An Adaptive Evolutionary Algorithm for the Satisfiability Problem

Claudio Rossi*
Dept. of Computer Science
Ca' Foscari Univ. of Venice
Via Torino 155
31073 Mestre-Venezia
Italy
rossi@dsi.unive.it

Elena Marchiori
Faculty of Sciences
Free University Amsterdam
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
elena@cs.vu.nl

Joost N. Kok
LIACS
Leiden University
P.O. Box 9512
2300 RA Leiden
The Netherlands
joost@liacs.nl

ABSTRACT

This paper introduces an adaptive heuristic-based evolutionary algorithm for the Satisfiability problem (SAT). The algorithm uses information about the best solutions found in the recent past in order to dynamically adapt the search strategy. Extensive experiments on standard benchmark problems for SAT are performed in order to asses the effectiveness of the proposed technique. The results of the experiments indicate that this technique is rather successful: it improves on previous approaches based on evolutionary computation and it is competitive with the best heuristic algorithms for SAT.

1. INTRODUCTION

The satisfiability problem is a well-known NP-hard problem with relevant practical applications (cf., e.g. [3]).

Given a boolean formula, one has to find an instantiation of its variables that makes the formula true. Recall that a boolean formula is a conjunction of clauses, where a clause is a disjunction of literals; a literal is a boolean variable or its negation, and a boolean variable is a variable which can assume only the values *true*, *false*. When all the clauses have the same number K of literals the problem is also called K-SAT.

The SAT problem has been extensively studied and many exact and heuristic algorithms for SAT have been introduced [2; 3]. Efficient heuristic algorithms for SAT include algorithms based on local search (cf. [2; 3]) as well as approaches based on evolutionary computation (e.g., [1; 4; 5; 9]).

The aim of this paper is to show how the integration of a local search meta-heuristic into a simple evolutionary algorithm yields a rather powerful hybrid evolutionary algorithm for solving hard SAT problems.

In our method a simple (1+1) steady-state evolutionary algorithm with preservative selection strategy is used to explore the search space, while a local search procedure is used for the exploitation of the search space. Moreover, a meta-heuristic similar to the one employed in TABU search [6] is used for adapting the value of the mutation rate during the execution, for prohibiting the exploration/exploitation of specific regions of the search space, and for re-starting the execution from a new search point when the search strategy does not show any progress in the recent past.

Extensive experiments conducted on benchmark instances from the literature support the effectiveness of this approach.

*This work has been done while the author was visiting LIACS.

SAC 2000 Como, Italy

2. EVOLUTIONARY LOCAL SEARCH

The idea of integrating evolutionary algorithms with local search techniques has been beneficial for the development of successful evolutionary algorithms for solving hard combinatorial optimization problems (e.g., [8; 9; 10]). In a previous work [9] we have introduced a simple local search based genetic algorithm for SAT. Here we consider the restriction of that algorithm to a population consisting of just one chromosome (thus crossover is not used). We call the resulting evolutionary algorithm EvoSAP. In the next section we show how EvoSAP can be improved by incorporating an adaptive diversification mechanism based on TABU search.

In EvoSAP a single chromosome is used, which produces an offspring by first applying mutation and next local search. The best chromosome between the parent and the offspring is selected for the next generation. The process is repeated until either a solution is found or a specified maximum number of iterations is reached. The pseudo-code of EvoSAP is given below.

```
PROCEDURE EvoSAP
 randomly generate chromosome C;
 apply Flip Heuristic to C;
 WHILE (optimum found or max num iterations reached) DO
  BEGIN
   C0=C;
   apply mutation to C;
   apply Flip Heuristic to C;
   IF (C0 better C) C=C0;
  END
END
```

Let us describe the main features of EvoSAP.

Representation. A chromosome is a bit string of length equal to the number of variables describing an instantiation of the variables of the considered SAT problem, where the value of the i-th gene of the chromosome describes the assignment for the i-th variable (with respect to an ordering of the variables).

Fitness function. The fitness function counts the number of clauses that are satisfied by the instantiation described by the chromosome. Clearly, a chromosome is better than another one if it has higher fitness.

Mutation. The mutation operator considers each gene and it flips it if a randomly generated real number in $[0, 1]$ is smaller than the considered mutation rate `mut_prob`.

```
PROCEDURE FLIP HEURISTIC
BEGIN
 generate a random permutation S of [1..n_vars]
 REPEAT
   improvement:=0;
   FOR i:=1..n_vars DO
    BEGIN
    flip S(i)-th gene of C;
    compute gain of flip;
    IF (gain >= 0)
     BEGIN
      accept flip;
      improvement:=improvement+gain;
     END
    ELSE
     flip S(i)-th gene of C; //restore previous value
    END
  UNTIL (improvement=0)
END
```

Flip Heuristic. In the local search algorithm we consider, called Flip Heuristic, each gene is flipped and the flip is accepted if the number of satisfied clauses increases or remains equal (gain$\geq$0). This process is repeated until no further improvement is obtained by flipping any of the genes. In the figure describing the Flip Heuristic in pseudo-code, `n_vars` denotes the number of the variables. The `gain` of the flip is computed as the number of clauses that become satisfied after the flip minus the number of clauses that become unsatisfied. If the gain is not negative then the flip is accepted, otherwise it is rejected. Note that we accept also flips that yield no improvement (gain=0), that is we allow side steps. The inner loop is repeated until the last scan produces no improvement.

3. ADDING ADAPTIVITY

In this section we describe how EvoSAP can be improved by incorporating an adaptive diversification mechanism based on TABU search. Observe that at each generation EvoSAP produces a local optimum. Suppose the Flip Heuristic directs the search towards similar (that is having small Hamming distance) local optima having equal fitness function values. Then we can try to escape from these local optima by prohibiting the flipping of some genes and by adapting the probability of mutation of the genes that are allowed to be modified.

To this aim, we use the following technique based on TABU search. A table is considered which is dynamically filled with chromosomes having best fitness. If the best fitness increases then the table is emptied. When the table is full, the chromosomes are compared gene-wise. Those genes which do not have the same value in all the chromosomes are labeled as 'frozen'.

Formally, the table can be represented by a (k, n) matrix T, where k is the number of chromosomes the table can contain, and n is the number of variables of the considered SAT problem. The entry $T(i, j)$ contains the value of the j-th gene in the i-th chromosome of T. Let *frozen* be an array of length n whose entry j is 0 if the j-th is not frozen, and it is 1 otherwise. Initially all genes are not frozen. When the table is filled, we consider the quantities $val(j) = \sum_{i=1}^{k} T(i, j)$, for every $j \in [1, n]$. If $val(j)$ is 0 or k then we set $frozen(j)$ to 1 (the j-th gene becomes frozen). We denote by *n_frozen* the number of frozen genes. The size k of the table T is a parameter. After computational testing, we decided to set k to 10. When the fitness of the best chromosome increases, the table is emptied and all genes are unfrozen, that is, $frozen(j)$ is set to 0 for every j, and *n_frozen* is set to 0. We use the information contained in T for adapting the search strategy during the execution: each time T is full, the mutation rate is recomputed, the flipping of frozen genes is prohibited, and possibly the execution is restarted from a new random search point. Let us describe how these three actions are performed. The mutation rate is set to $\frac{1}{2} \cdot n\_frozen/n$, thus $0 \leq mut\_prob \leq 0.5$. Frozen genes are not allowed to be flipped neither by the mutation operator nor by the Flip Heuristic. The rationale behind these two actions is the following. If table T becomes full it means that the search strategy has found for k times best chromosomes with equal fitness. A gene which is not frozen has the same value in all these chromosomes. This indicates that the search directs often to local optima containing the values of the not frozen genes. Therefore in the next iteration we allow to flip only not frozen genes in order to reach search points far enough from the attraction basin of those local optima. The mutation rate is chosen in such a way that the lower the number of not frozen genes is, the higher the probability will be to flip them. The term $\frac{1}{2}$ is used to keep the mutation rate smaller or equal than 0.5.

Finally the information in the table T is used for possibly restarting the search. The chromosomes in T are grouped into equivalence classes, each class containing equal chromosomes. If the number of equivalent classes is very small, that is less or equal than two, it means that the last k best chromosomes found so far are of just one or two forms, indicating that the search is strongly biased towards those chromosomes. Then it seems worth to re-start the search from a new randomly generated chromosome.

The overall Adaptive evolutionary algorithm for the SAtisfiability Problem, called ASAP, is summarized in pseudo-code below. Adaptive mutation is the mutation operator which allows to mutate only not frozen genes. Analogously, the adaptive Flip Heuristic allows only the flipping of non-frozen genes. The mutation rate is initially equal to 0.5, and the maximum number of iterations is set to 300000.

The termination condition in ASAP applies when either the optimum is found or the maximum number of iterations is reached.

```
PROCEDURE ASAP
 randomly generate chromosome C;
 apply Flip Heuristic to C;
 WHILE (not termination condition) DO
  BEGIN
   C0=C;
   apply adaptive mutation to C;
   apply adaptive Flip Heuristic to C;
   UPDATE_TABLE;
  END
END

PROCEDURE UPDATE_TABLE
BEGIN
 unfreeze all genes;
 IF (fitness C0 > fitness C)   C=C0;  /* discard C */
 ELSE
  IF (fitness C > fitness C0)
    BEGIN
     empty table T;
     add C to table T;
    END
  ELSE  /* fitness C0 = fitness C */
   BEGIN
    add C to table T;
    IF (table T full)
     BEGIN
      compute frozen genes;
      adapt mutation rate;
      count classes;
      IF (number of classes <= 2)
        RESTART;
      empty table T;
     END
   END
END
```

4. RESULTS OF EXPERIMENTS

In order to evaluate the performance of our algorithm we conduct extensive simulations on benchmark instances from the literature, and compare the results to those reported in previous work based on evolutionary computation as well as to the most effective local search algorithms for SAT.

4.1 Comparison with Evolutionary Algorithms

We will consider three evolutionary algorithms for SAT, here called FlipGA [9], RFGA [7] and SAW [1]. FlipGA is a heuristic based genetic algorithm combining a simple GA with the Flip Heuristic. RFGA uses an adaptive refining function to discriminate between chromosomes that satisfy the same number of clauses and a heuristic mutation operator. The SAW algorithm is a $(1,\lambda^*)$ (λ^* is the best λ found in a suitable number of test experiments) evolutionary strategy using the SAW-ing (stepwise adaptation of weights) mechanism for adapting the fitness function according to the behavior of the algorithm in the previous steps. We test ASAP on the same instances (test suites 1, 2) used in [1; 7; 9], which are 3-SAT instances generated using the generator developed by Allen van Gelder. These instances are available at http://www.in.tu-clausthal.de/~gottlieb/benchmarks/3sat. All instances lay in the phase transition, where the number of clauses is approximately 4.3 times the number of the variables.

- Test suite 1 contains four groups of three instances each. The groups have a number of variables of 30,40,50 and 100.
- Test suite 2 contains fifty instances with 50 variables.

The performance of the algorithms is measured first of all by the *Success Rate (SR)*, that is, the percentage of runs in which the algorithm found a solution for that instance or group of instances. Moreover, we use the *Average number of evaluations to Solution (AES)* index, which counts the average number of fitness evaluations performed to find the solution. Note that the AES takes into account only successful runs. Since our algorithm uses also local search, we use an estimation of the AES called *Average Flip cost in terms of fitness Evaluation to Solution(AFES)*. The AFES index is based on the number of flips performed during the execution of the local search (both accepted and not accepted flips are counted) and is an estimation of the cost of the local search step in terms of fitness evaluations. If the local search performs `n_flips` flips (including accepted and not accepted flips), one can estimate a cost of K*`n_flips/n_vars` fitness evaluations (cf. [9]), where `n_vars` is the number of variables in the instance and K is the clause length. This applies only to instances with fixed length clauses.

The results of the experiments are given in Tables 1, 2, where n and m denote the number of variables and of clauses, respectively. All the algorithms are run 50 times on each problem instance, and the average of the results is reported. Moreover, the algorithms terminate either if a solution is found or if a maximum of 300000 chromosomes have been generated.

The results show that ASAP has a very good performance, with SR equal to 1 in all instances, and smaller AFES than FlipGA in all but one instance (instance 2) where it has AFES slightly bigger than FlipGA.

Alg.	*SR*	*AFES*
ASAP	1	5843
FlipGA	1	6551
RFGA	0.94	35323

Table 2: Comparison of ASAP, FlipGA and RFGA on Test Suite 2

4.2 Comparison with Local Search Algorithms

We consider two local search techniques, GRASP [11] and GSAT [12], which are amongst the best local search algorithms for SAT. GRASP (Greedy Randomized Search Procedure) is a general search technique: a potential solution is constructed according to a suitable greedy heuristic, and improved by a local search procedure. These two steps are repeated until either an optimal solution is found or a maximum number of iterations has been reached. In (the extended version of) [11] four GRASP algorithms for SAT are introduced. GSAT is a greedy heuristic: one starts from a randomly generated candidate solution and iteratively tries to increase the number of satisfied clauses by flipping the value of a suitable variable. The variable chosen for flipping is the one that gives the highest increase in the number of satisfied clauses.

We compare these two algorithms with ASAP on a subset of the DIMACS instances reported in the extended version of [11]. All the considered instances are satisfiable. These instances for SAT stem from different sources and are grouped into families.

			ASAP		FlipGA		RFGA		SAW	
Inst.	n	m	SR	AFES	SR	AFES	SR	AES	SR	AES
1	30	129	1.00	27	1.00	120	1.00	253	1.00	754
2	30	129	1.00	2024	1.00	1961	1.00	14370	1.00	88776
3	30	129	1.00	498	1.00	784	1.00	6494	1.00	12516
4	40	172	1.00	59	1.00	189	1.00	549	1.00	3668
5	40	172	1.00	54	1.00	165	1.00	316	1.00	1609
6	40	172	1.00	1214	1.00	1618	1.00	24684	0.78	154590
7	50	215	1.00	85	1.00	219	1.00	480	1.00	2837
8	50	215	1.00	103	1.00	435	1.00	8991	1.00	8728
9	50	215	1.00	7486	1.00	11673	0.92	85005	0.54	170664
10	100	430	1.00	62939	1.00	132371	0.54	127885	0.16	178520
11	100	430	1.00	281	1.00	1603	1.00	18324	1.00	43767
12	100	430	1.00	298	1.00	1596	1.00	15816	1.00	37605

Table 1: Comparison of ASAP, FlipGA, RFGA and SAW on Test Suite 1

- The `aim` family contains artificially generated 3-SAT instances and are constructed to have exactly one solution. The number of variables is 50, 100 and 200 and the ratio n_clauses/n_vars is 2.0, 3.4 and 6.0. In total there are 36 instances.

- Family `par` instances arise from a problem in learning the parity function. These are 5 instances with a varying number of variables and clauses.

- The 16 `Jnh` instances are randomly generated and have a varying clause length.

- Instances `ii` arise from the "boolean function synthesis" problem; they have a number of variables ranging from 66 to 1728 and number of clauses ranging from few hundreds up to over 20 thousands. This family counts 41 instances.

Execution time is the performance measure used in the DIMACS Challenge to evaluate local search algorithms. Our code was written in `C` and ran on Intel Pentium II (Memory: 64 Mb ram, Clock: 350 MHz, Linux version: Red Hat 5.2). In order to compare ASAP with GRASP and GSAT, we report in Table 11 the results of the DIMACS Challenge machine benchmark on the Pentium II and on the SGI Challenge, the machine used in the experiments with GRASP and GSAT reported in (the extended version) [11]. The results indicate that the Pentium II is approximately 1.5 times faster than the SGI Challenge.

The results of the experiments are given in Tables 4-10. Again, n and m denote the number of variables and of clauses, respectively. All the algorithms are run 10 times on each instance. In the tables containing the results of ASAP we give the average number of iterations, of restarts, of accepted flips, and the average time (in seconds) together with the standard deviation. In the Tables comparing ASAP with GRASP and GSAT we give the average time of ASAP (run on Pentium II), and report the results contained in (the extended version of) [11] (run on SGI Challenge), where an entry labeled '-' means that the result for that instance has not been given in [11].

All algorithms were always able to find the solution on every instance, except on the instances relative to the entries labeled 'NF' (not found) where ASAP was not able to find a solution.

The results of the tables comparing ASAP with GRASP and GSAT show that ASAP is competitive with these two algorithms, except on the instance aim-100-2_0-yes1 and on those of the class `par16-c`, where ASAP is not able to find any solution within 300000 iterations. On the other instances, we can summarize the results as follows. The performance of ASAP on the class `aim` is rather satisfactory, finding the solution in much shorter time than GRASP on some instances, like aim-200-6_0-yes1. On the class `par8` ASAP outperforms GSAT and has performance comparable to the one of GRASP. However, on the class `par16` ASAP is not able to find any solution. On the class `Jnh` GSAT outperforms GRASP as well as ASAP, with ASAP and GRASP giving comparable results. Finally, on class `ii` ASAP outperforms GRASP and GSAT on instances `ii8` and `ii16` (except ii16e1 where GSAT is faster), while GSAT outperforms GRASP and ASAP on instances `ii32`, being ASAP on the average faster than GRASP.

5. DISCUSSION

It is interesting to analyze the role of the adaptation mechanism in ASAP. To this aim, we compare experimentally ASAP with its non-adaptive variant EvoSAP. On instances of the Test Suites 1,2 the performance of EvoSAP is similar to the one of ASAP. However, on the DIMACS instances EvoSAP has a worse performance than ASAP. For example, on the instance jnh212 EvoSAP has a success rate of 0.9, it takes 10855 iterations (on the average) to find a solution, and over 1.2 millions (on the average) of accepted flips.

As illustrated in the tables on the DIMACS experiments, the restart mechanism of ASAP is not used in some experiments (e.g., on the classes `aim-100-6_0` and `ii8`). However, in other experiments, the mechanism is more effective. For example, on the instance jnh212 the performance of ASAP without the restart mechanism becomes poor: ASAP is able to find a solution only in five of the ten runs. Thus the results indicate that the adaptation mechanism of ASAP improves the performance of the evolutionary algorithm.

In conclusion, on the tested benchmarks ASAP has a rather satisfactory performance, indicating that hybridization of evolutionary algorithms with meta-heuristics based on local search provides a powerful tool for solving hard satisfiability problems.

6. REFERENCES

[1] T. Bäck, A. Eiben, and M. Vink. A superior evolutionary algorithm for 3-SAT. In D. W. N. Saravanan and

Inst.	n	m	Iterations	Restarts	Accepted Flips	Time	
						avg	SDev
aim-50-2_0-yes1	50	100	28112	349	1693310	18.758	22.744
aim-50-3_4-yes1	50	170	42	1	1979	0.050	0.073
aim-50-6_0-yes1	50	300	3	0	143	0.006	0.008
aim-100-3_4-yes1	100	340	115	6	11672	0.329	0.420
aim-100-6_0-yes1	100	600	4	0	386	0.022	0.017
aim-200-3_4-yes1	200	680	2774	208	614675	20.70	28.348
aim-200-6_0-yes1	200	1200	6	0	1376	0.110	0.089

Table 3: Results of ASAP on class `aim`

Inst.	ASAP	GRASP-A	GRASP-B	GRASP-C	GRASP-D	GSAT
aim-50-2_0-yes1	18.76	2.23	4.86	2.27	2.14	3.33
aim-50-3_4-yes1	0.05	0.60	1.01	0.59	0.36	0.10
aim-50-6_0-yes1	0.01	0.08	0.07	0.08	0.05	0.03
aim-100-2_0-yes1	NF	543.51	1386.51	2048.52	836.12	5883.12
aim-100-3_4-yes1	0.33	30.94	54.71	46.71	34.00	0.85
aim-100-6_0-yes1	0.02	0.66	0.71	0.72	0.63	0.35
aim-200-3_4-yes1	20.70	-	-	-	-	-
aim-200-6_0-yes1	0.11	126.99	121.90	176.91	120.04	-

Table 4: Comparison of ASAP, GRASP and GSAT on class `aim`

A. Eiben, editors, *Proceedings of the 7th Annual Conference on Evolutionary Programming*, Lecture Notes in Computer Science, pages 125–136. Springer, 1998.

[2] R. Battiti and M. Protasi. Approximate algorithms and heuristics for MAX-SAT. In D.-Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 77–148. Kluver Academic Publisher, 1998.

[3] D. Du, J. Gu, and P. P. (Eds.). *Satisfiability Problem: Theory and Applications*. AMS, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol 35, 1997.

[4] A. Eiben and J. van der Hauw. Solving 3-SAT with adaptive Genetic Algorithms. In *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, pages 81–86. IEEE Press, 1997.

[5] G.Folino, C.Pizzuti, and G.Spezzano. Combining cellular genetic algorithms and local search for solving satisfiability problems. In *Proc. of ICTAI'98 10th IEEE International Conference Tools with Artificial Intelligence*, pages 192–198. IEEE Computer Society, 1998.

[6] F. Glover and D. D. Werra. *Tabu Search*. Vol.41, Baltzer Science, 1993.

problem	machine		SGI/P2 CPU Time Ratio
	P2/350	SGI	
r100.5.b	0.01	0.02	2
r200.5.b	0.41	0.58	1.49
r300.5.b	3.56	4.87	1.41
r400.5.b	22.21	30.35	1.45
r500.5.b	86.15	116.72	1.45

Table 11: Machine Benchmark statistics: Runnuning time (seconds) of `dfmax` on Pentium II and SGI Challenge

[7] J. Gottlieb and N. Voss. Improving the performance of evolutionary algorithms for the satisfiability problem by refining functions. In A. Eiben, T. Bck, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature (PPSN V)*, LNCS 1498, pages 755 – 764. Springer, 1998.

[8] E. Marchiori. A simple heuristic based genetic algorithm for the maximum clique problem. In J. C. et al., editor, *ACM Symposium on Applied Computing*, pages 366–373. ACM Press, 1998.

[9] E. Marchiori and C. Rossi. A flipping genetic algorithm for hard 3-SAT problems. In *Genetic and Evolutionary Computation Conference*, 1999.

[10] P. Merz and B. Freisleben. Genetic local search for the TSP: New results. In *IEEE International Conference on Evolutionary Computation*, pages 159–164. IEEE Press, 1997.

[11] M. Resende and T. Feo. A GRASP for satisfiability. In D. Johnson and M. Trick, editors, *Cliques, Coloring and Satisfiability*, pages 499–520. AMS, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol 26, 1996.

[12] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI-92*, pages 440–446. AAAI Press/The MIT Press, 1992.

Inst.	n	m	Iters	Rest.	Accepted Flips	Time	
						avg	SDev
par8-1-c	64	254	149	9	8418	0.26	0.41
par8-2-c	68	260	121	6	7374	0.23	0.16
par8-3-c	75	298	281	14	18487	0.58	0.54
par8-4-c	67	266	469	19	23015	0.80	0.49
par8-5-c	75	298	674	46	45501	1.43	1.16

Table 5: Results of ASAP on class `par`

Inst.	ASAP	GRASP-A	GRASP-B	GRASP-C	GRASP-D	GSAT
par8-c	0.65	0.16	0.17	3.62	2.80	99.37
par16-c	NF	1981.13	3541.71	5205.23	3408.44	25273.14

Table 6: Comparison of ASAP, GRASP and GSAT on class `par`

Inst.	n	m	Iterations	Restarts	Accepted Flips	Time	
						avg	SDev
jnh1	100	850	852	34	83459	16.32	13.28
jnh7	100	850	21	1	2351	0.43	0.16
jnh12	100	850	129	71	12669	2.57	2.45
jnh17	100	850	837	3	8477	1.67	1.40
jnh201	100	800	21	0	2458	0.42	0.34
jnh204	100	800	368	18	36046	6.49	4.01
jnh205	100	800	203	6	20037	3.59	2.20
jnh207	100	800	1529	97	151247	26.63	24.36
jnh209	100	800	359	22	35115	6.22	5.15
jnh210	100	800	21	1	2288	0.39	0.28
jnh212	100	800	4430	273	429970	78.00	85.15
jnh213	100	800	49	2	4890	0.85	0.97
jnh217	100	800	30	1	3219	0.56	0.35
jnh218	100	800	35	1	3820	0.66	0.65
jnh220	100	800	1644	105	162005	28.85	24.93
jnh301	100	900	1232	81	115059	25.58	16.65

Table 7: Results of ASAP on class `Jnh`

Inst.	ASAP	GRASP-A	GRASP-B	GRASP-C	GRASP-D	GSAT
jnh1	16.32	11.87	5.19	10.14	8.11	0.71
jnh7	0.43	3.61	1.76	2.07	1.09	0.07
jnh12	2.51	0.84	1.36	1.24	1.95	0.74
jnh17	1.67	1.66	2.00	3.52	2.89	0.19
jnh201	0.42	1.48	0.50	0.73	0.74	0.05
jnh204	6.49	14.64	17.67	17.67	22.75	0.77
jnh205	3.59	6.17	6.28	7.90	10.08	0.50
jnh207	26.63	3.61	4.39	5.93	3.30	1.74
jnh209	6.22	7.45	6.07	6.73	6.44	0.46
jnh210	0.39	2.35	0.89	1.89	2.59	0.12
jnh212	78.00	70.92	29.77	27.28	112.84	6.31
jnh213	0.85	9.43	5.92	2.46	4.30	0.41
jnh217	0.56	5.76	2.23	3.50	2.00	0.16
jnh218	0.66	1.45	1.06	2.19	1.60	0.09
jnh220	28.85	10.17	18.08	8.95	20.18	2.98
jnh301	25.58	46.23	22.13	36.79	43.41	1.10

Table 8: Comparison of ASAP, GRASP and GSAT on class `Jhn`

Inst.	n	m	Iterations	Restarts	Accepted Flips	Time	
						avg	SDev
ii8a1	66	186	8	0	767	0.009	0.010
ii8a2	180	800	2	0	494	0.010	0.009
ii8a3	264	1552	3	0	1407	0.043	0.032
ii8a4	396	2798	7	0	4807	0.187	0.248
ii8b1	336	2068	1	0	759	0.014	0.005
ii8b2	576	4088	17	0	20590	0.460	0.336
ii8b3	816	6108	24	0	44949	0.996	0.538
ii8b4	1068	8214	31	0	78649	1.775	1.519
ii8c1	510	3065	1	0	881	0.019	0.005
ii8c2	950	6689	4	0	7806	0.186	0.111
ii8d1	530	3207	7	0	6192	0.153	0.249
ii8d2	930	6547	4	0	7707	0.198	0.145
ii8e1	520	3136	2	0	1886	0.050	0.021
ii8e2	870	6121	4	0	7842	0.214	0.117
ii16a1	1650	19368	6	0	20726	1.32	1.57
ii16a2	1602	21281	182	14	573916	32.60	28.66
ii16b1	1728	24792	7	0	22959	3.33	1.48
ii16b2	1076	16121	40	2	66398	7.39	2.73
ii16c1	1580	16467	20	0	46468	5.09	3.25
ii16c2	924	13803	46	2	67029	8.18	4.71
ii16d1	1230	15901	9	0	21051	3.15	2.51
ii16d2	836	12461	49	3	63324	8.85	6.40
ii16e1	1245	14766	2	0	4778	1.22	0.42
ii16e2	532	7825	30	1	24379	6.11	3.73
ii32a1	459	9212	47	2	28236	13.07	11.28
ii32b4	381	9618	675	56	312122	207.93	251.55
ii32c4	759	20862	83	6	91839	100.46	63.66
ii32d3	824	19478	254	20	292584	188.41	144.29
ii32e5	522	11636	173	8	122631	84.62	68.92

Table 9: Results of ASAP on class `ii`

Inst.	ASAP	GRASP-A	GRASP-B	GRASP-C	GRASP-D	GSAT
ii8a4	0.187	0.23	0.30	0.32	0.24	0.09
ii8b4	1.775	369.37	681.60	163.25	129.07	15.62
ii8c1	0.019	37.26	82.19	32.02	12.33	0.03
ii8d2	0.198	3.23	3.12	3.45	4.31	0.64
ii8e2	0.214	21.97	10.00	19.57	15.30	0.62
ii16a2	32.70	1970.58	-	-	-	1373.2
ii16b1	3.33	449.99	-	-	-	9.03
ii16c2	8.18	43.30	11.20	16.89	78.71	39.08
ii16d2	8.85	56.32	20.97	7.47	47.71	19.54
ii16e1	1.22	74.62	17.80	10.82	52.93	0.61
ii32a1	13.07	68.36	8.93	1.66	53.66	1.85
ii32b4	207.93	28.21	3.64	3.38	40.24	1.50
ii32c4	100.46	200.97	43.21	47.25	139.21	7.78
ii32d3	188.41	666.73	119.68	20.03	1136.34	22.91
ii32e5	84.72	16.47	2.31	3.21	24.17	1.75

Table 10: Comparison of ASAP, GRASP and GSAT on class `ii`