Search for More Declarativity

Backward Reasoning for Rule Languages Reconsidered

Simon Brodt, François Bry, and Norbert Eisinger

Institute for Informatics, University of Munich,
Oettingenstraße 67, D-80538 München, Germany
http://www.pms.ifi.lmu.de/

Abstract. Good tree search algorithms are a key requirement for inference engines of rule languages. As Prolog exemplifies, inference engines based on traditional uninformed search methods with their well-known deficiencies are prone to compromise declarativity, the primary concern of rule languages. The paper presents a new family of uninformed search algorithms that combine the advantages of the traditional ones while avoiding their shortcomings. Moreover, the paper introduces a formal framework based on partial orderings, which allows precise and elegant analysis of such algorithms.

1 Introduction

The foremost advantage of rule languages is their *declarativity*. It allows problem-solving by specifying a problem's "what" without bothering about its "how". This separation of concerns makes it easy for rule authors to add or modify rules, thus supporting rapid prototyping, stepwise refinement, adaptation and evolution in application areas with unknown solution algorithms and/or frequently changing prerequisites.

Such unburdening of rule authors from control issues depends on a well-designed inference engine. Assuming that the underlying logical system features reasonable soundness and completeness properties, which it usually does, the most tricky design decision is to combine it with a search method that preserves all or most of these properties while still ensuring an adequate degree of efficiency.

The exact criteria for such design decisions are subject to several fundamental assumptions about the reasoning process, such as tuple-oriented vs. set-oriented or forward vs. backward reasoning. But we need not place special emphasis on those assumptions for the purpose of this paper. Although our motivation examples will use backward reasoning with definite rules, our concern is not the evaluation of this particular kind of rules, but a complete and space-efficient search method for rule engines in general. Such a search method is not only applicable to backward reasoning with and without memoization [13,15], but also to forward reasoning approaches using some goal guidance [3,4,6].

Given the wealth of research results on search [1,10,11,12,16, among many others], soberingly few actually come into consideration as candidates for rule

inference engines. Their bulk has been on *informed* search methods, on incorporating domain-specific knowledge into the search. But this is at odds with the very idea of rule-based systems: rules may represent domain-specific knowledge, but the inference engine evaluating them needs to be applicable to arbitrary rule sets and is therefore, for better or worse, generic and domain-independent. The same holds all the more for rules used in reasoning on the Web, where domain-specific knowledge is hardly available.

This narrows down the choice to *uninformed* search methods, of which there are barely a handful: breadth-first and depth-first search [7], iterative-deepening [8], iterative broadening [5]. All of them have weak points: storage requirements for breadth-first search can become prohibitive already for medium-size problems, depth-first search is incomplete in search spaces with infinite branches, the iterative variants re-evaluate parts of the search space over and over again.

Under these circumstances a sensible compromise seems to be the one chosen for Prolog: to use depth-first search and to give rule authors some control to avoid infinite dead ends, for example by ordering the rules. However, this compromise wreaks havoc on declarativity.

Assume a term representation for natural numbers where `zero` represents 0 and `succ(X,Y)` can provide the predecessor `X` to any `Y` representing a nonzero natural number. Consider the straightforward rules defining for this representation the predicates `nat`, $\texttt{nat}_2$ and `less`, together with four queries:

```
nat(zero) ←
nat(Y)    ← succ(X,Y) ∧ nat(X)
nat₂(X,Y) ← nat(X) ∧ nat(Y)
less(X,Y) ← "reasonably defined"
```

```
1 ← nat(X)
2 ← nat₂(X,Y)
3 ← less(zero,X) ∧ nat₂(X,Y)
4 ← nat₂(X,Y) ∧ less(zero,X)
```

Problem 1: Incomplete Enumeration. Query 1 results in an enumeration of $\mathbb{N}$, which is fine. One would expect query 2 to result in an enumeration of $\mathbb{N}\times\mathbb{N}$, but it only enumerates $\{0\}\times\mathbb{N}$. The reason is that depth-first backtracking search never reaches branches to the right of the first infinite one. Note that reorderings of rules or literals would not affect the problem.

Problem 2: Non-Commutativity of Logical Connectives. Assume single-answer mode[1] for queries 3 and 4. Then both queries ask about the existence of an $X > 0$ with $(X, Y) \in \mathbb{N}\times\mathbb{N}$ for some Y. The two queries are logically equivalent, but query 3 results in an affirmative answer and query 4 in a nonterminating evaluation giving no answer at all.

Such blatant infringements on declarativity are sometimes wrongly attributed to SLD-resolution, although it is perfectly sound and complete with any literal selection function [9]. The only cause of the problems is depth-first backtracking search. With a complete search method the problems would not arise.

Consequently, one way to avoid them is to replace depth-first search by iterative-deepening [14]. Unfortunately, this approach introduces a new problem.

```
even(zero) ←
even(Y)    ← succ(X,Y) ∧ odd(X)
odd(Y)     ← succ(X,Y) ∧ even(X)
```

```
5 ← constant(X) ∧ even(X)
```

[1] Single-answer mode in Prolog can be achieved by a cut at the end of each query.

Problem 3: Inefficiency on Functional Rule Sets. Let `constant(X)` bind `X` to the term representation of some fixed, large number $n \in \mathbb{N}$. The rules define relations that are functions. Evaluation of query 5 ought to require $O(n)$ steps, and so it does with depth-first backtracking search. Iterative-deepening, on the other hand, needs $O(n^2)$ steps.

Search should not slow down the evaluation of functional rules, which do not need any search in the first place. Some functional rules escape being slowed down thanks to the compiler's tail recursion optimisation. But this sidestepping the problem fails in "quasi tail recursive" cases like the above, which do not match typical tail recursion patterns but nevertheless induce almost linear search trees.

Desiderata for Search Methods. A search method for rule inference engines usually has to be *uninformed*, as discussed earlier. It ought to meet the following requirements, which are essentially a collection of all advantageous properties from traditional methods.

- *Completeness* (or *exhaustiveness/fairness*) on both finite and infinite search trees. It visits every node in the search space after finitely many steps. Recall that we want to apply it also for finding all solutions to a query, and if there are infinitely many, the method must be capable of a fair enumeration.
  Depth-first search and iterative broadening violate this requirement on infinite trees. Depth-bounded backtrack search and credit search [1] violate it even on finite trees.
- *Polynomial space complexity* $O(d^c)$ where c is a constant and d the maximum depth currently reached during the search (or of the entire tree, if it is finite).
  Breadth-first search has exponential worst-case space complexity $O(2^d)$.
- *Linear time complexity* $O(n)$ where n is the current number of nodes that have been visited at least once (or of the entire tree, if it is finite).
  Note that any non-repetitive method, which visits every node at most once, meets this requirement. Iterative-deepening does not, see problem 3 above.

Note that space and time complexity here depend on different variables. The desired space complexity $O(d^c)$ is *polynomial in depth* d. The desired time complexity $O(n)$ is *linear in size* n, and often $O(n) = O(b^d)$ for an upper bound b of the branching factor. Linear in size b^d is much larger than polynomial in depth d.

This paper introduces D&B-search, a new uninformed search method, which integrates depth-first and breadth-first search. It meets these desiderata, the basic algorithm even with space complexity linear in depth. D&B-search can be parameterised to turn it into a family of algorithms with breadth-first and depth-first search as its extremal cases. The parameter also allows to control the amount of storage provided for completeness.

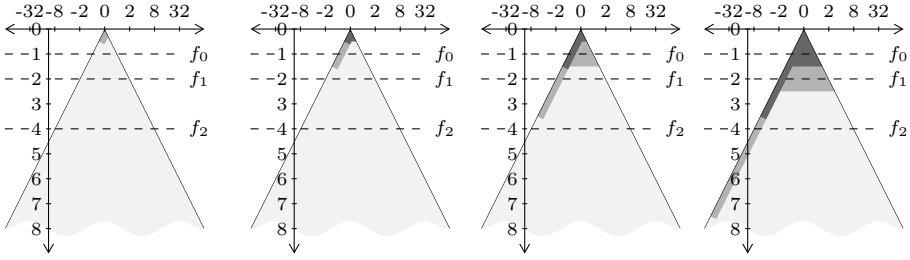
The paper is organised as follows. Section 2 presents D&B-search. A formal framework for the analysis of search methods follows in Section 3. Then Section 4 analyses D&B-search with this framework showing that it meets the desiderata above. Finally, Section 5 reports about the current state of development and plans for improvements.

2 D&B-Search and its Family of Algorithms

Let us abbreviate depth-first and breadth-first search by D-search and B-search, respectively. The idea of D&B-search is to alternate D-search with B-search, controlling their rotation by a sequence $f_0, f_1, f_2, \ldots$ of *depth bounds*. These are defined by a function $\mathbb{N} \to \mathbb{N},\ i \mapsto f_i$ with $i < f_i < f_{i+1}$ for $i \in \mathbb{N}$.

D-search starts, but may expand nodes at depth f_{i+1} or beyond only if all nodes at depth $\leq i$ have been expanded. If they haven't, B-search takes over. It may expand nodes at depth $i+1$ only if some node at depth f_{i+1} has been expanded before. If none has, D-search takes over again. And so on.

In this way no node is ever re-expanded, D&B-search is non-repetitive. Its principle bears some resemblance to the principle of A^*-search [10,11], which combines a heuristic estimate for fast advances into promising parts of the search space with a path-cost function ensuring a minimum degree of B-search behaviour and thus completeness. Likewise, D&B-search, which is uninformed and has no heuristics for "promising", combines fast D-search advancement with a minimum degree of B-search behaviour to ensure completeness. The following diagrams illustrate how D-search and B-search interact for $f_i = 2^i$.



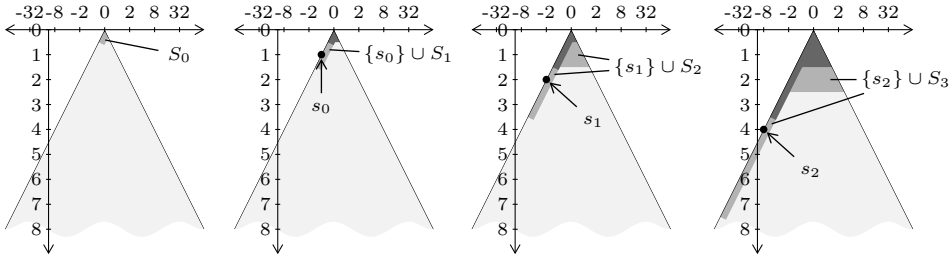
D-search advances exponentially faster than B-search. Hence the total number of nodes to be stored at any time (those on the branch traversed by D-search together with those at the deepest level reached by B-search) depends polynomially (for $f_i = 2^i$ even linearly) on the maximal depth reached up to that time. More details on space complexity will follow on page 6.

From an algorithm-oriented point of view it is better to focus not on the depth-bounds f_i, but on the nodes that serve as synchronisation points between D-search and B-search. Let us call a node "earlier" than a given one, if (unrestricted) D-search would expand it before expanding the given one.

For each depth-bound f_i its *pivot-node* s_i is the earliest (i. e., left-most) node at depth f_i. It is undefined if there are no nodes at depth f_i.

All other nodes are partitioned into finite sets. The *pre-pivot-set* S_0 is the set of nodes earlier than the pivot-node s_0. For each other pivot-node s_{i+1} let D_i be the set of nodes earlier than s_{i+1} and B_i the set of nodes at depth i. All nodes in these two sets must be expanded before expanding the pivot-node s_{i+1}, but some have already been expanded before earlier pivot-nodes. So the *inter-pivot-set*, i. e., the set of nodes expanded in-between s_i and s_{i+1}, is $S_{i+1} = (D_i \cup B_i) \backslash X_i$ where $X_0 = S_0 \cup \{s_0\}$ and $X_{i+1} = X_i \cup S_{i+1} \cup \{s_{i+1}\}$. Finally, the *post-pivot-set* R is empty if the tree is infinite. Otherwise there is a maximal $i_{\max}$ for which $s_{i_{\max}}$ is defined, and R is the set of all remaining nodes of the tree except $X_{i_{\max}}$.

Using these notions, reconsider the behaviour of D&B-search[2] for $f_i = 2^i$:



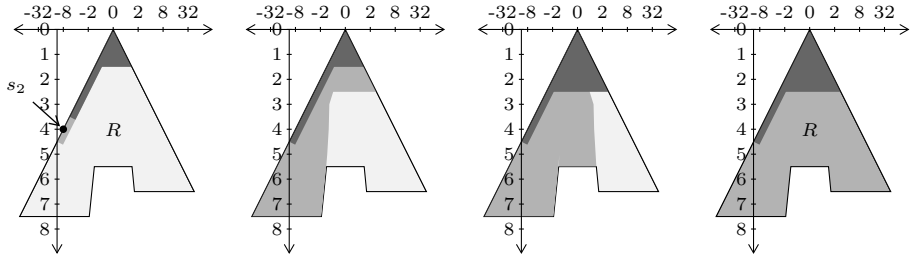
The general pattern is best seen for the last transition (the others are somewhat special). The third diagram shows the snapshot where all nodes in $X_1 \cup S_2 = S_0 \cup \{s_0\} \cup S_1 \cup \{s_1\} \cup S_2$ have been expanded (indicated by shading) and D-search is ready to expand the pivot-node s_2.

Next, D-search expands s_2, at which point the set of expanded nodes is X_2, and then continues until its next step would be to expand the pivot-node s_3. During this continuation it expands all nodes in $D_2 \backslash X_2 \subseteq S_3$. At this point control passes to B-search for expanding the remaining nodes in $B_2 \backslash X_2 \subseteq S_3$, the as-yet unexpanded nodes at depth 2. When done, all nodes in S_3 have been expanded and control passes back to D-search, which is now ready to expand the pivot-node s_3. This is the snapshot in the last diagram, the darkest shade indicating $X_2 \backslash \{s_2\}$, the medium shade indicating S_3 with $D_2 \backslash X_2$ on the left-most branch and $B_2 \backslash X_2$ at depth 2.

Let us now turn to the initial stages. D&B-search starts with D-search expanding all nodes in the pre-pivot-set S_0 (which contains only the root node for $f_0 = 1$, but would contain more for $f_0 > 1$). D-search is ready to continue with pivot-node s_0. The first diagram shows the snapshot at this point.

Then D-search expands s_0, then all nodes in $D_0 \backslash X_0 \subseteq S_1$ (of which there aren't any for $f_1 - f_0 = 1$). Its next step would be to expand s_1. Now control passes to B-search for expanding the remaining nodes in $B_0 \backslash X_0 \subseteq S_1$ (of which at depth 0 there aren't any). D-search is ready to continue with pivot-node s_1. This is the snapshot in the second diagram.

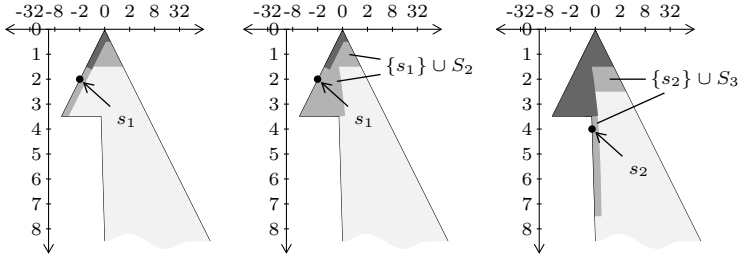
Altogether, D&B-search expands the nodes of the search tree in the order $S_0, s_0, \ldots, S_i, s_i, \ldots, R$. For finite trees this has interesting consequences:



[2] The pivot-nodes explain the *behaviour* of D&B-search, but not its implementation, where they are not directly available. They play an important role indirectly, though.

In a finite tree there are $i_{\max}$ pivot-nodes and $i_{\max} - 1$ inter-pivot-sets as well as sets B_i, part of whose nodes is all B-search ever expands. But $i_{\max}$ is small ($O(\log d)$ for $f_i = 2^i$) compared to the maximum depth d. So B-search stops quite soon. The overall behaviour is dominated by D-search in the post-pivot-set R.

In an infinite tree D-search cannot leave the left-most infinite branch. Everything "to the right" of this branch, the largest part of the search tree as it increases in size much faster than in depth, is therefore handled by B-search.



Although the merits of B-search for infinite trees are debatable, it is at least complete. So D&B-search has a kind of built-in adaptivity. Depending on the search tree it behaves essentially like the uninformed search method that best suits the tree, taking "best" with a pinch of salt.

This adaptivity effect would also apply to D&I-search (Section 5), a suggested combination of D-search with iterative-deepening search instead of B-search.

The D&B Family. Furthermore we can parameterise the function f_i with $c \in \mathbb{N} \cup \{\infty\}$ to define a family of algorithms.

Assume[3] that the tree's branching factor is bounded by $b \in \mathbb{N}$. An obvious idea is to let $f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor$ with $\frac{i}{\infty} := 0$, $\frac{i}{0} := \infty$ and $b^\infty := \infty$. However, these functions do not satisfy $i < f_{c,i} < f_{c,i+1}$ for $c \neq 1$. Therefore[4] let $f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor + i$. For this family of functions and algorithms we get:

- For $1 \leq c \leq \infty$ the algorithm is complete (for $c = 0$ it is not).
- For $1 \leq c < \infty$ its space complexity is $O(d^c)$, which is polynomial in depth.
- For $c = 0$ it corresponds to D-search because $f_{0,0} = \infty$.
  The pre-pivot-set S_0 contains all nodes of the whole tree.
- For $c = \infty$ it corresponds to B-search because $f_{\infty,i} = i + 1$, the slowest function with $i < f_i$. All sets $D_i \backslash X_i$ are empty, thus $S_{i+1} = B_i \backslash \{s_i\}$.

The parameter c is a means to express how much storage one is willing to invest into completeness. Between the two extremes "none" ($c = 0$, D-search) and "unlimited" ($c = \infty$, B-search) we now have available an almost arbitrary gradation of algorithms in-between, each of them with space complexity polynomial in depth and time complexity linear in size (since the algorithm is non-repetitive).

Moreover, the parameter c can easily be turned into a parameter of a single implementation for the whole family. It is even possible to adapt the parameter dynamically, i. e., during the traversal of the search tree.

[3] This assumption can be dropped for the implementation [2, Sec. 5.2].

[4] Alternatively, the requirement could be weakened to non-strict monotonicity. While possible in principle, this would make the formal analysis more complex.

3 A Framework for Analysing Tree Traversal Algorithms

Most of the definitions and theorems below refer to their counterparts in a technical report [2, `http://www.pms.ifi.lmu.de/publikationen#PMS-FB-2009-7`], which works out the formal framework in full detail.

An uninformed search algorithm cannot anticipate which parts of the search space contain or don't contain solutions. In order to be able to find *all* solutions it has to visit all nodes in the search tree, just like a traversal algorithm. Therefore our framework formalises *traversal* algorithms, gaining the advantage that it does not need to consider whether or not a node is a solution.

Let b be an upper bound for the out degree of trees under consideration. Let $\Sigma = \{0, \ldots, b-1\}$ and Σ^* the set of all words over Σ.

Definition 1 (Traversability [2, Def. 2.1.1]**).**
A set $\Omega \subseteq \Sigma^*$ is called *traversable*, iff $u \in \Omega$ holds for all $uv \in \Omega$. The set of all traversable $\Omega \subseteq \Sigma^*$ is denoted by Trav_Σ.

Definition 2 (Tree [2, Def. 2.1.2]**).**
Let $E := \{(w, wi) \mid w \in \Sigma^*, i \in \Sigma\}$. Then (Σ^*, E) is a complete infinite tree with out degree b. Any tree with maximum out degree b can be obtained by choosing a traversable $\Omega \subseteq \Sigma^*$ and restricting the edge set E to Ω. The resulting tree $(\Omega, E|_\Omega) = (\Omega, \{(w, w') \in E \mid w, w' \in \Omega\})$ is simply written Ω from now on.

Notation 3.
$$\begin{array}{lcl} \Omega_k & := & \{w \in \Omega \mid |w| = k\} \;=\; \Omega \cap \Sigma^k \\ \Omega_N & := & \{w \in \Omega \mid |w| \in N\} = \bigcup_{i \in N} \Omega_i \text{ for } N \subseteq \mathbb{N} \\ \Omega_{\geq k} & := & \Omega_{\{\geq k\}} \qquad \text{where} \qquad \{\geq k\} := \{i \in \mathbb{N} \mid i \geq k\} \end{array}$$

Notation 4. In the following we often talk about an α with $\alpha \preccurlyeq \omega$. Such an ordinal number α may be considered just the set $\mathbb{N}$ in the infinite case ($\alpha = \omega$) and some set of the form $\{0, 1, \ldots, n\}$ or $\emptyset$ in the finite case ($\alpha \prec \omega$), each together with the common (well-)ordering on natural numbers.[5]

The next three definitions model tree traversals at different levels of abstraction, each building on the former. A *traversal-sequence* assigns to each number corresponding to a "time point" the node of the tree visited at that time point. A *traversal-run* enriches a traversal-sequence by associating with each time point a subset of the nodes of the tree. This subset represents the nodes kept in memory at this point for later processing (one can reasonably assume this to be almost the only use of memory by a tree traversing algorithm). A *traversal-algorithm* can then be specified by assigning a traversal-run to each tree $\Omega \in \mathit{Trav}_\Sigma$.

Definition 5 (Traversal-sequence [2, Def. 2.2.3]**).**
Let $a : \alpha \to \Omega$ be a finite or infinite sequence of nodes in Ω. Then a is called *traversal-sequence* iff for each w occurring in a also its parent occurs in a and the first occurrence of its parent is located before the first occurrence of w.

[5] The notation was chosen for two reasons: (1) it calls attention to the "succession" of the numbers and (2) it covers finite and infinite cases in a uniform way.

Definition 6 (Traversal-run[6] [2, Def. 2.2.4]).
A *traversal-run* is a sequence $A : \alpha \to \Omega \times \mathcal{P}(\Omega)$ of pairs (node, node set) with:
1. The first node set contains exactly the root of the tree Ω.
2. For each pair the node is a member of the corresponding node set.

For all successive pairs (node_n, set_n) and (node_{n+1}, set_{n+1})
3. the children of node_n are included in set_{n+1} (modelling expansion of node_n).
4. from set_n to set_{n+1} an arbitrary number of nodes may be dropped.
5. for any node w in set_{n+1}, either it is member of set_n or its parent is node_n.

A traversal-run *induces* the traversal-sequence obtained by omitting the sets.

Definition 7 (Traversal-algorithm[6] [2, Def. 2.2.4]).
A *traversal-algorithm* is a family $(A_\Omega)_{\Omega \in Trav_\Sigma}$ of traversal-runs. In other words, the algorithm assigns to each tree $\Omega \in Trav_\Sigma$ a traversal-run over this tree.

As an example of a traversal-run, consider a typical queue-based implementation of B-search. For each time point n and pair (node_n, set_n) the set_n consists of all nodes in the queue at this time point (they are the nodes needed for future expansion) and node_n is the first in the queue (the node to be expanded in the step from n to $n+1$). This node is not a member of set_{n+1} because B-search removes the expanded node from the queue when inserting its children.

Modelling a stack-based implementation of iterative-deepening, each set_n consists of all nodes in the stack at this time point and node_n is the top of stack. This node is not a member of set_{n+1} unless it is the root ε, which needs to be kept in the stack as the bottom element for later re-expansion when starting another iteration.

Definition 8 (Completeness [2, Def. 2.2.9]).
- A traversal-sequence $a : \alpha \to \Omega$ is called *complete*, iff it is surjective.
- A traversal-run is *complete*, iff its induced traversal-sequence is.
- A traversal-algorithm is *complete*, iff its runs A_Ω are for all $\Omega \in Trav_\Sigma$.

Definition 9 (Weak completeness [2, Def. 2.2.14]).
A traversal-sequence $a : \alpha \to \Omega$ is called *weakly complete*, iff $|a[\alpha]| = |\Omega|$.
The definition of weak completeness for traversal-runs and traversal-algorithms is analogous to the definition of their completeness.

Obviously completeness implies weak completeness, but conversely only in the finite case. In the infinite case weak completeness intuitively means that traversal does not artificially stop when there are still unexpanded nodes in the search space. Weak completeness follows from some simple criteria [2, Lem. 2.2.16 & 3.3.3], which are easier to test than the condition defining (full) completeness.

When analysing tree traversals, it is in most cases sufficient to know whether some node is visited earlier than another one, comparing only their first visits. Time points of revisits, though represented in traversal-sequences, are usually irrelevant. This observation leads to the final abstraction level in our framework.

[6] The definition does not depend on any computability requirements. They are not needed for the framework and would not simplify anything either.

Definition 10 (Representing ordering [2, Def. 2.2.12]**).**
A partial ordering $\lhd$ on Ω is called *representing ordering* of a traversal-sequence a, iff all nodes occurring in a are

1. ordered by their first occurrence.
2. smaller than any node that does not occur in a, but is $\lhd$-comparable to the root ε of Ω.[7]

If the representing ordering of a is unique, it is referred to as $\lhd_a$.

Definition 11 (Traversal-ordering[8] [2, Def. 2.3.2]**).**
A partial ordering $\lhd$ on traversable $\Omega \subseteq \Sigma^*$ is called *traversal-ordering*, iff

1. it is compatible with the tree structure of Ω, i. e., for all $uvw \in \Omega$
   $u \not\rhd uvw$ and $u \trianglelefteq uvw \;\Rightarrow\; u \trianglelefteq uv \trianglelefteq uvw$ hold.
2. all nodes $\lhd$-comparable to the root ε are totally ordered by $\lhd$.[7]
3. no node not $\lhd$-comparable to ε is smaller than any node $\lhd$-comparable to ε.[7]

At least one of the representing orderings of any traversal-sequence is a traversal-ordering. Those that are not, can be disregarded. [2, Rem. 2.3.3]

Notation 12. For a partial ordering $(\Omega, \lhd)$, sets $M, N \subseteq \Omega$ and $w \in \Omega$ define

$$\begin{aligned} M \lhd N &:\Leftrightarrow \forall m \in M\, \forall n \in N : m \lhd n \\ \lhd\!\!\!\cdot\,(w) &:= \{w' \in \Omega \mid w' \lhd w\} \end{aligned}$$

Definition 13 (Completeness of an ordering [2, Def. 2.3.1]**).**
A partial ordering $\lhd$ on $\Omega \subseteq \Sigma^*$ is called *complete*, iff $(\Omega, \lhd) \cong \alpha$ for some $\alpha \preccurlyeq \omega$.

Theorem 14 (Characterisation of completeness [2, Thm. 2.3.5]**).**
A total ordering $(\Omega, \lhd)$ is complete iff $\exists f : \mathbb{N} \to \mathbb{N}$ with $\Omega_k \lhd \Omega_{\geq f(k)}$.

Theorem 15 (Characterisation of completeness [2, Thm. 2.3.8]**).**
A partial ordering $(\Omega, \lhd)$ is complete, iff it is isomorphic to a finite sum of complete ordinal numbers, where only the last[9] summand may be infinite (i. e., $= \omega$). Equivalently, $(\Omega, \lhd)$ is complete iff $(\Omega, \lhd)$ is isomorphic to a countably infinite sum of finite ordinal numbers.

The main result now is that these criteria essentially characterise also the completeness of traversal-sequences and thus of traversal-algorithms.

Theorem 16 (Equivalence of completeness definitions [2, Thm. 2.3.13]**).**

1. A traversal-sequence is complete in the sense of Definition 8 iff all its (traversable) representing orderings are complete according to Definition 13.
2. A weakly complete traversal-sequence is complete in the sense of Definition 8 iff its representing ordering is complete according to Definition 13.

[7] This requirement is mainly due to technical reasons. It excludes irrelevant but potentially troublesome cases that could otherwise be formally construed.
[8] A total traversal-ordering is a topological ordering of Ω.
[9] Recall that addition is generally not commutative in ordinal number arithmetic.

4 Analysis of Tree Traversal Algorithms

In this section we first analyse some well-known algorithms in order to illustrate that the proofs of their (in)completeness are significantly more concise when based on our framework than the proofs found in the literature. The main part of the section is then devoted to the analysis of D&B-search, which would be hardly possible without the framework.

4.1 Known Algorithms

This subsection demonstrates the expressive and analytic power of our framework and the degree to which it makes (in)completeness proofs more concise. It shows by the example of D-search and of B/A^*-search how Theorem 14 and 15 may be used to prove (in-)completeness.

In both cases we define some total traversal-ordering characterising the desired algorithm. Then we show the (in-)completeness of the traversal-ordering. The algorithm itself can be obtained by means of the induced algorithm of the ordering, which mainly traverses the nodes in order, i. e., starting with the minimum of the ordering and then moving to the next greater node each step.[10]

D-Search. In the representation introduced by Definition 2 each node w of a tree Ω can be considered a word over the alphabet Σ. Seen this way D-search traverses the nodes of Ω in lexicographical order. So we define $\lhd_{depth} := \lhd_{lex}$.

Obviously D-search is incomplete on most infinite trees. This can be shown easily even without the framework by some counterexample. But if you want to explain why D-search is incomplete, things become more complicated. Probably one would say something like "D-Search will never return from the first infinite branch". This statement is based on the reader's common understanding of an algorithm's behaviour. How to make it more precise beyond intuition, however, is not obvious. Our framework allows to formulate the statement precisely.

Let Ω contain an infinite number of nodes and thus an infinite branch (lemma by König). Let $t \in \Sigma^\omega$ be the lexicographically first infinite branch in Ω. The set of prefixes $T \subseteq \Omega$ of t is just the set of nodes on t. Define $S := \{w \in \Omega \mid T \rhd_{depth} w\}$ and $U := \{w \in \Omega \mid T \lhd_{depth} w\}$. Intuitively, S consists of all nodes "to the left" of the first infinite branch T and U of all nodes "to the right" of T. If $U \neq \emptyset$ then $\lhd_{depth}$ is incomplete on Ω:
Possibility 1: $\exists w \in \Omega \forall n \in \mathbb{N} \exists w' \in \Omega_n : w' \lhd_{depth} w$ since $\exists w \in \Omega : T \lhd_{depth} w$ and $T \cap \Omega_n \neq \emptyset$ for all $n \in \mathbb{N}$. Consequently $\exists k \in \mathbb{N} \forall n \in \mathbb{N} : \Omega_k \not\lhd_{depth} \Omega_n$ holds (set $k = |w|$). Incompleteness follows by Theorem 14.
Possibility 2: $S \lhd_{depth} T \lhd_{depth} U$ holds. The smallest β that fulfils this condition[11] is $\beta = \underset{\cong k \prec \omega}{|S|} + \underset{\cong \omega}{|T|} + \underset{\cong \alpha \succ 0}{|U|} \succ \omega$. Incomplete[12] by Theorem 15.

[10] "minimum" and "next greater node" are well-defined, see [2, Def. 2.3.9, Proof].

[11] An ordinal number β *fulfils* the condition $S \lhd_{depth} T \lhd_{depth} U$, if there is an isomorphic well-ordering $(\Omega, \lhd)$ that fulfils the condition.

[12] $\lhd_{lex}$ is generally not a well-ordering. Particularly $(\Omega, \lhd_{depth}) \ncong \beta$ in general. But even if $(\Omega, \lhd_{depth})$ were well-ordered it would still be incomplete as shown above.

B-Search and A^*-Search. The informed A^*-search uses an optimistic cost estimation function[13] $F(w) = G(w) + H(w)$ to prioritise more promising nodes. With $G(w) = |w|$ and $H(w) = 1$ we obtain B-search as special-case.

A^*-search prefers nodes with smaller estimated costs and takes the lexicographically smaller one first if the estimated costs are equal. Consequently $w \lhd_{A^*} w' \;:\Leftrightarrow\; F(w) < F(w')$ or $F(w) = F(w') \wedge w \lhd_{lex} w'$ defines the order in which A^*-search traverses the nodes.

To prove the completeness of A^*-search we apply Theorem 14 using the function $k \mapsto max_<(F[\Omega_k]) + 1$. We must show that $\Omega_k \lhd_{A^*} \Omega_{(max_<(F[\Omega_k])+1)}$, meaning $w \lhd_{A^*} w'$ for $w \in \Omega_k$ and $w' \in \Omega_{(max_<(F[\Omega_k])+1)}$. This is true because $F(w') \geq G(w') \geq |w'| = max_<(F[\Omega_k]) + 1 > max_<(F[\Omega_k]) \geq F(w)$.

In the special case of B-search completeness can be proved even faster using Theorem 15. One only has to convince oneself that the following is true:
$\lhd_{breadth} \cong |\Omega_0| + |\Omega_1| + |\Omega_2| + \ldots \cong \sum_{i=0}^{+\infty} |\Omega_i| \preccurlyeq \omega$

Compared to the proofs in [10,11] the argumentation above is extremely short and precise. Due to its formal character it doesn't even need a deeper understanding of the concrete procedure of A^*-search. At this point we benefit from the abstract level of our analytic framework.

4.2 D&B-Search

The analysis of D&B-search is based on its traversal-ordering $\lhd_{d\&b}$ too. First we give a constructive definition of $\lhd_{d\&b}$ which corresponds directly to the description of D&B-search in Section 2. Second an alternative axiomatic definition of $\lhd_{d\&b}$ is presented. We show the equivalence of the two definitions and then alternate between them when proving completeness and space complexity.

Definition 17 (pivot-nodes & pre/inter/post-pivot-sets [2, Def. 4.1.1]**).**

$$i_{max} := \begin{cases} -1 & \text{if } \Omega = \emptyset \\ max(\{i \mid \Omega_{f_i} \neq \emptyset\}) & \text{if } |\Omega| < \infty \\ \infty & \text{if } |\Omega| = \infty \end{cases} \qquad \begin{array}{l} s_i := min_{lex}(\Omega_{f_i}) \text{ for } 0 \leq i \leq i_{max} \\ D_i := \lhd_{lex}(s_{i+1}) \\ B_i := \Omega_i \end{array}$$

$$\begin{array}{ll} S_0 := \lhd_{lex}(s_0) & X_0 := S_0 \cup \{s_0\} \\ S_{i+1} := (D_i \cup B_i) \setminus X_i & X_{i+1} := X_i \cup S_{i+1} \cup \{s_{i+1}\} \\ R := \Omega \setminus X_{i_{max}} & X_{i_{max}} := \bigcup_{j=0}^{i_{max}} X_j \end{array}$$

The pivot-nodes and pre/inter/post-pivot-sets should look familiar from page 4. By means of these nodes and sets the next definition constructs $\lhd_{d\&b}$.

[13] G denotes the cost incurred so far on the path to w, and H denotes the optimistically estimated cost remaining for the path from w to a goal.

Definition 18 (D&B-ordering, constructive [2, Def. 4.3.1]**).**

1. $S_0 \lhd_{d\&b} s_0 \lhd_{d\&b} S_1 \lhd_{d\&b} s_1 \lhd_{d\&b} \cdots \lhd_{d\&b} S_{i_{max}} \lhd_{d\&b} s_{i_{max}} \lhd_{d\&b} R$
2. $\forall w, w' \in S_0 : \quad w \lhd_{d\&b} w' \Leftrightarrow w \lhd_{lex} w'$
3. $\forall w, w' \in S_{i+1} \cap \Omega_i : \quad w \lhd_{d\&b} w' \Leftrightarrow w \lhd_{lex} w'$
4. $\forall w, w' \in S_{i+1} \backslash \Omega_i : \quad w \lhd_{d\&b} w' \Leftrightarrow w \lhd_{lex} w'$
5. $\forall w, w' \in R \cap \Omega_{i_{max}} : \quad w \lhd_{d\&b} w' \Leftrightarrow w \lhd_{lex} w'$
6. $\forall w, w' \in R \backslash \Omega_{i_{max}} : \quad w \lhd_{d\&b} w' \Leftrightarrow w \lhd_{lex} w'$

Condition 1 already appeared on page 5. It defines the order between the pivot-nodes and sets and can be read as *D&B-search expands all members of the pre-pivot-set S_0 before expanding the pivot-node s_0, and it expands s_0 before expanding all members of the inter-pivot-set S_1*, and so on.

The rest affects the inner order of the sets. All equivalences can be read as *D&B-search expands w before w' iff depth-first search would.* Condition 2 matches exactly the description of D&B-search on page 5. Conditions 3 to 6 are a little bit less restrictive than the informal description. There D-search always had to finish work on some inter- or post-pivot-set before B-search could start, the nodes in the subset $S_{i+1} \backslash \Omega_i$ had to be expanded before the nodes in the subset $S_{i+1} \cap \Omega_i$. Here the two may interleave their work on such a set, the order between the two subsets is not restricted by the conditions above.

Recall the original view of D&B-search as introduced at the very beginning of Section 2. It was the view of alternating D-search and B-search, controlling their rotation by a sequence $f_0, f_1, f_2, \ldots$ of depth bounds. For the axiomatic definition of $\lhd_{d\&b}$ we reuse this view.

Definition 19 (D&B-ordering, axiomatic [2, Sec. 4.3 (D&B)]**).**

$$\begin{aligned} &\text{(Ax1)} \quad \Omega_k \lhd_{d\&b} \Omega_{f_{k+1}} \\ &\text{(Ax2)} \quad \forall w, w' \in \Omega_k : w \lhd_{d\&b} w' \Leftrightarrow w \lhd_{lex} w' \\ &\text{(Ax3)} \quad \forall w \in \Omega_k : \underbrace{\lhd_{lex}(w) \lhd_{d\&b} w}_{\text{(Ax3a)}} \vee \underbrace{\exists w' \in \Omega_{f_k} : w' \lhd_{d\&b} w}_{\text{(Ax3b)}} \end{aligned} \qquad \text{(D\&B)}$$

(Ax1) signifies that none of the nodes in $\Omega_{f_{k+1}}$ is expanded before all nodes of Ω_k have been expanded. Therefore it limits the depth of the depth-first-traversal.

(Ax3b) disjunction concerns the breadth-first-traversal. It means that a node may only be expanded if some node at sufficient depth has been expanded (by depth-first-traversal) before. This implies a depth limit for breadth-first-traversal because (Ax3) requires (Ax3b) or (Ax3a) to hold for each node.

(Ax2) and (Ax3a) are less interesting. (Ax3a) says that a node may be expanded if all lexicographically smaller nodes have been expanded before. This is the specification of D-search. If at some time (Ax3a) becomes false because of (Ax1), i. e., (Ax3b) is true, (Ax2) enforces exactly B-search because it requires every level to be traversed in lexicographical order.

We have to show that the two definitions of $\lhd_{d\&b}$ are equivalent. Though our framework is a great help when formulating the arguments, the proof is still more extensive than can be presented within the space limitations of this paper. Without the framework it would be quite impossible. The results are as follows.

Theorem 20 (D&B-ordering, constructive ⇒ axiomatic [2, Thm. 4.3.8]**).** If $\lhd$ satisfies Definition 17 and 18 then $\lhd$ is a model of (D&B).

Theorem 21 (D&B-ordering, axiomatic ⇒ constructive [2, Thm. 4.3.9]**).** If $\lhd$ is a model of (D&B) then it satisfies Definition 17 and 18.

Thus, the two definitions are equivalent. But what does this help? In particular, why do we need the axiomatic definition? We will see one reason[14] in the next theorem. Moreover the theorem emphasises again the power of our framework for completeness proofs as the proof uses one of its main results.

Theorem 22 (D&B-ordering, completeness [2, Thm. 4.3.10]**).** If $\lhd$ satisfies Definition 17 and 18 or is a model of (D&B), then $\lhd$ is complete.

Proof. Follows immediately from (Ax1) and Theorems 14 and 20. □

Finally, we are interested in the space complexity of D&B-search. In this context we find the constructive definition to be very helpful. Let us start with the general result for any function f with $i < f_i < f_{i+1}$. This result is independent from the family defined in Section 2.

Theorem 23 (D&B-search, general space complexity [2, Cor. 4.3.15]**).** The space complexity $M(d)$ at depth $d \in \mathbb{N}$ of the algorithm induced by $\lhd_{d\&b}$ is

$$M(d) \leq \begin{cases} b \cdot (d+1) & \text{if } d < f_0 \\ b \cdot \left(d+1+b^i\right) & \text{else} \end{cases} \qquad \text{where } i := \underset{j}{argmax} \{f_j \mid f_j \leq d\}.$$

But of course we are most interested in the space complexity of the family defined in Section 2. We obtain their complexity from Theorem 23 by specialising f_i to the corresponding functions:

Theorem 24 (D&B family $c = 0$, linear space complexity [2, Thm. 4.3.17]**).** Let $c = 0$, $f_i = f_{0,i} = \lfloor b^{\frac{i}{0}} \rfloor + i = \infty$ and $\lhd_0$ be the corresponding ordering. The space complexity of the induced algorithm is $M_0(d) \leq b \cdot (d+1)$.

Theorem 25 (D&B family, polynomial space complexity [2, Thm. 4.3.16]**).** Let $1 \leq c < \infty$, $f_i = f_{c,i} = \lfloor b^{\frac{i}{c}} \rfloor + i$ and $\lhd_c$ be the corresponding ordering. The space complexity of the induced algorithm is $M_c(d) \leq b \cdot (d+1+d^c)$.

5 Conclusion

In this paper we have presented D&B-search, a new uninformed search method based on integrating depth-first and breadth-first search into one. We have shown that the ratio of depth-first to breadth-first search can be balanced by a parameter, thus defining a family of search methods with depth-first and breadth-first search as its borderline cases.

[14] The second reason is that the axiomatic definitions provide the invariants for our implementation [2, Sec. 5.1].

We have also introduced a formal framework for analysing informed or uninformed search methods, which is based on partial orderings and uniformly covers finite and infinite search trees. We have illustrated its analytic power by giving very concise, yet formally precise proofs for well-known (in)completeness results on depth-first search, breadth-first search, and A^*-search.

Finally, we have analysed D&B-search using the formal framework. In the borderline cases the results are the known ones for depth-first and breadth-first search. In all non-borderline cases D&B-search is complete (exhaustive), it is non-repetitive and thus its time complexity is linear in size, and its space complexity is polynomial in depth. The polynomial is of degree c for the very parameter defining the D&B family, which therefore allows to control the amount of storage to be spent for the sake of completeness.

It should be noted that D&B-search is intrinsically better than running depth-first and breadth-first search in parallel, be it by round robin scheduling or more advanced time-sharing techniques or physically parallel on different processors. With all of these approaches the space complexity is exponential in depth for the process running breadth-first search. In contrast to that, D&B-search has space complexity polynomial in depth. It is this property that made necessary the somewhat involved form of integrating depth-first with breadth-first search.

In this paper we have not addressed implementation issues. The technical report [2] on which the paper is based also presents two implementation approaches to the level of detail of pseudo code showing that the required data structures are essentially a doubly-linked list of doubly-linked lists. Coding this pseudo code in a real programming language is rather straightforward.

We are about to start work on a prototype implementation of one of these approaches and plan to use it for empirical comparisons with other uninformed search methods. We intend to focus especially on logic programming applications using backward reasoning approaches without and with memoization [15,13].

At the conceptual level, we plan to follow-up the observation that the form of integrating depth-first with breadth-first search results in a kind of built-in adaptivity as explained in Section 2. The predominant behaviour of D&B-search corresponds to depth-first search if the search tree is finite and to breadth-first search if the search tree is infinite. This effect can be maintained if depth-first search is integrated with other complete search methods in the same way.

Its space complexity being exponential in depth, breadth-first search, although theoretically complete on infinite trees, cannot advance to very deep levels in practice. Iterative-deepening usually does better and is also complete. However, as pointed out in Section 1, iterative-deepening deteriorates time complexity in those cases in which depth-first search *is* complete. It would therefore be alluring if there was a possibility to use depth-first search whenever it is complete and iterative-deepening only when needed to ensure completeness. Alas, these conditions are not decidable as they stand.

But we can come very close to such a combination by transferring the principle of integration used for D&B-search to a combination of depth-first search

and iterative-deepening. The result, called D&I-search, behaves predominantly like depth-first search if the search tree is finite and like iterative-deepening if the search tree is infinite.

Technically, this can be achieved by the same depth bounds f_i as with D&B-search. D&I-search even has the same representing ordering as D&B-search, so its completeness is just a corollary. In order to make sure that iterative-deepening does not expand any nodes that have already been expanded by depth-first search, iterative-deepening's algorithm needs to be slightly modified and the underlying data structure becomes slightly more complicated. This optimisation even results in the converse effect: with D&I-search, iterative-deepening can to some extent also prune the search space of depth-first search.

We plan to investigate D&I-search and also, using the same principle, other promising combinations of search methods.

Acknowledgements. We are grateful to Tim Furche, who read a draft of this paper and gave many valuable hints for its improvement. We thank all of our colleagues in the group for stimulating discussions about the work presented here.

References

1. R. Barták. Incomplete depth-first search techniques: A short survey. In *Proceedings of 6th Workshop on Constraint Programming for Decision and Control (CPDC2004)*, pages 7–14, 2004.
2. S. Brodt. Tree-search, partial orderings, and a new family of uninformed algorithms. Research report PMS-FB-2009-7, Institute for Informatics, University of Munich, Oettingenstraße 67, D-80538 München, Germany, 2009. `http://www.pms.ifi.lmu.de/publikationen#PMS-FB-2009-7`.
3. F. Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. *Data and Knowledge Engineering*, 5(4):289–312, 1990.
4. S. Debray and R. Ramakrishnan. Abstract interpretation of logic programs using magic transformations. *Journal of Logic Programming*, 18:149–176, 1994.
5. M. L. Ginsberg and W. D. Harvey. Iterative broadening. In *Proc. Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 216–220, 1990.
6. J.-M. Kerisit. A relational approach to logic programming: the extended Alexander method. *Theoretical Computer Science*, 69:55–68, 1989.
7. D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Publishing Co., Reading, MA, USA, 3rd edition, 1997.
8. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
9. J. W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, Heidelberg, New York, Tokyo, 2nd edition, 1987.
10. N. J. Nilsson. *Principles of Artificial Intelligence.* Springer, Berlin, Heidelberg, New York, Tokyo, 1982.
11. J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Co., Reading, MA, USA, 1984.
12. W. Ruml. Heuristic search in bounded-depth trees: Best-leaf-first search. Technical report, Harvard University, 2002.

13. Y.-D. Shen, L.-Y. Yuan, and J.-H. You. SLT-resolution for the well-founded semantics. *Journal of Automated Reasoning*, 28:53–97, 2002.
14. M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.
15. H. Tamaki and T. Sato. OLDT resolution with tablulation. In *International Conference on Logic Programming*, pages 84–98, 1986.
16. P. H. Winston. *Artificial Intelligence*. Addison-Wesley Publishing Co., Reading, MA, USA, 3rd edition, 1992.