Randomization in Constraint Programming for Airline Planning

Lars Otten[1], Mattias Grönkvist[1,2], and Devdatt Dubhashi[1]

[1] Department of Computer Science and Engineering,
Chalmers University of Technology, 41296 Gothenburg, Sweden
`mail@lotten.net`, `dubhashi@cs.chalmers.se`
[2] Jeppesen (Carmen Systems AB),
Odinsgatan 9, 41103 Gothenburg, Sweden
`mattias.gronkvist@jeppesen.com`

Abstract. We extend the common depth-first backtrack search for constraint satisfaction problems with randomized variable and value selection. The resulting methods are applied to real-world instances of the tail assignment problem, a certain kind of airline planning problem. We analyze the performance impact of these extensions and, in order to exploit the improvements, add restarts to the search procedure. Finally computational results of the complete approach are discussed.

1 Introduction

Constraint programming has received increasing attention in a multitude of areas and applications and has successfully been incorporated into a number of commercial systems.

Among other things it has been deployed in planning for transportation: Grönkvist [7] uses constraint programming to solve the tail assignment problem, an airline planning variant. Gabteni and Grönkvist [3] combine this with techniques from integer programming, in order to obtain a complete solver and optimizer, which is currently in use at a number of medium-sized airlines.

In Gabteni and Grönkvist's work, special constraints that internally employ the pricing routine of a column generation integer programming solver are used to tackle arising computational issues. In this paper, however, we set out to take an orthogonal approach, which is not dependent on column generation but rather relies on pure constraint programming techniques. Furthermore, in contrast to the usage of specialized propagation algorithms, our approach is generic at heart, hence it is more easily adaptable to problems other than tail assignment.

Recently Gomes et al. [5] have made considerable progress in exploiting the benefits of randomization in backtrack search for constraint programming, they also applied their findings to a number of more or less artificially constructed problem instances.

We pursue this randomized approach with the tail assignment problem, for which we have access to a number of real-world instances. We show how this helps in overcoming the performance bottlenecks mentioned in [3, 7].

Our contribution is as follows: We review and present a number of new generic randomized schemes for variable and value selection for backtrack search. We discuss how these techniques enhance performance for the tail assignment problem. Finally, we explain how different restart strategies can systematically make use of the improvements. We then demonstrate that this does in fact work very well for practical purposes in real-life problem instances.

We introduce the tail assignment problem in the general context of airline planning in Sect. 2 and give a formulation as a constraint programming problem. Section 3 describes the randomized extensions to the backtrack search procedure and gives some first computational results. In Sect. 4 we present restart techniques as a way to exploit the benefits of randomization and discuss the respective performance results on our real-world instances. Section 5 concludes and outlines future research directions.

2 Problem Description

The process of airline planning is commonly divided into several isolated steps: During *timetable creation* a schedule of flights (referred to as *legs*) is assembled for a certain time interval. Given such an airline timetable, *fleet assignment* usually means determining which type of aircraft will cover which timetable entry while maximizing revenue. The *aircraft routing* process then assigns individual aircraft to the elements of the schedule for each fleet (or subfleet), mainly with focus on maintenance feasibility. This, in turn, is followed by *crew rostering*, which selects the required personnel for each flight leg.

In practice all these steps will be subject to a number of operational constraints (aircraft range limitations, noise level restrictions at certain airports etc.) and optimization criteria, for example minimizing the resulting costs or keeping the aircraft deployment as evenly distributed as possible.

Since airline planning has received widespread attention in a multitude of respects, there is a lot of related work. For lack of space we refer to Grönkvist [8] for a comprehensive overview and limit ourselves to two references here:

Gopalan and Talluri [6] give a general survey of several of the common problems and methods and Barnhart et al., for instance, discuss combined fleet assignment / aircraft routing in [2].

2.1 The Tail Assignment Problem

The problem of *tail assignment* denotes the process of assigning aircraft (identified by their *tail number*) to each leg of a given airline timetable. As a result one obtains a *route* for each aircraft, consisting of a sequence of legs. Hence tail assignment essentially combines fleet assignment and aircraft routing as describe before. There are several practical, operational advantages inherent to this approach, for details we refer to Gabteni and Grönkvist [3].

In their work they make use of a combined solution approach, employing techniques from both integer programming (in particular column generation) and constraint programming.

The motivation for this is that constraint programming usually finds a feasible but not necessarily optimal solution rather quickly, whereas column generation converges slowly but ensures optimality. We will henceforth focus on the constraint programming component.

2.2 A Constraint Programming Model

We now formulate the tail assignment problem as a constraint satisfaction problem (CSP). First we note that, instead of flight legs and aircraft, we will speak of *activities* and *vehicles*, which is more general and allows us for example to include scheduled maintenance into the problem. We then start by defining the following:

$$F = \{f_1, \ldots, f_n\}\,, \text{ the set of all activities.}$$
$$T = \{t_1, \ldots, t_m\}\,, \text{ the set of all vehicles.}$$

We will think of each activity as a node in an *activity network*: Initially each vehicle is assigned a unique start activity. Each activity will eventually be connected to the two activities pre- and succeeding it. Thus each vehicle's route becomes a path through the network. In fact, since we will connect a route's end activity to its start activity, we obtain exactly m cycles, one for each vehicle.

Now, to capture this within a CSP, we introduce a number of variables: For all $f \in F$ we have $\texttt{successor}_f$ with domain $D(\texttt{successor}_f) \subseteq F$ initially containing all activities possibly succeeding activity f. Equivalently, for all $f \in F$, we have $\texttt{vehicle}_f$ with domain $D(\texttt{vehicle}_f) \subseteq T$ initially containing all the vehicles that are in principle allowed to operate activity f.

Note that with the proper initial domains we cover a lot of the problem already. For example we can be sure that only legally allowed connections between activities will be selected in the final solution. Moreover we can implement a preassigned activity (like maintenance for a specific vehicle) by initializing the respective `vehicle` variable with an unary domain.

Also observe that any solution to the tail assignment problem can be represented by a complete assignment to either all `successor` or `vehicle` variables – we can construct any vehicle's route by following the `successor` links from its start activity, or we can group all activities by their `vehicle` value and order those groups chronologically to obtain the routes.

To obtain stronger propagation behavior later on we also introduce a third group of variables, similar to the `successor` variables: For all $f \in F$ we have $\texttt{predecessor}_f$ with domain $D(\texttt{predecessor}_f) \subseteq F$ initially containing all activities possibly preceding activity f.

Introducing constraints to our model, we first note that we want all routes to be disjoint – for instance two different activities should not have the same succeeding activity. Hence, as a first constraint, we add a global `alldifferent` over all `successor` variables.

Moreover, since the `successor` and `predecessor` are conceptually inverse to each other, we add a global constraint `inverse(successor,predecessor)` to our model, which is in practice implemented by means of the respective number of binary constraints and defined likewise:

$$\forall\, i, j : \quad f_i \in D(\texttt{successor}_j) \Longleftrightarrow f_j \in D(\texttt{predecessor}_i) \ .$$

This also implicitly ensures disjointness with respect to the `predecessor` variables, hence we need not add an `alldifferent` constraint over those.

Finally, to obtain a connection with the `vehicle` variables, we define another global constraint, which we call `tunneling`. It observes all variable domains and, each time a variable gets instantiated, posts other constraints according to the following rules (where $\texttt{element}(a, \texttt{B}, c)$ requires the value of $\texttt{B}_a$ to be equal to c):

$$\begin{aligned}
\texttt{vehicle}_f == t \quad &\Rightarrow \quad \texttt{POST element}(\texttt{successor}_f, \texttt{vehicle}, t) \\
& \quad \texttt{POST element}(\texttt{predecessor}_f, \texttt{vehicle}, t) \\
\texttt{successor}_f == f' \quad &\Rightarrow \quad \texttt{POST vehicle}_f = \texttt{vehicle}_{f'} \\
\texttt{predecessor}_f == f' \quad &\Rightarrow \quad \texttt{POST vehicle}_f = \texttt{vehicle}_{f'}
\end{aligned}$$

These constraints already suffice to model a basic version of the tail assignment problem as described above, with slightly relaxed maintenance constraints. Still we will in practice add a number of constraints to improve propagation and thus computational performance: For example we can add `alldifferent` constraints over `vehicle` variables of activities that overlap in time (for certain cleverly picked times).

When solving this CSP we will only branch on the `successor` variables, meaning only these are instantiated during search. We do this because `successor` modifications are propagated well thanks to the consistency algorithm of the `alldifferent` constraint imposed on them.

2.3 Remarks

The CSP modeled above is essentially what Gabteni and Grönkvist [3] refer to as CSP-TAS^{relax} – "relax" since it does not cover some of the more complicated maintenance constraints. Still this model is used in the final integrated solution presented in [3].

What is of interest to us, however, is that for this model Gabteni and Grönkvist [3] report problems with excessive *thrashing* for problem instances containing several different types of aircraft (and thus more flight restrictions inherent in the initial `vehicle` variable domains). Thrashing means that the search procedure spends large amounts of time in subtrees that do not contain a solution, which results in a lot of backtracking taking place and consequently very long search times.

Gabteni and Grönkvist [3] try to resolve this by extending the model and introducing additional constraints, for which they implement strong propagation algorithms that perform elaborate book-keeping of reachable activities and subroutes and thereby are able to rule out certain parts of the search tree in advance.

It is worth noting that these propagation algorithms make use of the pricing routine of a column generation system from the area of integer programming. The resulting extended model is then referred to as CSP-TAS.

However, with the results of Gomes et al. [5] in mind, we take a different and, as we believe, more general approach to reduce thrashing, modifying the backtrack search procedure itself while leaving the model CSP-TAS^{relax} unchanged. Our solution also supersedes the use of elements from integer programming, which are often not readily available for a CSP but require additional effort.

On another note we should point out that, although in principle tail assignment depicts an optimization problem, in this paper we neglect the aspect of solution quality and focus on finding any one solution. Experience shows that computing any solution is often sufficient, especially if it can be achieved quickly. Moreover, even in situations where one wants a close-to-optimal solution, finding any feasible solution is useful as a starting point for improvement heuristics and as a proof that a solution exists.

3 Randomizing Backtrack Search

Standard backtrack search for constraint programming interleaves search with constraint propagation: The constraints are propagated, one search step is performed, the constraints are propagated and so on. In case of a failure, when the CSP becomes inconsistent, i. e. impossible to solve, we rollback one or several search steps and retry.

Our focus is on the search step: Generally one starts with choosing the next variable to branch on; after that one of the values from the variable's domain is selected, to which the variable is then instantiated. Both these choices will be covered separately. A number of ideas have already been introduced by Gomes et al. [4, 5]; we will briefly review their findings and adapt the concept to our problem by introducing enhanced randomized selection schemes.

3.1 Variable Selection

There exist several common, nonrandomized heuristics for finding the most promising variable to branch on: For example we pick the variable with the smallest domain size. We will call this scheme *min-size*, it is sometimes also known as *fail-first*.

Alternatively we choose the variable with the smallest degree, where the degree of a variable is the number of constraints related to it; this heuristic will be referred to as *min-degree*. Grönkvist [7] uses *min-size*, yet we performed our tests with both heuristics.

In case of ties, for instance when two or more variables have the same minimal domain size, these heuristics make a deterministic choice, for example by lexicographical order of the variables. This already suggests a straightforward approach to randomize the algorithm: When we encouter ties we pick one of the candidates at random (uniformly distributed).

With sophisticated heuristics it can happen that only one or very few variables are in fact assigned the optimal heuristic value, meaning that the random tie-breaking will have little or no effect after all. To alleviate this, Gomes et al. suggest a less restrictive selection process, where the random choice is made between all optimal variables and those whose heuristic value is within an $H\,\%$ range of the optimum.

For our problem, however, we found that the said situation rarely occurs, and in fact the less restrictive $H\,\%$ rule had a negative effect on search performance.

Instead we tried another similar but more general variable selection scheme: For a certain base $b \in \mathbb{R}$, we choose any currently unassigned variable x with a probability proportional to the value $b^{-s(x)}$, where $s(x)$ is the current size of the variable's domain. Observe that for increasing values of b we will gradually approach the *min-size* scheme with random tie-breaking as described above.

3.2 Value Selection

Once we have determined which variable to branch on, the simplest way of adding randomization for the value selection is to just pick a value at random (again uniformly distributed), which is also suggested by Gomes et al. [4]. This already works quite well, yet for the problem at hand we developed something more specific.

As a nonrandomized value ordering heuristic for the tail assignment problem, Grönkvist [7] suggests to choose the successor activities by increasing connection time (recall that we only branch on the `successor` variables). Hence in our model we order the activities by increasing start time, so that obtaining the shortest possible connection time is equivalent to selecting the smallest value first.

We then take this idea as an inspiration for the following randomized value selection scheme: Assuming a current domain size of n possible values, we pick the smallest value (representing the shortest possible connection time) with probability $p \in (0,1)$, the second smallest with probability $p \cdot q$, the third smallest with probability $p \cdot q^2$ and so on, for a $q > 0$; in general, we choose the i-th smallest with probability $p \cdot q^{i-1}$.

Having this idea in mind, we note the following: Given n and either p or q, we can compute q or p, respectively. To do so we take the sum over all elements' probabilities, which has to be 1 for a valid probability distribution. The resulting equation can then be solved for either p or q:

$$1 \overset{!}{=} p + pq + pq^2 + \ldots + pq^{n-1} = p \sum_{i=0}^{n-1} q^i = p \cdot \frac{1-q^n}{1-q}$$

Obviously, if we set $q = 1$ we obtain the uniform distribution again. With $q \in (0,1)$ we assign the highest probability to the smallest value in the domain, whereas $p > 1$ gives preference to the hightest value. Trying different values, we eventually settled with $q = 0.3$ and computed p accordingly each time.

Also note that for $n \to \infty$ we obtain $q \to (p-1)$ and thus a standard geometric distribution. For this reason we will refer to this scheme as the "geometric" distribution in general, even though we have in practice only finite values of n.

3.3 Notes on Randomness

As usual the terms "randomness" and "at random" may be a bit misleading – in fact, for our random choices we use the output of a linear congruential random number generator [9], which is purely deterministic at heart. Thus, for any given random seed (the initial configuration of the generator), the sequence of numbers produced is always the same and we can only try to create the illusion of randomness from an external point of view. That's why these numbers are often referred to as *pseudorandom numbers*.

It has been shown, for instance by Bach [1] and Mulmuley [11], that pseudorandomness still works very well in practice and that the said theoretical shortcoming does not considerably impair the algorithm's performance.

Now it is clear that, for a given problem instance, each run of the randomized backtrack search algorithm is solely dependent on the random seed[1]. For a certain seed the search will always explore the search space in the same way – in particular the same solution will be produced and an identical number of backtracks will be required.

With this in mind it is understandable why this concept is sometimes also referred to as "deterministic randomness". From a commercial point of view, however, this is actually a welcome and sometimes even necessary property, since it enables unproblematic reproduction of results, for instance for debugging purposes.

3.4 Computational Results

For our performance tests we obtained several real-world instances of the tail assignment problem from Carmen Systems, varying in size and complexity:

- `1D17V`: 17 vehicles (only one type), 191 activities over one day.
- `1W17V`: 17 vehicles (only one type), 727 activities over one week.
- `1D30V`: 30 vehicles (three different types), 129 activities over one day.
- `3D74V`: 74 vehicles (nine different types), 780 activities over three days.

In the first two instances, `1D17V` and `1W17V`, all aircraft are of the same type, therefore there are almost no operational constraints as to which vehicle may and may not operate a flight. The two instances `1D30V` and `3D74V`, however, comprise several different types of aircraft and hence exhibit numerous operational constraints.

All practical tests were performed using the Gecode constraint programming environment [15]. The package's C++ source code is freely available, hence it was rather easy for us to implement our custom propagators and add randomization to the search engine.

We first tried to solve all instances with the nonrandomized backtrack search, using different variable selection heuristics and the smallest-value-first value selection heuristic . The results are shown in Table 1. As expected, given the results

[1] Note that the nonrandomized run can be identified with a specific randomized run resulting from a suitable random seed.

Table 1. Backtracks before finding a solution using nonrandomized search

	`1D17V`	`1W17V`	`1D30V`	`3D74V`
min-size	8	1	> 1000000	5
min-degree	14	1	> 1000000	> 1000000

from Gabteni and Grönkvist [3], the two instances involving just one vehicle type can be solved rather easily, with only a few backtracks. The more constrained instances, however, exhibit more problematic runtime behavior: We aborted the search after running without result for several hours on a 2 GHz CPU. In the light of our further results, we regard the short run using the *min-size* heuristic on the `3D74V` instance as a "lucky shot".

To assess the impact of the randomized extensions specified above, we introduce a random variable X denoting the number of backtracks needed by the randomized backtrack search instance to find a solution. We are then interested in the probability of finding a solution within a certain number of backtracks x, formally $P(X \leq x)$.

In Fig. 1 we plot this probability for all four problem instances, using the geometric value distribution and both *min-size* or *min-degree* with random tie-breaking. For each curve we performed 1.000 independent runs, with different random seeds.

Consistent with the previous findings, the two less constrained instances show a rather satisfactory search cost distribution, with `1W17V` approaching 100% at around 10 backtracks only; at this point `1D17V` is around 60% successful, it reaches 100% after a few thousand backtracks – although the respective CP model has fewer variables, the instances's connection structure appears to be more complex.

However, the situation is not as good for the distributions arising from the more constrained instances: For `1D30V` with the randomized *min-degree* heuristic we get close to 50% after only 10 backtracks, but from then on the probability does not increase notably; with *min-size* the success rate is slightly lower.

The biggest and most constrained instance, `3D74V`, can be solved within 20 backtracks in roughly 40% of the runs, using the randomized *min-size* heuristic. The success ratio then slowly increases towards 45% as we allow more backtracks. The randomized *min-degree* performs considerably worse, with slightly more than 20% success after 20 backtracks, subseqeuently increasing towards 25% with more backtracks.

In a next step we applied the inversely exponential scheme described previously, where each unassigned variable x is selected with probability proportional to $b^{-s(x)}$ ($s(x)$ being the current domain size of x). We tested this for $b \in \{e, 2.0, 2.5, 3.0, 3.5, 10.0\}$ with `1D30V` and `3D74V` and compared it to the best solutions from Fig. 1. The results are given in Figs. 2(a) and 2(b), respectively.

For the `1D30V` instance the inversely exponential scheme can partly outperform *min-degree* with randomized tie-breaking. Applied to the `3D74V` instance,

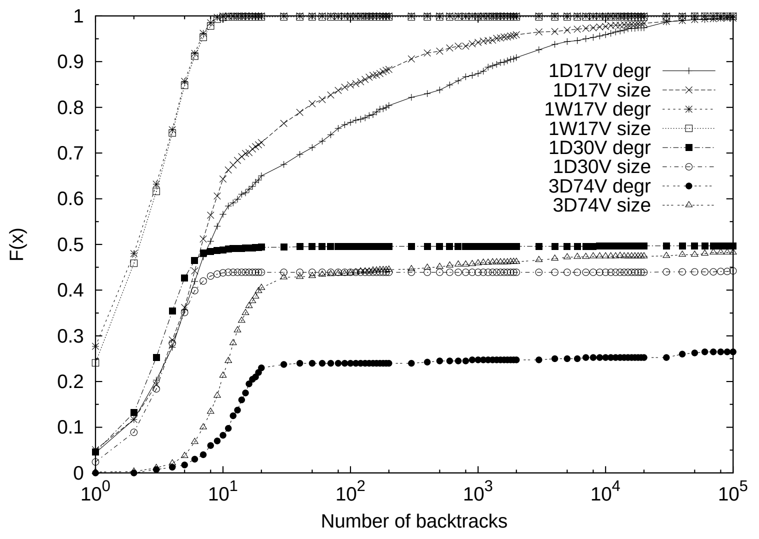


Fig. 1. Plot of the search cost distributions, where $F(x) := P(X \leq x)$

however, it is clearly inferior to *min-size* with randomized tie-breaking, only as b grows does the success ratio get close (since, as noted earlier, for increasing b we approach the randomized *min-size* again).

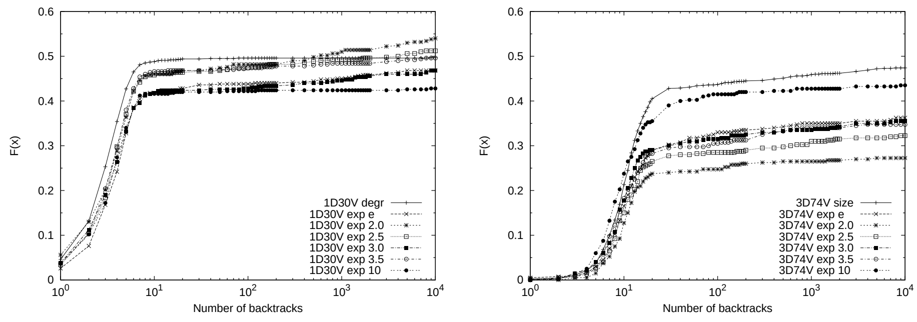
All in all we think that the *min-size* heuristic with randomized tie-breaking is the best choice. Although it does not produce the best search cost distributions in some cases, its performance is never far from the respective optimum.

3.5 Analysis of Results

We have noticed before that the two instances with only one vehicle type involved can be solved rather easily by the nonrandomized backtrack search already. This is also confirmed by the cost distribution for the randomized search and thus not very surprising. The other two instances, however, seem considerably harder.

This behavior can be explained by the presence of *critically constrained variables* in these instances (and the related concept of *backdoors* [13, 14]): Once this subset of critical variables has been fixed, the remaining problem is potentially a lot easier and everything else more or less matches up automatically.

Therefore, if the randomized heuristic picks these variables right at the start and “good” values are assigned, a solution will probably be found within a few backtracks. On the other hand, if we start out with noncritical variables or assign “bad” values, the algorithm will explore large portions of the search space to no avail. The former represents the left-hand tail of the distribution, whereas the latter results in the long, rather flat remaining distribution.



(a) 1D30V (b) 3D74V

Fig. 2. Results for the inversely exponential variable selection

4 Restarts

As we saw in Sect. 3, for the more constrained instances the randomized search can quite easily "get stuck" in the right-hand side of the distribution, so that it will take a lot of backtracks to reach a solution. On the other hand we also observed (via the left-hand tail of the distribution) a nonnegligible probability of finding a solution very quickly, with only a few backtracks.

Naturally we want to exploit this fact; a straightforward way to achieve that is the introduction of restarts into the search procedure: We explore the search tree until a certain number of backtracks has been necessary (the *cutoff value*), at which point we assume that we have descended into one of the "bad" subtrees. Thus we abort and restart the search from the initial configuration – this time hoping to make better randomized choices along the way.

4.1 Restart Strategies

The crucial question is then how many backtracks we allow before restarting. A number of such *restart strategies* have previously been proposed.

Gomes et al. [4] propose a fixed cutoff value, meaning we restart the search every $c \in \mathbb{N}$ backtracks; they call this the *rapid randomized restart* strategy. In their work the optimal cutoff value is determined by a trial-and-error process.

If one has more detailled knowledge about the specific search cost distribution of a problem, one can mostly avoid the trial-and-error approach – however, this knowledge is not always available. Moreover the optimal cutoff value potentially needs to be redetermined for every problem, which does not make this approach very general.

Therefore Walsh [12] suggests a strategy of *randomization and geometric restarts*, where the cutoff value is increased geometrically after each restart by a constant factor $r > 1$. This is obviously less sensitive to the underlying distribution and is reported to work well by Walsh [12] and Gomes et al. [4].

An alternative general approach is the *universal strategy* introduced by Luby et al. [10]. They show that the expected number of backtracks for this strategy is only a logarithmic factor away from what you can expect from an optimal strategy. The sequence of cutoff values begins with 1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,..., it can be computed via the following recursion:

$$c_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \ , \\ c_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1 \ . \end{cases}$$

4.2 Computational Results

We henceforth focus on the two instances `1D30V` and `3D74V`, as their constrainedness and the resulting randomized search cost distribution (cf. Sect. 3.4) predestines them for the introduction of restarts.

We solved each of the two instances with different cutoff values (i. e. the constant cutoff value or the initial value in case of the geometric and universal strategy). For each configuration we ran several hundred iterations with differing random seeds and computed the arithmetic mean of the number of backtracks required to find a feasible solution.

The resulting graphs for the constant cutoff and the universal strategy are given in Fig. 3(a), where the x-axis value is used as a constant multiplier for the universal strategy. For the geometrically increasing cutoff we varied the factor $r \in \{1.1, 1.2, 1.3\}$, the plots of the respective averages are shown in Fig. 3(b).

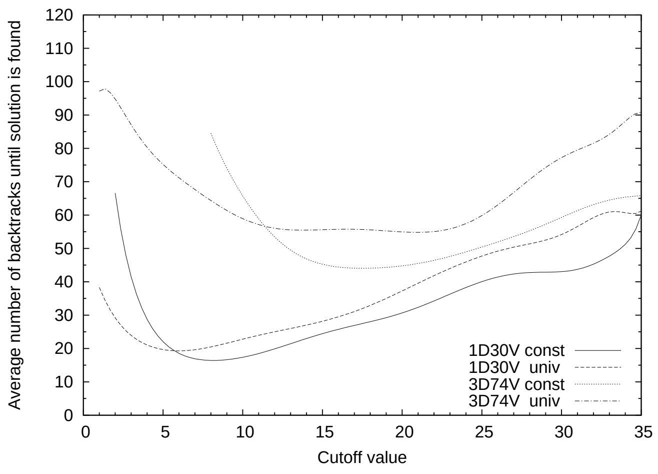
4.3 Analysis of Results

To begin with, we note that the introduction of restarts does exactly what it was intended to do, averting the long tails of the randomized search cost distributions observed in Sect. 3.4. For most configurations a couple of dozen total backtracks suffices to find a solution.

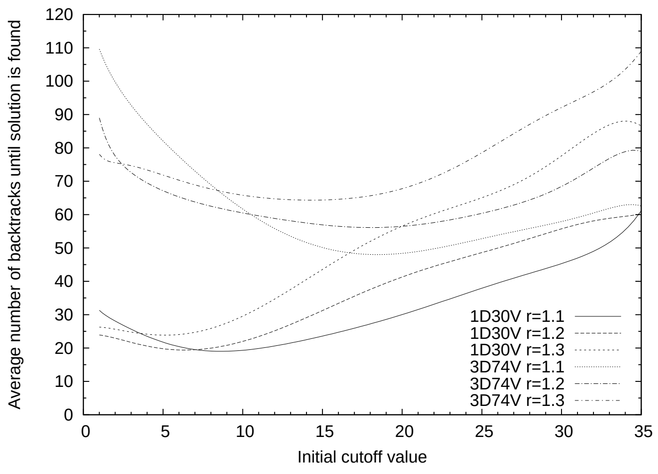
Looking at the left-hand side of the distributions in Fig. 1, it was to be expected that the constant cutoff strategy can deal only poorly (or not at all) with low cutoff values – the chance of a very short run is too small, especially for the `3D74V` instance. Both other restart strategies, with the cutoff increasing over time, can handle these low initial values considerably better, since they will eventually allow a sufficiently high cutoff anyway.

On the other hand, just as noted by Luby et al. in [10], a constant cutoff permits exploiting well-fitting cutoff values more effectively than via the other strategies. This is because, above a certain threshold, increasing the cutoff does not result in a considerably higher probability of finding a solution (cf. the long, almost horizontal tail of the distributions in Fig. 1).

But as we pointed out before, setting an optimal or close-to-optimal cutoff requires knowledge about an instance's search cost distribution, which is mostly not available and may be computationally expensive to obtain. This is



(a) Constant and universal cutoff strategy



(b) Geometric cutoff strategy

Fig. 3. Average number of backtracks after applying restarts to the search procedure

the strength of the variable strategies, where the geometric one seems to hold a slight advantage over the universal one, despite the theoretical logarithmic upper bound on the latter's performance (cf. Sect. 4.1) – the universal strategy's intermediate fallbacks to low values do not fit the distributions at hand.

4.4 Search Completeness

One issue with the randomized extensions as described above is that we sacrifice search completeness: Although the random number generator used for the random choices will most probably be in a different state after each restart, one might still end up making the same decisions as before, thereby exploring the same parts of the search space over and over again. Hence, although the probability is evidently low, it is in theory possible to search indefinitely, either missing an existing solution or not establishing the problem's infeasibility.

In principle, with the geometric and universal restart strategy one will eventually have a sufficiently high cutoff value, so that the whole search space will be explored before restarting and completeness is implicitly ensured. But in practice, given the exponential size of the search space, this will take far too long – also it does not apply to the constant cutoff value strategy.

Therefore we extended the randomized search with a special tree datastructure for the search history, where all visited search tree nodes and “dead ends” (where a backtrack was required) are recorded, thus making sure that the search will not descend into a previously failed subtree.

However, while completeness is a nice theoretic property to attain, we found that in practice it didn't result in enough of a difference to justify the additional processing time and memory consumption, especially since all our instances were known in advance to have at least one solution.

A comparison for the `3D74V` instance, using both the constant cutoff and universal strategy, is plotted in Fig. 4. In some cases the average number of backtracks increases with the introduction of the book-keeping, in some it decreases slightly – probably the differences are partly also due to statistical variations in the relatively low number of runs we performed for each cutoff value.

5 Conclusion and Outlook

Real-world instances of the tail assignment problem impose serious performance problems on standard backtrack search algorithms. Previously this has been solved by the introduction of specialized constraints, that internally make use of the pricing routine of a column generation system.

As an alternative to this we have demonstrated how the use of randomization and restarts can greatly improve the performance of such search algorithms when run on tail assignment instances, reducing the required number of backtracks by several orders of magnitude.

In particular we have shown that with suitable but still generic randomized extensions to the backtrack search procedure we can obtain a substantial probability of finding a solution within just a few backtracks, which we related to the

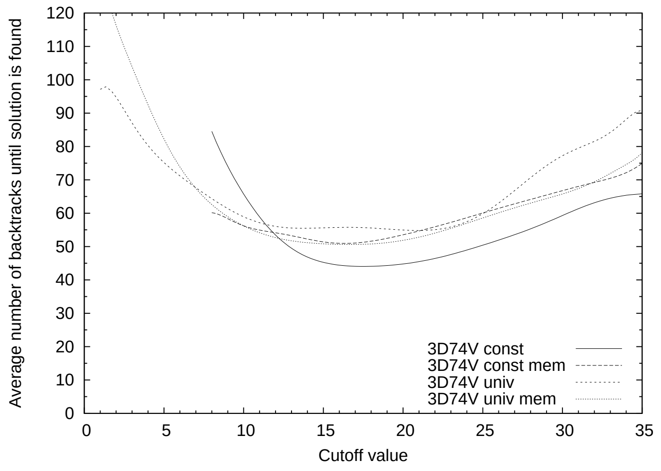


Fig. 4. Impact of memorizing the search history across restarts

concept of critically constrained variables and backdoors. However, depending on the random choices throughout the search process, we still encounter a lot of very longs runs as well.

Therefore we added restarts to the search engine, which, as we noted, has proven rewarding for other authors before [5, 12]. The intention in mind is to exploit the presence of relatively short runs, at the same time avoiding to "get stuck" in the long tail of the search cost distribution. The presented results confirm that this idea works very well for practical purposes.

We have also argued that the randomness is kept "controllable", thereby ensuring reproducibility, which is an important consideration for a potential deployment in a commercial system.

So far we have not been able to experiment with a number of really big problem instances, spanning over a month and comprising well above 2000 activities. This was due to memory limitations on the machines we had at our disposal, in connection with the underlying concept of the Gecode environment, which employs *copying and recomputation* rather than the potentially more memory-efficient *trailing*. However, based on our results we believe that these instances would profit from our approach as well.

To summarize, we feel that randomization and restarts are an effective yet general way to combat computational hardness in constraint satisfaction problems. Consequently, as Gomes et al. note [4], this concept is already being deployed in a number of production systems.

In fact, given that our findings show great potential, the possibility of extending the current Carmen Systems tail assignment optimizer accordingly will be investigated further. In this respect it will certainly be interesting to explore how well randomization and restarts interact with the aforementioned specialized constraints currently used in the system.

Acknowledgments

We would like to thank the developers of Gecode, Christian Schulte, Guido Tack and Mikael Lagerkvist, for their constant willingness to answer our questions and their help with a number of implementation pitfalls.

References

1. E. Bach: Realistic analysis of some randomized algorithms. *Journal of Computer and System Sciences* **42** (1991): 30–53.
2. C. Barnhart, N. L. Boland, L. W. Clarke, E. L. Johnson, G. L. Nemhauser, and R. G. Shenoi: Flight string models for aircraft fleeting and routing. *Transportation Science* **32** (1998): 208–220.
3. S. Gabteni and M. Grönkvist: A hybrid column generation and constraint programming optimizer for the tail assignment problem. In *Proceedings of CPAIOR'06.*
4. C. Gomes, B. Selman, N. Crato, and H. Kautz: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning* **24** (2000): 67–100.
5. C. Gomes, B. Selman, and H. Kautz: Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98).*
6. R. Gopalan and K. T. Talluri: Mathematical models in airline schedule planning: A survey. *Annals of Operations Research* **76** (1998): 155–185.
7. M. Grönkvist: A constraint programming model for tail assignment. In *Proceedings of CPAIOR'04*: 142–156.
8. M. Grönkvist: The tail assigment problem. *PhD thesis, Chalmers University of Technology, Gothenburg, Sweden* (2005).
9. D. H. Lehmer: Mathematical methods in large-scale computing units. In *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machinery (1949)*: 141–146.
10. M. Luby, A. Sinclair, and D. Zuckerman: Optimal speedup of Las Vegas algorithms. *Information Processing Letters, Vol. 47* (1993): 173–180.
11. K. Mulmuley: Randomized Geometric Algorithms and Pseudorandom Generators. *Algorithmica* **16** (1996): 450–463.
12. T. Walsh: Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*: 1172–1177.
13. R. Williams, C. Gomes, and B. Selman: Backdoors to typical case complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03).*
14. R. Williams, C. Gomes, and B. Selman: On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03).*
15. Gecode: Generic constraint development environment. `http://www.gecode.org/`