A Time-Sensitive System for Black-Box Combinatorial Optimization

Vinhthuy Phan, Pavel Sumazin, and Steven Skiena*

State University of New York at Stony Brook, Stony Brook, NY 11794-4400 USA
{phan,psumazin,skiena}@cs.sunysb.edu

1 Introduction

When faced with a combinatorial optimization problem, practitioners often turn to black-box search heuristics such as simulated annealing and genetic algorithms. In *black-box optimization*, the problem-specific components are limited to functions that (1) generate candidate solutions, and (2) evaluate the quality of a given solution. A primary reason for the popularity of black-box optimization is its ease of implementation. The basic simulated annealing search algorithm can be implemented in roughly 30-50 lines of any modern programming language, not counting the problem-specific local-move and cost-evaluation functions. This search algorithm is so simple that it is often rewritten from scratch for each new application rather than being reused.

In this paper, we examine whether it pays to develop a more sophisticated, general-purpose heuristic optimization engine. The issue is whether a substantial performance improvement or ease-of-use gain results from using such an engine over the naïve implementation of search heuristics.

The case for well-engineered optimization engines has clearly been made for linear programming, where commercial LP packages (such as CPLEX) significantly outperform homegrown implementations of the simplex algorithm. The answer is not so clear for local search heuristics. A review [13] of six existing black-box optimization packages describes generally disappointing results. Indeed, one line of theoretical research has yielded the "no free lunch theorem" [19], suggesting that under certain assumptions the choice of search engine has no effect on the expected quality of the solution. Despite or perhaps because of this, an enormous variety of different heuristic search algorithms and variants have been proposed; see [1] for an excellent overview of this literature.

We believe that there is a role for a well-engineered heuristic search system, and this paper describes our preliminary efforts to build such a system. Table 1 provides at least anecdotal evidence of our success, by comparing the performance of our current combined search heuristic after one year of development to the tuned simulated annealing engine we developed after six months of work. The new engine produces substantially better results for all problems across almost all time scales.

Our paper is organized as follows. In Sect. 2, we outline some of the potential benefits which can accrue from building a well-engineered heuristic search system. In Sect. 3, we describe the general architecture of our prototype implementation *discropt*, which currently supports five different search heuristics to evaluate on each of six different combinatorial optimization problems. We then describe the search heuristics currently implemented in Sect. 4. In Sect. 5, we report on a series of experiments designed to compare the search heuristics on a wide variety of problems and time scales. We conclude with an analysis of these results and directions for further research.

2 Rationale

Many of the potential benefits of a general search system result from amortizing the cost of building a substantial infrastructure over many problems. These include:

- *Awareness of Time* – Running time is an important consideration in choosing the right heuristic for a given optimization problem. The choice of the right heuristic (from simple greedy to exhaustive search)

* Supported in part by NSF Grant CCR-9988112 and ONR Award N00149710589.

Table 1. A comparison of the solution scores produced by our current combined heuristic with a previous version after six months of development. The better score appears in boldface.

Problem	Heuristic	1	5	10	15	20	25	30	45	60	90	120
VertCover	Early Version	2299	2260	2315	642	654	644	650	641	643	598	586
	Combination	**648**	**1100**	**571**	**554**	**541**	**546**	**528**	**531**	**521**	**521**	**523**
Bandwidth	Early Version	1970	**604**	**616**	**611**	**603**	**595**	622	620	609	**595**	**602**
	Combination	**693**	633	620	614	615	613	**609**	**607**	**608**	603	**602**
MaxCut	Early Version	**3972**	3972	3972	3972	3972	3969	3972	3972	3885	1373	1349
	Combination	**3972**	**1401**	**1385**	**1346**	**1462**	**1452**	**1441**	**1971**	**1689**	**1276**	**1240**
SCS	Early Version	3630	3688	1152	1158	1225	1120	1035	1013	994	902	982
	Combination	**1538**	**614**	**596**	**570**	**562**	**544**	**541**	**515**	**505**	**502**	**505**
TSP	Early Version	5101	1043	846	730	642	687	568	551	508	577	504
	Combination	**1133**	**933**	**633**	**472**	**411**	**406**	**397**	**378**	**381**	**375**	**374**
MaxSat	Early Version	27725	**572**	705	735	533	403	518	344	518	352	318
	Combination	**912**	636	**359**	**465**	**503**	**398**	**374**	**262**	**206**	**255**	**236**

depends upon how much time you are willing to give to it. A general search engine allows the user to specify the time allotted to the computation, and manages the clock so as to return the best possible answer on schedule.

A time budget, constructed using an experimental determination of solution-evaluation and construction times, is required for proper time management. Once the budget and basic search landscape have been determined, a reasonable search strategy can be chosen. By monitoring progress and time as the search unfolds we can judge the soundness of our selection and, if necessary, change strategies.

- *Ease of Implementation* – By incorporating objects for common solution representations (such as permutations and subsets) and interfaces for common input data formats, a generic system can significantly simplify the task of creating a code for a new optimization problem. Indeed, the six problems currently implemented in our system each require less than 50 lines of problem-specific code.
- *Inclusion of Multiple Heuristics* – A well-engineered heuristic search system can contain a wide variety of greedy, dynamic programming, and non-local search heuristics; uncovering solutions which would not be revealed just by local search. The cost of creating such an infrastructure cannot be justified for any single application. Our experience with generic greedy heuristics, reported in this paper, shows that they are valuable for a general search engine.
- *Parameter Tuning* – Search variants such as simulated annealing require a distressingly time-consuming effort to tune parameters such as the cooling schedule and distribution of moves. A well-engineered system can perform statistical analysis of the observed evaluation function values and automatically tune itself to obtain reasonable performance.
- *Testing and Evaluation Environment* – The choice of the best optimization engine for a given job is a difficult and subtle problem. Randomized search heuristics have a high variance in the quality of solutions found. Creating a proper testing and evaluation environment is a substantial enough task that it must be amortized over many problems. The reported success of many published heuristics is due to inadequate evaluation environments.
- *Visualization Environment* – Animation tools which plot search progress as a function of time are very helpful in analyzing the behavior of the system and tuning parameters. Figures 1 and 4 show examples of plots built by our current implementation constructs, including (a) the value of the current solution as a function of time, (b) positions in the solution where changes proved beneficial, i.e. hot-spots, and (c) the number of evaluation function calls as a function of time, to measure the effect of system load over a long computation. This level of visualization support is difficult to justify building for each individual application.
- *Solution Quality Estimation* – A statistical analysis of the distribution of evaluated solution candidates can be used to estimate the number of solutions better than any given solution. Provided that the scores are normally distributed, this is a measure for the quality of the given solution. There is theoretical

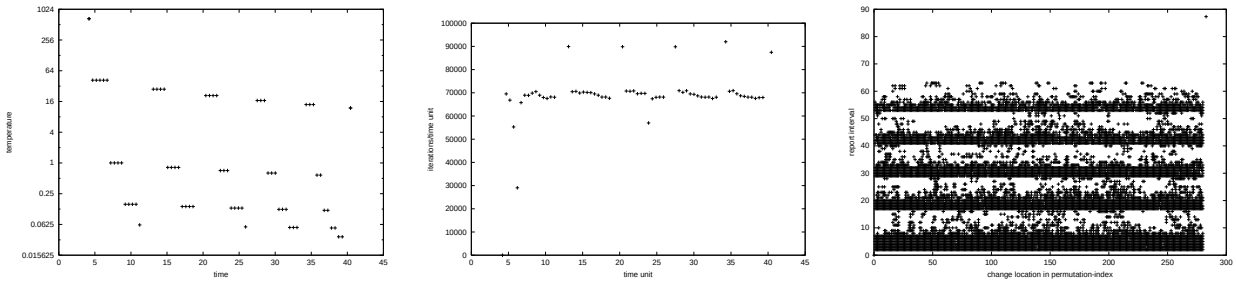

Fig. 1. Animating the fractional-restart search algorithm, including the temperature schedule (left), the evaluation rate (center), and positional hot spots vs. time (right).

and experimental evidence to support the assumption that the distribution of scores is normal for many problems.

2.1 Literature

The literature on heuristic search is vast, encompassing dozens of books and a great number of papers. Heuristics such as *simulated annealing* [10], *genetic algorithms* [5] and *tabu search* [4] are particularly popular. We refer the interested reader to reviews of search heuristics for combinatorial optimization problems, including [1, 2, 20]. Our view on search algorithms has been shaped by our own combinatorial dominance theory of search, developed in [14].

Johnson, Aragon, McGeoch, and Schevon [8, 9] conducted experimental evaluations of simulated annealing on graph partitioning, graph coloring and number partitioning. They found that simulated annealing was more effective on graph partitioning and coloring than number partitioning. They concluded that no variant of simulated annealing under consideration outperforms the rest across the board in any optimization problem. Our system provides a sound framework to conduct such systematic experiments.

A review of six black-box optimization systems is given by Mongeau, Karsenty, Rouz, and Hiriart-Urruty [13]. They tested the engines on three types of problems: least median square regression, protein folding and multidimensional scaling, before concluding that no good self-adjusting black-box optimization engine is available in the public domain today. Most of the reviewed optimization engines [3, 6, 7, 11, 12, 21] require a user adjustment of parameters, which heavily influences the engine's performance.

3 System Issues

In this section, we discuss the architecture of our prototype system and our test system, including a description of the optimization problems we evaluate it upon.

3.1 System Architecture

The general blueprint of our prototype system, written in C++, is given in the UML diagram in Figure 2. Our design consists of a central control unit, *solver*, that assigns work to the appropriate heuristic(s), and controls the information and statistics-gathering processes. The solver controls the:

- *Combinatorial engine*, which is a generalization of our search heuristics; it forces a common interface on the heuristics and provides access mechanisms.
- *Evaluator*, which maintains a combinatorial solution, and information related to the solution (most notably its cost, correctness and state).
- *Combinatorial solution*, which is a generalization of permutation and subset solution types.
- *Monitor*, which maintains real time and system time information; it notifies the engine when to report solutions, when to evaluate its position in the search process and when to terminate.

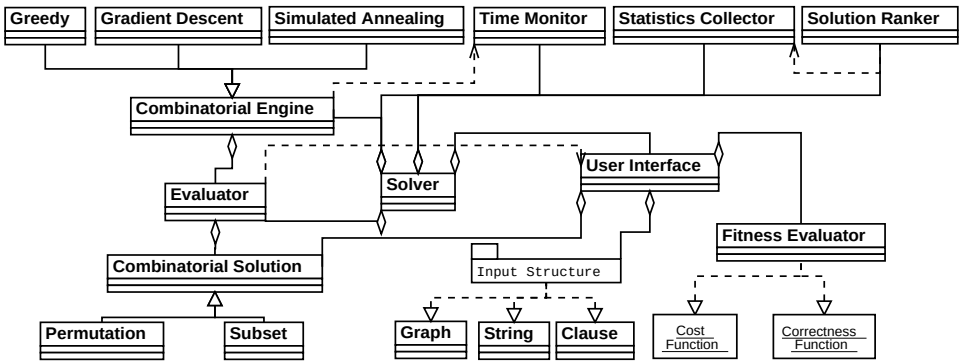

Fig. 2. The general blueprint of the system.

- *Statistics collector*, which gathers information about cost function evaluation time (including partial solution evaluation), search progress in the form of solution evaluation snapshots, score distribution and more.
- *User interface*, which is the only component accessible to system users; it is used to define cost functions and specify the solution type.

User-defined cost functions include a complete solution cost function (see example in Figure 3), and (optionally) correctness, partial cost, and partial correctness functions. The partial-cost function is particularly useful for a fast evaluation of solutions related to previously-evaluated solutions. An example is the partial-cost function for $Swap$: given a previously evaluated solution s_0, its swap-neighbor s_1, and the cost of $Swap(s_0, s_1)$, the user can define the operation $\oplus$ so that $\text{cost}(s_1) = \text{cost}(s_0) \oplus \text{cost}(Swap(s_0, s_1))$. In the case of TSP, the partial solution is evaluated in constant time, while a complete solution is evaluated in $O(n)$ time.

```
double UserInterface::cost_function(CombinatorialSolution & solution)
{
  double weight=0;
  solution_index first=solution.get_first_element(), last=solution.get_last_element(), i;
  element next, current = solution.element_lookup(first);
  for(i=first; i<last; current=next, i++){
    next = solution.element_lookup(solution.next_element(i));
    weight += input->edge_weight(current, next);
  }
  weight += input->edge_weight(solution.element_lookup(last), solution.element_lookup(first));
  return weight;
}
```

Fig. 3. An example of cost function for the Traveling Salesman Problem.

3.2 Test Problems

We have carefully selected a suite of combinatorial optimization problems for their diverse properties. Our primary interest is in the performance of the search heuristics across a general spectrum of problems. Due to lack of space, we limit the discussion to one instance for each problem. However, we assure the reader that the reported results are representative of the other instances in our test bed. Indeed, our full experimental results are available at [15].

- *Bandwidth* - given a graph, order the vertices to minimize the maximum difference between the positions of the two vertices defining each edge. Bandwidth is a difficult problem for local search, because the solution cost depends completely on the maximum stretch edge. The representative test case is file

alb2000 from TSPlib [16, 17], a graph of 2000 vertices and 3996 edges. The number of C++ lines of code used to define the complete and partial cost functions are, respectively, 9 and 30.

- *Traveling Salesman* - given an edge-weighted graph, find a shortest tour visiting each vertex exactly once. TSP is well suited to local search heuristics, as move generation and incremental cost evaluation are inexpensive. The representative test case is a complete graph on 666 vertices from TSPlib [16], gr666. Lines of code: 9 and 23.
- *Shortest Common Superstring* - given a set of strings, find a shortest string which contains as substrings each input string. SCS can be viewed as a special case of directed, non-Euclidean TSP, however here it is deliberately implemented by explicitly testing the maximum overlap of pairs of strings on demand. This makes the evaluation function extremely slow, limiting the performance of any search algorithm. Our test case consists of 50 strings, each of length 100, randomly chosen from 10 concatenated copies of the length-500 prefix of the Book of Genesis. Lines of code: 13 and 23.
- *Max Cut* - partition a graph into two sets of vertices which maximize the number of edges between them. Max cut is a subset problem which is well suited to heuristic search. The representative test case is the graph alb4000 from TSPlib [16] with 4000 vertices and 7997 edges. Lines of code: 10 and 17.
- *Max Satisfiability* - given a set of weighted Boolean clauses, find an assignment that maximizes the weights of satisfiable clauses. Max sat is a difficult subset problem not particularly well suited to local search. The representative test case is the file jnh10 from [18], a set of 850 randomly weighted clauses of 100 variables. Lines of codes: 12 and 14.
- *Vertex Cover* - find a smallest set of vertices incident on all edges in a graph. Vertex cover is representative of constrained subset problems, where we seek the smallest subset satisfying a correctness constraint. The changing importance of size and correctness over the course of the search are a significant complications for any heuristic. The representative test case is a randomly generated graph of 1000 vertices and 1998 edges. Lines of code: 8 and 2 for cost, 19 and 2 for correctness.

4 Search Algorithms

We present two fundamentally different approaches for combinatorial optimization, specifically variants of simulated annealing – a neighborhood search technique, and incremental greedy heuristics. A neighborhood search heuristic traverses a solution space S, using a *cost function*, $f : S \to \Re$, moving from one solution to another solution in its *neighborhood* $N : S \to 2^S$. We use the *swap* operation as our neighborhood operator. In the case of permutation solutions, a swap exchanges the positions of two permutation elements; in the case of subset solutions, a swap operation moves a given element in or out of the subset. For certain problems, the solution space may contain invalid solutions. It is then necessary to have a *correctness function*, $c : S \to \Re$, which indicates a solution's degree of validity. A non-neighborhood search method such as *incremental greedy* constructs solutions by incrementally adding solution fragments.

4.1 Simulated Annealing

Simulated annealing is a gradient descent heuristic that allows backward moves at a rate that is probabilisticaly proportional to $e^{-\frac{\Delta}{c}}$, where Δ is the difference in score between two solutions and c is the current temperature. The higher the temperature, the more likely the acceptance of a backward move. The initial temperature is set high to allow wide solution-space exploration, and is cooled down as the search progresses, eventually leading to a state where no backward moves are accepted. At each temperature, the search explores L_k states or solutions. Performance of a simulated annealing strategy depends its cooling strategy. Our system includes three variants of simulated annealing; typical search progress for these variants is given in Figure 4.

One-run Simulated Annealing This strategy reduces its temperature from the highest to the lowest once, using the full running time. After L solutions are explored, the current temperature c is replaced by $c \cdot \Delta c$, for some Δc computed by (1) estimating the number of decrements of temperature, $k = \frac{T}{\Delta t}$, where T is the remaining time; Δt is the time to explore L solutions, and (2) letting the search reach the estimated

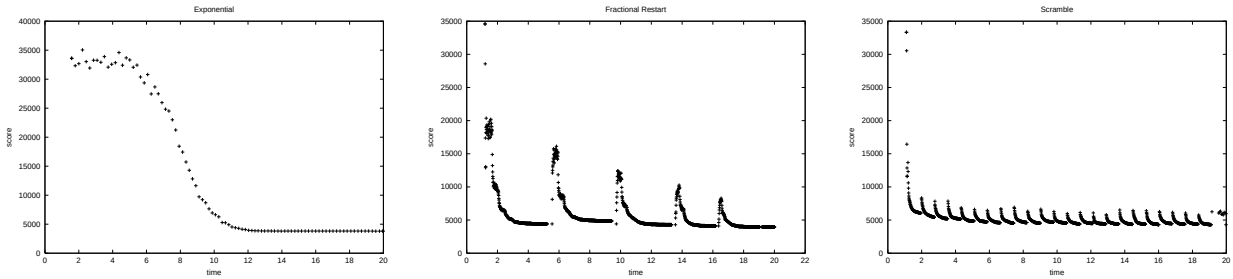

Fig. 4. Solution score vs. time for three different simulated annealing schedules: exponential, fractional-restart and scramble.

lowest temperature. Holding L and therefore Δt constant, we can deduce Δc. For greater time accuracy, Δc is updated dynamically each time the temperature is reduced.

When Δt is too large in comparison to T, k is too small to allow for a smooth temperature reduction. When Δt is too small in comparison to T, temperature may get decreased too quickly, leading to a premature termination. To balance these extremes, L assumes a small initial value and is increased gradually.

Multiple-restart Simulated Annealing The multiple-restart strategy consists of two steps: first converge quickly to a local optimum; second make a number of backward moves in the vicinity of that local optimum. We study two restart variants. *Fractional restart* resets the initial temperature to a fraction of the previous initial temperature; the fraction is computed so that at least one restart trial is carried out before time runs out. *Scrambling* explicitly *backtracks* from the local optimum; the backtracking distance is a predetermined fraction of the previous restart.

4.2 General Greedy Heuristics

The greedy heuristic constructs a solution by incrementally adding elements to a partial solution. At each increment, greedy selects the best of k candidate elements for inclusion; k is determined by the specified running time. For ordered solutions such as permutations, the system includes two greedy variants: (1) *interleave* places the best candidate at the best solution position, and (2) *append* appends the best candidate to the existing solution. For subset problems, there is no distinction between the two heuristics.

4.3 A Combined Strategy

The performance of search strategies vary across different problems, and across different problem instances. The *combined* strategy allots each heuristic a fraction of the time, then chooses the best candidate heuristic based on the information gathered from the initial runs. While a heuristic's performance after a short running time need not be indicative of its performance over a longer time, in practice we are able to predict whether greedy will perform better than simulated annealing on a vast majority of our test instances.

4.4 Engine Efficiency

While some system and optimization overhead is unavoidable, an engine that spends only a small portion of its execution time on solution evaluation is doomed to poor performance. We evaluate the overhead of each of our heuristics in Table 2.

The proportion of time spent examining new solutions depends on the following factors:

– *The amount of overhead done by the engine* – this includes initialization, recalibration, statistics gathering, and time management. For example, if the specified running time is barely larger than the time needed for initialization, the proportion of time spent on evaluation must be small. The greedy heuristic spends more than 95% of the time in evaluation of shortest common superstring solutions over 5 seconds, but less than 50% when the running time is under 1 second.

– *The evaluation time for each solution* – much of the overhead work is not instance dependent, therefore when evaluation is costly, the proportion of the time spent on evaluation is larger. The evaluation function for shortest common superstring is more time-intensive than the evaluation of a comparable size traveling-salesman solution, and the proportion of time spent on evaluation is accordingly greater.
– *The frequencies of partial and complete solution evaluations* – the evaluation time for each solution is dependent on the evaluation method. Partial evaluation is often less time-intensive than complete evaluation. The interleave heuristic performs many partial solution evaluations, while the append heuristic performs mostly complete solution evaluations for permutation based solutions. Thus, for instances of bandwidth and TSP the portion of time spent on evaluation by append is much larger than that of interleave.

Table 2. The percentage of time spent on evaluating solutions. The instances are the same as those reported in Fig. 5. The *append* and *interleave* heuristics are identical for *subset* problems (maxcut, maxsat, and vertex cover) as discussed previously. Note that some solution evaluations are used in the initialization phase and do not contribute to search progression.

Program	Append	Interleave	Exponential	Frac. Restart	Scramble	Combination
Bandwidth	81.9%	42.1%	58.7%	55.0%	51.7%	63.0%
TSP	80.7%	55.1%	60.5%	62.0%	56.1%	58.4%
SCS	96.5%	94.8%	97.0%	97.4%	96.9%	94.9%
MaxCut	34.7%	34.7%	44.9%	46.2%	44.3%	42.3%
maxSat	97.8%	97.8%	97.2%	97.6%	97.3%	98.1%
VertCover	51.5%	51.5%	43.4%	51.1%	51.2%	55.0%

5 Experimental Results

Our experimental results are given in Fig. 5 and Table 3. The experiments were carried out on an AMD 1Ghz personal computer running Linux with 768MB of RAM. Each heuristic was performed 10 times for each heuristic on running times ranging from 1 to 120 seconds.

For each instance/heuristic, there is a time threshold below which its performance is uncharacteristically bad. When given a running time below this threshold, the heuristic aborts its intended strategy and resorts to simple gradient descent search. For example, note the performance of interleave on the bandwidth and TSP instances in Fig. 5. Some heuristics perform well for linear objective functions but not for non-linear objective function. For example, observe the behavior of append on bandwidth. Our experiments suggest that greedy heuristics are useful in the context of a time-sensitive combinatorial system, and that a combination of greedy and simulated annealing often outperforms either stand-alone component heuristic.

Results for one instance of each problem are presented, but the analyzed behavior appears representative of all problem instances (see [15]). The results support the following claims:

– No single heuristic outperforms the rest across all six problems.
– Given a collection of heuristics, there is an advantage in combining and selecting the best heuristics for a given problem or instance. Indeed, our combined heuristic performs significantly better than the five pure heuristics as reported in Table 3. The combined heuristic produces the best results for at least some time interval for every problem, and for every time interval on some problem.
– Greedy heuristics do better than simulated annealing when time is short, but for longer running times, the simulated annealing heuristics will excel even on problems that have constant-factor greedy approximation algorithms, such as vertex cover. For example, simulated annealing does not perform as well as greedy on our traveling salesman instance within the 120 seconds allotted. However, given 3 hours, the exponential simulated annealing heuristic finds a solution of score 326828, which is within near optimal optimal and far better than the solution found by the greedy heuristic.
– Greedy heuristics and other constructive strategies do better than local search when energy landscapes are relatively flat; for example, shortest common superstring.

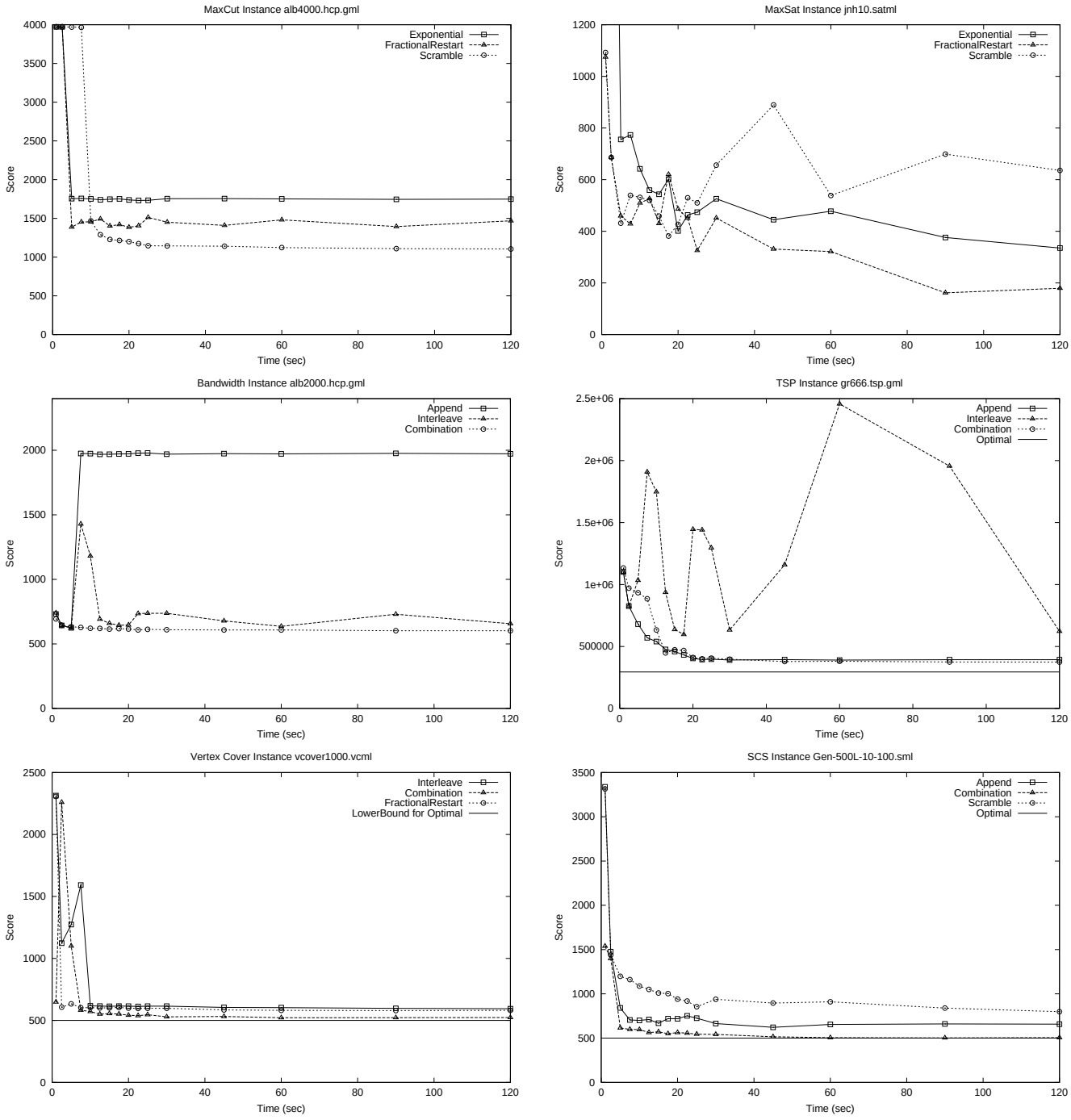

Fig. 5. Best score versus running time. In the top figures we compare the simulated annealing heuristics; in the middle figures we compare the greedy heuristics; and in the bottom figures we compare the best performing simulated annealing heuristics and the best performing greedy heuristics. For complete results see Table 3.

Table 3. Mean and standard deviation of solution scores at given running-time. Scores with the best mean, and those whose means are smaller than the sum of the best mean and its standard deviation appear in boldface. Results for TSP are given in units of 1000.

		Append		Interleave		Combination		Exponential		Fractional	Restart	Scramble	
Problem	Time	Mean	Stdv	Mean	Stdv	Mean	Stdv	Mean	Stdv	Mean	Stdv	Mean	Stdv
SCS	1	3338.6	91.8	3327.1	128.5	**1538.3**	109.0	**1521.6**	70.7	3672.4	23.6	3314.0	159.6
	2.5	1476.7	95.5	1506.7	93.4	**1398.4**	67.7	**1413.5**	44.0	3617.3	105.6	1438.0	68.3
	5	840.5	101.7	1554.7	212.9	**614.8**	55.2	1500.8	115.3	3585.2	96.1	1198.3	119.3
	7.5	705.2	125.8	1202.7	87.9	**598.1**	38.9	1392.1	122.2	3609.7	69.4	1161.2	88.2
	10	699.9	125.0	1116.4	72.3	**596.9**	23.0	1339.9	35.8	1378.2	104.1	1087.8	92.7
	12.5	709.9	50.9	1052.0	125.9	**562.2**	54.1	1265.7	162.5	1132.4	97.0	1050.4	116.7
	15	668.7	85.1	917.6	84.9	**570.7**	45.3	1129.3	118.1	1135.7	121.3	1009.4	106.5
	20	718.4	94.8	996.0	117.8	**562.2**	42.2	1059.4	111.4	1037.9	95.3	940.2	94.2
	25	724.9	77.5	823.3	114.6	**544.0**	42.8	1085.4	126.1	1006.4	143.1	854.2	51.7
	30	663.9	90.9	889.6	100.1	**541.7**	38.1	1085.8	124.4	1024.4	83.5	938.3	71.1
	45	621.6	77.0	868.6	129.9	**515.2**	29.6	1048.7	105.9	875.7	115.7	895.5	78.4
	60	654.0	85.5	839.9	62.8	**505.4**	10.8	946.0	57.1	880.8	90.6	909.9	71.1
	90	660.1	55.2	847.8	101.6	**502.7**	8.1	902.1	103.9	823.5	127.8	839.3	148.2
	120	657.3	66.7	777.2	132.8	**505.4**	10.8	878.4	120.4	844.2	88.2	798.1	119.2
Bandwidth	1	731.3	11.4	739.2	16.0	693.5	7.8	**660.4**	12.8	1967.3	10.4	734.3	11.4
	2.5	643.2	6.6	643.5	13.5	644.2	9.1	**632.2**	11.6	1973.4	13.5	645.2	8.3
	5	624.8	9.9	**622.7**	9.5	633.1	6.8	**615.7**	8.6	790.8	17.7	**619.9**	11.3
	7.5	1974.6	12.8	1428.5	14.1	627.3	9.5	**615.7**	9.2	641.7	11.5	**614.1**	11.7
	10	1973.9	10.6	1181.0	18.5	**620.9**	9.6	**611.5**	8.4	**616.5**	8.9	**615.6**	14.3
	12.5	1968.6	13.9	691.7	17.3	619.8	8.8	734.1	381.0	**606.1**	10.3	**613.7**	8.3
	15	1969.3	11.7	659.6	16.9	**614.8**	8.2	**606.8**	9.5	**606.3**	8.6	**613.4**	13.7
	20	1972.1	11.6	646.1	9.7	615.5	7.2	982.5	559.7	**602.6**	9.3	**610.8**	13.7
	25	1979.0	6.9	737.2	16.2	**613.0**	10.9	**609.4**	14.9	**603.2**	10.9	**604.7**	10.6
	30	1969.7	14.6	737.0	16.0	609.9	5.0	**483.5**	3.7	608.4	7.6	606.0	9.4
	45	1973.6	10.8	678.5	13.5	607.9	4.8	**482.2**	5.6	605.2	7.3	608.4	8.9
	60	1971.9	14.6	635.9	19.7	608.2	3.7	**477.9**	4.5	599.6	9.7	605.7	9.0
	90	1976.0	10.5	729.9	12.7	602.4	7.1	**462.5**	5.4	600.9	9.0	603.6	8.0
	120	1972.4	12.6	656.3	10.9	602.2	5.8	**453.0**	3.4	596.8	13.9	606.2	16.4
MaxCut	1	**3972.0**	0.0	**3972.0**	0.0	**3972.0**	34.7	**3972.0**	0.0	**3972.0**	0.0	**3972.0**	0.0
	2.5	**3972.0**	0.0	**3972.0**	0.0	**3971.7**	31.5	**3972.0**	0.0	**3972.0**	0.0	**3972.0**	0.0
	5	3963.8	15.0	3963.8	15.0	**1401.8**	14.1	1754.4	25.0	**1387.9**	20.9	3969.9	6.0
	7.5	3964.4	12.0	3964.4	12.0	**1354.0**	26.2	1756.8	19.2	1453.6	52.9	3965.9	17.3
	10	1575.4	20.7	1575.4	20.7	**1385.5**	53.1	1751.9	27.5	1455.0	72.5	1467.6	120.0
	12.5	**1321.1**	36.9	**1321.1**	36.9	**1355.2**	26.8	1740.0	21.6	1492.5	222.6	**1288.5**	97.7
	15	**1224.7**	16.2	**1224.7**	16.2	1346.7	18.9	1747.2	29.4	1402.1	61.3	**1227.5**	15.3
	20	1221.4	16.6	1221.4	16.6	1462.9	64.5	1738.4	31.0	1385.2	69.1	**1199.6**	8.8
	25	1167.2	19.7	1167.2	19.7	1452.0	76.6	1733.0	22.6	1512.6	102.9	**1146.9**	6.2
	30	**1150.3**	12.8	**1150.3**	12.8	1441.6	46.0	1753.9	25.3	1450.2	56.3	**1144.9**	10.6
	45	1662.2	27.9	1662.2	27.9	1971.2	12.8	1755.3	23.5	1409.8	51.3	**1140.6**	13.5
	60	1630.0	42.5	1630.0	42.5	1689.1	90.2	1751.5	13.7	1480.5	262.1	**1123.2**	13.8
	90	1281.4	35.7	1281.4	35.7	1276.6	36.0	1746.0	33.5	1394.3	49.0	**1109.5**	11.9
	120	1278.2	51.9	1278.2	51.9	1240.0	22.6	1748.7	24.1	1468.7	219.1	**1104.6**	13.1
VertCover	1	2313.90	20.33	2313.90	20.33	**648.49**	5.56	**648.90**	4.21	2307.37	29.31	2324.30	11.93
	2.5	1123.29	766.20	1123.29	766.20	2259.57	23.77	650.00	7.58	**606.80**	6.94	616.90	4.87
	5	1272.97	817.17	1272.97	817.17	1100.23	760.28	650.70	4.80	633.17	3.71	**604.80**	4.66
	7.5	1592.26	808.24	1592.26	808.24	**582.28**	15.23	649.20	4.83	602.50	5.30	599.10	5.34
	10	617.17	5.91	617.17	5.91	**571.32**	34.84	653.00	8.41	603.20	4.69	632.30	11.40
	12.5	615.47	5.50	615.47	5.50	**550.75**	15.64	647.70	7.54	600.50	3.69	626.80	9.10
	15	613.43	3.43	613.43	3.43	**554.95**	17.59	646.80	6.00	601.00	3.03	621.40	5.06
	20	614.92	3.39	614.92	3.39	**541.86**	13.98	651.50	5.70	599.00	4.22	625.60	5.39
	25	614.95	5.38	614.95	5.38	**546.13**	13.80	650.60	3.85	597.50	3.64	621.10	6.73
	30	614.89	5.95	614.89	5.95	**528.80**	12.39	649.40	5.22	597.50	5.43	615.80	4.31
	45	604.80	4.02	604.80	4.02	**531.68**	13.42	604.50	5.95	584.40	3.04	609.60	2.33
	60	602.90	1.92	602.90	1.92	**521.49**	10.82	630.40	3.88	581.20	2.96	606.10	3.83
	90	596.90	4.04	596.90	4.04	**521.89**	9.54	594.90	4.89	579.30	1.79	598.40	3.69
	120	594.20	2.75	594.20	2.75	**523.05**	9.03	591.80	16.58	580.90	2.66	598.40	4.76
TSP	1	**1102.0**	42.8	**1100.7**	38.5	**1133.4**	39.1	**1074.8**	32.2	5085.0	30.6	**1093.5**	33.4
	2.5	**825.8**	31.9	**825.4**	34.6	971.0	36.2	908.0	29.5	1339.3	39.9	**823.3**	19.1
	5	**680.6**	42.8	1033.9	29.9	933.7	52.3	883.0	19.2	835.3	27.9	**676.4**	30.4
	7.5	**570.5**	27.8	1907.7	139.2	885.3	16.4	725.8	23.3	781.5	46.8	627.1	23.2
	10	**539.7**	32.0	1747.8	77.3	633.3	189.8	769.0	64.1	716.7	29.0	635.2	62.7
	12.5	476.3	12.7	936.3	60.5	**448.1**	13.6	697.3	29.6	698.3	22.8	667.8	67.3
	15	**459.3**	18.4	636.9	24.0	**472.9**	129.0	672.8	39.7	667.7	26.8	676.4	73.2
	20	**404.4**	16.1	1445.0	167.3	**411.5**	12.5	614.1	33.4	617.3	31.5	693.4	28.1
	25	**397.8**	4.9	1295.6	98.6	406.1	13.6	591.3	27.5	578.8	14.3	658.9	17.7
	30	**390.8**	10.3	633.2	36.4	**397.4**	18.0	573.4	11.2	544.0	16.6	659.8	27.7
	45	393.9	9.5	1159.4	169.0	**378.7**	5.6	524.6	17.5	509.3	14.2	630.4	23.5
	60	390.4	5.5	2457.0	661.2	**381.0**	6.0	488.2	14.2	529.0	30.8	615.5	19.6
	90	393.6	5.0	1955.5	276.0	**375.0**	5.0	472.2	14.4	487.1	21.8	591.2	22.0
	120	393.6	6.8	622.4	51.7	**374.5**	3.3	449.5	15.2	475.9	13.1	569.5	23.5
MaxSat	1	**877.9**	378.8	**877.9**	378.8	**912.3**	320.1	3119.2	747.6	**1074.9**	285.8	**1092.3**	473.3
	2.5	**674.5**	215.6	**674.5**	215.6	**779.3**	169.9	3668.5	488.8	**685.9**	265.8	**686.2**	285.2
	5	**473.2**	282.3	**473.2**	282.3	636.4	237.7	755.8	193.7	**459.5**	208.7	**431.5**	198.9
	7.5	**530.4**	179.0	**530.4**	179.0	**601.9**	270.3	773.2	318.7	**428.6**	199.1	**539.2**	230.7
	10	**443.0**	159.0	**443.0**	159.0	**359.2**	114.7	642.0	188.9	**509.5**	234.9	531.1	238.5
	12.5	**510.9**	184.8	**510.9**	184.8	**493.5**	212.2	**559.4**	179.4	**526.2**	144.9	**520.1**	183.9
	15	**448.9**	214.5	**448.9**	214.5	**465.5**	159.1	**543.5**	233.2	**429.9**	213.5	**458.8**	179.7
	20	**402.4**	198.6	**402.4**	198.6	**503.4**	211.6	**401.3**	127.9	**486.0**	354.2	**426.8**	169.3
	25	685.3	424.3	685.3	424.3	**398.9**	210.6	474.1	230.4	**325.6**	158.1	510.0	290.3
	30	644.9	230.1	644.9	230.1	**374.1**	142.8	**526.1**	169.3	**451.5**	147.8	656.3	364.8
	45	796.3	390.2	796.3	390.2	**262.1**	154.5	444.9	214.2	**331.1**	166.7	889.3	299.2
	60	719.0	385.2	719.0	385.2	**206.5**	101.9	477.7	157.9	321.4	102.7	538.6	265.7
	90	700.3	147.0	700.3	147.0	**255.8**	97.6	375.9	108.0	**161.7**	84.2	698.8	162.8
	120	560.0	284.0	560.0	284.0	**236.2**	78.7	335.1	82.0	**179.4**	123.3	635.8	321.3

References

[1] E. Aarts and J.-K. Lenstra. *Local Search in Combinatorial Optimization*. Wiley-Interscience, Chichester, England, 1997.

[2] D. Corne, M. Dorigo, and F. Glover. *New Ideas in Optimization*. McGraw-Hill, London, 1999.

[3] T. Csendes and D. Ratz. Subdivision direction selection in interval methods for global optimization. *SIAM Journal on Numerical Analysis*, 34(3):922–938, 1997.

[4] F. Glover. Tabu search– part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[5] J. H. Holland. *Adaptation in natural artificial systems*. University of Michigan Press, Ann Arbor, 1975.

[6] L. Ingber. Adaptive simulated annealing (asa): Lessons learned. *Control and Cybernetics*, 25(1):33–54, 1996.

[7] M. Jelasity. Towards automatic domain knowledge extraction for evolutionary heuristics. In *Parallel Problem Solving from Nature - PPSN VI, 6th International Conference*, volume 1917 of *Lecture Notes in Computer Science*, pages 755–764, Paris, France, Sept. 2000. Springer.

[8] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part 1, graph partitioning. *Operations Research*, 37(6):865–892, 1989.

[9] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part 2, graph coloring and number partitioning. *Operations Research*, 39(3):878–406, 1991.

[10] S. Kirpatrick, C. Gelatt, Jr., and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[11] A. V. Kuntsevich. Fortran-77 and fortran-90 global optimization toolbox: User's guide. Technical Report A-8010, Institut fur Mathematic, Karl Franzens Universitat, Graz, Austria, 1995.

[12] L. Lukšan, M. Tůma, M. Šiška, J. Vlček, and N. Ramešová. Interactive system for universal functional optimization (ufo). Technical Report 826, Institute of computer science, Academy of sciences of the Czech Republic, Prague, Czech Republic, 2000.

[13] M. Mongeau, H. Karsenty, V. Rouz, and J.-B. Hiriart-Urruty. Comparison of public-domain software for black box global optimization. *Optimization Methods and Software*, 13(3):203–226, 2000.

[14] V. Phan, S. Skiena, and P. Sumazin. A model for analyzing black box optimization. in preparation, 2001.

[15] V. Phan, P. Sumazin, and S. Skiena. Discropt web page. http://www.cs.sunysb.edu/~discropt.

[16] G. Reinelt. *TSPLIB*. University of Heidelberg, www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95.

[17] G. Reinelt. TSPLIB— A traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[18] M. Resende. *Max-Satisfiability Data*. Information Sciences Research Center, AT&T, www.research.att.com/~mgcr.

[19] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[20] M. Yagiura and T. Ibaraki. On metaheuristic algorithms for combinatorial optimization problems. *The Transactions of the Institute of Electronics, Information and Communication Engineers*, J83-D-1(1):3–25, 200.

[21] Q. Zheng and D. Zhuang. Integral global optimization: Algorithms, implementations and numerical tests. *Journal of Global Optimization*, 7(4):421–454, 1995.