Using Backdoors to Generate Learnt Information in SAT Solving

Supplementary material for paper #1245

The document is divided into two informative sections. The first section offers an in-depth view of our `Interleave` procedure, showcasing the pseudocode with detailed clarifications. The second section focuses on the computational experiments. It provides a breakdown of the results via two tables. The first table summarizes statistics, including the number of SAT/UNSAT instances and the PAR-2 score of the tested configurations. The second table details the parameters of the tested configurations, providing a clear view of our experimental setup.

1 Interleave algorithm

The Interleave procedure, presented in Algorithm 1, functions akin to a SAT solver, accepting CNF and its parameters as input and outputting the result as SAT/UNSAT. The parameters of the algorithm can be broadly divided into three categories:

- Parameters for the Evolutionary Algorithm (EA) responsible for finding backdoors: backdoor_size, ban_used, seed, num_iters, stagnation_limit.
- Parameters for the procedure that handles hard tasks, such as applying a limited solver to their filtering: max_product, num_conflicts.
- Parameters for the general procedure that interleaves the usual solving via CaDiCaL and the ρ-backdoor based methods described in the paper: budget_filter, budget_solve, factor_budget_filter, factor_budget_solve, derive_ternary, budget_presolve.

The algorithm operates in two alternating phases. The first, *ρ-backdoor phase* involves the search for backdoors, the construction of hard tasks, their filtration through a limited solver, and the derivation of short clauses from these tasks. The second phase solely involves running CaDiCaL. Both phases continue until a predefined budget for the number of conflicts is exhausted. These budgets are distinct and adjustable, with each being multiplied by a factor after each phase ends. Overall, these alternating phases resemble how CDCL solvers switch between what is now known as *focused* and *stable* phases.

Before the actual interleave procedure, we "presolve" the given CNF using CaDiCaL with the specified budget_presolve. This allows for filtering out "easy" problems that do not require a complex approach, i.e. for which "just running CaDiCaL" is enough.

The *ρ-backdoor phase* begins by searching for a backdoor using an evolutionary algorithm similar to the well-known (1+1)-EA, but with the mutation operator designed to transform sets of variables of size n to sets of the same size. This is more convenient as we always obtain the backdoors of a fixed size. The maximum number of mutations is determined by the num_iters parameter. Alternatively, if stagnation_limit is specified, it allows to break out of the EA loop early if the best fitness value have not improved in stagnation_limit iterations. The algorithm employs ρ as the fitness function, which is computed efficiently using unit propagation (UP).

Central to the *ρ-backdoor phase* is the concept of a set of hard tasks, which can be viewed as a representation of a ρ-backdoor. By combining ρ-backdoors, we can construct a larger ρ-backdoor with a better (larger) ρ value by exclusively working with the hard tasks of the original backdoors (refer to Section 3.2 of the paper for details). Therefore, each new backdoor discovered by EA is used to refine and expand the current set of hard tasks. This set is further filtered both by UP and by the limited solver (with the specified conflicts budget num_conflicts per single hard task).

We use the greedy algorithm to select the most promising hard tasks for filtering with the limited solver, as described in Section 3.5 of the paper, aiming primarily at binary clauses ($k = 2$). This means that the bipartite graph consists of vertices corresponding to all $4\binom{n}{2}$ potential binary clauses on one side, and all hard tasks (each of length n) on the other. When we prove that a hard task is UNSAT, we update the scores of all its neighbors in the graph. If the budget was insufficient to prove SAT or UNSAT, we update the graph by removing all potential binary clauses adjacent to the processed hard task (and update the scores of their neighbors), as they have no chance of being derived in the current round of filtering.

Finally, to derive clauses, we utilize a simple algorithm similar to Failed Literal Probing. We traverse through all hard tasks and check whether a particular combination of literals does not appear in any of them. If such a combination is absent, we derive its negation, i.e., the clause consisting of negated literals.

Algorithm 1: Interleave

Data: CNF is the input formula, seed is the random seed, ban_used is the option for excluding the seen variables from being repeatedly found in different backdoors in the consecutive EA runs, backdoor_size is the size of each backdoor, num_iters is the number of EA iterations, stagnation_limit is the number of stagnations for early termination of EA, budget_presolve is the conflicts budget for the pre-solving phase, max_product is the maximum size of the set of hard tasks, num_conflicts is the conflicts budget for each invocation of limited solving during the filtration phase, budget_filter is the total conflicts budget for the filtration phase, budget_solve is the conflicts budget for the solving phase, factor_budget_filter and factor_budget_solve are factors for exponentially scaling the budgets after filtration/solving, derive_ternary is a flag for deriving ternary clauses.

Result: SAT/UNSAT as the result of solving SAT for the given CNF.

```
solver ← initialize_sat_solver(CNF)
ea ← initialize_evolutionary_algorithm(seed, ban_used)
hard_tasks ← {[ ]}      /* set of hard tasks, initialized with a single empty task */
switch solver.solve(budget_presolve) do
    case SAT do return SAT
    case UNSAT do return UNSAT
    case INDET do continue
while true do
    backdoor ← ea.search(backdoor_size, num_iters, stagnation_limit)
    hard_tasks_in_backdoor ← get_hard_tasks(backdoor)
    new_hard_tasks ← ∅                  /* initialize an empty set of hard tasks */
    for (old, new) in cartesian_product(hard_tasks, hard_tasks_in_backdoor) do
        cube ← concatenate(old, new)
        if cube does not contain complementary literals then
            new_hard_tasks ← new_hard_tasks ∪ {cube}
    hard_tasks ← new_hard_tasks                  /* override the set of hard tasks */
    /* Filter hard tasks using UP                                                  */
    foreach cube in hard_tasks do
        solver.assume(cube)
        switch solver.propagate() do
            case SAT do return SAT
            case UNSAT do hard_tasks = hard_tasks \ cube
            case INDET do continue
    if len(hard_tasks) = 0 then return UNSAT
    if len(hard_tasks) > max_product then
        hard_tasks ← {[ ]}                          /* reset the set of hard tasks */
        break
    /* Filter hard tasks using limited solver                                      */
    conflicts_limit ← solver.get_conflicts() + budget_filter
    indet_cubes ← ∅
    /* Proceed until the conflicts budget is exhausted                             */
    while solver.get_conflicts() ≤ conflicts_limit do
        best_hard_task ← greedily_choose_best_hard_task(hard_tasks)
        hard_tasks = hard_tasks \ best_hard_task
        solver.assume(best_hard_task)
        switch solver.solve(num_conflicts) do
            case SAT do return SAT
            case UNSAT do rescore()
            case INDET do indet_cubes ← indet_cubes ∪ best_hard_task
    hard_tasks ← hard_tasks ∪ indet_cubes
```

Algorithm 1: Interleave (continued)

```
(continuation of the outer while loop)
    if len(hard_tasks) = 0 then return UNSAT
    if solver.get_conflicts() > conflicts_limit then
        budget_filter ← budget_filter · factor_budget_filter          /* Scale the budget */

    /* Derive unit clauses                                                            */
    foreach literal l do
        if count_occurrences(l, hard_tasks) = 0 then
            solver.add_clause(¬l)

    /* Derive binary clauses                                                          */
    foreach pair of literals (l1, l2) do
        if count_occurrences((l1, l2), hard_tasks) = 0 then
            solver.add_clause(¬l1 ∨ ¬l2)

    /* Derive ternary clauses                                                         */
    if derive_ternary then
        foreach triple of literals (l1, l2, l3) do
            if count_occurrences((l1, l2, l3), hard_tasks) = 0 then
                solver.add_clause(¬l1 ∨ ¬l2 ∨ ¬l3)

    switch solver.solve(budget_solve) do
        case SAT do return SAT
        case UNSAT do return UNSAT
        case INDET do continue
    budget_solve ← budget_solve · factor_budget_solve                 /* Scale the budget */
```

2 Additional Details on Computational Experiments

In the experiments we used several `Interleave` configurations, that are denoted `Int-118` to `Int-126`. They all share the following values of parameters:

- backdoor_size = 10
- num_iters = 10000
- num_conflicts = 1000
- stagnation_limit = 1000

The values of the remaining parameters are shown in Table 1.

Table 1: Parameters of Interleave configurations

Parameter	Int-118	Int-120	Int-121	Int-126
ban_used	yes	yes	yes	yes
budget_presolve	100000	—	100000	100000
derive_ternary	no	no	no	no
max_product	10000	10000	10000	10000
budget_filter	100000	10000	10000	10000
factor_budget_filter	1.1	1.1	1.1	1.1
budget_solve	100000	100000	100000	100000
factor_budget_solve	1.2	1.2	1.3	1.5

The detailed statistics on the results of interleave configurations and CaDiCaL 1.9.5 on the SAT competition 2022 and 2023 benchmarks is presented in Table 2.

Table 2: Detailed statistics on the number of solved instances for each configuration and CaDiCaL 1.9.5

Configuration	SAT	UNSAT	TOTAL	PAR-2
CaDiCaL 1.9.5	18	70	88/116	4196
Interleave-118	19	70	89/116	4188
Interleave-120	21	76	97/116	3659
Interleave-121	18	75	94/116	3955
Interleave-126	17	74	91/116	3943
Interleave-VBS	21	76	98/116	3221