J Comb Optim (2006) 12:303–323
DOI 10.1007/s10878-006-9635-y

Phased local search for the maximum clique problem

Wayne Pullan

Published online: 14 August 2006

Abstract This paper introduces Phased Local Search (PLS), a new stochastic reactive dynamic local search algorithm for the maximum clique problem. PLS interleaves sub-algorithms which alternate between sequences of iterative improvement, during which suitable vertices are added to the current clique, and plateau search, where vertices of the current clique are swapped with vertices not contained in the current clique. The sub-algorithms differ in their vertex selection techniques in that selection can be solely based on randomly selecting a vertex, randomly selecting within highest vertex degree or randomly selecting within vertex penalties that are dynamically adjusted during the search. In addition, the perturbation mechanism used to overcome search stagnation differs between the sub-algorithms. PLS has no problem instance dependent parameters and achieves state-of-the-art performance for the maximum clique problem over a large range of the commonly used DIMACS benchmark instances.

Keywords Maximum clique · Adaptive search · Local search · Dynamic search

1 Introduction

The maximum clique problem (MC) calls for finding the maximum sized sub-graph of pairwise adjacent vertices in a given graph. MC is a prominent combinatorial optimisation problem with many applications, for example, information retrieval, experimental design, signal transmission and computer vision (Balus and Yu, 1986). More recently, applications in bioinformatics have become important (Pevzner and Sze, 2000; Ji et al., 2004). The search variant of MC can be stated as follows: Given an undirected graph $G(V, E)$, where V is the set of all vertices and E the set of all

W. Pullan (✉)
School of Information and Communication Technology, Griffith University,
Gold Coast, QLD, Australia
e-mail: w.pullan@griffith.edu.au

edges, find a maximum size clique in G, where a clique in G is a subset of vertices, $K \subseteq V$, such that all pairs of vertices in K are connected by an edge, i.e., for all $v, v' \in K, \{v, v'\} \in E$, and the size of a clique K is the number of vertices in K. MC is NP-hard and the associated decision problem is NP-complete (Garey and Johnson, 1979); furthermore, it is inapproximable in the sense that no deterministic polynomial-time algorithm can find cliques of size $|V|^{1-\epsilon}$ for any $\epsilon > 0$, unless NP = ZPP (Håstad, 1999).[1] The best polynomial-time approximation algorithm for MC achieves an approximation ratio of $O(|V|/(\log |V|)^2)$ (Boppana and Halldórsson, 1992). Therefore, large and hard instances of MC are typically solved using heuristic approaches, in particular, greedy construction algorithms and stochastic local search algorithms such as simulated annealing, genetic algorithms and tabu search. (For an overview of these and other methods for solving MC, see (Bomze et al., 1999)). It may be noted that the maximum clique problem is equivalent to the independent set problem as well as to the minimum vertex cover problem, and any algorithm for MC can be directly applied to these equally fundamental and application relevant problems (Bomze et al., 1999).

From the recent literature on MC algorithms, it seems that, somewhat unsurprisingly, there is no single best algorithm. Although most algorithms have been empirically evaluated on benchmark instances from the Second DIMACS Challenge (Johnson and Trick, 1996), it is quite difficult to compare experimental results between studies, mostly because of differences in the respective experimental protocols and run-time environments. Nevertheless, particularly considering the comparative results reported by Pullan and Hoos (2006), it seems that there are five stochastic local search MC algorithms that achieve state-of-the-art performance: *Reactive Local Search (RLS)* (Battiti and Protasi, 2001), an advanced and general tabu search method that automatically adapts the tabu tenure parameter; *Deep Adaptive Greedy Search (DAGS)* algorithm (Grosso et al., 2005) which uses an iterated greedy construction procedure with vertex weights; *k-opt* algorithm (Katayama et al., 2004) is based on a conceptually simple Variable Depth Search (VDS) procedure that uses elementary search steps in which a vertex is added to, or removed from, the current clique; *VNS* (Hansen et al., 2004) is a basic variable neighbourhood search heuristic that combines greedy search with simplical vertex tests in its descent steps; and *Dynamic Local Search—Maximim Clique* (DLS-MC) (Pullan and Hoos, 2006), an algorithm which alternates between phases of iterative improvement, during which suitable vertices are added to the current clique, and plateau search, where vertices of the current clique are swapped with vertices not contained in the current clique. The selection of vertices is solely based on vertex penalties that are dynamically adjusted during the search, and a perturbation mechanism is used to overcome search stagnation. The behaviour of DLS-MC is controlled by a single parameter, penalty delay, which controls the frequency at which vertex penalties are reduced. Unfortunately, the DLS-MC algorithm is sensitive to the penalty delay parameter, and its calibration time-consuming and instance-dependent. This weakness of DLS-MC has been explicitly stated (Pullan and Hoos, 2006), where elimination of tuning for penalty delay is indicated as a valuable improvement.

[1] ZPP is the class of problems that can be solved in expected polynomial time by a probabilistic algorithm with zero error probability.

In this work, a new stochastic local search algorithm for MC, based on DLS-MC and dubbed Phased Local Search (PLS), is introduced. PLS is a reactive algorithm, that requires no run-time parameters, and interleaves sub-algorithms which differ primarily in their vertex selection methods and the perturbation mechanisms used to overcome search stagnation. Based on extensive computational experiments, it is shown that PLS has comparable performance to DLS-MC, on a range of widely studied benchmark instances.

The remainder of this paper is structured as follows. The PLS algorithm and key aspects of its efficient implementation are first described. Next, empirical performance results are presented that establish PLS as state-of-the-art in heuristic MC solving. This is followed by a more detailed investigation of the behaviour of PLS and the factors determining its performance. Finally, a summary of the main contributions of this work, insights gained from this study, and an outline of some directions for future research are described.

2 The PLS algorithm

> ...for any algorithm, any elevated performance over one class of problems is exactly paid for in performance over another class. Wolpert and Macready (1997)

This observation is substantiated by the results presented in Pullan and Hoos (2006) and Grosso et al. (2005) where it became apparent that, to efficiently solve all the DIMACS benchmark instances, algorithms with conflicting characteristics were required. In particular: a random sub-algorithm is able to solve instances where the maximum cliques are a combination of high, average and low degree vertices (for example, the DIMACS p_hat1500-1 instance); a greedy sub-algorithm that favours the higher degree vertices is efficient where the vertices in the maximum cliques are biased towards the higher degree vertices (for example, randomly generated instances such as the DIMACS Cn family); and, a greedy sub-algorithm that favours the lower degree vertices is efficient for maximum cliques whose vertices are biased towards the lower degree vertices (for example, the DIMACS brock family of instances). PLS interleaves three such sub-algorithms which use random selection (*Random* sub-algorithm), random selection within vertex degree (*Degree* sub-algorithm), and random selection within vertex penalties (*Penalty* sub-algorithm), and is now described using the following notation: $G(V, E)$—an undirected graph with $V = \{1, 2, \ldots, n\}$, $E \subseteq \{\{i, j\} : i, j \in V\}$; $N(i) = \{j \in V : \{i, j\} \in E\}$—the vertices adjacent to i; K—current clique of G; and $C_p(K) = \{i \in V : |K \setminus N(i)| = p\}$, $p = 0, 1, \ldots$—the set of all vertices not adjacent to exactly p vertices in K. That is, C_0 is the increasing set of vertices while C_1 is the level set of vertices.

Algorithm PLS(G, ***tcs***, ***max-selections***)
Input: graph G; integers *tcs* (target clique size); *max-selections*;
Output: Clique of cardinality *tcs* or 'failed';
1. *selections* := 0;
2. $U := \emptyset$;
3. <Randomly select a vertex $v \in V$, $K := \{v\}$ >;
4. **do**
5. *Phase*(50, *RandomSelect*, *Reinitialise*);

```
 6.      if |K| = tcs then return K;
 7.      Phase(50, PenaltySelect, Reinitialise);
 8.      if |K| = tcs then return K;
 9.      Phase(100, DegreeSelect, Initialise);
10.     if |K| = tcs then return K;
11.   while selections < max-selections;
12.   return 'failed';
```

The PLS algorithm operates as follows: an initial vertex is selected from the given graph G uniformly at random and the current clique K is initialised to the set consisting of this single vertex. Then, the search, (lines 4–11 of PLS) repeatedly performs 50 iterations of the *Random* sub-algorithm (by initiating the function **Phase** with appropriate parameters), followed by 50 iterations of the *Penalty* sub-algorithm and then 100 iterations of the *Degree* sub-algorithm. This sequence terminates when either a clique of cardinalty *tcs* is found or the maximum number of selections (additions to the current clique K) has occurred. PLS invokes the function **Phase** to implement the *Random*, *Penalty* and *Degree* sub-algorithms described above. The structure of function **Phase** is:

function Phase (*iterations*, *Select*, *Perturb*)

```
      Input:  iterations; function Select; function Perturb;
      Output: K (current solution);
 1.   do
 2.       do
 3.           while C0(K) \ U ≠ ∅ do
 4.               v :=Select(C0(K));
 5.               K := K ∪ {v};
 6.               selections := selections + 1;
 7.               if |K| = tcs then return K;
 8.               U := ∅;
 9.           end while
10.           if C1(K) \ U ≠ ∅ then
11.               v := Select(C1(K) \ U);
12.               K := [K ∪ {v}] \ {i}, U := U ∪ {i}, where {i} = K \ N(v);
13.               selections := selections + 1;
14.           end if;
15.       while C0(K) ≠ ∅ or C1(K) \ U ≠ ∅;
16.       iterations := iterations − 1;
17.       UpdatePenalties;
18.       Perturb;
19.   while (iterations > 0 and selections < max-selections);
20.   return K;
```

Each iteration of **Phase** (within which a single complete execution of lines 1–19 is referred to as an "iteration") alternates between an iterative improvement phase, during which vertices from the increasing set $C_0(K)$ are added to the current clique K, and a plateau search phase, in which vertices from the level set $C_1(K)$ are swapped with the vertex in K with which they do not share an edge. The search phase terminates when $C_0(K) = \emptyset$ and either $C_1(K) = \emptyset$ or all vertices that are in $C_1(K)$ have already

been an element of K during the current iteration. As the final step of the iteration, a perturbation of K is performed to generate a new starting point for the next iteration. Iterations are repeated until either the maximum clique is found or the number of allowed *iterations* have been performed or the number of allowed *selections* (additions to the current clique) is exceeded. The different vertex selection methods for each sub-algorithm are implemented within the input function *Select* while the different perturbations for each sub-algorithm are implemented within the input function *Perturb*. Note that the differences between the sub-algorithms are wholly contained within these input functions. Finally, penalty updates are performed (*UpdatePenalties*) during all sub-algorithms but penalties are only used for vertex selection when the *Penalty* sub-algorithm is active. The three sub-algorithms of PLS are now described in more detail:

- ***Random* Sub-algorithm:** For this sub-algorithm, within *RandomSelect*, vertices are chosen uniform randomly from $C_0(K)$ and $C_1(K)$ to be added or swapped into K. At the completion of the iteration, function *Reinitialise* is invoked to uniform randomly select a vertex v, add this to K and remove all vertices from K that are not connected to v. This perturbation mechanism provides for some continuity in the search and also maintains K as relatively large at all times.
- ***Degree* Sub-algorithm:** For this sub-algorithm, *DegreeSelect* selects a vertex from $C_0(K)$ (or $C_1(K)$) with maximum degree—ties broken randomly. Note that the computation of vertex degree is made once only as part of PLS initialisation. This vertex selection rule turns out to be particularly efficient when dealing with random graphs such as the DIMACS Cn family of instances. The perturbation mechanism for this sub-algorithm is the *Reinitialise* method described above for the *Random* sub-algorithm.
- ***Penalty* Sub-algorithm:** The purpose of vertex penalties is to provide additional diversification to the search process, which otherwise could easily stagnate in situations where the current clique has few or no vertices in common with an optimal solution for a given MC instance. Perhaps the most obvious approach for avoiding this kind of search stagnation is to simply restart the constructive search process from a different initial vertex. However, even if there is random (or systematic) variation in the choice of this initial vertex, there is still a risk that the heuristic guidance built into the greedy construction mechanism causes a bias towards a limited set of suboptimal cliques. Therefore, integer penalties are associated with the vertices that modulate the heuristic selection function used in the greedy construction procedure in such a way that vertices that repeatedly occur in the cliques obtained from the constructive search process are discouraged from being used in future constructions.

  For this sub-algorithm, within *PenaltySelect*, vertices are chosen uniform randomly from the lowest penalty sub-set of $C_0(K)$ and $C_1(K)$ to be added or swapped into K. At the completion of the iteration, function *Initialise* is invoked to uniform randomly select a vertex v and initialise K to contain only this vertex. Typically this perturbation mechanism provides for relatively large discontinuities in the search trajectory.

  The penalty phase of the DLS-MC algorithm described in Pullan and Hoos (2006) required a exogenous, family or sub-family instance-dependent, parameter penalty delay which specified how frequently penalties should be decreased. However, as

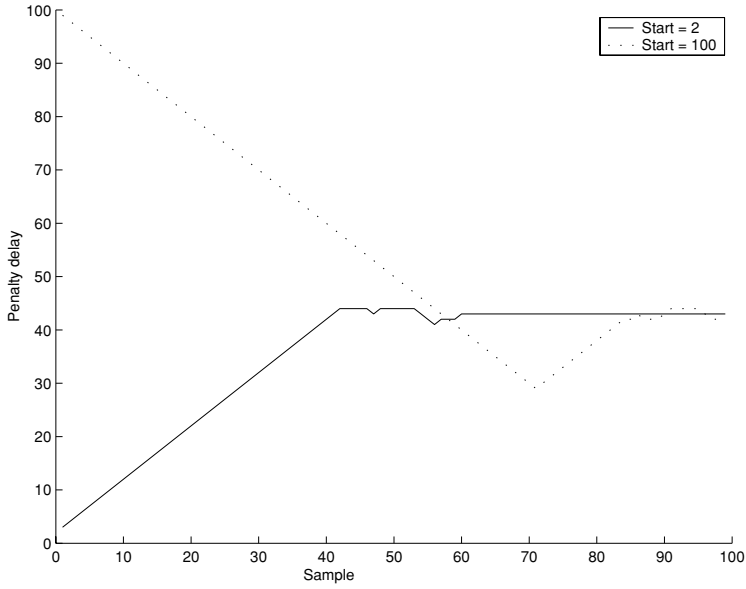

Fig. 1 Adaptive updating of the PLS penalty delay variable by the PLS function *UpdatePenalties*, from two different initial starting points, for the brock800_1 instance. Starting from PLS default value of two, the optimal value of 43 is obtained within 41 samples and is then tracked closely

was pointed out in Pullan and Hoos (2006), there seemed to be a correlation between the optimal value for penalty delay and the percentage of vertices that had a penalty at any point in time. Based on this observation, PLS uses the same penalty scheme as DLS-MC but, after initially setting penalty delay to two, reactively adjusts the penalty delay parameter at each penalty decrease cycle, with the goal of ensuring that 75% of vertices have a penalty value greater than zero. When the number of penalised vertices is less than this target, penalty delay is incremented by unity (which decreases the frequency of penalty decreases). Alternatively, if the number of penalised vertices is greater than 75% of the total vertices, penalty delay is decreased (which increases the frequency of penalty decreases). This update scheme is able to drive penalty delay towards the correct value for the instance. Figure 1 illustrates the behaviour of penalty delay for the `brock800_1` instance where DLS-MC required a high penalty delay value (45) for reaching good performances. Two experiments are shown, the first with an initial penalty delay of two (continuous line) and the second with an initial value of 100 for penalty delay (dotted line). The penalty delay value is sampled each 500 iterations; the "stable" value reached for penalty delay of 43 is very close to the optimal value of 45 identified in the tests of Pullan and Hoos (2006).

Transitioning between sub-algorithms was implemented so that the *Random* and *Degree* sub-algorithms always resumed from the point at which their previous invocation completed. However, the *Penalty* sub-algorithm continues from the point at which the preceeding *Random* sub-algorithm invocation terminated.

3 Empirical performance results

In order to evaluate the performance and behaviour of PLS, extensive computational experiments on all MC instances from the Second DIMACS Implementation Challenge (1992–1993)[2] were performed. These instances have also been used extensively for benchmarking purposes in the recent literature on MC algorithms. The 80 DIMACS MC instances were generated from problems in coding theory, fault diagnosis problems, Keller's conjecture on tilings using hypercubes, and the Steiner triple problem, in addition to randomly generated graphs and graphs where the maximum clique has been "hidden" by incorporating low-degree vertices. These problem instances range in size from less than 50 vertices and 1000 edges to greater than 3300 vertices and 5 000 000 edges.

All experiments for this study were performed on a dedicated 2.4 GHz Pentium IV machine with 512KB L2 cache and 512MB RAM using the g++ C++ compiler with the '−O2' option. To execute the DIMACS Machine Benchmark[3] this machine required 0.62 CPU seconds for r300.5, 3.80 CPU seconds for r400.5 and 14.50 CPU seconds for r500.5. In the following, unless explicitly stated otherwise, all CPU times refer to the reference machine.

3.1 PLS performance

To evaluate the performance of PLS on the DIMACS benchmark instances, 100 independent trials were performed for each instance using target clique sizes (tcs) corresponding to the respective provably optimal clique sizes or, in cases where such provably optimal solutions are unknown, largest known clique sizes. The only parameter of PLS, $maxSelections$, was set to 100 000 000 (1 000 000 000 for MANN_a45 and MANN_a81), in order to maximise the probability of reaching the target clique size in every trial.

The PLS performance results (averaged over 100 independent trials) are shown in Table 1 for the complete set of 80 DIMACS benchmark instances. Note that PLS finds optimal (or best known) solutions with a success rate of 100% over all 100 trials per instance for 76 of the 80 DIMACS instances; the instances where the target clique size was not reached consistently within the allotted $maxSelections$ were C2000.9, where all 100 trials achieved 78 as compared to 80 (Grosso et al., 2004); MANN_a45 where all 100 trials achieved 344 as compared to 345 (Hansen et al., 2004; Battiti and Protasi, 2001); MANN_a81 where all 100 trials achieved 1098 as compared to 1099 (Katayama et al., 2004), and keller6 where 36 of 100 trials were successful giving a maximum clique size (average clique size, minimum clique size) of 59(57.75, 57). For these cases, the reported CPU time statistics are over successful trials only and are shown in parentheses in Table 1. Furthermore, the expected time required by PLS to reach the target clique size is less than 1 CPU second for 67 of the 80 instances, and an expected run-time of more than 10 CPU seconds is only required for 8 of the 13 remaining instances, all of which have at least 800 vertices.

[2] http://dimacs.rutgers.edu/Challenges/.

[3] *dmclique*, ftp://dimacs.rutgers.edu in directory /pub/dsj/clique.

Table 1 PLS performance results, averaged over 100 independent trials, for the complete set of DIMACS benchmark instances. The maximum known clique size, for each instance, is shown in the ω column; CPU(s) is the run-time in CPU seconds, averaged over all successful trials, for each instance. Average CPU times less than 0.0001 seconds are shown as $<\epsilon$; 'Sels.' is the number of vertices added to the clique, averaged over all successful trials, for each instance. All 100 trials for each instance achieved the best known cliques size shown with the exceptions of: C2000.9, where all 100 trials achieved 78; MANN_a45 where all 100 trials achieved 344; MANN_a81 where all 100 trials achieved 1098 and keller6, where 36 of 100 trials were successful giving a maximum clique size (average clique size, minimum clique size) of 59(57.75, 57)

Instance	ω	CPU(s)	Sels.	Instance	ω	CPU(s)	Sels.	Instance	ω	CPU(s)	Sels.
brock200_1	21	0.0036	2958	DSJC500_5	13	0.0078	1683	p_hat1500-2	65	0.0116	1702
brock200_2	12	0.0294	14143	gen200_p0.9_44	44	0.0013	1761	p_hat1500-3	94	0.0319	7058
brock200_3	15	0.0272	16235	gen200_p0.9_55	55	0.0003	327	p_hat300-1	8	0.0006	134
brock200_4	17	0.0776	52799	gen400_p0.9_55	55	0.2491	220501	p_hat300-2	25	0.0003	100
brock400_1	27	1.0757	481300	gen400_p0.9_65	65	0.0016	1640	p_hat300-3	36	0.0011	954
brock400_2	29	0.3771	173598	gen400_p0.9_75	75	0.0005	476	p_hat500-1	9	0.0009	133
brock400_3	31	0.1798	81699	hamming10-2	512	0.0029	3352	p_hat500-2	36	0.0007	273
brock400_4	33	0.1038	47535	hamming10-4	40	0.0042	1037	p_hat500-3	50	0.0049	2660
brock800_1	23	30.0919	5850513	hamming6-2	32	$<\epsilon$	38	p_hat700-1	11	0.0132	1647
brock800_2	24	24.4061	4736217	hamming6-4	4	$<\epsilon$	3	p_hat700-2	44	0.0008	251
brock800_3	25	15.0795	2937434	hamming8-2	128	$<\epsilon$	89	p_hat700-3	62	0.0035	1390
brock800_4	26	6.5407	1263170	hamming8-4	16	0.0001	28	san1000	15	4.7187	368133
C1000.9	68	1.8839	701067	johnson16-2-4	8	$<\epsilon$	7	san200_0.7_1	30	0.0027	1555
C125.9	34	0.0001	199	johnson32-2-4	16	0.0001	15	san200_0.7_2	18	0.0216	12362
C2000.5	16	0.7271	44977	johnson8-2-4	4	$<\epsilon$	3	san200_0.9_1	70	0.0006	760
C2000.9	80	(112.8189)	(23563287)	johnson8-4-4	14	$<\epsilon$	17	san200_0.9_2	60	0.0004	502
C250.9	44	0.0022	2689	keller4	11	$<\epsilon$	36	san200_0.9_3	44	0.0012	1640
C4000.5	18	149.6532	4651204	keller5	27	0.0506	12485	san400_0.5_1	13	0.055	10019
C500.9	57	0.1857	128858	keller6	59	(550.9461)	(50206021)	san400_0.7_1	40	0.0563	19287
c-fat200-1	12	0.0001	24	MANN_a27	126	0.0311	54389	san400_0.7_2	30	0.0983	31739
c-fat200-2	24	0.0009	306	MANN_a45	345	(28.7605)	(44230353)	san400_0.7_3	22	0.1874	59775
c-fat200-5	58	0.0002	108	MANN_a81	1099	(269.6636)	(227500971)	san400_0.9_1	100	0.0026	2226
c-fat500-10	126	0.0014	306	MANN_a9	16	$<\epsilon$	2	sanr200_0.7	18	0.0009	651
c-fat500-1	14	0.0003	45	p_hat1000-1	10	0.0044	277	sanr200_0.9	42	0.0083	10205
c-fat500-2	26	0.0002	46	p_hat1000-2	46	0.0027	545	sanr400_0.5	13	0.009	2404
c-fat500-5	64	0.0018	306	p_hat1000-3	68	0.0224	6657	sanr400_0.7	21	0.0083	3368
DSJC1000_5	15	0.4674	54680	p_hat1500-1	12	3.2765	197988				

As with DLS-MC, the time-complexity of search steps in PLS is generally very low. For example, for instance `brock800_1` with 800 vertices, 207 505 edges, and a maximum clique size of 23 vertices, PLS performs, on average, 194 421 selections (i.e., additions to the current clique K) per CPU second.

3.2 Comparative results

The results reported in the previous section demonstrate that PLS achieves excellent performance on the standard DIMACS benchmark instances. In Pullan and Hoos (2006), extensive comparisons are presented between DLS-MC and DAGS (Grosso et al., 2005), GRASP (Resendeand et al., 1998) (using the results contained in Grosso et al. (2005)), k-opt (Katayama et al., 2004), RLS (Battiti and Protasi, 2001), GENE (Marchiori, 2002), ITER (Marchiori, 2002) and QUALEX-MS (Busygin, 2002). It is important to note that the performance results for the algorithms that DLS-MC was compared with have been reported in a variety of ways (e.g. statistics on the clique size obtained after a fixed run-time) with generally only a very basic indication of the computer processor performance. As was noted in Pullan and Hoos (2006), this makes the comparisons between algorithms an approximation at best and a goal of the Pullan and Hoos (2006) study was to document DLS-MC using techniques such that future algorithms could be compared with greater accuracy. Building on this, the approach taken in this study is to compare PLS directly with DLS-MC by measuring performance using the same techniques as utilised in Pullan and Hoos (2006). Indirectly this then allows the performance of PLS to be compared with the performance of all the other algorithms documented in Pullan and Hoos (2006).

Table 2 contrasts the performance results for PLS with the respective performance results for DLS-MC. Overall, with the exception of keller6, PLS can be classed as comparable or more efficient than DLS-MC for all DIMACS instances.

The significant results in Table 2 from this comparative performance evaluation can be summarised as follows:

- For the DIMACS brock family, with the exception of brock800_2, PLS has, in some cases, considerably improved performance. This is an interesting result as the *Penalty* sub-algorithm is the only sub-algorithm to locate the maximum clique, while the other two PLS sub-algorithms would appear to be redundant (and time consuming). In contrast, DLS-MC only uses a penalty algorithm for the brock family of instances so the expectation would be that DLS-MC would require less computational effort to locate the maximal clique. This apparent anomaly is investigated further in Section 4.
- For the DIMACS Cn family of instances, PLS clearly out-performs DLS-MC. This is primarily due to the incorporation of the *Degree* sub-algorithm within PLS which, even given the computational requirements of the completely redundant *Penalty* and partially redundant *Random* sub-algorithms, still allows PLS to out-perform DLS-MC.
- For the MANN_a45 and MANN_a81 instances, a performance enhancement, related to searching for the clique "partner" for a vertex to be swapped into the current clique for dense graphs, enabled PLS to out-perform DLS-MC in that it required, on average, less CPU time to obtain better quality results. However, PLS did require considerably more selections than DLS-MC to obtain this improvement.

Table 2 The PLS and DLS-MC success rate and average CPU times (based on 100 trials per instance). The SCPU(s) columns for DLS-MC are the DLS-MC CPU(s) (Pullan and Hoos, 2006) scaled by 0.83 to account for the different computers used. To obtain a meaningful comparison for PLS and DLS-MC, for MANN_a45 and MANN_a81, 344 and 1098 respectively were used as best known results in producing this table. For both PLS and DLS-MC, the average CPU time is over successful trials only

	PLS		DLS-MC			PLS		DLS-MC	
Instance	Success	CPU(s)	Success	SCPU(s)	Instance	Success	CPU(s)	Success	SCPU(s)
brock200_1	100	0.0036	100	0.0151	johnson32-2-4	100	0.0001	100	$<\epsilon$
brock200_2	100	0.0294	100	0.0201	johnson8-2-4	100	$<\epsilon$	100	$<\epsilon$
brock200_3	100	0.0272	100	0.0305	johnson8-4-4	100	$<\epsilon$	100	$<\epsilon$
brock200_4	100	0.0776	100	0.0388	keller4	100	$<\epsilon$	100	$<\epsilon$
brock400_1	100	1.0757	100	1.8508	keller5	100	0.0506	100	0.0167
brock400_2	100	0.3771	100	0.3962	keller6	36	550.9461	100	141.5008
brock400_3	100	0.1798	100	0.1459	MANN_a27	100	0.0311	100	0.0395
brock400_4	100	0.1038	100	0.0559	MANN_a45	100	28.7605	100	43.1270
brock800_1	100	30.0919	100	46.8926	MANN_a81	100	269.6636	96	219.1278
brock800_2	100	24.4061	100	13.0588	MANN_a9	100	$<\epsilon$	100	$<\epsilon$
brock800_3	100	15.0795	100	18.1934	p_hat1000-1	100	0.0044	100	0.0028
brock800_4	100	6.5407	100	7.3710	p_hat1000-2	100	0.0027	100	0.0020
C1000.9	100	1.8839	100	3.6852	p_hat1000-3	100	0.0224	100	0.0051
C125.9	100	0.0001	100	0.0001	p_hat1500-1	100	3.2765	100	2.2463
C2000.5	100	0.7271	100	0.8049	p_hat1500-2	100	0.0116	100	0.0051
C2000.9	100	112.8189	93	160.3759	p_hat1500-3	100	0.0319	100	0.0085
C250.9	100	0.0022	100	0.0007	p_hat300-1	100	0.0006	100	0.0006
C4000.5	100	149.6532	100	150.4241	p_hat300-2	100	0.0003	100	0.0002
C500.9	100	0.1857	100	0.1056	p_hat300-3	100	0.0011	100	0.0006
c-fat200-1	100	0.0001	100	0.0002	p_hat500-1	100	0.0009	100	0.0008
c-fat200-2	100	0.0009	100	0.0008	p_hat500-2	100	0.0007	100	0.0004
c-fat200-5	100	0.0002	100	0.0002	p_hat500-3	100	0.0049	100	0.0019
c-fat500-10	100	0.0014	100	0.0003	p_hat700-1	100	0.0132	100	0.0161

(*Continued on next page*)

Table 2 (*Continued*)

	PLS		DLS-MC			PLS		DLS-MC	
Instance	Success	CPU(s)	Success	SCPU(s)	Instance	Success	CPU(s)	Success	SCPU(s)
c-fat500-1	100	0.0003	100	0.0012	p_hat700-2	100	0.0008	100	0.0008
c-fat500-2	100	0.0002	100	0.0003	p_hat700-3	100	0.0035	100	0.0012
c-fat500-5	100	0.0018	100	0.0017	san1000	100	4.7187	100	6.9418
DSJC1000_5	100	0.4674	100	0.6632	san200_0.7_1	100	0.0027	100	0.0024
DSJC500_5	100	0.0078	100	0.0115	san200_0.7_2	100	0.0216	100	0.0568
gen200_p0.9_44	100	0.0013	100	0.0008	san200_0.9_1	100	0.0006	100	0.0002
gen200_p0.9_55	100	0.0003	100	0.0002	san200_0.9_2	100	0.0004	100	0.0002
gen400_p0.9_55	100	0.2491	100	0.0222	san200_0.9_3	100	0.0012	100	0.0012
gen400_p0.9_65	100	0.0016	100	0.0008	san400_0.5_1	100	0.055	100	0.1362
gen400_p0.9_75	100	0.0005	100	0.0004	san400_0.7_1	100	0.0563	100	0.0903
hamming10-2	100	0.0029	100	0.0007	san400_0.7_2	100	0.0983	100	0.1752
hamming10-4	100	0.0042	100	0.0074	san400_0.7_3	100	0.1874	100	0.3527
hamming6-2	100	$<\epsilon$	100	$<\epsilon$	san400_0.9_1	100	0.0026	100	0.0024
hamming6-4	100	$<\epsilon$	100	$<\epsilon$	sanr200_0.7	100	0.0009	100	0.0017
hamming8-2	100	$<\epsilon$	100	0.0002	sanr200_0.9	100	0.0083	100	0.0105
hamming8-4	100	0.0001	100	$<\epsilon$	sanr400_0.5	100	0.009	100	0.0326
johnson16-2-4	100	$<\epsilon$	100	$<\epsilon$	sanr400_0.7	100	0.0083	100	0.0191

Table 3 Percentage allocation of total number of selections to the PLS sub-algorithms and percentages of ocurrances that each sub-algorithm was active when the maximal clique was located for selected DIMACS instances

	Random		*Penalty*		*Degree*	
Instance	% selections	% success	% selections	% success	% selections	% success
C1000.9	12.0	20	67.2	0	20.8	80
p_hat1500-1	23.2	42	31.2	26	45.6	32
brock800_1	21.2	0	38.9	100	39.9	0
keller6	16.7	100	46.9	0	36.4	0
san1000	19.9	41	39.5	22	40.6	37

- For keller6, PLS had poor performance relative to DLS-MC. As this instance has vertices that form 8 groups with respect to vertex degree and the maximal clique consists of vertices from all these groups, the *Random* sub-algorithm should be the effective algorithm. However Table 3 shows that only 16.7% of selections occurred when the *Random* sub-algorithm was active.

Given that PLS performs three sub-algorithms cyclically, of which generally only one is efficient for a particular instance, whereas the DLS-MC results were obtained by "tuning" the algorithm to each instance, these results are somewhat surprising. In particular, while the MANN and Cn results arise from the efficiency improvements and the use of the *Degree* sub-algorithm (for which there is no counterpart in DLS-MC), there is no direct implementation change to account for the improvement in some of the brock results.

4 Discussion

To gain a deeper understanding of the run-time behaviour of PLS and the efficacy of its underlying mechanisms, additional empirical analyses were performed. Specifically, the relationship between the PLS sub-algorithm selection/perturbation methods and the characteristics of the problem instances, the ordering of the sub-algorithms, and the effect of allocating relatively more iterations to each sub-algorithm were investigated.

4.1 MC instance characteristics

To justify the selection and perturbation methods used in the PLS sub-algorithms, a conceptual model of the "hyper-surface" associated with the MC instance is useful. If the characteristic vector $\mathbf{s} \in \{0, 1\}^n$ is defined by $s_i = 1 \Leftrightarrow v_i \in K$ then the hyper-surface associated with an MC instance is that defined by the function $f(\mathbf{s}) = \sum_{i=1}^{n} s_i$ $(= |K|)$. When a PLS iteration adds vertex v to K to produce K' then $f(\mathbf{s}')$ will be in the range $1 \ldots f(\mathbf{s}) + 1$. When $f(\mathbf{s}') = f(\mathbf{s})$ the addition of vertex v can be visualised as a movement along a "plateau" on the hyper-surface, when $f(\mathbf{s}') = f(\mathbf{s}) + 1$ the addition of vertex v is a movement "up-hill" on the hyper-surface, and when $f(\mathbf{s}') < f(\mathbf{s})$ the addition of vertex v is a movement "down-hill" on the hyper-surface. The hyper-surface clearly has a lower bound of $f(\mathbf{s}) = 0$ $(K = \phi)$ and an upper bound of

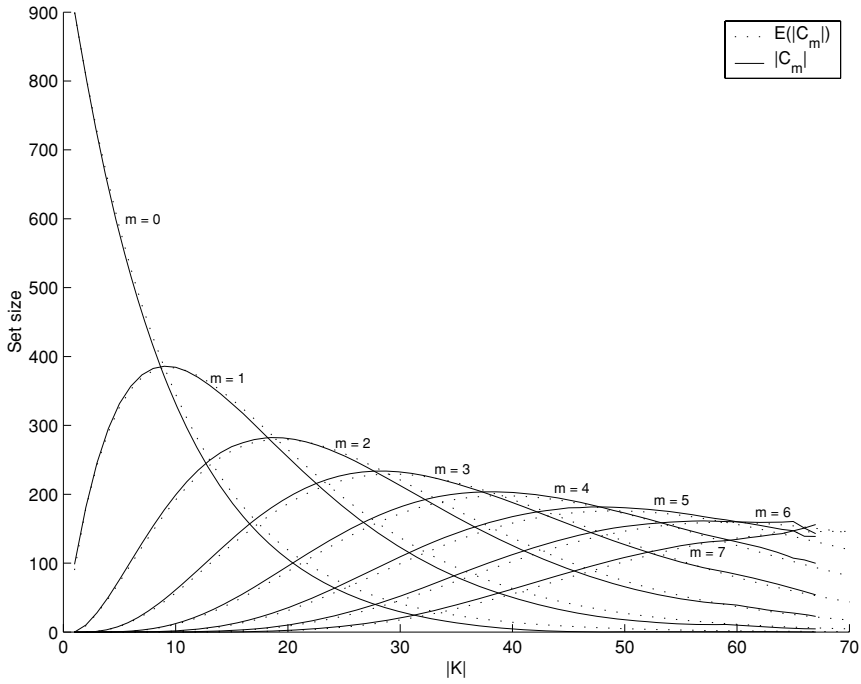

Fig. 2 Expected cardinalities of $C_m(K), m = 0 \ldots 7$ from Eq. (1) for a randomly generated graph with number of vertices $n = 1000$ and edge probability $P = 0.9$ as compared to experimental observations for C1000.9

$f(\mathbf{s}) = n$ $(K = V)$ and, within these bounds, plateaus can exist at any level. Starting from the hyper-surface lower bound ($f(\mathbf{s}) = 0$), there are two possibilites for reaching the point(s) on the hyper-surface corresponding to the maximal clique(s). Either they can be reached directly using up-hill moves only, or the point is surrounded by a plateau and one or more plateau moves are required to access it. In the PLS context, movement on a plateau corresponds to selecting a vertex from $C_1(K)$ and an up-hill move corresponds to selecting a vertex from $C_0(K)$. As PLS does not perform any selections from $C_m(K)$, $1 < m < n$ there is no movement on the down-hill portions of the hyper-surface. Accordingly, for any clique K, the sequence $|C_m(K)|$, $0 \leq m \leq 1$ defines the topography of the hyper-surface, as perceived by PLS, at the point on the hyper-surface corresponding to the clique K. For a randomly generated graph with n vertices and an edge probability of P, the expected values for the terms of this sequence are given by:

$$E(|C_m(K)|) = (n - |K|)\binom{|K|}{m} P^{|K|-m}(1 - P)^m \tag{1}$$

Figure 2 compares the expected cardinality of $C_m(K)$ from Eq. (1),with $n = 1000$ and $P = 0.9$, with that found experimentally for instance C1000.9, for $m = 0 \ldots 7$. As can be seen, there is close agreement between the predicted and measured results.

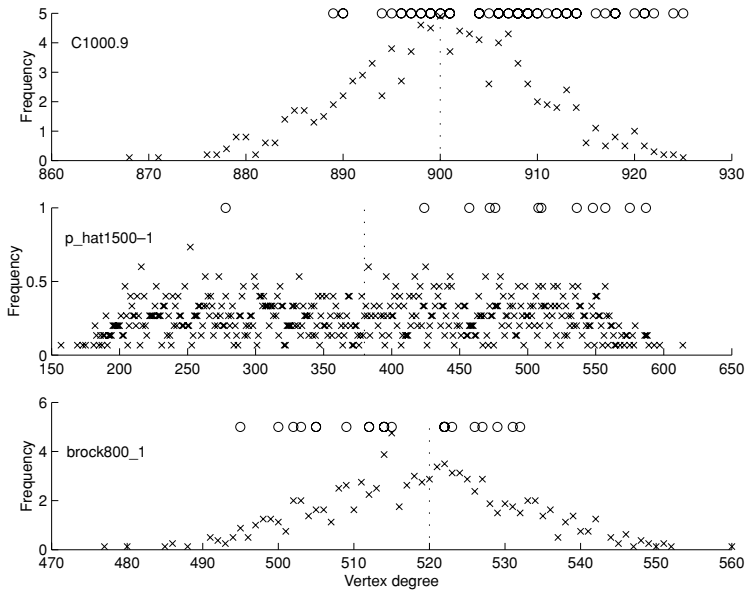

Fig. 3 Distribution of vertex degree (x) and average vertex degree (dotted line) for the C1000.9, p_hat1500-1 and brock800_1 instances. Also shown are the vertex degree (*) of the vertices in the maximum clique. As can be seen, for C1000.9, the maximal clique vertices are biased towards the higher degree vertices which makes this instance susceptible to the *Degree* sub-algorithm. For p_hat1500-1, for which only a single maximal clique appears to exist (Pullan and Hoos, 2006), with a single exception, there is a bias towards the higher degree vertices. This combination of a low degree vertex with high degree vertex plus the single maximal clique increases the difficulty of p_hat1500-1 as compared to the other instances of the p_hat family. This wide distribution of maximal clique vertex degrees makes p_hat1500-1 susceptible to the *Random* sub-algorithm. For brock800_1, the maximal clique vertices are biased towards the lower degree vertices which makes these instances susceptible to the *Penalty* sub-algorithm

4.2 Test instances

The DIMACS instances chosen for the empirical analysis of PLS were C1000.9, p_hat1500-1 and brock800_1 (collectively referred to as the "DIMACS subset" in this study). These instances were selected because, firstly, they are of reasonable size and difficulty and secondly, they are all solved effectively by a different PLS sub-algorithm. C1000.9 is a randomly generated instance where, as shown in Fig. 3, the maximal clique vertices are biased towards the higher degree vertices (intuitively it would seem reasonable that, for a randomly generated instance, vertices in the optimal maximum clique would tend to have higher vertex degrees). The p_hat1500-1 instance was created with a generator which is a generalization of the classical uniform random graph generator and has wider vertex degree spread and a larger maximal clique than the corresponding uniform random graph. For p_hat1500-1, only a single maximal clique appears to exist (Pullan and Hoos, 2006) and, as shown in Fig. 3, with a single exception, there is a bias towards the higher degree vertices. For brock800_1 however, the vertices in the optimal maximum clique are biased towards the lower than average vertex degree (Note that the DIMACS brock instances were created in an attempt to

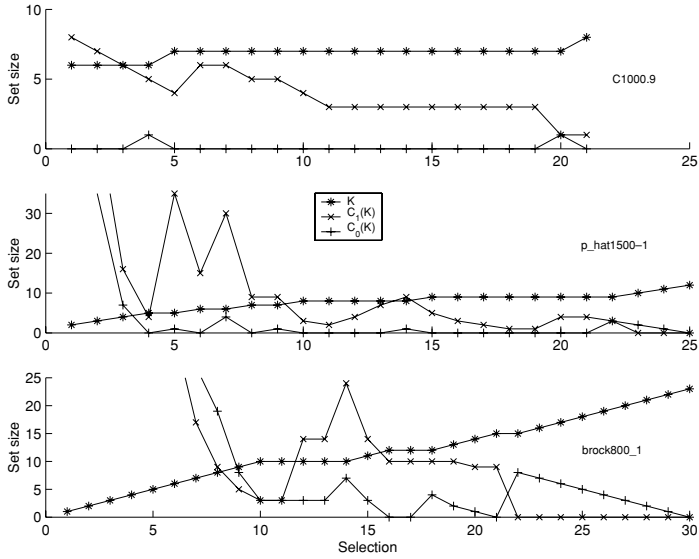

Fig. 4 $|K|$ (scaled by -60 for C1000.9), $|C_0(K)|$ and $|C_1(K)|$ during the final iteration for each of the C1000.9, p_hat1500-1 and brock800_1 instances. For C1000.9, the final vertex for the maximal clique was found after a sequence of 15 successive selections from $C_1(K)$ (plateau search) followed by the addition of the 68th vertex to the maximal clique. At the maximal clique, there is a single $C_1(K)$ vertex available to produce a second maximal clique. For p_hat1500-1, the situation is basically the same except for the final 3 selections which are all from $C_0(K)$ with no $C_1(K)$ vertices available at the maximal clique. For brock800_1, the final sequence of selections are 3 from $C_0(K)$, 1 from $C_1(K)$ followed by 8 from $C_0(K)$ with no $C_1(K)$ vertices available at the maximal clique

defeat greedy algorithms that used vertex degree for selecting vertices to be added to the current clique (Brockington and Culberson, 1996)).

A quantitative analysis of the maximum cliques for the DIMACS subset instances, showed that, for C1000.9, averaged over all maximal cliques found by PLS, the average vertex degree of vertices in the maximal cliques is 906 (standard deviation of 9) as compared to 900 (9) when averaged over all vertices; for p_hat1500-1, corresponding figures were 494 (84) and 380 (111) respectively while for brock800_1, they are 515 (11) and 519 (13) respectively.

From Table 4, for C1000.9, when $|K| = \omega(\text{C1000.9})(= 68)$, $E(|C_0(K)|) = 0.72$ and $E(|C_1(K)|) = 5.45$ which confirms that there is a high probability of multiple maximal cliques (70 distinct maximal cliques were found in 100 trials in Pullan and Hoos (2006)) for this instance and raises the possibility of a larger maximal clique existing. For p_hat1500-1, when $|K| = \omega(\text{p\_hat1500-1})(= 12)$, $E(|C_0(K)|) = 0.000088$ and $E(|C_1(K)|) = 0.0031929$ which suggests that p_hat1500-1 is an extreme variant of the corresponding random graph and that it is highly unlikely that more than a single maximal clique exists (a single unique maximal clique was found in 100 trials in Pullan and Hoos (2006)). A similar situation exists for brock800_1 as, when $|K| = \omega(\text{brock800\_1})(= 23)$, $E(|C_0(K)|) = 0.04$ and $E(|C_1(K)|) = 0.48$ (a single unique maximal clique was found in 100 trials in Pullan and Hoos (2006)).

Figure 4 provides a direct experimental indication of the nature of the hyper-surface, close to the maximal clique, for the three DIMACS subset instances used in this

discussion. For C1000.9, the final iteration to attain the maximal clique consisted of the sequence $3P, 1U, 15P, 1U$ where kP denotes k consecutive selections from $C_1(K)$ (plateau) and kU denotes k consecutive selections from $C_0(K)$ (up-hill). At the maximal clique, there is a single $C_1(K)$ vertex available to produce a second maximal clique (from which there may be additional maximal cliques available). This suggests that the hyper-surface for C1000.9, around the maximal clique, is predominantly a large, high level ($\approx$0.99 of the maximum hyper-surface height) plateau with the point corresponding to the maximal clique being just a single selection above the plateau. For p_hat1500-1, the final sequence is $3U, 1P, 1U, 1P, 1U, 1P, 1U, 4P, 1U, 7P, 3U$. At the maximal clique, there are no $C_1(K)$ vertices available. Note that, in this case, the final iteration for p_hat1500-1 was performed by the *Penalty* sub-algorithm which uses the *Initialise* perturbation. This sequence tends to suggest that the maximal clique for p_hat1500-1 is a moderate size "peak" on a mid- to high-level (≈ 0.75 of the maximum hyper-surface height) plateau. For brock800_1, the final sequence of selections is $10U, 4P, 2U, 2P, 3U, 1P, 8U$ with no $C_1(K)$ vertices available at the maximal clique. This suggests that the point corresponding to the maximal clique is a prominent peak (no $C_1(K)$ selections were available during the final sequence of 8 selections from $C_0(K)$) on a relatively small, low-level (≈ 0.65 of the maximum hyper-surface height) plateau.

4.3 Vertex selection

The vertex selection techniques implemented in the PLS sub-algorithms either bias the selection towards the higher degree vertices (*Degree*), the lower degree vertices (*Penalty*) or have no directly implemented bias (*Random*). However, in the *Random* case, there is an inherent bias towards the higher degree vertices simply from the fact that they are more likely to be present in $C_0(K)$ and $C_1(K)$. Table 3 gives the success rate, for each sub-algorithm, over 100 trials of PLS on the DIMACS subset. For C1000.9, the *Degree* sub-algorithm actually located the maximal clique in 80% of the trials with *Random* being responsible for the remaining 20% of successes. For p_hat1500-1, which is basically a randomly generated instance, the maximal clique contains a single low degree vertex and this makes p_hat1500-1 less susceptible to the *Degree* sub-algorithm and the *Random* sub-algorithm, which uniformly samples a wider degree range of vertices, is more effective. Because of the bias towards the lower degree vertices in the maximal clique for brock800-1, the *Penalty* sub-algorithm had a 100% success rate on this instance.

4.4 Perturbation techniques

For all PLS sub-algorithms, a perturbation is invoked when $C_0(K) = \emptyset$ and either $C_1(K) = \emptyset$, or all vertices that are in $C_1(K)$ have already been an element of K during the current iteration. Whilst it is possible to perform perturbations that generate a change in $|K|$ in the range $0, \ldots, |K| - 1$ (Grosso et al., 2005), PLS only uses two of these possibilities. The *Reinitialise* perturbation will generate, for a randomly generated instance, a starting clique for the next iteration of expected cardinality $P|K|$ where K is the final current clique from the preceeding iteration. This perturbation is appropriate when the maximal clique lies within a high-level plateau on the

Table 4 Expected cardinalities of $C_0(K)$ and $C_1(K)$ from Eq. (1) for randomly generated graphs corresponding to the three DIMACS subset instance near their maximal cliques

C1000.9			p_hat1500-1			brock800_1		
$\|K\|$	$C_0(K)$	$C_1(K)$	$\|K\|$	$C_0(K)$	$C_1(K)$	$\|K\|$	$C_0(K)$	$C_1(K)$
64	1.10356	7.84753	8	5.82813	46.625	19	0.21777	2.22799
65	0.99214	7.16547	9	2.91211	26.209	20	0.14137	1.52246
66	0.89197	6.54114	10	1.45508	14.5508	21	0.09177	1.03775
67	0.80192	5.96982	11	0.72705	7.99756	22	0.05958	0.70575
68	0.72095	5.44719	12	0.36328	4.35938	23	0.03868	0.47897

hyper-surface associated with the instance as it will tend to keep the search at a high-level and not waste effort searching the lower level plateaus. Such a hyper-surface can be produced for randomly generated instances such as C1000.9 where the maximal clique consists of vertices whose degree is relatively high. However, as the *Initialise* perturbation always generates a starting clique of cardinality 1 it is more appropriate when the maximal clique lies within a low to mid-level plateau as these plateaus will be searched as each iteration proceeds up-hill on the hyper-surface from the initially low starting point. Such a hyper-surface would be produced where the maximal clique contains some number of vertices whose degree is relatively low such as brock800_1.

In summary, where the search is focused on the higher degree vertices (higher level plateaus), a perturbation such as *Reinitialise* is more appropriate. However, where the search is focused on the lower degree vertices (lower level plateaus), a perturbation such as *Initialise* is more appropriate.

4.5 PLS sub-algorithms

To investigate the performance of the PLS sub-algorithms further, a variant of PLS, dubbed PLS−<Sub-algorithm>, where only the specified sub-algorithm was allowed to be active, was used. Table 5 shows the performance of PLS compared to the optimal sub-algorithm while Fig. 5 shows the proportion of selections performed, as a function of $|K|$, for the PLS sub-algorithms on the DIMACS subset instances. For:

- **C1000.9** Starting with $K = \phi$, the average number of consecutive selections from $C_0(K)$ is 44 and the maximum number is 55. As ω(C1000.9) = 68 this clearly indicates that plateau search is essential for locating the maximal clique for C1000.9. Also shown is that PLS-*Degree* performs relatively more searching of the higher level plateaus than PLS-*Random* and hence is more efficient at locating the maximal clique. The effectiveness of the *Degree* sub-algorithm is highlighted by comparing the results in Table 5 with those in Table 2 where the CPU time for C2000.9 decreased from 160.3759 seconds for DLS-MC to 112.8189 seconds for PLS to 24.1423 seconds for PLS-*Degree*.
- **p_hat1500-1** Starting with $K = \phi$, the average number of consecutive selections from $C_0(K)$ is 6 and the maximum number is 11 which indicates that it is feasible to attain the maximal clique (ω(p_hat1500-1) = 12) using $C_0(K)$ only. This is reflected in Table 3 which shows that, for p_hat1500-1, all 3 PLS sub-algorithms achieved a significant degree of success.

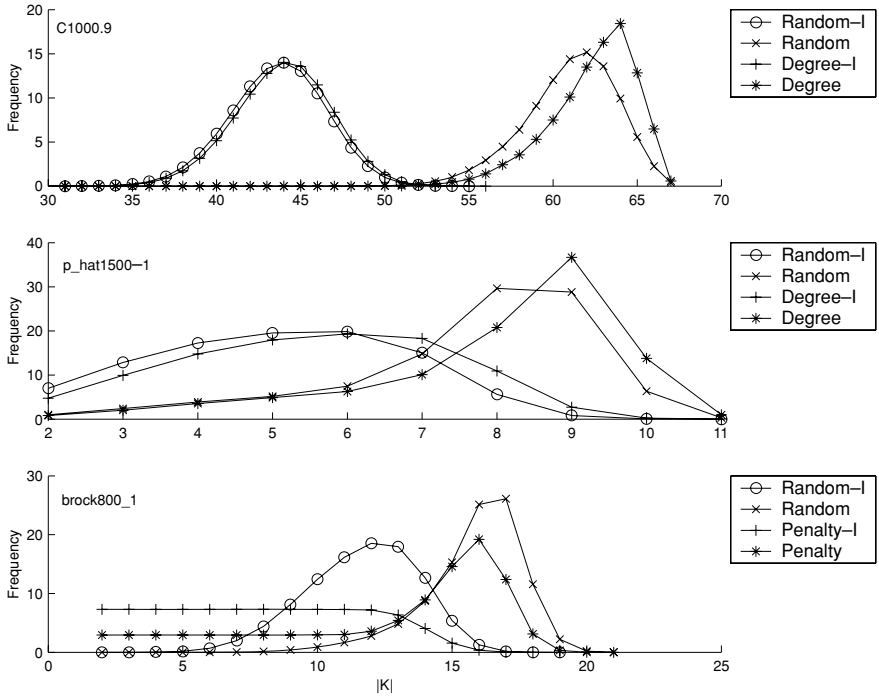

Fig. 5 Frequency of selections, as a function of $|K|$, for the *Random*, *Degree* and *Penalty* sub-algorithms for the C1000.9 (optimal sub-algorithm *Degree*), p_hat1500-1 (*Random*) and brock800_1 (*Penalty*) instances. The '-I' versions of the sub-algorithms did not perform any plateau searches

- **brock800_1** Starting with $K = \phi$, the average number of consecutive selections from $C_0(K)$ is 12 and the maximum number is 19. As ω(brock800_1) = 23 this clearly indicates that plateau search is essential for locating the maximal clique for brock800_1. Also shown is that PLS-*Random* performs relatively more searching of the higher level plateaus while PLS-*Penalty* tends to concentrate on the lower level plateaus.

The distributions of the average number of $C_1(K)$ selections before either a $C_0(K)$ or perturbation was performed for the DIMACS subset instances are shown in Fig. 6. The results for C1000.9 were obtained by PLS-*Degree*, those for p_hat1500-1 by PLS-*Random* and those for brock800_1 by PLS-*Penalty* sub-algorithms. Clearly shown is that PLS-*Penalty* spends considerably more effort on searching the mid-level plateaus than PLS-*Degree*. As described below for Fig. 4, the maximal clique for C1000.9 was attained in a single $C_0(K)$ selection from the plateau at $f(\mathbf{s}) = 67$, for p_hat1500-1 the maximal clique was attained in a direct up-hill climb from the plateau at $f(\mathbf{s}) = 9$, and that for brock800_1 from a direct up-hill climb from the plateau at $f(\mathbf{s}) = 15$.

There are only two choices for the sequencing of the sub-algorithms within PLS. The results presented in this paper are for the sequence *Random* $\Rightarrow$ *Penalty* $\Rightarrow$ *Degree*.

Table 5 Comparative results for PLS and PLS-<Sub-algorithm> (that is, PLS where only the specified sub-algorithm was used)

	PLS			PLS-<Sub-algorithm>			
Instance	CPU(s)	Sels.	Sels/Sec	Sub-algorithm	CPU(s)	Sels.	Sels/Sec
brock200_1	0.0036	2958	821624	*Penalty*	0.03	25032	834385
brock200_2	0.0294	14143	481059	*Penalty*	0.0163	8406	515703
brock200_3	0.0272	16235	596886	*Penalty*	0.0123	7695	625648
brock200_4	0.0776	52799	680402	*Penalty*	0.4757	327035	687481
brock400_1	1.0757	481300	447429	*Penalty*	2.3982	1134343	472997
brock400_2	0.3771	173598	460351	*Penalty*	0.4971	229240	461155
brock400_3	0.1798	81699	454388	*Penalty*	0.1987	92297	464503
brock400_4	0.1038	47535	457948	*Penalty*	0.0766	35652	465426
brock800_1	30.0919	5850513	194421	*Penalty*	43.8343	8962868	204471
brock800_2	24.4061	4736217	194058	*Penalty*	14.6352	3031502	207137
brock800_3	15.0795	2937434	194796	*Penalty*	15.7624	3247824	206048
brock800_4	6.5407	1263170	193124	*Penalty*	9.8108	2014630	205348
C1000.9	1.8839	701067	372135	*Degree*	0.5854	212777	363473
C125.9	0.0001	199	1989599	*Degree*	$<\epsilon$	56	–
C2000.5	0.7271	44977	61857	*Degree*	0.6712	40096	59737
C2000.9	112.8189	23563287	208859	*Degree*	24.1423	4669919	193433
C250.9	0.0022	2689	1222377	*Degree*	0.0009	854	949177
C4000.5	149.6532	4651204	31079	*Degree*	160.7299	4846521	30153
C500.9	0.1857	128858	693904	*Degree*	0.0782	40567	518757
keller6	550.9461	50206021	91126	*Random*	432.5392	44968698	103964
MANN_a45	28.7605	44230353	1537885	*Penalty*	46.0563	46823913	1016667
MANN_a81	269.6636	227500971	843647	*Penalty*	293.9578	120167634	408792
p_hat1500-1	3.2765	197988	60426	*Random*	3.0711	172692	56231
san1000	4.7187	368133	78015	*Random*	8.7664	725082	82711

The other possible sequence, *Random* $\Rightarrow$ *Degree* $\Rightarrow$ *Penalty*, gave similar results with the exceptions of MANN_a45 and MANN_a81 where the results were poor compared to those shown in Table 1 for the sequence *Random* $\Rightarrow$ *Penalty* $\Rightarrow$ *Degree*. The results shown in Table 5, which basically confirm the results obtained for the brock family in Table 2, show that there appears to be an inherent advantage in cycling around sub-algorithms (as compared to executing them in parallel). The influence of the preceeding *Degree* and *Random* sub-algorithms on the *Penalty* sub-algorithm occurs via the vertex penalties where, in effect, they are "re-set" towards those that occur following a random selction of vertices. In effect, a major perturbation periodically occurs in the search trajectory of the *Penalty* sub-algorithm and this appears to be beneficial to the search for these instances.

The other possible variation for sub-algorithms is the number of iterations allowed for each sub-algorithm invocation. Experimentally, it was determined that, overall, 50 iterations for *Random*, 50 iterations for *Penalty* and 100 iterations for *Degree* produced the best overall results for the DIMACS instances. Whilst the stage size is fixed for each sub-algorithm, as shown in Table 3, the relative number of selections performed by each sub-algorithm is dependent on the particular instance.

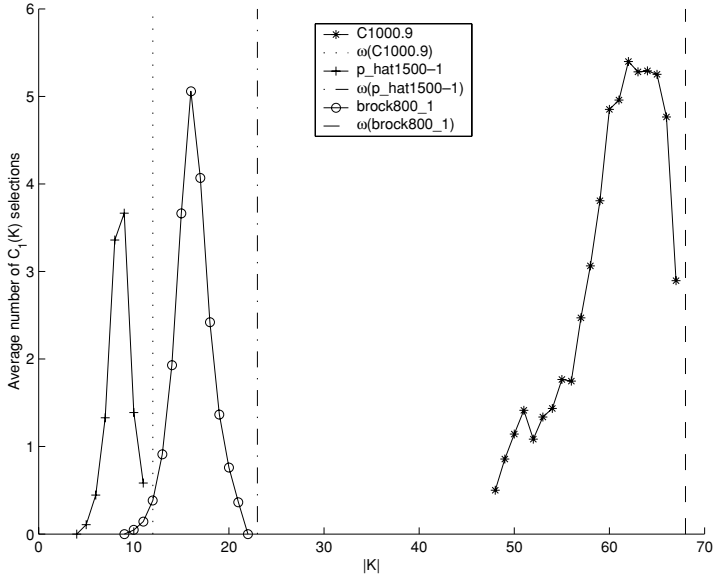

Fig. 6 Distributions of the average number of $C_1(K)$ selections before either a $C_0(K)$ selection or perturbation was performed for the DIMACS subset instances

5 Conclusions and future work

This study has demonstrated that by applying variants of the general paradigm of dynamic local search to the maximum clique problem, the state-of-the-art in MC solving can be improved. PLS builds on previous MC algorithms, in particular the recently introduced DLS-MC algorithm. Both algorithms use a combination of sub-algorithms when searching for maximum cliques with each sub-algorithm consisting of a clique building phase followed by a perturbation. Unlike DLS-MC, PLS does not require the user to supply an exogenous, family or sub-family instance-dependant, parameter to nominate and adapt the sub-algorithm to the instance. PLS combines three sub-algorithms which are effective for three different instance types. The first sub-algorithm, *Random*, effectively solves instances where the maximal clique consists of vertices with a wide range of vertex degrees. The second sub-algorithm, *Penalty*, uses vertex penalties to bias the search towards cliques containing lower degree vertices. The vertex penalties are increased when the vertex is in the current clique when a perturbation occurs and are subject to occasional decrease, which effectively allows the sub-algorithm to 'forget' vertex penalties over time. Unlike DLS-MC, where the frequency with which these decrease is fixed and externally nominated, the *Penalty* sub-algorithm of PLS adaptively modifies the frequency of penalty decreases to obtain near optimal performance. The final PLS sub-algorithm, *Degree*, uses vertex degrees to bias the search towards cliques containing higher degree vertices.

The fact that PLS has comparable, and sometimes improved, performance to DLS-MC on almost all of the standard DIMACS benchmark instances in combination with its excellent performance compared to other high-performance MC algorithms

clearly demonstrates the value of the underlying paradigm of combining variants of a basic local search algorithm.

The overall performance of PLS on standard MC instances reported here suggests that the underlying dynamic local search method has substantial potential to provide the basis for high-performance algorithms for other combinatorial optimisation problems, particularly weighted versions of MC and conceptually related clustering problems.

Acknowledgments The author would like to thank an anonymous referee for the considerable number of constructive comments which lead to significant improvements in this paper.

References

Balus E, Yu C (1986) Finding a maximum clique in an arbitary graph. SIAM J Comp 15(4):1054–1068

Battiti R, Protasi M (2001) Reactive local search for the maximum clique problem. Algorithmica 29:610–637

Bomze I, Budinich M, Pardalos P, Pelillo M (1999) The maximum clique problem. In: Du DZ, P.P. (ed) Handbook of Combinatorial Optimization, vol. A, pp 1–74

Boppana R, Halldórsson M (1992). Approximating maximum independent sets by excluding subgraphs. Bit 32:180–196

Brockington M, Culberson J (1996) Camouflaging independent sets in quasi-random graphs. In: Johnson DS, M. T. (ed) Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, vol. 26 of DIMACS Series. American Mathematical Society

Busygin S (2002) A new trust region technique for the maximum clique problem. Internal report, http://www.busygin.dp.ua.

Garey MR, Johnson DS (1979) Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness. Freeman, San Francisco, CA, USA

Grosso A, Locatelli M, Croce FD (2004) Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem. J. Heur 10:135–152

Grosso A, Locatelli M, Pullan W (2005) Randomness, plateau search, penalties, restart rules: simple ingredients leading to very efficient heuristics for the maximum clique problem. J. Heur (submitted)

Hansen P, Mladenović N, Urosević D (2004) Variable neighborhood search for the maximum clique. Disc Appl Math 145:117–125

Håstad J (1999). Clique is hard to approximate within n^{1-c}. Acta Math 182:105–142

Ji Y, Xu X, Stormo GD (2004) A graph theoretical approach for predicting common RNA secondary structure motifs including pseudoknots in unaligned sequences. Bioinformatics 20(10):1591–1602.

Johnson D, Trick M (Eds) (1996) Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, Vol. 26 of DIMACS Series. American Mathematical Society

Katayama K, Hamamoto A, Narihisa H (2004) Solving the maximum clique problem by k-opt local search. In: Proceedings of the 2004 ACM Symposium on Applied Computing, pp 1021–1025

Marchiori, E (2002). Genetic, iterated and multistart local search for the maximum clique problem. In Applications of Evolutionary Computing, vol. 2279 of Lecture Notes in Computer Science, pp 112–121. Springer Verlag, Berlin, Germany

15 Pevzner PA, Sze S-H (2000) Combinatorial approaches to finding subtle signals in DNA sequences. In: Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology, AAAI Press, pp 269–278

Pullan W, Hoos H (2006) Dynamic local search for the maximum clique problem. J Arti Intel Res 25:159–185

Resende M, Feo T, Smith S (1998). Algorithm 786: FORTRAN subroutine for approximate solution of the maximum independent set problem using GRASP. ACM Trans Math Soft 24:386–394

Wolpert D, Macready G (1997) No free lunch theorems for optimization. IEEE Trans Evolut Comp 1:67–82