Submitted to *Operations Research*
manuscript OPRE-2012-05-278(R)

Exploiting erraticism in search

Matteo Fischetti, Michele Monaci
Department of Information Engineering, University of Padova, Via Gradenigo 6/B, 35121 Padova, Italy,
matteo.fischetti@unipd.it michele.monaci@unipd.it

High-sensitivity to initial conditions is generally viewed as a drawback of tree search methods, as it leads to an erratic behavior to be mitigated somehow. In this paper we investigate the opposite viewpoint, and consider this behavior as an opportunity to exploit. Our working hypothesis is that erraticism is in fact just a consequence of the exponential nature of tree search, that acts as a chaotic amplifier, so it is largely unavoidable. We propose a bet-and-run approach to actually turn erraticism to one's advantage. The idea is to make a number of short sample runs with randomized initial conditions, to bet on the "most promising" run selected according to certain simple criteria, and to bring it to completion. Computational results on a large testbed of mixed-integer linear programs from the literature are presented, showing the potential of this approach even when embedded in a proof-of-concept implementation.

Key words: enumerative algorithms, mixed integer programming, restart
History: 25 May 2012, Rev. 21 March 2013

1. Introduction

It is a common observation that tree search methods tend to exhibit high-sensitivity to initial conditions, possibly leading to an erratic behavior. This fact was explicitly pointed out, for the first time, in [4] who reported large differences—in terms of both computing times and number of nodes—for solving a same instance with the same release of `IBM ILOG Cplex` solver on different machines. In [15] an explanation was given for such a large variability: although deterministic, all algorithms have to decide how to break ties during the solution process (e.g., within heuristics, when defining the branching variable, etc.). It is thus not surprising that different ways of breaking ties can produce a considerably different performance of the algorithms.

Erraticism is generally viewed as a drawback to be avoided. Instead, we argue that a certain degree of erraticism is unavoidable, as it is related to the exponential nature of tree search that acts as a chaotic amplifier, so one has to cope with it.

Restart policies in search are popular approaches that can be used to exploit erraticism. They essentially implement the idea of aborting the current run after a while, and then restarting it from scratch with an improved set of constraints (and/or just with different initial condition) in the hope that the new search will exhibit a better behavior or that new information on the problem is available. These strategies were originally proposed for solving SAT problems through AI techniques, where each aborted run improves the SAT formulation through additional clauses, thus increasing the chances that the next run will exhibit a better behavior. An analysis of the effect of erraticism on the performance of a SAT solver was given in [8], while a general method for adding controlled randomization into enumerative algorithms for SAT and CSP problems was proposed in [10, 9], and for scheduling problems in [11]. Restart strategies are considered nowadays one of the most robust tools for solving hard SAT instances, and are in fact embedded in modern clause-learning solvers where restart is executed with a very high frequency [13, 12].

The adoption of restart policies by the MIP community is more recent: the concept of *backdoor* for MIPs was introduced in [5], whereas in [14] a restart strategy is used to collect useful information for branching variable selection. Backdoor branching [6] is another example of a branching strategy based on restart, where a high-priority set of branching variables is defined by solving a set-covering model feeded-up by a collection of diversified fractional solutions.

Another way to exploit erraticism has been implemented for the solution of very hard problems, where computing times can be tremendously large. A notable example is the work of [1] on solving some extremely hard Quadratic Assignment Problem instances on a very large computational grid.

Erraticism is also exploited in the context of massive parallel computing, which nowadays is becoming widely available due to multi-core technology and computer grids, allowing for the development of effective parallel MIP solvers [19, 18, 17]. Taking full advantage of the new architecture is far from trivial, in particular because the branching nodes produced in the earliest "ramp up" phase of the enumeration cannot be distributed in a balanced way among the processors. *Racing ramp-up* is a technique proposed in [17, 16] with the aim of avoiding idle processors: a same MIP solver is initially run with different settings, in parallel, until a stopping criterion is reached. Then it is decided which of the generated trees performed best according to some criterion (not described in full details). The nodes of this tree are then distributed among the parallel processors, while all the other results, with the exception of primal solutions, are discarded. According to [16], however, so far these approaches have not proven to be effective enough in terms of number of subproblems solved per thread per second in the initial parts of the search.

In this paper we analyze a related approach intended to add just little overhead to a sequential tree search method, yet being able to produce improved average results for medium-to-hard instances. As in racing ramp-up, we make a number of short runs with randomized initial conditions, bet on the most promising run, and bring it to completion. The resulting approach, that we call *bet-and-run*, is intended for sequential computation, and is not driven by the need of avoiding idle processors in a parallel setting.

Of course, bet-and-run could be applied by trying different parameter settings for the solver—instead of just using blind randomization—or by choosing alternative solvers in a portfolio of options [7]. These latter approaches are however beyond the scope of the present paper, which is devoted to investigate pure erraticism.

To be effective, our bet-and-run framework needs to address two main questions: (i) how to produce diversified runs, and (ii) how to choose, after short computing time, the "best run" to be brought to conclusion.

Point (i) calls for a perturbation method that triggers diversity but that does not deteriorate the average tree-search performance—producing different runs that are systematically worse than the first run is of course of little use in practice.

Point (ii) is more intriguing. If erraticism implies non-predictabilty, there is no hope to come out with a criterion that is systematically able to select a good run—any implementable criterion will have a certain probability of actually selecting the worst one. Therefore, one has to content him/herself with a criterion that gives a positive correlation between the selected run and those that *a posteriori* behave better than the average.

The success of the resulting framework then relies on finding the right balance between sampling overhead and expected improvement achievable by a clever (yet intrinsically unperfect) selection of the best run.

To test the potential of our framework, we concentrate on tree search applied to a generic Mixed-Integer linear Program (MIP) of the form:

$$
\begin{aligned}
(P) \quad & \min c^T x && (1)\\
& Ax = b,\ x \geq 0 && (2)\\
& x_j \text{ integer},\ \forall j \in \mathcal{I}, && (3)\\
& x_j \text{ continuous},\ \forall j \in \mathcal{C}, && (4)
\end{aligned}
$$

where A is an $m \times n$ input matrix, and b and c are input vectors of dimension m and n, respectively. The variable index set $\mathcal{N} := \{1, \ldots, n\}$ is partitioned into $(\mathcal{I}, \mathcal{C})$, where $\mathcal{I}$ is the index set of the

integer variables, while $\mathcal{C}$ indexes the continuous variables, if any. Bounds on the variables are assumed to be part of system (2). Removing the integrality requirement on the integer variables leads to the LP relaxation $\min\{c^T x : x \in P\}$ where $P := \{x \in \mathcal{R}^n_+ : Ax = b\}$.

Our proof-of-concept implementation is built on top of a state-of-the-art MIP solver (`IBM ILOG Cplex` 12.3 in its default setting), with the ambitious goal of improving its average performance on medium-to-hard instances.

The paper is organized as follows. Preliminary experiments to illustrate the role of randomness in MIP solvers are discussed in Section 2. Possible policies to trigger diversity are sketched in Section 3, whereas rule-of-thumb criteria for selecting the most promising run are described in Section 4. Computational results on a proof-of-concept implementation are given in Section 5. Section 6 finally draws some conclusions and outlines possible directions for future research.

2. Erraticism in MIP solvers

To evaluate the effect of small changes in the initial conditions of a MIP solver, we conducted the following preliminary experiment.

We considered 58 problems[1] of medium difficulty, taken from the testbed libraries COR@L and MIPLIB 2010, and repeatedly solved each of them 100 times, using a same solver (`IBM ILOG Cplex` 12.3) with different input random seeds, so so to diversify the runs in a completely random way—no clever parameter tuning was performed.

Each execution was done in single-thread mode and had a time limit of 3,600 CPU seconds. 11 instances were not solved to proven optimality with default settings of the parameters (i.e., in the first execution), but all of them were eventually solved after 100 runs. Table 1 reports the minimum number of branch-and-bound nodes and LP simplex iterations that were required, on average (geometric and arithmetic means), after different numbers k of executions; the number of unsolved instances is given in column # uns. CPU time is not given in the table as the runs were executed on a workstation with different load conditions, but it is proportional to the reported figures. Each line k in the table then gives a measure of the running time needed by a method that just executes k times, in parallel, a same solver with randomized initial conditions, and stops as soon as one run solves the problem—or the time limit is reached for all runs.

# executions	# uns	# nodes		# LP iter.	
k		geom. mean	aritm. mean	geom. mean	aritm. mean
1	11	13,207	320,138	2,212,849	9,882,765
2	7	7,781	266,811	1,444,085	8,104,845
3	5	6,344	254,196	1,170,356	7,118,396
5	5	5,601	238,561	1,090,606	6,574,864
10	4	4,445	217,472	864,700	5,890,291
25	2	3,060	175,976	680,443	5,648,135
50	1	2,192	159,203	494,159	4,660,565
100	0	1,880	149,399	424,593	3,731,679

Table 1 Reduction in the number of nodes by exploiting randomness (58 instances)

[1] namely: `aflow40b`, `app1-2`, `biella1`, `bienst2`, `dano3_5`, `eilB101`, `gmu-35-50`, `n3seq24`, `n4-3`, `neos-1056905`, `neos-1171737`, `neos-1324574`, `neos-1337307`, `neos-1396125`, `neos-1440460`, `neos-1595230`, `neos-1601936`, `neos-1605075`, `neos-1622252`, `neos-551991`, `neos-555343`, `neos-565672`, `neos-631694`, `neos-641591`, `neos-662469`, `neos-785912`, `neos-820146`, `neos-820157`, `neos-831188`, `neos-848845`, `neos-849702`, `neos-859770`, `neos-863472`, `neos-905856`, `neos-935627`, `neos-936660`, `neos-941313`, `neos-950242`, `neos17`, `neos18`, `net12`, `netdiversion`, `noswot`, `ns1208400`, `ns1688347`, `ns1830653`, `ns894788`, `pw-myciel4`, `rail507`, `ran14x18`, `ran14x18_1`, `rocII-4-11`, `rococoC10-001000`, `roll3000`, `satellites1-25`, `tanglegram1`, `timtab1`, and `vpphard`.

The table shows that even changing just the random seed introduces a considerable variability in the solver's performance—at least, in our testbed. Indeed, a single additional run (row $k = 2$ in the table) is enough to reduce the number of nodes by about 40% in geometric mean, whereas 7x reduction is obtained for 100 runs. This is particularly interesting when a large number of parallel processors is available. Indeed, it is known that in this situation the performance of parallel MIP solvers scale-up with some difficulty [16], while randomization can play a role in improving scalability.

Incidentally, during these experiments we were able to solve to proven optimality, for the first time, MIPLIB 2010 instance `buildingenergy`. In fact, our second run (k=2) converged after 10.899 nodes and 2,839 CPU seconds of a IBM power7 computer (using `IBM ILOG Cplex` 12.3, single thread), finding an integer solution of value 33,285.4433 that was considered optimal according to the solver's default tolerances. We then redefined the optimality tolerance to zero, and reran `IBM ILOG Cplex`12.3 (8 threads) by providing 33,285.4433 as the initial upper bound. This run took 623,861 additional nodes and 7,817 CPU seconds, and found a 0-tolerance optimal solution of value 33,283.8532.

3. Triggering diversity

Our first order of business is to find a cheap way to produce diversified runs of our MIP solver, without deteriorating its average performance.

A direct access to all internal mechanisms of the solver would give us plenty of choices on how to randomize the search path in an effective way. As external users, however, we had to find different ways to achieve diversification.

A first possibility, also used in [15], is the following

- (RAN0) just permute, in a random way, the rows and columns of the input instance.

Another possibility is the one used in the experiments of the previous section, namely

- (RAN1) randomly perturb some MIP-solver input parameters (e.g, the random seed, if available), with the aim of changing the very first LP optimal solution at the root node—and then the whole search path.

We stress again that the aim of (RAN1) is not to find an hypothetical best-possible parameter combination, but just to change the search path in a random way.

Both approaches above require to solve each randomized instance from scratch, hence the relatively time-consuming processing of the root node, and in particular the preprocessing and solution of the very first LP, must be reapplied at each run. Because for easy instances the resulting overhead may be significant, one is interested in perturbation schemes that take place right after the preprocessing step and the solution of the very first LP—even if this choice might lead to a significant reduction of diversification. Possible strategies to perturb the choice of the optimal basis of the initial LP at the root node and require only a limited interaction with the MIP solver are as follows:

- (RAN2) after preprocessing and solution of the very first LP, create a copy of the LP, replace its objective function with a random one, reoptimize it to get to a different LP basis, load this basis into the original LP, and reoptimize it;
- (RAN3) as before, but in the LP copy fix all nonbasic variable with nonzero reduced cost to their value in the optimal solution of the original LP.

To keep computing time under control, one can impose an upper bound on the number of simplex iterations allowed for solving the randomized LP problem.

Because of variable fixing, policy (RAN3) will return an alternative vertex in the optimal LP face—assuming dual degeneracy, that arises almost invariably in MIP problems. Note that we are considering a MIP with equality constraints. In presence of inequalities (including variable upper

bounds), our variable fixing needs to implicitly address slack variables as well, and requires to temporarily impose as an equality each inequality with nonzero optimal dual variable.

In our experiments, we implemented scheme (RAN3) by defining an auxiliary problem according to the optimal solution, say $\widetilde{x}$, of the LP relaxation of (P), as follows. Let N^u and N^ℓ denote the set of nonbasic variables with zero reduced cost that hit their upper and lower bound, respectively, in the LP optimal basic solution $\widetilde{x}$. Our auxiliary objective function (to be minimized) is defined as follows: each variable $j \in N^u$ receives a uniformly-randomly cost in range $[1, 100]$, each variable $j \in N^\ell$ receives a uniformly-randomly cost in $[-100, -1]$, whereas the cost of all other variables was not changed. The maximum number of simplex pivots allowed for the reoptimization of the perturbed LP is at most 10% of the number of iterations for solving the first LP, provided that the resulting number is not smaller than 10 and not larger than 100 (in which case we allow for, at most, 10 or 100 pivots, respectively).

According to our extensive computational experiments, (RAN3) is able to produce diversified optimal solutions to the first LP relaxation in a short computing time. The subsequent application of cutting planes algorithms, as well as of primal heuristics, leads to different solutions and cutting planes at the root node. Because of erraticism, even minor change of the status at the end of the root node are amplified by branching, leading to very diversified search paths.

4. Exploiting diversity

A pseudo-code of our `bet-and-run` algorithm is given in Figure 1. By *clone* we mean a copy of the original MIP, to be solved with diversified initial conditions.

Algorithm `bet-and-run`

parameters: C = number of clones;
N = number of nodes for each clone;

initialization
1. solve the LP relaxation of the problem and get solution $\widetilde{x}$;
sampling phase
2. **for** $i = 1$ **to** C **do**
3. apply randomization (RAN3) to $\widetilde{x}$ and define a new clone;
4. solve the clone for at most N nodes;
5. **if** the current clone is solved to optimality **then** stop;
6. **endfor**
long run
7. determine the "best" clone;
8. solve the chosen clone up to the time limit.

Figure 1 Conceptual `bet-and-run` algorithm.

At Step 7, one has to decide the clone to bring to completion. To this aim we selected a few main indices (called *indicators*) of the performance of the short run on each clone. To be specific, at the end of each short run, for each clone we collect the following indicators, listed in priority order:

I_1) number of open nodes;

I_2) percentage lower bound improvement with respect to the root node, in geometric mean for all open nodes;

I_3) sum of the number of integer-infeasible variables among all open nodes;

I_4) sum of the depths of the open nodes;

I_5) sum of integer infeasibilities among all open nodes, as computed by `IBM ILOG Cplex` 12.3;

I_6) best lower bound;
I_7) best upper bound;
I_8) total number of simplex iterations.

As a general rule, the smaller an indicator the better, so indicators I_2 and I_6 are actually stored with opposite sign.

At the end of the last short run, all clone indicators are processed to define the "winning" clone according to the following selection criterion.

For the instance at hand, an indicator is just disregarded in case it provides no discriminant information among clones, i.e., if it is not useful to distinguish between good and bad clones. More specifically, let C denote again the number of clones and let $I_i(k)$ be the stored value of indicator I_i for the k-th clone ($k=1,\dots,C$). For each indicator I_i, we compute its minimum over all clones, say $I_i^{MIN}=\min_k I_i(k)$, and define its scaled value

$$\overline{I}_i(k)=\begin{cases} I_i(k)-I_i^{MIN}, & \text{if } I_i^{MIN}<0;\\ I_i(k), & \text{otherwise.}\end{cases}$$

Indicator $\overline{I}_i$ $(i=1,\dots,8)$ is disregarded if

$$\max_k \overline{I}_i(k)-\min_k \overline{I}_i(k) < \theta\ \frac{\sum_k \overline{I}_i(k)}{C}$$

where θ is an input parameter set to 0.2 in our implementation.

After the above preprocessing, non-discarded clone indicators are scanned according to their priority order. We first keep only the (at most) $K=\lfloor 0.6\times C\rfloor$ clones having the minimum value of the first non-discarded indicator. Then, we consider all the remaining clones having a minimum value for the second non-discarded indicator, and in case of further ties we select the first clone having the minimum third non-discarded indicator. This scheme is intended to break ties by favoring clone $k=1$, corresponding to the unperturbed run in our implementation—so as to select a perturbed clone only when this seems to be really beneficial.

We finally observe that our clone selection procedure (including the choice of the relevant indicators) is quite unsophisticated, hence we expect that an even better rule—possibly using sound machine learning tools—could be derived.

5. Computational experiments

We tested the potential of our bet-and-run approach within a proof-of-concept implementation where diversification criterion (RAN3) was implemented as discussed in Section 3.

All codes were executed on an Intel i5-750 CPU running at 2.67 GHz with `IBM ILOG Cplex` 12.3 as a MIP solver, using callback functions. For a fair comparison, all codes use a (possibly dummy) callback function, thus deactivating `IBM ILOG Cplex`'s proprietary dynamic search. Experiments were performed in single-thread mode, with no concurrent threads running on the same PC (this ensures reliability of the reported computing times). In all tables, computing times are expressed in CPU seconds (geometric means shifted by 0.01 seconds).

We compare the following algorithms:

• `default`, i.e., `IBM ILOG Cplex` 12.3 with default parameters (but no dynamic search because of the dummy callback);

• our `bet-and-run` algorithm which executes the sampling phase for 5 nodes and for 5 clones, and selects the best clone according to the rule described in Section 4.

Furthermore, we report additional information on the best possible performance that one could hope to obtain with our perturbation scheme, and consider the following two "ideal" algorithms:

• `best`, i.e., the algorithm that exploits an "ideal oracle" to find, for each instance and with no computational overhead, the *a-posteriori* best (with respect to computing time) between algorithms `default` and `bet-and-run`;

• `ideal`, i.e., the algorithm that exploits an "ideal oracle" to find, for each instance and with no computational overhead, the *a-posteriori* best (with respect to computing time) among all the 5 clones that have been generated.

We stress here that the latter two idealized algorithms are reported for benchmark purposes only, to quantify performance variability triggered by randomization over the instances of a given class.

Note that in `bet-and-run` we consider just 5 clones, namely, the "default" one (with no randomization after the very first LP) that would produce the same search path as `default`, plus 4 additional clones. This conservative choice was motivated by the need of reducing the sampling overhead for easy instances; more aggressive policies are of course possible. For each clone, only 5 branching nodes are explored in the sampling phase, again with the aim of reducing sampling overhead. We also tried to explore just one node and use the clone indicators available at the end of the root node, but this setting produced worse results on average—at least, by using our clone-selection policy based on the ranked indicators described in the previous section.

We first considered two sets of instances from the literature, namely:

• COR@L [2]: we considered all the 372 instances in this library, and removed two instances, namely `neos-1417043` which is just an LP model, and `neos-578379` which cannot be downloaded in a correct format, plus three instances (`neos-1346382`, `neos-933364` and `neos-641591`) that were duplicated in the library; thus we got 367 problems.

• MIPLIB 2010 [15]: we considered all the 166 instances belonging to classes `benchmark` and `tree`, plus all the instances that were marked as `hard` when we conducted our experiments.

It turns out that 41 instances belong to both sets, thus they were included only once. For each instance, both `default` and `bet-and-run` were run with a time limit equal to 10,000 CPU seconds. The 148 instances that were not solved by `best` within the time limit were disregarded; this lead to a final testbed made by 344 instances.

As this set of instances includes problems of different difficulty, we partitioned the problem set into classes. In particular, each problem was classified according to the maximum number of nodes that was required by the two competing algorithms, `default` and `bet-and-run`, to provide a proven optimal solution. This is a fair policy as the competing methods play an indistinguishable role in the class definition, so no biasing is expected.

Table 2 reports how many time each clone was selected by `bet-and-run` in the various runs of Table 3; clone 0 refers to the default run. Class $[0-10)$ is not reported because almost all instances in this class were solved to optimality during the sampling phase of the default (first) clone. As expected, no significant biasing in the clone choice seems to exist; in particular, clone 0 does not play a distinguished role as it exhibits a "median" behavior.

Node range	# inst.	clone 0	clone 1	clone 2	clone 3	clone 4
$[10-100)$	34	3	15	4	5	7
$[100-1,000)$	46	7	12	7	4	16
$[1,000-10,000)$	66	11	13	8	9	25
$[10,000-100,000)$	50	7	15	6	5	17
$\geq 100,000$	72	9	17	10	9	27
total	268	37	72	35	27	92

Table 2 Number of times a clone was selected by `bet-and-run`; clone 0 is the default one.

Table 3 reports, for each class of instances, the associated range for the maximum number of nodes, the total number of instances and, for each algorithm, the following information:

• number of instances solved to proven optimality—this information is omitted for the "ideal" algorithms as, by definition, this figure equals the number of instances in the class;

• geometric mean of the computing time and its percentage increase w.r.t. `default` (negative if better than `default`);

• geometric mean of the number of nodes and its percentage increase w.r.t. `default` (negative if better than `default`);

Note that computing times for `bet-and-run` include the sampling phase. For the sake of completeness, we also report in column T_{last} the geometric mean of the computing time required by the last (long) run of `bet-and-run`, along with the associated percentage increase w.r.t. `default`. To be conservative, instances that hit the time limit for algorithm `bet-and-run` are counted as a time limit for column T_{last} as well, although the saved sampling time could allow for their exact solution in some cases.

As expected, for "easy" instances our approach is not beneficial in that the sampling overhead does not pay off. However, our approach turns out to produce interesting outcomes for the hard instances—those for which both `default` and `bet-and-run` take more than 1,000 nodes. On the whole, `bet-and-run` solves 3 more instances within the time limit than `default` (which is by itself a main accomplishment), and also allows for a significant saving in terms of both CPU time and number of nodes for medium-to-hard instances, even with respect to a very effective code such as `IBM ILOG Cplex` 12.3. In particular, for the instances in class $[10,000 - 100,000)$ `bet-and-run` allows for a CPU time and node saving with respect to `default` of about 28% and 51%, respectively. Savings are less impressive (but still relevant) for class $\geq 100,000$, where they amount to about 17% (CPU time) and 28% (number of nodes). A possible explanation of the reduced savings is that the clone-selection criterion used by `bet-and-run` is only based on the very early part of the search, so its positive effect tends to be overtaken by erraticism after a very large amount of enumeration.

Of course, a much better improvement could be obtained if we were able to reduce the overhead due to the preprocessing phase, getting closer to the very attractive computing times reported in column T_{last}. However, this task is hardly achievable without having a direct access to the solver's source code, hence it is out of the scope of the present paper.

Also very interesting are the savings achieved by `best`, i.e., by running `default` and `bet-and-run` in parallel and aborting their execution as soon as one of two terminates—very much in the spirit of the experiments reported in Table 1—and by `ideal`. For parallel architectures, this is in fact an option that deserves future investigation.

We have seen that the speedup achieved by `bet-and-run` is quite important for medium-to-hard instances, while (as expected) for easy instances the new approach is not competitive with `default`. In our view, this behavior is not really an issue and just suggests an implementation where bet-and-run can be switched on/off through an input parameter. Indeed, most MIP-solver users routinely solve problems of the same nature (e.g., crew scheduling instances related to different scenarios) and thus know in advance whether their model is likely to require a small or large amount of enumeration—so they can decide in advance whether to activate or not the bet-and-run flag.

Whether bet-and-run strategy can be included by default in a MIP solver is instead less obvious. Although the investigation of this implementation issue is outside the scope of the present paper, we observe that two main approaches for reducing sampling overhead can be addressed, thus improving bet-and-run for easy instances.

One is to exploit parallelism/multi-threading for running the sampling phase in parallel. This is an interesting option in that it is known that parallel enumerative codes do not scale well in the "ramp-up" phase when the number of open tree nodes is smaller than the available CPUs. Therefore running our sampling phase in parallel is likely to add a negligible overhead when a reasonable number of CPUs is available (say, 8 or more, as it is the case for any modern architecture).

A second option, not based on parallelism, is based on the following simple restart policy. Run the default enumerative algorithm, `default`, for a certain number of nodes, say NR. If the instance

Node range	Algorithm	# opt.	Time	%incr	# Nodes	%incr	T_{last}	%incr
	default	76	1.50	0.0	1	0.0		
$[0-10)$	bet-and-run	76	1.50	-0.7	1	0.0	1.47	-2.0
# inst. 76	best		1.49	-1.3	1	0.0		
	ideal		0.92	-38.7	1	0.0		
	default	33	6.95	0.0	18	0.0		
$[10-100)$	bet-and-run	33	16.70	140.3	21	16.7	4.93	-29.2
# inst. 34	best		6.84	-1.7	18	0.0		
	ideal		3.90	-44.0	5	-72.2		
	default	46	8.73	0.0	302	0.0		
$[100-1,000)$	bet-and-run	46	16.03	83.5	210	-30.5	6.64	-23.9
# inst. 46	best		8.03	-8.1	225	-25.5		
	ideal		5.64	-35.5	94	-68.9		
	default	65	21.89	0.0	2,503	0.0		
$[1,000-10,000)$	bet-and-run	66	33.47	53.0	2,702	8.0	22.02	0.6
# inst. 66	best		20.86	-4.7	2,434	-2.8		
	ideal		17.38	-20.6	1,799	-28.1		
	default	48	212.73	0.0	23,942	0.0		
$[10,000-100,000)$	bet-and-run	50	152.10	-28.5	11,551	-51.8	129.59	-39.1
# inst. 50	best		122.99	-42.2	9,592	-59.9		
	ideal		93.81	-55.9	7,546	-68.5		
	default	62	1,561.65	0.0	649,103	0.0		
$\geq 100,000$	bet-and-run	62	1,282.88	-17.9	463,201	-28.6	1,211.58	-22.4
# inst. 72	best		971.90	-37.8	357,213	-45.0		
	ideal		736.08	-52.9	264,589	-59.2		

Node range	Algorithm	# opt.	Time	%incr	# Nodes	%incr	T_{last}	%incr
	default	330	32.51	0.0	951	0.0		
≥ 0	bet-and-run	333	38.12	17.2	778	-18.2	26.64	-18.1
# inst. 344	best		26.52	-18.4	638	-32.9		
	ideal		18.86	-42.0	471	-50.5		
	default	221	110.41	0.0	14,807	0.0		
≥ 100	bet-and-run	224	122.89	11.3	10,868	-26.6	87.20	-21.0
# inst. 234	best		82.37	-25.4	8,694	-41.3		
	ideal		63.24	-42.7	6,355	-57.1		
	default	175	205.43	0.0	38,351	0.0		
$\geq 1,000$	bet-and-run	178	202.29	-1.5	28,518	-25.6	163.73	-20.3
# inst. 188	best		145.61	-29.1	22,822	-40.5		
	ideal		114.25	-44.4	17,816	-53.5		
	default	110	689.87	0.0	167,863	0.0		
$\geq 10,000$	bet-and-run	112	535.36	-22.4	102,034	-39.2	484.72	-29.7
# inst. 122	best		416.58	-39.6	80,991	-51.8		
	ideal		316.41	-54.1	61,580	-63.3		
	default	62	1,561.65	0.0	649,103	0.0		
$\geq 100,000$	bet-and-run	62	1,282.88	-17.9	463,201	-28.6	1,211.58	-22.4
# inst. 72	best		971.90	-37.8	357,213	-45.0		
	ideal		736.08	-52.9	264,589	-59.2		

Table 3 Results on the 344 instances in our testbed (geometric means).

turns out to be easy, the problem is solved and no overhead is incurred. Otherwise, pause `default` and execute the sampling phase of `bet-and-run`: if the winning clone is the default one[2], just continue with `default`; otherwise, abort `default` and continue with `bet-and-run`. This approach has of course the drawback of wasting some amount of CPU time for all instances requiring more than NR nodes, namely, the CPU time required by either the first NR nodes of `default` or by the sampling phase of `bet-and-run`. Computational results for this modified algorithm, called `hybrid` in what follows, are reported in Table 4 for $NR = 500$, and in Table 5 for $NR = 1,000$. We also report statistics for `best`, now defined as the best between `default` and `hybrid`, while we omit `ideal` that is not affected by the initial execution of `default`. Note that 2 more instances were not solved within the time limit by this new `best` (because of the overhead introduced in `bet-and-run`) and thus were removed from the testbed. Also to be noted is that a same instance can belong to different classes in Tables 3-5, because our class definition depends on the maximum number of nodes required by the two analyzed algorithms. As a result, a direct comparison of, e.g., the number of solved instances in each class can be misleading.

By design, the execution of `default` before `bet-and-run` has a positive effect when easy instances are considered. On the other hand, the additional overhead can be nonnegligible for "not easy nor too hard" problems. In fact, class $[1,000, 10,000)$ is the most critical one when $NR = 500$ or $1,000$ is chosen, as the restart overhead is not compensated by the improved performance of the hybrid method. On the whole, the results confirm the viability of using a restart policy within the overall bet-and-run scheme.

Of course, different NR thresholds would produce different statistics, moving criticality into different classes. A more sophisticated approach, not investigated in the present paper, would be to determine NR on the fly, by using a (possibly rough) estimate of the remaining branching nodes, e.g., by using the early tree-node estimator proposed in [3].

6. Conclusions and future directions of work

Erraticism is typically viewed as a drawback of tree search. We have argued that a certain degree of erraticism is unavoidable, as this is in fact an intrinsic property of such a method, whose exponential nature acts as a chaotic amplifier. We have then presented a simple *bet-and-run* approach to turn erraticism to one's advantage. Computational results on a simple proof-of-concept implementation show the potential of the approach.

Future research should be devoted to improving the diversification mechanism used, a task that would be most successful if a complete access to the source code of the solver was available. Also of interest is a better classification mechanism—possibly using machine learning tools—to discriminate between "good" and "bad" sample runs, so as to increase the correlation of the chosen clone with the *a posteriori* best one.

Also to be investigated is an improved restart strategy borrowed from the AI/CP community, that (i) modifies the backtrack condition on the fly, and (ii) takes advantage of information resulting from the previous runs—in the MIP context, this include primal solutions, collected cuts, and variable pseudocosts for branching. The use of portfolio approaches (on sequential or parallel machines) is also worth investigating.

Finally, an interesting research topic is the role of erraticism in improving the quality of the incumbent solution found after a short while by an enumerative exact method.

Acknowledgments

This research was supported by the *Progetto di Ateneo* on "Computational Integer Programming" of the University of Padova, and by MiUR, Italy (PRIN project "Integrated Approaches to Discrete and Nonlinear

[2] in our experiments, this situation occurred 29 out of 200 times (for $NR = 500$) and 27 out of 181 times (for $NR = 1,000$).

Node range	Algorithm	# opt.	Time	%incr	# Nodes	%incr
	default	80	1.51	0.0	1	0.0
$[0-10)$	hybrid	80	1.51	0.0	1	0.0
# inst. 80	best		1.51	0.0	1	0.0
	default	31	6.38	0.0	21	0.0
$[10-100)$	hybrid	31	6.38	0.0	21	0.0
# inst. 31	best		6.38	0.0	21	0.0
	default	37	9.81	0.0	279	0.0
$[100-1,000)$	hybrid	37	10.58	7.8	277	-0.7
# inst. 37	best		9.74	-0.8	279	0.0
	default	72	17.69	0.0	2,308	0.0
$[1,000-10,000)$	hybrid	72	31.19	76.4	2,964	28.4
# inst. 72	best		17.53	-0.8	2,245	-2.7
	default	49	220.78	0.0	24,203	0.0
$[10,000-100,000)$	hybrid	50	175.09	-20.7	14,876	-38.5
# inst. 51	best		135.81	-38.5	12,415	-48.7
	default	61	1,563.83	0.0	674,703	0.0
$\geq 100,000$	hybrid	62	1,291.98	-17.4	527,052	-21.9
# inst. 71	best		1,002.00	-35.9	428,666	-36.5

Node range	Algorithm	# opt.	Time	%incr	# Nodes	%incr
	default	330	31.44	0.0	964	0.0
≥ 0	hybrid	332	33.11	5.3	897	-7.0
# inst. 342	best		26.47	-15.8	779	-19.2
	default	219	111.42	0.0	15,836	0.0
≥ 100	hybrid	221	120.59	8.2	14,237	-10.1
# inst. 231	best		86.95	-22.0	11,565	-27.0
	default	182	177.11	0.0	34,201	0.0
$\geq 1,000$	hybrid	184	191.80	8.3	30,165	-11.8
# inst. 194	best		132.01	-25.5	23,613	-31.0
	default	110	689.87	0.0	167,863	0.0
$\geq 10,000$	hybrid	112	560.28	-18.8	118,623	-29.3
# inst. 122	best		434.56	-37.0	96,119	-42.7
	default	61	1,563.83	0.0	674,703	0.0
$\geq 100,000$	hybrid	62	1,291.98	-17.4	527,052	-21.9
# inst. 71	best		1,002.00	-35.9	422,144	-37.4

Table 4 Hybrid method (geometric means) when $NR = 500$ nodes are explored before `bet-and-run`.

Optimization"). We thank Gianfranco Bilardi for the use of the computers of the Center of Excellence "Scientific and Engineering Applications of Advanced Computing Paradigms". We also thank two anonymous referees for their helpful suggestions on how to improve the presentation of the paper.

References

[1] K.M. Anstreicher, N.W. Brixius, J.-P. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91:563–588, 2002.

[2] COR@L. Computational optimization research at Lehigh. 2005. available at http://coral.ie.lehigh.edu/data-sets/mixed-integer-instances/.

[3] G. Cornuéjols, M. Karamanov, and Y. Li. Early estimates of the size of branch-and-bound trees. *INFORMS Journal on Computing*, 18:86–96, 2006.

[4] E. Danna. Performance variability in mixed integer programming. Presentation at the 5th Workshop on Mixed Integer Programming, Columbia University, New York, August 4-7, 2008.

[5] B.N. Dilkina, C.P. Gomes, Y. Malitsky, A. Sabharwal, and M. Sellmann. Backdoors to combinatorial optimization: Feasibility and optimality. In Willem Jan van Hoeve and John N. Hooker, editors, *Integration of AI and OR*

Node range	Algorithm	# opt.	Time	%incr	# Nodes	%incr
	default	80	1.51	0.0	1	0.0
$[0-10)$	hybrid	80	1.51	0.0	1	0.0
# inst. 80	best		1.51	0.0	1	0.0
	default	31	6.38	0.0	21	0.0
$[10-100)$	hybrid	31	6.38	0.0	21	0.0
# inst. 31	best		6.38	0.0	21	0.0
	default	50	8.97	0.0	362	0.0
$[100-1,000)$	hybrid	50	8.97	0.0	362	0.0
# inst. 50	best		8.97	0.0	362	0.0
	default	58	20.94	0.0	2,933	0.0
$[1,000-10,000)$	hybrid	58	37.56	79.4	3,933	34.1
# inst. 58	best		20.81	-0.6	2,906	-0.9
	default	50	219.96	0.0	23,355	0.0
$[10,000-100,000)$	hybrid	51	185.47	-15.7	15,868	-32.1
# inst. 52	best		140.84	-36.0	12,972	-44.5
	default	61	1,563.83	0.0	674,703	0.0
$\geq 100,000$	hybrid	62	1,314.72	-15.9	542,047	-19.7
# inst. 71	best		1,019.18	-34.8	440,800	-34.7

Node range	Algorithm	# opt.	Time	%incr	# Nodes	%incr
	default	330	31.44	0.0	964	0.0
≥ 0	hybrid	332	32.57	3.6	913	-5.3
# inst. 342	best		26.73	-15.0	798	-17.2
	default	219	111.42	0.0	15,836	0.0
≥ 100	hybrid	221	117.78	5.7	14,609	-7.7
# inst. 231	best		88.19	-20.9	11,983	-24.3
	default	169	223.46	0.0	44,943	0.0
$\geq 1,000$	hybrid	171	239.70	7.3	40,548	-9.8
# inst. 181	best		165.88	-25.8	31,487	-29.9
	default	111	682.43	0.0	162,767	0.0
$\geq 10,000$	hybrid	113	574.45	-15.8	121,823	-25.2
# inst. 123	best		441.44	-35.3	97,700	-40.0
	default	61	1,563.83	0.0	674,703	0.0
$\geq 100,000$	hybrid	62	1,314.72	-15.9	542,047	-19.7
# inst. 71	best		1,019.18	-34.8	433,999	-35.7

Table 5 Hybrid method (geometric means) when $NR = 1,000$ nodes are explored before `bet-and-run`.

Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, Pittsburgh, PA, USA, May 27-31, 2009, Proceedings, volume 5547 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2009.

[6] M. Fischetti and M. Monaci. Backdoor branching. In Oktay Günlük and Gerhard J. Woeginger, editors, *Integer Programming and Combinatorial Optimization (IPCO 2011)*, pages 183–191, Berlin Heidelberg, 2011. Springer Lecture Notes in Computer Science 6655.

[7] C.P. Gomes and B. Selman. Algorithm portfolio design: theory vs. practice. In *Proceedings of the Thirteenth conference on Uncertainty in Artificial Intelligence*, UAI'97, pages 190–197, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[8] C.P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *Principles and Practices of Constraint Programming (CP-97)*, volume 1330 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1997.

[9] C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24:2000, 2000.

[10] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the National Conference on Artificial Intelligence*, pages 431–437. AAAI Press, 1998.

[11] C.P. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In Reid G. Simmons, Manuela M. Veloso, and Stephen F. Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS), Pittsburgh, Pennsylvania, USA, 1998*, pages 208–213. AAAI, 1998.

[12] S. Haim and M. Heule. Towards ultra rapid restarts. Technical report, UNSW and TU Delft, 2010.

[13] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pages 2318–2323. Morgan Kaufmann Publishers Inc., 2007.

[14] F.K. Karzan, G.L. Nemhauser, and M.W.P. Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation*, 1:249–293, 2009.

[15] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R.E. Bixby, E. Danna, G. Gamrath, A.M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D.E. Steffy, and K. Wolter. MIPLIB 2010 mixed integer programming library version 5. *Mathematical Programming Computation*, 3:103–163, 2011.

[16] T. Koch, T. Ralphs, and Y. Shinano. What could a million CPUs do to solve integer programs? Technical report, ZIB 11-40, 2011.

[17] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch. ParaSCIP: a parallel extension of SCIP. Technical report, ZIB 10-27, 2010.

[18] Y. Shinano, T. Achterberg, and T. Fujie. A dynamic load balancing mechanism for new paralex. In *Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems*, 2008.

[19] Y. Shinano and T. Fujie. Paralex: A parallel extension for the cplex mixed integer optimizer. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science 4757, pages 97–106, 2007.