Coloring Graphs With a General Heuristic Search Engine

Vinhthuy Phan Steven Skiena
State University of New York at Stony Brook, NY 11794
{phan, 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. We have been investigating 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 näive 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 [2] of six existing black-box optimization packages describes generally disappointing results.

Our system, *Discropt* [4], is intended as a general-purpose platform to experiment with heuristic optimization. Characteristics of *Discropt* include:

- *Ease of Use* – The system is designed to provide general data structures, search heuristics, and visualization tools to speed the development of algorithm implementation and minimize the need for performance tuning.
- *Flexibility* – The system is designed to be applicable to a wide range of optimization problems. In [4], we apply our system to six significantly different optimization problems: traveling saleperson, shortest common superstring, graph bandwidth, maxcut, maximum satisfiability, and vertex cover.
- *Awareness of Time* – The most effective heuristic to use for a particular problem is clearly is a function of the size of the instance, the power of the computer, and the time alloted for computation. Our system actively manages the alloted time to return the best answer possible in the given window.

In this paper, we report on our experiences building implementations of vertex coloring and graph bandwidth coloring with *Discropt*. We use the same problem-independent heuristic implementations reported in [4] without fundamental changes or problem-specific tuning. Adapting our heuristics to graph coloring problems provides a good test of the flexibility of our system for two reasons. First, graph coloring is well known as a very challenging problem for local search heuristics such as employed within our system. Second, the fundamental solution representation for graph colorings (set partitions) had not been implemented in our system as of [4]. In this paper, we

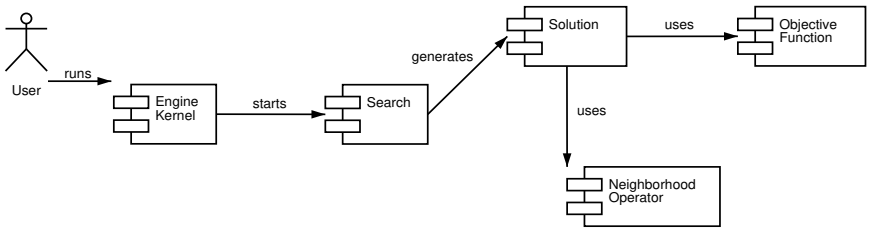

Figure 1: Basic system flow

test the performance of optimization routines developed for permutation and subset problems on a completely different class of object.

Our organization is as follows. In Section 2, we briefly report on the architecture of our system. In Section 3, we discuss an important issue in heuristics for graph coloring, namely the trade-off between quality (number of colors) and validity (number of edge-violations) in partial solutions. We present an general method for combining the two objective functions such that (1) requires no problem-specific information from the programmer, (2) enables us to optimize over the full solution space instead of restricting movement to correct examples, and (3) guarantees termination with valid solutions for all problems in a large and well-defined class. In Sections 4 and 5 we report on our experimental results for vertex and bandwidth coloring, respectively.

We emphasize that the goal of *Discropt* is significantly reducing the development time to build *reasonable* optimization routines for a widely varying problems. Our generality naturally exacts a tax on performance – we cannot expect our general optimization routines to compete with carefully crafted, problem-specific algorithms. One of the goals of our participating in this workshop is to assess our results against the highly-engineered problem-specific programs that we anticipate will represent most of the other submissions.

2 System Architecture

The architecture of *Discropt* has evolved somewhat since the reported in [4]. We outline the four central components of the system, a more complete system description is available online at [3].

- *Solution objects* are defined by (1) a combinatorial structure such as a permutation, subset, or set partition; (2) an objective cost over this structure; (3) a mechanism to disallow further generation of solutions when time is up. The combinatorial structure is separated from the solution implementation so that new types can be easily added. Incrementally constructive heuristics are supported through *partial solutions*, which can be *extended* by inserting an element into any position of the combinatorial structure.

- *Neighborhood objects* implement the function of a neighborhood operator. Its separation from the other components make it easy to experiment with different types of neighborhood operators. Currently, we use a *swap* operation that exchanges two random elements of a combinatorial structure. The specifics vary from one structure to another. Together, *solution* and *neighborhood* capture the notion of a search space.

- *Search objects* define an exploration through the search space. The system currently includes a few basic heuristics, namely *hill climbing*, *simulated annealing*, and *incremental greedy*. In this paper, we will consider two different flavors of simulated annealing: (1) the basic

annealing algorithm with a time-sensitive exponentially cooled temperature schedule, and (2) simulated annealing with a restart mechanism that resets the temperature after the search reaches a local optimum. The incremental greedy heuristic extends a solution by picking the best partial solution from a pool of candidates. It is made time-sensitive by controlling the number of candidate partial solutions at each incremental step.

- *Objective function objects* are defined by users to compute the objective value of a solution in the search space. Optionally, a faster *delta* function can be defined to speed up local search.

  Although certain problems such as traveling salesman on complete graphs only have valid solutions, problems such as graph coloring have two dimensions: validity and cost. The user has the option of defining *validity* and *delta-validity* functions that measure the relative validity of a given solution.

These components together define the performance of a search. In [4], we demonstrated that simple combinations of the heuristics themselves can outperform each individual one.

3 Combining Cost and Validity Objective Functions

Problems whose search space contains invalid solutions (including both vertex coloring and bandwidth coloring) pose a special challenge for heuristic optimization. Considering invalid solutions seems essential to avoid getting trapped in local minima, and yet we need to ensure that the final solution is valid.

One approach is to define an ad-hoc objective function that *internally* combines the cost and validity. To ensure that the search terminate on a valid solution, the combined function must have the property specified by Johnson et. al., [1], that a locally optimal solution with respect to the objective function must always be valid.

The design of a correct combined function requires deeper understanding of the specific problem than we assume users of *Discropt* necessarily possess. Separate definitions of cost and validity are inherently more intuitive and natural than any combination of them. For example, the validity of a coloring can be measured by counting the number of edges whose vertices have the same color. We believe that maintaining this separation allows the user refine his understanding of the problem and develop with better definitions of validity as the project progresses. Instead, we propose a general approach to combining cost and validity functions which guarantees ultimate solution validity for a large class of problems while requiring no specific action on the part of the user.

We say that a problem P is *validity-monotonic* under a given neighborhood operator and validity function v if every solution that is not optimally valid with respect v has a more valid neighboring solution. Graph coloring under the swap neighborhood clearly has this property, since we can always swap any offending vertex to a new part of the partition. This increases the number of colors by one, but improves validity. There is always a path from any set partition to a valid solution. Validity-monotonic problems include vertex/set cover, independent set, and maximum clique under the swap neighborhood with natural validity functions.

Proposition 1 *Let P be a* validity-monotonic *under a given neighborhood operator with validity function v. Consider the linear objective function f, where*

$$f(s, c, v) = c(s) + k \cdot v(s)$$

and c and v are cost and validity functions respectively evaluated on candidate solution s, and k is a dynamically updated variable

$$k > \max \frac{|c(s_i) - c(t_i)|}{|v(s_i) - v(t_i)|} \tag{1}$$

for all s_i and t_i which are neighbors and are visited in the ith step by the search process in its exploration of the search space. Then, any solution s that is locally optimal with respect to f is globally optimal with respect to validity.

Proof: Suppose s is locally optimal with respect to f, and that its validity is not globally optimal. Then there is a neighboring solution t such that $f(s) = c(s) + k \cdot v(s) \leq c(t) + k \cdot v(t) = f(t)$ and $v(s) > v(t)$. This implies that $0 < k \cdot (v(s) - v(t)) \leq (c(t) - c(s))$ or $0 < k \leq \frac{c(t)-c(s)}{v(s)-v(t)}$, which is a contradiction to the definition of k. ∎

Such a combined function f allows the search to proceed through the search space improving cost, while always preferring valid solutions to invalid ones. In a more sophisticated setting, k may be set to converge to an upper bound of the dynamic maximum so that $k \rightarrow \epsilon + \max \frac{|c(s)-c(t)|}{|v(s)-v(t)|}$, after a certain number of steps. In other words, k can be made sensitive to search time so as validity is less emphasized at the beginning, more at later stages, and is optimally achieved at the end.

In our experiments where instances are run in 2, 20, and 100 seconds, validity is achieved in most cases except in a few instances of 2-second running times when the heuristics do not have sufficient time to explore the whole neighborhood of their local optima.

4 Experimental Results: Graph Coloring

The experiments are run on a low-end PC of 1Ghz Athlon K7 AMD processor, 768MB of RAM, and under the RedHat 7.2 operating system. Data are obtained from mat.gsia.cmu.edu/COLORING02/. We do not run 5 of them as the results are not meaningful for 100 second running times. In the interest of space, we do not show instances for which performance does not improve much between 2 and 100 seconds. The rest of the data are shown in table 1 and 2.

The combinatorial structure of a graph coloring is a set partition on the set of vertices, with each part representing a distinct color. A neighbor of a given coloring contains exactly one vertex with a different color. Our neighborhood operator selects a random vertex and assigns to it a random color $r, 1 \leq r \leq p+1$, where p is the number of existing colors. If a vertex with color of cardinality 1 is assigned a new color, the number of colors get reduced by 1. If $r = p + 1$ the number of colors gets increased by 1. We consider graph coloring under three different objective functions:

1. An external combination (described in Proposition 1) of simpler cost and validity functions, where the cost is the number of colors, and validity is the number of edges whose vertices have the same colors.

2. An external combination of the cost and validity components of [1] and equation 2 below, automatically combined using Proposition 1. Cost is $-\sum c_i^2$ and validity is $\sum 2 \cdot c_i \cdot e_i$.

3. An internally combined cost function, studied by Johnson et. al. [1]:

$$-\sum c_i^2 + \sum 2 \cdot c_i \cdot e_i \tag{2}$$

   where c_i denotes the number of vertices of color i, and e_i is the number of edges whose vertices have color i. As proven in [1], any locally optimal solution with respect to this function must be a valid solution.

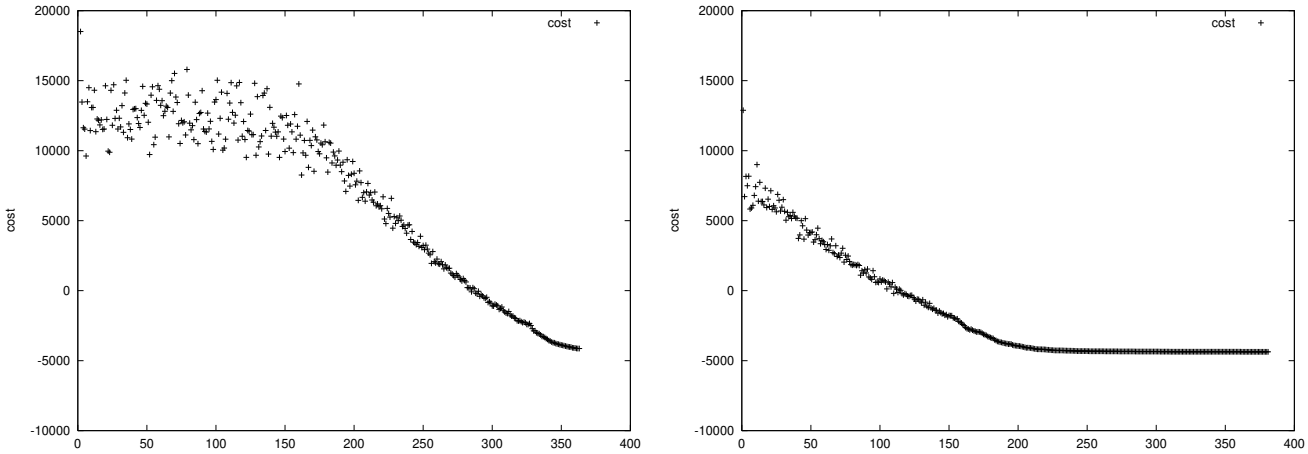

Figure 2: Cost (see equation 2) vs. running time (in seconds): simulated annealing on r500.5 before (left) and after (right) readjustment of temperature schedule.

The results in table 1 show the number of colors used for each of the three versions at 3 different running times. The performance gets expectedly better at longer running times. Secondly, the third version of objective function, equation 2, performs better than the simple objective function (first version) despite the first being much simpler to implement. The second version performs slightly worse than the third.

Table 2 compares four heuristics at 100-second running time. The results here are quite different from those reported in [4] for six other optimization problems. Greedy heuristic does not do as well as the others. Hill-climb performs surprisingly well at 100-second running time.

Running times:	120	240	480	960
hill climb	69	67	69	66
simulated annealing	146	66	66	72
simulated annealing with restart	149	141	102	64
incremental greedy	217	205	141	99
simulated annealing w. readjusted schedule	65	63	63	61
simulated annealing w. restart, readjusted schedule	69	65	63	62

The above table shows comparative performance of heuristics for the benchmarking instance r500.5. The results are medians of 5 different runs. Hill climbing does not perform appreciably better with longer running times, which means it has reached its potential at 120 seconds. The next three heuristics do rather badly by comparison, but do improve with longer times. We suspect this is due to a more difficult landscape of this instance. Shown in figure 2, simulated annealing takes much time to explore the landscape before converging. It seems the landscape requires very precise automatic time-sensitive adjustments. Adjusting the annealing schedule to be more aggressive results in better performance. This is the cost of automating a time-sensitive schedule.

5 Experimental Results: Bandwidth Coloring

The combinatorial structure of bandwidth coloring is a different from classical unordered set partitions, since it is possible to have colors i and j with $i < j$ and no color in between. Our neighborhood

Instance	2 sec			20 sec			100 sec			who wins at 100 sec
	*	**	***	*	**	***	*	**	***	
1-FullIns_5	30	21	21	28	11	10	26	9	7	***
1-Insertions_5	21	12	12	20	7	6	18	6	7	**
1-Insertions_6	42	49	41	40	21	22	32	13	15	**
2-FullIns_4	24	15	13	20	8	8	20	8	8	
2-FullIns_5	49	71	43	55	37	37	46	22	23	**
2-Insertions_4	17	8	7	16	5	6	12	5	5	
2-Insertions_5	36	44	37	33	21	16	35	10	11	**
3-FullIns_4	30	29	25	30	13	14	30	10	11	**
3-FullIns_5	75	62	71	75	101	69	73	63	59	***
3-Insertions_4	25	17	15	20	7	7	20	6	5	***
3-Insertions_5	57	65	57	55	58	47	61	31	29	***
4-FullIns_4	43	51	46	39	25	24	39	17	19	**
4-Insertions_4	30	29	23	26	12	12	28	7	7	
5-FullIns_3	20	11	10	16	9	9	17	9	8	***
5-FullIns_4	53	55	54	48	45	42	45	27	27	
abb313GPIA	77	67	68	78	91	75	78	49	48	***
anna	15	13	15	16	12	12	14	13	11	***
ash608GPIA	47	67	46	47	42	38	50	22	23	**
ash958GPIA	63	87	72	67	70	62	60	45	41	***
DSJC1000.1	75	69	60	71	64	61	75	42	41	***
DSJC125.1	17	8	9	15	9	8	14	8	7	***
DSJC125.5	34	25	23	32	24	25	33	23	24	**
DSJC125.9	63	53	55	61	55	52	60	53	52	***
DSJC250.9	113	100	100	115	95	96	112	89	90	**
DSJC500.1	46	54	44	48	25	24	43	19	21	**
DSJR500.1c	123	91	92	162	116	117	159	99	105	**
DSJR500.5	134	95	86	168	166	164	167	167	168	
fpsol2.i.2	44	52	40	46	41	38	44	34	35	**
fpsol2.i.3	48	47	46	43	39	38	43	34	33	***
homer	32	37	29	31	20	22	34	17	18	**
inithx.i.2	56	57	57	54	48	45	52	39	37	***
inithx.i.3	55	57	47	52	44	49	51	41	39	***
le450_15a	45	52	38	42	26	26	41	23	23	
le450_15b	44	47	38	40	26	27	37	22	23	**
le450_15c	58	58	56	56	36	35	52	30	32	**
le450_15d	55	57	57	53	36	34	52	31	31	
le450_25a	45	49	43	43	33	34	40	30	30	
le450_25b	43	46	46	42	33	32	40	31	30	***
le450_25c	55	66	54	57	40	39	54	36	36	
le450_25d	61	63	60	56	42	43	59	38	37	***
le450_5a	37	35	32	35	16	16	33	13	14	**
le450_5b	39	36	31	36	18	16	33	13	13	
le450_5c	48	44	40	42	22	21	41	18	17	***
le450_5d	42	45	41	45	21	21	41	16	16	
miles1000	47	47	48	44	46	46	46	45	46	**
miles1500	76	75	75	74	76	75	73	74	73	
miles250	16	9	10	12	10	9	13	10	9	***
miles750	36	34	38	34	35	35	33	35	33	
mulsol.i.2	41	36	35	38	32	32	38	31	31	
mulsol.i.4	41	36	36	36	32	32	36	32	31	***
mulsol.i.5	39	34	35	37	32	32	37	31	31	
qg.order30	67	71	59	69	50	47	58	35	36	**
qg.order40	82	68	67	84	102	85	87	61	61	
queen16_16	39	27	26	37	22	24	35	22	22	
queen8_12	21	16	15	19	14	16	20	15	14	***
school1	68	67	69	72	44	39	66	36	38	**
school1_nsh	62	61	56	62	40	41	59	36	35	***
will199GPIA	45	50	44	37	24	24	38	15	16	**
zeroin.i.2	41	33	33	36	32	32	35	31	31	
zeroin.i.3	37	33	33	36	32	33	34	31	30	***

Table 1: Comparison at three different running times of hill-climb using three different objective functions: * is the first, ** the second, and *** the third objective function defined in section 4.

Instance	Hill-climb	Sim. Anneal.	Sim. Anneal. w restart	Greedy	winner
1-FullIns_5	7	8	9	96	hc
1-Insertions_5	7	6	6	73	
1-Insertions_6	15	29	18	92	hc
2-FullIns_4	8	7	8	80	sim
2-FullIns_5	23	36	30	37	hc
2-Insertions_4	5	6	6	5	
2-Insertions_5	11	19	14	142	hc
3-FullIns_4	11	13	12	155	hc
3-FullIns_5	59	142	121	*	hc
3-Insertions_4	5	5	6	145	
3-Insertions_5	29	59	39	*	hc
4-FullIns_4	19	27	22	70	hc
4-Insertions_4	7	11	8	209	hc
5-FullIns_3	8	8	9	8	
5-FullIns_4	27	66	37	*	hc
abb313GPIA	48	145	72	*	hc
anna	11	13	11	11	
ash608GPIA	23	52	29	*	hc
ash958GPIA	41	84	61	*	hc
DSJC1000.1	41	65	61	*	hc
DSJC125.1	7	7	7	8	
DSJC125.5	24	21	21	26	
DSJC125.9	52	46	52	58	sim
DSJC250.9	90	92	79	173	sim-r
DSJC500.1	21	25	20	26	sim-r
DSJR500.1c	105	221	234	*	hc
DSJR500.5	168	155	180	354	sim
fpsol2.i.2	35	47	35	66	
fpsol2.i.3	33	44	35	91	hc
homer	18	23	18	221	
inithx.i.2	37	64	46	81	hc
inithx.i.3	39	54	44	94	hc
le450_15a	23	35	25	48	hc
le450_15b	23	25	25	32	hc
le450_15c	32	32	32	36	
le450_15d	31	49	33	37	hc
le450_25a	30	34	32	51	hc
le450_25b	30	30	30	70	
le450_25c	36	43	39	67	hc
le450_25d	37	43	60	43	hc
le450_5a	14	16	14	96	
le450_5b	13	15	16	117	hc
le450_5c	17	22	18	22	hc
le450_5d	16	18	17	21	hc
miles1000	46	44	44	44	
miles1500	73	73	73	74	
miles250	9	9	9	9	
miles750	33	32	33	34	sim
mulsol.i.2	31	33	32	31	
mulsol.i.4	31	31	32	50	
mulsol.i.5	31	33	32	71	hc
qg.order30	36	81	42	50	hc
qg.order40	61	117	256	*	hc
queen16_16	22	22	21	24	sim-r
queen8_12	14	14	13	15	sim-r
school1	38	77	37	87	sim-r
school1_nsh	35	42	33	42	sim-r
will199GPIA	16	26	21	75	hc
zeroin.i.2	31	30	31	83	sim
zeroin.i.3	30	33	31	66	hc

Table 2: Comparison of four heuristics at 100-second running time. The * symbol means the time-sensitive component of the heuristic finds that it does not have enough time to run meaningfully.

operator is defined similarly to that of vertex coloring.

We use the following definitions of cost and validity. Cost is defined as the number of colors used, while the validity $v(G)$ is defined as $v(G) = \sum_{(u,v)\in E} \max\{0, d_{uv} - |c(u) - c(v)|\}$, where d_{uv} is the bandwidth limit of each edge (u, v) and $c(u)$ is the color at vertex u.

This definition together with the neighborhood operator favor local search heuristics that defines a smooth transition of validity over incremental greedy heuristic that requires validity for each partial solution at each incremental step. The results show the performance of the three heuristics. Simulated annealing with restarts and hill-climb are slightly ahead of of simulated annealing.

Instance	Hill-climb	Sim. Anneal.	Sim. Anneal. w restart	winner (100 sec)
GEOM100a	86	85	84	sim-r
GEOM100b	92	89	87	sim-r
GEOM100	56	56	55	sim-r
GEOM110a	91	92	88	sim-r
GEOM110b	97	103	97	
GEOM110	62	59	59	
GEOM120a	101	103	107	hc
GEOM120b	103	111	115	hc
GEOM120	68	69	67	sim-r
GEOM20a	20	22	20	
GEOM20b	13	15	14	hc
GEOM20	20	22	22	hc
GEOM30a	27	28	29	hc
GEOM30b	26	27	27	hc
GEOM30	27	29	29	hc
GEOM40a	38	40	40	hc
GEOM40b	36	36	36	
GEOM40	27	29	29	hc
GEOM50a	54	55	56	hc
GEOM50b	42	40	41	sim
GEOM50	30	29	30	sim
GEOM60a	56	55	54	sim-r
GEOM60b	52	47	50	sim
GEOM60	34	35	36	hc
GEOM70a	64	67	65	hc
GEOM70b	58	56	54	sim-r
GEOM70	40	43	41	hc
GEOM80a	69	69	70	
GEOM80b	70	71	70	
GEOM80	46	44	49	sim
GEOM90a	75	75	74	sim-r
GEOM90b	83	86	87	hc
GEOM90	50	48	50	sim

References

[1] D. Johnson, L. M. C. Aragon, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 39:378–406, 1991.

[2] 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.

[3] V. Phan, S. Skiena, and P. Sumazin. *A Discrete Optimization Engine.* SUNY at Stony Brook, www.cs.sunysb.edu/∼discropt/manual.

[4] V. Phan, S. Skiena, and P. Sumazin. A time-sensitive system for black-box optimization. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX02)*, 2002.