A best-first-search approach to constraint satisfaction problems[*]

Zoltán Ádám Mann

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Magyar tudósok körútja 2., 1117 Budapest, Hungary
e-mail: zoltan.mann@gmail.com

Tamás Szép

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Magyar tudósok körútja 2., 1117 Budapest, Hungary
e-mail: szep.tamas@mail.datanet.hu

Abstract: Backtrack-style exhaustive search algorithms for NP-hard problems, such as constraint satisfaction, tend to have large variance in their runtime. This is because „fortunate" branching decisions can lead to finding a solution quickly, whereas „unfortunate" decisions in another run can lead the algorithm to a region of the search space with no solutions.

In this paper, we show how a best-first-search heuristic can be used to mitigate this problem and significantly speed up the algorithm, without sacrificing the optimality of the output.

Keywords: best-first search, backtrack, branch-and-bound, constraint satisfaction problem

1 Introduction

Constraint satisfaction problems (CSPs) often exhibit significant variability in their complexity. Backtrack-style exhaustive search algorithms are especially prone to this kind of behaviour. This is because „fortunate" branching decisions can lead to finding a solution quickly, whereas „unfortunate" decisions in another run can lead the algorithm to a region of the search space with no solutions. This high variability in algorithm runtime poses a significant challenge on its practical application, because it is hard to predict if the algorithm will solve a given problem instance within a couple of seconds or will run for several days (or even longer). This phenomenon is quite common in the case of exact algorithms for NP-hard problems [2, 1, 3, 5].

A possible remedy for this issue, that has been suggested in the literature, is restarting [4]. If an algorithm involves random choices, it might make sense to run it several times on a given problem instance. For example, suppose that the median runtime of a random algorithm on problem instances of a given size is 1 minute. Assume that it has been running on a problem instance for 5 minutes without any results yet. Intuitively, one could think that the algorithm will

[*]This paper was published in: *Proceedings of the 7th Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, pages 409-418, 2011.

most probably finish very soon, so we should keep waiting. However, empirical evidence shows that – for many exact algorithms for NP-hard problems – it is better to stop the current run of the algorithm and restart it. The rationale is that it might actually happen with surprisingly high probability that the current run of the algorithm will take several hours, days, or even longer. On the other hand, if we restart the algorithm, chances are high that the next run will be more fortunate and may finish in a minute or so.

Thinking of a backtrack search, the reason why restarting improves the performance of the algorithm is that this way long useless searches in areas of the search tree with no solutions are stopped; the restarted search might be more lucky and finds its way directly to a more promising part of the search tree.

Although restarting works quite well in practice, it is a very simplistic approach to solve the problems with backtrack search. In a way, it is a brute-force approach because there is no guarantee whatsoever that the new run will be better; instead, the rationale is that among several runs of the algorithm, there will be probably a lucky one.

In this paper, we propose a more sophisticated approach. We observed that the problem with backtrack search is rooted in its depth-first-search nature. This is why it cannot „give up" searching a useless part of the search tree and move on to other, more promising areas. Therefore, we propose to implement backtrack search with a best-first-search heuristic that will guide it to different parts of the search tree, always aiming for the most promising area. It should be noted though that this modified algorithm is also an exact algorithm. If there is no solution, the modified algorithm will also search through the whole search tree and thus prove the unsolvability. However, if the problem instance is solvable, then this modified algorithm might find a solution much faster than a normal backtrack algorithm.

2 Preliminaries

2.1 Problem formulation

We consider a constraint satisfaction problem with variables $x_1, \ldots, x_n$. The domain of x_i is a finite, non-empty set denoted by D_i, consisting of the possible values for variable x_i. We are also given a set of constraints $C_1, \ldots, C_m$. Each C_j is a pair (V_j, R_j), consisting of a subset of the variables $V_j \subseteq \{x_1, \ldots, x_n\}$ and a relation R_j. If $V_j = \{x_{j_1}, \ldots, x_{j_k}\}$, then $R_j \subseteq D_{j_1} \times \ldots \times D_{j_k}$. R_j defines which tuples of possible values of the involved variables satisfy the given constraint. The aim is to assign to each variable a value from its domain, such that all constraints are satisfied. If a constraint C_j is not satisfied, then there is a *conflict* between the values of the variables in V_j. The variables x_i and x_j are *neighbours* if there is a constraint containing both of them.

2.2 The backtrack algorithm

The main idea of the algorithm is that it assigns values to the variables, one at a time, as long as no conflict occurs. If all variables can be assigned a value this way, the algorithm terminates. On the other hand, if there is a conflict, the algorithm backtracks, i.e. it goes back to the last consistent state by undoing the last assignment. Then it proceeds to an unexplored branch by trying a new value assignment for the currently selected variable. When all possible branches from a given state have been tried without success, the algorithm backtracks.

The algorithm traverses the space of partial solutions in a tree structure. There are two possible termination situations: either a solution is found, or the algorithm checks all branches from the root of the tree without success, and tries to backtrack from the root. In this case, we can be sure that the input problem instance is unsolvable. Clearly, the algorithm terminates in finite time, since the size of the complete search tree is an upper bound on the number of steps of the algorithm. Unfortunately, this number is exponentially high. However, in many cases the algorithm can prune large subtrees of the search tree, which can considerably decrease its runtime.

We use the number of backtracks to characterize the number of steps of a run of the algorithm in a machine-independent manner.

3 Used improvement techniques

Beyond the basic backtrack algorithm described above, we used a number of techniques to make our algorithm competitive [6]. In the following, we describe these techniques.

Variable selection: we use the MRV (Minimum Remaining Values) heuristic, which selects the variable with the least remaining choosable values in its domain. In case of a draw, we use the degree heuristic, which chooses the variable with the most free (not yet assigned and not unimportant) neighbours. If there is still a draw, we choose simply the variable with the lowest index. After choosing a variable to assign, we should sort its assignable values in ascending order of possible conflicts after the assignment.

Initial assignments: in many CSP problems, we can easily find some heuristic algorithm that returns a set of assignments, for which it holds that the problem is solvable if and only if it is solvable after fixing this set of assignments. For example, in graph colouring, an initial set of assignments is to colour a clique of the graph.

Unimportant variables: a variable, for which after any consistent assignment of the remaining variables surely a choosable value remains, is called an unimportant variable, and can be removed without affecting solvability. Two simple examples are the following. If we can surely state, that a variable has more choosable values in its domain than the number of values that the remaining (not yet satisfied) constraints can possible remove, then it is an unimportant variable, because we can surely satisfy all its constraints. In the second case, the variable X is unimportant, if there is another variable Y, such that the domain of Y is a subset of the domain of X and the constraint set of Y includes the constraint set of X. This is because in this case the same assignment for X as the one for Y will satisfy all its constraints.

Maintaining edge consistency: the aim of this technique is to detect failures earlier. If in a search state there is a variable X, to which it is only possible to assign one value, then we can remove from each neighbour Y all those values that are not compatible with the only possible value of X.

Symmetry breaking: in many CSP problems, two different assignment sets may describe the same situation. For instance, in graph colouring all permutations of the colors describe essentially the same colouring, creating many equivalent branches in the search tree. We should search through one such branch and prune all the other equivalent branches.

Conflict driven backtracking: if the variable selection heuristic is not „wise” enough in the current situation, it is possible that an inconsistent state occurs in such a way, that some ancestors of the current node (*conflicted node*) in the search tree are not responsible for failure of the search subtree of the current node. In such a situation, we can directly backtrack to the *last conflicting node* in the search tree, as shown in Figure 1. Using such a backtracking requires to administrate the *usual conflict sets* of the free variables, in which the causers of their own conflicts are stored, and the *collecting conflict sets* of the assigned variables, in which the causers of their branch's failure are stored.

After every forward step, we have to maintain the usual conflict sets of the currently free variables. If we use edge consistency maintaining and a conflicted-conflicting relationship arises between two free variables, then the usual conflict set of the conflicting node must be inserted in the usual conflict set of the conflicted node.

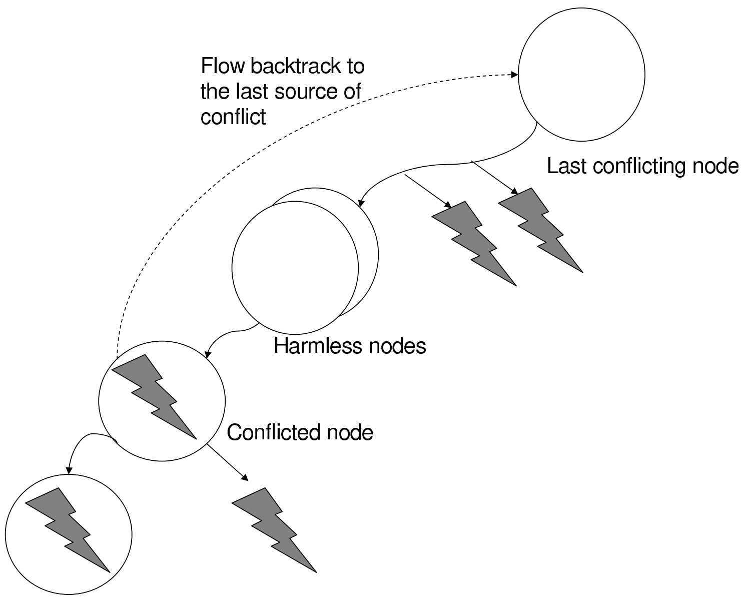


Figure 1: Schematic example of conflict-driven backtracking

If there is a direct conflict under the last assigned variable X, then we have to insert the whole usual conflict set of the conflicted variable Y to the collecting conflict set of X. If there is an indirect conflict under an assigned variable X and we backtrack to X, then we have to insert the whole collecting conflict set of the variable Y, from which we directly backtracked to X, to the collecting conflict set of X. (Note that in both cases the inserted conflict sets will surely contain X as the last conflicting node, so before starting to backtrack from X, we first have to remove it from its own collecting conflict set.)

If we have to backtrack from a node, then we first have to insert its usual conflict set into its collecting conflict set that will represent the whole conflict set of the branch, and then start the backtrack. Such a backtrack, which may include several normal backtracks, will be referred to as a *flow backtrack*. A flow backtrack is stopped when we reach the last conflicting node in the search tree.

In the example of Figure 1, conflicted node represents an assigned variable, and we returned to it for some reason. If there is a direct inconsistency under conflicted node (right side), then it is sure that we have to backtrack to conflicted node, because before assigning it, there was no inconsistency. In other words, conflicted node surely belongs to the usual conflict set of a free variable (or set of free variables). If there is no direct inconsistency under conflicted node (left side), then conflicted node must be in the collecting conflict set of an assigned variable (not necessarily directly) under it.

4 Best-first-search

If the given problem instance is solvable, then our aim is to find the solution quickly using a best-first-search heuristic; if it is not solvable, then the algorithm should be capable to prove the unsolvability.

Our best-first-search algorithm uses the backtrack algorithm described above. Informally, the idea is that it should run as the backtrack algorithm would, but sometimes it can jump forward in the search, so that it may find a solution faster. We implement this by launching several copies of the backtrack algorithm at different points in the search tree. We start one of these searches and let it run for a while. Afterwards, we pause this run and transfer control to another search instance etc.

4.1 Creating search instances

We must be careful not to waste time by visiting the same node in the search tree more than once. Hence, the algorithm should handle the situation when a search runs into another one. Therefore we decided to implement it in the following manner. We create an *initial search* and some *normal searches.*

For each search S, let $sn(S)$ denote the start node of S, and let $st(S)$ denote the subtree of S, that is, the subtree with root $sn(S)$ of the search tree. A normal search S is only allowed to search within $st(S)$. Moreover, for any two searches S_1 and S_2, $sn(S_1)$ must not be within $st(S_2)$ and vice versa, $sn(S_2)$ must not be within $st(S_1)$. Because of these restrictions, the only way that searches may run into each other is when the initial search runs into a normal search. In this case, it has to merge its own knowledge with that of the normal search and continue with its current state. (The merging will be explained in detail.)

It is important that all searches must run in the same search tree so that we can join them. Hence, we use a deterministic backtrack algorithm, and so all search instances work in the same way. Assuming such a backtrack algorithm, a node in the search tree can be exactly described by the chosen values for the variables in the order as they were assigned, so we only have to use the vector of chosen values to determine the actual search path. Checking whether a search has run into another one is costly, so we decided to check it only after every forward step. This works because it is not allowed to start a search from a node, the subtree of which contains the start node of another search.

The creation of the searches is carried out by emulating the behaviour of the backtrack algorithm and steering it by giving the chosen values to the assigned variables. This way, we quickly get a number of different, with each other consistent search instances in the search tree. For creating the initial search, we pick the first possible value for each variable, so that the start node of the initial search will be in the left-most branch of the search tree. When creating other searches, if we decide to emulate a forward step by assigning the ith value to the next variable, we have to remove all of the preceding $i - 1$ possible choices. The only problem, that will be discussed later, is that we cannot maintain the collecting conflict set, which would be used to store conflict information, that led to the removal of these choices. Note that it is not possible to emulate a backtrack because of an external constraint by the problem creation, because in this case we would not be able to create a consistent search regarding the collecting conflict sets.

4.2 Operating the search instances

We run one of the searches for a given number of steps. If it found a solution, we can return with this solution. If it was a normal search and just got back to its start node, we are no longer allowed to use this search. If it was the initial search (which is not stopped at its start node) and it returned that it did not find a solution, then we also return unsolvable.

It is possible that, after a forward step, the initial search IS runs into the start node of another search S. Without conflict-driven backtracking, we could simply remove IS and continue with S, because it would mean that we searched through the whole search tree until the current position of S; thus, we would get a completely consistent search instance. But because of conflict-driven backtracking, S would become inconsistent after backtracking from its start node, because the information about conflicts caused by variable assignments above $sn(S)$ is not known, since S did not actually visit those parts of the search tree. This is why only the initial search is allowed to continue the search after getting back to its start node.

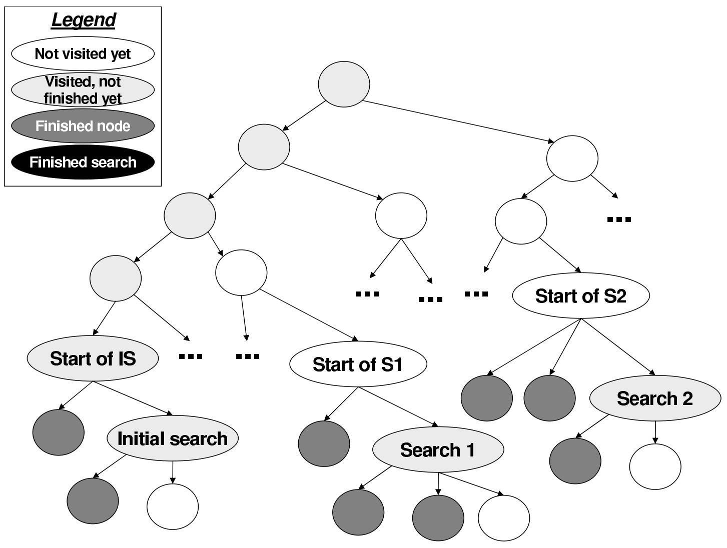
Therefore, we have to complete S with the knowledge of IS about the collecting conflict sets encountered so far. By doing so, we get a fully consistent search object that includes the knowledge of the initial and a now merged normal search. We can now continue S from its current position, and this search will be the initial search from now on. That means we get to a search state that is the same, as if we would have used only one search.

If the problem is unsolvable, then we merge most of the normal searches with the initial search, but not necessarily all of them. Namely, assume the following situation: there are two searches, an initial search IS and a normal search S. The normal search starts from node B in the search tree. The parent of B is node Y, and the parent of Y is node X. Beside B, Y has another child A. IS visits node X, then Y, but before visiting B, it goes to the subtree with root A. This subtree contains no solution, therefore from node A the search must backtrack, but because of conflict-driven backtracking, and assuming that the assignment represented by node Y created no conflicts, IS backtracks directly to node X and will never visit node B. Hence, S will never be merged into IS. This is not a problem, the algorithm still remains exact. This only means that the subtree of B contains no solutions, thus S is useless. We decided not to address this issue, because the described situation happens rarely, and thus the time wasted by such useless searches is marginal. It should be noted that, if the given problem instance is unsolvable, then the steps taken by such useless searches are the *only* overhead (in terms of the number of backtracks) of the best-first-search algorithm compared to the underlying backtrack algorithm. Our empirical results suggest that this overhead is minimal.

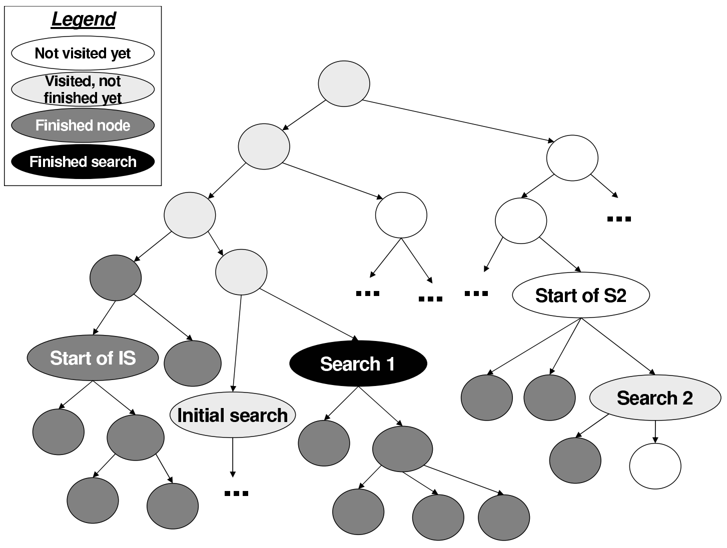
Now we can easily build the best-first-search logic upon this idea. In order to decide which search to run, we have to evaluate the usefulness of each search based on information about its start node and current node, used number of steps, estimated subtree size etc. This evaluation will be referred to as the fitness value of the search. We can simply run the search with the highest fitness value, or we can combine this technique with randomized selection. It is also possible to create new searches during the algorithm, using the knowledge gathered so far.

5 An example

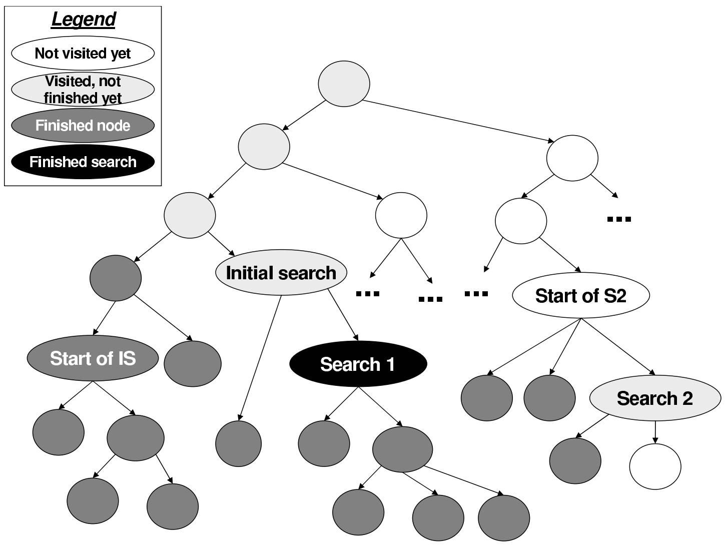
Now we show a simple scenario for an unsolvable problem instance, using 3 searches: an initial and 2 normal searches. First, we let all searches run a little bit:



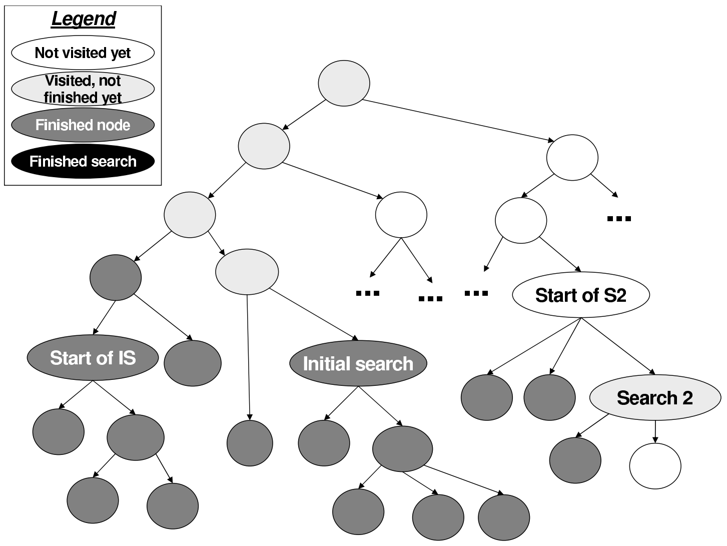
The initial search finishes its original branch and searches further in the search tree. Search 1 finishes its own subtree, so it is not allowed to search further:



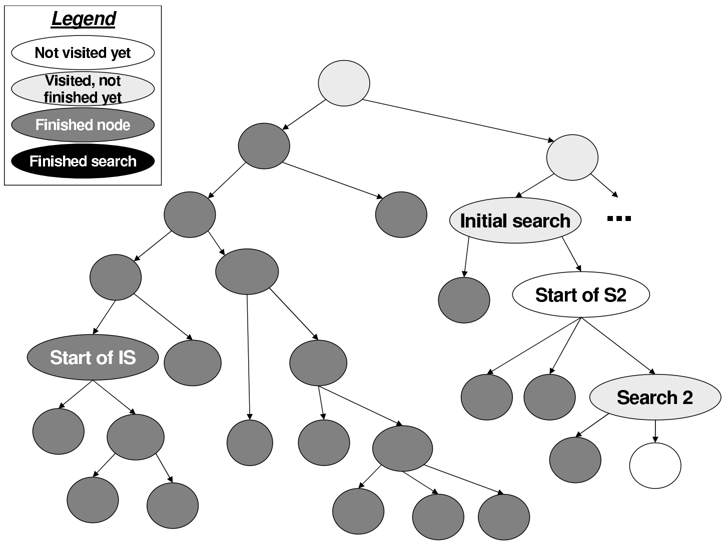
Then, the initial search arrives to Search 1:



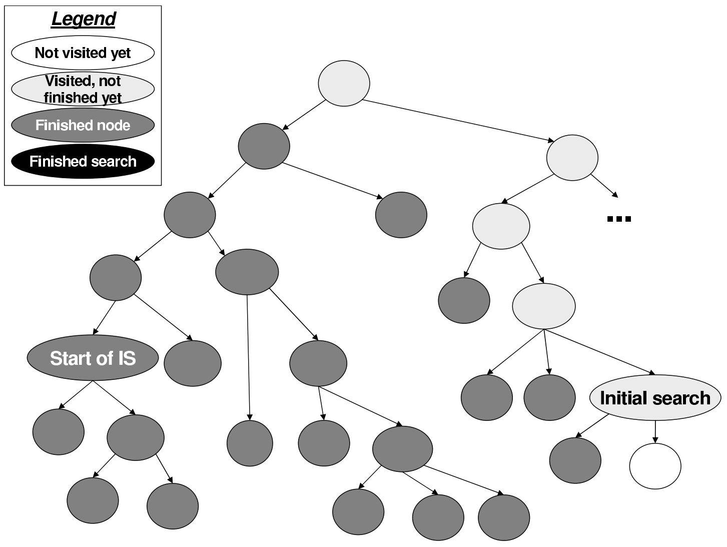
Now, the initial search reaches the start node of Search 1, merges its own conflict knowledge with that of Search 1, and continues with it, so the next track will be a backtrack:



Next, the initial search arrives to the start node of Search 2:



Then, the initial search merges its own conflict knowledge into Search 2 and continues with it, so the next step will be a forward track in the subtree of Search 2. See figure below. Afterwards, the initial search will search through the whole search tree and will backtrack from all branches to the root node with failure.



6 Empirical results

In a set of empirical measurements using graph coloring benchmarks, we compared the efficiency of the basic backtrack algorithm, the backtrack algorithm with frequent restarts as suggested previously in the literature, and the backtrack algorithm with the proposed best-first-search heuristic. The results are shown in Figure 2. As can be seen, both restarts and best-first-search are useful in decreasing the runtime of the backtrack algorithm. However, the best-first-search clearly outperforms the restarts.

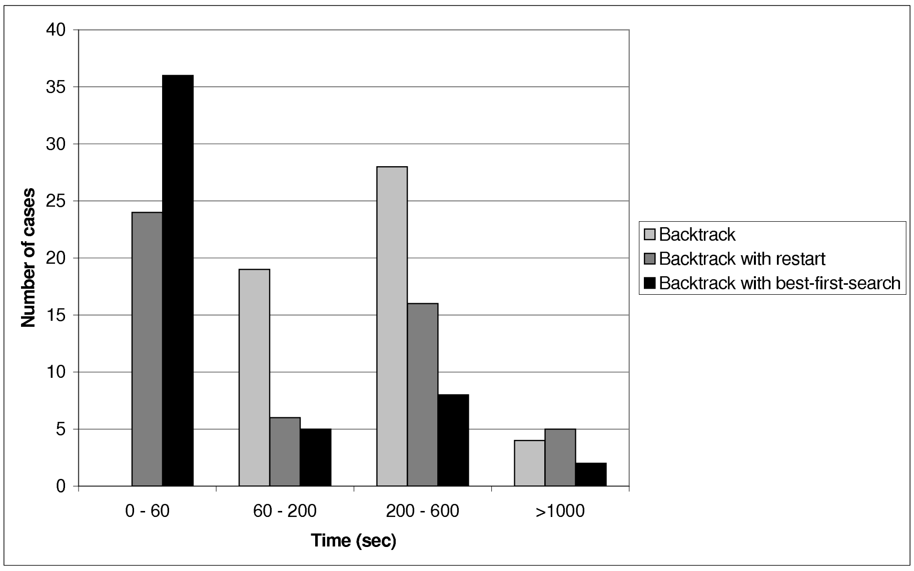


Figure 2: Histogram of the runtime distribution of three versions of the backtrack algorithm

Acknowledgements

This work was partially supported by the Hungarian National Research Fund and the National Office for Research and Technology (Grant Nr. OTKA 67651).

References

[1] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *12th International Joint Conference on Artificial Intelligence (IJCAI '91)*, pages 331–337, 1991.

[2] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1-2):67–100, 2000.

[3] Tad Hogg and Colin P. Williams. The hardest constraint problems: A double phase transition. *Artificial Intelligence*, 69(1-2):359–377, 1994.

[4] Malik Magdon Ismail and Amir F. Atiya. The early restart algorithm. *Neural Computation*, 12(12):2991–3010, 2000.

[5] Haixia Jia and Cristopher Moore. How much backtracking does it take to color random graphs? Rigorous results on heavy tails. In *Principles and Practice of Constraint Programming (CP 2004)*, pages 742–746, 2004.

[6] Tamás Szép and Zoltán Ádám Mann. Graph coloring: the more colors, the better? In *Proceedings of the 11th IEEE International Symposium on Computational Intelligence and Informatics*, pages 119–124, 2010.