Quantifying the Similarity of Algorithm Configurations

Lin Xu[1], Ashiqur R. KhudaBukhsh[2], Holger H. Hoos[1(✉)], and Kevin Leyton-Brown[1(✉)]

[1] University of British Columbia, Vancouver, BC, Canada
{xulin730,hhoos,kevinlb}@cs.ubc.ca
[2] Carnegie Mellon University, Pittsburgh, PA, USA
akhudabu@cs.cmu.edu

Abstract. A natural way of attacking a new, computationally challenging problem is to find a novel way of combining design elements introduced in existing algorithms. For example, this approach was made systematic in `SATenstein` [15], a highly parameterized stochastic local search (SLS) framework for SAT that unifies techniques across a wide range of well-known SLS solvers. The focus of such work so far has been on building frameworks and identifying high-performing configurations. Here, we focus on analyzing such frameworks, a problem that currently requires considerable manual effort and domain expertise. We propose a quantitative alternative: a new metric that measures the similarity between a new configuration and previously known algorithm designs. We first introduce concept DAGs, a data structure that preserves the hierarchical structure of configurations induced by conditional parameter dependencies. We then quantify the degree of similarity between two configurations as the transformation cost between the respective concept DAGs. In the context of analyzing `SATenstein` configurations, we demonstrate that visualizations based on transformation costs can provide useful insights into the similarities and differences between existing SLS-based SAT solvers and novel solver configurations.

Keywords: SAT · Stochastic local search · Algorithm configuration similarity

1 Introduction

When faced with a new, computationally hard problem to solve, researchers do not typically want to reinvent the wheel. Instead, it makes sense to draw on design ideas from existing high-performance solvers. Such an approach can be made systematic by designing a single, highly parameterized solver that incorporates these different ideas, and then identifying a parameter configuration that achieves good performance via an automatic algorithm configuration method

Lin Xu and Ashiqur R. KhudaBukhsh contributed equally to this work.

P. Festa et al. (Eds.): LION 2016, LNCS 10079, pp. 203–217, 2016.
DOI: 10.1007/978-3-319-50349-3_14

[9,15]. Indeed, many powerful configuration procedures have recently become available to meet this challenge [10,11,14,20]. The types of solvers configured in this way can range from simple heuristic switching [30] to a complex combination of multiple algorithms [28]. While the result is often an algorithm with excellent performance characteristics, it can be difficult to understand such an algorithm, e.g., in terms of how similar (or dissimilar) it is to the existing solvers from which design ideas were drawn—a problem that has received little attention to date by the research community. This work seeks to address this gap. We propose a new metric for quantitatively assessing the similarity between configurations for highly parametric solvers, which computes the distance between two algorithm configurations in two steps. In the first step, the hierarchical structure of algorithm parameters is represented by a novel data structure called a concept DAG. In the second step, we estimate the similarity of two configurations as the transformation cost from one configuration to another, using concept DAGs.

In order to demonstrate the effectiveness of our approach, we investigate the configurations of `SATenstein`, a well-known, highly parameterized SLS solver. `SATenstein` has a rich and complex design space with 43 parameters, drawing design ideas from several existing solvers, and is one of the most complex SLS solvers in the literature. We show that visualizations based on transformation costs can provide useful insights into similarities and differences between solver configurations. In addition, we argue that this metric can help to suggest potential links between algorithm structure and algorithm performance.

To our knowledge, there is little previous work directly relevant to the problem of quantifying the similarity of algorithm configurations. Visualization techniques have been used previously to characterize the structure of instances of the well-known propositional satisfiability problem (SAT) [26]; instead, we focus on algorithm design elements. Most similar to our work, Nikolić et al. [21] used the notion of edit distance to automatically quantify algorithm similarity. Our main innovation is to address hierarchies of conditional parameters by saying that edits to lower-level parameters are less significant than edits to higher-level parameters. Conditional parameters are increasingly important as algorithm development shifts to rely on algorithm configuration tools and hence parameter spaces become richer and more complex; see e.g., recent work on assessing parameter importance [13] and finding critical parameters [4].

The remainder of this paper is organized as follows. We present a high-level description of `SATenstein` in Sect. 2. Next, we describe concept DAGs (Sect. 3) and then present our experimental setup (Sect. 4). We describe our results on quantifying similarities between algorithm configurations in Sect. 5 and then conclude (Sect. 6).

2 SATenstein

In this section, we provide a short description of the overall design of `SATenstein`. A detailed description of `SATenstein` is given in [16]. As shown in the high-level algorithm outline, any instantiation of `SATenstein` proceeds as follows:

1. Optionally execute $B1$, which performs search diversification.
2. Execute either $B2$, $B3$ or $B4$, thus performing WalkSAT-based local search, dynamic local search or G^2WSAT-based local search, respectively.
3. Optionally execute $B5$ to update data structures such as promising list, clause penalties, dynamically adaptable parameters or tabu attributes.

`SATenstein` consists of five building blocks and eight components, some of which are shared across different building blocks. It has 43 parameters in total. The choice of building block is encoded by several high-level categorical parameters, while the design strategies within each component are determined by a larger number of low-level parameters.

3 Concept DAGs

We now introduce *concept DAGs*, a novel data structure for representing algorithm configurations that preserves the hierarchical structure of parameter dependencies. Our notion of a concept DAG is based on that of a concept tree [31]. We work with a DAG-based data structure because parameters may have more than one parent, where the child is only active if the parents take certain values (e.g., SATenstein's noise parameter *phi* is only activated when both *useAdaptiveMechanism* and *singleClauseAsNeighbor* are turned on). We then define four operators whose repeated application can be used to map between arbitrary concept DAGs, and assign each operator a cost. To compare two parameter configurations, we first represent them using concept DAGs and then define their similarity as the minimal total cost of transforming one DAG into the other.

A *concept DAG* is a six-tuple $G = (V, E, L^V, R, D, M)$, where V is a set of nodes, E is a set of directed edges between the nodes in V such that (V, E) is an acyclic graph, L^V is a set of lexicons (terms) for concepts used as node labels, R is a distinguished node called the root, D is the domain of discourse (i. e., the set of all possible node labels), and M is an injective mapping from V to L^V that assigns a unique label to every node. A parameter configuration can be expressed as a concept DAG in which each node in V represents a parameter, and each directed edge in E represents the conditional dependence relationship between two parameters. L^V is the set of parameter values used in a particular configuration (i. e., a set containing exactly one value from the domain of each parameter), D is the union of the domains of all parameters, and M specifies which value assigned to each parameter $v \in V$ in the given configuration. We add an artificial root node R, which connects to all parameter nodes that do not have any parent, and refer to these parameters as *top-level parameters*.

We can transform one concept DAG into another by a series of delete, insert, relabel and move operations, each of which has an associated cost. For measuring the degree of similarity between two algorithm configurations, we first express them as concept DAGs, DAG_1 and DAG_2. We define the distance between these DAGs as the minimal total cost required for transforming DAG_1 into DAG_2. Obviously, the distance between two identical configurations is 0.

```
    Input: CNF formula φ; real number cutoff;
           Booleans performDiversification, singleClauseAsNeighbor,
                    usePromisingList;
    Output: Satisfying variable assignment
    Start with random assignment A;
    Initialize parameters;
    while runtime < cutoff do
        if A satisfies φ then
            return A;
        end
        varFlipped ← FALSE;
        if performDiversification then
B1          with probability diversificationProbability() do
B1              c ← selectClause();
B1              y ← diversificationStrategy(c) ;
B1              varFlipped ← TRUE;
        end
        if not varFlipped then
            if not usePromisingList then
                if singleClauseAsNeighbor then
B2                  c ← selectClause();
B2                  y ← selectHeuristic(c) ;
                else
B3                  sety ← selectSet();
B3                  y ← tieBreaking(sety);
                end
            else
B4              if promisingList is not empty then
B4                  y ← selectFromPromisingList() ;
                else
B4                  c ← selectClause();
B4                  y ← selectHeuristic(c) ;
                end
            end
            flip y ;
B5          update();
        end
    end
```

Procedure. SATenstein(...)

The parameters with the biggest impact on an algorithm's execution path are likely to appear high in the DAG (i.e., to be conditional upon few or no other parameters) and/or to turn on a complex mechanism (i.e., to have many parameters conditional upon them). Therefore, we say that the importance of a parameter v is a function of its depth (the length of the longest path from the root R of the given concept DAG to v) and the total number of other parameters

conditional on it. To capture this definition of importance, we define the cost of each of the four DAG-transforming operations as follows.

Deletion cost $C(\text{delete}(v)) = \frac{1}{|V|} \cdot (height(DAG) - depth(v) + 1 + |DE(v)|)$, where $height(DAG)$ is the height of the DAG, $depth(v)$ is the depth of node v and $DE(v)$ is the set of descendants of node v. This captures the idea that it is more costly to delete top-level parameters and parameters that (de-)activate complex mechanisms.

Insertion cost $C(\text{insert}(u, v)) = \frac{1}{|V|} \cdot (height(DAG) - depth(u) + 1 + |DE(v)|)$, where $DE(v)$ is the set of descendants of v after the insertion, and u is the node under which v is inserted.

Moving cost $C(\text{move}(u, v)) = \frac{|V|-2}{2 \cdot |V|} \cdot [C(\text{delete}(v)) + C(\text{insert}(u, v))]$, where $|V| > 2$.

Relabelling cost $C(\text{relabel}(v, l^v, l^{v^*}) = [C(\text{delete}(v)) + C(\text{insert}(u, v))] \cdot s(l^v, l^{v^*})$, where u is the parent node of v and $s(l^v, l^{v^*})$ is a measure of the distance between the old label, l^v, and the new label, l^{v^*}, of node v. For parameters with continuous domains, $s(l^v, l^{v^*}) = |l^v - l^{v^*}|$. For parameters whose domains are some finite, ordinal and discrete set $\{l^{v_1}, l^{v_2}, \ldots, l^{v_k}\}$, $s(l^v, l^{v^*}) = \text{abs}(v - v^*)/(k - 1)$, where $\text{abs}(v - v^*)$ measures the number of intermediate values between v and v^*. For categorical parameters, $s(l^v, l^{v^*}) = 0$ if $l^v = l^{v^*}$ and 1 otherwise. Since ParamILS, the algorithm configurator we used, requires discrete parameter domains, all our parameters are either categorical or have finite, ordinal and discrete domains; therefore, $s(l^v, l^{v^*})$ is always bounded between [0, 1].

In our implementation, we did not use the move operator, because the structure of `SATenstein`'s parameter space does not provide much scope for its application. Also, instead of finding the minimal transformation cost over all sequences of delete and insert operations (a potentially expensive computation), we used an easily implemented, yet effective stochastic local search procedure to produce upper bounds. This procedure is based on randomised iterative first improvement (with a random walk probability of 0.01); starting from a permutation of the delete and insert operations required to transform the first concept DAG into the second that is chosen uniformly at random, it swaps two operations in each step. The search process is restarted whenever no improvement in the best transformation cost seen so far has been obtained within 200 iterations and terminates upon the 10th such restart. For each sequence of inserts and deletes, it is straightforward to compute the transformation cost that also takes into account the corresponding relabelling operations.

4 Experimental Setup

Our quantitative analysis of `SATenstein` configurations is based on performance comparisons with eleven high-performance SLS solvers on six well-known SAT distributions, listed in Table 1 (we call each of these solvers a challenger) and Table 2, respectively.

Table 1. Our eleven challenger algorithms.

Algorithm	Abbrev	Reason for inclusion	Parameters
`Ranov` [22]	Ranov	gold 2005 SAT Competition (random)	wp
`G`^{2}`WSAT` [17]	G2	silver 2005 SAT Competition (random)	novNoise, dp
`VW` [24]	VW	bronze 2005 SAT Competition (random)	c, s, wpWalk
`gNovelty`$^+$ [23]	GNOV	gold 2007 SAT Competition (random)	novNoise, wpWalk, ps
`adaptG`^{2}WSAT$_0$ [18]	AG20	silver 2007 SAT Competition (random)	NA
`adaptG`^{2}WSAT$_+$ [19]	AG2+	bronze 2007 SAT Competition (random)	NA
`adaptNovelty`$^+$ [8]	ANOV	gold 2004 SAT Competition (random)	wp
`adaptG`^{2}`WSAT`$_p$ [19]	AG2p	performance comparable to `G`^{2}`WSAT` [17], `Ranov`, and `adaptG`^{2}`WSAT`$_+$; see [18]	NA
`SAPS` [12]	SAPS	prominent DLS algorithm	alpha, ps, rho, sapsthresh, wp
`RSAPS` [12]	RSAPS	prominent DLS algorithm	alpha, ps, rho, sapsthresh, wp
`PAWS` [27]	PAWS	prominent DLS algorithm	maxinc, pflat

Table 2. Our six benchmark distributions.

Distribution	Description	Generator parameters	Train/Test size
QCP	SAT-encoded quasi-group completion problems [6]	order $O \in [10, 30]$; holes $H = h * O^{1.55}$, $h \in [1.2, 2.2]$	1000/1000
SW-GCP	SAT-encoded small-world graph-colouring problems [5]	ring lattice size $S \in [100, 400]$; nearest neighbors connected: 10; rewiring probability: 2^{-7}; chromatic numbers: 6	1000/1000
R3SAT	uniform-random 3-SAT instances [25]	variable: 600; clauses-to-variables ratio: 4.26	250/250
HGEN	random instances generated by HGEN2 [7]	variable $n \in [200, 400]$	1000/1000
FAC	SAT-encoded factoring problems [29]	prime number $\in [3000, 4000]$	1000/1000
CBMC(SE)	SAT-encoded bounded model checking [1], preprocessed by SatELite [3]	array size $s \in [1, 2000]$; loop unwinding $n \in 4, 5, 6$	302/302

We performed algorithm configuration using ParamILS [11], a well-known automatic algorithm configurator. On each benchmark distribution, we configured `SATenstein` on the training set, and evaluated its performance of the

configuration on the test set. For each test set instance, we ran each solver 25 times with a per-run cutoff of 600 CPU seconds. Following [11], we evaluate performance in terms of penalized average run time (PAR), which is defined as average run time with each timed out run counted as having completed in 10 times the cutoff time (in this case, 6000 CPU seconds). For a particular solver, we consider an instance solved if a majority of runs found a satisfying assignment. In practice, PAR can be sensitive to the choice of cutoff; however, in past work [15], we showed that PAR did not affect the qualitative evaluation of `SATenstein`'s performance in all six distributions we considered. We conducted all of our experiments on a cluster of 55 machines each equipped with dual 3.2 GHz Intel Xeon CPUs with 2 MB cache and 2 GB RAM, running OpenSuSE Linux 11.1 and managed by Sun Grid Engine (version 6.0). Code for `SATenstein` and our transformation cost computation can be found at http://www.cs.ubc.ca/labs/beta/Projects/SATenstein/.

5 Quantitative Comparison of Algorithm Configurations

In previous work, we performed an extensive performance evaluation on six well-known benchmark distributions, finding that `SATenstein` outperformed all challengers in every distribution [16]. Moreover, we found that `SATenstein` outperformed tuned challengers as well, albeit to a reduced extent. In order to refer to them in what follows, we summarize these results in Tables 4 and 5.

Table 3 gives a high-level description of `SATenstein` solvers in terms of building blocks used and overall SLS category. Recall that `SATenstein` draws components from three major SLS solver categories: WalkSAT, dynamic local search and G^2WSAT-based algorithms.

5.1 Comparison of `SATenstein` Configurations

We now compare our automatically identified `SATenstein` solver designs to all of the challengers. As shown in Table 1, 3 of our 11 challengers (`AG2p`, `AG2+`, and `AG20`) are parameter-less. Furthermore, `RANOV` only differs from `ANOV` by the addition of a preprocessing step, and so can be understood as a variant of the

Table 3. High-level summary of `SATenstein` solvers.

Solver	Uses building blocks	Broad category
`SATenstein[QCP]`	1, 2 and 5	WalkSAT
`SATenstein[SW-GCP]`	2 and 5	WalkSAT
`SATenstein[R3SAT]`	1, 3 and 5	Dynamic local search
`SATenstein[HGEN]`	1, 2 and 5	WalkSAT
`SATenstein[FAC]`	3 and 5	Dynamic local search
`SATenstein[CBMC(SE)]`	1, 3 and 5	Dynamic local search

Table 4. Performance of `SATenstein` and the 11 challengers. Every algorithm was run 25 times on each instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm i on distribution j as a/b, where a (top) is the penalized average runtime; b (bottom) is the percentage of instances solved (i.e., those with median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold, and the best-scoring challenger(s) are underlined.

SATenstein[D] [15]	0.08 100%	0.03 100%	1.11 100%	0.02 100%	10.89 100%	4.75 100%
Solvers	QCP	SW-GCP	R3SAT	HGEN	FAC	CBMC(SE)
AG20 [18]	1054.99	0.64	2.14	137.02	3594.40	2169.77
	81.2%	**100%**	**100%**	98.1%	35.9%	61.1%
AG2p [19]	1119.96	0.43	2.35	105.30	1954.83	2294.24
	80.1%	**100%**	**100%**	98.4%	80.6%	61.1%
AG2+ [19]	1091.37	0.67	3.04	148.28	1450.89	2181.92
	80.3%	**100%**	**100%**	98.0%	91.0%	61.1%
ANOV [8]	25.42	4.86	11.17	109.94	2897.52	2021.22
	99.6%	**100%**	**100%**	98.6%	51.4%	61.1%
G2 [17]	2942.13	4092.29	3.69	104.55	5947.80	2139.12
	50.9%	31.0%	**100%**	98.7%	0%	65.4%
GNOV [23]	414.69	1.20	11.14	52.58	5935.39	2236.85
	93.3%	**100%**	**100%**	99.4%	0%	61.5%
PAWS [27]	1127.84	4495.50	1.77	62.18	22.05	1693.82
	81.0%	24.3%	**100%**	99.4%	**100%**	70.8%
RANOV [22]	73.38	0.15	18.29	151.11	887.33	1227.07
	99.1%	**100%**	**100%**	98.2%	96.8%	79.7%
RSAPS [12]	1255.94	5635.54	18.42	33.28	17.86	827.81
	79.2%	5.4%	**100%**	99.7%	**100%**	85.0%
SAPS [12]	1248.34	3864.74	22.93	40.17	16.41	646.89
	79.4%	34.2%	**100%**	99.5%	**100%**	89.7%
VW [24]	1022.69	161.74	12.45	176.18	3382.02	385.12
	81.9%	99.4%	**100%**	97.8%	35.3%	93.4%

same algorithm. This leaves us with 7 parameterized challengers to consider. For each, we sampled 50 configurations (consisting of the default configuration, one configuration optimized for each of our 6 benchmark distributions, and 43 random configurations). We then computed the pairwise transformation cost between the resulting 359 configurations (7×50 challengers' configurations + 6 `SATenstein` solvers + `AG2p` + `AG2+` + `AG20`). The result can be understood as a graph with 359 nodes and 128 522 edges, with nodes corresponding to concept DAGs, and edges labeled by the minimum transformation cost between them. To visualize this graph, we used a dimensionality reduction method to map it onto a plane, with the aim of positioning points so that the Euclidean distance between every pair of points approximates their transformation cost as accurately as possible; in particular, we used the `MDS` algorithm [2] as implemented in MATLAB's *mdscale* function, with option *sstress*.

Table 5. Performance summary of the automatically configured versions of 8 challengers (three challengers have no parameters). Every algorithm was run 25 times on each problem instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm i on distribution j as a/b, where a (top) is the penalized average runtime; b (bottom) is the percentage of instances solved (i.e., having median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold.

Solvers	QCP	SW-GCP	R3SAT	HGEN	FAC	CBMC(SE)
ANOV[D] [8]	26.13	0.06	2.68	119.75	1731.16	994.94
	99.6%	**100%**	**100%**	98.2%	90.1%	83.4%
G2[D] [17]	514.29	**0.05**	3.64	98.70	617.83	1084.60
	91.4%	**100%**	**100%**	99.1%	97.8%	81.4%
GNOV[D] [23]	417.33	0.22	8.87	68.24	5478.75	2195.76
	92.9%	**100%**	**100%**	99.4%	0.3%	61.8%
PAWS[D] [27]	68.06	0.70	1.91	64.48	22.01	1925.56
	99.2%	**100%**	**100%**	99.4%	**100%**	67.7%
RANOV[D] [22]	75.06	0.15	13.85	141.61	336.27	1223.83
	98.9%	**100%**	**100%**	98.1%	100%	80.4%
RSAPS[D] [12]	868.37	0.19	1.32	42.99	12.17	67.59
	85.2%	**100%**	**100%**	99.5%	**100%**	99.0%
SAPS[D] [12]	27.69	0.31	1.54	**31.77**	**10.68**	62.63
	99.8%	**100%**	**100%**	99.6%	**100%**	99.0%
VW[D][24]	**0.33**	417.71	**1.26**	57.44	32.38	**16.45**
	100%	94.8%	**100%**	99.6%	100%	**100%**

The final layout of similarities among 359 configurations (16 algorithms) is shown in Fig. 1. Observe that in most cases the 50 different configurations for a given challenger solver were so similar that they mapped to virtually the same point in the graph.

As noted earlier, the distance between any two configurations shown in Fig. 1 only approximates their true distance. In addition, the result of the visualization also depends on the number of configurations considered: adding an additional configuration may affect the position of many or all other configurations. Thus, before drawing further conclusions about the results illustrated in Fig. 1, we validated the fidelity of the visualization to the original distance data. As can be seen from Fig. 2, there was a strong correlation between the computed and mapped distances (Pearson correlation coefficient: 0.96). Also, the mapping preserved the relative ordering of the true distances between configurations quite well (Spearman correlation coefficient 0.96)—in other words, distances that appear similar in the 2D plot tend to correspond to similar true distances (and vice versa). Digging deeper, we confirmed that the top two closest challengers in Fig. 1 to each given `SATenstein` were always the ones having the lowest true

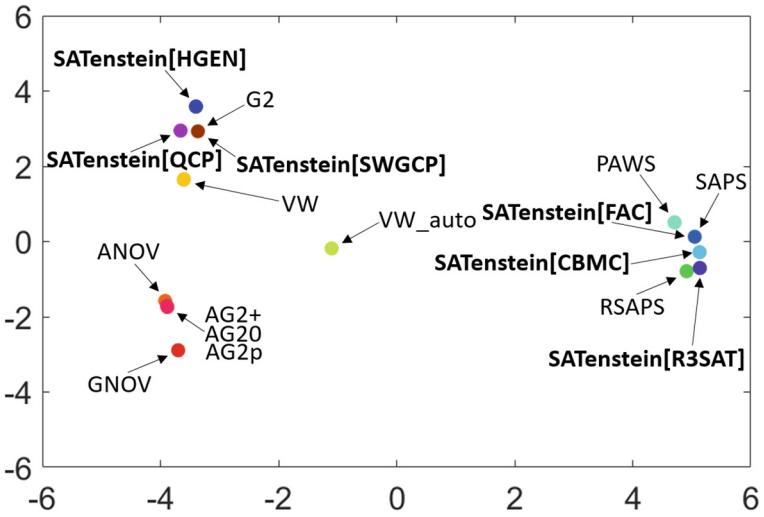

Fig. 1. Visualization of the transformation costs in the design of 16 high-performance solvers (359 configurations) obtained via `MDS`.

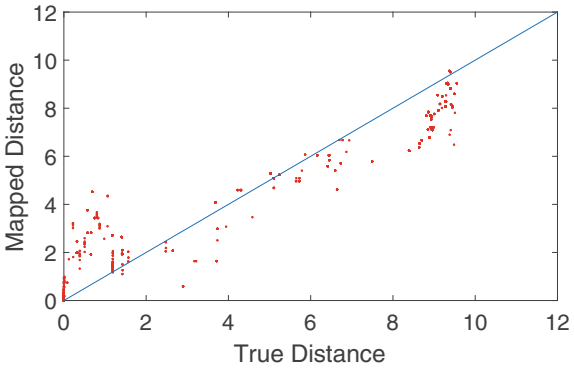

Fig. 2. True *vs* mapped distances in Fig. 1. The data points correspond to the complete set of `SATenstein`[D] for all domains and all challengers with their default and domain-specific, optimized configurations.

transformation costs. For distant challengers, relative distance in the visualization did not always reflect true relative transformation costs; however, we find this acceptable, since we are mainly interested in examining which configurations are similar to each other.

Having confirmed that our dimensionality reduction method is performing reliably, let us examine Fig. 1 in more detail. Overall, and unsurprisingly, we first note that the transformation cost between two configurations in the design space is very weakly related to their performance difference (quantitatively, the Spearman correlation coefficient between performance difference (PAR ratio) and

configuration difference (transformation cost) was 0.25). Examining algorithms by type, we note that all dynamic local search algorithms are grouped together, on the right side of Fig. 1; likewise, the algorithms using adaptive mechanisms are grouped together at the bottom-left of Fig. 1. `SATenstein()` solvers were typically more similar to each other than to challengers, and fell into two broad clusters. The first cluster also includes the `SAPS` variants (`SAPS`, `RSAPS`), while the second also includes `G2` and `VW`. None of the `SATenstein` solvers uses an adaptive mechanism to automatically adjust other parameters. In fact, as shown in Table 5, the same is true of the best performance-optimized challengers as neither `SAPS`, `G2`, or `VW` use adaptive mechanism. This suggests that in many cases, contrary to common belief (see, e.g., [8,19]) it may be preferable to expose parameters so they can be instantiated by sophisticated configurators rather than automatically adjusting them at running time using a simple adaptive mechanism.

We now consider benchmarks individually. For the `FAC` benchmark, `SATenstein[FAC]` had similar performance to `SAPS[FAC]`; as seen in Fig. 1, both solvers are structurally very similar as well. Overall, for the 'industrial' distributions, `CBMC(SE)` and `FAC`, dynamic local search algorithms often yielded the best performance amongst all challengers. Our automatically-constructed `SATenstein` solvers for these two distributions are also dynamic local search algorithms. Due to the larger search neighbourhood and the use of clause penalties, dynamic local search algorithms are more suitable for solving industrial SAT instances, which often have some special global structure.

For `R3SAT`, a well-studied distribution, many challengers showed good performance (the top three challengers were `VW`, `RSAPS`, and `SAPS`). The performance of `SATenstein[R3SAT]` is only slightly better than that of `VW[R3SAT]`. Figure 1 shows that `SATenstein[R3SAT]` is a dynamic local search algorithm similar to `RSAPS` and `SAPS`.

For `HGEN`, even the best performance-optimized challengers, `RSAPS[HGEN]` and `SAPS[HGEN]`, performed poorly. `SATenstein[HGEN]` achieves more than 1 000-fold speedups against all challengers. Its configuration is far away from any dynamic local search algorithm (the best challengers), and closest to `VW`, a Walk-SAT algorithm, and `G2`.

For `QCP`, `VW[QCP]` does not reach the performance of `SATenstein[QCP]`, but significantly outperforms all other challengers. Our transformation cost analysis shows that `VW` is the closest neighbour to `SATenstein[QCP]`. For `SWGCP`, many challengers achieve similar performance to `SATenstein[SWGCP]`. Figure 1 shows that `SATenstein[SWGCP]` is close to `G2[SWGCP]`, which is the best performing challenger on `SWGCP`.

5.2 Comparison to Configured Challengers

Since there were large performance gaps between default and configured challengers, we were also interested in the transformation cost between the configurations of individual challenger solvers. Table 5 shows that after configuring each challenger for each distribution, we found that `SAPS` was best on `HGEN` and `FAC`;

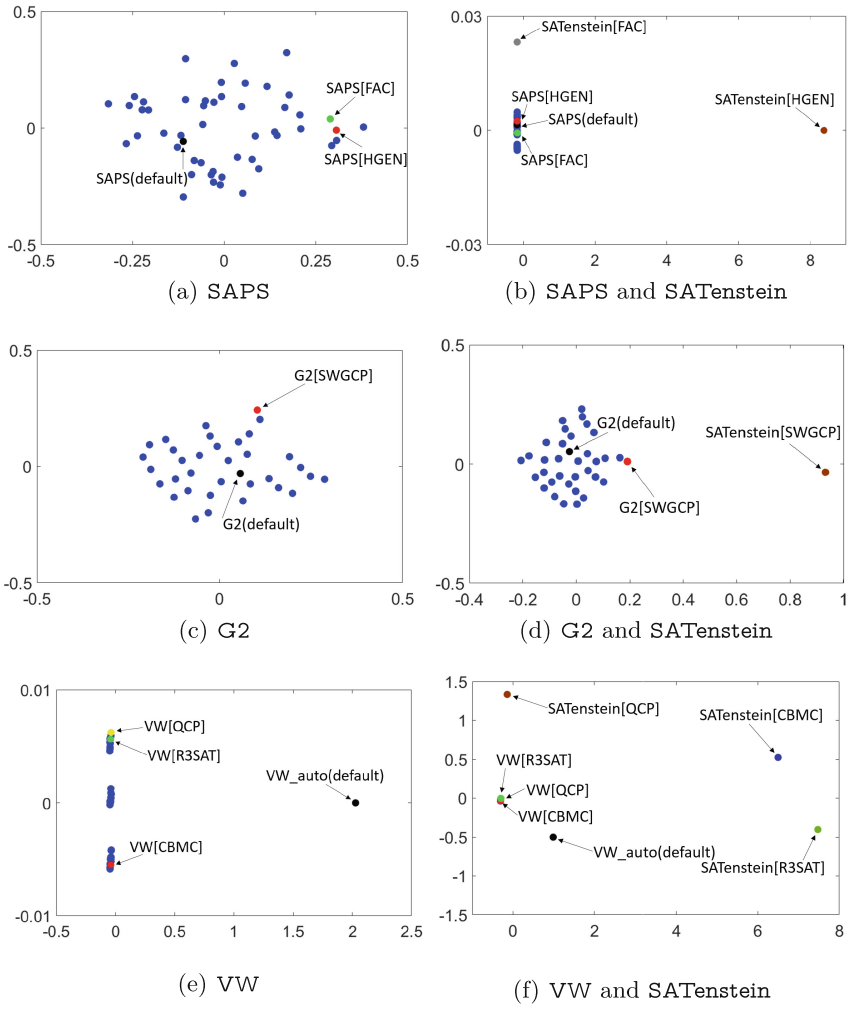

Fig. 3. The transformation costs of configuration of individual challengers and selected `SATenstein` solvers. (a): `SAPS` (best on `HGEN` and `FAC`); (b): `SAPS` and `SATenstein[HGEN, FAC]`; (c): `G2` (best on `SWGCP`); (d): `G2` and `SATenstein[SWGCP]`; (e): `VW` (best on `CBMC(SE)`, `QCP`, and `R3SAT`); (f): `VW` and `SATenstein[CBMC, QCP, R3SAT]`.

`G2` was best on `SWGCP`, and `VW` was best on `CBMC(SE)`, `QCP`, and `R3FIX`. Figure 3 (left) visualizes the parameter spaces for each of these three solvers (43 random configurations + default configuration + 6 optimized configurations). Figure 3 (right) shows the same thing, but also adds the best `SATenstein()` configurations for each benchmark on which the challenger exhibited top performance.

Examining these figures in the left column of Fig. 3, we first note that the `SAPS` configurations optimized for `FAC` and `HGEN` are very similar but differ

substantially from SAPS's default configuration. On SWGCP, the optimized configuration of G2 not only performs much better than the default but, as seen in Fig. 3(c), is also quite different. All three top-performing VW configurations are rather different from VW's default, and none of them uses the adaptive mechanism for choosing parameter *wpWalk*, *s*, and *c*. Since the parameter useAdaptiveMechanism is a top-level parameter and many other parameters are conditionally dependent on it, the transformation costs between VW default and optimized configurations of VW are very large, due to the high relabelling cost for these nodes in our concept DAGs.

The right column of Fig. 3 illustrates the similarity between optimized SATenstein() solvers and the best performing challenger for each benchmark. As previously noted, SATenstein[FAC] and SAPS[FAC] are not only very similar in performance, but also structurally similar. Likewise, SATenstein[SWGCP] is similar to G2[SWGCP]. On R3SAT, many challengers had similar performance. SATenstein[R3SAT] (PAR = 1.11) was quite different from the best challenger VW[R3SAT] (PAR = 1.26), but resembled SAPS[R3SAT] (PAR = 1.53). For the three remaining benchmarks, SATenstein() solvers exhibited much better performance than the best optimized challengers, and their configurations likewise differed substantially from the challengers' configurations.

As an aside, it might initially be surprising that qualitative features of the visualizations in Fig. 3 appear to be absent from Fig. 1. In particular, the sets of randomly sampled challenger configurations that are quite well-separated in Fig. 3 are nearly collapsed into single points in Fig. 1. The reason for this lies in the fact that the 2D-mapping of the highly non-planar pairwise distance data performed by MDS focuses on minimal overall distortion. For example, when visualizing the differences within a set of randomly sampled SAPS configurations (Fig. 3(a)), MDS spreads these out into a cloud of points to represent their differences. However, the presence of a single SATenstein configuration that has large transformation costs from all of these SAPS configurations forces MDS to use one dimension to capture those differences, leaving essentially only one dimension to represent the much smaller differences between the SAPS configurations (Fig. 3(b)). Adding further very different configurations (as present in Fig. 1) leads to mappings in which the smaller differences between configurations of the same challenger become insignificant.

6 Conclusion

We have proposed a new metric for quantitatively assessing the similarity between configurations of highly parametric solvers. Our metric is based on a data structure, concept DAGs, that preserves the internal hierarchical structure of parameters. We estimate the similarity of two configurations as the transformation cost from one configuration to another. In the context of SATenstein, a highly parameterized SLS-based SAT solver, we have demonstrated that visualizations based on transformation cost can provide useful insights into similarities and differences between solver configurations. In addition, we believe that this

metric could be useful for suggesting potential links between algorithm structure and algorithm performance further exploration of which could be an interesting future research direction.

References

1. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24730-2_15
2. Cox, T.F., Cox, M.A.: Multidimensional scaling. CRC Press, Boca Raton (2000)
3. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). doi:10.1007/11499107_5
4. Fawcett, C., Hoos, H.H.: Analysing differences between algorithm configurations through ablation. J. Heuristics **22**, 1–28 (2013)
5. Gent, I.P., Hoos, H.H., Prosser, P., Walsh, T.: Morphing: combining structure and randomness. In: Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI 1999), pp. 654–660 (1999)
6. Gomes, C.P., Selman, B.: Problem structure in the presence of perturbations. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997), pp. 221–226 (1997)
7. Hirsch, E.A.: Random generator hgen2 of satisfiable formulas in 3-CNF (2002). http://logic.pdmi.ras.ru/~hirsch/benchmarks/hgen2-1.01.tar.gz. Accessed 18 December 2015
8. Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI 2002), pp. 655–660 (2002)
9. Hoos, H.H.: Programming by optimization. Commun. ACM **55**(2), 70–80 (2012)
10. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) LION 2011. LNCS, vol. 6683, pp. 507–523. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25566-3_40
11. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. (JAIR) **36**(1), 267–306 (2009)
12. Hutter, F., Tompkins, D.A.D., Hoos, H.H.: Scaling and probabilistic smoothing: efficient dynamic local search for SAT. In: Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 233–248. Springer, Heidelberg (2002). doi:10.1007/3-540-46135-3_16
13. Hutter, F., Hoos, H., Leyton-Brown, K.: An efficient approach for assessing hyperparameter importance. In: Proceedings of the 31st International Conference on Machine Learning (ICML 2014), pp. 754–762 (2014)
14. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC - instance specific algorithm configuration. Eur. Conf. Artif. Intell. (ECAI) **2010**, 751–756 (2010)
15. KhudaBukhsh, A.R., Xu, L., Hoos, H.H., Leyton-Brown, K.: SATenstein: automatically building local search SAT solvers from components. In: Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 517–524 (2009)
16. KhudaBukhsh, A.R., Xu, L., Hoos, H.H., Leyton-Brown, K.: Satenstein: automatically building local search sat solvers from components. Artif. Intell. **232**, 20–42 (2016)

17. Li, C.M., Huang, W.Q.: Diversification and determinism in local search for satisfiability. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 158–172. Springer, Heidelberg (2005). doi:10.1007/11499107_12
18. Li, C.M., Wei, W., Zhang, H.: Combining adaptive noise and promising decreasing variables in local search for SAT (2007). Solver description, SAT competition 2007
19. Li, C.M., Wei, W., Zhang, H.: Combining adaptive noise and look-ahead in local search for SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 121–133. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72788-0_15
20. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm portfolios based on cost-sensitive hierarchical clustering. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, pp. 608–614. AAAI Press (2013)
21. Nikolić, M., Marić, F., Janičić, P.: Instance-based selection of policies for SAT solvers. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 326–340. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02777-2_31
22. Pham, D.N., Anbulagan, A.: Resolution enhanced SLS solver: R+AdaptNovelty+ (2007). Solver description, SAT competition 2007
23. Pham, D.N., Thornton, J., Gretton, C., Sattar, A.: Combining adaptive and dynamic local search for satisfiability. J. Satisf. Boolean Model. Comput. (JSAT) **4**, 149–172 (2008)
24. Prestwich, S.: Random walk with continuously smoothed variable weights. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 203–215. Springer, Heidelberg (2005). doi:10.1007/11499107_15
25. Simon, L.: SAT competition random 3CNF generator (2002). www.satcompetition.org/2003/TOOLBOX/genAlea.c. Accessed 18 December 2015
26. Sinz, C.: Visualizing the internal structure of sat instances (preliminary report). In: SAT. Citeseer (2004)
27. Thornton, J., Pham, D.N., Bain, S., Ferreira, V.: Additive versus multiplicative clause weighting for SAT. In: Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI 2004), pp. 191–196 (2004)
28. Tompkins, D.A.D., Balint, A., Hoos, H.H.: Captain Jack: new variable selection heuristics in local search for SAT. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 302–316. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21581-0_24
29. Uchida, T., Watanabe, O.: Hard SAT instance generation based on the factorization problem (1999). http://www.is.titech.ac.jp/~watanabe/gensat/a2/GenAll.tar.gz
30. Wei, W., Li, C.M., Zhang, H.: A switching criterion for intensification, and diversification in local search for sat. J. Satisf. Boolean Model. Comput. **4**, 219–237 (2008)
31. Xue, Y., Wang, C., Ghenniwa, H., Shen, W.: A tree similarity measuring method and its application to ontology. J. Univ. Comput. Sci. **15**(9), 1766–1781 (2001)