Limited Assignment Number Search Algorithm[⋆]

Kamil Veřmiřovský and Hana Rudová

Faculty of Informatics, Masaryk University
Botanická 68a, Brno 602 00, Czech Republic
{xvermir,hanka}@fi.muni.cz

Abstract. A nonsystematic iterative search algorithm for hard or over-constrained problems is proposed. This linear time complexity algorithm seeks a (partial) assignment of the problem variables. One step of the algorithm is a special incomplete version of chronological backtracking with constraint propagation. Subsequent searches try to improve the last computed partial assignment. This is done by developing variable and value heuristics based on the results of previous iterations. The algorithm was applied to solve random problems and a large scale timetabling problem from Purdue University.

1 Introduction

Many search trees arising in practical applications are too large to explore exhaustively. A solution can be found by searching only a small part of the space by following carefully tuned heuristics which guide the search toward the regions of space which are likely to contain solutions. For many problems, this may lead directly to a solution. There is a variety of search algorithms available, e.g., limited discrepancy search [3], backjumping [2], or local search with constraint propagation [4]. Our intent is devoted to the problems where it is difficult or impossible to find a solution. The heuristics may not be good enough or/and the problem may be over-constrained. We propose a linear time iterative search algorithm which can be used in this case. This algorithm was defined as an approach to solve a large scale timetabling problem from Purdue University [5].

The search directed by this algorithm is incomplete (i.e., overall search tree is not explored) and does not necessarily find a complete solution (i.e., only a partial assignment may be found). The aim is simply to assign most of the problem variables. The algorithm also develops its own variable and value ordering heuristics iteratively. They can be used together with the heuristics we have already developed for the problem to improve the current solution. The heuristics that are constructed can be influenced after each iteration by a user who can direct continuation of the search. Another possibility is to relax the constraints in the problem and continue with the relaxed problem in subsequent iterations.

[⋆] This work is partially supported by the Grant Agency of Czech Republic under the contract 201/01/0942 and by Purdue University. We would also like to thank to the Supercomputer Center Brno where the experiments with random problems were accomplished.

2 Description of LAN Search

The *limited assignment number (LAN)* search algorithm is based on backtracking with constraint propagation [2]. The aim is to always find a solution (though not necessarily a complete one) of a constraint satisfaction problem [6].

The object of a *constraint satisfaction problem* is to find an assignment of values to given variables from domains which satisfy the given conditions (*constraints*). Such a problem is *over-constrained* if it is not possible to satisfy all constraints. While classical backtracking would explore the overall solution space, constraint propagation can efficiently prune it. At each step, an uninstantiated variable is chosen and a value is assigned to it (called *labeling*). Each value assignment is propagated through the constraints into the domains of other variables.

For each variable, the LAN search algorithm maintains a count of how many times a value has been assigned to it. A *limit* is set on this count. If the limit is exceeded, the variable is left unassigned and the search continues with the other variables. A labeling of unassigned variables is not processed even during backtracking. As a result of this search, a partial assignment of variables is obtained together with the set of the remaining *unassigned variables.* The limit ensures the finiteness of the search.

The result of this search is used in the subsequent iterations. The LAN search develops the following variable and value ordering heuristics based on the results of the former iteration:

- values of variables successfully assigned in the previous iteration are tried first during the following iteration — once a suitable value for the variable has been found it remains a promising assignment for this variable;
- each variable left unassigned in the previous iteration gives values to be tried last (those values that were tried unsuccessfully) — the suitable value will probably be among those values not tried in the last iteration;
- any variable unassigned in the previous iteration is labelled first — it may be difficult to assign a value to the variable, therefore it should be given preference in labeling.

In the first iteration of the algorithm, we can use either problem-specific or standard heuristics (*initial heuristics*) like first-fail [6]. In each of the following iterations, heuristics based on the previous iteration are used primarily. Any ties are broken by using the initial heuristics. The user may also manually modify the results after each iteration to influence the behavior of the developed heuristics or relax the constraints to eliminate contradictory requirements.

There are two cases when the LAN search does not help and all variables remain unassigned. It may fail as a consequence of the initial constraint propagation prior to labeling. Here the user should relax some constraints and restart the algorithm. The search may also fail if the domain of a variable is emptied before its limit is exceeded and, it is the first variable or all of its preceding variables have already exceeded the limit. In this case, the algorithm can be restarted without these variables. A detailed analysis of such situations will be the subject of our future work.

The current assignment limit is set to the maximal domain size d of any labelled variable. As each of n variables can be tried d times, one iteration of the LAN search is of linear complexity $\mathcal{O}(dn)$.

The pseudo-code of the algorithm can be found in Appendix A.

3 Description of Experimental Problems

The *random placement problem (RPP)* seeks to place a set of rectangles (called *objects*) of different sizes into a larger rectangle (*placement area*) in such a way that no objects overlap and all object borders are parallel to the border of the placement area. In addition, each object is defined by a set of allowable axial coordinates for the object's bottom-left corner, zero being the bottom-left corner of the placement area. The ratio between the size of the placement area and the total area of all objects is called the *filled area ratio*. The higher the filled area ratio is, the more constrained the problem is.

We have proposed the RPP because it allows us to generate various instances of the problem similar to the timetabling problem we are looking to solve. The objects correspond to courses to be timetabled — the x-coordinates to different times, the y-coordinates to different classrooms. For example, a course with three time units corresponds to an object with dimensions 3x1 (course should be taught in one classroom only). Each course can be placed only in a classroom of sufficient capacity — each object will have a randomly generated lower y-bound.

The timetabling problem from Purdue University (TPPU) consists of timetabling approximately 750 classes attended by 30,000 students into 41 large lecture rooms with capacities up to 474 students. The classes are taught several times a week resulting in 1,600 meetings (objects in RPP) to be timetabled. The space covered by all meetings fills approximately 85 % of the total available space (filled area ratio). Special meeting patterns defined for each class direct possible time and location placement. Classroom allocation must respect instructional requirements and preferences of faculty. All instructors may have specific time requirements and preferences for each class. A major objective is to minimize the number of potential student course conflicts. A full description of the problem together with the presented results can be found in [5].

4 Empirical Results

The problem solvers were implemented in the CLP(*FD*) library of SICStus Prolog [1] version 3.9.1. Results presented were accomplished under Linux on a PC with an AMD Athlon 850 MHz processor with 128 MB of memory.

We have generated 7 sets of RPPs each of 50 problems and 200 objects distinguished by the filled area ratio (see the Table 1). The average lower bound (34 %), object sizes and placement area size were chosen to correspond to the TPPU and to meet the required filled area ratio. The problems in the last two sets are overconstrained, but this may also be true for other problems with the filled area

Table 1. Description of the generated random placement problems

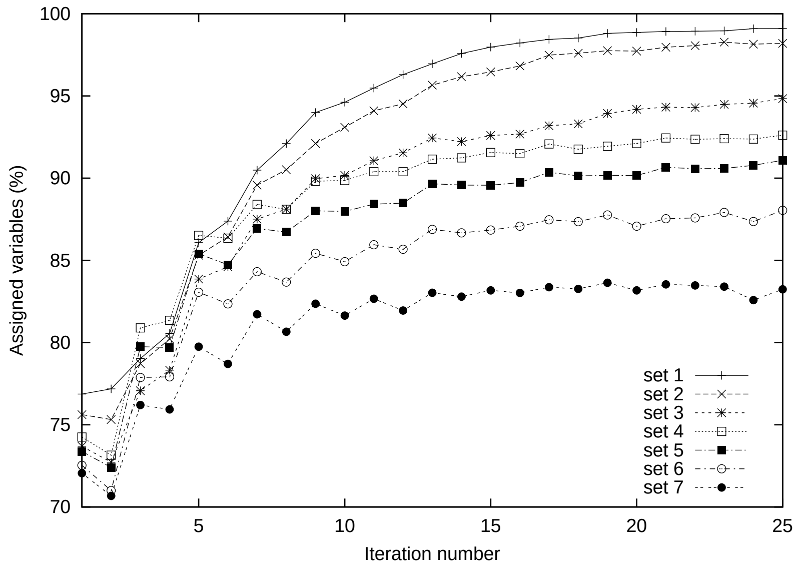
Problem set number	1	2	3	4	5	6	7
Filled area ratio (%)	80	85	90	95	100	105	110
Placement area	40x14	38x14	35x14	36x13	35x13	33x13	33x12

Objects	2x1	3x1	4x1	6x1
Occurrences of objects (%)	80.4	16.6	2.6	0.4

ratio "close" to 100 %. The initial variable ordering applied was first-fail, values in the domain were explored in ascending order in the first iteration. Each problem was solved using 25 iterations. Each iteration took in average 10–60 seconds. The average percentage of assigned variables after each iteration for each of the seven sets of problems is shown in Fig. 1. The complete assignment of variables was found for 32, 21, and 4 problems with the filled area ratio 80, 85, and 90 %, resp. No problem was completely solved in the data sets with the higher filled area ratio. Problems solvable by LAN search were also tested by backtracking and limited discrepancy search (LDS) applying our initial value and variable ordering heuristics. No problem was solved within 10 hours time limit by backtracking and LDS (with up to 3 allowed discrepancies).

The solution of the TPPU was implemented with the help of a new soft constraints solver. Experiments with standard backtracking did not lead to a solution after 10 hours of run time (too many failed computations were repeated exploring parts of search tree with no solution). The LAN search algorithm was

Fig. 1. Average percentage of assigned variables



able to find solution with 99.6 % of classes assigned. This solution was found in 5 iteration steps where the number of unassigned classes was 19, 15, 10, 5, and 3, respectively. Subsequent iterations made no further improvement. One iteration took 2–3 minutes. Let us take a look at a short summary of the other results (for details see [5]). The final solution was able to satisfy 98.1 % of the student requirements from course pre-enrollment, 79.7 % of classes were assigned at the preferred times while 4.0 % classes must be taught at discouraged times. A secondary requirement on selection of preferred classroom was satisfied up to 49.0 %.

5 Conclusion and Future Work

We have proposed an incomplete, iterative search algorithm with linear time complexity computing a (partial) assignment of variables. It is suitable for over-constrained problems or hard problems where other search algorithms have not succeeded. For such problems, it is important to compute at least a partial solution, as a complete solution may not exist, or may be difficult to find.

Our algorithm was verified for random problems and for a large scale timetabling problem. Amount of assigned variables increases with the iteration number. The more constrained the problem was, the less complete solution was found. At the same time, it is clear that it is not possible to assign all variables for many problems (at least 9 % of the variables could not be assigned for the most constrained problem). Neither backtracking nor LDS were able to solve the problems solvable by LAN search. In the real timetabling problem from Purdue University, we were able to assign almost all classes and also achieve very good results in the optimizations.

Currently we are improving the heuristics to reflect all previous iterations, not just the last one. For each variable, we would like to maintain the number of iterations where it was unassigned and use this information during following iterations. As a part of our future research, we would like to compare the LAN search with more algorithms and test it on other types of problems. Future work will also focus on analysis of a suitable value for the assignment limit and the number of iterations. The dependence of the algorithm on the initial heuristics will be explored. We would like to extend the algorithm towards problems where all variables remain unassigned.

References

1. Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Programming Languages: Implementations, Logics, and Programming*. Springer-Verlag LNCS 1292, 1997.
2. Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.
3. William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–615. Morgan Kaufmann, 1995.

4. Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.
5. Hana Rudová and Keith Murray. University course timetabling with soft constraints. In Edmund Burke and Patrick De Causmaecker, editors, *PATAT 2002 — Proceedings of the 4th international conference on the Practice And Theory of Automated Timetabling*, pages 73–89, 2002.
6. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

A LAN Search Algorithm

The function *LAN_Search* implements one iteration of the algorithm. It should be called with the list of *Variables* to be labelled and the assignment limit, the parameters *Unassigned* and *Constraint* should be set to $\emptyset$. Lists of *Assigned* and *Unassigned* variables are returned as a result. The functions *select_var* and *select_val* represent the variable and value ordering heuristics.

```
function LAN_Search(Variables, Unassigned, Constraint, Limit)
  if Constraint $\neq \emptyset$ then
    Success $\leftarrow$ propagate(Constraint, Variables)
    if Success = fail then
      return $\langle$fail, Unassigned$\rangle$
Instantiated $\leftarrow$ instantiated variables from Variables
RestVars $\leftarrow$ Variables $-$ Instantiated
if RestVars = $\emptyset$ then /* no variables left for labeling */
  move instantiated variables from Unassigned into Instantiated /* some
    variables from Unassigned could have got assigned due to propagation */
  return $\langle$Instantiated, Unassigned$\rangle$
X $\leftarrow$ select_var(RestVars)
if current assignment number of X $\leq$ Limit then
  increase current assignment number of X by 1
  Value $\leftarrow$ select_val(X)
  $\langle$Assigned, Unassigned$\rangle$ $\leftarrow$
    do_labeling(RestVars, X = Value, Unassigned, Limit)
  if Assigned = fail then
    $\langle$Assigned, Unassigned$\rangle$ $\leftarrow$
      do_labeling(RestVars $-$ Unassigned, X $\neq$ Value, Unassigned, Limit)
else
  /* max assignment limit exceeded, skip the variable */
  $\langle$Assigned, Unassigned$\rangle$ $\leftarrow$
    do_labeling(RestVars $-$ {X}, $\emptyset$, Unassigned $\cup$ {X}, Limit)
if Assigned = fail then
  return $\langle$fail, Unassigned$\rangle$
else
  return $\langle$Instantiated $\cup$ Assigned, Unassigned$\rangle$
end
```