Supplementary Material *for* What Works Best When? A Systematic Evaluation of Heuristics for Max-Cut and QUBO

Iain Dunning, Swati Gupta, John Silberholz

1 Standard and Expanded Instance Libraries

We consider the *standard instance library* for Max-Cut and QUBO to be the set of instances used for computational experiments by at least four papers out of the 95 papers identified in Section 3 of the main paper; the six sources that make up this library are described in Table 1. All QUBO instances were converted to Max-Cut instances using the reduction from Hammer (1965), and node counts and densities in Table 1 are for the resulting Max-Cut instances.

			Description		
Problem	Name	Count	Nodes	Density	Reference
Max-Cut	G-set	71	800–20,000	0.0%–6.0%	Helmberg and Rendl (2000)
	Spin Glass	30	125–2,744	0.2%–4.8%	Burer et al. (2002)
	Torus	4	512–3,375	0.2%–1.2%	7^{th} DIMACS Implementation Challenge
QUBO	GKA	45	21–501	8.0%–99.0%	Glover et al. (1998)
	Beasley	60	51–2,501	9.9%–14.7%	Beasley (1998)
	P3-7	21	3,001–7,001	49.8%–99.5%	Palubeckis (2006)

Table 1: Sources of instances considered to be the standard instance library for Max-Cut and QUBO.

The expanded instance library includes the standard library but also draws from a wide collection of additional sources, including ORLib (Beasley 1990), SteinLib (Koch et al. 2001), the 2nd, 7th, and 11th DIMACS implementation challenges, TSPLib (Reinelt 1991), and additional instances from the Biq Mac library that are not used extensively in testing Max-Cut and QUBO heuristics (Wiegele 2007). We also generated a number of random graphs using Culberson random graph generators (Culberson et al. 1995), the Python NetworkX library (Hagberg et al. 2008), and the machine-independent random graph generator `rudy` (Rinaldi 1996). Many different types of graphs were constructed, including Erdös-Rényi graphs (Erdös and Rényi 1960), Barabási-Albert graphs (Barabási and Albert 1999), Watts-Strogatz graphs (Watts and Strogatz 1998), and their variants. To add another dimension to our test instances, we considered edge weights sampled from 65 different probability distributions including uniform, triangular, beta, and exponential. We provide the full set of parameters for our database in the instance library at `https://github.com/MQLib`.

2 Graph Metrics

We calculated 58 metrics describing each instance in our library. Ten of these metrics are "global" metrics that consider the graph as a whole, and the remainder are "local" metrics that are summary statistics of various node- and edge-specific quantities. Figure 1 shows the complete list of metrics and how well they are covered by our expanded instance library versus the standard library. Each of the metrics in the figure is referred to by a short name, which we provide in the following descriptions.

Global metrics

The first four metrics are simple to calculate: the logarithm[1] of the number of nodes and edges in the graph (**log_n** and **log_m**, respectively), the proportion of edges with a positive weight (**percent_pos**), and a zero-one indicator for whether the graph is disconnected or not (**disconnected**). The next three metrics are all more complex measures of structure: the approximate chromatic number of the graph as computed by the Welsh-Powell heuristic (Welsh and Powell 1967), normalized by the number of nodes (**chromatic**); the degree assortativity of the graph, a measure of correlation in the degree of linked nodes[2] (**assortativity**); and the approximate size of the maximum independent set of the graph calculated by a fast heuristic,[3] normalized by the number of nodes (**mis**). The final three metrics are all calculated from the weighted graph Laplacian matrix: the logarithm of the first and second largest eigenvalues normalized by the average node degree (**log_norm_ev1** and **log_norm_ev2**, respectively) and the logarithm of the ratio of the two largest eigenvalues (**log_ev_ratio**).

Local metrics

Six different local quantities were calculated, and for each quantity eight statistical measures were calculated to give a total of 48 local metrics. The first quantity is simply the weight of each edge, normalized by the maximum absolute value across all the edge weights (**weight**). Next, we computed the average degree connectivity (**avg_deg_conn**), where the degree connectivity of degree k is the average degree of the neighbors of nodes with degree k. If no nodes have a particular degree value that degree is ignored in the computation. The last four local quantities were specific to nodes and their neighbors:

- the degree of each node (**deg**), normalized by the maximum possible degree.
- the local clustering coefficient of a random subset of size $3\lceil \ln(n) \rceil$ out of the n nodes (**clust**), computed as the number of triangles a node forms with its neighbors divided by the total number of possible triangles. Nodes with fewer than two neighbors are not included.
- the average degree of the neighbors of each node, (**avg_neighbor_deg**), normalized by the maximum possible degree. Nodes with no neighbors are not included.

[1]It is a standard practice to take the logarithm for some features with highly varied values to reduce the effect of outliers.

[2]Degree assortativity is computed as $\frac{\sum_{(i,j)\in E} 2(d(i)-\bar{d})(d(j)-\bar{d})}{\sum_{(i,j)\in E} (d(i)-\bar{d})^2+(d(j)-\bar{d})^2}$, where $\bar{d} = \sum_{(i,j)\in E} \frac{d(i)+d(j)}{2|E|}$.

[3]We use a greedy heuristic for finding an approximate maximum independent set by at each step choosing a vertex with the minimum degree and removing its neighbors.

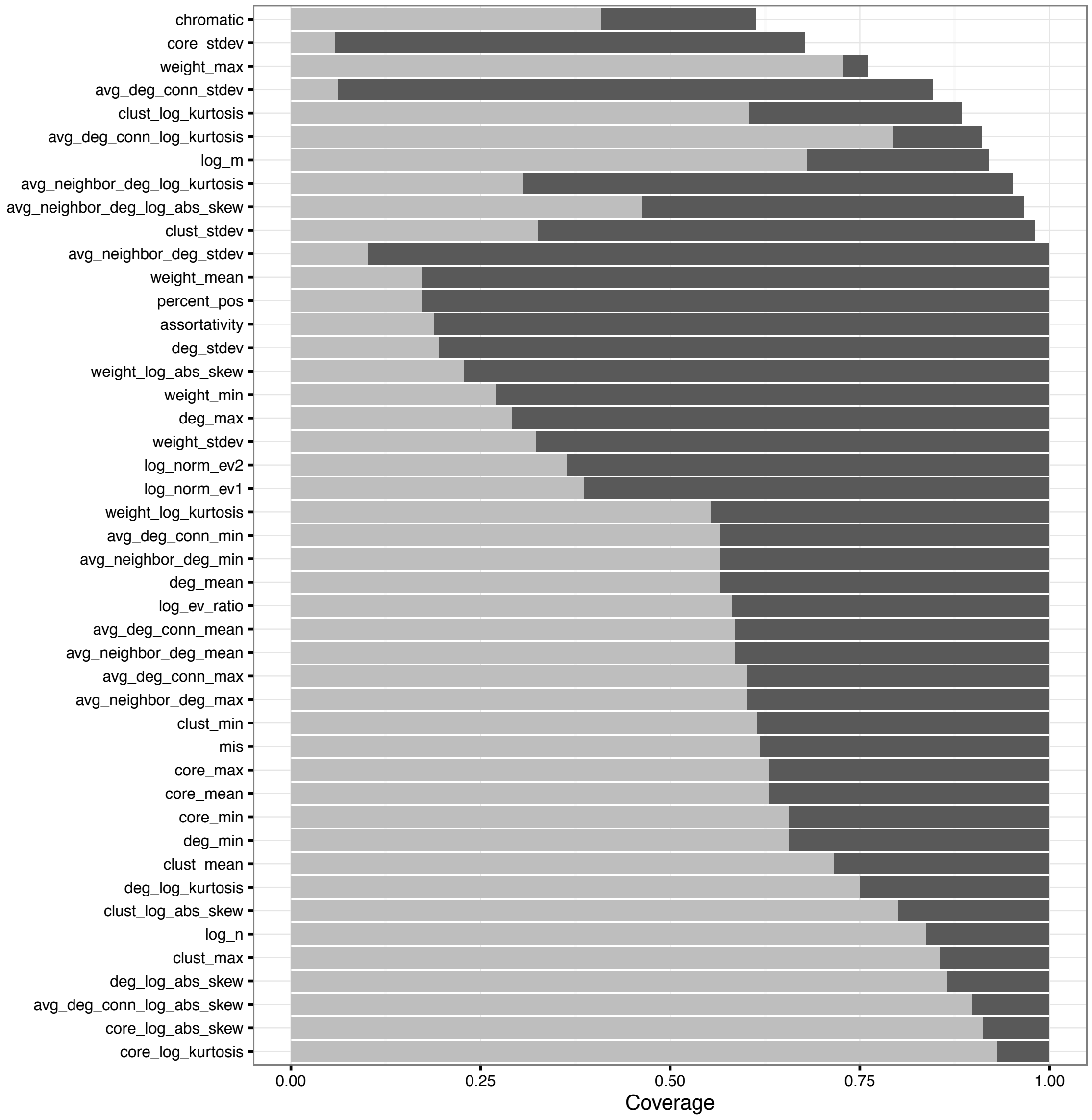


Figure 1: Coverage of each metric by instances with 500 or more nodes in the standard instance library (gray) versus the expanded instance library (gray and black together). Binary metrics are not included.

- the core number for each node in the graph (**core**), normalized by the the number of nodes minus one.[4]

The first four statistical measures are simply the minimum (**_min**), maximum (**_max**), mean (**_mean**), and standard deviation (**_stdev**) of each of the vectors of local quantities. We also considered the skewness of the quantities using the logarithm of one plus the absolute value of skewness (**_log_abs_skew**) and a zero-one indicator if the skewness was non-negative (**_skew_positive**). Further, we captured the logarithm of four plus the excess kurtosis of each vector of local quantities (**_log_kurtosis**). Finally we included a zero-one indicator to capture whether all values for a quantity were the same (**_const**).

3 Identification and Implementation of Heuristics

As described in Section 1 of the main paper, one of the steps of our process is to identify and implement heuristics described in the literature. Here we elaborate on the description in Section 3 of the main paper of our methodology for identifying heuristics in a systematic way, our approach to implementing them, the modifications made to include some heuristics, and the memory scaling of the heuristics.

Selection of heuristics

We searched all English-language publications indexed in the Compendex and Inspec databases published in 2013 or earlier with the query

$$\text{maxcut} \vee \text{max-cut} \vee \text{“max cut”} \vee (\text{quadratic} \wedge \text{binary} \wedge \text{unconstrained})$$

which identified an initial set of 810 results. For a paper to be considered relevant it had to satisfy the following criteria:

1. The paper must describe at least one new heuristic for Max-Cut or QUBO.
2. The paper must have computational experiments for the new heuristic(s) it describes.
3. Heuristics must contain an element of randomness. For example, a deterministic greedy heuristic to construct a solution would not be accepted. We added this criterion as deterministic heuristics do not support arbitrary running times, making head-to-head comparisons with a pre-determined runtime limit challenging.
4. Heuristics must not rely on external libraries or rely on complex subroutines such as a semidefinite program or mixed-integer linear program solver.

Table 2 lists the final set of 19 papers selected and the 37 heuristics they describe, including their asymptotic memory consumption in terms of the number of nodes/variables n, the number of edges/non-zero matrix elements m and various heuristic-specific parameters. The asymptotic memory consumption displayed in Table 2 assumes only the most recent best-known solution is

[4]A k-core of the graph is a maximal subgraph in which all nodes in the subgraph have degree k or more. The maximum k such that the graph has a k-core is called its core number. The core number for a node is the maximum k such that the node lies in a k-core.

stored for each heuristic. Our software library also gives an option to store the history of all best-known solutions encountered throughout the run of a heuristic. In general, heuristics may encounter any number of best-known solutions as they are run, and each requires $O(n)$ memory if they are stored.

Implementation of heuristics

While implementing all the heuristics in `C++` is important to reduce the impact of implementation details on observed performance, perhaps even more important are the shared components used across heuristics.

Instance We define classes `MaxCutInstance` (stores the instance as a graph adjacency list) and `QUBOInstance` (stores the instance as a sparse matrix and dense diagonal vector), which both share file loading code. Examples of functionality provided by the instance classes include efficient iterators, shuffled edge lists, and vertex degree counts. One caveat with our implementation is that if an instance is dense then the use of sparse storage formats may introduce some inefficiency. As the internal instance data structure is hidden from the heuristic code, dynamic selection of the correct storage format on a per-instance basis is a possible extension.

Solution The `MaxCutSolution` and `QUBOSolution` classes both inherit from an abstract `ExtendedSolution` class that consists of the solution vector, the objective value, and a vector quantifying how much the objective will change from flipping a vertex across the cut (Max-Cut) or flipping a variable's value between 0 and 1 (QUBO). This is essential to the efficient implementation of a local search, as not only can the best swap be found in $O(n)$ time for an instance with n nodes/variables, but the vector of objective value changes can also be updated efficiently. This data structure is described in some of the implemented papers, e.g. Pardalos et al. (2008).

Operations The base solution class also implements commonly used subroutines. For example, we implement the local search subroutines `AllBest1Swap` (repeatedly flip the node/variable that best improves the solution until no improving moves exist), `AllFirst1Swap` (repeatedly flip the lexicographically first occurring node/variable that improves the solution until no improving moves exist), and `AllShuffle1Swap` (shuffle the nodes/variables and then repeatedly flip the lexicographically first occurring node/variable that improves the solution until no improving moves exist). An additional example is `UpdateCutValues`, which efficiently flips a vertex across the cut (Max-Cut) or flips a variable's value between 0 and 1 (QUBO).

In our work all heuristics need to be able to run indefinitely with a chance of continuing to make progress (if not already optimal), as mentioned in Section 3 of the main paper. Many heuristics satisfy this property without modification: BUR02, FES02*, KAT00, LOD99, MER99*, MER02*, MER04, PAR08, and half of the PAL04* variants. We wrapped the remaining heuristics in random restarts.

Memory Scaling of Heuristics

To complement the asymptotic memory consumption of each heuristic displayed in Table 2, we also performed scaling analysis of the memory consumption of each heuristic. Because in this work

Paper	Type	Short name	Description	Space Req.
Alkhamis et al. (1998)	Q	ALK98	Simulated annealing	$O(n+m)$
Beasley (1998)	Q	BEA98SA	Simulated annealing	$O(n+m)$
		BEA98TS	Tabu search	$O(n+m)$
Burer et al. (2002)	M	BUR02	Non-linear optimization with local search	$O(n+m)$
Duarte et al. (2005)	M	DUA05	Genetic algorithm with VNS as local search	$O(pn+m)$
Festa et al. (2002)	M	FES02G	GRASP with local search	$O(n+m)$
		FES02GP	GRASP with path-relinking	$O(\lambda n+m)$
		FES02V	VNS	$O(n+m)$
		FES02VP	VNS with path-relinking	$O(\lambda n+m)$
		FES02GV	GRASP with VNS local search	$O(n+m)$
		FES02GVP	GRASP & VNS with path-relinking	$O(\lambda n+m)$
Glover et al. (1998)	Q	GLO98	Tabu search	$O(tn+m)$
Glover et al. (2010)	Q	GLO10	Tabu search with long-term memory	$O(\lambda n+m)$
Hasan et al. (2000)	Q	HAS00GA	Genetic algorithm	$O((p+\tau)n+m)$
		HAS00TS	Tabu search	$O(n+m)$
Katayama et al. (2000)	Q	KAT00	Genetic algorithm with k-opt local search	$O(pn+m)$
Katayama and Narihisa (2001)	Q	KAT01	Simulated annealing	$O(n+m)$
Laguna et al. (2009)	M	LAG09CE	Cross-entropy method	$O(5.87\rho n^2+m)$
		LAG09HCE	Cross-entropy method with local search	$O(0.031n^2+m)$
Lodi et al. (1999)	Q	LOD99	Genetic algorithm	$O(pn+m)$
Lü et al. (2010)	Q	LU10	Genetic algorithm with tabu search	$O(pn+m)$
Merz and Freisleben (1999)	Q	MER99LS	Genetic algorithm, with crossover and local search	$O(\beta n+m)$
		MER99MU	Genetic algorithm, with mutation only	$O(\beta n+m)$
		MER99CR	Genetic algorithm, with crossover only	$O(\beta n+m)$
Merz and Freisleben (2002)	Q	MER02GR	GRASP without local search	$O(n+m)$
		MER02LS1	1-opt local search with random restarts	$O(n+m)$
		MER02LSK	k-opt local search with random restarts	$O(n+m)$
		MER02GRK	k-opt local search with GRASP	$O(n+m)$
Merz and Katayama (2004)	Q	MER04	Genetic algorithm, with k-opt local search	$O(\alpha n+m)$
Palubeckis (2004)	Q	PAL04T1	Tabu search	$O(n+m)$
		PAL04T2	Iterated tabu search	$O(n+m)$
		PAL04T3	Tabu search with GRASP	$O(n+m)$
		PAL04T4	Tabu search with long-term memory	$O((\lambda+\gamma)n+m)$
		PAL04T5	Iterated tabu search	$O(n+m)$
		PAL04MT	Tabu search	$O(n+m)$
Palubeckis (2006)	Q	PAL06	Iterated tabu search	$O(n+m)$
Pardalos et al. (2008)	Q	PAR08	Global equilibrium search	**

Table 2: Implemented heuristics for the Max-Cut ("M") and QUBO ("Q") problems with their asymptotic space requirements. For the space requirements, n is the number of nodes, m is the number of edges (Max-Cut) or number of non-zero matrix entries (QUBO), p is the initial population size, λ is the maximum number of elite solutions maintained, γ is the maximum size of the set of solutions to be avoided that the algorithm maintains, t is the number of tabu neighborhoods, τ is the number of tournaments, ρ is the fraction of the population retained in each iteration, β is the recombination rate, and α is the crossover rate. For Pardalos et al. (2008), the memory requirement is at least $O(\lambda n+m)$.

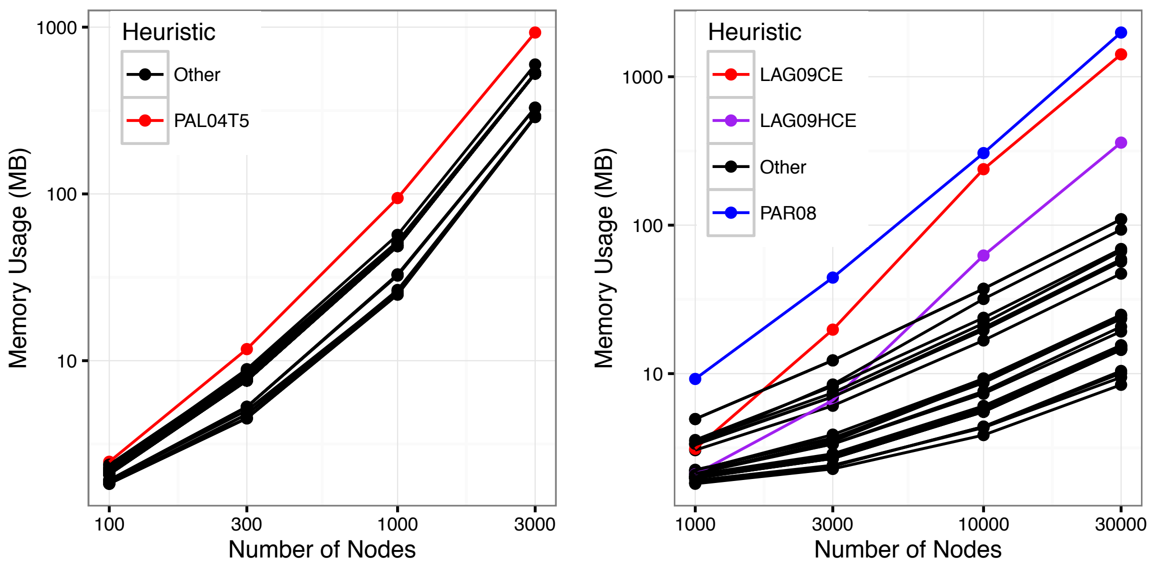


Figure 2: A comparison of the memory scaling behavior of the heuristics (note the log-transformed axes). (Left) Complete graphs with randomly selected edge weights. (Right) Sparse Erdős-Rényi graphs with expected degree 5 and randomly selected edge weights.

all heuristics are provided the same runtime limit, no scaling analysis was performed of heuristic runtimes.

First, we tested each heuristic on randomly generated dense Max-Cut instances consisting of complete graphs with each edge weight randomly selected with equal probability from the set {-1, 1}. To observe memory scaling behavior on these dense problem instances, we randomly generated instances of size 100, 300, 1,000, and 3,000 nodes and tested each heuristic on each instance. For each instance, all heuristics were given the same runtime limit, as described in Section 4.1 of the main paper. The peak memory consumption of each heuristic for different dense instance sizes is plotted in Figure 2 (left). The observed scaling confirms an $O(n^2)$ memory consumption for all heuristics on dense problem instances, as expected from the asymptotic analysis. Heuristic PAL04T5 stands out as having more memory usage than the other heuristics, which is because that heuristic constructs perturbed versions of the problem instance, a memory-intensive procedure.

Further, we tested each heuristic on randomly generated sparse Max-Cut instances. To construct an instance with n nodes, we randomly generated an Erdős-Rényi graph with n nodes and edge probability $5/(n-1)$, meaning each node in the graph has expected degree 5. Edge weights for selected edges were again randomly selected with equal probability from the set {-1, 1}. To observe memory scaling behavior on these sparse problem instances, we randomly generated instances of size 1,000, 3,000, 10,000, and 30,000 nodes and tested each heuristic on each instance. Again, we gave each heuristic the same runtime limit, as described in Section 4.1 of the main paper. The peak memory consumption of each heuristic for different sparse instance sizes is plotted in Figure 2 (right). The observed scaling confirms an $O(n)$ memory consumption for all heuristics except LAG09CE, LAG09HCE, and PAR08 on sparse problem instances, as expected from the asymptotic analysis. Further, LAG09CE and LAG09HCE exhibit expected $O(n^2)$ memory consumption on sparse problem instances. Finally, heuristic PAR08 consumes more memory than any other heuris-

variable	MDGr	MDG	pct	variable	MDGr	MDG	pct
log_n	7.90	23.00	74.00	log_m	8.10	13.00	74.00
log_norm_ev2	8.50	12.40	68.00	log_ev_ratio	10.00	10.10	71.00
log_norm_ev1	10.90	9.00	65.00	weight_log_kurtosis	11.40	11.30	53.00
weight_min	13.00	10.80	53.00	weight_mean	13.40	9.90	47.00
weight_stdev	13.40	8.40	47.00	deg_stdev	14.40	7.60	47.00
weight_log_abs_skew	16.60	6.80	32.00	mis	18.60	8.90	29.00
assortativity	18.70	7.60	26.00	avg_deg_conn_min	19.90	6.30	21.00
avg_deg_conn_stdev	20.20	5.30	26.00	deg_log_abs_skew	20.40	5.60	21.00
deg_max	21.50	6.80	15.00	avg_deg_conn_log_kurtosis	21.80	6.20	26.00
avg_neighbor_deg_min	22.30	5.80	15.00	chromatic	23.10	5.50	6.00
deg_log_kurtosis	23.60	5.10	12.00	avg_neighbor_deg_stdev	24.70	5.30	24.00
avg_deg_conn_max	25.40	3.50	12.00	avg_deg_conn_mean	25.40	3.80	6.00
avg_neighbor_deg_mean	25.40	3.90	12.00	core_mean	25.50	4.80	18.00
avg_neighbor_deg_max	26.10	3.80	15.00	deg_mean	26.40	4.20	12.00
core_stdev	26.60	4.50	6.00	deg_min	26.80	3.60	9.00
avg_neighbor_deg_log_abs_skew	27.30	4.20	0.00	avg_neighbor_deg_log_kurtosis	27.40	4.10	6.00
core_log_kurtosis	27.90	4.40	9.00	clust_stdev	28.00	4.60	6.00
core_min	28.40	3.50	6.00	clust_mean	28.60	5.10	3.00
avg_deg_conn_log_abs_skew	29.10	3.90	9.00	core_log_abs_skew	30.50	3.90	3.00
clust_max	31.10	6.00	6.00	core_max	33.10	2.90	0.00
clust_log_abs_skew	33.40	3.60	0.00	percent_pos	34.80	3.60	3.00
clust_log_kurtosis	35.30	3.50	0.00	clust_min	35.40	3.60	6.00
avg_neighbor_deg_skew_positive	44.40	1.90	3.00	deg_skew_positive	44.90	1.50	3.00
weight_skew_positive	47.50	0.50	0.00	avg_deg_conn_skew_positive	48.00	0.60	0.00
clust_skew_positive	49.70	0.40	0.00	weight_const	50.80	0.30	0.00
weight_max	51.10	0.40	0.00	core_skew_positive	51.40	0.30	0.00
clust_const	51.60	0.40	0.00	core_const	51.70	0.20	0.00
deg_const	54.50	0.10	0.00	avg_deg_conn_const	54.80	0.10	0.00
avg_neighbor_deg_const	55.10	0.10	0.00	disconnected	55.40	0.10	0.00

Table 3: The variable importance of each feature averaged over all the heuristic-specific random forest models, showing overall importance in predicting heuristic performance for any given instance, as described in Section 6.1. Here, MDGr is the mean decrease in Gini rank, MDG is the mean decrease in Gini, and pct is the percentage of random forest models for which this feature was in the top 10 most important variables. Variables are sorted in increasing order by the mean decrease in Gini rank, with ties broken by the mean decrease in Gini.

tic. This is because PAR08 stores all locally optimal solutions encountered during its tabu search procedure, which can be quite memory intensive.

4 Random Forest Variable Importance

Table 3 lists the variable importance values for all the 58 metrics used in this work, averaged over the random forest models for the 37 heuristics.

5 Interpretable Models for All Heuristics

In this supplementary material, we give the CART models for each heuristic as described in Section 5.1, in the main paper. Table 4 summarizes the R^2 value of each fitted CART model and gives the list of relevant plots for all the 37 heuristics.

Heuristic	R^2	Figures	Heuristic	R^2	Figures	Heuristic	R^2	Figures
ALK98	0.49	Figure 3	BEA98SA	0.41	Figure 4	BEA98TS	0.75	Figure 5
BUR02	0.61	Figure 6	DUA05	0.55	Figure 7	FES02G	0.78	Figure 8
FES02GP	0.76	Figure 9	FES02GV	0.60	Figure 10	FES02GVP	0.68	Figure 11
FES02V	0.37	Figure 12	FES02VP	0.28	Figure 13	GLO10	0.60	Figure 14
GLO98	0.58	Figure 15	HAS00GA	0.54	Figure 16	HAS00TS	0.43	Figure 17
KAT00	0.48	Figure 18	KAT01	0.39	Figure 19	LAG09CE	0.20	Figure 20
LAG09HCE	0.49	Figure 21	LOD99	0.50	Figure 22	LU10	0.71	Figure 23
MER02GR	0.75	Figure 24	MER02GRK	0.50	Figure 25	MER02LS1	0.54	Figure 26
MER02LSK	0.58	Figure 27	MER04	0.30	Figure 28	MER99CR	0.50	Figure 29
MER99LS	0.46	Figure 30	MER99MU	0.24	Figure 31	PAL04MT	0.67	Figure 32
PAL04T1	0.75	Figure 33	PAL04T2	0.29	Figure 34	PAL04T3	0.32	Figure 35
PAL04T4	0.75	Figure 36	PAL04T5	0.68	Figure 37	PAL06	0.61	Figure 38
PAR08	0.65	Figure 39						

Table 4: The R^2 and figure number for each heuristic's CART model predicting instance-specific performance, as described in Section 5.1.

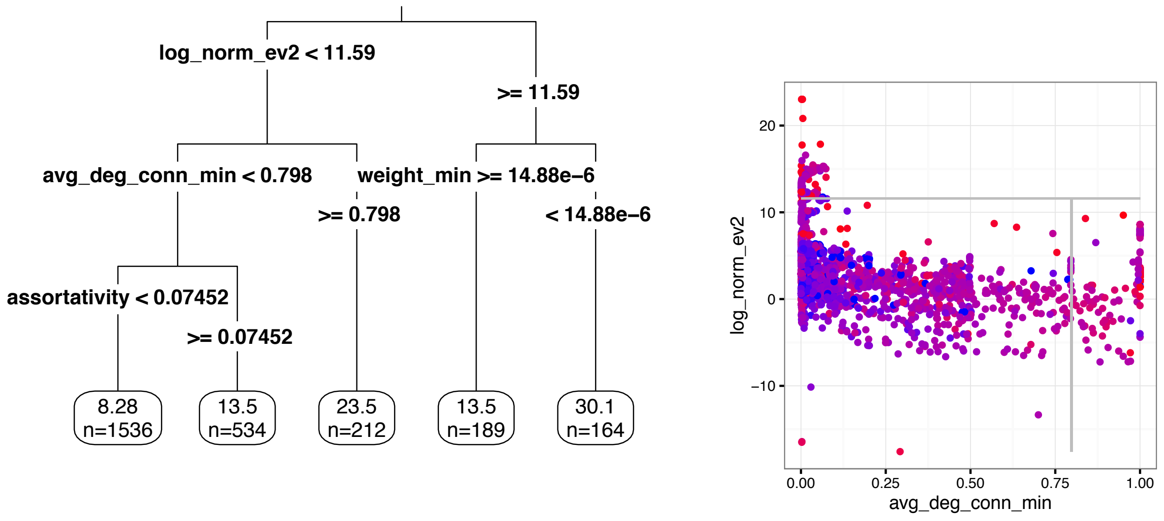


Figure 3: A CART model identifying instances on which ALK98 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

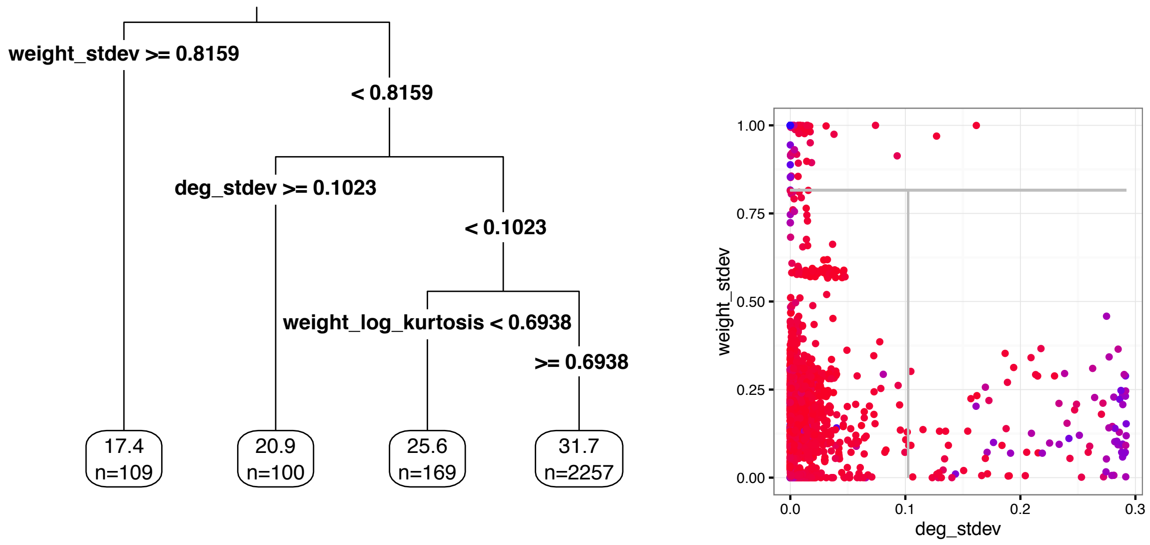


Figure 4: A CART model identifying instances on which BEA98SA performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

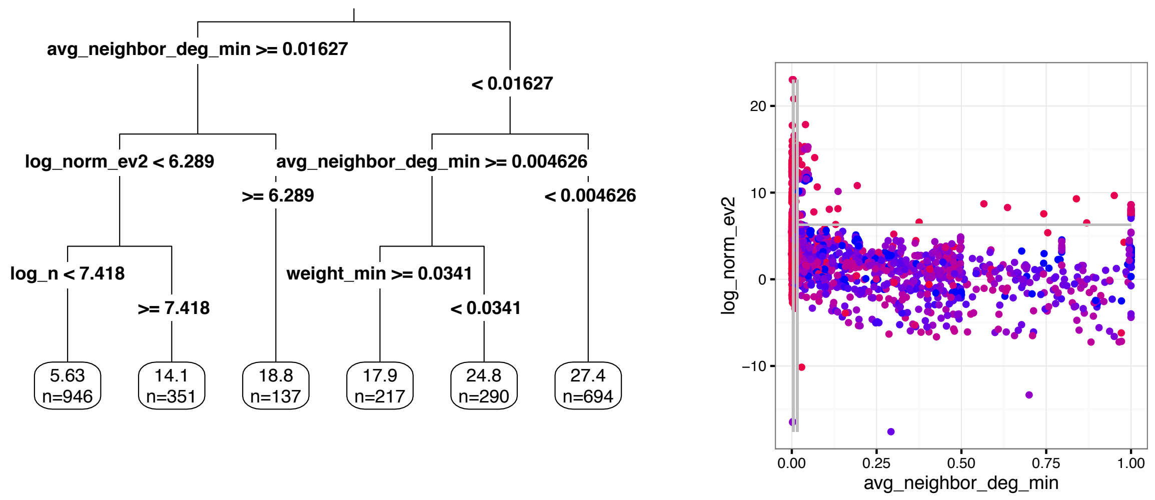


Figure 5: A CART model identifying instances on which BEA98TS performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

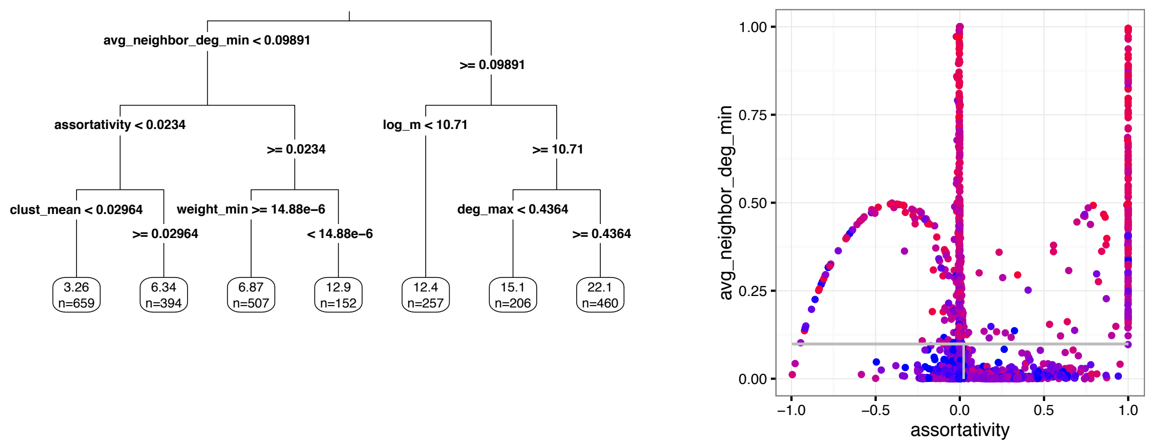


Figure 6: A CART model identifying instances on which BUR02 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

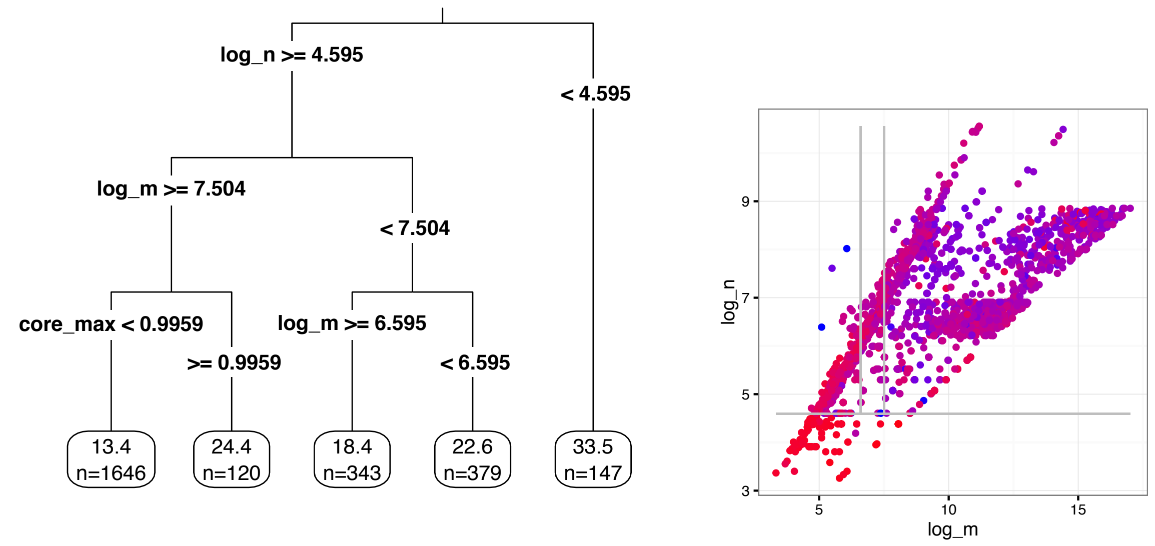


Figure 7: A CART model identifying instances on which DUA05 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

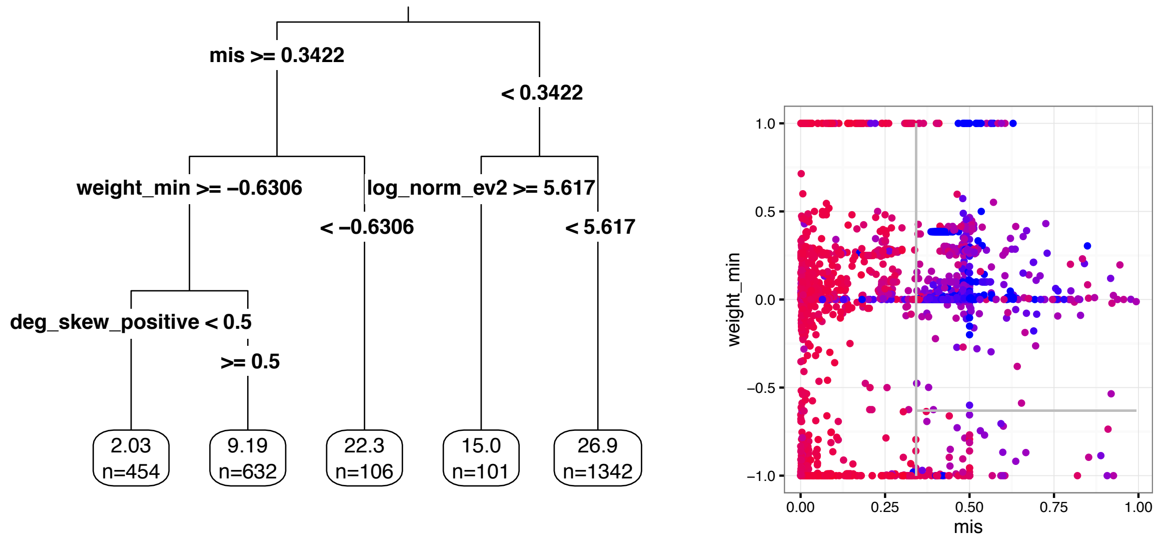


Figure 8: A CART model identifying instances on which FES02G performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

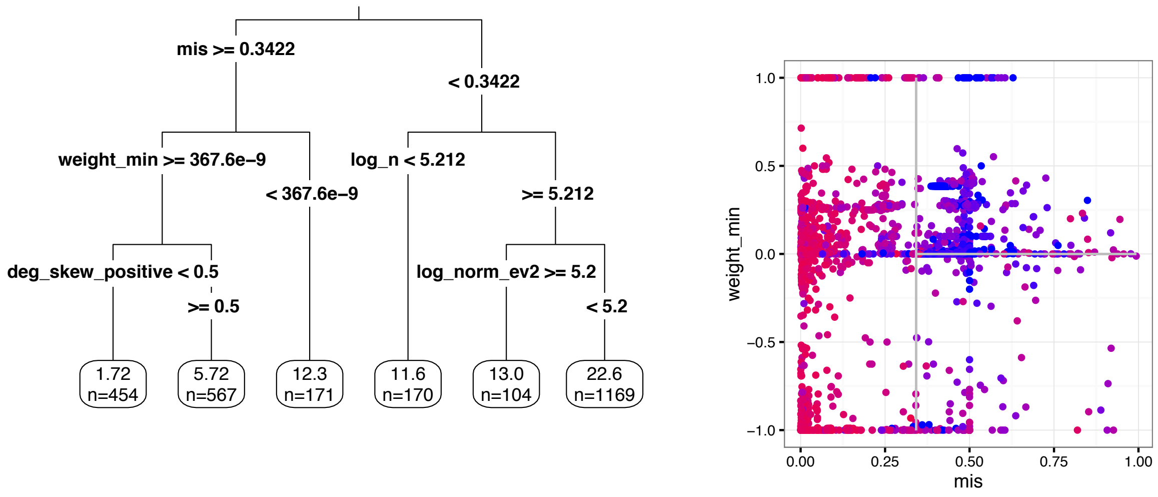


Figure 9: A CART model identifying instances on which FES02GP performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

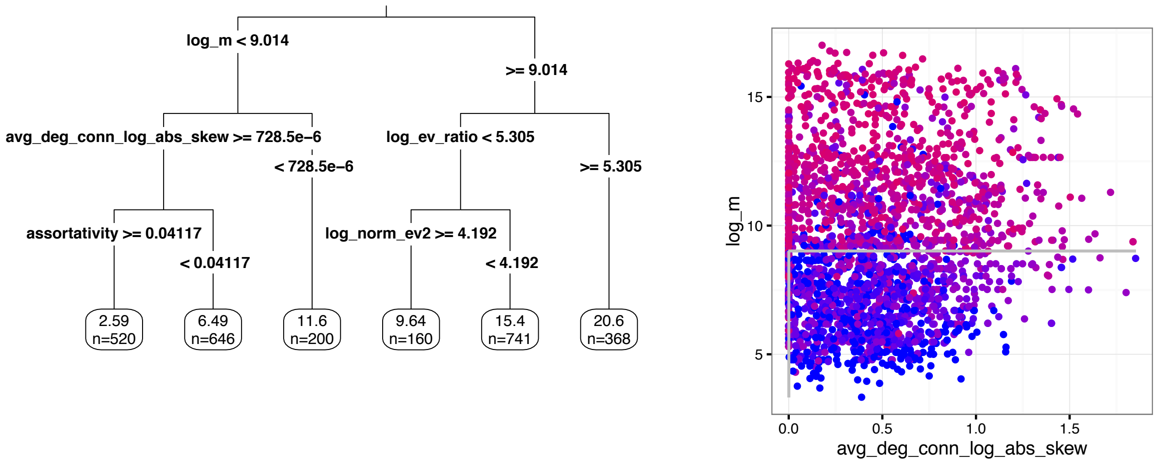


Figure 10: A CART model identifying instances on which FES02GV performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

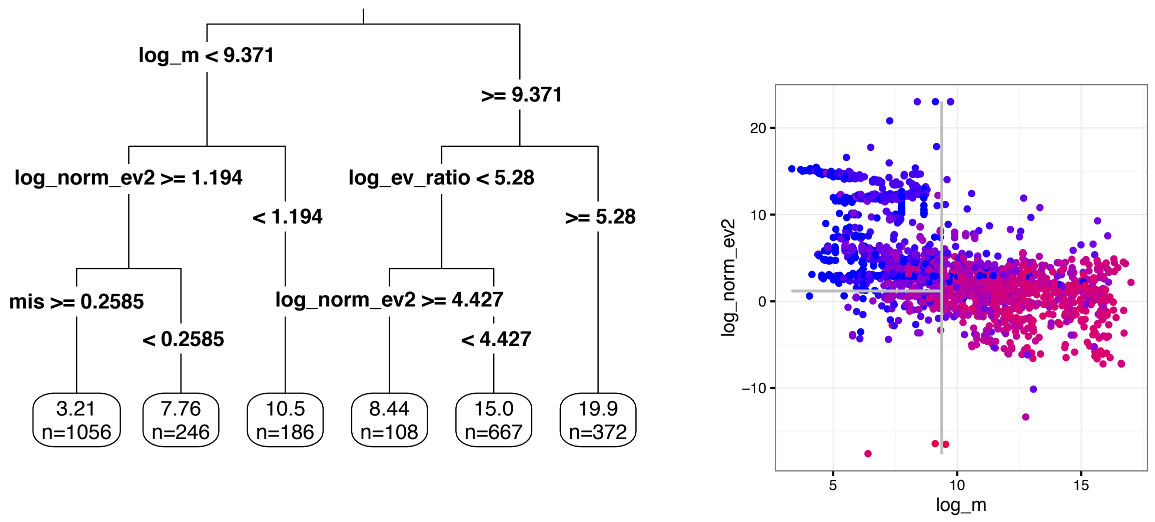


Figure 11: A CART model identifying instances on which FES02GVP performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

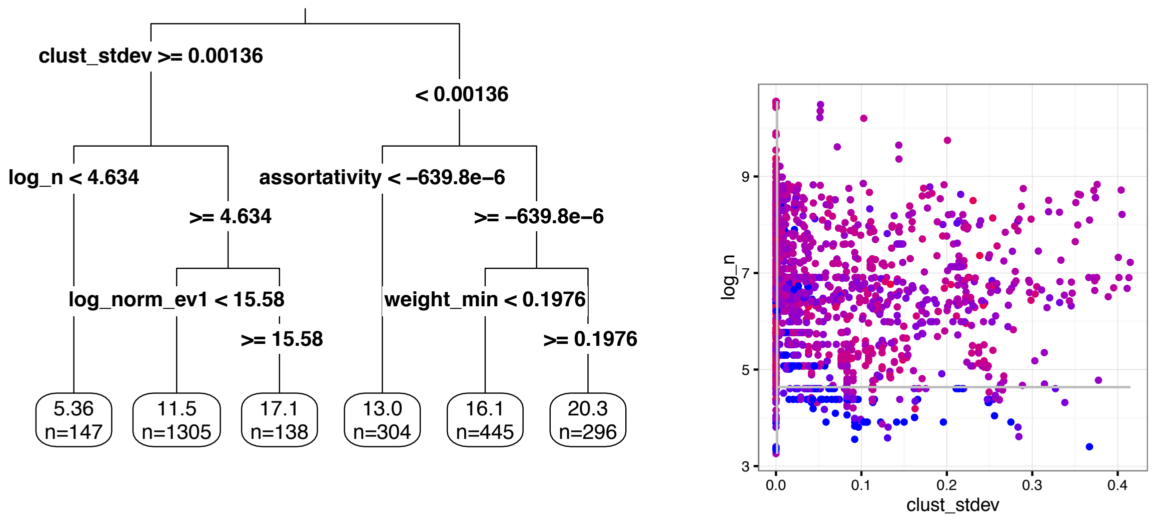


Figure 12: A CART model identifying instances on which FES02V performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

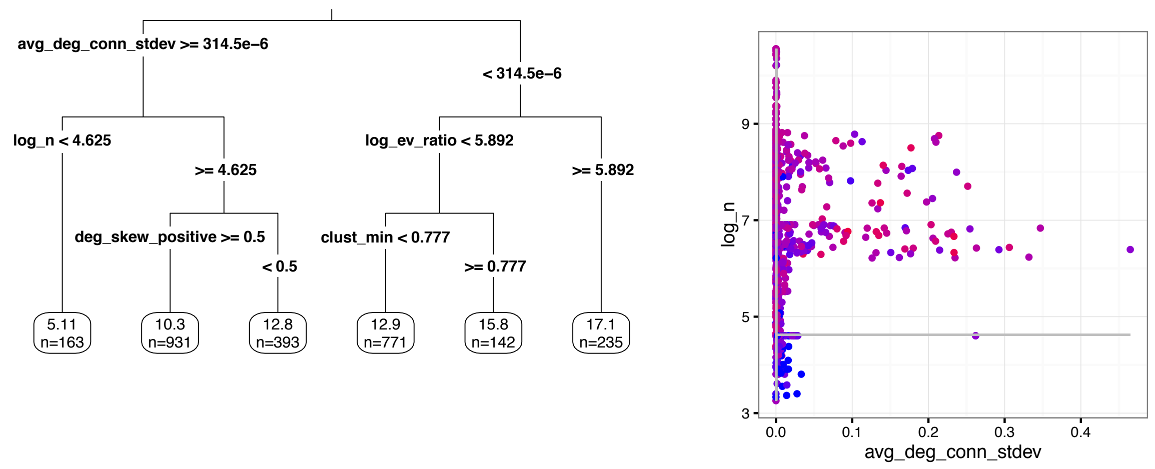


Figure 13: A CART model identifying instances on which FES02VP performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

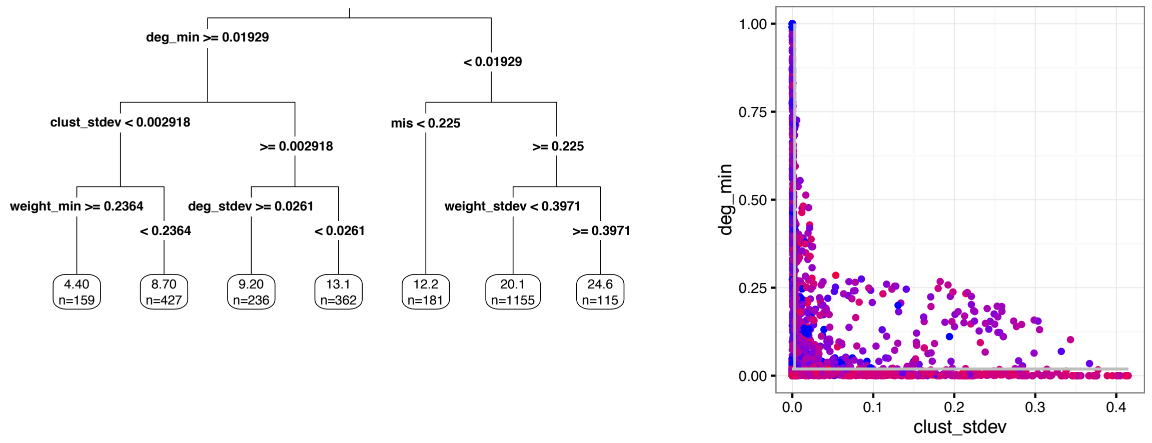
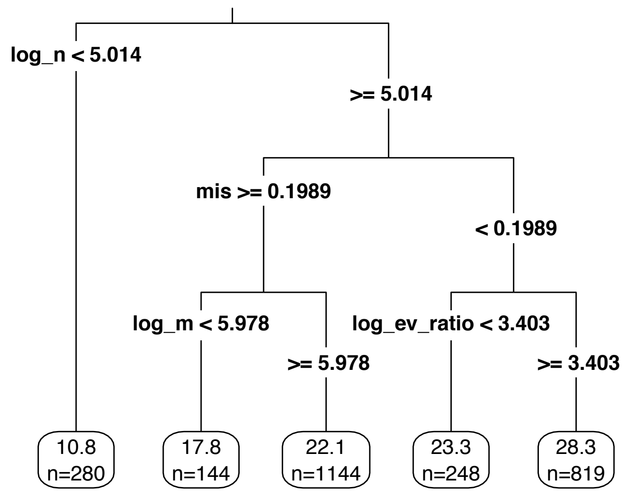


Figure 14: A CART model identifying instances on which GLO10 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).



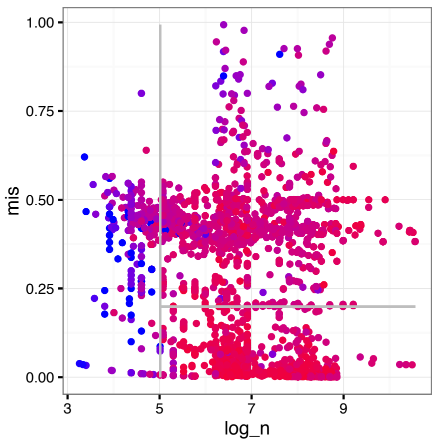
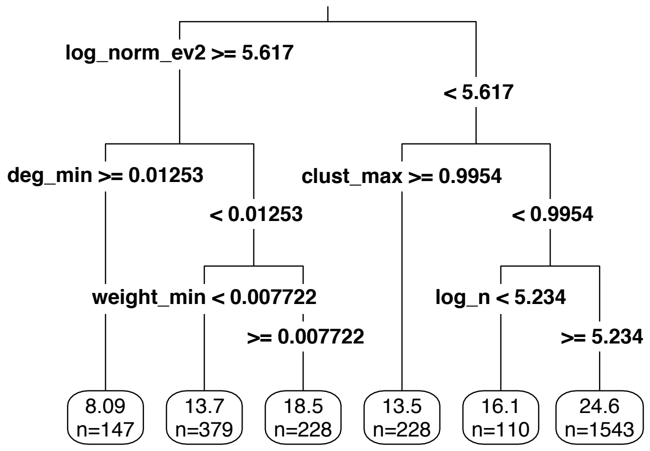


Figure 15: A CART model identifying instances on which GLO98 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).



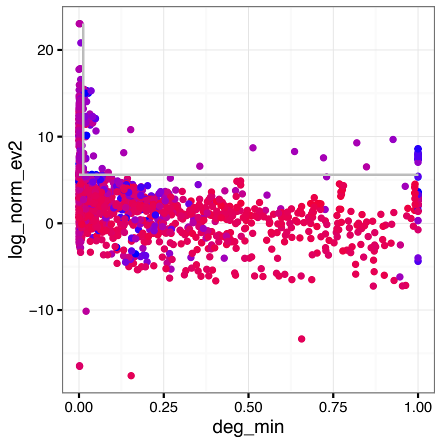


Figure 16: A CART model identifying instances on which HAS00GA performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

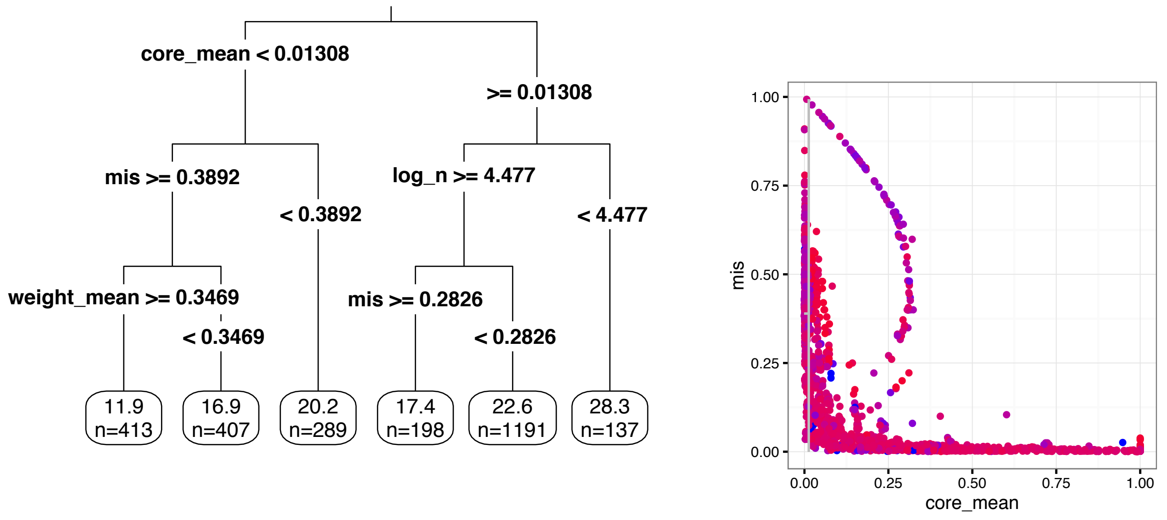


Figure 17: A CART model identifying instances on which HAS00TS performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

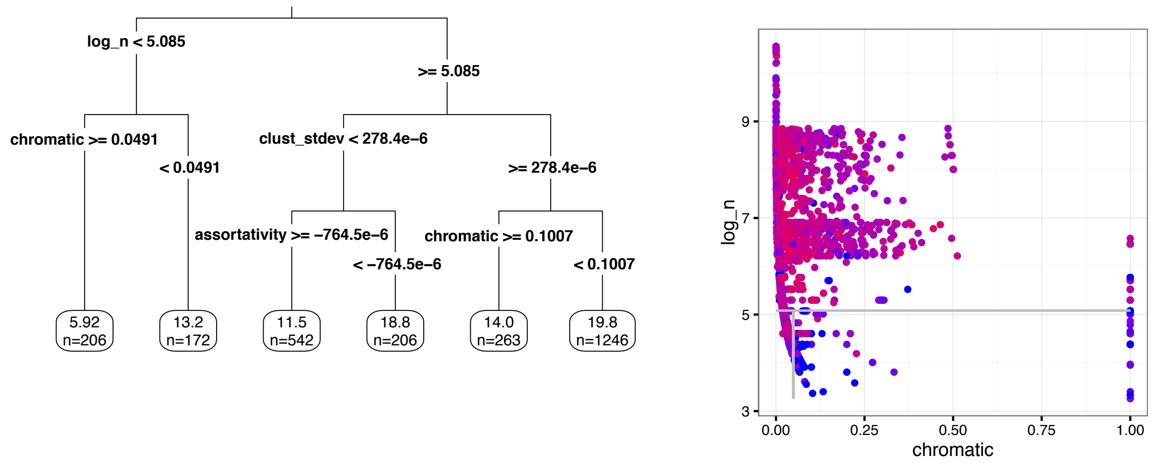


Figure 18: A CART model identifying instances on which KAT00 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

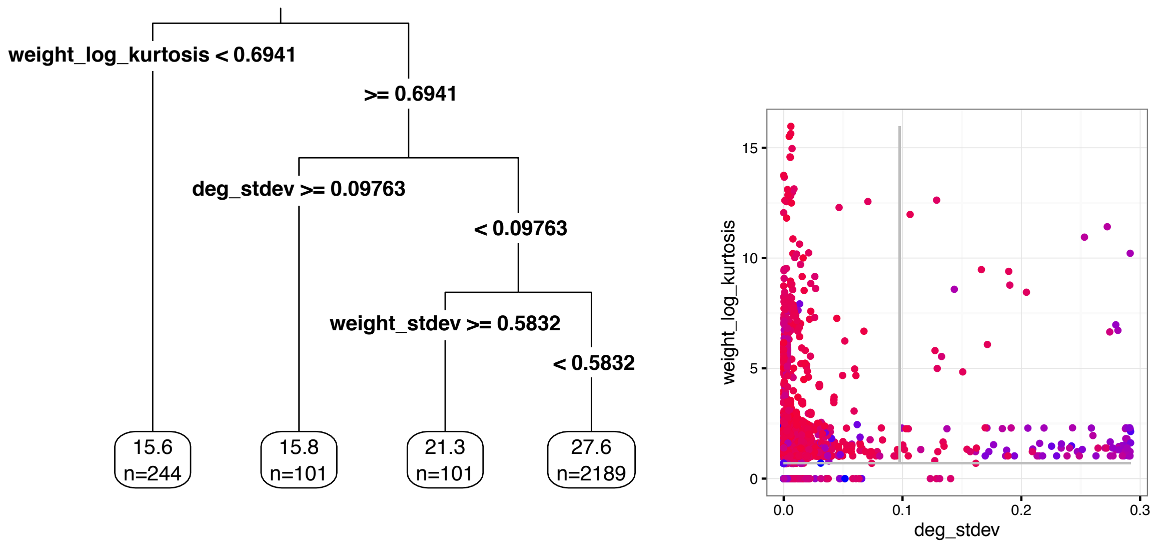


Figure 19: A CART model identifying instances on which KAT01 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

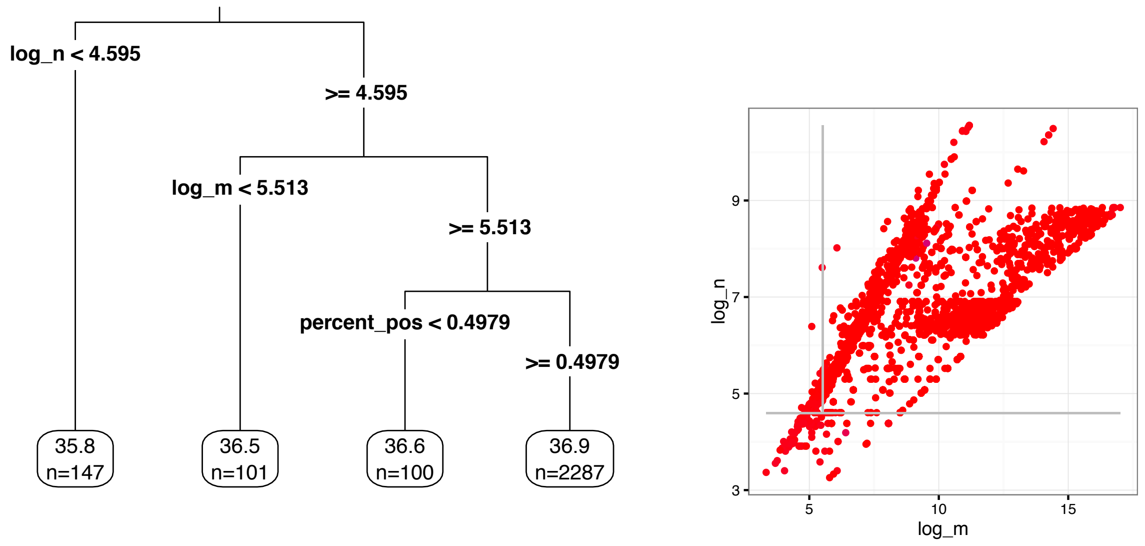
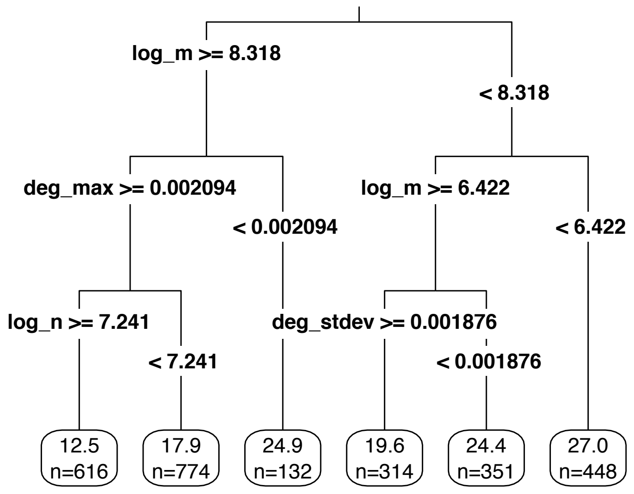


Figure 20: A CART model identifying instances on which LAG09CE performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).



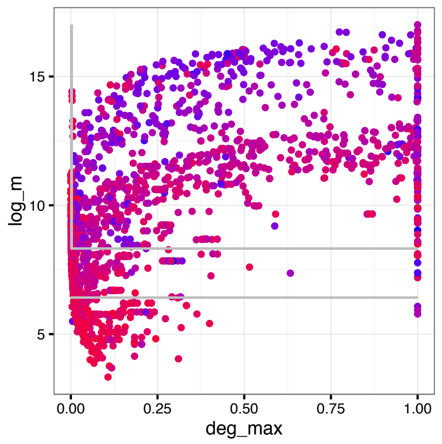
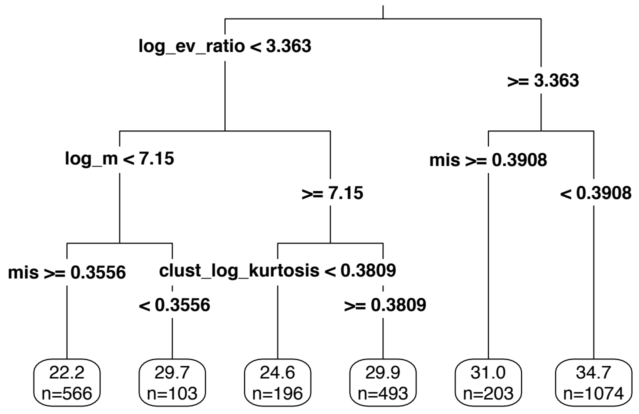


Figure 21: A CART model identifying instances on which LAG09HCE performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).



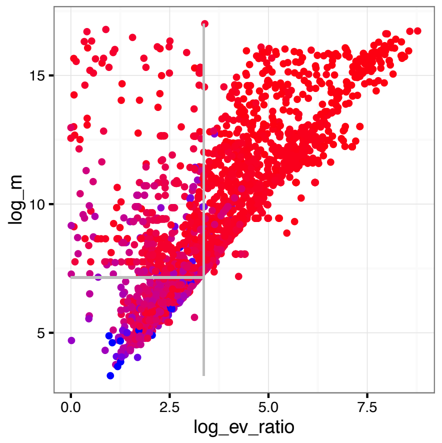


Figure 22: A CART model identifying instances on which LOD99 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

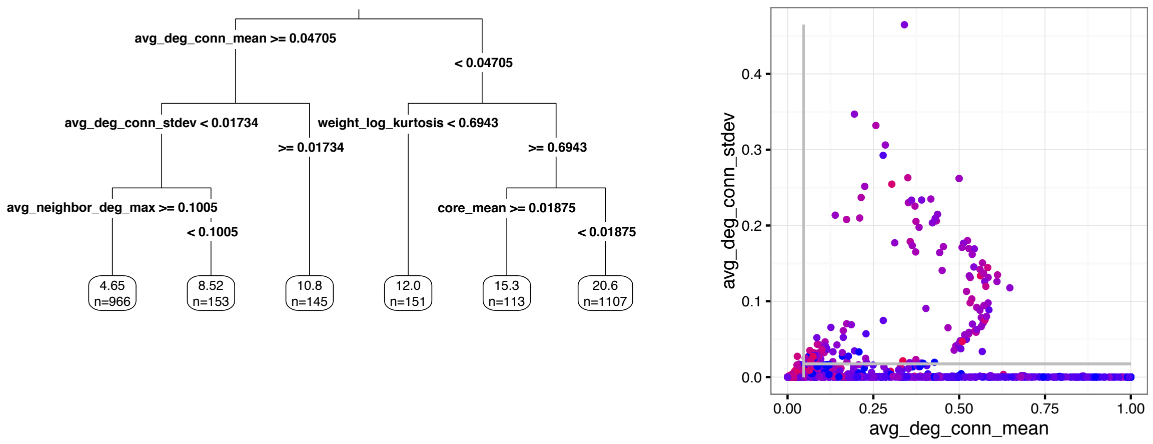


Figure 23: A CART model identifying instances on which LU10 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

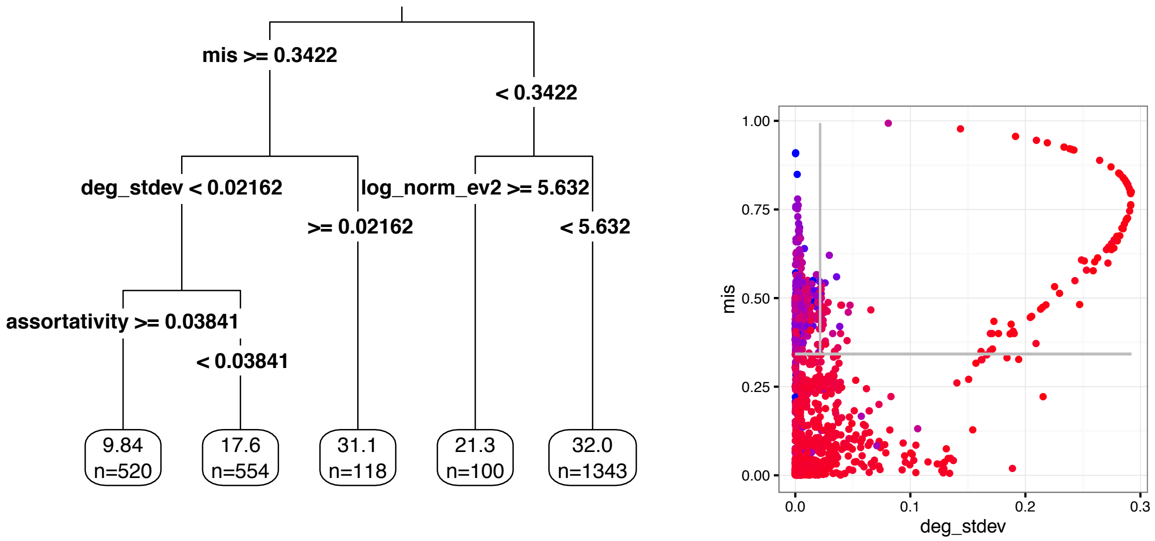
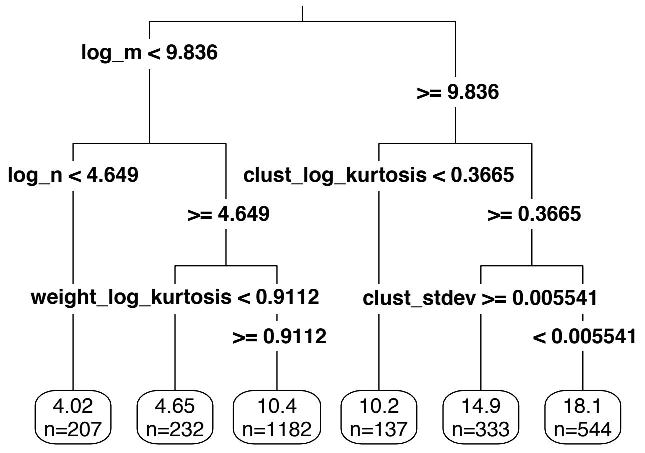


Figure 24: A CART model identifying instances on which MER02GR performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).



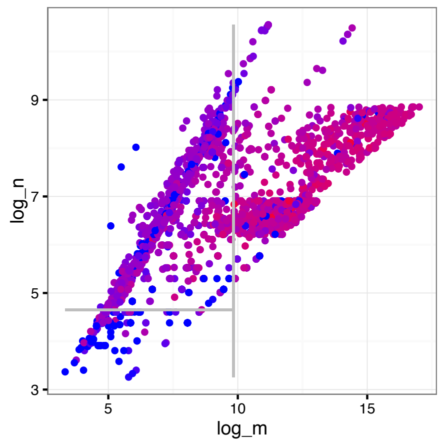
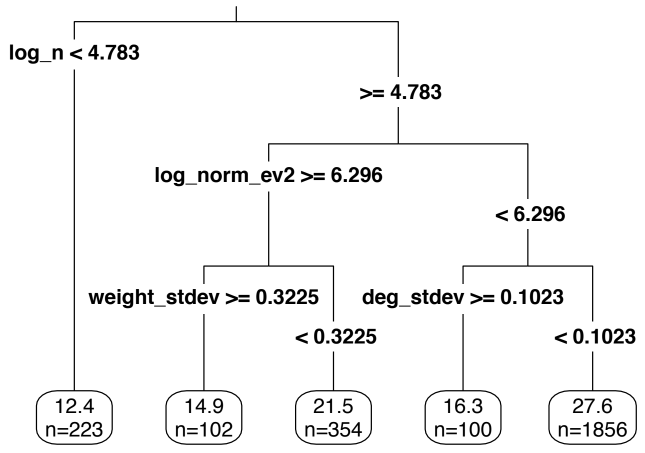


Figure 25: A CART model identifying instances on which MER02GRK performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).



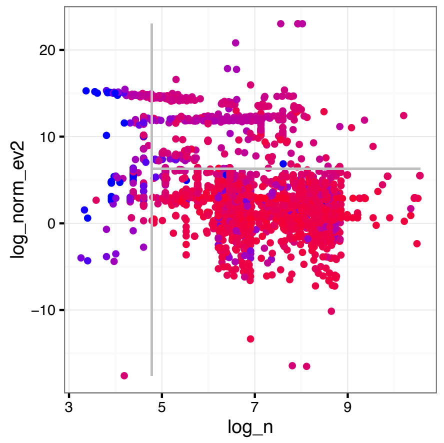


Figure 26: A CART model identifying instances on which MER02LS1 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

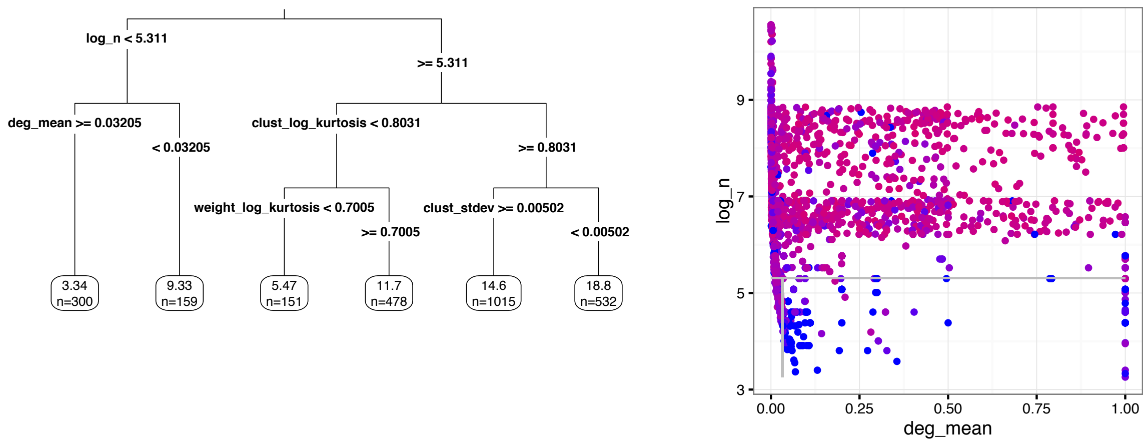


Figure 27: A CART model identifying instances on which MER02LSK performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

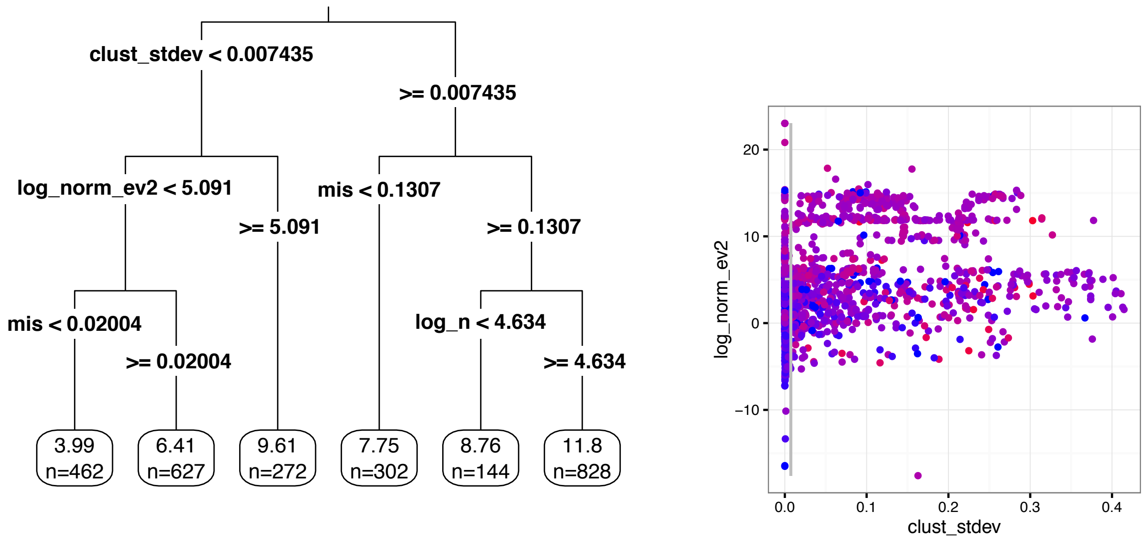


Figure 28: A CART model identifying instances on which MER04 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

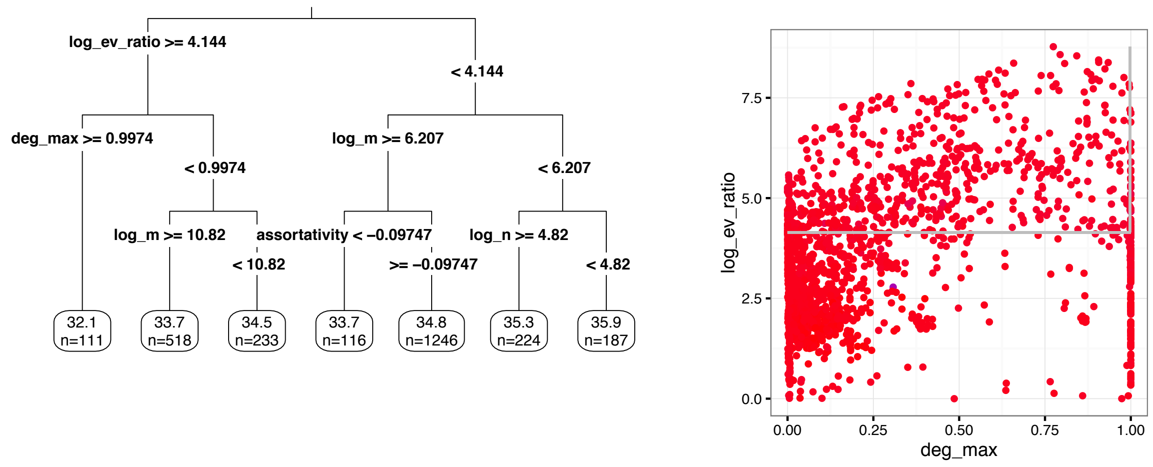


Figure 29: A CART model identifying instances on which MER99CR performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

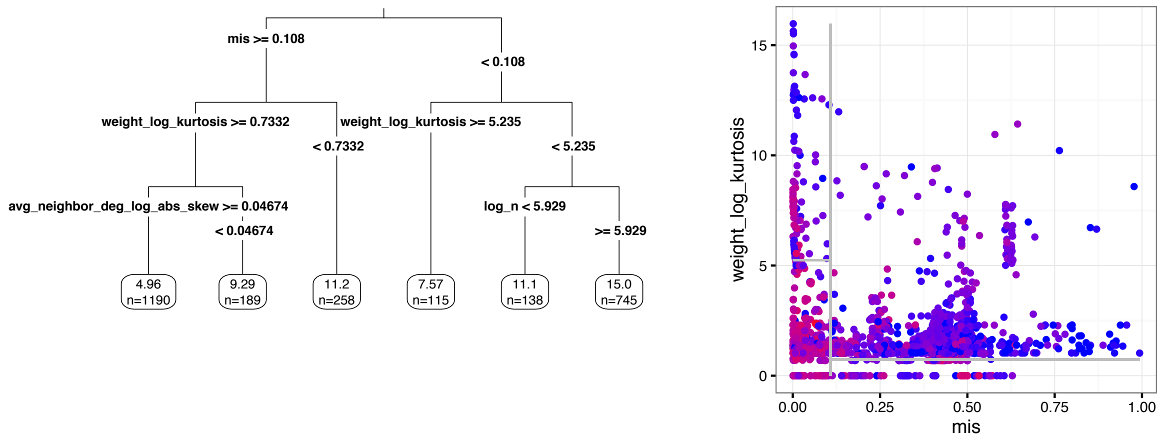
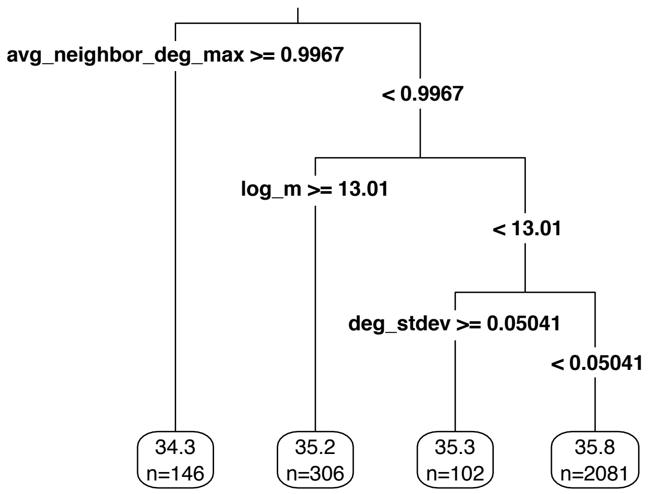


Figure 30: A CART model identifying instances on which MER99LS performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).



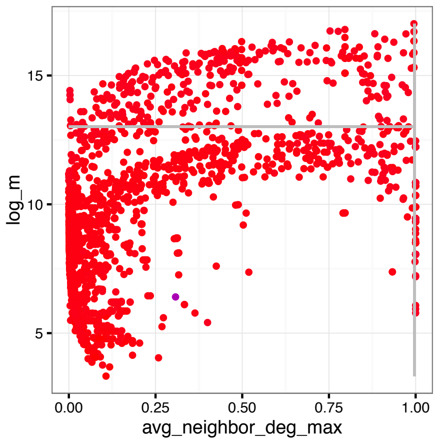
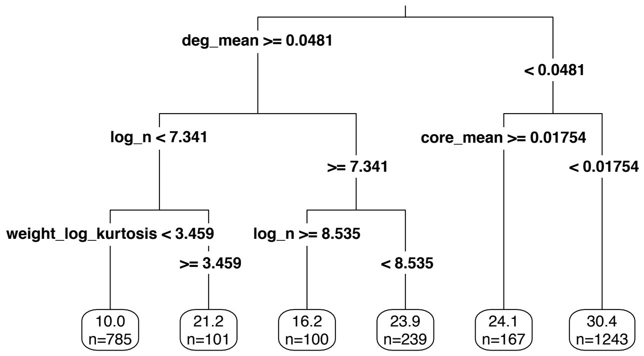


Figure 31: A CART model identifying instances on which MER99MU performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).



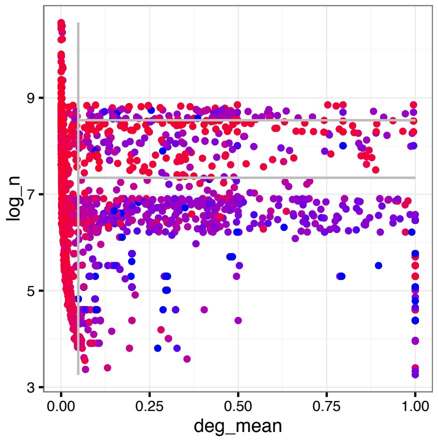


Figure 32: A CART model identifying instances on which PAL04MT performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

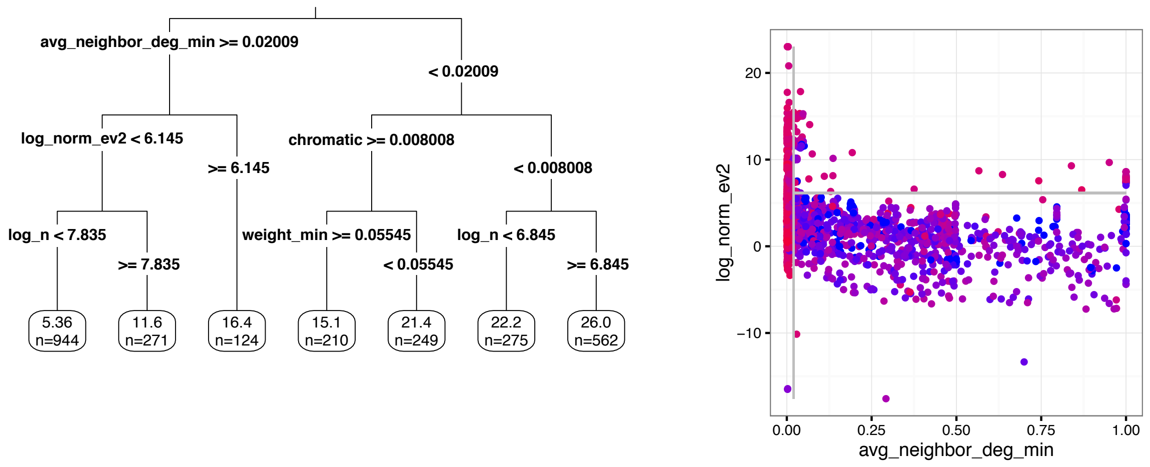


Figure 33: A CART model identifying instances on which PAL04T1 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

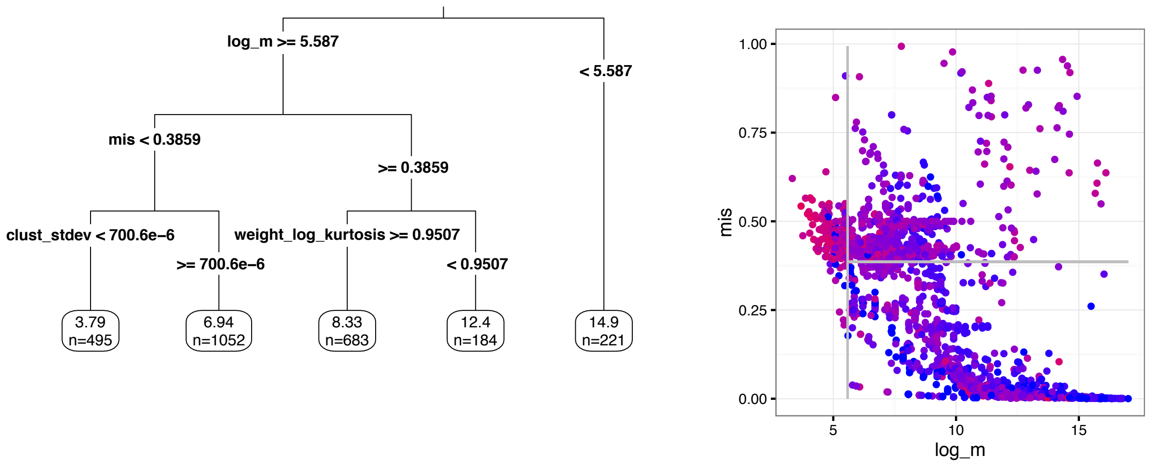


Figure 34: A CART model identifying instances on which PAL04T2 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

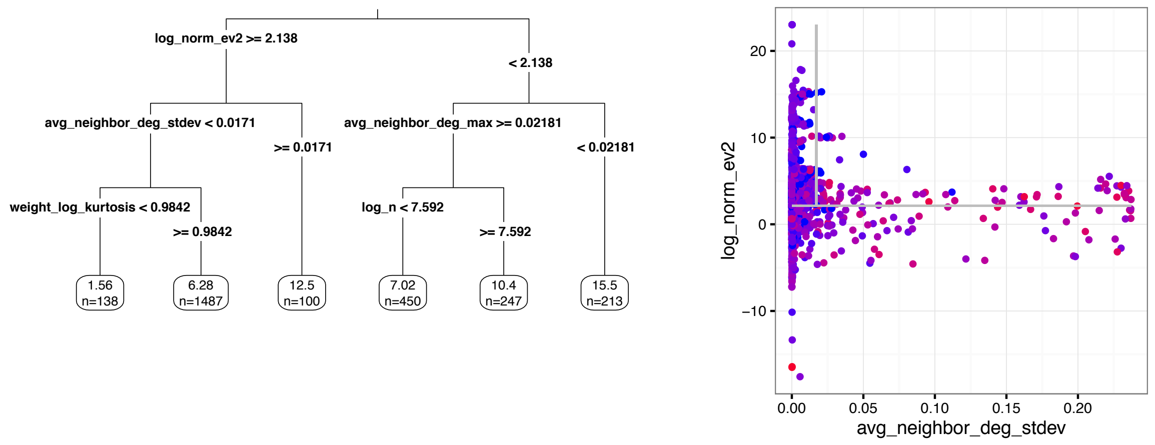


Figure 35: A CART model identifying instances on which PAL04T3 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

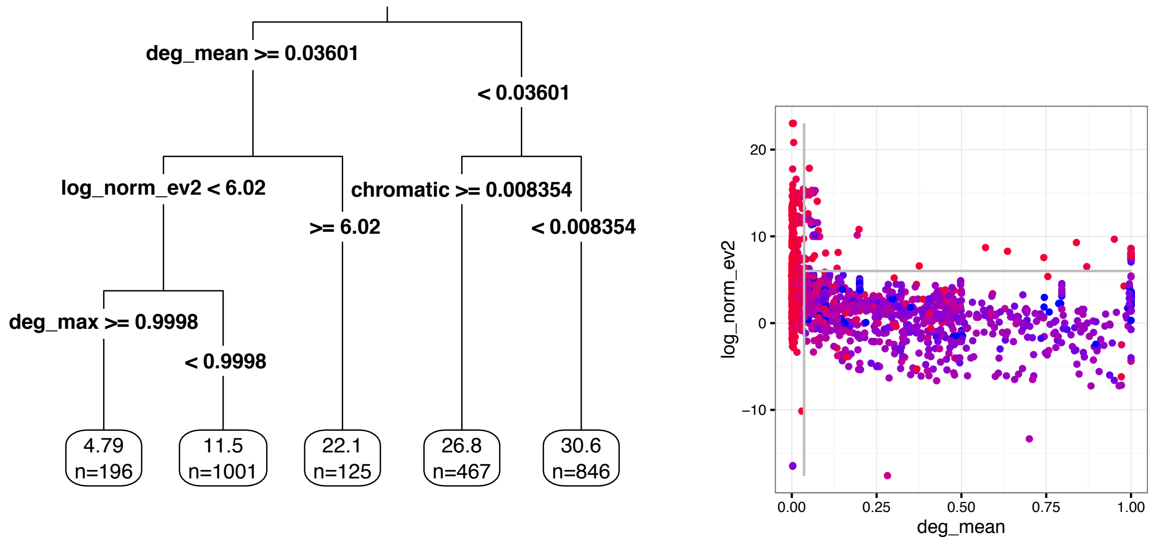


Figure 36: A CART model identifying instances on which PAL04T4 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

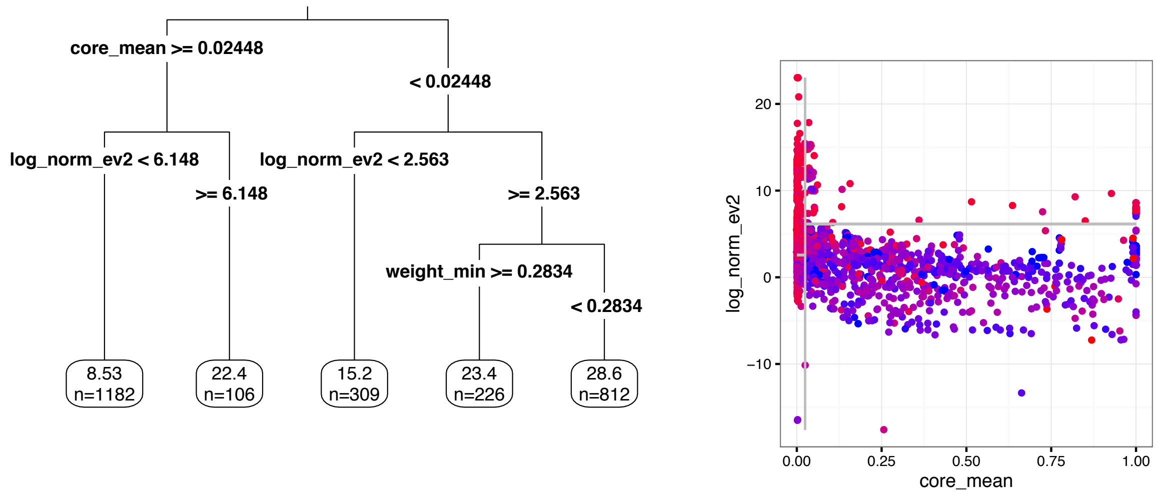


Figure 37: A CART model identifying instances on which PAL04T5 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

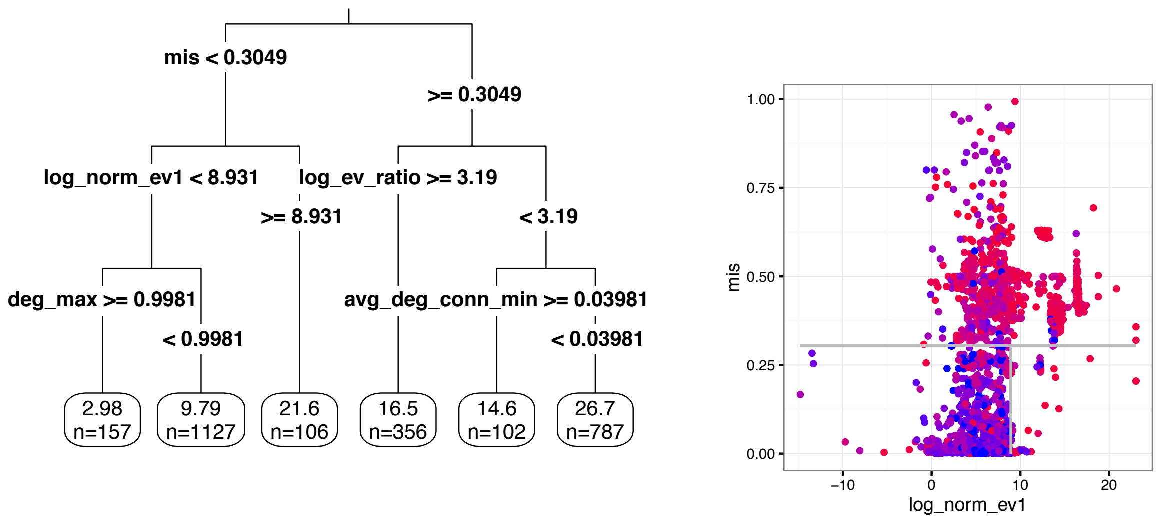
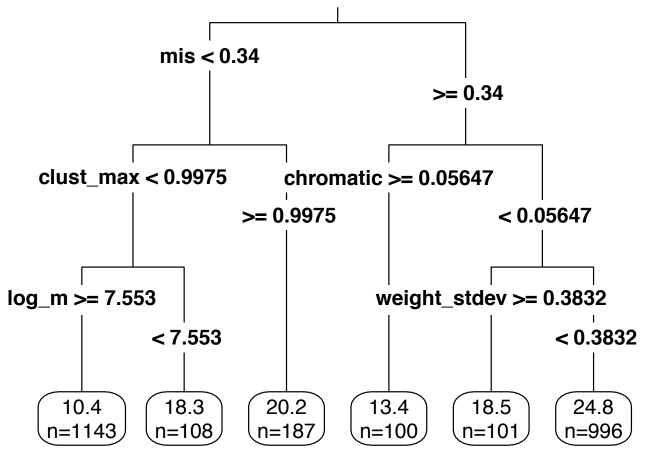


Figure 38: A CART model identifying instances on which PAL06 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).



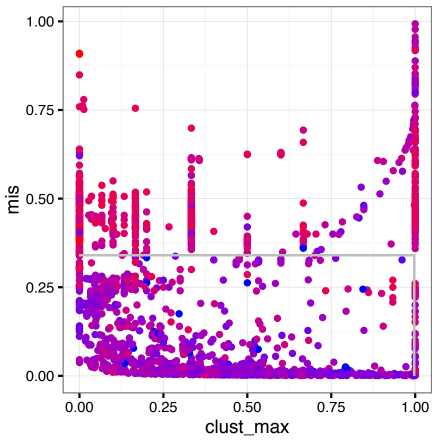


Figure 39: A CART model identifying instances on which PAR08 performs particularly well or poorly. Blue indicates the heuristic performed well (rank near 1) and red indicates the heuristic performed poorly (rank near 37).

References

Alkhamis, Talal M, Merza Hasan, Mohamed A Ahmed. 1998. Simulated annealing for the unconstrained quadratic pseudo-Boolean function. *European Journal of Operational Research* **108** 641–652.

Barabási, Albert-László, Réka Albert. 1999. Emergence of scaling in random networks. *Science* **286** 509–512.

Beasley, John E. 1990. OR-Library: distributing test problems by electronic mail. *Journal of the operational research society* **41** 1069–1072.

Beasley, John E. 1998. Heuristic algorithms for the unconstrained binary quadratic programming problem. Tech. rep., Imperial College, London, UK.

Burer, Samuel, Renato DC Monteiro, Yin Zhang. 2002. Rank-two relaxation heuristics for Max-Cut and other binary quadratic programs. *SIAM Journal on Optimization* **12** 503–521.

Culberson, Joseph, Adam Beacham, Denis Papp. 1995. Hiding our colors. *CP95 Workshop on Studying and Solving Really Hard Problems*. Citeseer, 31–42.

Duarte, Abraham, Ángel Sánchez, Felipe Fernández, Raúl Cabido. 2005. A low-level hybridization between memetic algorithm and VNS for the Max-Cut problem. *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM, 999–1006.

Erdös, Paul, A Rényi. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci* **5** 17–61.

Festa, Paola, Panos M Pardalos, Mauricio GC Resende, Celso C Ribeiro. 2002. Randomized heuristics for the MAX-CUT problem. *Optimization methods and software* **17** 1033–1058.

Glover, Fred, Gary A Kochenberger, Bahram Alidaee. 1998. Adaptive memory tabu search for binary quadratic programs. *Management Science* **44** 336–345.

Glover, Fred, Zhipeng Lü, Jin-Kao Hao. 2010. Diversification-driven tabu search for unconstrained binary quadratic problems. *4OR* **8** 239–253.

Hagberg, Aric A., Daniel A. Schult, Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. Gaël Varoquaux, Travis Vaught, Jarrod Millman, eds., *Proceedings of the 7th Python in Science Conference*. Pasadena, CA, USA, 11–15.

Hammer, P.L. 1965. Some network flow problems solved with pseudo-boolean programming. *Operations Research* **13** 388–399.

Hasan, Merza, T AlKhamis, J Ali. 2000. A comparison between simulated annealing, genetic algorithm and tabu search methods for the unconstrained quadratic Pseudo-Boolean function. *Computers & Industrial Engineering* **38** 323–340.

Helmberg, Christoph, Franz Rendl. 2000. A spectral bundle method for semidefinite programming. *SIAM Journal on Optimization* **10** 673–696.

Katayama, Kengo, Hiroyuki Narihisa. 2001. Performance of simulated annealing-based heuristic for the unconstrained binary quadratic programming problem. *European Journal of Operational Research* **134** 103–119.

Katayama, Kengo, Masafumi Tani, Hiroyuki Narihisa. 2000. Solving large binary quadratic programming problems by effective genetic local search algorithm. *GECCO*. 643–650.

Koch, Thorsten, Alexander Martin, Stefan Voß. 2001. SteinLib: An updated library on Steiner tree problems in graphs. Xiu Zhen Cheng, Ding-Zhu Du, eds., *Steiner Trees in Industry*, *Combinatorial Optimization*, vol. 11. Springer US, 285–325.

Laguna, Manuel, Abraham Duarte, Rafael Martí. 2009. Hybridizing the cross-entropy method: An application to the max-cut problem. *Computers & Operations Research* **36** 487–498.

Lodi, Andrea, Kim Allemand, Thomas M Liebling. 1999. An evolutionary heuristic for quadratic 0–1 programming. *European Journal of Operational Research* **119** 662–670.

Lü, Zhipeng, Fred Glover, Jin-Kao Hao. 2010. A hybrid metaheuristic approach to solving the UBQP problem. *European Journal of Operational Research* **207** 1254–1262.

Merz, Peter, Bernd Freisleben. 1999. Genetic algorithms for binary quadratic programming. *Proceedings of the genetic and evolutionary computation conference*, vol. 1. Citeseer, 417–424.

Merz, Peter, Bernd Freisleben. 2002. Greedy and local search heuristics for unconstrained binary quadratic programming. *Journal of Heuristics* **8** 197–213.

Merz, Peter, Kengo Katayama. 2004. Memetic algorithms for the unconstrained binary quadratic programming problem. *BioSystems* **78** 99–118.

Palubeckis, Gintaras. 2004. Multistart tabu search strategies for the unconstrained binary quadratic optimization problem. *Annals of Operations Research* **131** 259–282.

Palubeckis, Gintaras. 2006. Iterated tabu search for the unconstrained binary quadratic optimization problem. *Informatica* **17** 279–296.

Pardalos, Panos M, Oleg A Prokopyev, Oleg V Shylo, Vladimir P Shylo. 2008. Global equilibrium search applied to the unconstrained binary quadratic optimization problem. *Optimisation Methods and Software* **23** 129–140.

Reinelt, Gerhard. 1991. TSPLIB—a traveling salesman problem library. *ORSA Journal on Computing* **3** 376–384.

Rinaldi, G. 1996. RUDY: A generator for random graphs. `http://web.stanford.edu/~yyye/yyye/Gset/rudy.c`. Accessed: 2014-09-30.

Watts, Duncan J, Steven H Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* **393** 440–442.

Welsh, D. J. A., M. B. Powell. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal* **10** 85–86.

Wiegele, Angelika. 2007. Biq Mac library — a collection of Max-Cut and quadratic 0-1 programming instances of medium size. Tech. rep., Alpen-Adria-Universität Klagenfurt.