International Journal on Artificial Intelligence Tools

Building Search Methods with Self-Confidence in a Constraint Programming Library

Nikolaos Pothitos and Panagiotis Stamatopoulos

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
Panepistimiopolis, 157 84 Athens, Greece
{pothitos,takis}@di.uoa.gr

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

In the late 1990s, *Constrained Programming* (CP) promised to separate the declaration of a problem from the process to solve it. This work attempts to serve this direction, by implementing and presenting a modular way to define *search methods* that seek solutions to arbitrary *Constraint Satisfaction Problems* (CSPs). The user just declares their CSP, and it can be solved using a portfolio of search methods already in place. Apart from the pluggable search methods framework for any CSP, we also introduce pluggable *heuristics* for our search methods. We found an efficient stochastic heuristics' paradigm that smoothly combines randomness with normal heuristics. We consider a factor of *disobedience* to normal heuristics, and we fine-tune it each time, according to our estimation of normal heuristics' reliability (confidence). We prove mathematically that while the disobedience factor decreases, the stochastic heuristics approximate deterministic normal heuristics. Our algebraic evidence is supported by empirical evaluations on real life problems: A new search method, namely PoPS, that exploits this heuristics' paradigm, can outperform regular well-known constructive search methods.

Keywords: Randomness; stochastic methods; discrepancy; constructive search; CSP.

1. Introduction

Artificial Intelligence (AI) methodologies aim to tackle difficult computational and real life problems, such as scheduling,[1,2] radio frequency assignment,[3] other NP-hard problems, and also problems stemming from various disciplines, e.g. Bioinformatics.[4]

All these Constraint Satisfaction Problems or Constrained Optimization Problems have been declared in our Constraint Programming Naxos Solver.[5] In such solvers, the solution phase is completely independent of the CSP declaration phase, as this serves the original promise of Constraint Programming: *The user states the problem, the computer solves it.*[6]

In this work, we go one step further and allow the user/programmer to state their own *search methods* that can apply to any CSP. We found a framework where the user can compose their search methods out of conjunctive and disjunctive goals.

Naxos Solver is a C++ constraint programming library. We implemented on top of it our search methods' framework, but the framework can be adopted by other solvers too. Our goal is not to compare Naxos against other solvers, but to use it as an open source ground/environment for this work's contributions.

Of course, Constraint Programming is not the only paradigm that solves Artificial Intelligence problems like the aforementioned. For example, course scheduling can be addressed via plain simulated annealing.[7] In comparison to Constraint Programming, this is not a complete method, i.e. it doesn't explore all the possible solutions, and therefore it doesn't always find the best solution, if any.

Another way to solve the course scheduling problem is by employing the simplex method in the context of linear programming.[8] In comparison to Constraint Programming, linear programming is less expressive, as it requires the formulation of every problem in a strict mathematical model.[9]

Naive search methods explore the whole candidate solutions spectrum, in order to find a real solution. The issue here is that the candidate solution range is exponential in the problem instance parameters, and, unavoidably, an iteration through every candidate solution becomes infeasible as the problem scales.

Heuristics' role in this situation is to change the order of the candidate solutions, so as to favor the "promising" ones. In other words, heuristics make an estimation of the possibility of an incomplete or candidate solution being a real solution, and label it with a priority. A high priority means that the candidate solution should be examined soon.

This reordering cannot make the search space tractable—this is most probably impossible[10]—but it is able to dramatically decrease the time needed to guide a search method toward a real solution. In this direction, we study heuristic properties, such as reliability/confidence, and we propose a generic framework in order to exploit them by incorporating a randomness factor into them. This work is an extended and revised version of a preliminary conference paper.[11]

In Section 2 we introduce just the necessary formal definitions for CSPs. In Section 3 we found the framework that can be used to define search methods; the algorithmic details are isolated in Appendix A. In Section 4 we explore a search tree by consulting heuristics. In Section 5 we link heuristics to probabilities and we bridge total determinism to total randomness while consulting them. Section 6 illustrates a new search method, namely PoPS, that exploits the values of the heuristic evaluations. Finally, in Section 7 we conduct experiments to support the previous theoretical sections.

In summary, the contributions of the paper are

- the foundation of a modular *search methods framework* for Constraint Programming,
- the introduction of a *confidence* factor into regular heuristics and their gradual randomization when we aren't confident about them, in order to make them more flexible, and finally

- the implementation of an efficient *new search method* PoPS that exploits heuristics as values/evaluations—as they are—and not simply as ranks of possible choices.

2. Preliminaries

We focus on *constraint satisfaction problems* (CSPs)[12] that can be solved via a plethora of available *constraint programming* (CP) solvers.[13,14]

2.1. *Constraint satisfaction problems*

Every single CSP can be stated using commonplace formalizations.[15] It is a triplet of

(i) *constrained variables* $X_1, \ldots, X_n$,
(ii) their corresponding *domains* $D_{X_1}, \ldots, D_{X_n}$, which are normally finite sets of integers, and
(iii) the *constraints* between variables; a constraint contains the tuples of all the valid assignments for a specific pair/set of variables. To put it differently, a constraint is a relation between the variables, such as $X_1 < X_2$.

In the attempt to find a solution to a CSP, we have to make assignments.

Definition 2.1. We say that a variable X is *assigned* a value $v \in D_X$, if its domain is made singleton, i.e. $D_X \leftsquigarrow \{v\}$.

A *solution* is an assignment that involves all variables and also satisfies all the constraints. A *search method* leads a CSP after consecutive assignments into a solution.

2.2. *Map-coloring problem*

There exists a huge list of interesting CSPs.[16] For example, *map-coloring* is a CSP for assigning colors to each prefecture in a given map, so as no neighboring prefectures have the same color. Figure 1 illustrates a map of the Greek region "Thessaly," containing four prefectures; the colors in the figure form an indicative solution.

Problem 1. Typically, *"Thessaly-coloring"* is a CSP with:

(i) Four constrained variables: X_1, X_2, X_3, X_4. Each one of them represents a prefecture color.
(ii) The corresponding domains are $D_{X_1} = D_{X_3} = \{1, 2\}$ and $D_{X_2} = D_{X_4} = \{1, 3\}$. Numbers 1 2 3 represent respectively red, green, blue.[a]
(iii) The constraints are $X_1 \neq X_2$, $X_1 \neq X_3$, $X_2 \neq X_3$, and $X_2 \neq X_4$.

[a]We could initially set all the domains equal to $\{1, 2, 3\}$. We used smaller initial domains just to simplify the problem.

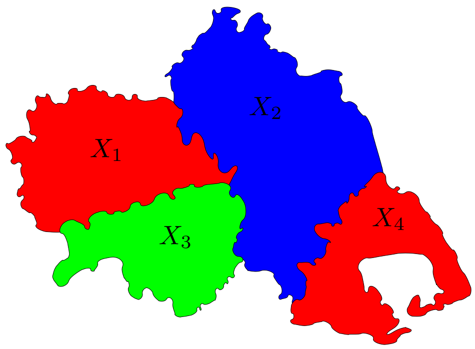

Fig. 1. The four Thessaly prefectures

The solution in Fig. 1 is represented by the assignment

$$\{X_1 \leftarrow 1,\, X_2 \leftarrow 3,\, X_3 \leftarrow 2,\, X_4 \leftarrow 1\}\,. \tag{1}$$

2.3. *Constrained optimization*

A variation of Constraint Satisfaction Problems (CSPs) are the so-called *Constrained Optimization Problems* (COPs). A COP consists of variables, domains, and constraints, just like any CSP, but there are two differences.

- A COP also requires an *objective function* which maps any assignment/solution to a number, which is called the *cost* of the solution.
- The target while solving a COP isn't just to find a solution, but to find a *best* solution, i.e. a solution with a minimum cost.[b]

COPs can be solved like CSPs, using a *branch and bound* strategy: When a solution is found, its cost is recorded, and a new constraint is added to guarantee that the next solution cost will have a smaller cost than the recorded one. This is repeated until all candidate solutions have been examined.

In relation to CSP solving, the only additional requirement of the above COP solving procedure is adding dynamically a new constraint while searching. This makes it compatible with plain CSP search methods, so this work covers both CSPs and COPs as a whole.

On the other hand, this work does not cover *convex optimization*, a variant introduced in Mathematics which paved the way for advances in Computer Science.[17] Besides, convex optimization applies to *continuous* domains, e.g. $[0.5, 3.1]$, while in Constraint Programming we focus on *discrete* domains of constrained variables, e.g. $\{1, 2, 3\}$.

[b]There is also a COP variation which requires to find the solution with the *maximum* cost. However, for simplicity reasons, we will not focus on it, as it can be easily transformed into a COP with a minimization objective.

3. A Goal-driven Search Methods' Framework

Apart from a way to state CSPs, a user/programmer needs an elegant way to state search methods that solve them. The CSPs should be "search-methods-agnostic," while the search methods should be "CSP-agnostic" in order to keep the independence between Constraint Programming stages.

In related works, a lot of search methods have been implemented "out of the box" in modern solvers.[18] This means, at least to our knowledge, that the implemented search methods are coupled with the existing solvers. Nevertheless, our contribution is to introduce an extensible framework, so that the user can easily define their own "custom" search methods.

3.1. *Search methods are made up of goals*

Every constructive search method is built up of goals. Each goal executes an operation, e.g. an assignment of a value to a constrained variable. Otherwise, one goal returns another goal to be executed. The goal returned can be a *meta*-goal, that is a goal that refers to another two goals. There are two meta-goal kinds:

(i) The $\mathsf{AND}(g_1, g_2)$, which implies that the two sub-goals g_1 and g_2 must be executed and succeed both.
(ii) The $\mathsf{OR}(g_1, g_2)$, which executes g_1. If g_1 does not succeed, i.e. if it does not lead to a solution, then g_2 is executed.

This goal-driven framework is able to describe most of the common search methods.

3.2. *The Depth-First Search example*

An elementary search method that can be straightforwardly described via goals is *depth-first search* (DFS). This method iterates through the variables of a CSP. For each variable X selected, it selects a value v from its domain and makes the assignment $X \leftarrow v$. It subsequently proceeds to the next unassigned variable and makes another assignment, etc.

If every variable is assigned a value and no constraint is violated, the assignments set comprises a *solution*. In any case, if there is a constraint violated, the last assignment to a variable is undone and we try to assign another value from its domain. If all the alternative values are exhausted, we *backtrack* to the previous variable selected and we undo its assignment and so forth.

3.3. *Defining DFS using goals*

The ultimate goal in DFS and in every constructive search method is to `Label` every variable with a value. Each `Label`'s call aims to `Instantiate` a variable.

- $\texttt{Label}(\varnothing) := \mathsf{success}$.
- $\texttt{Label}(\mathscr{X}) := \mathsf{AND}(\texttt{Instantiate}(X), \texttt{Label}(\mathscr{X} - \{X\}))$, with $X \in \mathscr{X}$,

where $\mathscr{X}$ is the set of all the variables. While `Label` iterates recursively through the CSP variables, an `Instantiate` call attempts to assign a selected value v to the variable X. If the assignment fails to produce a solution, the value v is deleted and another instantiation is attempted, until all the alternatives in D_X are exhausted.

- `Instantiate`(X) := failure, with $D_X = \varnothing$.
- `Instantiate`(X) := OR$(X \leftarrow v$, AND$(D_X \leftarrow D_X - \{v\}$, `Instantiate`$(X)))$, with $v \in D_X$.

The interdependencies between the above DFS goals are graphically displayed in Fig. 2.

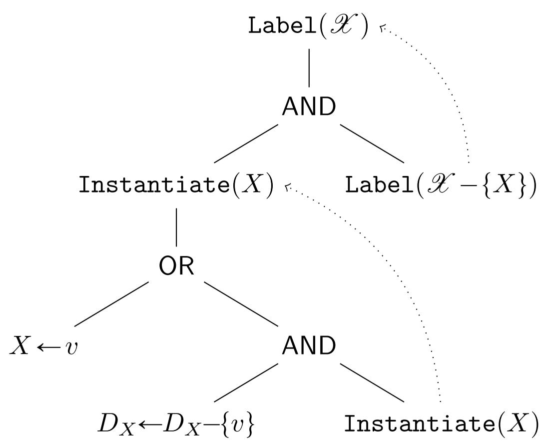

Fig. 2. The combination of the goals that compose DFS

3.4. *Defining Iterative Broadening using goals*

Figure 3 displays the corresponding goals' graph for the *Iterative Broadening* search method.[19] The goals' structure is similar to DFS. However, one basic difference is that there is one more level, namely `Broadening`, above the ordinary DFS goals.

- `Broadening`$(Breadth)$:= failure, if $Breadth > d$,
- `Broadening`$(Breadth)$:= OR(`Label2`$(\mathscr{X})$, `Broadening`$(Breadth + 1)$), otherwise.

For each Iterative Broadening iteration, the *Breadth* parameter defines the maximum number of values that a constrained variable can be successively assigned. This value is initially 1. The *Breadth* value cannot exceed d, which in this context is the maximum cardinality (size) of the domains of all constrained variables. If *Breadth* exceeds d, `Broadening` fails.

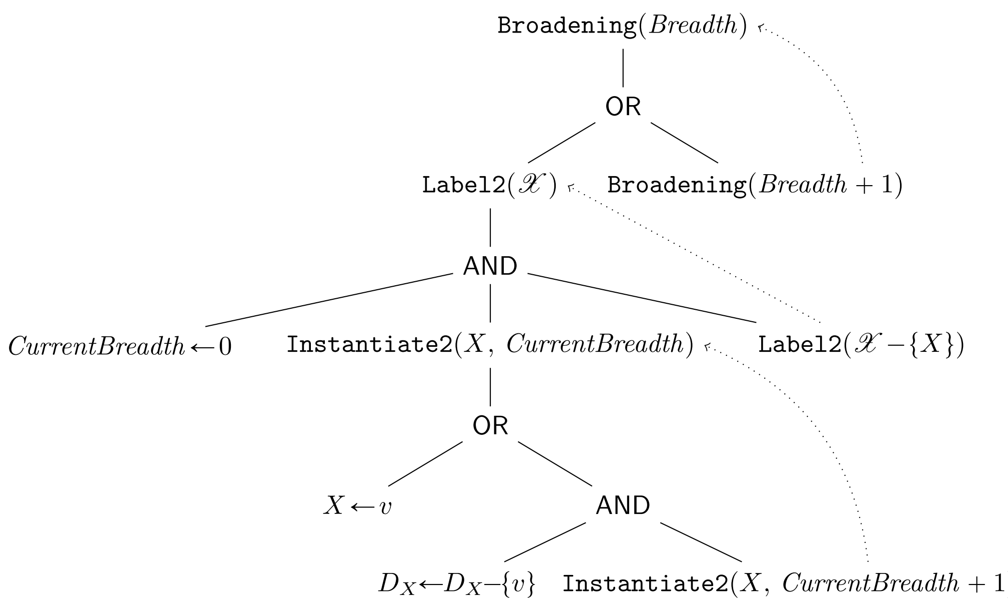

Fig. 3. The goals composing Iterative Broadening

Therefore, a second basic difference in comparison with DFS comes into play. The `Instantiate` goal is now named `Instantiate2`, and it takes one more argument.

- `Instantiate2`$(X,\ \mathit{CurrentBreadth}) :=$ failure, if $D_X = \emptyset$,
- `Instantiate2`$(X,\ \mathit{CurrentBreadth}) :=$ failure, if $\mathit{CurrentBreadth} > \mathit{Breadth}$,
- `Instantiate2`$(X,\ \mathit{CurrentBreadth}) :=$ OR$(X \leftarrow v,$
  AND$(D_X \leftarrow D_X - \{v\},$ `Instantiate2`$(X,\ \mathit{CurrentBreadth} + 1)))$, otherwise.

This implements Iterative Broadening's semantics: The number of consecutive instantiations to the same variable cannot exceed *Breadth*.

4. Search Tree Exploration

A *search tree* is a descriptive way to depict every possible assignment in a CSP, such as map-coloring. Figure 4 displays the search tree for the Thessaly-coloring problem. The struck-out nodes have been pruned as no-goods.

Each path from the root (i.e. the uppermost node) represents an *assignment*. If the path from the root ends up into a leaf (lowest node), we have a *complete* assignment. E.g., the dotted path in Fig. 4 is an alternative form of the solution assignment in (1).

4.1. *The goals are the search tree nodes*

There is a direct relationship between a search tree and the goals hierarchy.

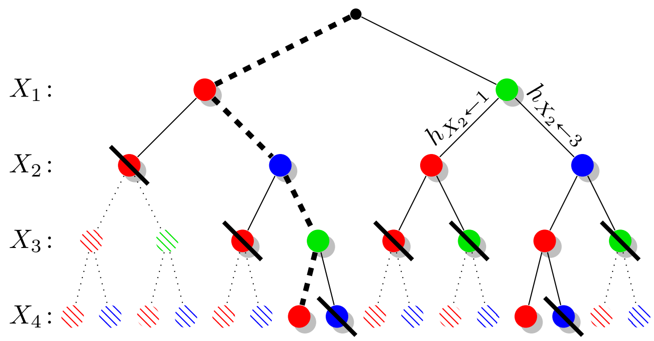

Fig. 4. The search tree for Thessaly-coloring

- When the first goal, e.g. `Label`$(\mathscr{X})$ in DFS, is called, the search tree *root* is created.
- When an OR goal occurs, the current node is extended into two branches that represent the two alternative choices.

In this work, we consider only sequential search methods. Nevertheless, the presented search methods framework naturally supports distributed search methods too. We can simply distribute a search tree when we encounter an OR goal. The left and right branches of selected OR nodes can be explored concurrently to reduce the total tree exploration time. There are many different approaches regarding which OR nodes should be selected in order to split their two sub-trees.[20,21]

4.2. *Heuristic estimation as a real number*

A heuristic function maps every possible choice in the search tree to a number that corresponds to the estimation that it will eventually guide us toward a solution.

Definition 4.1. For a specific search tree node, let Choices be the set with the alternative assignments that one may follow. The *heuristic function* h_i maps each alternative assignment $i \in$ Choices to a positive number, i.e. $h : \mathsf{Choices} \to \mathbb{R}^+$.

Example 4.1. In Fig. 4 uppermost right node, there are two alternative assignments in Choices $= \{X_2 \leftarrow 1, X_2 \leftarrow 3\}$. One heuristic function may provide the estimations, e.g. $h_{X_2 \leftarrow 1} = 0.7$ and $h_{X_2 \leftarrow 3} = 2.8$; that is, the assignment $X_2 \leftarrow 3$ is more promising.

The above example is almost ideal, as the heuristic function h favors the assignment $X_2 \leftarrow 3$ over $X_2 \leftarrow 1$. Besides, the latter leads to a dead end, as its two descendants are struck-out in Fig. 4, because they violate the constraints.

Unfortunately, this is not always the case, i.e. the heuristic value for an assignment that leads to a dead end (say $X_2 \leftarrow 1$ in Fig. 4) may be overestimated or, even worse, may be greater than the heuristic estimation for an assignment that really leads to a solution (e.g. $X_2 \leftarrow 3$).

A heuristic value h_i is actually a *prediction* whether a specific assignment will ultimately guide us to a solution or not. Being a prediction, it implies an inherent *reliability/confidence* level.

In the above definition, we excluded negative values as the heuristic function's output. A negative heuristic evaluation could probably mean "don't make this choice at all." But heuristics are normally used to favor one choice over another and not to prune a choice. In any case, if we had a function h with $\min h < 0$, we could transform it into $h' = h + |\min h|$ to make it comply with the above definition.

4.3. *Heuristics exploitation in related work*

In *constructive search*, one can build a solution either with a deterministic/ systematic search method or by making one-by-one random assignments. Do these methods exploit heuristics and how?

4.3.1. *Deterministic search methods*

To our knowledge, existing search methods such as *limited discrepancy search* (LDS) use heuristics only to *order* the possible assignments and do not exploit the *difference* of the one heuristic estimation to another, but only their *rank*.[22] For example, the *iterative broadening* method explores only a limited children's number for each search tree node.[19] Of course, it chooses to visit only the children with the highest ranks. *Credit search*[23] and *limited assignment number* (LAN)[24] are other deterministic methods that also take into account the rank of the heuristic estimations and not the heuristic values themselves.

Last but not least, there are also methods that make the assumption that *the heuristic function is more reliable as the search tree node depth increases.* E.g., *depth-bounded discrepancy search* (DDS) allows to override a heuristic estimation, only when we have not yet reached a specific search tree depth.[25] Finally, there are some methodologies that take into account two or more heuristic functions and *learn* as the search proceeds which heuristic is the best to use.[26]

4.3.2. *Random search methods*

On the other hand, stochastic search methods completely ignore heuristics, as they choose to make an assignment at random.[27] For example, *depth first search with restarts* traverses the search tree making random choices, and when a specific time limit is reached, it restarts from the beginning.

4.3.3. *Local search methods*

The aforementioned search methods belong to *constructive search*, as they build a solution from scratch, step by step, by assigning a value to a variable each time.

On the other hand, there are non-systematic indirect search methods, also known as *local search* methods, which assemble a *candidate* solution, and then try to fix it by eliminating conflicting sets of variables and constraints. Local search iteratively tries to *repair* the candidate solution, in order to satisfy the constraints a posteriori.[28]

Stochastic local search makes a random repair action in each step. There are many other local search variants.

Hill climbing. A well-known variant is *hill climbing*, also known as *iterative improvement*. In each step, it changes only one variable assignment (1-exchange). Normally, we make the change which will reduce the violated constraints number as much as possible.[29]

Simulated annealing. The above practice is prone to be trapped into *local minima*. This means that we can end up in a candidate solution that cannot be improved by modifying only one assignment any more. In this case, we have to escape the current local minimum by making a random step.

Simulated annealing methodology permits random steps to skip local minima while a parameter called *temperature* is high; as time passes by and temperature drops, the method becomes less tolerant in random steps, especially if their evaluation is poor.[30] In this work we attempt to bring this (local search) approach in constructive search methods.

4.3.4. *Heuristics and probabilities*

Constructive search methods either use heuristics as $\mathsf{Choices}$ ranks, or completely ignore them. In 1996, Bresina transformed the heuristic ranks into *probabilities* via the so-called *heuristic-biased stochastic sampling* (HBSS).[31] He provided a set of various decreasing functions $\mathsf{bias}(r)$, e.g. $\frac{1}{r}$ or e^{-r} etc., that take a specific integer choice rank $r \in \{1, 2, \ldots\}$ and return a number that corresponds to the probability of the choice to be selected. Cicirello and Smith improved HBSS by introducing the *value-biased stochastic sampling* (VBSS). The bias function now takes as argument the heuristic *value* itself.[32]

On the other hand, Gomes et al. exploit the so-called *heuristic-equivalence* to equate the choices with the highest heuristic values.[33] In this way, we can exclude the choices with the lower heuristic values and select at random amongst the choices with the most prevailing values.

5. New Probabilistic Heuristics

Our contribution lies in the mathematical foundation of a framework that covers both deterministic and random heuristics in constructive search. In contrast to existing methodologies, we leverage on the *smooth* transition from the total randomness to determinism.

5.1. *Heuristics probabilistic foundations*

Probabilities are a more precise way to depict heuristics than orderings, because heuristics are actually *estimations* whether a choice will guide us to a solution; they are not a strict quality rank.

Definition 5.1. A function $P : \mathsf{Choices} \to [0\,,1]$, namely a *heuristic distribution function*, maps each available choice to a corresponding probability, i.e. $P(i)$.

As in Definition 4.1 and the Example 4.1 that follows it, Choices may include all the possible/candidate assignments to a constrained variable.

Property 1. It should hold that $\sum_i P(i) = 1$, as P denotes a probability for each $i \in \mathsf{Choices}$.

Regarding random search methods (Section 4.3.2), the probability is distributed uniformly along the Choices. Conclusively,

Property 2. The heuristic distribution for a random method is always $P(i) = \frac{1}{|\mathsf{Choices}|}, \forall\, i$.

Example 5.1. Say that $\mathsf{Choices} = \{v_1, v_2, \ldots, v_5\}$. Every v_i denotes a possible assignment. Furthermore, in a specific search tree node we can make five different assignments, and their corresponding heuristic estimations h_i are 1, 5, 2, 4, 3 respectively, as in Fig. 5.

Figure 6 depicts the corresponding heuristic distribution function for a random method, that is $P(i) = \frac{1}{5}, \forall\, i$.

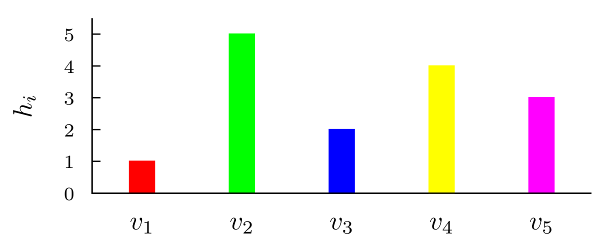

Fig. 5. Heuristic estimations h_i for each value v_i

On the other extreme, deterministic search methods (Section 4.3.1) always select the choice v_i that corresponds to the h_i with the highest rank.

Property 3. Formally, in deterministic search methods, if $i = \arg\max_j h_j$, then $P(i) = 1$, otherwise $P(i) = 0$.

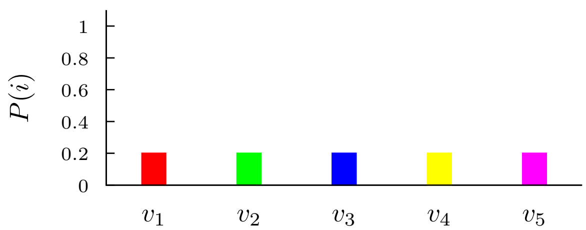

Fig. 6. The probability is spread uniformly

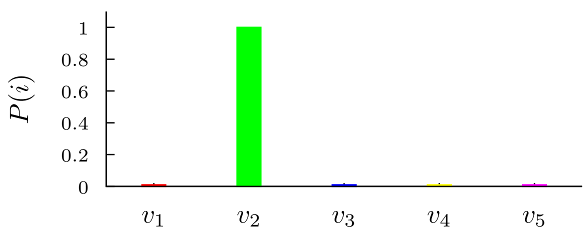

Fig. 7. Systematic search favors the highest h_i

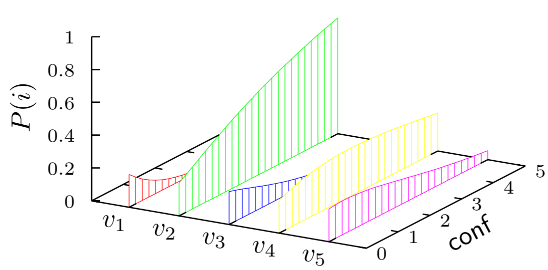

Fig. 8. As conf rises, the effect to $P(i)$ is greater

Example 5.2. The greatest heuristic value in Example 5.1 is $h_2 = 5$. Hence, a deterministic search method would select v_2 with a certain probability $P(2) = 1$. Consequently, the rest of the probabilities are zero, as in Fig. 7.

If there is more than one maximum heuristic value, deterministic methods arbitrarily concern only one of them as maximum. To simplify the following equations, we will assume that there is only one maximum. Without loss of generality, we also assume that heuristic values are non-zero.

5.2. *Bridging the two opposites*

We extend our previous formulation of the heuristic distribution function (Definition 5.1) in order to compromise random and deterministic methods. We introduce a parameter $\mathsf{conf} \in \mathbb{R}^+$, that signifies how much the heuristic estimations will be taken into account; it is the heuristic's *confidence*. This confidence parameter is the basis to define the condition when a heuristic distribution function is "balanced."

Definition 5.2. A parameterized heuristic distribution function $P_{\mathsf{conf}}(i)$ is *balanced* if and only if:

1. $\forall\, i,\ \lim\limits_{\mathsf{conf}\to 0} P_{\mathsf{conf}}(i) = \frac{1}{|\mathsf{Choices}|}$, and

2a. if $i = \arg\max_j h_j,\ \lim\limits_{\mathsf{conf}\to\infty} P_{\mathsf{conf}}(i) = 1$,

2b. otherwise, $\lim\limits_{\mathsf{conf}\to\infty} P_{\mathsf{conf}}(i) = 0$.

Moreover, the function $P_{\mathsf{conf}}(i)$ must be monotonic and continuous with respect to conf and for fixed i.

Intuitively, conf is the link between random and deterministic search methods, as the above definition covers both Property 2 when $\mathsf{conf} \to 0$ and Property 3 when $\mathsf{conf} \to \infty$. In other words, conf is the position along the random-deterministic axis.

What happens for intermediate conf values? This depends on the precise parameterized heuristic distribution function instance. We define the following function that gradually scales randomness.

Lemma 5.1. *The function* $P_{\mathsf{conf}}(i) = \frac{h_i^{\mathsf{conf}}}{\sum_j h_j^{\mathsf{conf}}}$ *is balanced.*[c]

Proof. We prove Definition 5.2 three requirements.

1. $\lim\limits_{\mathsf{conf}\to 0} P_{\mathsf{conf}}(i) = \frac{h_i^0}{\sum\limits_{j\in\mathsf{Choices}} h_j^0} = \frac{1}{\sum\limits_{j\in\mathsf{Choices}} 1} = \frac{1}{|\mathsf{Choices}|}$.

2a. Let $n = |\mathsf{Choices}|$. This number is bounded as the possible assignments in a CSP are a finite set. Thus, the distribution function can be analyzed as

$$P_{\mathsf{conf}}(i) = \frac{h_i^{\mathsf{conf}}}{\sum_j h_j^{\mathsf{conf}}} = \frac{h_i^{\mathsf{conf}}}{h_1^{\mathsf{conf}} + h_2^{\mathsf{conf}} + \cdots + h_{\max}^{\mathsf{conf}} + \cdots + h_n^{\mathsf{conf}}}.$$

Let $h_{\max}$ be the maximum h_i. If we divide by $h_{\max}^{\mathsf{conf}}$ both the nominator and

[c] For $\mathsf{conf} = 1$, the function $P_1(i) = \frac{h_i}{\sum_j h_j}$ is equivalent to the *fitness proportionate selection* function—resembling a *roulette wheel*—that is used in Genetic Algorithms.[34]

denominator, we have

$$P_{\mathsf{conf}}(i) = \frac{\left(\frac{h_i}{h_{\max}}\right)^{\mathsf{conf}}}{\left(\frac{h_1}{h_{\max}}\right)^{\mathsf{conf}} + \cdots + 1 + \cdots + \left(\frac{h_n}{h_{\max}}\right)^{\mathsf{conf}}} = \frac{\left(\frac{h_i}{h_{\max}}\right)^{\mathsf{conf}}}{1 + \sum_{j \neq \max} \left(\frac{h_j}{h_{\max}}\right)^{\mathsf{conf}}} . \tag{2}$$

Here, max is an abbreviation for $\arg\max_i h_i$. Therefore, $\forall\, j \neq \max$,

$$h_j < h_{\max} \;\Rightarrow\; \frac{h_j}{h_{\max}} < 1 \;\Rightarrow\; \lim_{\mathsf{conf} \to \infty} \left(\frac{h_j}{h_{\max}}\right)^{\mathsf{conf}} = 0 . \tag{3}$$

As a result from (2) and (3),

$$\lim_{\mathsf{conf} \to \infty} P_{\mathsf{conf}}(i) = \frac{\lim_{\mathsf{conf} \to \infty} \left(\frac{h_i}{h_{\max}}\right)^{\mathsf{conf}}}{1 + \sum_{j \neq \max} \lim_{\mathsf{conf} \to \infty} \left(\frac{h_j}{h_{\max}}\right)^{\mathsf{conf}}} = \lim_{\mathsf{conf} \to \infty} \left(\frac{h_i}{h_{\max}}\right)^{\mathsf{conf}} . \tag{4}$$

A direct derivation is that for $i = \max \equiv \arg\max_j h_j$, we have $\lim_{\mathsf{conf} \to \infty} P_{\mathsf{conf}}(\max) = 1$, which is the second prerequisite for a balanced function.

2b. Finally, the last prerequisite of Definition 5.2 involves $i \neq \max \Rightarrow h_i < h_{\max} \Rightarrow \frac{h_i}{h_{\max}} < 1$, which, combined with (4), gives $\lim_{\mathsf{conf} \to \infty} P_{\mathsf{conf}}(i) = 0$, which had to be demonstrated. □

The above function (in Lemma 5.1) is balanced, and it also moves smoothly from the random extreme to the deterministic one, because it is a *continuous* function, with regard to $\mathsf{conf} \in \mathbb{R}^+$.

Hence, the overall function is a transition from the total randomness to the almost total determinism. This is illustrated in the three-dimensional Fig. 8, which for $\mathsf{conf} = 0$, is equivalent to the two-dimensional Fig. 6, and when $\mathsf{conf} \to \infty$, it is equivalent to Fig. 7.

Furthermore, our initial goal was to propose flexible heuristics which perform better than purely deterministic or purely stochastic ones. To implement and measure the transition from randomness to determinism, we just introduced a *confidence* value. However, new questions now arise. Which conf value should be used? Which is the best way to bind the proposed hybrid heuristics to search processes?

6. Piece of Pie Search

The probabilistic framework founded in the previous section, naturally complies with existing search methods; it affects only the heuristic function and not the methods themselves. But in order to fully exploit the introduced heuristics framework, we built the new constructive search method *Piece of Pie Search* (PoPS).

6.1. *The algorithm's core*

Figure 9 describes PopsSample, which is the PoPS core. It is called inside PoPS in order to solve a CSP by providing a complete and valid $\mathsf{Assignments}$ set, which is initially empty. From now on, we consider that the value $\mathsf{conf} = 100$ represents infinity.

function PopsSample($\mathsf{PieceToCover}, \mathsf{conf}$)
  arguments:
    $\mathsf{PieceToCover}$: The proportion of the heuristics' pie to be explored
    conf: A "confidence" value between 0 and 100
  local variables:
    $\mathsf{Assignments}$: set with all the assignments made until this call
    $\mathscr{X}$: set with all the constrained variables
    X: constrained variable that is going to be instantiated
    value: value that is going to be assigned
    $h_{X \leftarrow v}$: heuristic value for the assignment $X \leftarrow v$
    $D_{X_{\text{init}}}$: initial domain of X, before any assignment was made
    D_X: current domain of X
    CoveredPiece: current covered proportion of the pie

  if $\mathsf{Assignments}$ violate any constraint **then**
    return failure
  else if $\mathsf{Assignments}$ include every variable **then**
    Record $\mathsf{Assignments}$ as solution
    return success
  end if
  $X \leftarrow$ VariablesOrderHeuristic($\mathscr{X}$)
  $D_{X_{\text{init}}} \leftarrow D_X$
  $\mathit{CoveredPiece} \leftarrow 0$
  while $\mathit{CoveredPiece} \leqslant \mathsf{PieceToCover}$ **do**
    $\mathsf{value} \leftarrow$ ValuesOrderHeuristic(D_X, conf)
    $\mathit{CoveredPiece} \leftarrow \mathit{CoveredPiece} + \frac{h^{\mathsf{conf}}_{X \leftarrow \mathsf{value}}}{\sum_{v \in D_{X_{\text{init}}}} h^{\mathsf{conf}}_{X \leftarrow v}}$
    Assign value to X and add it to $\mathsf{Assignments}$
    PopsSample($\mathsf{PieceToCover}$, $\mathsf{conf} + \frac{100 - \mathsf{conf}}{|\mathscr{X}|}$)
    Undo the assignment
    $D_X \leftarrow D_X - \{\mathsf{value}\}$
  end while
  $D_X \leftarrow D_{X_{\text{init}}}$ ▻ Restores initial domain
  return failure ▻ All alternative values are exhausted
end function

Fig. 9. The recursive PopsSample called by PoPS

In each PopsSample call we get an unassigned variable returned by the function VariablesOrderHeuristic($\mathscr{X}$), where $\mathscr{X}$ is the set of all the constrained variables. Then, it stores its domain D_X, in order to restore it in a future backtrack. All the above steps are common in constructive search methods.

The crucial and novel part of this function is inside the **while** iteration where we iterate through the different values in D_X. The call ValuesOrderHeuristic(D_X, conf) returns the best value out of D_X, according to the heuristic estimation, using the heuristic function in Lemma 5.1.

Normal search methods, like Depth First Search (DFS), Limited Discrepancy Search (LDS), and other known deterministic methods explore in their steps a specific *number* of values in D_X or every value in it (cf. Section 4.3.1). In PopsSample, we explore a specific *subset* D'_X of D_X, which corresponds to a proportion of the heuristics pie. The proportion is the argument PieceToCover $\in [0, 1]$. When PieceToCover is 1, PopsSample becomes a complete search method as it explores all the D_X set values.

Example 6.1. Figure 10 demonstrates the heuristics-probabilities pie for the Example 5.1: Each $P(i)$ corresponds to a value v_i in D_X. In this case, a PopsSample$(0.5, 1)$ invocation would explore half the pie. E.g., the choices that correspond to the heuristics $P(1) + P(2) + P(3)$ or $P(2) + P(5)$ make half the pie and more.

A more detailed step by step explanation follows.

- We are inside the **while** loop of a PopsSample$(0.5, 1)$ call.
- *CoveredPiece* is initially 0; the loop stops when *CoveredPiece* exceeds 0.5.
- ValuesOrderHeuristic$(D_X, 1)$ is called.
- According to Example 5.1, this function will return a value out of $\{v_1, v_2, v_3, v_4, v_5\}$.
- Each value v_i has been evaluated with a heuristic value h_i.
- Most specifically, $h_1 = 1$, $h_2 = 5$, $h_3 = 2$, $h_4 = 4$, and $h_5 = 3$.
- The probability that v_i is selected by ValuesOrderHeuristic is $P(i)$, which is calculated using the above evaluations together with Lemma 5.1.
- Thus, the respective probabilities are $P(1) = 0.07$, $P(2) = 0.33$, $P(3) = 0.13$, $P(4) = 0.27$, and $P(5) = 0.20$.
- Again, all the above are *probabilities* $(P(i))$ of the event that a specific value (v_i) will be selected. Therefore, every value can be selected in each iteration.
- Suppose that v_5 is selected at the first iteration with $P(5) = 0.20$.
- This probability is also used to increase the current *CoveredPiece*, which becomes 0.20 too.
- X is assigned v_5.

After the assignment, PopsSample$(0.5, 1 + \frac{99}{|\mathscr{X}|})$ is called. Please note the slight increase of the conf value. This recursive call will choose another variable out of $\mathscr{X}$ and enter the **while** loop again. This loop will try to assign a value to the new variable from its domain. If the attempts fail, we continue back to the first **while** loop, which was described in the above bullets.

- The assignment of v_5 to X is undone, v_5 is removed from the domain, and another iteration begins.
- We proceed to the second iteration, as the PieceToCover (0.5) is still greater than the *CoveredPiece* (0.2).
- Let's say that v_2 is then chosen by ValuesOrderHeuristic with a $P(2) = 0.33$ probability.
- *CoveredPiece* now equals $0.20 + 0.33 = 0.53$.

Again, PopsSample$(0.5, 1 + \frac{99}{|\mathscr{X}|})$ is called. If the attempts to instantiate the next variable fail, we are back to the first **while** loop:

- The assignment of v_2 to X is undone.
- We proceed to the third iteration.
- However, *CoveredPiece* (0.53) is now greater than PieceToCover (0.5).
- More than half of the pie of the choices for X has been already explored; no other alternatives are examined.
- The rest of the values v_1, v_3, v_4 are left unused/unexplored. This makes PopsSample an *incomplete* search method, as it may override a solution (which involves for example these values) for the sake of speed.

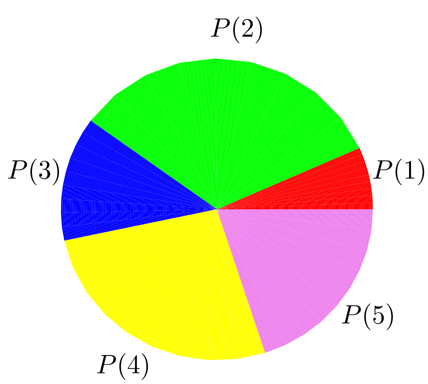

Fig. 10. The heuristics-probabilities pie chart for Example 5.1

Again, LDS is an incomplete search method too; at each search tree node, it may explore only a limited *number* of the available choices. The difference with our method is that we may explore a limited *proportion* of the heuristics pie of choices, which makes our method more "heuristics-aware." This means that the number of the explored choices by our method in a specific node may vary, depending on how the heuristics pie is distributed to the choices. On the other hand, LDS explores a fixed number of choices, independently of the heuristics pie distribution.

It's worth noting that while more variables get instantiated, the conf value gradually increases. Besides, heuristic estimations tend to be more reliable when we have less unassigned variables.

Example 6.2. We will consider the above Example 6.1 for a PopsSample$(0.5, 2)$ call, i.e. for $\mathsf{conf} = 2$.

According to Lemma 5.1, the probabilities for $\mathsf{conf} = 2$ are computed as $P(i) = \frac{h_i^2}{\sum_j h_j^2}$. For example, $P(1) = \frac{h_1^2}{\sum_j h_j^2} = \frac{1^2}{1^2+5^2+2^2+4^2+3^2} = 0.02$. The other probabilities are $P(2) = 0.45$, $P(3) = 0.07$, $P(4) = 0.29$, and $P(5) = 0.16$. Thus, the pie is redistributed as in Figure 11.

While conf-idence increases, the value v_2 which had initially the greatest heuristic evaluation h_2 is even more likely to be selected, as $P(2)$ increases too. In other words, we get closer to total determinism and closer to complete confidence in the highest heuristic evaluation: In total determinism (in systematic search) v_2 would have been always selected with a certain probability 1.

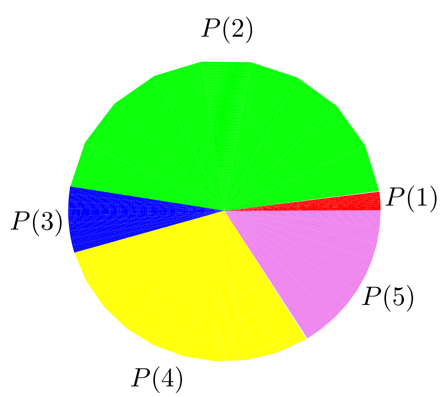

Fig. 11. The previous heuristics-probabilities pie chart when $\mathsf{conf} = 2$

6.2. *PopsSample declaration using our search methods framework*

Figure 12 expresses graphically the PopsSample method using our search methods framework (Section 3).

One essential difference with DFS is the instantiation goal for each variable. `Instantiate3` takes one more argument, namely *CoveredPiece*. When this exceeds PieceToCover, the goal fails.

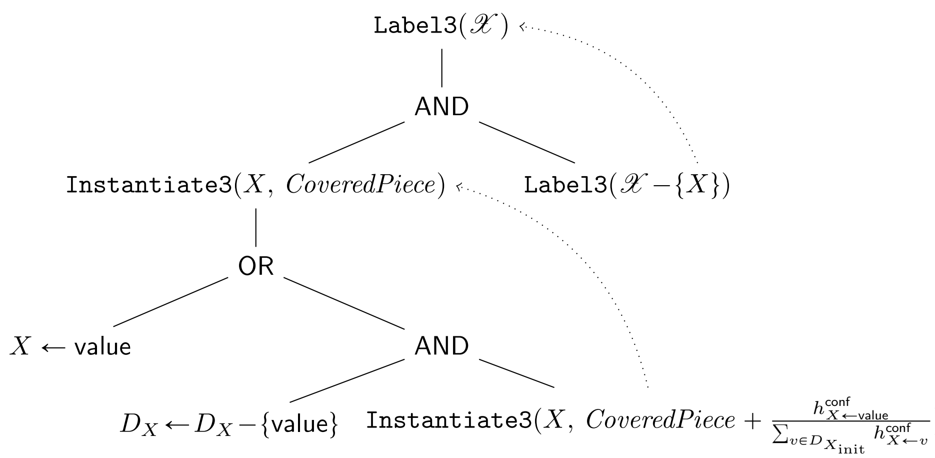

Fig. 12. The goals that build up PopsSample

- `Instantiate3`$(X,\ \mathit{CoveredPiece}) :=$ failure, if $D_X = \emptyset$,
- `Instantiate3`$(X,\ \mathit{CoveredPiece}) :=$ failure, if $\mathit{CoveredPiece} >$ PieceToCover,
- `Instantiate3`$(X,\ \mathit{CoveredPiece}) :=$ OR$(X \leftarrow$value, AND$(D_X \leftarrow D_X - \{\text{value}\}$, `Instantiate3`$(X,\ \mathit{CoveredPiece} + \frac{h^{\text{conf}}_{X\leftarrow\text{value}}}{\sum_{v\in D_{X_{\text{init}}}} h^{\text{conf}}_{X\leftarrow v}})))$, otherwise.

6.3. *Heuristic confidence vs. node level*

An important detail in PopsSample appearing in Fig. 9, is the increase in conf as the current search tree node level deepens.

When we make the first recursive PopsSample call (inside **while**), we have already made an assignment. Hence, the current tree level will be augmented by 1 and conf will be increased by $\frac{100-\text{conf}}{|\mathscr{X}|}$.

Each subsequent recursive call deepens search by 1, until the current depth reaches $|\mathscr{X}|$, which means that every variable in $\mathscr{X}$ has been assigned a value. For a specific depth k the conf value is increased by $k \cdot \frac{100-\text{conf}}{|\mathscr{X}|}$. Finally, when $k = |\mathscr{X}|$, the conf argument of PopsSample will become equal to the value 100.

The following is not guaranteed, but in the deepest node levels, heuristics are *usually* more accurate, because more variables have been instantiated, and we have a clearer picture of the problem. In our framework, more accuracy means more confidence, that's why we increase conf as the search method proceeds with the assignments.

6.4. *PopsSample average complexity*

The PopsSample complexity depends on PieceToCover argument and the heuristic function distribution.

Lemma 6.1. *Let n be the constrained variables number and let d be the average* *domain size. Then, the average complexity of a* PopsSample($\mathsf{PieceToCover}, \mathsf{conf}$) *call is* $O(d^n \cdot \mathsf{PieceToCover}^n)$.

Proof. An initial PopsSample($\mathsf{PieceToCover}, \mathsf{conf}$) call iterates through the values of, let's say, the first variable X_1. If the heuristic function numbers for the values in D_{X_1} are uniformly distributed, the expected value for $h_{X_1 \leftarrow \mathsf{value}}$ would be $\mu = \frac{\sum_{v \in D_{X_1}} h_{X \leftarrow v}}{|D_{X_1}|}$.

Thus, to reach the pie proportion $A = \mathsf{PieceToCover} \cdot \sum_{v \in D_X} h_{X \leftarrow v}$, we need $A/\mu = \mathsf{PieceToCover} \cdot |D_{X_1}|$ iterations, i.e. $O(\mathsf{PieceToCover} \cdot d)$ loops.

The total time needed is $T_1 = O(\mathsf{PieceToCover} \cdot d) \cdot T_2$, where T_2 is the time for the PopsSample call *inside* the loop. It also holds that $T_2 = O(\mathsf{PieceToCover} \cdot d) \cdot T_3$, etc., and finally $T_n = O(\mathsf{PieceToCover} \cdot d)$. In conclusion, the aggregate complexity is $O(\mathsf{PieceToCover}^n \cdot d^n)$ for the initial call. □

We can observe that PopsSample($1, \mathsf{conf}$) is equivalent to a complete search space exploration, which has an $O(d^n)$ time complexity.

6.5. *The motivation behind PoPS*

Finding the best conf is the motivation behind PoPS. Unfortunately, we do not know a priori which conf is the best parameter for PopsSample. However, we can find it by trial and error. In Fig. 13, the PoPS function invokes PopsSample for SamplesNum different conf_i values, including the values 0 and 100.

Each different conf_i is used in turn. Initially, the Cover_i parameter in the PoPS algorithm is zero for every conf_i. When a specific conf_i has been examined, the corresponding Cover_i is increased by $\frac{1}{d}$, where d is the average domain size. When the second iteration over a specific conf_i ends, the Cover_i is increased again by $\frac{1}{d}$ and so on.

In this way, each conf_i is given the same opportunity (search space) to find a solution. If some conf_i does not produce a solution, it is deactivated. It is reactivated only if all other conf_i's fail to produce a solution.

7. Empirical Evaluations

The gradual switch from randomness to determinism can boost search in demanding CSPs, such as course scheduling and the radio frequency assignment problems. With the help of our free constraint programming C++ library Naxos Solver,[5] we solved official instances of these problems for different heuristic distribution configurations.

The source code for our evaluations is freely available at `http://di.uoa.gr/~pothitos/PoPS` including the problem datasets. The experiments were conducted on an HP computer with an Intel dual-core E6750 processor clocked at 2.66 GHz with 2 GB of memory and a Xubuntu Linux 12.04 operating system.

function PoPS
  local variables:
    SamplesNum: how many different conf values are initially employed
    conf_i: array with all the initially employed conf values
    Sample_i: a Boolean array; if its i^{th} element is false, the corresponding conf_i value is currently ignored
    Cover_i: corresponding "piece to cover" argument for PopsSample call
    d: average domain size of the constrained variables

  for i **from** 1 **to** SamplesNum **do**
    Sample_i is activated
    $\mathsf{Cover}_i \leftarrow 0$
    $\mathsf{conf}_i \leftarrow 100 \cdot \frac{i-1}{\mathsf{SamplesNum}-1}$
  end for
  while the available time is not exhausted **do**
    for each active Sample_i **do**
      if PopsSample($\mathsf{Cover}_i, \mathsf{conf}_i$) did not return a solution **then**
        Sample_i is deactivated
      end if
      $\mathsf{Cover}_i \leftarrow \mathsf{Cover}_i + \frac{1}{d}$
    end for
    if every Sample_i is deactivated **then**
      Activate every Sample_i ▷ to keep searching.
    end if
  end while
end function

Fig. 13. Piece of Pie Search (PoPS) Method

In the following three subsections (7.1, 7.2, 7.3) the experiments are repeated for different conf values, as we do not use PoPS. On the other hand, in the last subsection 7.4, PoPS automatically chooses by itself the employed conf values.

7.1. *University course scheduling*

Automated timetabling is nowadays a crucial application, as many educational institutions still use ad hoc manual processes to schedule their courses. The International Timetabling Competition (ITC) is an attempt to unify all these processes. We borrowed the fourteen instances of the latest contest track concerning *universities*.[35]

In these problems, we have to assign valid teaching periods and rooms to the curriculum lectures. The objective is to distribute them evenly during the week but without having gaps between them, if scheduled on the same day; each gap increases the solution cost.[36] As *variables ordering heuristic*, we used minimum remaining values and degree for tie breaking, and we randomized it using the function in Lemma 5.1.

Due to the ITC specifications, we had 333 seconds in our machine to solve each instance and minimize the solution cost as much as we could. Figures 14 and 15 display the minimum solution costs found per instance for various `conf` values. We observe that as `conf` increases the costs tend to a specific number, whilst for small `conf` values we have fluctuations because search becomes more random.

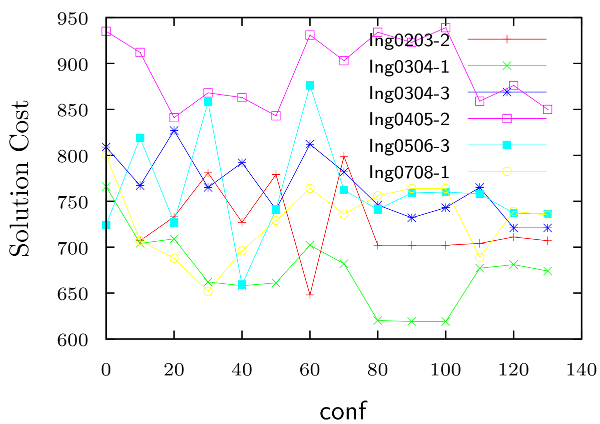

Fig. 14. Timetabling solutions costs vs. `conf`

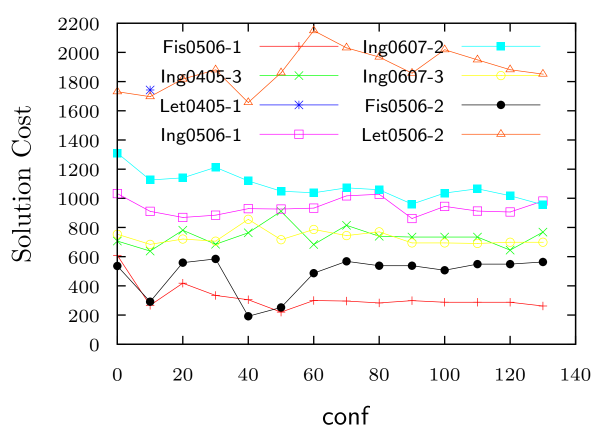

Fig. 15. Solutions for the rest of the ITC instances

It was expected that for high `conf` values the results would be more constant, as the search process approximates the default depth-first-search (DFS). For the marginal low values, e.g. conf $= 0$, search is completely stochastic and the results are worse on average, as we have higher solution costs. However, the evaluations for intermediate conf values, e.g. conf ≈ 20, are more promising. Remember that an intermediate conf value favors the selection which corresponds to the best heuristic evaluation, *but* it also gives room to other selections (the "outsiders") as their probabilities are not zero.

The automatic selection of the best conf is an open question here; in Section 7.4, PoPS finds automatically the best conf values.

In practice, as shown in Fig. 14 and 15, a conf value around 100 actually represents infinity, because search tends to produce the same solutions for conf $\geqslant 100$.

It is worth to mention that in Fig. 15 the only solution found for the Let0405-1 instance, depicted with an asterisk $*$, was for an intermediate conf $= 10$.

7.2. *Radio link frequency assignment*

Another important real problem is the frequency assignment, in which we have to assign a frequency to each radio transmitter with the objective to minimize the interference. The interference is minimized by assigning different frequencies to every two transmitters that are close to each other.

The Centre Electronique de l'Armement (CELAR) provides a set of real datasets for this NP-hard problem.[3] We chose to solve the five so-called "MAX" problem instances, namely SCEN06–SCEN10, in which, generally speaking, we try to maximize the number of the satisfied soft constraints. Similarly to the above course scheduling experiments, as *variables ordering heuristic* we used minimum remaining values and degree for tie breaking, and we randomized it using the function in Lemma 5.1.

For each of these instances, we had 15 minutes to explore the search space. We recorded the best (lowest) solution costs found so far in Fig. 16 for several conf values. Approximately the same as in course scheduling, the lowest solution costs occur around conf ≈ 10, which gives better results on average than the marginal conf values. This means that we achieve best results when the confidence to our heuristic is neither too high nor very low.

7.3. *PopsSample during hard optimization*

The conf parameter can refine any search method that adopts our heuristic framework. The PopsSample method goes a step further: it incorporates our heuristic *confidence semantics* into its search engine.

In order to solve the first university course timetabling instance (Fis0506-1 of Section 7.1), we invoked PopsSample for various PieceToCover and conf values and we plotted the best solution costs found in Figure 17. The third dimension is the *cost* of the solutions found: the lower the solution cost is, the more qualitative timetable is produced.

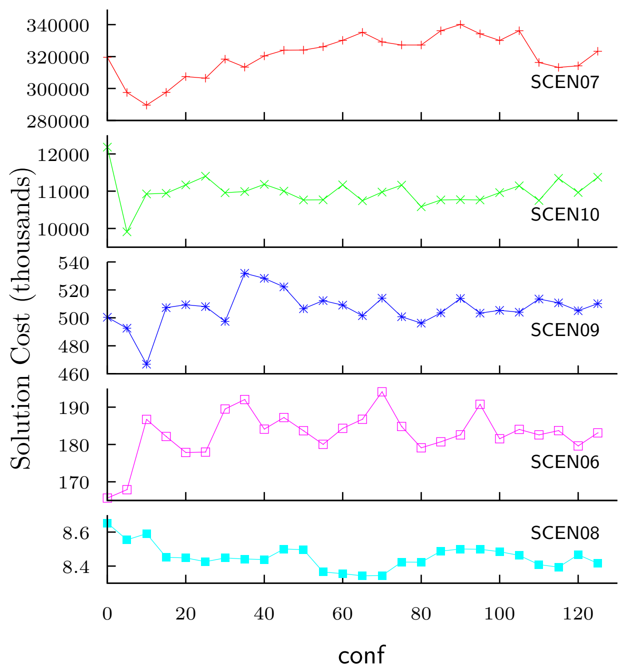

Fig. 16. Unsatisfied soft constraints increase cost

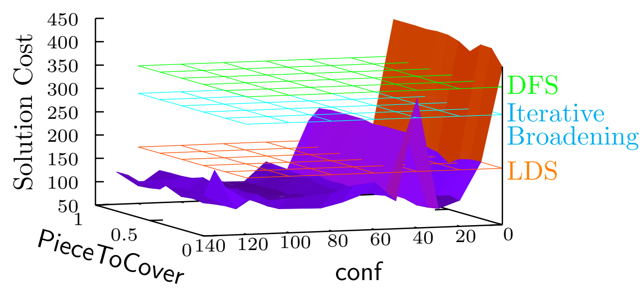

Fig. 17. PopsSample for the first ITC instance

In the same graphs, we include some of the well-known search methods results, such as DFS, LDS, and Iterative Broadening, implemented in the same solver, with only their best solution cost depicted as a plane grid, in order to make comparisons easily.

7.4. *PoPS vs. other search methods*

In the above sections, it was not easy to figure out which is the best PieceToCover and conf combination. That is why we employed PoPS to solve the fourteen course timetabling instances.

As described in Section 6.5, PoPS uses several conf values and favors the most fruitful ones. We used five conf samples, i.e. 0, 25, 50, 75, and 100, by setting SamplesNum equal to 5. In this way, PoPS constructed solutions with lower costs than the other methods, except for the fifth instance, as illustrated in Table 1.

In this section, we used least constraining value as ValuesOrderHeuristic, and we randomized it using the function in Lemma 5.1. The time limit for all the methods was set to 15 minutes.

Table 1. Solution costs for fourteen ITC instances

Instance	PoPS	LDS	DFS	It. Broad.
Fis0506-1	*105*	171	345	286
Ing0203-2	*241*	288	698	321
Ing0304-1	*279*	307	578	353
Ing0405-3	*195*	215	817	235
Let0405-1	655	*627*	X	X
Ing0506-1	*307*	311	812	342
Ing0607-2	*282*	283	1184	328
Ing0607-3	*223*	239	635	262
Ing0304-3	*288*	294	675	370
Ing0405-2	*265*	284	877	344
Fis0506-2	*12*	33	486	34
Let0506-2	*713*	783	1621	937
Ing0506-3	*231*	256	660	280
Ing0708-1	*223*	227	660	264

8. Conclusions and Perspectives

The initial contribution of this work is the provision of an interface that everyone can use to define their search methods. Apart from easing the declaration of custom search methods, we elaborated on the algorithm behind the scenes supporting our interface in an open source solver.

We also presented a well-founded paradigm to exploit both stochastic and deterministic heuristics. Empirical evaluations showed that our hybrid approach can produce better results than fully random or fully deterministic methodologies.

In order to achieve this, we approached and used heuristics as a *confidence* and *reliability* measure. By exploiting these heuristic semantics, we were able to produce a new efficient search method, namely PoPS, that can outperform other methodologies. In general, our proposed framework gives the opportunity to exploit “on the fly” whichever heuristic confidence fluctuations occur.

In the future, it will be challenging to parallelize it, as it supports a whole grid of strategies, by concurrently invoking PopsSample with several PieceToCover and conf arguments.

Constraint Programming consists of the CSP *definition* and *search* phases. A crucial goal in this area is to make these two phases as independent as possible of each other.[6] The presented search methods interface was one step into making the search phase more transparent. But the search phase doesn't include only the search method; it includes also mechanisms that check if the constraints are violated and enforces a kind of consistency between the domains of the variables that are connected via constraints.[37]

Therefore, in the next years, it would be also important to modularize the consistency enforcement part of the search phase, as we did in this work for the search methods.

Acknowledgments

We want to thank Foivos Theocharis who initially built the search methods library Amorgos, which is available together with Naxos.[5] We also thank the anonymous reviewers for their constructive comments.

References

1. M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, fourth edn. (Springer, New York, 2012).
2. A. M. Malik, J. McInnes and P. van Beek, Optimal basic block instruction scheduling for multiple-issue processors using constraint programming, *International Journal on Artificial Intelligence Tools* **17**(01) (2008) 37–54.
3. B. Cabon, S. de Givry, L. Lobjois, T. Schiex and J. P. Warners, Radio link frequency assignment, *Constraints* **4**(1) (1999) 79–89.
4. P. Barahona, L. Krippahl and O. Perriquet, Bioinformatics: A challenge to constraint programming, in *Hybrid Optimization*, eds. P. Van Hentenryck and M. Milano, **45** (Springer, New York, 2011) pp. 463–487.
5. N. Pothitos, Naxos Solver `http://github.com/pothitos/naxos`, (2017).
6. E. C. Freuder and B. O'Sullivan, Grand challenges for constraint programming, *Constraints* **19**(2) (2014) 150–162.
7. D. Zhang, Y. Liu, R. M'Hallah and S. C. Leung, A simulated annealing with a new neighborhood structure based algorithm for high school timetabling problems, *European Journal of Operational Research* **203**(3) (2010) 550–558.
8. E. K. Burke, J. Mareček, A. J. Parkes and H. Rudová, Decomposition, reformulation, and diving in university course timetabling, *Computers & Operations Research* **37**(3) (2010) 582–597.
9. R. J. Vanderbei, *Linear Programming: Foundations and Extensions*, fourth edn. (Springer, New York, 2014).
10. L. Fortnow, The status of the P versus NP problem, *Communications of the ACM* **52**(9) (2009) 78–86.
11. N. Pothitos and P. Stamatopoulos, Piece of Pie Search: Confidently exploiting heuristics, in *SETN 2016: 9th Hellenic Conference on Artificial Intelligence*, eds. N. Bassili-

ades, A. Bikakis, D. Vrakas, I. P. Vlahavas and G. A. Vouros (ACM, New York, 2016), pp. 8:1–8:8.
12. E. Tsang, *Foundations of Constraint Satisfaction* (Books on Demand, Norderstedt, 2014).
13. ECL^{i}PS^{e} constraint programming system `http://eclipseclp.org`, (2017).
14. ILOG SOLVER `http://ilog.com/products/cp`, (2017).
15. S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach* (Prentice Hall, 2010), ch. 6, pp. 202–233, third edn.
16. I. P. Gent and T. Walsh, CSPLIB: A benchmark library for constraints, in *CP 1999: 5th International Conference on Principles and Practice of Constraint Programming, Alexandria, Virginia*, ed. J. Jaffar *LNCS* **1713**, (Springer, Heidelberg, 1999), pp. 480–481. `http://CSPLib.org`.
17. S. Boyd and L. Vandenberghe, *Convex Optimization* (Cambridge University Press, 2004).
18. Gecode: Generic constraint development environment `http://gecode.org`, (2017).
19. M. L. Ginsberg and W. D. Harvey, Iterative broadening, *Artificial Intelligence* **55**(2-3) (1992) 367–383.
20. J.-C. Régin, M. Rezgui and A. Malapert, Improvement of the embarrassingly parallel search for data centers, in *CP 2014*, ed. B. O'Sullivan, *LNCS* **8656** (Springer International Publishing, Switzerland, 2014) pp. 622–635.
21. N. Pothitos and P. Stamatopoulos, Constraint Programming MapReduce'd, in *SETN 2016: 9th Hellenic Conference on Artificial Intelligence*, eds. N. Bassiliades, A. Bikakis, D. Vrakas, I. P. Vlahavas and G. A. Vouros (ACM, New York, 2016), pp. 5:1–5:4.
22. P. Prosser and C. Unsworth, Limited discrepancy search revisited, *J. Experim. Algor.* **16** (2011) 1.6:1–1.6:18.
23. R. Barták, Incomplete depth-first search techniques: A short survey, in *CPDC 2004: 6th Workshop on Constraint Programming for Decision and Control* (Institute of Automation Control, Silesian University of Technology, 2004), pp. 7–14.
24. R. Barták and H. Rudová, Limited assignments: A new cutoff strategy for incomplete depth-first search, in *SAC 2005: Symposium on Applied Computing, Santa Fe, New Mexico*, eds. H. Haddad, L. M. Liebrock, A. Omicini and R. L. Wainwright (ACM, New York, 2005), pp. 388–392.
25. T. Walsh, Depth-bounded discrepancy search, in *IJCAI 1997: 15th International Joint Conference on Artificial Intelligence, Nagoya, Japan*, ed. M. E. Pollack **2**, (Morgan Kaufmann, San Francisco, 1997), pp. 1388–1393.
26. Y. Xu, D. Stern and H. Samulowitz, Learning adaptation to solve constraint problems, in *LION 3: 3rd International Conference on Learning and Intelligent Optimization* (Springer, 2009).
27. B. Jafari and M. Mouhoub, Heuristic techniques for variable and value ordering in CSPs, in *GECCO 2011: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, Dublin* (ACM, New York, 2011), pp. 457–464.
28. H. H. Hoos and E. Tsang, Local search methods, in *Handbook of Constraint Programming*, eds. F. Rossi, P. van Beek and T. Walsh, *Foundations of Artificial Intelligence* (Elsevier Science, Amsterdam, 2006) pp. 135–167.
29. W. Cohen, R. Greiner and D. Schuurmans, Probabilistic hill-climbing, in *Comp. Learn. Theory and Natural Learn. Syst.*, eds. S. J. Hanson, T. Petsche, M. Kearns and R. L. Rivest **II**, (The MIT Press, Cambridge, 1994), pp. 171–181.
30. S. Kirkpatrick, C. D. Gelatt, Jr. and M. P. Vecchi, Optimization by simulated annealing, *Science* **220**(4598) (1983) 671–680.
31. J. L. Bresina, Heuristic-biased stochastic sampling, in *AAAI 1996: 13th National Con-*

ference on Artificial Intelligence, Portland, Oregon, eds. W. J. Clancey and D. S. Weld **1**, (AAAI Press, Menlo Park, 1996), pp. 271–278.
32. V. A. Cicirello and S. F. Smith, Enhancing stochastic search performance by value-biased randomization of heuristics, *Journal of Heuristics* **11**(1) (2005) 5–34.
33. C. P. Gomes, B. Selman, N. Crato and H. Kautz, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, *Journal of Automated Reasoning* **24**(1-2) (2000) 67–100.
34. D. Sharma, V. Singh and C. Sharma, GA-based scheduling of FMS using roulette wheel selection process, in *SocProS 2011: International Conference on Soft Computing for Problem Solving* **131**, (Springer, 2012), pp. 931–940.
35. B. McCollum, A. Schaerf, B. Paechter, P. McMullan, R. Lewis, A. J. Parkes, L. Di Gaspero, R. Qu and E. K. Burke, Setting the research agenda in automated timetabling: The second international timetabling competition, *INFORMS Journal on Computing* **22**(1) (2010) 120–130.
36. N. Pothitos, P. Stamatopoulos and K. Zervoudakis, Course scheduling in an adjustable constraint propagation schema, in *ICTAI 2012: 24th IEEE International Conference on Tools with Artificial Intelligence* **1**, (IEEE, 2012), pp. 335–343.
37. H. Chen, V. Dalmau and B. Grußien, Arc consistency and friends, *Journal of Logic and Computation* **23**(1) (2013) 87–108.

Appendix A. An Algorithm that Satisfies Search Methods' Goals

In Section 3 we described a high-level language to define search methods. This framework consists of goals that support recursion (goals that return another goal or success) and meta-goals used to combine other goals in a conjunctive (AND) or disjunctive (OR) manner.

This is a not only a theoretical model; it has been implemented in a C++ Constraint Programming NAXOS SOLVER. DFS and Iterative Broadening have been already declared in NAXOS without losing much of the above expressiveness, along with many other search methods in the solver's repository.[5]

The search methods' goals cannot solve by themselves any CSP; we need a procedure to *satisfy* these goals. SOLVE algorithm in Figure 18 uses an advanced "stack of stacks" data structure to store goals. Initially, the stack of stacks contains just a single goal, e.g. `Broadening`(1) or `Label`($\mathscr{X}$) for DFS, as in Figure 19(a).

Appendix A.1. *The "stack of stacks" data structure*

In order to understand the SOLVE pseudocode, we should look closer at the underlying data structure in Fig. 19. Each subfigure from (a) to (v) displays a snapshot of the "stack of stacks" main data structure while trying to satisfy the `Label`($\mathscr{X}$) goal for the Thessaly-coloring problem. Here we focus on the data structure itself.

- The "stack of the stacks" is outlined by the outer borders of each subfigure.
  - The frames of the (outer) stack are connected with an OR relationship.
- Each frame of the outer stack is a stack too, containing goals.
  - The goals of each (inner) stack are connected with an AND relationship.

```
function Solve
  local variables:
    stacks: the "stack of stacks" instance
    Goal: current goal that has to be satisfied
    NextGoal: generated goal by current goal's execution

  while time limit has not been reached do
    if stacks.top is not empty then
      Goal ← stacks.top.pop()
    else
      Goal ← stacks.top.pending
      stacks.top.pending ← get the next goal of the frame that "pending"
                          points to or (if there is not such a goal) get
                          the next "pending" goal of that frame
    end if
    if Goal is an AND-goal then
      stacks.top.push(2nd subgoal of Goal)
      stacks.top.push(1st subgoal of Goal)
    else if Goal is an OR-goal then
      stacks.top.push(2nd subgoal of Goal)
      stacks.push()                                  ▷ Push an empty frame
      stacks.top.push(1st subgoal of Goal)
      stacks.top.pending ← the goal after the 2nd subgoal in the below frame
    else
      NextGoal ← Goal.execute()
      if PropagateConstraints() = false then
        stacks.pop()                          ▷ Backtrack to previous frame
        if stacks empty then
          return failure
        end if
      else if NextGoal ≠ null then
        stacks.top.push(NextGoal)
      else if stacks.top is empty and stacks.top.pending = null then
        return success
      end if
    end if
  end while
  return failure
end function
```

Fig. 18. Goals' Satisfaction Algorithm

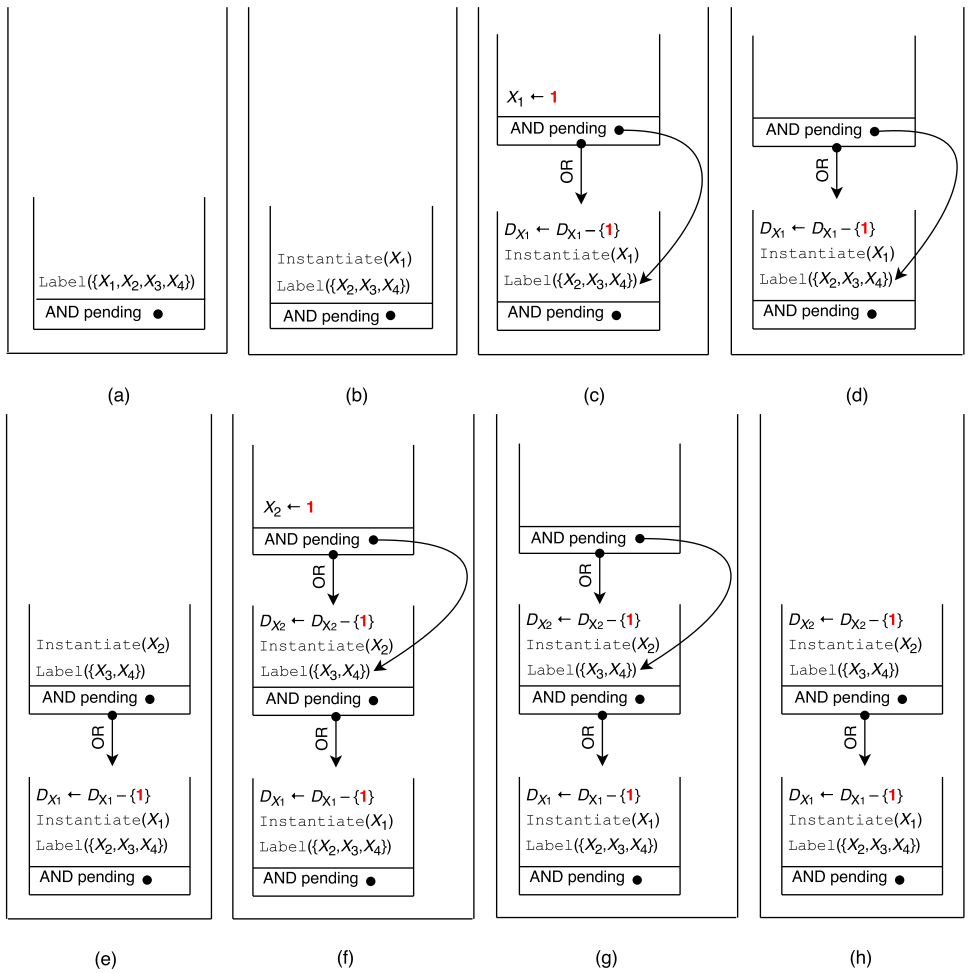

Fig. 19. SOLVE with DFS example: X_1 instantiation and X_2 instantiation attempt

- Each inner stack contains additionally a pending pointer.
  - It points to the first unsatisfied goal of the previous inner stacks.

Appendix A.2. *SOLVE in a nutshell*

We can see the “stack of stacks” data structure inside the SOLVE algorithm.

- The outer stack corresponds to the “stacks” variable.
- The “stacks.top” expression corresponds to the top frame of the “stacks.”
- The “stacks.top.pending” corresponds to the pointer of the top frame.

The algorithm is wrapped into a **while** loop. Each iteration examines another goal.

Inside the loop we have two "if" statements. In the first "if," we select the goal that we will attempt to satisfy. In the second "if," we check if the *Goal* is a meta-goal and we handle properly its two subgoals. If it is not a meta-goal, we simply "execute()" it and store the returned value into the *NextGoal* variable. A "null" returned value means that the executed *Goal* did not generate another goal to be satisfied.

After each *Goal* execution, PropagateConstraints is called. A simple implementation for this function would just check if every constraint is still valid. If even a single constraint is violated, PropagateConstraints should return false. However, as its name implies, PropagateConstraints can do more just than checking constraints: it may remove no-good values from the variables and enforce a kind of "consistency" between them, but this is beyond the scope of this paper to further analyze.[37]

Appendix A.3. *Solve in action*

Let's execute Solve to find a solution to the Thessaly-coloring (Problem 1) using the DFS method goals (Section 3.3). The snapshots of the "stack of stacks" data structure are visible in the corresponding subfigures of Figures 19 to 22:

(a) Before Solve begins, the first goal to be satisfied should be already in the single inner stack. In the case of DFS, the initial goal is `Label`$(\mathscr{X})$ which in Thessaly-coloring is equivalent to `Label`$(\{X_1, X_2, X_3, X_4\})$.

(b) Solve begins and pops the `Label` goal out of the inner stack and executes it. By definition, `Label` returns an AND goal. In other words, `Label` is substituted by an AND goal at the inner stack.

This AND goal is then popped out in turn by the next Solve iteration, and after its execution it returns its two subgoals `Instantiate`$(\{X_1\})$ and `Label`$(\{X_2, X_3, X_4\})$. At this time, the "stack of stacks" looks like Fig. 19(b).

(c) By definition, `Instantiate`$(\{X_1\})$ is substituted by an OR goal. The complete expression for this goal is OR$(X_1 \leftarrow 1$, AND$(D_{X_1} \leftarrow D_{X_1} - \{1\}$, `Instantiate`$(X_1)))$. In this case, Solve algorithm should cover three requirements.

1. Execute the first subgoal and all the returned/generated goals.
2. Execute the "pending" goals that were unsatisfied before the OR-goal.
3. If the above fail, undo all the actions and execute the second subgoal.

In Fig. 19(c), all three requirements have been implemented.

1. The first subgoal is in the top stack and will be executed in the next iteration.
2. The "pending" pointer of the top stack points to the next unsatisfied goal.
3. Most importantly, a new stack has been created on top of the previous inner stack. The new stack was added in order to isolate the previous inner stack. This would be useful if any of the top stack goals fails: The top stack will

be popped, and the second OR-subgoal that is stored in the previous inner stack will be executed.

For simplicity reasons, instead of adding the whole second subgoal $\mathsf{AND}(D_{X_1} \leftarrow D_{X_1} - \{1\}, \texttt{Instantiate}(X_1))$ into the bottom stack in Fig. 19(c), we "unwrapped" it and added directly its two subgoals.

(d) $X_1 \leftarrow 1$ is executed. This assignment does not violate any constraint. This goal did not return another goal, so the top stack is now empty.

(e) As the top stack is empty, SOLVE uses its "pending" pointer to fetch the next goal. Therefore, $\texttt{Label}(\{X_2, X_3, X_4\})$ is fetched and the "pending" pointer is moved one step further, to the next goal. As there aren't any more goals, the top stack "pending" pointer is assigned the bottom stack "pending" value, which is "null."

The `Label` goal is executed and returns $\mathsf{AND}(\texttt{Instantiate}(X_2), \texttt{Label}(\{X_3, X_4\}))$, which is executed in turn in the next iteration. The result is depicted in Fig. 19(e).

(f) $\texttt{Instantiate}(X_2)$ is executed. As in (c), this is an OR goal, and a new stack is pushed.

(g) $X_2 \leftarrow 1$ gets executed.

(h) The assignment makes PROPAGATECONSTRAINTS fail, as it violates the $X_1 \neq X_2$ constraint. Backtracking, i.e. popping the whole top stack, is activated.

(i) $D_{X_2} \leftarrow D_{X_2} - \{1\}$ is successfully executed. This removes the "no-good" value.

(j) $\texttt{Instantiate}(X_2)$ is executed again. However, this time the corresponding domain D_{X_2} does not contain the removed value 1. Thus, the only value left to assign to X_2 is 3.

(k) $X_2 \leftarrow 3$ is successfully executed.

(l) Top stack is empty, so we fetch the "pending" goal $\texttt{Label}(\{X_3, X_4\})$ and set "pending" equal to "null." The `Label` goal returns an AND goal. Its two subgoals are pushed on the top stack.

(m) $\texttt{Instantiate}(X_3)$ returns an OR goal. When this is executed, another stack is pushed on top of the others.

(n) $X_3 \leftarrow 1$ is executed.

(o) PROPAGATECONSTRAINTS fails, as the constraint $X_1 \neq X_3$ is violated. The top stack is immediately popped.

(p) The no-good value is removed after $D_{X_3} \leftarrow D_{X_3} - \{1\}$ execution.

(q) $\texttt{Instantiate}(X_3)$ is executed again for the new domain.

(r) The new assignment $X_3 \leftarrow 2$ is executed.

(s) PROPAGATECONSTRAINTS now succeeds and, as the top stack is empty, we proceed to the "pending" goal $\texttt{Label}(\{X_4\})$. The execution of this goal generates $\mathsf{AND}(\texttt{Instantiate}(X_4), \texttt{Label}(\{\}))$.

(t) Again, $\texttt{Instantiate}(X_4)$ produces an OR goal which, in turn, forces SOLVE to push another stack on top of the others.

(u) $X_4 \leftarrow 1$ is executed.

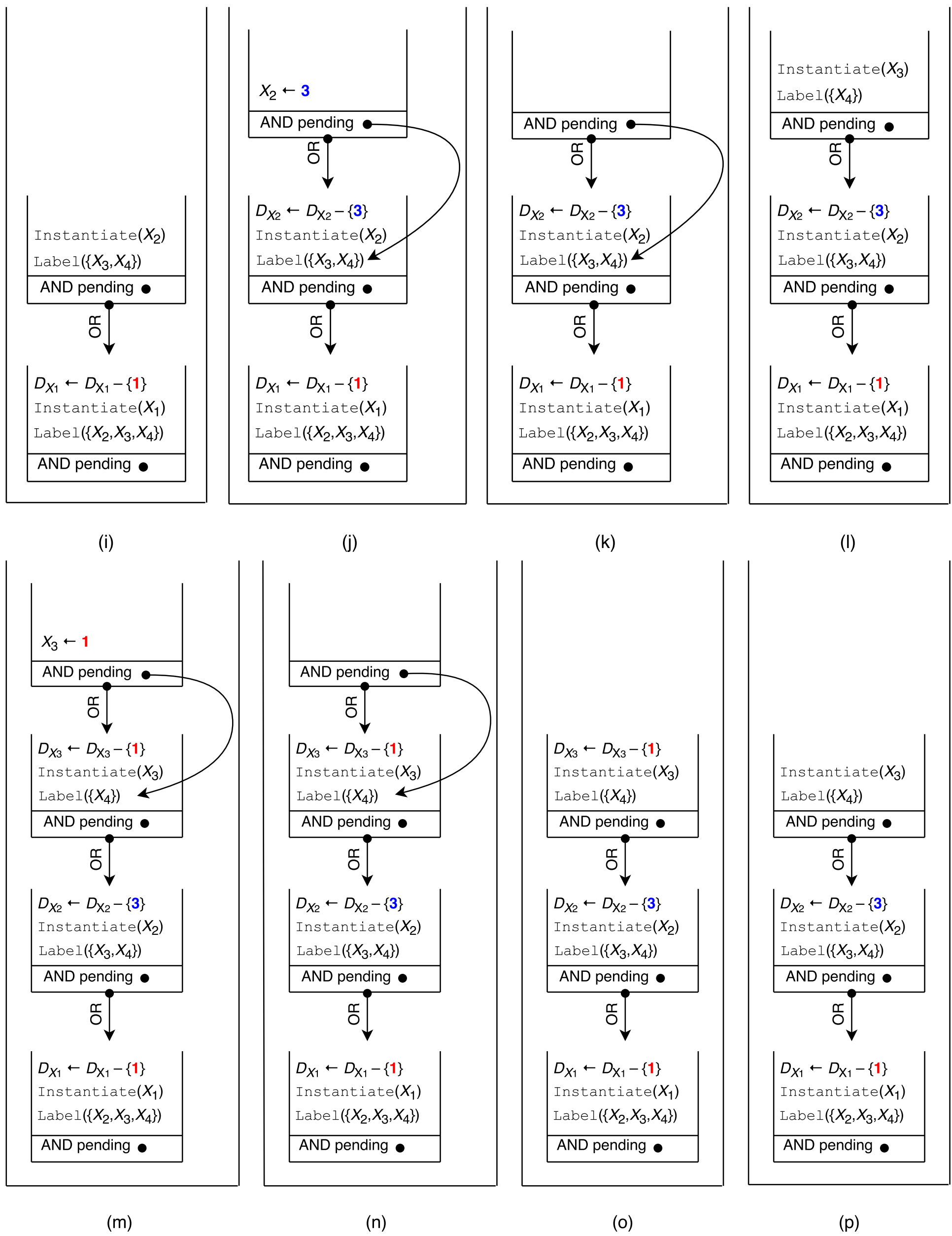

Fig. 20. Solve with DFS example: X_2 successful instantiation and X_3 instantiation attempt

(v) The "pending" `Label`($\varnothing$) is executed. By definition, this goal does not return any other goal. This means that the top stack is empty. And as there isn't any other "pending" goal, Solve has reached a solution and returns success!

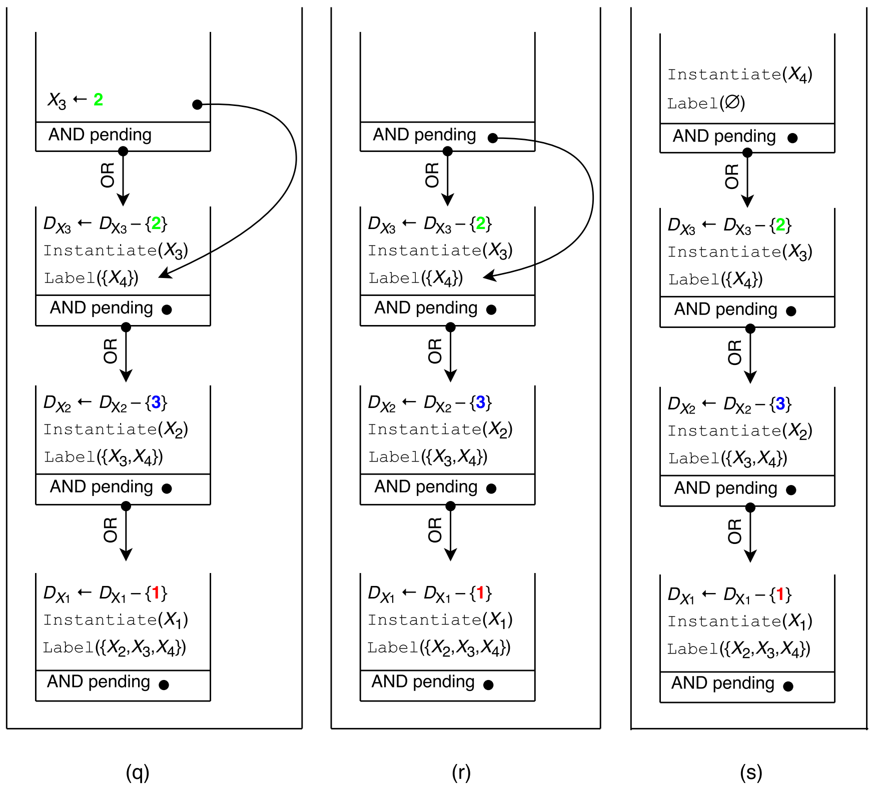

Fig. 21. SOLVE with DFS example: X_3 successful instantiation

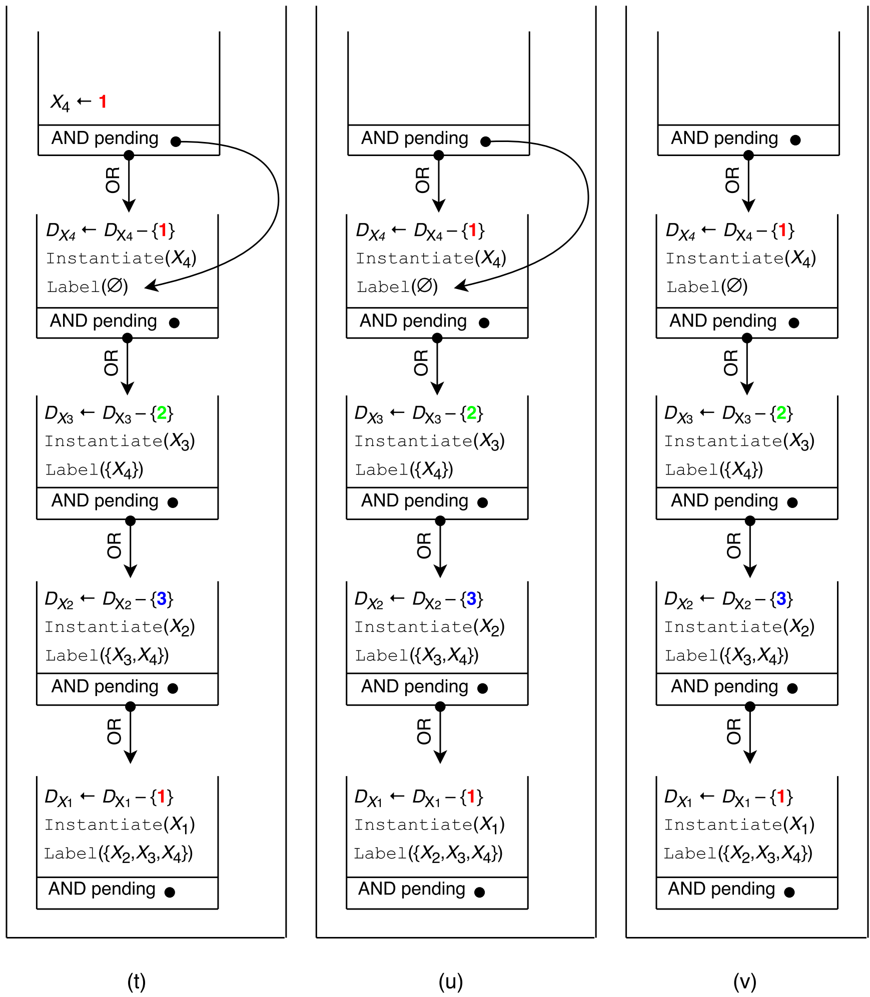

Fig. 22. Solve with DFS example: X_4 instantiation