runs of Section 2.1 with the exception of the local search cutoff parameter `iscoff`, that was set to 32, and we conditioned on four vertices instead of two.

Table VI and Figure 9 summarize these runs. In 108 of the 200 instances an independent set of size 37 was found and in 3 instances we found a set of size 38. No set of size 39 or greater was found. In the 89 instances where no set of size 37 or larger was found, independent sets of size 36 were encountered.

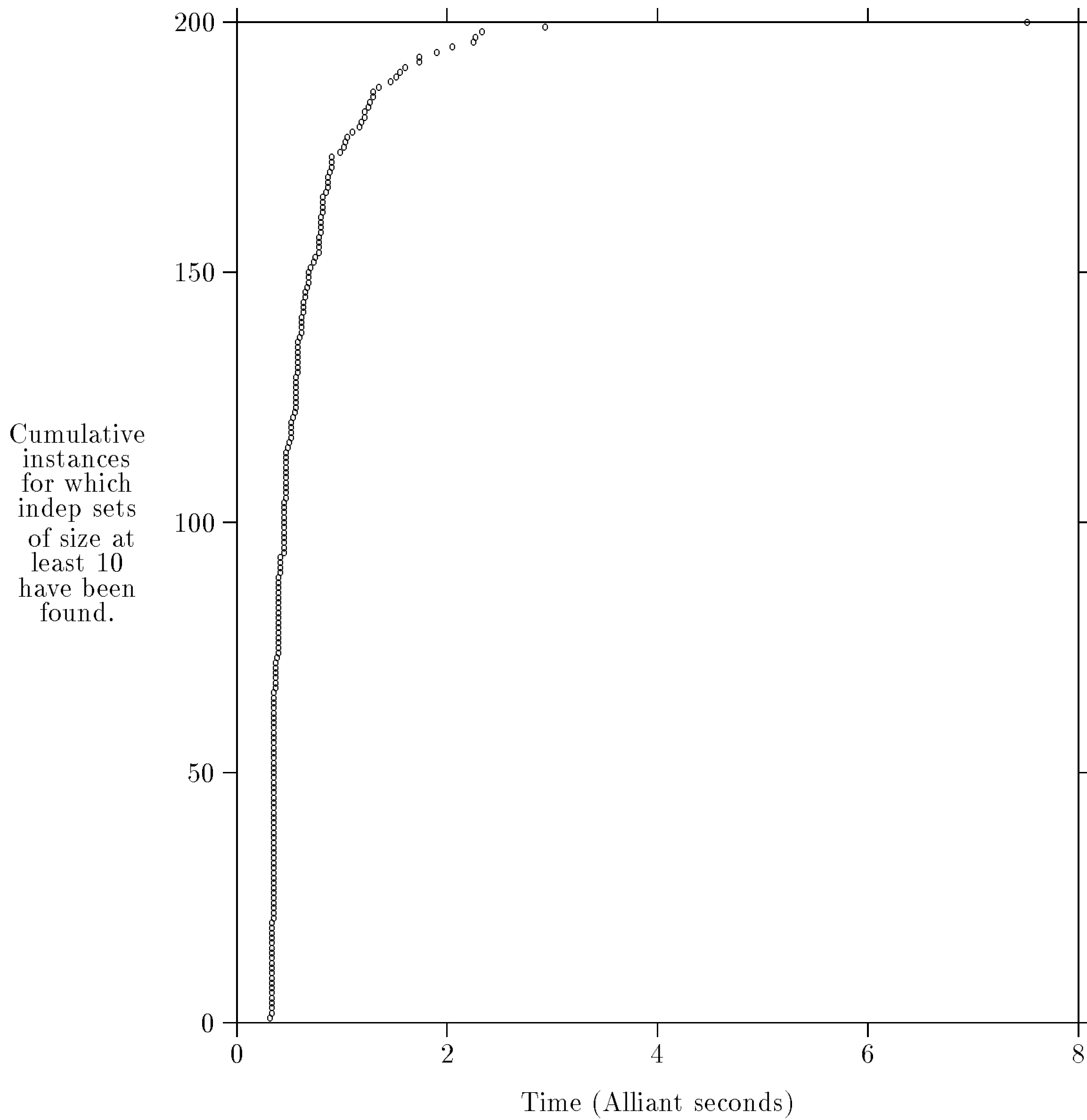

Figure 6: Running times ($|V| = 400, p = 0.6,$ `nlow` $= 50,$ `niter` $= 100,$ `nproc` $= 8$)

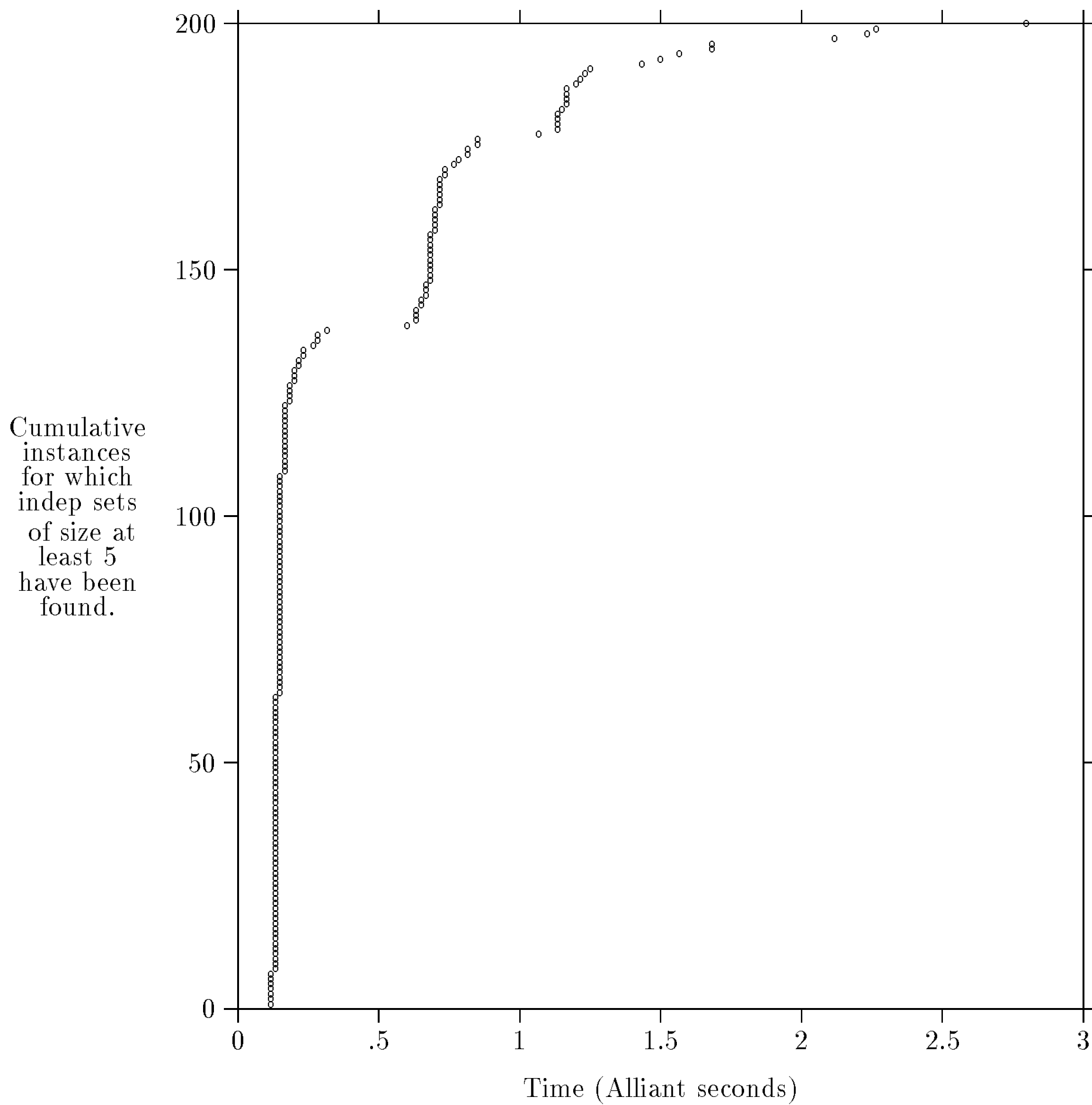

Figure 7: Running times ($|V| = 400, p = 0.6, \texttt{nlow} = \texttt{50}, \texttt{niter} = 100, \texttt{nproc} = 8$)

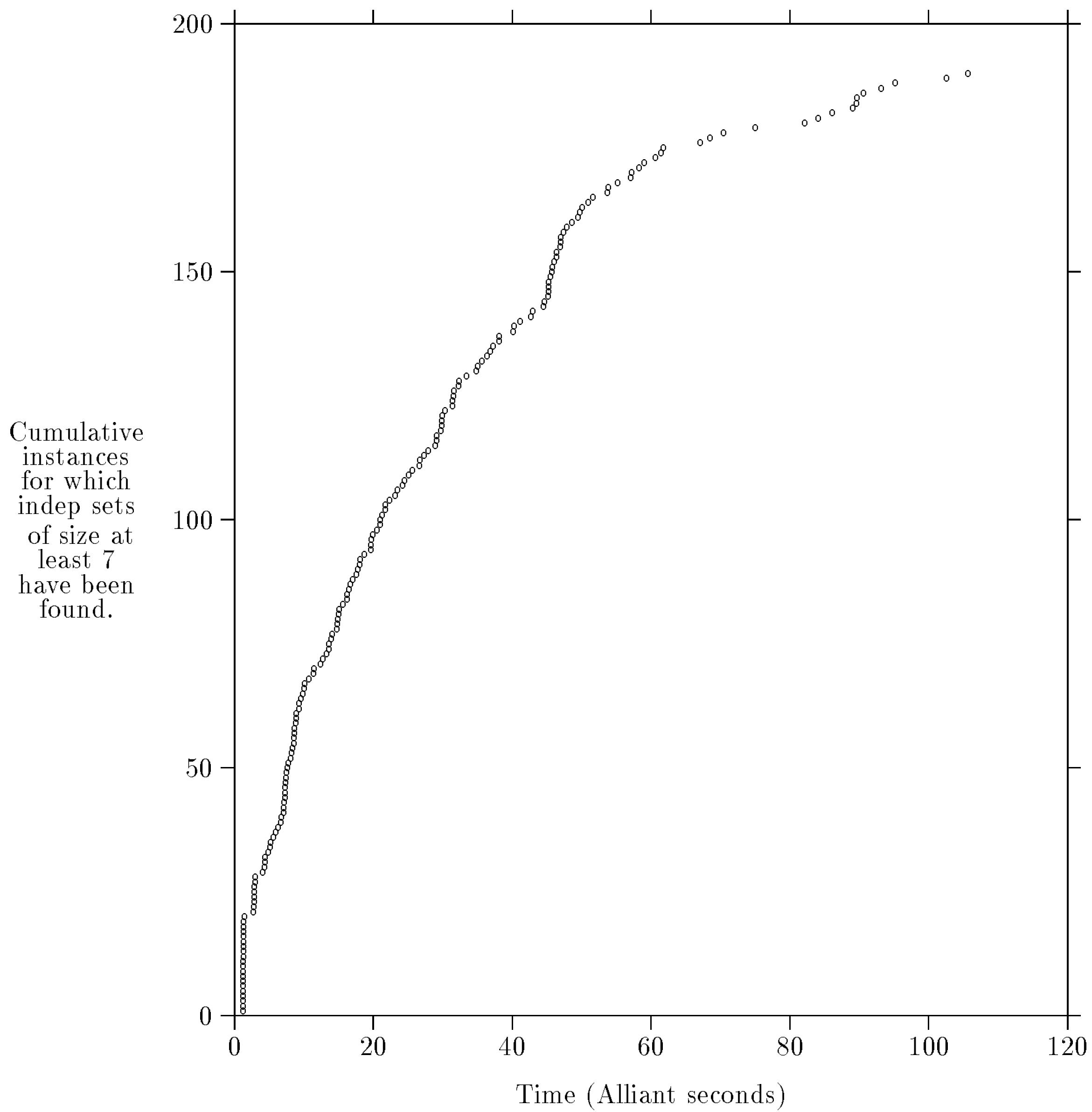

Figure 8: Running times ($|V| = 1000, p = 0.83, \texttt{nlow} = \texttt{50}, \texttt{niter} = 100, \texttt{nproc} = 8$)

Table VI: Summary of GRASP runs on $G_{1000,.2}$, $G_{2000,.5}$ (searching for sets of size 16) and $G_{2000,.5}$ (searching for sets of size 17).

		$G_{1000,.2}$	$G_{2000,.5(\text{size } 16)}$	$G_{2000,.5(\text{size } 17)}$
Sets found with:	$\|\mathcal{S}^*\| = 38$	3		
	$\|\mathcal{S}^*\| = 37$	107		
	$\|\mathcal{S}^*\| = 36$	89		
	$\|\mathcal{S}^*\| = 17$			8
	$\|\mathcal{S}^*\| = 16$		50	42
Construction time:	average	1201.90	84.64	2294.58
	minimum	6.30	11.36	272.88
	maximum	3851.04	308.02	6338.85
Local search time:	average	394.25	123.90	4315.00
	minimum	1.28	4.19	512.00
	maximum	1799.94	471.43	11973.40
Tuples considered:	average	126.95	22.46	256.40
	minimum	8	8	32
	maximum	397	72	704

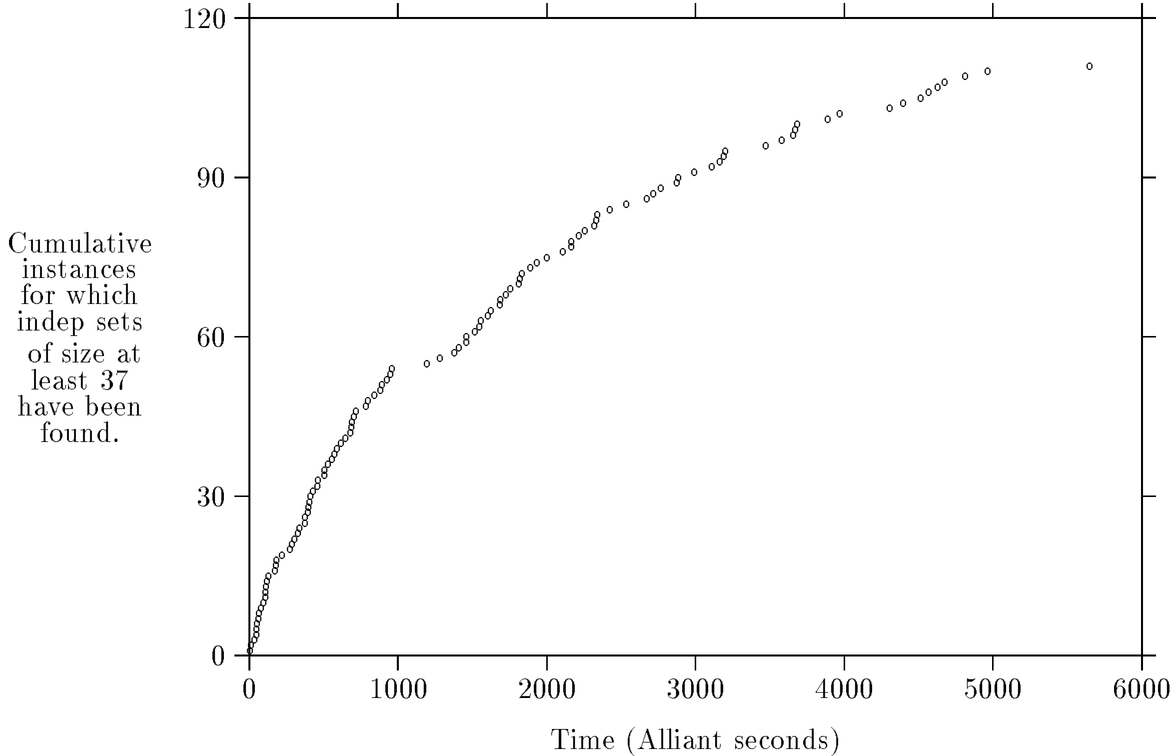

Figure 9: Running times ($|V| = 1000, p = 0.2,$ `nlow` $= 50,$ `niter` $= 100,$ `nproc` $= 8$)

We finally consider the class of random graphs $G_{2000,.5}$ which is characterized in Tables I-II. These graphs have on average 1,000,000 edges. One should expect to find sets of size 16 or 17, since independent sets of size 18 are very rare. We generated 50 instances, with random seeds $s = 1, 2, \ldots, 50$ and searched for independent sets of size greater than or equal to 16. The parameters were set as they were for the 8 processor runs of Section 2.1 with the exception of the local search cutoff parameter `iscoff`, that was set to 12 and we conditioned on three vertices instead of the usual two with a maximum of 800 tuples.

Table VI and Figure 10 summarize these runs. In a separate experiment, using the same 50 instances, we also searched for sets of size 17 or greater. Parameter settings were identical to those of the previous experiment with the exception of `niter` $= 250$ and `iscoff` $= 12$. Table VI and Figure 11 summarize these runs.

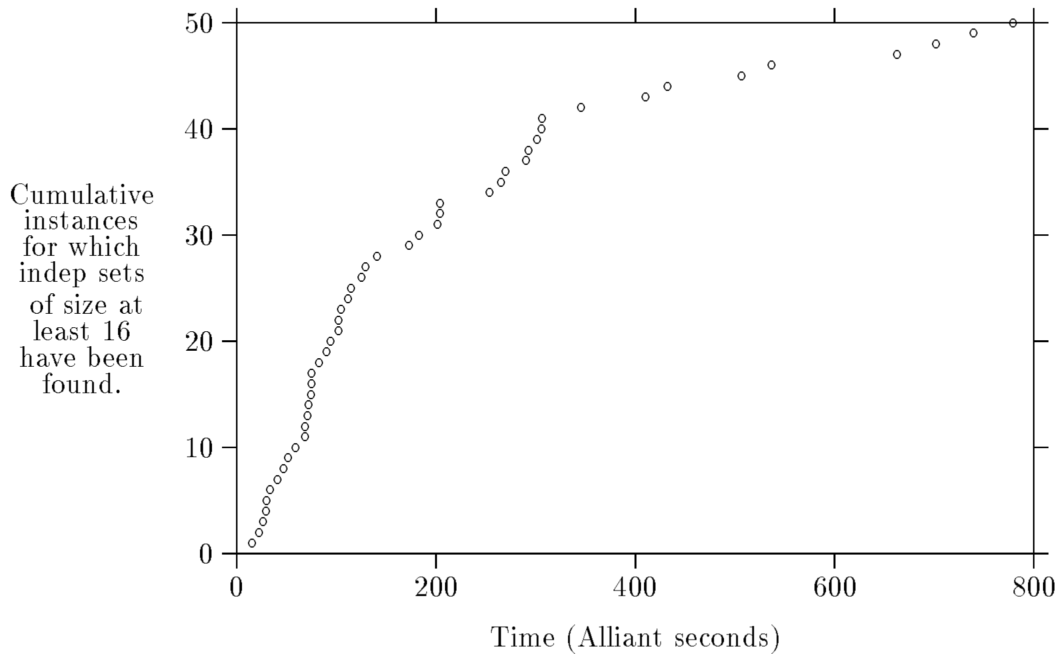

Figure 10: Running times ($|V| = 2000, p = 0.5,$ nlow = 50, niter = 100, nproc = 8). Searching for sets of size 16.

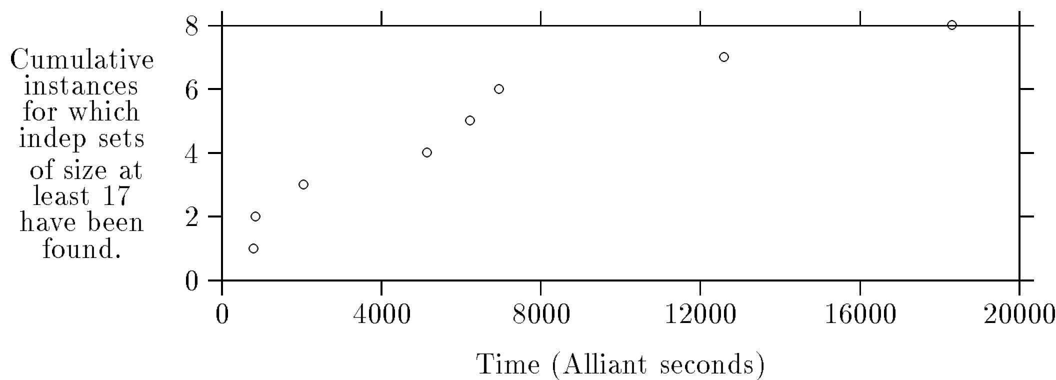

Figure 11: Running times ($|V| = 2000, p = 0.5,$ nlow = 50, niter = 250, nproc = 8). Searching for sets of size 17.

2.4. Simulated Annealing

The solutions obtained by GRASP are now compared to a randomized algorithm that has received much attention in recent years: simulated annealing. For a detailed discussion of simulated annealing, see (Aarts & Korst, 1989). We briefly describe the implementation used in this study.

Let K and $\bar{K}$ be a partition of the vertex set V of G. Let E_K be the edge set of G_K, the graph induced by K. The objective function that is to be minimized by the annealing algorithm is

$$f(K) = -|K| + \lambda|E_K|,$$

where λ is a positive integer constant. If $|E_K| = 0$, K is an independent set. The objective function $f(K)$ is minimized when $|K|$ is maximized and $|E_K| = 0$, corresponding to a maximum independent set. A *solution* is completely specified by the set K. Let $v \in V$. A *neighbor* solution $K'(v)$ to K is

$$K'(v) = \begin{cases} K \cup \{v\} & \text{if } v \in \bar{K} \\ K \setminus \{v\} & \text{if } v \in K. \end{cases}$$

Given a solution K, the simulated annealing algorithm randomly selects $v \in V$ and generates a neighbor solution $K'(v)$. The neighbor is accepted (i.e. made the current solution) if it does not worsen the objective function, i.e. if

$$f(K'(v)) \leq f(K).$$

On the other hand, if the neighbor worsens the objective function, i.e. if

$$f(K'(v)) > f(K),$$

the neighbor solution is accepted with probability

$$e^{-\Delta/T_k}, \tag{3}$$

where

$$\Delta = f(K'(v)) - f(K)$$

and T_k is a control parameter called the *temperature*. If the temperature is high, the procedure accepts neighbors that worsen the objective function with high probability. As the temperature decreases, so does the probability of accepting a neighbor that degrades the objective function.

The temperature is decreased according to a *cooling schedule*. A temperature *cycle* is a period where the temperature remains constant. In each cycle, at most $\overline{TRL}$ *trials* are

carried out. A trial consists of selecting a neighbor of the current solution. A *change* takes place if the neighbor is accepted. At most $\overline{CHG}$ changes are allowed for a fixed temperature. In the cooling schedule we chose to implement, the new temperature is determined by

$$T_{k+1} = T_k * \alpha,\ k = 0, \ldots,$$

where $0 < \alpha < 1$. The algorithm terminates when $\overline{FRZ}$ temperature cycles go by with less than $\underline{CHG}$ changes in the solution K. Figure 12 illustrates the simulated annealing algorithm implemented for this experiment. In this implementation, the solution is represented by the array Γ, where $\Gamma(i) = 1$ if $i \in K$ and $\Gamma(i) = 0$ if $i \in \bar{K}$.

Procedure `simanl` takes as input the graph $G = (V, E)$, an initial temperature T_1 and the parameters $\overline{FRZ}, \overline{CHG}, \overline{TRL}, \alpha$ and $\underline{CHG}$ and returns an array Γ with indicator variables for each vertex in G as well as the value of the objective function corresponding to Γ. In Line 1, an initial random partition of V is made. Every vertex is assigned a 1 ($\in K$) or a 0 ($\in \bar{K}$). The objective function value of the initial solution is computed in Line 2, where the stopping criterion counter, $frzcnt$, and the temperature cycle counter, k, are initialized. This is the only time in the procedure where the objective function is computed from scratch. In all other instances, the objective function is updated from its current value.

Since the computation of the exponential (3) is expensive, the following table lookup scheme is used to compute the acceptance probabilities. This scheme was inspired by the one described in (Johnson et al., 1991). Since $e^{-10} = 0.00005$ is small, a hill climbing move is rejected if $\Delta > 10T_k$. Every time the parameter T_k is decreased, a lookup table is built with the values $e^{-1/T_k}, e^{-2/T_k}, \ldots, e^{-\lfloor 10T_k \rfloor / T_k}$. Since Δ is an integer, all values of $e^{-\Delta/T_k}$, for $1 \leq \Delta \leq 10T_k$ are in the table. In Line 3, the initial table is built.

The loop 4-26 is repeated until $\overline{FRZ}$ temperature cycles go by with less than $\underline{CHG}$ in the solution. In Line 5, the temperature cycle counter is incremented and change and trial counters, `chg` and `trl`, are set to 0. Loop 6-22 is repeated until $\overline{CHG}$ changes in the state vector Γ occur or $\overline{TRL}$ tentative solutions (trials) are generated. In Line 7, the trial counter is incremented and in Line 8 a vertex v is chosen at random from V. In Lines 9-10, a tentative state vector, $\bar{\Gamma}$ is defined by flipping the value of $\Gamma(v)$. The objective function value F_{new} of the tentative solution is computed in Line 11. In the following four lines, if the new solution is better than the current state, it replaces the current solution. The objective function value is then recorded, and the change counter is incremented. On the other hand (Lines 16-21), if the solution is worse but the difference is less than $10T_k$, it is still accepted with probability $e^{-(F_{new} - F_{old})/T_k}$. Here, all that is needed is a lookup on the exponential table. The objective function value is recorded and the solution change counter is incremented in Line 19.

procedure simanl($V, E, T_1, \overline{FRZ}, \overline{CHG}, \overline{TRL}, \alpha, \underline{CHG}, \Gamma, F_{new}$)

```
1    for i = 1,...,|V| → Γ(i) := irand(0,1) rof;
2    F_old := f(Γ); frzcnt := 0; k := 0;
3    make_lookup_table(T_1, exptbl);
4    do frzcnt < FRZ‾ →
5         k := k + 1; chg := 0; trl := 0;
6         do chg < CHG‾ and trl < TRL‾ →
7              trl := trl + 1;
8              v := irand(1, |V|);
9              for i = 1,...,v − 1, v + 1,...,|V| → Γ̄(i) := Γ(i) rof;
10             Γ̄(v) := not Γ(v) ;
11             F_new := f(Γ̄);
12             if F_new < F_old →
13                  for i = 1,...,|V| → Γ(i) := Γ̄(i) rof;
14                  F_old := F_new; chg := chg + 1
15             fi;
16             if 0 ≤ F_new − F_old < 10T_k →
17                  if exptbl(F_new − F_old) > rrand(0, 1) →
18                       for i = 1,...,|V| → Γ(i) := Γ̄(i) rof;
19                       F_old := F_new; chg := chg + 1
20                  fi
21             fi
22        od;
23        T_{k+1} := T_k * α;
24        make_lookup_table(T_{k+1}, exptbl);
25        if chg < CHG_ → frzcnt := frzcnt + 1 fi;
27   od
end simanl;
```

Figure 12: Pseudo-Code - Simulated annealing

In Line 23, the temperature is decreased, and in Line 24, the new exponential lookup table is computed. Finally, if the number of solution changes during the last temperature cycle was less than $\underline{CHG}$, the stopping criterion counter is incremented in Line 25.

It can be shown (c.f. (Aarts & Korst, 1989)) that if $\overline{FRZ}$, $\overline{TRL}$, $\overline{CHG}$ and T_1 are large enough and $1-\alpha$ is small enough, the simulated annealing algorithm converges to the global optimum. One of the drawbacks to simulated annealing is that the various parameters must be picked with some care. Using values that lead to tractable running times often limits the method to approximate solutions. After extensive preliminary testing, the following three settings were selected based on their speed of convergence and quality of solutions. $\lambda = 1$, $\overline{FRZ} = 10$ and $T_1 = 10.0$. We ran three sets of experiments using:

- Setup A: $\overline{TRL} = 125000$, $\overline{CHG} = 10000$ and $\alpha = 0.975$,
- Setup B: $\overline{TRL} = 250000$, $\overline{CHG} = 20000$ and $\alpha = 0.975$,
- Setup C: $\overline{TRL} = 250000$, $\overline{CHG} = 20000$ and $\alpha = 0.99$.

Since simulated annealing requires far more CPU time than GRASP, we have limited this approach to 50 of the 200 $G_{1000,.5}$ test problems corresponding to random number seeds $s = 1, \ldots, 50$. All runs were conducted on a Silicon Graphics IRIS RISC based computer running IRIX System V Release 4d1-3.1F. The code was compiled on the `f77` compiler using optimization flag `-O`. All times reported are user times given by the system call `times()`. For comparison, our serial simulated annealing code runs approximately three times faster on the Silicon Graphics machine than on a single Alliant processor.

For the runs with setup A, only in four instances were independent sets of size 15 found. No set of size 16 was found. In 37 instances the best set found was of size 14, while in 9 instances the best set found was of size 13. For the runs with setup B, independent sets of size 15 were found in 12 instances. In 38 instances the best set found was of size 14. No set of size 16 was found. For the runs with setup C, independent sets of size 15 were found in 15 instances. In 35 instances the best set found was of size 14. No set of size 16 was found. Table VII and Figure 13 summarize these runs. The times reported are for the first occurrence of the best set size found.

Table VII: Simulated annealing running times (secs)

Setup	Size: Best Set Found	Instances	Time: Minimum	Time: Average	Time: Maximum
A	13	9	13389	15104	16193
	14	37	13578	16638	19960
	15	4	15456	16205	16542
B	14	38	29199	32549	36508
	15	12	29749	33294	35635
C	14	35	73910	78607	82147
	15	15	81451	86354	94538

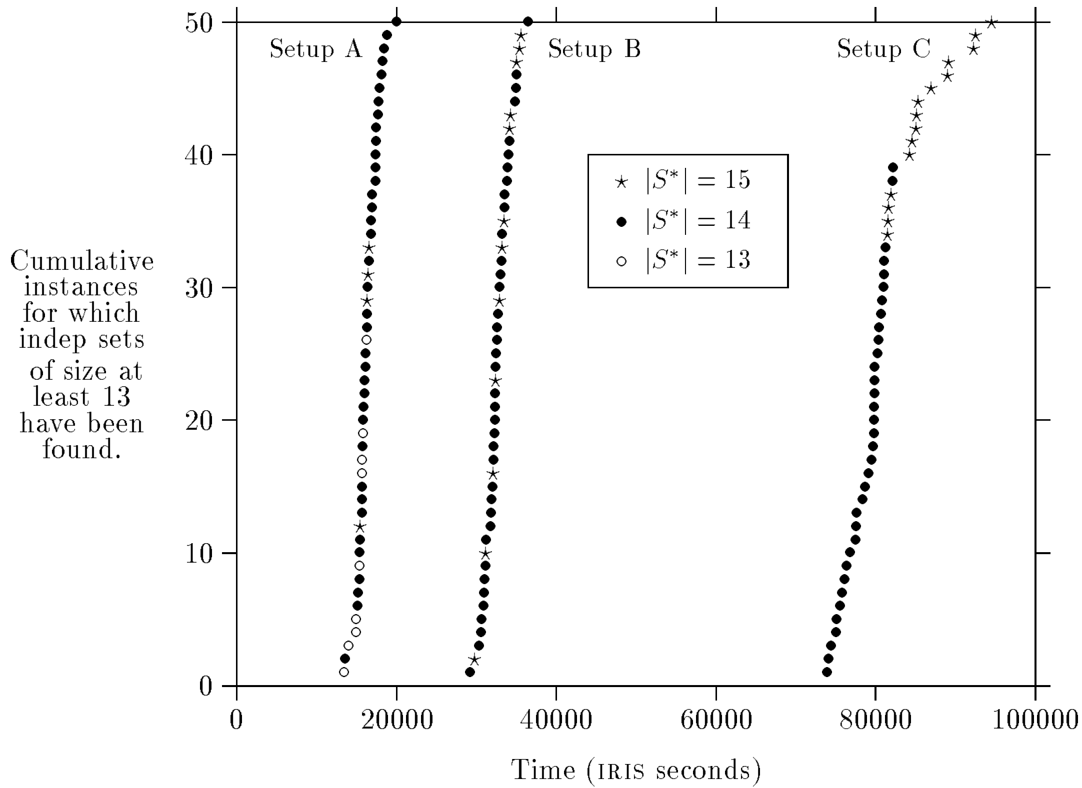

Figure 13: Simulated annealing running times

2.5. Tabu Search and Exact Methods

In this section, we compare the GRASP code with implementations of other algorithms recently described in the literature. We choose two implementations for this exercise. The first is of tabu search, an approximate algorithm, and the second of an exact partial enumeration method.

The numerical experiment described in this section was conducted on a serial (i.e. non-parallel) Silicon Graphics Indigo workstation. This computer is configured with a MIPS R2000A/R3000 processor chip revision 3.0, a MIPS R2010A/R3010 VLSI floating point chip revision 4.0, a 33MHz IP12 processor, 32 Kbytes of data cache and 32 Kbytes of instruction cache, and 32 Mbytes of main memory. The codes were compiled with Silicon Graphics `cc`, `f77` and `pc` compilers using optimization flags `-O2 -Olimit 800`. Times reported exclude problem input/generation time and are measured with the system call `times()`.

A tabu search Pascal code due to Friden et al. (1989) called STABULUS has recently been noted for achieving independent sets of good quality. Tabu search consists of several components (see, e.g., Glover (1989) and Glover (1990)). The short term memory component, that has been the focus of many studies, is applied in STABULUS.

In the first part of this experiment, we compare GRASP with STABULUS on three classes of problems: $G_{600,.5}$, $G_{1500,.5}$ and $G_{3500,.5}$. Tables I-II summarize the independent set distributions for these problem classes. We instructed the codes to search for sets of size 14 in $G_{600,.5}$, of size 16 in $G_{1500,.5}$, and of size 17 in $G_{3500,.5}$. 100 instances of each class were generated (using seeds $= 1, 2, \ldots, 100$). For the $G_{600,.5}$ and $G_{1500,.5}$ problems, the GRASP code was run with `alpha` $= 0.1$, `ntup` $= 1000$, `nfix` $= 2$, `niter` $= 100$, `iscoff` $= 11$, `nlow` $= 50$, `k` $= 2$, and `nproc` $= 1$. For the $G_{3500,.5}$ problems, the parameter settings were identical, except for `iscoff` $= 14$ and `nfix` $= 3$. STABULUS was run with default parameter settings, except for maximum number of random restarts, which was set to 30. The setting produces better results for STABULUS than those reported in Friden et al. (1989). On all runs, if a set of the size requested was not found, the requested set size was decreased by 1. The statistics displayed are for the search for the smaller independent set.

Table VIII summarizes the GRASP runs for $G_{600,.5}$, $G_{1500,.5}$ and $G_{3500,.5}$. The table list the number of instances with sets of sizes 13, 14, 15, 16 and 17 found, and the average, minimum and maximum construction time, local search time, total time and number of tuples considered. Table IX summarizes the STABULUS runs for $G_{600,.5}$, $G_{1500,.5}$ and $G_{3500,.5}$. The table lists the number of instances with sets of sizes 13, 14, 15, 16 and 17 found, and the average, minimum and maximum solution time, total searches and number of random restarts. Running times are in seconds.

The results in this experiment show that GRASP and STABULUS produce independent

Table VIII: Summary of GRASP runs on $G_{600,.5}$, $G_{1500,.5}$ and $G_{3500,.5}$.

		$G_{600,.5}$	$G_{1500,.5}$	$G_{3500,.5}$
Sets found with:	$\|\mathcal{S}^*\| = 17$			100
	$\|\mathcal{S}^*\| = 16$		99	
	$\|\mathcal{S}^*\| = 15$	1	1	
	$\|\mathcal{S}^*\| = 14$	87		
	$\|\mathcal{S}^*\| = 13$	12		
Construction time:	average	133.1	1015.9	736.7
	minimum	0.1	2.1	8.9
	maximum	1181.4	4785.1	2930.7
Local search time:	average	123.5	1213.2	270.8
	minimum	0.1	2.8	1.3
	maximum	1044.8	5880.5	1092.5
Total time:	average	256.6	2229.1	1007.5
	minimum	0.2	4.9	10.2
	maximum	2226.2	10665.6	4023.2
Tuples considered:	average	104.6	148.9	29.9
	minimum	1	1	1
	maximum	948	709	119

Table IX: Summary of STABULUS runs on $G_{600,.5}$, $G_{1500,.5}$ and $G_{3500,.5}$.

		$G_{600,.5}$	$G_{1500,.5}$	$G_{3500,.5}$
Sets found with:	$\lvert\mathcal{S}^*\rvert = 17$			100
	$\lvert\mathcal{S}^*\rvert = 16$		99	
	$\lvert\mathcal{S}^*\rvert = 15$		1	
	$\lvert\mathcal{S}^*\rvert = 14$	88		
	$\lvert\mathcal{S}^*\rvert = 13$	12		
Total solution time:	average	140.4	3011.9	1848.1
	minimum	0.1	18.9	428.0
	maximum	1279.2	12533.4	6379.2
Total searches:	average	32915.8	216993.0	26564.0
	minimum	35	1464	1116
	maximum	299979	811062	117163
Random restarts:	average	1.7	6.5	1.4
	minimum	1	1	1
	maximum	10	23	4

sets, on average, of approximately the same size. For the class of smaller graphs ($G_{600,.5}$) the GRASP code found a set of size 15, whereas the tabu search code produced sets of size at most 14. The GRASP code produced the same number of size 13 sets as did STABULUS. However, the tabu search code was approximately 1.8 times faster than the GRASP code. When compared on the class of graphs $G_{1500,.6}$ both codes produced sets of size 16 for all instances except one. In terms of running times, the GRASP code was on average approximately 1.4 times faster than STABULUS. On the class of largest graphs ($G_{3500,.5}$) both codes found sets of size 17 for all instances. The GRASP implementation was, on average, approximately 1.8 times faster than STABULUS.

The results indicate a clear trend in favor of the GRASP implementation as the size of the graph increases. However, the ratio of the average running times fails to show how much faster the GRASP code was on most of the $G_{3500,.5}$ instances tested. For example, in 34 of the 100 instances, the GRASP code found an independent set for size 17 in less than 425 seconds, while the tabu search code took over 425 seconds on all 100 instances to find such sets. Table X and Figure 14 illustrate this point. In that table and figure, the instances have been ordered in increasing order of running time. A possibly different ordering was made for each code. The first row in the table, for example, displays average running times for the 10 instances each code solved fastest. The second row displays average

Table X: GRASP and STABULUS average running times ($|V| = 600, 1500, 3500$ and $p = 0.5$). Instances in increasing order of running time for each code.

Instances	$G_{600,.5}$		$G_{1500,.5}$		$G_{3500,.5}$	
	GRASP	STABULUS	GRASP	STABULUS	GRASP	STABULUS
$1, \ldots, 10$	0.8	0.5	57.4	210.4	43.6	446.9
$1, \ldots, 20$	3.7	1.6	148.9	492.4	51.6	546.9
$1, \ldots, 30$	11.7	4.4	271.7	723.1	163.1	630.8
$1, \ldots, 40$	20.9	10.7	374.6	960.5	234.7	739.3
$1, \ldots, 50$	32.6	19.6	530.9	1240.8	314.0	844.0
$1, \ldots, 60$	49.4	28.2	730.6	1511.7	413.8	950.8
$1, \ldots, 70$	71.2	38.3	959.4	1771.1	529.9	1066.0
$1, \ldots, 80$	98.4	52.5	1206.0	2059.9	649.8	1234.5
$1, \ldots, 90$	148.1	76.1	1584.0	2428.4	788.3	1467.1
$1, \ldots, 100$	256.6	140.4	2229.1	3011.9	1007.5	1848.1

running times for the 20 instances each code solved fastest, and so on. One can observe there that for the instances that were solved quickly the increase in speedup with problem size is accentuated. While, STABULUS was on average 1.6 times faster than GRASP on the 600-node instances, it was 3.7 times slower on the 1500-node instances and over 10 times slower on the 3500-node instances. In fact, in one instance from $G_{3500,.5}$, GRASP found a set of size 17 in 11.1 seconds, while the least time taken by STABULUS on any instance was 428.0 seconds, i.e. over 38 times more CPU time. This can be seen in Figure 14, where CPU time ratios are given, in log scale, for STABULUS running times divided by GRASP running times for the 3 classes of graphs tested.

It should be emphasized that we used serial implementations of GRASP and tabu search in the above experiments. Whereas GRASP was shown to benefit greatly from a parallel implementation, it is not clear how much tabu search can benefit. The assignment of the different random restarts to different processors may not benefit STABULUS because the number of restarts in STABULUS is small compared to the number of tuples searched in the GRASP runs.

It should be noted that the tabu search and GRASP methodologies are not being compared here, but rather, we consider two particular implementations of these methodologies. Furthermore, we observe that GRASP and tabu search are not antithetical. We may incorporate a tabu search to perform the local search function for GRASP, or equivalently incorporate a GRASP component to perform a multi-start function for tabu search, as was

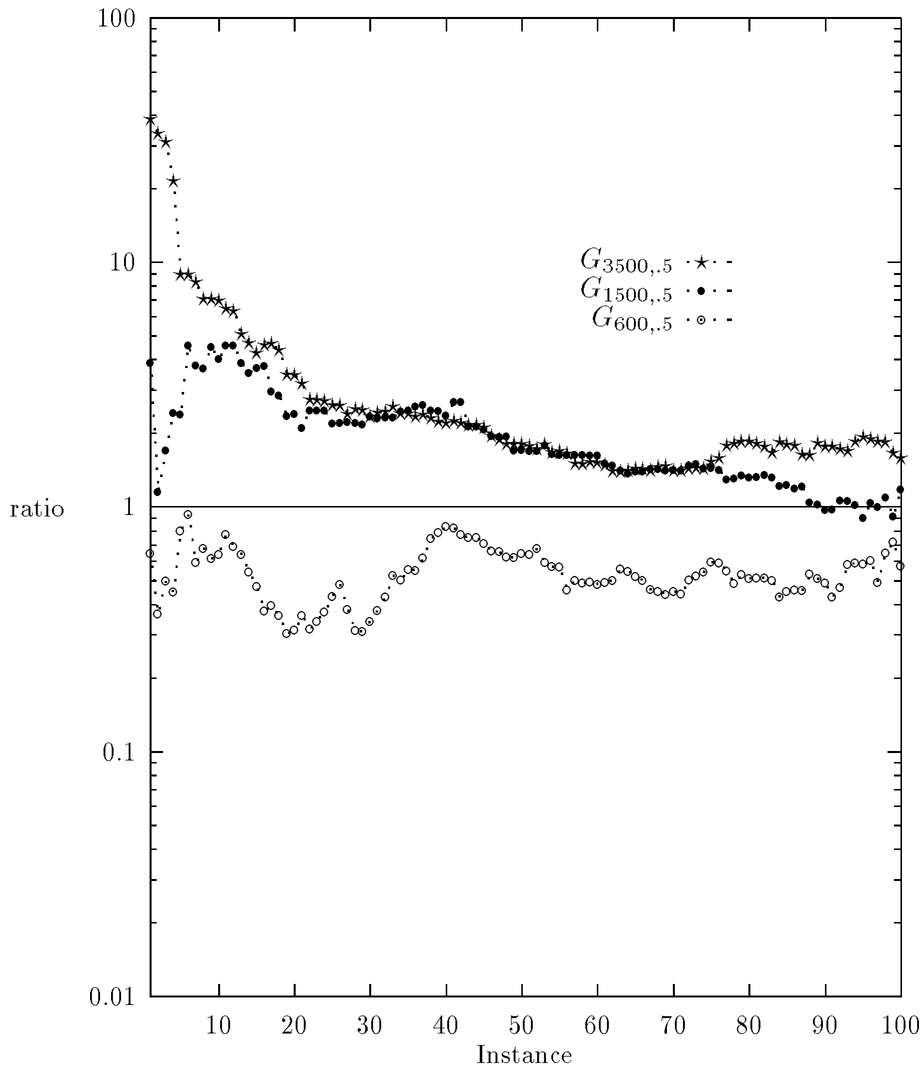

Figure 14: STABULUS to GRASP running time ratios ($|V| = 600, 1500, 3500$, $p = 0.5$). Instances in increasing order of running time for each code.

Table XI: GRASP and the Exact Method on 25 instances with $|V| = 600$ and $p = 0.5$.

Code	Size of set found			Average
	$\lvert S^*\rvert = 15$	$\lvert S^*\rvert = 14$	$\lvert S^*\rvert = 13$	CPU time
EXACT	24	1	-	12553.8
GRASP	-	23	2	310.8

done in Laguna and González-Velarde (1991). Nevertheless, our results show the power of the GRASP approach even in an implementation using simple ascent.

GRASP and tabu search are not guaranteed to produce an optimal solution, i.e. they produce approximate solutions. To study the tradeoff between CPU time requirements and obtaining a certificate of optimality, we compare GRASP to the exact method implemented by Carraghan and Pardalos (1990). This exact method uses a simple-to-implement partial enumeration scheme. Its FORTRAN source code is available in Carraghan and Pardalos (1990), where the authors claim their code to be "faster than any previous known method" for producing exact optimal solutions to maximum clique problems. In their paper, they describe solving an instance in $G_{500,.5}$ in 1114.8 seconds on an IBM 3090 with Vector Facility. We attempted to run their code on an instance from $G_{1000,.5}$, but did not produce an optimal solution after 10 CPU days of running time on the Silicon Graphics Indigo (33 MHz MIPS M3000-based workstation). We were successful in running 25 instances (seeds $= 1, 2, \ldots, 25$) from $G_{600,.5}$ with the exact method code and obtaining optimal solutions. Table XI illustrates solution times and independent set sizes obtained by the two codes on these instances. Running times are given in seconds. The GRASP code was run with the parameter settings described earlier in this section. The exact method code was run as published in Carraghan and Pardalos (1990).

This experiment shows that for the first 25 instances generated in $G_{600,.5}$, GRASP rarely produced an optimal solution. On the other hand, to produce a certificate of optimality, the exact code took about 40 times longer than the GRASP code. Because of the extended running times of the exact code, it appears that it should prove infeasible to run the exact code on graphs in $G_{|V|,.5}$ with $|V| > 700$.

3. Concluding Remarks

This paper presented a greedy randomized adaptive search procedure (GRASP) for maximum independent set. This heuristic can benefit from parallelism in a very natural manner. A parallel implementation of the heuristic was tested on hundreds of instances of large

randomly generated graphs.

For the class of random graphs $G_{1000,.5}$, the heuristic found independent sets of size 15 or 16 in all 200 instances tested. Finding sets of size 15 in $G_{1000,.5}$ has been long considered a difficult task (Bollobás & Thompson, 1985). Sets of size 16 are rare in $G_{1000,.5}$ (Bollobás, 1985) and have not previously been reported.

The heuristic was also tested on 200 denser graphs from the class $G_{1000,.83}$ and 200 more sparse graphs from $G_{1000,.2}$. Results on the dense graphs resembled those of $G_{1000,.5}$ but required less time. For the sparse graphs, running times were greater, and sets of size 37 and 38 were found in more than half of the instances.

The GRASP was tested on 50 instances of random graphs from the class $G_{2000,.5}$ that have 2000 vertices and on average 1 million edges. In all instances, sets of size 16 or 17 were found.

We implemented a version of simulated annealing for maximum independent set and tested it on 50 of the 200 instances of $G_{1000,.5}$ used to test the GRASP. By increasing the number of trials ($\overline{TRL}$), maximum number of changes ($\overline{CHG}$) per temperature cycle and α, the number of sets of size 15 found by simulated annealing increased. One could infer from our experimentation that for large enough values of $\overline{TRL}$, $\overline{CHG}$ and α simulated annealing may find sets of size 15 or better in all instances. However, the running time required would be prohibitive. For parameter settings yielding sets of size 15, simulated annealing took from 30 to over 150 times the CPU time required by the single processor implementation of GRASP.

In comparison with the tabu search code STABULUS, the GRASP code provides high quality and fast solutions, especially for the larger instances with 3500 or more vertices. Similarly, favorable comparisons can be implied with respect to the exact method offered by Carraghan and Pardalos (1990) for instances of $G_{600,.5}$, where solutions within one unit of the optimal size were found in a fraction of the time taken by the exact method. For larger instances, the exact method becomes prohibitively expensive to apply, whereas GRASP still finds high quality solutions.

Reference

Aarts, E., & Korst, J. (1989). *Simulated annealing and Boltzman machines: A stochastic approach to combinatorial optimization and neural computing.* John Wiley and Sons.

Avondo-Bodeno, G. (1962). *Economic applications of the theory of graphs.* Gordon and Breach Science Publishers.

Balas, E., & Yu, C. (1986). Finding a maximum clique in an arbitrary graph. *SIAM Journal of Computing*, *15*, 1054–1068.

Bard, J., & Feo, T. (1989). Operations sequencing in discrete parts manufacturing. *Management Science*, *35*, 249–255.

Bard, J., & Feo, T. (1991). An algorithm for the manufacturing equipment selection problem. *IIE Transactions*, *23*, 83–92.

Berge, C. (1962). *The theory of graphs and its applications.* Methuen.

Bollobás, B. (1985). *Random Graphs.* Academic Press.

Bollobás, B., & Thompson, A. (1985). Random graphs of small order. *Ann. Discrete Math.*, *28*, 47–97.

Carraghan, R., & Pardalos, P. (1990). An exact algorithm for the maximum clique problem. *Operations Research Letters*, *9*, 375–382.

Deo, N. (1974). *Graph theory with applications to engineering and computer science.* Prentice-Hall.

Feo, T., & Bard, J. (1989). Flight scheduling and maintenance base planning. *Management Science*, *35*, 1415–1432.

Feo, T., & Resende, M. (1989). A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, *8*, 67–71.

Friden, C., Hertz, A., & de Werra, D. (1989). STABULUS: A technique for finding stable sets in large graphs with tabu search. *Computing*, *42*, 35–44.

Garey, M., & Johnson, D. (1979). *Computers and intractability - A guide to the theory of NP-completeness.* W.H. Freeman and Company.

Glover, F. (1989). Tabu Search – Part I. *ORSA Journal on Computing*, *1*, 190–206.

Glover, F. (1990). Tabu Search – Part II. *ORSA Journal on Computing*, *2*, 4–32.

Johnson, D., Aragon, C., McGeoch, L., & Schevon, C. (1991). Optimization by simulated annealing: An experimental evaluation; Part II, Graph coloring and number partitioning. *Operations Research*, *39*, 378–406.

Klincewicz, J. (1989). Avoiding local optima in the p-hub location problem using tabu search and GRASP. Tech. rep., AT&T Bell Laboratories, Holmdel, NJ.

Laguna, M., & González-Velarde, J. (1991). A search heuristic for just-in-time scheduling in parallel machines. *Journal of Intelligent Manufacturing*, *2*, 253–260.

Pardalos, P., & Xue, J. (1992). The maximum clique problem. Tech. rep., Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL.

Schrage, L. (1979). A more portable Fortran random number generator. *ACM Transactions on Mathematical Software*, *5*, 132–138.

Trotter Jr., L. (1973). Solution characteristics and algorithms for the vertex packing problem. Tech. rep. 168, Dept. of Operations Research, Cornell University, Ithaca, NY.