Warped Landscapes and Random Acts of SAT Solving

Dave A.D. Tompkins & Holger H. Hoos

Department of Computer Science
University of British Columbia
Canada
Outline

1. Dynamic Local Search (DLS) for SAT and MAX-SAT
2. Do DLS Algorithms Learn?
3. Is Randomness Needed?
4. Conclusions & Future Work
Dynamic Local Search (DLS) for (MAX-)SAT

Propositional Satisfiability Problem (SAT):

*Given:* Propositional formula \( \Phi \) in conjunctive normal form.

*Objective:* Find an assignment of truth values to variables in \( \Phi \) such that \( \Phi \) is satisfied, or declare \( \Phi \) as unsatisfiable.

*Example:*

\[(a \lor b) \land (\neg a \lor \neg b)\]

\(\neg\) satisfiable, solution: \( a = \text{true}, b = \text{false} \)
Maximum Propositional Satisfiability Problem (MAX-SAT):

Given: Propositional formula $\Phi$ in conjunctive normal form.

Objective: Find an assignment of truth values to variables in $\Phi$ that maximises the number of satisfied clauses in $\Phi$.

Weighted MAX-SAT:

Given: Propositional formula $\Phi$ in conjunctive normal form, weights $w(c)$ associated with each clause $c \in \Phi$

Objective: Find an assignment of truth values to variables in $\Phi$ that maximises the total weight of satisfied clauses in $\Phi$.

$\sim$ hard vs. soft constraints
Stochastic Local Search (SLS)

**Approach:**

- Guess *(i.e., randomly generate)* initial candidate solution *(SAT: randomly determine truth value for each variable).*

- Iteratively perform *search steps* by modifying small parts of the candidate solution guided by *evaluation function* *(SAT: pick a variable and change its truth value in order to reduce number of unsatisfied clauses).*

- Stop this process when *termination condition* is satisfied, *e.g.*, solution found or time-limit reached.

- Stochastic decisions are used to overcome / avoid search stagnation caused by, *e.g.*, local minima.
Note:

- SLS algorithms are amongst the best-performing methods for solving hard, satisfiable SAT instances.
- SLS algorithms are (by a large margin) the best-performing methods for solving hard MAX-SAT instances.
Dynamic Local Search (DLS)

Key idea: Modify evaluation function during search process to escape from local minima in objective function $g$.

DLS for SAT:

- associate *penalty values* $clp(c)$ with every clause $c$
- initialise clause penalties (typically $clp(c) := 1$)
- perform local search on

$$g'(clp, a) := \sum_{c \text{ is unsat under } a} clp(c)$$

- modify clause penalties (important choices: when? how?)
Dynamic Local Search

solution
Note:

- DLS for SAT effectively finds locally optimal solutions for a series of weighted MAX-SAT instances, where the clause weights correspond to the $clp$ values.

- Many DLS algorithms are motivated by methods from continuous optimisation, but important theoretical properties do not carry over.

- Modifications of clause weights typically have high time complexity compared to local search steps.
Some DLS Algorithms for SAT

- Breakout Method [Morris, 1993]

* GSAT with clause weights [Selman & Kautz, 1993]

- GSAT with rapid weight adjustment [Frank, 1997]

* Discrete Lagrangian Method (DLM) [Wah et al., 1998-2000]

- Smoothed Descent and Flood (SDF) algorithm
  [Schuurmans & Southy, 2000]

* Exponentiated Subgradient (ESG) algorithm
  [Schuurmans et al., 2001]

** Scaling and Probabilistic Smoothing (SAPS) algorithm
  [Hutter, Tompkins, & Hoos, 2002]
Scaling And Probabilistic Smoothing (SAPS)

- Random Init:
  - $cpl(c) := 1$ [all $c$]

- Iterative Best Improvement using clp's
  - $lmin$ PROB(wp)
  - not $lmin$ PROB(1-sp)

- Random Walk
  - PROB(1-wp)

- Scaling:
  - $cpl(c) := a \cdot cpl(c)$
    - $[all \ unsat \ c]$ PROB(wp)

- Smoothing:
  - $cpl(c) := clp(c) \cdot r + \text{avg}(clp(c)) \cdot (1-r)$
    - $[all \ c]$ PROB(sp)
<table>
<thead>
<tr>
<th>Problem Instance</th>
<th>Novelty$^+$</th>
<th>ESG</th>
<th>SAPS</th>
<th>s.f.</th>
</tr>
</thead>
<tbody>
<tr>
<td>uf100-hard</td>
<td>0.046</td>
<td>0.006</td>
<td>0.006</td>
<td>1.00</td>
</tr>
<tr>
<td>uf250-med</td>
<td>0.015</td>
<td>0.0195</td>
<td>0.011</td>
<td>1.36</td>
</tr>
<tr>
<td>uf250-hard</td>
<td>2.745</td>
<td>0.461</td>
<td>0.291</td>
<td>1.58</td>
</tr>
<tr>
<td>uf400-med</td>
<td>0.160</td>
<td>0.324</td>
<td>0.103</td>
<td>1.55</td>
</tr>
<tr>
<td>uf400-hard</td>
<td>22.3</td>
<td>9.763</td>
<td>1.973</td>
<td>4.95</td>
</tr>
<tr>
<td>flat100-med</td>
<td>0.008</td>
<td>0.013</td>
<td>0.008</td>
<td>1.00</td>
</tr>
<tr>
<td>flat100-hard</td>
<td>0.089</td>
<td>0.037</td>
<td>0.032</td>
<td>1.16</td>
</tr>
<tr>
<td>flat200-med</td>
<td>0.208</td>
<td>0.237</td>
<td>0.087</td>
<td>2.39</td>
</tr>
<tr>
<td>flat200-hard</td>
<td>18.862</td>
<td>5.887</td>
<td>3.052</td>
<td>1.93</td>
</tr>
<tr>
<td>bw_large.a</td>
<td>0.014</td>
<td>0.016</td>
<td>0.009</td>
<td>1.56</td>
</tr>
<tr>
<td>bw_large.b</td>
<td>0.339</td>
<td>0.280</td>
<td>0.179</td>
<td>1.56</td>
</tr>
<tr>
<td>logistics.c</td>
<td>0.226</td>
<td>0.229</td>
<td>0.037</td>
<td>6.10</td>
</tr>
<tr>
<td>ais10</td>
<td>4.22</td>
<td>0.139</td>
<td>0.051</td>
<td>2.73</td>
</tr>
</tbody>
</table>
SAPS on MAX-SAT: test-set wjnh
SAPS on MAX-SAT: test-sets rnd100-1000u, rnd150-1500u
Do DLS Algorithms Learn?

Original motivation of DLS:

- Fill in local minima
- Learn important / hard clauses

\[ \leadsto \text{Hypothesis:} \]

Clause penalties determined by DLS algorithm render problem instance easier to solve

Note: This hypothesis was never tested!
Dynamic Local Search

solution
Dynamic Local Search

solution
Experiment:

1. Solve benchmark instances using SAPS; measure search cost (median # variable flips).

2. Take snapshots of clause penalty values at end of characteristic successful runs.

3. Initialise clause penalties according to snapshots; measure search cost for SAPS.

4. Initialise clause penalties randomly; measure search cost for SAPS.

5. Analyse differences in search cost for “learned” and random penalties.
Selecting Characteristic Runs
Flat100: SAPS-generated vs. random weights

The graph shows the comparison of median run-lengths on randomly generated weights versus SAPS generated weights. The x-axis represents the median run-length on SAPS generated weights in steps, while the y-axis shows the median run-length on randomly generated weights in steps. The data points follow a nearly linear trend, indicating a strong correlation between the two sets of weights.
UF100: SAPS-generated vs. random weights

Median run-length on randomly generated weights [steps]
Median run-length on SAPS generated weights [steps]
<table>
<thead>
<tr>
<th>Instance</th>
<th>Unweighted</th>
<th>SAPS Generated Weighted Instances</th>
<th>Randomly Generated Weighted Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>q0.25</td>
<td>Median</td>
</tr>
<tr>
<td>uf100-easy</td>
<td>81</td>
<td>0.98</td>
<td>1.01</td>
</tr>
<tr>
<td>uf100-hard</td>
<td>3,763</td>
<td>1.08</td>
<td>1.11</td>
</tr>
<tr>
<td>uf250-hard</td>
<td>197,044</td>
<td>0.98</td>
<td>1.06</td>
</tr>
<tr>
<td>uf400-hard</td>
<td>2,948,181</td>
<td>0.92</td>
<td>1.04</td>
</tr>
<tr>
<td>flat100-hard</td>
<td>24,248</td>
<td>0.99</td>
<td>1.02</td>
</tr>
<tr>
<td>bw_large.a</td>
<td>2,499</td>
<td>0.90</td>
<td>0.93</td>
</tr>
<tr>
<td>bw_large.b</td>
<td>34,548</td>
<td>0.97</td>
<td>1.02</td>
</tr>
<tr>
<td>logistics.c</td>
<td>9,446</td>
<td>0.97</td>
<td>1.03</td>
</tr>
<tr>
<td>ssa7552-038</td>
<td>3,960</td>
<td>0.86</td>
<td>0.91</td>
</tr>
<tr>
<td>ais10</td>
<td>20,319</td>
<td>1.06</td>
<td>1.09</td>
</tr>
</tbody>
</table>
Result:

No support for hypothesis that clause penalties determined by SAPS render problem instances easier.
So … why does SAPS work?

- Main effect of scaling: escape from local minimum and avoid being immediately sucked back in.

- *But:* adverse side effects (*e.g.*, very likely new / more local minima) due to large “footprints” of clauses.

- *Hence:* Need mechanism for undoing unwanted effects of scaling $\leadsto$ smoothing!
Note:

The main role of penalty modifications appears to be search diversification, which in many other SLS algorithms is achieved through strong randomisation of the search.
Is Randomness Needed?

Random decisions in SAPS:

1. random initialisation of variable assignment
2. random tie-breaking in subsidiary local search
3. random walk steps (in local minimum)
4. probabilistic smoothing
SAPS/NR:

- deterministic tie-breaking
- no random walk steps ($\omega_p = 0$)
- deterministic periodic smoothing

$\Rightarrow$ after initialisation, SAPS/NR is completely deterministic
Experiment:

1. Compare performance and behaviour of SAPS and SAPS/NR.

2. Study variants of SAPS/NR in which only a fraction of variables is initialised with random truth values (others set deterministically).
<table>
<thead>
<tr>
<th>Instance</th>
<th>SAPS</th>
<th></th>
<th></th>
<th>SAPS/NR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>c.v.</td>
<td>Mean</td>
<td>c.v.</td>
<td></td>
</tr>
<tr>
<td>uf100-easy</td>
<td>102</td>
<td>0.75</td>
<td>103</td>
<td>0.70</td>
<td></td>
</tr>
<tr>
<td>uf100-hard</td>
<td>5,572</td>
<td>0.95</td>
<td>5,458</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>uf250-hard</td>
<td>296,523</td>
<td>0.98</td>
<td>282,668</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>uf400-hard</td>
<td>4,349,480</td>
<td>0.75</td>
<td>3,662,192</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td>flat100-hard</td>
<td>35,124</td>
<td>1.02</td>
<td>33,519</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>bw_large.a</td>
<td>3,374</td>
<td>0.85</td>
<td>3,245</td>
<td>0.81</td>
<td></td>
</tr>
<tr>
<td>bw_large.b</td>
<td>50,025</td>
<td>0.95</td>
<td>50,266</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>logistics.c</td>
<td>12,873</td>
<td>0.76</td>
<td>12,458</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td>ssa7552-038</td>
<td>4,460</td>
<td>0.44</td>
<td>4,399</td>
<td>0.41</td>
<td></td>
</tr>
<tr>
<td>ais10</td>
<td>32,810</td>
<td>1.01</td>
<td>31,527</td>
<td>0.99</td>
<td></td>
</tr>
</tbody>
</table>
SAPS vs. SAPS/NR (100 random decisions)

The graph shows the comparison between SAPS and SAPS/NR for uf100-hard instances. The x-axis represents the number of search steps, while the y-axis represents the probability of solving the problem, denoted as P(solve) [%]. The graph illustrates how both algorithms perform across different numbers of search steps, with the solid line representing SAPS and the dashed line representing SAPS/NR [100].
SAPS vs. SAPS/NR (0 random decisions)
SAPS vs. SAPS/NR (1 random decision)

Number of Search Steps

P(solve) [%]

uf100-hard

SAPS
SAPS/NR [1]
SAPS vs. SAPS/NR (2 random decisions)
SAPS vs. SAPS/NR (4 random decisions)
SAPS vs. SAPS/NR (8 random decisions)

![](image.png)
Result:

- Behaviour and performance of SAPS/NR + random initialisation is indistinguishable from fully randomised SAPS
- Performance of (deterministic) SAPS/NR shows sensitive dependence on initial conditions
  $\sim$ central component in definition of chaotic behaviour!
- Diversifying effect of penalty updates is sufficient to propagate small amount of randomness throughout entire search process.
Conclusions

- Penalty mechanism in DLS $\not\Rightarrow$ global simplification (no “long-term memory”)
- Local (“short-term memory”) effects dominate search behaviour
- Penalty mechanism in SAPS primarily provides search diversification
- Only few initial random decisions are sufficient for obtaining same behaviour as fully randomised SAPS algorithm
- Behaviour of deterministic SAPS/NR algorithm sensitively depends on initial conditions (chaotic behaviour?)
Future Work

- characterisation of “warped” search spaces
- separation of short-term and long-term memory in DLS
- optimally weighted SAT instances
- advanced initialisation methods for SAPS/NR
- further investigation of “chaotic” behaviour in SAPS/NR