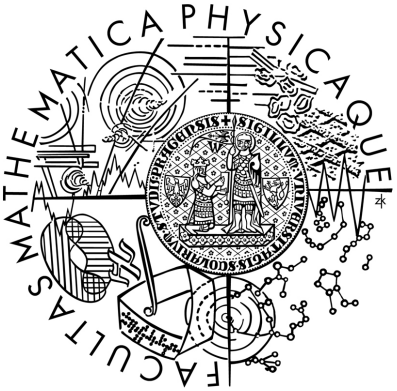
Charles University, Prague, Czech Republic
Faculty of Mathematics and Physics

MASTER THESIS



Jakub Lehotský

Incomplete Search Techniques

Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Doc. RNDr. Roman Barták, Ph.D.
Study program: Computer Science, Theoretical Computer Science

.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on 10^{th} April 2011 Jakub Lehotský

Contents

Abstract

Název práce: Neúplné vyhledávací algoritmy
Autor: Jakub Lehotský
Katedra: Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: Doc. RNDr. Roman Barták, Ph.D.
e-mail vedoucího: bartak@ktiml.mff.cuni.cz
Abstrakt: Problémy s omezujícími podmínkami jsou množinou diskrétních optimalizačních problémů, které řeší mnoho problémů ze skutečného života. Jsou řešeny inferenčními a vyhledávacími algoritmy. Ve většině případů úplné vyhledávací algoritmy dokážou najít řešení, existují ale problémy, kterých zložitost je příliš vysoká na to, aby byl prostor řešení prozkoumán kompletně. V týchto případech musíme zavést omezení, které ořežou velikost stavového prostoru pro vyhledávání. Algoritmy založené na diskrepancích omezují počet případů, kde se algoritmus rozhodne proti dané heuristice. Neúplné vyhledávací algoritmy negarantují nalezení řešení. Mnoho vyhledávacích algoritmů má vlastnost any-time, která nám poskytuje nějaké řešení optimalizačního problému v libovlný časový okamih, i když řešení ještě není optimální.

Klíčová slova: *CSP, neúplné vyhledávání, vyhledávání s diskrepancemi*

Title: Incomplete Search Techniques
Author: Jakub Lehotský
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Doc. RNDr. Roman Barták, Ph.D.
Supervisor's email address: bartak@ktiml.mff.cuni.cz
Abstract: The constraint satisfaction problems is set of discrete combinatorial problems which address to solve many of the real life problems. They are commonly solved by inference and search algorithms. In most cases the complete search algorithm can find a solution to a problem, but in many cases, search space is too large to be explored completely. In these case the limitation of search space is necessary in a way which gives us some way to still find a solution without having to search whole search space. Discrepancy-based search algorithms limit the search space by limiting the number where search decisions go against the heuristic in given search. Incomplete algorithms don't guarantee finding a solution. Many incomplete algorithms have any-time property useful in optimization problems, where algorithm provides some solution at any time even if it is not an optimal one.

Keywords: *CSP, incomplete search, discrepancy search*

1 Introduction

1.1 Constraint Satisfaction Problems

Every day we encounter lot of problems which are defined in the terms of restriction and constraints. These constraints leave us with a space of possible combinations which deem a solution of a problem. Most of the tasks human brain performs every day can be described in terms of constraints. Typical tasks solved by humans are building a monthly budget, planning a schedule to drive children to school and get to work on time within certain time frames, arrangement of seating of guests at family celebration where we need to consider the constraints on who can sit next or opposite to other persons. These tasks can be quite often solved by simple human reasoning. When it comes to more complicated problems such as creating timetable for university, planning a plane production or army operation logistic strategies, the complexity of tasks rises to very high level and we need advanced computer power to get the solution.

Similarly as we would solve these every day tasks of real life, there are two basic methods of solving constrained problems. First method is inference. We keep deducing new constraints from the already existing ones up to the point where the solution can come immediately or we come to the conclusion that there isn't any solution. The second approach is simple trial and error. This is basis of search algorithms which systematically keep trying to fill in values and simply check if they match the defined constraints until a solution is found.

Both inference and search form an extensive groups of algorithms which has been in focus of research in artificial intelligence for decades. Very often, both of these approaches are combined in the algorithms giving better results than if used separately.

1.2 Incomplete Search Algorithms

In this work we will briefly mention inference algorithm and focus on search algorithms. The real-world problems to be solved are often very large, in fact we can see that CSP problems are NP-complete in general (for example k-colorability, SAT among others can be modeled as a CSP), which leave us with very large search spaces. In order to be able to have some feasible algorithms that give us possibly a result in a reasonable time, we need to restrict the search space, which leads us to incomplete search algorithms.

There are some differences between basic ideas between complete and

incomplete algorithms. Complete algorithm try to search state space systematically. In systematic search, in each step of algorithm, next generated and tested state is very close to the previous one. Incomplete algorithms try to spread the effort over the search space more evenly. Most of incomplete algorithms are designed in a way that they can be restarted and they search larger portion of search space in each iteration. If the extension of a search space in each restart is systematic, these incomplete algorithms can be turned into complete ones if arbitrary amount of time is provided.

There is a specific subclass of incomplete algorithms which are designed to be used together with heuristics. Heuristic is a helping function which guides the search towards the solution. It is defined over the search states (partial assignments for CSPs) and it provides the next state in search state. In order for heuristic to be useful the probability that it leads toward a solution must be greater than 0.5. We need to address the situation where the heuristic does not lead directly to the solution. We assume that the solution is close to the heuristic recommendations. Discrepancy based incomplete algorithms try to limit the number of times when we decide to search the branch against the heuristic decision. These decision in states where the algorithm doesn't follow heuristic are called discrepancies.

Another quite widely spread group is local search algorithms. Instead of systematically trying to assign values to a variables and see if they match defined constraints, these algorithms randomly assign all the variables and they try to modify these assignments in each step trying to satisfy more constraints in each step eventually getting the solution. Focus of this thesis is on depth-first search algorithms, so local search mehotds are mentioned just briefly.

1.3 Goal of the thesis

The goal of this thesis is to provide survey of currently existing incomplete search algorithms that are in use and have been published. Then design a new algorithm or an improvement of some of already existing algorithm in some real-life applications. Theoretical comparison is possible and has been done for some general incomplete search algorithms, however most of specific real-life problems and their heuristics are modelled in a way, where empirical research is more appropriate than theoretical calculations.

Various frameworks were considered for empirical comparison. The prolog language has advantage of natural definition of the problem itself but has disadvantage that many data structures needed for certain algorithms cannot be simply implemented and many walk-arounds are used. C++

and Java frameworks share common drawbacks of procedural language. Algorithms have to be written explicitly and not in a declarative way, but the implementation of all the data structure is direct. Due to vast experience with implementation of Java, I have chosen this language.

1.4 Structure of the text

In the Section 2, we put all the formal definitions which form the basis of future explanations. Section 3 contains introduction of search algorithms and short summary of other methods, as they are often used to improve search algorithms. Section 4 contains introduction to search algorithms and description of general incomplete search algorithms based on limiting the search space according to various criteria. Section 5 contains description of discrepancy based search algorithms which take advantage of heuristics. Section 6 provides possible future work.

2 Formal definitions

We define a Constraint Satisfaction problem (CSP):

Definition 1. CSP is a triple (X, D, C) where:
X is a set of finite *variables* $X = \{x_1, .., x_n\}$
D is a set of *domains* of variables $\{D_1, ..., D_n\}$ such that $x_i \in D_i$ and D_i is a set of variables for each $i = 1, .., n$
C is a set of *constraints* $\{C_1, ..., C_t\}$. Constraint C_i is a subset of Cartesian relation R_i defined on a subset of variables S_i, $S_i \subseteq X$. The relation denotes the variables' simultaneous legal value assignment.

Definition 2. A solution of CSP is such an instation of a problem, where each variable x_i has assigned its value a_i and $a_i \in D_i$ for each i and all constraints are satisfied:
$\forall j \in \{1, ..t\} | (a_1, ..., a_n) \in C_j$

Definition 3. Two CSPs are *equivalent* if they have same solution.

Definition 4. Partial solution

Definition 5. "subproblem"/"consistent subproblem"

3 CSP Solving

We have defined constraint satisfaction problems (CSP) and a solution of a CSP. The first class of CSP solving algorithms described in this chapter is

inference. Inference algorithms are based on the notion that from a given set of constraints, new constraints are deduced and these new constraints further reduce the domain of variables or possible combination of values assigned to a given variable.

3.1 Inference algorithms

Inference algorithms try to crate a new constraints based on the existing ones, these constraints are logically deduced so that we get the CSP that is equivalent with the previous one. In this way we want to introduce consistency on a level where each partial solution can be extended further. For formal background, see [1].

We define arc-consistency as a level of consistency, where any consistent assignment can be extended.

Definition 6. Variable x_i is $arc-consistent$ relative to x_j iff for each $a_i \in D_i$ there exist a value $a_j \in D_j$ such that no constraints are broken.
CSP problem is $arc-consistent$ iff for all $i, j \in \{1, .., n\}$ variables x_i and x_j are arc-consistent.

The algorithm which forces arc consistency is filtering domains of variables so that

Example 1. *We define a CSP* (X, D, C) *where:*
$X = (x_1, x_2)$
$D_{x_1} = \{1, 2, 3\}$
$D_{x_2} = \{1, 2, 3\}$
$C = \{(x_1, x_2) | x_1 < x_2\}$

Arc-consistency enforcing algorithms produce a new equivalent CSP problem:

$X = (x_1, x_2)$
$D_{x_1} = \{1, 2\}$
$D_{x_2} = \{2, 3\}$

We see that e.g. for variable x_1 *and its value* 3*, there is no possible value in the domain of* x_2*, so that* $3 < x_2$*. Therefore algorithm enforcing this level of consistency will deduce that value* 3 *is filtered out from domain of* x_1*. Similarly for the value* 1 *in the domain of* x_2*.*

We define further levels of consistency.

Definition 7. The CSP is $path - consistent$ where any subproblem of size 2 can be consistently extended by assigning one more value to a variable.

Algorithms enforcing arc-consistency filter the domains of variables. Algorithms enforcing path-consistency are adding new constraints to the pairs of variables.

Definition 8. In general, we define $i - consistency$ as a level of consistency, where any consistent subproblem of size $i - 1$ can be extended to consistent subproblem of size i.

We see that all these level of consistency are there to provide guide for search algorithm. If there is a strong enough consistency enforced, the given step of the algorithm can immediately have a consistent assignment without having to backtrack.

Definition 9. If CSP problem is i-consistent for all i, we call it *globally consistent* and the search in this case directly provides solutions without need of backtrack.

In general, the time and space complexity of algorithms enforcing complexity is exponential in i. One of the way how to solve this problem is to use so-called directional consistency, where consistency is not forced among all the variables, but just along some fixed order of variables heavily decreasing complexity (based on some properties of graph representation of a CSP problem) and allowing backtrack free search along this ordering. The generalized version of these principles is used in one of the basic algorithms used throughout computer science. It is called dynamic programming. For more information about these algorithms, please see [1] or [2].

We have briefly described these techniques as they are used to boost search algorithms and basic forms of consitency algorithms are used commonly in search techniques.

3.2 Search algorithms

No matter how much inference we do on our problem, at one moment we must turn to trial and error. The search systematically assigns variables values from their domains and checks for the consistency. In this chapter we will describe the basic complete search algorithms, in particular complete DFS algorithms and local search algorithms. Complete DFS algorithms serve as the base for incomplete DFS algorithms. Local search algorithms are incomplete but are not based on DFS principle and are mentioned for reference.

3.2.1 Backtrack

The backtracking algorithms explores the search space in a depth-first manner. It is also known as depth first search (DFS). It is generating partial consistent assignments and in each step it tries to assign a value to a next uninstantiated variable. If the assignment is not consistent, it tries another value. If there are no more values to try, it has to return back to deal with the dead end. In this case it backtracks and tries a new value with previously assigned variable. If necessary it might continue up to the root of the search space. [1]

Algorithm 1 backtrackSearch(Variables)

```
if all_variables_assigned(Variables) then
  return true
end if
var ← select_free_variable(Variables)
while value = get_next_value(var) do
  assign(Variables, var, value)
  if backtrackSearch(Variables) then
    return true
  end if
  unassign(Variables, var)
end while
```

In this manner, backtracking algorithm systematically explores whole search spaces until the solution is found or coming to a conclusion that there is no solution. We can see that complexity of basic backtrack search algorithm is $O(d^n)$ where d is the domain size for variables and n is the number of variables.

3.2.2 Combining search and inference

Research and solving of real life problems show that combination of inference and search gives good results and better performance than if used as a standalone algorithms.[1]

Currently the most efficient complete search algorithm is MAC. The algorithm combines the backtrack search and maintaining generalized arc consistency. [2]

3.2.3 Local search algorithms

The basic principle of systematic search algorithms is that we assign a value to a variable, check if the assignment is consistent, assign a value to a next variable, check for consistency, and so on until we find a solution. If it finds inconsistency it discards the last assigned value and try another one, if needed the more several last assignments are discarded. During the whole search it maintains consistency and systematically searches the space of partial consistent assignments until it reaches full consistent assignment or algorithm finishes with no full consistent assignment found meaning that problem has no solution.

In contrast to this, local search algorithm maintains all the variables assigned some values all the time, but this assignment is not necessarily consistent. Instead, in each step of algorithm we try to change the values assigned so that we decrease the number of satisfied constraints. This basic algorithm from this category is commonly known as the hill-climbing. [1]

The local search algorithms are incomplete algorithms. They do not guarantee the finding of solution. These algorithms can get stuck in the local minimum. This is the situation where changing the value of any assigned variable doesn't reduce the count of violated constraints but overall, there is a better solution. For these reason, various methods known for hill-climbing algorithms are used - such as random restart, taboo search, simulated annealing, and others. [1]

4 General incomplete DFS algorithms

We have shown the basic depth first search algorithm, which is complete. If we have a search space which is too large to explore completely we need to introduce some limits on various criteria that allows us to prune the search space. These limits can be depth, branching or number of assignments to a given variable. Based on what the limit is, we get various algorithms described in this chapter.

4.1 Depth bounded backtrack search (DBS)

Depth bounded backtrack search limits the maximum depth in a search tree, where all the alternatives are explored. Below this depth only a single alternative is tried. The idea behind this algorithm is that goal nodes are distributed evenly in the search tree. Thus we don't want to direct our search

into a single area of the tree, but instead to spread the focus on the whole tree.

Algorithm 2 depthBoundedSearch(variables, depth)

```
if all_variables_assigned(variables) then
  return true
end if
var ← select_free_variable(variables)
tried ← false
while value = get_next_value(var) and (!tried or (depth > 0)) do
  assign(variables, var, value)
  if depthBoundedSearch(variables, depth − 1) then
    return true
  end if
  unassign(variables, var)
end while
```

We will explain function used throughout this algorithm. We need to check if all variables has been instantiated in each step of the algorithm. This happens in $all\_variables\_assigned(variables)$ which returns boolean value identifying the fact. In this case we have found a solution to a problem. Another function, $select\_free\_variable(variables)$ selects the first unassigned variables available. The order in which the variables are chosen can be determined by a heuristic. The function $assign(variables, var, value)$ assign the *value* to a variable *var*. Also, this step may contain any inference algorithm that helps to guide the search. If we introduce the inference algorithm, it prunes the possible search state space further either by filter domains by using arc consistency algorithms or by introducing new constraints in higher level of consistency.

It may happen that during the run of consistency algorithm, all possible values are filtered from domains of a variable to be assigned. In this case, algorithm backtracks similarly as in the case of exhaustively checking all possible values from a domain of a variable to be assigned. This is called *shallowbacktracking*.

If we execute the algorithm successively with increasing depth limit, we eventually obtain full DFS algorithm when we reach the maximal depth of the tree. Also, this algorithm may be combined with other incomplete search algorithms, where tree is exhaustively searched up to a given depth and below this level, some other algorithm is used. This may have performance advantages in combination with discrepancy based algorithms described in further sections.

4.2 Credit search (CS)

Credit search is very similar to depth bounded backtrack search, the amount of branching is limited. However, we have more control over on this limit. The algorithm starts with the so-called credit which is the natural number that tells the algorithm how many branching it is allowed to do altogether before it reduces to a simple direct path by choosing the first value for each variable. In each step, this credit is evenly distributed among the child nodes. Each node receives (n div k) credit where n is the credit in a given node and k is the number of branches. The rest of the credit (n mod k) is evenly distributed in a way that each branch from left receives one more credit. If there is only credit value of 1, the search continues to the bottom of the tree only with a single alternative tried. [3]

Algorithm 3 creditSearch(Variables)

```
if all_variables_assigned(Variables) then
  return true
end if
var ← select_free_variable(Variables)
baseCredit ← |D'_i|/|D_i|
restCredit ← |D'_i|%|D_i|
valueCounter ← 0
while value = get_next_value(var) do
  assign(Variables, var, value)
  valueCounter ← valueCounter + 1
  if valueCounter > 0 then
    valueCredit ← baseCredit
  else
    valueCredit ← baseCredit + 1
  end if
  if valueCredit > 0 then
    if creditSearch(Variables, valueCredit) then
      return true
    end if
  end if
  unassign(Variables, var)
end while
```

Restarting of the algorithm with the increasing credit also leads to a complete search. This algorithm has the time complexity of $O(c)$ where c is credit value for algorithm run. This gives us a very fine control about how much time is spent on processing during search.

4.3 Iterative broadening (IB)

Iterative broadening is based on a premise that decision on each level of the search tree has equal weight and the probability of error is evenly distributed. Depth bounded search and credit search preferred variables selected earlier as backtracking on the first nodes is the last option where all others fail. It does so by introducing cutoff of number of branching on each level. It establishes a breadth limit, which tells how many alternatives can be tried at each node.

Algorithm 4 iterativeBroadeningSearch(Variables)

```
if all_variables_assigned(Variables) then
  return true
end if
var ← select_free_variable(Variables)
breadthCounter ← 0
while value = get_next_value(var) and breadthCounter < breadthLimit
do
  assign(Variables, var, value)
  breadthCounter ← breadthCounter + 1
  if iterativeBroadeningSearch(Variables, breadthLimit) then
    return true
  end if
  unassign(Variables, var)
end while
```

With each restart of algorithms we increase this breadth limit. Eventually we end up with a complete search algorithm. Notice that this algorithm is exponential, each restart takes up to $O(n^b)$ where b is factor of branching. [4]

4.4 Limited assignment number search (LAN)

Limited Assignment Number search builds up on the idea of Iterative Broadening. It sticks to the idea that each variable has equal chance of mistake, but instead of limiting number of branching for a given node, we cutoff on a limited assignment number of a given variable through the search. Once the variable has hit its cutoff limits of assignments, only single alternative is tried.

In this algorithm, function *filter_expired* filters out all the expired variables from processing, i.e. variables where *counter* has reached the $LANLimit$.

Algorithm 5 limitedAssignmentNumberSearch(Variables, LANLimit)

```
FreeVariables ← filter_expired(Variables)
if all_variables_assigned(FreeVariables) then
  return true
end if
var ← select_free_variable(FreeVariables)
while value = get_next_value(var) and counter(var) < LANLimit do
  assign(Variables, var, value)
  counter(var) ← counter(var) + 1
  if limitedAssignmentNumberSearch(Variables, LANLimit) then
    return true
  end if
  unassign(Variables, var)
end while
```

Where algorithms mentioned before don't work with variables and thus they may be used in other areas of artificial inteiligence which utilize incomplete search algorithms, Limited Assignment Number search is specialized in problems, where branching is represented as an assigning different values to a specific variable, in particular CSP problems.

5 Discrepancy based search algorithms

So far we have discussed general incomplete search algorithms. Many times when we solve common types of problems we have a heuristics that can help us to get to the goal state more quickly. Heuristics in general are functions that can be calculated quite easily in comparison to search algorithm and provide us with an advice which branch should we choose in branching step. In order for heuristic to be effective, the probability, that a given choice leads to a solution must be higher than 0.5.

In the ideal world, heuristic would lead us directly to the goal state every time. In this case the heuristic itself would be a solution to a problem and we wouldn't to perform any search algorithm. In the real life, heuristics do mistakes. A lot of algorithms deal with situations where heuristics do mistakes.

Most of them are based on one or both of two simple principles. The first principle is that heuristic in general tend to do few mistakes. That leads us to a standpoint where we want to discover the states of the search space where we can get during the search with just few deviation from heuristic decision and if solution is not found there, gradually allow the number of

deviations from the heuristic. We call these decision where we deviate from heuristics decision a discrepancy. The second principle is based on that the more information we have, the more accurate heuristic tends to be. Therefore we assume that heuristic makes more mistakes early in the search and much less mistakes in the lower parts of the tree.

Definition: discrepancy is a choice made by search algorithm, where it choose the alternative that is against heuristic.

In this chapter we will introduce incomplete search algorithms where the limiting cutoff factor is based on discrepancies.

5.1 Limited discrepancy search (LDS)

The limited discrepancy search introduces a limit to discrepancies along the road. We specify a limit k. In a given search node, we follow this alternative only if it is recommended by a heuristic or the limit of the discrepancies so far has not been excluded. During the restarts we are increasing the limit of discrepancies until the solution is found. This way we take advantage of the idea, that there is higher probability that there are just few discrepancies along the search path.

Algorithm 6 limitedDiscrepancySearch(variables, discrepancyLimit)

```
if all_variables_assigned(variables) then
  return true
end if
var ← select_free_variable(variables)
first_value ← select_first_value(var)
if first_value ≠ NIL then
  assign(variables, var, value)
  if limitedDiscrepancySearch(variables, discrepancyLimit) then
    return true
  end if
  unassign(variables, var)
end if
while value = get_next_value(var) and DiscrepancyLimit > 0 do
  assign(variables, var, value)
  if limitedDiscrepancySearch(variables, discrepancyLimit − 1) then
    return true
  end if
  unassign(variables, var)
end while
```

With subsequent restarts of this algorithm we achieve complete algorithm [5]

5.2 Improved Limited Discrepancy Search (ILDS)

Limited Discrepancy Search has a disadvantage that with each restart it revisits all the nodes it has explored before. In [?] there was proposed in improvement to this algorithm, where instead of limiting the number of discrepancies by some number in each restart of algorithm, we specify exactly the number of discrepancies which must be on the path in search state generated by search algorithm.

Algorithm 7 improvedLimitedDiscrepancySearch(variables, discrepancyLimit)

```
if all_variables_assigned(variables) then
  return true
end if
freeVarCount ← free_variables_count(variables)
var ← select_free_variable(variables)
first_value ← select_first_value(var)
if dicrepancyLimit < freeVarCount then
  assign(variables, var, value)
  if improvedLimitedDiscrepancySearch(variables, discrepancyLimit)
  then
    return true
  end if
  unassign(variables, var)
end if
while value = get_next_value(var) and DiscrepancyLimit > 0 do
  assign(variables, var, value)
  if improvedLimitedDiscrepancySearch(variables, discrepancyLimit−
  1) then
    return true
  end if
  unassign(variables, var)
end while
```

In this algorithm, function *free_variables_count* returns the number of still unassigned variables. With subsequent restarts of this algorithm we achieve complete algorithm.

5.3 Depth-bounded Discrepancy search (DDS)

Depth-bounded Discrepancy Search sets the parameter - depth limit d, up to which discrepancies are allowed. After this limit is reached the algorithm must follow the heuristic. This is discrepancy-based analogy to DBS algorithm.

The idea behind is algorithm is that heuristics tend to do more errors in the beginning of the search, where they have just few information. Thus the discrepancies should be allowed in these intial stages of search.

Algorithm 8 depthBoundedDiscrepancySearch(Variables)

```
if all_variables_assigned(variables) then
  return true
end if
var ← select_free_variable(variables)
tried ← false
while value = get_next_value(var) and (!tried or (depth > 0)) do
  assign(variables, var, value)
  if depthBoundedSearch(variables, depth − 1) then
    return true
  end if
  unassign(variables, var)
end while
```

5.4 Other Discrepancy-based Search algorithms

There are several more discrepancy-based search algorithms. Interleaved depth-first search (IDFS) is similar do DDS. It tries to keep discrepancies up in the tree. However it tries to search paralels branches and in the moment it should backtrack, it tries to continue with finding solution in a parallel branch of the algorithm. See [6] for more details

Discrepancy-bounded depth first search algorithm tries to preserve discrepancy order of branches explored while try to limit the number of revisited nodes as much as possible. It parametrized by the value k. In each iteration, it explores branches between $(i-1)k$ and ik discrepancies.

5.5 Discrepancy-based sliced neighborhood search (SNS)

In [7] Parsini, Lombardi and Milano noticed that LDS and ILDS algorithms as they search exhaustively the neighborhood of a state in search space to

where heuristic points to. First, the direct solution is explored, then all states in search space with one discrepancy, then search space with two discrepancies and so on. They observed, that for some classes of problems, this space may grow very fast and provides algorithm performance drawback.

The algorithm is optimized for iterative search in optimization problems, where incumbent solution is used as a heuristic in next iteration of the algorithm. They proposed that in each step we fix some amount of variables, and perform the search only on the rest of variables, effectively searching just a slices of the neighborhood and not scanning it exhaustively. They provided experimental results showing the algorithm outperforms classic ILDS algorithm for assymetric travelling salesperson with time window.

6 Evaluation and future work

The field of constraint satisfaction problems is dynamically evolving discipline. In the recent years most of research is concentrated in inference algorithms introducing methods such as contraint weighting and symmetry breaking allowing to increase performance on the lot of tasks commonly solved in operational analysis and constraint satisfaction in general.

Combining these new methods provide a great opportunity of research trying to combine them with both complete and incomplete search algorithms.

References

[1] Dechter, R. (2003): Contraint Processing, Morgan Kaufmann

[2] Lecoutre, Ch. (2009): Constraint Networks, Wiley-ISTE

[3] Barták, R. (2004): Incomplete depth-first search techniques: a short survey, CDPC 2004

[4] Harvey, W.D., Ginsberg M.L. (1990): Iterative Broadening. In Proceedings of National Conference on Artificial Intelligence (AAAI-90). AAAI Press, pp. 216-220

[5] Harvey, W.D., Ginsberg M.L. (1995): Limited discrepancy search. In Proceedings of the 14th International Joint Conference on Artificial Intelligence, pp. 607-615

[6] Meseguer, P. (1997): Interleaved Depth-First Search. In Proceedings of 15th International Joint Conference on Artificial Intelligence, pp. 1382-1387

[7] Parsini F.,Lombardi M.,Milano M. (2010): Discrepancy-Based Sliced Neighborhood Search