The COMPSET Algorithm for Subset Selection

Yaniv Hamo and **Shaul Markovitch**
{hamo,shaulm}@cs.technion.ac.il
Computer Science Department, Technion, Haifa 32000, Israel

Abstract

Subset selection problems are relevant in many domains. Unfortunately, their combinatorial nature prohibits solving them optimally in most cases. Local search algorithms have been applied to subset selection with varying degrees of success. This work presents COMPSET, a general algorithm for subset selection that invokes an existing local search algorithm from a random subset and its complementary set, exchanging information between the two runs to help identify wrong moves. Preliminary results on complex SAT, Max Clique, 0/1 Multidimensional Knapsack and Vertex Cover problems show that COMPSET improves the efficient stochastic hill climbing and tabu search algorithms by up to two orders of magnitudes.

1 Introduction

The subset selection problem (SSP) is simply defined: Given a set of elements $E = \{e_1, e_2, \ldots, e_n\}$ and a utility function $U : 2^E \mapsto R$, find a subset $S \subseteq E$ such that $U(S)$ is optimal. Many real-life problems are SSPs, or can be formulated as such. Classic examples include SAT, max clique, independent set, vertex cover, knapsack, set covering, set partitioning, feature subset selection (classification) and instance selection (for nearest-neighbor classifiers) to name a few.

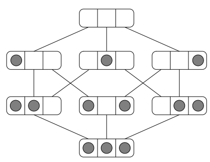
Since the search space is exponential in the size of E, finding an optimal subset without relaxing assumptions is intractable. Problems associated with subset selection are typically NP-hard or NP-complete. Local search algorithms are among the common methods for solving hard combinatorial optimization problems such as subset selection. Hill climbing, simulated annealing [Kirkpatrick *et al.*, 1983] and tabu search [Glover and Laguna, 1993] have all been proven to provide good solutions in a variety of domains. The general technique of random restarts is applicable to all of them, yielding anytime behavior as well as the PAC property (probability to find the optimal solution converges to 1 as the running time approaches infinity).

The problem with these local search algorithms is, however, that they are *too* general. The only heuristic they have about the domain is in a form of a black-box, the utility function which they try to optimize. It is therefore common to have modifications to these algorithms that trade being general for additional domain-specific knowledge. The SAT domain is full of such variants [Gent and Walsh, 1993; Hoos and Stützle, 2005] but they are common in other domains as well [Khuri and Bäck, 1994; Evans, 1998].

In this paper we present a general modification to known local search algorithms which improve their performance on SSPs. The idea behind it is to exploit attributes that are specific to search spaces of subset selection. Knowing that it is a subset search space allows us to infer which moves were likely to be wrong. By reversing these moves and trying again, we start from a new context, and the probability to repeat the mistake is reduced. In experiments performed on complex SAT, max clique, 0/1 multidimensional knapsack and vertex cover problems, the new method has shown to significantly improve the underlying search algorithm.

2 The COMPSET Algorithm

Subset selection can be expressed as a search in a graph. Each node (state) in the graph represents a unique subset. Edges correspond to adding or removing an element from the subset, thus there are $n = |E|$ edges to every node. The following figure shows the search graph for 3 elements.



A state S can be represented by a bit vector where bit $S_{[i]}$ is 1 iff $e_i \in S$. Moving to a neighboring state in the graph is equivalent to flipping one bit.

Each state is associated with a utility value, which is the value of U on the subset it represents. Local search algorithms typically start from a random state and make successive improvements to it by moving to neighboring states. They vary from each other mainly in their definition of neighborhood, and their selection method.

Using this representation, all local search algorithms are also applicable to subset selection. However, being general, they overlook the specific characteristics of subset selection. COMPSET guides a given local search algorithm using knowledge specific to subset selection.

2.1 Characteristics of Subset Selection

In a selection problem of n elements, there are n operators: $F = \{f_1, f_2, \ldots, f_n\}$ where f_i is the operator of toggling the membership of element i in a set (if it was in the set remove it , or add it if it was out). Applying f_i is equivalent to flipping the ith bit in the bit vector representation. Throughout the following discussion we assume a single optimal subset (solution) which we donate S^*.

We make the following observations about subset search:

Observation 1. *Let S' be an arbitrary state (subset). From any state S, there exists a subset of the operators, $\sigma(S, S') \subseteq F$, that when applied to S results in S'.*

Proof. Since S', as S, is an n bits long vector, there are at most n bits of S' that do not agree with S and need to be flipped using an operator. □

Observation 2. *Let $\overline{S}$ be the complementary state of S i.e., the state derived from S by flipping all its bits. The subset of operators $\sigma(S, S^*)$ leading from S to the solution S^*, is the complementary subset of $\sigma(\overline{S}, S^*)$ in F. That is, $\sigma(S, S^*) \cup \sigma(\overline{S}, S^*) = F$ and $\sigma(S, S^*) \cap \sigma(\overline{S}, S^*) = \emptyset$.*

Proof. We need to show, that for every $f_i{\in}F$ either $f_i \in \sigma(S, S^*)$ or $f_i \in \sigma(\overline{S}, S^*)$. If $S_{[i]} = S^*_{[i]}$ then $f_i \notin \sigma(S, S^*)$ (it does not need to be flipped). Moreover, if $S_{[i]} = S^*_{[i]}$, then necessarily $\overline{S}_{[i]} \neq S^*_{[i]}$, since in $\overline{S}$ all bits are flipped. Thus, $f_i \notin \sigma(S, S^*) \rightarrow f_i \in \sigma(\overline{S}, S^*)$. The same goes for the other possible case, in which $\overline{S}_{[i]} = S^*_{[i]}$. □

The inherent problem in finding $\sigma(S, S^*)$ using local search, is that operators are applied successively and their effect is not necessarily of monotonic improvement due to interdependencies between elements. Such non-monotonic behavior of U confuses local search algorithms and often makes them trapped in local optima. In such a case, there are two options: either the search is progressing on the correct path to the solution (but the algorithm does not see a way of continuing), or it is off the correct path altogether. It would be beneficial to distinguish between these two scenarios.

Consider two independent hill climbing runs, one from S and one from $\overline{S}$. Given that the optimal solution is not found, the two runs have stopped in local optima, L_S and $L_{\overline{S}}$ respectively. We consider the subsets of operators leading from S to L_S and from $\overline{S}$ to $L_{\overline{S}}$. By observation 2, it is not possible that an operator f_i appears in both operator subsets if they are both on the path to S^*. If we do observe the same operator in both, it is a clear sign that one of them is wrong. This is the idea behind COMPSET, which is described next.

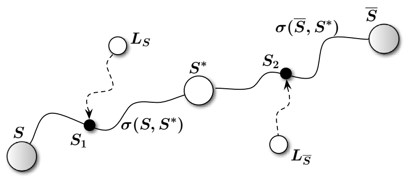
2.2 Description of the COMPSET Algorithm

Interdependencies between elements are distracting when searching for good solutions. Had all elements been independent, a simple linear search, which adds element after element as long as the utility value improves, would suffice. Local optima are an example where such interdependency brings the search to a full stop. It is likely that by applying the operators in a different order, or eliminating some, the local optimum would have been avoided. COMPSET uses the above observations to try and identify wrong invocations of operators. It then cancels them (reverses their effect) and resumes the run towards another local optimum or, hopefully, the solution.

Procedure COMPSET(S, LOCALSEARCHALG)
$S_1 \leftarrow S$; $S_2 \leftarrow \overline{S}$; agree ← false
loop until agree
  $L_S \leftarrow$ LOCALSEARCHALG*(S_1)*
  $L_{\overline{S}} \leftarrow$ LOCALSEARCHALG*(S_2)*
  C ← $\sigma(S, L_S) \cap \sigma(\overline{S}, L_{\overline{S}})$
  if C is empty then
    agree ← true
  else
    $S_1 \leftarrow C(L_S)$ *// apply all operators in C on L_S*
    $S_2 \leftarrow C(L_{\overline{S}})$
return the better between L_S and $L_{\overline{S}}$
end

Given a start state S, COMPSET initiates two runs of the given local search algorithm, one from S and one from its complementary $\overline{S}$. Once two local optima are achieved, the series of operators that has led to each one is examined. Every operator that appears in both series must be wrong in one of them. We do not know which of the runs went wrong, so we reverse the effect of such operators in both local optima. Once all obviously wrong operators are undone, the local search is continued. The process repeats upon encountering the next pair of local optima. When no conflicting operators exist, and the optimal solution was not found, COMPSET ends and returns the better of the two local optima in hand.

The rational behind COMPSET is illustrated here:



The solution S^* is reachable by applying all operators from $\sigma(S, S^*)$ to S (in any order), and by applying all operators from $\sigma(\overline{S}, S^*)$ to $\overline{S}$. The underlying search algorithm from S is in a correct path if it is using only operators from $\sigma(S, S^*)$.

However, a search can easily divert from this path, and it is often necessary for its overall convergence to the solution. Diversion occurs if an operator from $\sigma(\overline{S}, S^*)$ is applied to S, or an operator from $\sigma(S, S^*)$ is applied $\overline{S}$. The problem is, that since we do not know $\sigma(S, S^*)$ or $\sigma(\overline{S}, S^*)$, it is difficult to detect such diversions. However, what we do know, is that the solution S^* conforms to $\sigma(S, S^*) \cap \sigma(\overline{S}, S^*) = \emptyset$. We can use this fact to try and identify diversions.

Once stopped in local optima L_S from S and $L_{\overline{S}}$ from $\overline{S}$, we check $\sigma(S, L_S) \cap \sigma(\overline{S}, L_{\overline{S}})$. If the intersection is not empty then by definition either L_S or $L_{\overline{S}}$ are off the path. Note, that if the intersection is empty, the local optima might still be off path, because either $\sigma(S, L_S)$ uses an operator from $\sigma(\overline{S}, S^*)$, or $\sigma(S, L_{\overline{S}})$ uses an operator from $\sigma(S, S^*)$. However, $\sigma(S, L_S) \cap \sigma(\overline{S}, L_{\overline{S}}) \neq \emptyset$ means that *for sure* at least one of the local optima is off path.

In general it is possible that the search algorithm would continue applying operators and finally return to the path, but being in a local optimum means that it has essentially "given up". Elimination of all conflicting operators from both sides brings us to S_1 and S_2, $L_S \rightarrow S_1$ and $L_{\overline{S}} \rightarrow S_2$. Since all common operators were eliminated, $\sigma(S, S_1) \cap \sigma(\overline{S}, S_2) = \emptyset$ and thus S_1 and S_2 are on a possibly correct path. It is still possible that $\sigma(S, S_1)$ contains operators from $\sigma(\overline{S}, S^*)$ or that $\sigma(S, S_2)$ contains operators from $\sigma(S, S^*)$ thus still being an obstacle for reaching the solution.

An important point to notice, is that S_1 and S_2 are not necessarily states that the algorithm has visited before. Groups of operators are simultaneously eliminated, an operation which interdependencies would prohibit had the operators were successively eliminated. COMPSET effectively switches to another context which is mostly correct, in which the eliminated operators can be tried again.

3 Empirical Evaluation

The following algorithms were considered:

- *Stochastic Hill Climbing* (SHC) - starts from a random subset, iteratively picks a neighboring subset (differs in exactly one element) in random and moves there if it has a better or equal utility value. The simplicity of SHC often misleads; several works [Mitchell *et al.*, 1994; Baluja, 1995] showed that SHC does not fall from the complex GA mechanism. In the SAT domain such stochastic local search (SLS) methods have been shown to be comparable with state-of-the-art domain-specific algorithms [Hoos and Stützle, 2005].
- *Tabu Search* (TS) [Glover and Laguna, 1993] - examines the neighborhood of the current state for the best replacement. It moves to the chosen state even if it does not improve the current state, which might result in cycles. To avoid cycles, TS introduces the notion of a *tabu list* that stores the last t (*tabu tenure*) operators used. TS is prevented from using operators from the tabu list when it generates the neighborhood to be examined, unless certain conditions called *aspiration criteria* are met. In this paper we use a common aspiration criterion that allows operators which lead to better state than the best obtained so far.
- *Simulated Annealing* (SA) [Kirkpatrick *et al.*, 1983] - begins at high *temperature* which enables it to move to arbitrary neighboring states, including those which are worse than the current one. As the temperature declines, the search is less likely to choose a non-improving state, until it settles in a state which is a local minimum from which it cannot escape in low temperatures.

To test the effectiveness of COMPSET we have applied it to SHC and TS. COMPSET is not applicable to SA since SA begins with a high temperature at which it randomly moves far from the initial state. The concept of COMPSET is to set new start points for the underlying algorithm and by randomly moving away from them SA defeats its purpose.

The algorithms were tested in the following domains:

- *Propositional Satisfiability (SAT)* - the problem of finding a truth assignment that satisfies a given boolean formula represented by a conjunction of clauses (CNF) $C_1 \wedge \ldots \wedge C_m$. SAT is a classic SSP since we look for a subset of variables that when assigned a true value, makes the entire formula true. The utility function is the number of unsatisfied clauses when assigning true to all variables in S:

$$U(S) \equiv |\{C_i | C_i \text{ is false under } S, 0 \leq i \leq m\}|$$

The global minimum for U is 0, for satisfied formulas. A search algorithm using this utility function will attempt to maximize the number of satisfied clauses, which is a generalization of SAT called MAX-SAT. Problem instances for SAT were obtained from the SATLIB [Hoos and Stützle, 2000] repository of random 3-SAT. We use problems from the solubility phase transition region[1] [Cheeseman *et al.*, 1991].

- *Max Clique* - another classic SSP, where the goal is to find the maximum subset of vertices that forms a clique in a graph. Given a graph $G = (V, E)$ and a subset $S \subseteq V$, we define the following utility function:

$$U(S) \equiv \begin{cases} |V| - |S| & \text{S is a clique} \\ |V| - |S| + |V| + |S| \cdot (|S| - 1) - |E_S| & \text{else} \end{cases}$$

A clique should be maximized but our implementation always minimizes U, therefore we use $|V| - |S|$. Incomplete solutions are penalized by the number of additional edges they require for being a clique ($|S| \cdot (|S| - 1) - |E_S|$), plus a fixed value $|V|$ that is used to separate them from the legal solutions. By striving to minimize U, the search algorithm finds feasible solutions first, and then continues by minimizing their size. The global minimum of U corresponds to the maximum clique. Problem instances were obtained from the DIMACS [1993] benchmark for maximum clique.

- *0/1 Multidimensional Knapsack* (MKP) - the problem of filling m knapsacks with n objects. Each object is either placed in all m knapsacks, or in none of them (hence "0/1"). The knapsacks have capacities of $c_1, c_2, \ldots, c_m$. Each object is associated with a profit p_i and m different weights, one for each knapsack. Object i weighs w_{ij} when put into knapsack j. The goal is to find a subset of objects yielding the maximum profit without overfilling any of the knapsacks. Knapsack j is overfilled in state S iff $\sum_{i=1}^{n} S_{[i]} \cdot w_{ij} > c_j$. Let k be the number of overfilled knapsacks. We define:

$$U(S) \equiv \begin{cases} -\sum_{i=0}^{n} S_{[i]} \cdot p_i & \text{k=0} \\ k & \text{k} > 0 \end{cases}$$

The utility of feasible subsets is simply their profit (with minus sign for minimization purposes). Infeasible solutions are penalized for each knapsack they overfill. Problem instances for MKP were obtained from the OR-library [Beasley, 1997].

- *Vertex Cover* - the goal is to find the smallest subset of vertices in a graph that covers all edges. Given a graph $G = (V, E)$, we define:

$$U(S) \equiv \begin{cases} |S| & \text{S covers all edges} \\ |S| + |V| + |E \backslash E_S| & \text{else} \end{cases}$$

[1] Random 3-SAT problem with 4.26 clauses per variable that are the hardest to solve using local search.

For legal vertex covers, U takes values less than or equal to $|V|$. Incomplete solutions are penalized by the number of edges they do not cover, plus a fixed value $|V|$ that is used to separate them from the legal solutions. The global minimum of U corresponds to the optimal vertex cover.

The complementary graphs of the instances from the original DIMACS benchmark were taken, so that the known maximum clique sizes could be translated to corresponding minimum vertex covers[2].

3.1 Experimental Methodology

We have tested five algorithms: SHC, TS, SA (with $T = 100, \alpha = 0.95$), COMPSET over SHC and COMPSET over TS. TS was used with $t = 5$ for all domains other than SAT, and $t = 9$ for SAT. Each run was limited to 10^7 U evaluations.

All algorithms use random restart to escape from local optima when they have still not exhausted their evaluations quota. They use random restart also when there is no improvement over the last k steps. We use $k = 10$ for domains other than SAT, and $k = 20$ for SAT. SAT is characterized by wide and frequent plateaus [Frank *et al.*, 1997] therefore we chose higher values of t and k for it.

100 runs of each algorithm were performed on each problem in the test sets. Each run started from a random state, that was common to all algorithms. We measured the number of U evaluations needed to obtain the optimal solution in each run, as well as the time taken.

3.2 Results

The results are summarized in Tables 1, 2, 3 and 4. For brevity, we did not include the timing information in these tables. The considered algorithms do not introduce a significant overhead, so the execution time is a linear function of the number of U evaluations. The tables show the characteristics of the problem instance, followed by the number of successful runs (columns titled *#ok*) and the average number of U evaluations for each algorithm. A successful run is a run in which the algorithm has found the optimal solution within the limit. We tested the statistical significance of the improvement introduced by COMPSET using the Wilcoxon matched-pairs signed-ranks test with the extension by Etzioni and Etzioni [1994] to cope with censored data[3]. A "+" sign in the *sg.* column between SHC and COMPSET/SHC indicates that COMPSET improved SHC with $p < 0.05$. A "-" sign indicates that SHC performed better with $p < 0.05$. A "?" sign indicates that the difference is not significant. Whenever it is not possible to draw definitive conclusions since there is too much censored data, *n/a* appears. The same holds for the *sg.* column between TS and COMPSET/TS.

The superiority of COMPSET over the other algorithms is striking, both in the number of evaluations, and the number and difficulty of instances solved. In the SAT domain, the best performing algorithms are COMPSET/TS and COMPSET/SHC. The average success ratio of COMPSET/TS is 85% and of COMPSET/SHC is 76%. For comparison, the success ratios for SA, TS and SHC are 37%, 23% and 28% respectively. The speedup factor gained by using COMPSET is as large as 462 for SHC (instance uf50-011) and as large as 156 for TS (instance uf75-013). Note that these are lower bounds since SHC and TS were terminated because of resource limit for some of the runs.

The best performing algorithms in the Max Clique domain are COMPSET/SHC and SHC with average success ratios of 94% and 80% respectively. The speed up factor of COMPSET/SHC over SHC is as large as 14 (instnace sanr200_0.7).

In the Knapsack domain, the best performing algorithm is COMPSET/SHC with an average success ratio of 89%. For comparison, the success ratios for TS, SA, COMPSET/SHC and SHC are 50%, 25%, 19% and 17% respectively. The speedup factor gained by using COMPSET is as large as 127 for TS (instance WEISH07) and as large as 3.8 for SHC (instance WEISH04).

The best performing algorithms in the Vertex Cover domain are COMPSET/SHC and SHC with average success ratios of 94% and 77% respectively. SA is relatively close with 71% but TS is far behind with 23%, improved by COMPSET to 28%. The speedup factor gained by using COMPSET is as large as 310 for TS (instance hamming6-2) and as large as 10 for SHC (instance sanr200_0.7).

Another interesting statistics is the number of random restarts required by the underlying search algorithm and COMPSET, as well as the number of operator eliminations performed by COMPSET and how many operators they spanned. We have collected this data throughout 100 runs on the p_hat1000-1 vertex cover problem, a graph of 1000 vertices. SHC required 517.37 random restarts on average (in each run), while COMPSET required only 3.13. COMPSET has performed 20.52 operator eliminations, reversing the effect of 2.97 operators each time.

4 Conclusions and Future Work

In this paper, we have provided useful insights into the domain of subset selection. We have realized that using local search, paths from complementary subsets to the solution must be distinct in terms of the operators used. This has led us to conclude that if the paths contain common operators, it may serve as an indication of a mistake. To test our conjecture, we introduced COMPSET, a new guiding policy for local search algorithms in the context of SSP. The results show a significant improvement over both TS and SHC by up to two orders of magnitudes.

We currently in the process of running COMPSET on other subset selection domains, progressing towards a better understanding of its behavior. One interesting direction is to research for ways to incorporate knowledge of the entire search paths, instead of only the local minima at their end. In addition, it is beneficial to find out how characteristics of a specific problem affect its performance. Overall, the general idea of incorporating such SSP specific insights seems to be a promising lead to better subset selection algorithms.

[2]Note that while it is possible to take the complementary graph, solve the Max Clique problem, and then translate back to Vertex Cover, none of the algorithms in this paper has done so.

[3]The information about runs in which the solution was not found within the given bound is called *truncated* or *censored*.

3-SAT Instances			SHC			COMPSET/SHC		TS			COMPSET/TS		SA	
name	vars	clauses	#ok	evals	sg.	#ok	evals	#ok	evals	sg.	#ok	evals	#ok	evals
uf20-011	20	91	100	211	+	100	167	100	309	?	100	275	82	1,800,168
uf20-012	20	91	100	115	?	100	102	100	190	?	100	201	100	197
uf20-013	20	91	100	576	?	100	655	100	1,089	?	100	940	57	4,300,196
uf20-014	20	91	100	590	?	100	596	100	872	?	100	863	72	2,800,287
uf20-015	20	91	100	243	?	100	256	100	421	?	100	361	98	201,003
uf50-011	50	218	74	5,920,013	+	100	12,792	100	160,575	+	100	17,250	11	8,900,193
uf50-012	50	218	100	1,162,273	+	100	10,624	100	36,955	?	100	35,072	22	7,800,249
uf50-013	50	218	100	1,682,981	+	100	3,803	100	45,616	+	100	9,275	65	3,501,795
uf50-014	50	218	100	1,995,884	+	100	6,704	100	66,185	+	100	14,317	40	6,001,598
uf50-015	50	218	99	1,005,804	+	100	2,547	100	33,894	+	100	7,335	68	3,201,009
uf75-011	75	325	0	-	+	100	80,928	16	9,137,745	+	100	142,969	37	6,306,563
uf75-012	75	325	0	-	+	100	409,036	12	9,375,096	+	100	579,683	5	9,500,223
uf75-013	75	325	0	-	+	100	9,700	63	5,890,614	+	100	37,587	56	4,404,356
uf75-014	75	325	0	-	+	100	22,677	50	7,364,576	+	100	77,866	41	5,904,975
uf75-015	75	325	0	-	+	100	89,098	27	8,595,672	+	100	179,802	53	4,762,145
uf100-011	100	430	0	-	+	100	289,359	0	-	+	100	450,070	16	8,404,825
uf100-012	100	430	0	-	+	100	130,835	0	-	+	100	309,667	37	6,321,626
uf100-013	100	430	0	-	+	100	158,180	0	-	+	100	376,166	40	6,011,208
uf100-014	100	430	0	-	+	100	872,275	0	-	+	100	966,083	41	6,175,183
uf100-015	100	430	0	-	+	100	283,024	0	-	+	100	361,632	19	8,110,739
uf125-011	125	538	0	-	+	100	544,188	0	-	+	100	496,277	28	7,211,118
uf125-012	125	538	0	-	+	100	786,882	0	-	+	100	1,048,369	21	7,908,011
uf125-013	125	538	0	-	n/a	52	7,202,045	0	-	+	87	4,471,437	11	8,906,967
uf125-014	125	538	0	-	n/a	46	7,653,790	0	-	n/a	67	6,388,535	4	9,601,212
uf125-015	125	538	0	-	+	100	1,574,893	0	-	+	100	1,597,965	10	9,001,632
uf150-011	150	645	0	-	+	100	1,129,916	0	-	+	100	650,899	66	3,458,095
uf150-012	150	645	0	-	n/a	7	9,525,999	0	-	n/a	48	6,722,510	17	8,332,034
uf150-013	150	645	0	-	+	82	4,429,762	0	-	+	100	1,037,124	64	3,750,319
uf150-014	150	645	0	-	n/a	49	6,748,236	0	-	+	97	3,384,813	15	8,507,697
uf150-015	150	645	0	-	n/a	11	9,373,759	0	-	n/a	48	7,193,000	15	8,517,484
uf200-011	200	860	0	-	n/a	2	9,811,767	0	-	n/a	7	9,703,756	10	9,013,804
uf200-012	200	860	0	-	n/a	0	-	0	-	n/a	24	8,434,746	14	8,642,899
uf200-013	200	860	0	-	n/a	7	9,506,131	0	-	n/a	39	7,769,246	15	8,546,974
uf200-014	200	860	0	-	n/a	2	9,857,446	0	-	n/a	41	6,907,585	40	6,121,901
uf200-015	200	860	0	-	n/a	1	9,900,146	0	-	n/a	3	9,766,815	4	9,648,972

Table 1: SAT: average over all 100 runs, including censored data

Graphs			SHC			COMPSET/SHC		TS			COMPSET/TS		SA	
name	$\|V\|$	opt	#ok	evals	sg.	#ok	evals	#ok	evals	sg.	#ok	evals	#ok	evals
brock200_1	200	21	41	7,953,200	+	98	2,485,541	0	-	n/a	0	-	24	8,880,969
hamming6-2	64	32	100	659	?	100	633	80	2,984,249	+	100	14,163	100	1,129
hamming6-4	64	4	100	302	?	100	275	100	1,794	?	100	1,807	100	517
hamming8-2	256	128	100	4,317	?	100	4,519	6	9,402,266	n/a	14	8,612,975	100	6,486
hamming8-4	256	16	100	8,924	+	100	6,609	45	5,514,223	n/a	54	4,619,182	100	18,546
hamming10-2	1024	512	100	36,837	?	100	38,312	6	9,438,856	n/a	6	9,438,856	100	54,520
hamming10-4	1024	40	62	6,716,761	n/a	70	6,336,176	0	-	n/a	0	-	36	7,956,686
johnson8-2-4	28	4	100	81	?	100	90	100	299	?	100	305	100	204
johnson8-4-4	70	14	100	1,208	+	100	898	45	5,575,650	+	97	1,005,066	100	2,802
johnson16-2-4	120	8	100	740	?	100	631	100	6,470	?	100	6,305	100	745
johnson32-2-4	496	16	100	4,661	?	100	4,661	83	1,798,068	+	85	1,600,804	100	4,843
p_hat700-1	700	11	38	8,029,518	+	92	3,471,758	1	9,902,542	n/a	1	9,902,542	6	9,658,510
p_hat700-2	700	44	100	361,250	+	100	92,712	1	9,902,564	n/a	1	9,902,564	100	639,270
p_hat700-3	700	62	100	912,534	+	100	224,936	0	-	n/a	0	-	100	1,513,576
p_hat1000-1	1000	10	92	3,764,601	+	100	891,693	4	9,619,971	n/a	4	9,619,971	69	6,161,992
p_hat1000-2	1000	46	100	1,177,655	+	100	211,710	0	-	n/a	0	-	100	1,645,523
p_hat1000-3	1000	68	43	7,490,258	+	100	1,253,011	0	-	n/a	0	-	31	8,610,929
p_hat1500-1	1500	12	0	-	n/a	3	9,926,760	0	-	n/a	0	-	0	-
p_hat1500-2	1500	65	99	2,215,890	+	100	369,480	0	-	n/a	0	-	98	2,629,112
p_hat1500-3	1500	94	86	4,982,543	+	100	1,943,943	0	-	n/a	0	-	73	5,884,335
sanr200_0.7	200	18	100	2,189,803	+	100	151,344	0	-	n/a	1	9,900,505	91	3,530,091
sanr200_0.9	200	42	96	2,871,830	+	100	530,846	0	-	n/a	0	-	73	5,535,066
sanr400_0.5	400	13	26	8,561,953	+	86	4,167,521	0	-	n/a	0	-	7	9,683,128
sanr400_0.7	400	21	37	8,029,818	+	95	3,493,629	0	-	n/a	0	-	11	9,252,956

Table 2: Maximum Clique: average over all 100 runs, including censored data

References

[Baluja, 1995] S. Baluja. An empirical comparison of seven iterative and evolutionary function optimization heuristics. Technical Report CMU-CS-95-193, School of Computer Science, CMU, 1995.

[Beasley, 1997] J. E. Beasley. OR-library: a collection of test data sets. Technical report, Management School, Imperial College, London, 1997.

[Cheeseman *et al.*, 1991] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of IJCAI-91*, pages 331–336, 1991.

[DIMACS, 1993] DIMACS. Challenge problems for maximum clique, dimacs.rutgers.edu., 1993.

[Etzioni and Etzioni, 1994] O. Etzioni and R. Etzioni. Statistical methods for analyzing speedup learning experiments. *Machine Learning*, 14(1):333–347, 1994.

[Evans, 1998] Isaac K. Evans. Evolutionary algorithms for vertex cover. In V. W. Porto, N. Saravanan, D. Waagen, and A. E. Eiben, editors, *Evolutionary Programming VII*, pages 377–386, 1998.

[Frank *et al.*, 1997] J. Frank, P. Cheeseman, and J. Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.

Problems				SHC			COMPSET/SHC		TS			COMPSET/TS		SA	
name	n	m	opt	#ok	evals	sg.	#ok	evals	#ok	evals	sg.	#ok	evals	#ok	evals
HP1	28	4	3418	80	4,911,555	n/a	56	6,544,038	100	719,239	-	98	1,968,905	96	3,404,960
HP2	35	4	3186	13	9,252,758	n/a	7	9,594,786	53	6,704,029	n/a	65	6,402,176	17	9,235,896
PB1	27	4	3090	96	3,072,586	-	87	3,952,437	100	670,184	-	100	1,200,320	100	1,982,159
PB2	34	4	3186	16	9,158,395	n/a	8	9,670,627	61	5,895,593	n/a	76	5,213,605	31	8,258,914
PB4	29	2	95168	12	9,400,912	n/a	14	9,278,001	100	141,267	-	100	186,156	2	9,958,448
PB5	20	10	2139	100	352,913	?	100	299,587	100	80,253	-	100	735,297	100	318,187
PB6	40	30	776	74	5,420,914	n/a	74	5,318,641	30	7,715,915	+	100	17,418	25	8,949,831
PB7	37	30	1035	0	-	n/a	2	9,866,520	44	7,526,288	+	100	168,889	40	7,654,095
SENTO1	60	30	7772	0	-	n/a	0	-	1	9,900,037	+	100	167,722	0	-
SENTO2	60	30	8722	0	-	n/a	0	-	0	-	+	93	3,742,251	0	-
WEING1	28	2	141278	1	9,951,662	n/a	1	9,967,393	100	430,212	?	100	295,032	1	9,913,058
WEING2	28	2	130883	1	9,988,929	n/a	1	9,920,886	86	4,325,976	?	90	4,249,753	0	-
WEING3	28	2	95677	7	9,669,592	n/a	3	9,851,757	37	8,002,117	n/a	38	7,917,398	1	9,922,652
WEING4	28	2	119337	45	7,854,172	n/a	39	8,067,189	100	352,464	+	100	122,820	19	8,968,454
WEING5	28	2	98796	9	9,355,108	n/a	7	9,720,010	79	4,962,961	n/a	80	4,560,088	3	9,894,039
WEING6	28	2	130623	2	9,896,760	n/a	1	9,961,823	100	2,190,253	?	99	2,263,775	1	9,950,248
WEISH01	30	5	4554	0	-	n/a	8	9,627,601	100	142,532	+	100	6,808	56	6,542,273
WEISH02	30	5	4536	15	9,256,801	n/a	19	9,029,940	100	203,812	+	100	4,981	96	3,118,697
WEISH03	30	5	4115	13	9,367,427	n/a	44	7,691,588	100	133,341	+	100	15,052	92	4,043,836
WEISH04	30	5	4561	50	7,207,287	+	100	1,860,648	100	27,689	+	100	1,944	100	1,645,413
WEISH05	30	5	4514	93	3,749,094	+	100	1,856,242	100	45,793	+	100	3,300	96	2,753,007
WEISH06	40	5	5557	0	-	n/a	0	-	34	7,450,036	+	100	237,437	5	9,752,363
WEISH07	40	5	5567	0	-	n/a	2	9,904,913	41	7,047,862	+	100	55,207	16	9,189,652
WEISH08	40	5	5605	0	-	n/a	1	9,993,609	15	9,186,377	+	99	1,988,697	5	9,797,464
WEISH09	40	5	5246	0	-	n/a	1	9,965,165	51	6,243,931	+	100	62,502	1	9,981,328
WEISH10	50	5	6339	0	-	n/a	0	-	12	9,041,645	+	100	185,651	0	-
WEISH11	50	5	5643	0	-	n/a	0	-	7	9,393,532	n/a	79	4,126,644	0	-
WEISH12	50	5	6339	0	-	n/a	0	-	12	8,933,081	+	100	287,267	0	-
WEISH13	50	5	6159	0	-	n/a	0	-	11	9,123,859	+	99	1,041,062	0	-
WEISH14	60	5	6954	0	-	n/a	0	-	1	9,901,672	+	94	2,668,066	0	-
WEISH15	60	5	7486	0	-	n/a	0	-	11	9,008,543	+	100	729,395	0	-
WEISH16	60	5	7289	0	-	n/a	0	-	0	-	n/a	49	6,847,342	0	-
WEISH17	60	5	8633	0	-	n/a	0	-	1	9,906,889	+	100	1,344,074	2	9,844,304
WEISH18	70	5	9580	0	-	n/a	0	-	2	9,843,393	n/a	49	7,382,229	0	-
WEISH19	70	5	7698	0	-	n/a	0	-	0	-	n/a	4	9,705,819	0	-
WEISH20	70	5	9450	0	-	n/a	0	-	1	9,935,892	+	83	4,260,796	0	-

Table 3: Knapsack: average over all 100 runs, including censored data

Graphs			SHC			COMPSET/SHC		TS			COMPSET/TS		SA	
name	$\|V\|$	opt	#ok	evals	sg.	#ok	evals	#ok	evals	sg.	#ok	evals	#ok	evals
brock200_1	200	179	44	7,703,031	+	97	2,407,132	0	-	n/a	0	-	24	8,706,894
hamming6-2	64	32	100	705	?	100	663	74	3,241,435	+	100	10,444	100	1,258
hamming6-4	64	60	100	325	?	100	299	100	1,813	?	100	1,790	100	535
hamming8-2	256	128	100	4,442	?	100	4,642	11	8,903,987	n/a	27	7,320,998	100	6,176
hamming8-4	256	240	100	9,399	+	100	6,287	39	6,112,570	n/a	49	5,117,786	100	18,168
hamming10-2	1024	512	100	44,727	+	100	38,674	5	9,535,915	n/a	7	9,353,124	100	59,437
hamming10-4	1024	984	58	6,412,744	n/a	71	5,645,469	0	-	n/a	0	-	37	7,811,420
johnson8-2-4	28	24	100	75	?	100	75	100	292	?	100	292	100	201
johnson8-4-4	70	56	100	1,122	?	100	962	52	4,911,854	+	98	632,835	100	2,889
johnson16-2-4	120	112	100	618	?	100	618	99	106,329	?	100	6,423	100	742
johnson32-2-4	496	480	100	4,564	?	100	4,564	79	2,192,741	n/a	79	2,192,741	100	4,339
p_hat700-1	700	689	24	8,778,686	+	96	3,214,600	1	9,902,454	n/a	1	9,902,454	10	9,588,452
p_hat700-2	700	656	100	461,639	+	100	102,745	0	-	n/a	0	-	100	630,340
p_hat700-3	700	638	100	1,189,058	+	100	210,935	0	-	n/a	0	-	100	1,507,522
p_hat1000-1	1000	990	86	3,917,288	+	100	829,765	1	9,905,190	n/a	1	9,905,190	60	6,770,941
p_hat1000-2	1000	954	100	1,562,316	+	100	195,142	1	9,904,961	n/a	1	9,904,961	100	2,048,317
p_hat1000-3	1000	932	32	8,093,992	+	100	1,163,768	0	-	n/a	0	-	22	8,633,648
p_hat1500-1	1500	1488	0	-	n/a	2	9,958,079	0	-	n/a	0	-	0	-
p_hat1500-2	1500	1435	99	2,210,577	+	100	408,447	0	-	n/a	0	-	96	2,669,373
p_hat1500-3	1500	1406	75	5,533,569	+	100	2,070,512	0	-	n/a	0	-	74	5,381,934
sanr200_0.7	200	182	99	1,922,341	+	100	181,238	0	-	n/a	0	-	85	4,625,237
sanr200_0.9	200	158	98	3,074,483	+	100	551,969	0	-	n/a	1	9,900,390	78	5,005,884
sanr400_0.5	400	387	15	9,083,301	+	87	4,860,526	0	-	n/a	0	-	8	9,662,423
sanr400_0.7	400	379	27	8,500,584	+	96	2,838,407	0	-	n/a	0	-	9	9,543,268

Table 4: Vertex cover: average over all 100 runs, including censored data

[Gent and Walsh, 1993] Ian P. Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. In *National Conference on AI*, pages 28–33, 1993.

[Glover and Laguna, 1993] F. Glover and M. Laguna. Tabu search. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, 1993.

[Hoos and Stützle, 2000] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In *SAT20000: Highlights of Satisfiability Research in the year 2000*, pages 283–292. 2000.

[Hoos and Stützle, 2005] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search - Foundations and Applications*. Morgan Kaufman Publishers, 2005.

[Khuri and Bäck, 1994] S. Khuri and T. Bäck. An evolutionary heuristic for the minimum vertex cover problem. In *Genetic Algorithms within the Framework of Evolutionary Computation*, pages 86–90, 1994.

[Kirkpatrick *et al.*, 1983] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220, 4598:671–680, 1983.

[Mitchell *et al.*, 1994] M. Mitchell, J. H. Holland, and S. Forrest. When will a genetic algorithm outperform hill climbing. In *Advances in NIPS*, volume 6, pages 51–58, 1994.