Proceedings of the 4th WSEAS Int. Conf. on Information Security, Communications and Computers, Tenerife, Spain, December 16-18, 2005 (pp374-378)

Seeded search for joins in BCPOs

Richard Elling Moe
Department of information science and media studies
University of Bergen

ABSTRACT
First we devise an algorithm for finding joins in BCPO's given a set of upper bounds as seeds for the search. Secondly, simple tests are performed to assess the possibility of an increase in efficiency as a result of having the seeds as a starting point. Finally, we briefly consider join-computation by first finding some arbitrary upper bounds and subsequently use them as seed to the algorithm described above.

KEY WORDS
BCPOs, Join computation, Seeded algorithms

1 Introduction

There is a widespread use of ordering relations in computer- and information sciences, both in theory and applications. In addition comes a rich selection of applications owing to the great number of other disciplines that make use of such structures. Accordingly, the computer representation and processing of ordering relations has received much attention [1, 2, 7, 8, 9, 12, 13, 14].

Often, the approach to ordering relations is intuitive and informal, with no detailed account of the kind of structure in question. Take for instance the wealth of 'hierarchies' found in the literature which lack a precise definition. When specified, we find that they sometimes amount to trees [10] or even total orders [14], but typically they describe something between preorders [12] and lattices [11].

We shall concentrate on the structure used in [2, 3, 4, 8]: The *bounded complete partial order (BCPO)* is more restricted than pre-orders, but not quite a lattice. The operations of prime importance includes *Join* and *Meet*. These are almost entirely dual operations so we will discuss joins only.

Specifically we shall explore the situation in which the operations can be *seeded*. That is, when extra information is available and might be exploited. In the case of the join, such seeds could be in the form of known upper bounds. We focus our attention on the use of seeds and do not consider *how* such information may have become available, whether it is collected from external sources, the result of guessing, heuristics or some other process.

The different kinds of join-algorithms are sensitive to the BCPO in question. Anyhow, it seems possible that the use of seeds could better the performance. We shall try to shed light on this question.

2 Some basics

We assume the reader is familiar with the basic notions concerning partially ordered sets[1] and the standard terms that accompany them and only summarise some notation and terminology.

We write $\langle A, \leq \rangle$ to express that the set A is partially ordered by the relation $\leq \subseteq A \times A$. As usual $x < y$ is shorthand for $x \leq y$ *and* $x \neq y$. We let $Min_{\leq}(X)$ and $Max_{\leq}(X)$ denote the sets of minimal and maximal elements of a set $X \subseteq A$. If X has a unique minimal element we refer to it as the *least* element, and denote it by $min_{\leq}(X)$. Similarly, when a *greatest* element of X exists it is denoted by $max_{\leq}(X)$. X^u denotes $\{a \mid \in A, x \leq a \text{ for all } x \in X\}$, i.e. the set of upper bounds for X.

We say that a' is an *upper cover* for a if $a < a'$ and there is no $a'' \in A$ such that $a < a'' < a'$. Likewise, a is a lower cover of a'.

Two elements a and a' are said to be *comparable* iff either $a \leq a'$ or $a' \leq a$. For any $a \in A$ we define the *down-set* of a, to be the set $\{x \mid x \in A, x \leq a\}$ and denote it by $\mathscr{D}(a)$. Similarly the *up-set* of a is defined by $\mathscr{U}(a) = \{x \mid x \in A, a \leq x\}$.

A subset S of A is said to be *consistent* if every finite subset of S has an upper bound in A. We shall be concerned with cases where A is finite and will then use the term consistent about any subset of A that has an upper bound. We adopt the convention of letting $\sqcup S$ denote the *least* upper bound of S (provided, of course, it exists). We treat $\sqcup$ as a partial function with signature $\mathscr{P}(A) \rightarrow A$ and refer to it as the *join*-function.

For subsets X and Y of A we define the *segment bounded by X and Y*, denoted by $Seg_{\leq}(X,Y)$, to be the set $\{a \mid a' \leq a \leq a'' \text{ for some } a' \in X \text{ and } a'' \in Y\}$. We refer to X and Y as the lower and upper *boundaries* of the segment, alternatively the *floor* and *ceiling*. An element $a \in Seg_{\leq}(X,Y)$ is a *hub* in the segment iff it is both a lower bound for Y and an upper bound for X. Note that a segment does not necessarily have a hub.

A *covering-sequence* is of the form $a_0a_1a_3\ldots$ such that every a_{i+1} is an upper cover of a_i. The functions $first(\sigma)$ and $last(\sigma)$ denotes the first and last elements of

[1]Otherwise, see for instance Grimaldi [6] for a textbook introduction, or Davey and Priestley [5] for a comprehensive exposition.

the sequence σ. The relation $a \vec{\in} \sigma$ holds iff the element a occurs in the sequence σ. The empty sequence is denoted by ε, while $a \dashv \sigma$ denotes the sequence resulting from appending a in front of the sequence σ. $Seq_{\leq}(X,Y)$ denotes the set of covering sequences σ for which $first(\sigma) \in X$ and $last(\sigma) \in Y$. Obviously, $a \in Seg_{\leq}(X,Y)$ iff $a \vec{\in} \sigma$ for some $\sigma \in Seq_{\leq}(X,Y)$

We adopt Carpenters' definition [2] of BCPOs.

Definition 2.1 *A* bounded complete partial order *(BCPO) is a partial order $\langle A, \leq \rangle$ in which every consistent subset of A has a* least *upper bound. In particular, the empty set is regarded as consistent, having a bottom element as its least upper bound.*

Compare the BCPOs with the so called *consistently complete* CPOs [5] and you will find that they amount to the same thing.

Following Carpenter, we restrict our attention to *finite* BCPO's, and refer to $\leq$ as the *subsumption* relation. I.e. if $x \leq y$ we say that x subsumes y.

3 Computing joins from upper bounds

Suppose a set X' of upper bounds for X is already known, then the search for $\sqcup X$ can be confined in two obvious ways.

First, $\sqcup X$ must be contained in the intersection of the down-sets of all $x \in X'$. This is of course so since $\sqcup X$ subsumes all upper bounds for X. Clearly, it will suffice to consider the down-sets for only the *minimal* elements of X', since an element subsumes all members of X iff it subsumes the minimal elements of X.

Secondly, $\sqcup X$ must lie in the segment bounded by X and X', since it is subsumed by all elements in X and subsumes all upper bounds for X. Clearly, the segment will still contain $\sqcup X$ if these boundaries are trimmed down to $Max_{\leq}(X)$ and $Min_{\leq}(X')$ respectively.

Note that within the limits given by these two restrictions, there may still be elements that are not upper bounds for X. The point is that the search for $\sqcup X$ can be confined to a smaller space. So, having reduced the search-space, $\sqcup X$ can be found by extracting the remaining upper bounds for X, and selecting the least among them. Hence, we arrive at the following:

Lemma 3.1 *Let $\langle A, \leq \rangle$ be a BCPO and suppose $X \subseteq A$ is consistent. Then, for any $X' \subseteq X^u$:*

$$\sqcup X = min_{\leq}(X^u \cap Seg_{\leq}(Max_{\leq}(X), Min_{\leq}(X')) \cap \bigcap_{x \in Min_{\leq}(X')} \mathscr{D}(x))$$

Our goal is to refine this observation into an algorithm for computing $\sqcup X$ given some arbitrary set X' of upper bounds for X. Let us focus our attention on the segment $Seg_{\leq}(Max_{\leq}(X), Min_{\leq}(X'))$, and identify the joins within. The following observations tell us what to look for in a segment.

Lemma 3.2 *Let $\langle A, \leq \rangle$ be a BCPO. Let X and Y be non-empty subsets of A. If $Seg_{\leq}(X,Y)$ has hubs at all, then $\sqcup X$ is the least hub.*

Proof Assume that $Seg_{\leq}(X,Y)$ has a hub h. By definition, h is an upper bound for the lower boundary X, that is, X is consistent so $\sqcup X$ exists, and subsumes h. Being a hub h also subsumes all elements of Y, then so does $\sqcup X$. Hence, $\sqcup X$ is a member of the segment, and a hub. Since every hub is an upper bound for X, $\sqcup X$ is the least hub in $Seg_{\leq}(X,Y)$.
■

Lemma 3.3 *Let $\langle A, \leq \rangle$ be a BCPO. Let X and Y be non-empty subsets of A. If $Y \subseteq X^u$ then $Seg_{\leq}(X,Y)$ has a hub.*

Proof Assume $Y \subseteq X^u$. I.e. all elements of Y are upper bounds for X. Then $\sqcup X$ exists and subsumes them all. Hence $\sqcup X$ is also a lower bound of Y and thus, a hub.
■

This gives us the following clue to finding joins:

Corollary 3.1 *Let $\langle A, \leq \rangle$ be a BCPO. Let X and Y be non-empty subsets of A. If $Y \subseteq X^u$ then $\sqcup X = min_{\leq}(\{a \mid a \text{ is a hub in } Seg_{\leq}(Max_{\leq}(X), Min_{\leq}(Y))\})$.*

Proof This follows directly from Lemmas 3.3 and 3.2 since the segment between the boundaries X and Y obviously is the same as the segment bounded by $Max_{\leq}(X)$ and $Min_{\leq}(Y)$ respectively.
■

Consequently, the hunt for a join consists in computing the segment, finding the hubs and pick the least among them.

We have already noted that the members of $Seg_{\leq}(X,Y)$ are precisely those that occur in the covering-sequences in $Seq_{\leq}(X,Y)$. Working with $Seq_{\leq}(X,Y)$ in stead of the segment directly has advantages since the covering-sequences can readily be used to find the hubs as well. The idea is to construct from the sequences, a relation $lb \subseteq Seg_{\leq}(X,Y) \times \mathscr{P}(Y)$ such that $lb(a, Y')$ holds iff Y' is the greatest subset of Y for which a is a lower bound.

The actual program would naturally have to build a suitable representation of the relation lb but we do not go to this level of detail. Instead we view relations as sets of tuples in the traditional way.

In the following algorithm, LB and S are (updatable) set variables. Upon termination LB will hold the relation lb.

```
LB← $\emptyset$; {Initialize LB}
for each $\sigma \in Seq_{\leq}(X,Y)$ do
  for each $a \vec{\in} \sigma$ do
    LB← LB$\cup\{(a,\emptyset)\}$
  end for
end for{Initialization done}
for each $\sigma \in Seq_{\leq}(X,Y)$ do
  for each $a \vec{\in} \sigma$ do
    Locate $(a, S)$ in LB;
    S← S$\cup\{last(\sigma)\}$;
  end for
end for
```

There is a dual algorithm for computing a relation $ub \subseteq Seq_{\leq}(X,Y) \times \mathscr{P}(X)$ such that $ub(a,X')$ holds iff X' is the greatest subset of X for which a is an upper bound.

Now we can apply corollary 3.1 to specify an algorithm for computing $\sqcup X$ from a set Y of seeds, i.e. arbitrary upper bounds for X: Given the relations *lb* and *rb* corresponding to $Seq_{\leq}(Max_{\leq}(X), Min_{\leq}(Y))$, a hub is an element h for which both $lb(h, Min_{\leq}(Y))$ and $ub(h, Max_{\leq}(X))$ hold. Provided that these relations are represented in a suitable way, for instance by means of association lists, all the hubs in the segment can easily be extracted. Then, picking the least among them will produce $\sqcup X$. (A special case will have to be made for the instance $\sqcup\emptyset = \bot$ since our procedure requires X to be non-empty.)

It remains to determine how $Seq_{\leq}(X,Y)$ is to be computed. A straightforward approach is to generate the covering-sequences backwards, starting with the elements of the upper boundary Y and expanding each such sequence with a lower cover of the last added element. Should there be several such lower covers, the sequence is duplicated. Each sequence is expanded downwards until it reaches the lower boundary X[2], or an attempt is made to extend it beyond $\bot$ in which case the generated sequence has bypassed the lower boundary and should not be included in the result. In the following pseudo-code for for computing $Seq_{\leq}(X,Y)$ SEQ and CHECK are set variables. Upon termination SEQ will hold the result.

```
SEQ← $\emptyset$;
CHECK← $\emptyset$;
**for** each $y \in Y$ **do**
    SEQ← SEQ$\cup\{y \dashv \varepsilon\}$
**end for**{Initialization done}
**while** SEQ $\neq$ CHECK **do**
    CHECK← SEQ;
    **for** each $\sigma \in$ SEQ **do**
        **if** $first(\sigma) \notin X$ **then**
            **for** each lower cover $a$ of $first(\sigma)$ **do**
                SEQ← SEQ$\cup\{a \dashv \sigma\}$
            **end for**
        **end if**
    **end for**
**end while**
{Drop sequences that have bypassed $X$}
**for** each $\sigma \in$ SEQ **do**
    **if** $first(\sigma) \notin X$ **then**
        SEQ← SEQ$-\{\sigma\}$
    **end if**
**end for**
```

Because of our confinement to finite BCPO's, termination is guaranteed for this, and the previous, algorithm.

[2]In general, X may contain comparable elements, in which case only the maximal ones would be included in the result of this algorithm. This poses no problems since we will only apply it to $Max_{\leq}(X)$, cf corollary 3.1

4 Assessment and tests

In the following we shall try to assess the value of bringing seeds into the picture, when possible. We start by comparing our algorithm, referred to as SEED, to a straightforward seedless algorithm. Initially we compare by means of (somewhat superficial) judgments of their relative complexities, but lessons from working with join-algorithms tell us that they are likely to be sensitive to the nature of the BCPO in question and we will therefore run some tests in addition.

For comparison we choose the algorithm founded on the following observation concerning BCPOs:

$$\text{If } X \text{ is consistent then } \sqcup X = min_{\leq}(\bigcap_{m \in Max_{\leq}(X)} \mathscr{U}(m))$$

This particular strategy was chosen because of its simplicity, making it easy to implement it faithfully, and for additional reasons that will become clear later.

We refer to this algorithm as the *Maximals Up-set Intersection Minimum*, MUIM: The task of finding the join of a set consists in finding its maximal elements, generate their up-sets, compute the intersection of these up-sets and finally, search the result for the least element.

Every now and then we rely on the computation of least elements and sets of minimal elements, etc. Such algorithms obviously exists and we have left out the details. In our tests they are performed as simple depth-first searches in the space spanned by the covering-relation. In deed, the entire task of computing a join could be seen as a search-process within the BCPO. On this view, our algorithms serve to reduce the job to searches over smaller spaces.

Both algorithms operate by confining the attention to a reduced set of elements: MUIM considers elements of an intersection of up-sets, while SEED looks for hubs in specific segments. An important difference is that SEED handles the segment $Seg_{\leq}(X,Y)$ in terms of the set $Seq_{\leq}(X,Y)$ of covering-sequences, in which elements can be duplicated many times. On the other hand, no duplicates occur in the intersection computed by MUIM. Hence, there is a danger that SEED's handling of sequences and hubs has substantial overhead compared to the workings of MUIM.

In order to test this, SEED and MUIM have been run on the same input, with their performance (in time) measured for comparison. Specifically, the test comprised 10 BCPOs of varying 'shapes' from which a total of 10000 joins was picked randomly to be computed by both algorithms. In each instance, a random seed was generated for the SEED algorithm. The results are in favour of MUIM. The average running-time ratio (SEED/MUIM) being 1.11. So, our tests provide no evidence of performance gained from using seeds in join-computation. However, some comments are in place here.

As mentioned, join-algorithms can be sensitive to the 'shape' of the BCPO. For that reason we have used a variety of shapes in our test-set but we cannot exclude the possibility that particular shapes are missing from our experiment.

Secondly, the seeds we have discussed for the computation of $\sqcup X$ are given sets of upper bounds for X. I.e, X is assumed to be consistent, but normally join-algorithms will also be used to discover inconsistency. Given an inconsistent set X, the SEED algorithm can also be used for this purpose if provided with the seed $\emptyset$ or, alternatively, a seed containing the maximal elements in the up-sets of members of X. However, since we have not specified the origin of seeds, we can only assume whether or not the information they hold is perfect with regard to consistency. A full discussion of this matter should consider different degrees of quality of the given information and corresponding strategies for producing suitable seeds that can detect inconsistencies. However, we disregard this debate and focus only on consistent sets. Technically, we circumvent the problem by using BCPOs with top elements in our test, i.e. they are all lattices.

Finally, a particular join could have several alternative seeds. It seems obvious that their respective quality may vary. A random sample of seeds may involve bad ones and this may have affected our test. This brings us to the question of whether making an effort to find the seeds would pay off.

4.1 Finding some upper bounds

Given Corollary 3.1, it is clear that any algorithm for finding upper bounds will give rise to an algorithm for computing joins. Upper bounds can be accounted for in terms of up-sets:

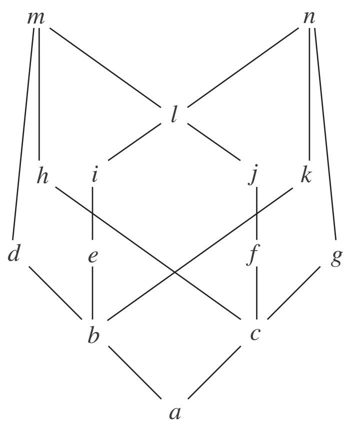
$$b \in X^u \text{ iff } b \in \bigcap_{x\in X} \mathscr{U}(x)$$

Accordingly, the up-sets provide the basis for a very straightforward algorithm for finding upper bounds.

The covering relation induces *layers* in the up-set of an element $x \in A$ in the sense that x itself constitutes the bottom layer and, on any layer, the next layer consists of the upper covers of the elements on the present layer. Now, an algorithm for finding upper bounds for a set X would proceed by simultaneously constructing the up-sets for each of the elements in X by iteration upwards layer by layer. If some elements common to all up-sets is encountered during this process, the algorithm may present these as the result and terminate, free of concern for what might be contained on any remaining layers.

Presumably the reader can easily accept this idea. The question remains, will this algorithm provide good seeds for SEED to compute the join? We refer to the combined algorithm, where seeds are computed first and then fed into SEED, with the acronym PUSS (*Partial Up-Sets Seed*). We shall take a closer look at the performance of PUSS shortly, but let us first consider why this approach is particularly interesting with respect to the MUIM algorithm used above for computing $\sqcup X$.

Whereas MUIM generates entire up-sets for the members of X, PUSS generates partial up-sets: The process continues until some upper bounds have been found, and then terminated. However, upon termination there is no guarantee that the join has been encountered in the process. Consider the figure below where $\sqcup\{b,c\} = l$. The upper bounds m and n lie on a lower layer than l so the sketched algorithm would terminate as soon as it is recognized that m and n are elements of $\mathscr{U}(b) \cap \mathscr{U}(c)$. That is, the process stops before running into l. In this situation, it is clear



that the process can not continue by simply computing the intersection of the partial up-sets since the join may have been left out. So, either must the up-sets be completed, like MUIM does, or the found upper bounds must be used in some other way, like PUSS. We have not considered other options for continuing the process, but we observe that the PUSS algorithm can be seen as an adaptation of MUIM to incorporate the ideas behind SEED. Therefore MUIM is a natural choice for comparison.

The example above also illustrates that the partial up-set algorithm may come up with more than one upper bound (m and n), and that these may be mutually incomparable. When seen in light of lemma 3.1 this is not necessarily a bad thing. The more elements to begin with, for which the down-sets are intersected, the greater are the chances of reducing the search-space. However, this effect does not carry over to the SEED-algorithm since the corresponding intersection is not computed explicitly, the way lemma 3.1 suggests.

Returning to the performance of the PUSS algorithm, it is reasonable to ask whether the computed seeds are better than random ones. We investigate this by performing the same test as earlier, running PUSS on the same sample of joins and compare its running times with those obtained by feeding random seeds to SEED. The running time ratio (SEED/PUSS) is 0.96. Hence, we have no evidence to conclude that the quality of the computed seeds makes it worth while to compute them.

5 Conclusion

We have investigated the possibility of enhancing the computation of joins with side-information, if available. Specifically we have devised an algorithm (SEED) that tries to exploit seeds in the form of sets of known upper bounds. Furthermore, this has been extended to the PUSS algorithm which first computes some seeds which are then passed on to SEED. We have run tests comparing their performance to a seedless algorithm. Although the results provide only indications of the value of using seeds, they are not very promising. The successful use of the SEED algorithm seems to rely on the availability of good seeds.

References

[1] M.F van Bommel, P. Wang, Encoding multiple inheritance hierarchies for lattice operations, *Data and knowledge engineering* 50, 2004

[2] Bob Carpenter, *The Logic of Typed Feature Structures*, Cambridge University Press, 1992

[3] Ann Copestake, Definitions of typed feature structures (appendix), *Natural Language Engineering* 6 (1), Cambridge University Press, 2000

[4] Ann Copestake, *Implementing typed feature structures grammars*, CSLI Publications, 2002

[5] B. A. Davey, H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, 1991

[6] Ralph P. Grimaldi, *Discrete and Combinatorial Mathematics*, Addison Wesley, 2000.

[7] Nicola Guerino, Formal ontology and information systems, In N. Guarino (ed), Formal ontology in information systems, Proceedings of FOIS'98, IOS-press.

[8] Richard Elling Moe, First order typed feature structures, Dr. scient thesis, Department of informatics, University of Bergen, 1996.

[9] Antoni Olivé, Ernest Teniente, Derived types and taxonomic constraints in conceptual modeling, *Information systems* 27, 2002

[10] A. Renear, E. Mylonas, D. Durand, Refining our notion of what text really is: the problem of overlapping hierarchies, In: Nancy Ide and Susan Hockey (eds) *Research in humanities computing*, Oxford University Press, 1996

[11] Gert Smolka, A Feature Logic with Subsorts, In J. Wedekind, C. Rohrer (eds.) *Unification in Grammar*, The MIT press, 1992

[12] G. Smolka, H. Aït-Kaci, Inheritance Hierarchies: Semantics and Unification, *Journal of Symbolic Computation* 7, 1989

[13] John F. Sowa, *Knowledge representation*, Brooks/Cole, 2000

[14] Michael Wooldridge, Intelligent agents, In Gerhard Weiss (ed) *Multiagent systems*, MIT Press 2001