Abduction and Memoing

Tim Menzies * Andrew Taylor †

December 22, 1995

Abstract

Abduction is a inference procedure with wide applicability to knowledge-based processing. Abduction is known to be NP-hard; i.e. over models of since N, runtimes are expected to be $O(k^N)$. Memoing is a technique for storing the results of prior computations, thus decreasing the inferencing time required if some particular inference is required more than once. Memoing is known to decrease the runtimes of certain computations. Here, we experiment with memoing to reduce the computational cost of abduction. These experiments were unsuccessful. We conclude that memoing is not suitable for optimising abduction due to the context-dependent nature of abductive inference.

KEYWORDS: Abduction, memoing, complexity.

1 Introduction

The premises of this paper are that (i) abduction is an important inference procedure; and (ii) techniques for optimising abduction would be applicable to a wide variety of KBS tasks. Memoing is a general optimisation technique which we tried to apply to abduction. We found that memoing is not useful for optimising abduction over theories containing invariants. We spent some time on this unfruitful optimisation attempt and advise others to avoid this approach. We report this negative result since the reasons for this failure highlights a significant difference between deductive and abductive inference. Further, exploring memoing and abduction provides a succinct tutorial on the internals of an abductive inference engine.

Section 2 introduces memoing. Section 3 introduces abduction and Section 5 discusses its computational cost. Section 4 describes the internals of HT4 [11], the abductive inference engine used for our experiments. The memoing experiments are described in Sections 6 & 7. Portions of this work have appeared previously [9, 12, 10].

2 Memoing

Memoing was first proposed by Michie in 1968 [14]. Warren [20] argues that memoing has been subsequently rediscovered and renamed in several different guises. Conceptually, it is simple process. Given a program with a set of function calls, maintain one "memo" for each function. Whenever a function returns a value, this value is stored as a memo for that function. When we want to call a function, we first check its memo. If return values are known for that function,

*Dept. of Software Development, Monash University, Caulfield East, Melbourne, VIC, Australia, 3145; +61-3-9903-1033; +61-3-9903-1077(fax);
Email: timm@insect.sd.monash.edu.au; *URL*: http://www.sd.monash.edu.au/~timm

†School of Computer Science, University of New South Wales, P.O. Box 1 Kensington, Sydney, NSW, Australia, 2033; +61-3-9697-5318 *Email*: andrewt@cse.unsw.edu.au

we simply return the memo thus avoiding the computational cost of calling the function again. Memoing can significantly reduce the runtimes of a program.

Memoing can be a very general process. Warren [20] reports experiments with memoing for logic programming. Warren's general logic programming memoing algorithm decreases the complexity of certain tasks to that of special-purpose algorithms designed especially for those tasks. The generality of Warren's work motivated us to explore memoing for the HT4 abductive inference procedure. To our knowledge, the complexity of abduction has not been tackled before using memoing techniques. Two memoing algorithms were devised: CACHE and CLUMP (see Sections 6 & 7). These algorithms are introduced after an overview of abduction and HT4 .

3 Abduction

Abduction is the search for assumptions $\mathcal{A}$ which, when combined with some theory $\mathcal{T}$ achieves some set of goals $\mathcal{OUT}$ without causing some contradiction [6]. That is:

EQ_1: $\mathcal{T} \cup \mathcal{A} \vdash \mathcal{OUT}$

EQ_2: $\mathcal{T} \cup \mathcal{A} \nvdash \bot$

Note that EQ_1 and EQ_2 are more than just "inference to the best explanation" (the usual informal decription of abduction). EQ_1 and EQ_2 can be summarised as follows: *make what inferences you can that are relevant to some goal, without causing any contradictions.* Menzies argues that collecting and studying the proof trees that satisfy these two equations is a general framework for a range of KBS tasks such as prediction, classification, explanation, qualitative reasoning, planning, monitoring, set-covering diagnosis, consistency-based diagnosis, validation, and verification [9, 10].

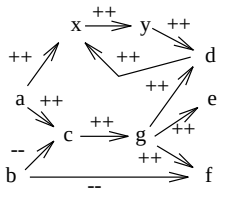
Unfortunately, abduction can be very slow. Selman & Levesque show that even when only one abductive explanation is required and the theory is restricted to be acyclic, then abduction is NP-hard [19]. Bylander *et. al.* make a similar pessimistic conclusion [1]. Most known abductive inference engines exhibit exponential runtimes for real-world inputs. For example, many the articles in [15] are concerned with heuristic optimisations of abduction. Computationally tractable abductive inference algorithms (e.g. [1, 6]) typically make restrictive assumptions about the nature of the theory or the available data. Such techniques are not applicable to arbitrary theories.

In practice, we find that our abductive framework is practical for KBS with less than 800 literals [9]. 800 literals is larger than many of the systems we see in contemporary practice [17]. Therefore, we have argued elsewhere that our abductive framework is practical for the types of system we see in contemporary practice [10]. Nevertheless, we would like some confidence that we scale up our abductive framework. Therefore, we explore methods of reducing its complexity.

4 HT4

Given a set of goal $\mathcal{OUT}$puts and known $\mathcal{IN}$puts, then HT4 can use (e.g.) the qualitative theory of Figure 1 to build a set of proof trees $\mathcal{P}$ connecting $\mathcal{OUT}$=`{dUp,eUp,fDown}` to $\mathcal{IN}$=`{aUp,bUp}`. In Figure 1, (i) `x` $\stackrel{++}{\rightarrow}$ `y` denotes that `y` being `up` or `down` could be explained by `x` being `up` or `down` respectively while (ii) `x` $\stackrel{--}{\rightarrow}$ `y` denotes that `y` being `up` or `down` could be explained by `x` being `down` or `up` respectively. In qualitative reasoning, we cannot believe in different values for the same object at the same time. Some of these invariants $\mathcal{I}$ are shown in Figure 1. The proofs $\mathcal{P}$ that do not violate our invariants are: $\mathcal{P}_1$=

aUp → xUp → yUp → dUp, $\mathcal{P}_2$= aUp → cUp → gUp → dUp, $\mathcal{P}_3$= aUp → cUp → gUp → eUp, $\mathcal{P}_4$= bUp → cDown → gDown → fDown, $\mathcal{P}_5$= bUp → fDown.



i(aUp,aDown).
i(bUp,bDown).
i(cUp,cDown).
i(dUp,dDown).
.....

Figure 1: A qualitative theory.

Some of these proofs make assumptions; i.e. use a literal that is not one of the known $\mathcal{FACTS}$ (typically, $\mathcal{FACTS} = \mathcal{IN} \cup \mathcal{OUT}$). A major problem of an abductive inference procedure such as HT4 is managing the assumptions space generated during inferencing. Note that some of the assumptions will contradict other assumptions and will be *controversial* (denoted $\mathcal{A}_{\mathrm{C}}$). In terms of uniquely defining an assumption space, the key controversial assumptions are those controversial assumptions that are not dependent on other controversial assumptions. These *base* controversial assumptions are denoted $\mathcal{A}_{\mathrm{B}}$. In our example, $\mathcal{A}_{\mathrm{C}}$={cUp,cDown,gUp,gDown} and $\mathcal{A}_{\mathrm{B}}$ = {cUp, cDown} (since Figure 1 tells us that g is fully determined by c). Depending on which base controversial assumptions we elect to make, HT4 will condone different inferences. If we assume cUp, then we can believe in the *world* containing the proofs $\mathcal{P}_1$ $\mathcal{P}_2$ $\mathcal{P}_3$ $\mathcal{P}_5$ since those proofs do not assume cUp. If we assume cDown, then we can believe in the *world* containing the proofs $\mathcal{P}_1$ $\mathcal{P}_4$ $\mathcal{P}_5$ since these proofs do not assume cDown. These worlds are shown in Figure 2.

Once the worlds are known, domain-specific $\mathcal{BEST}$ operators can be applied to return the world(s) that satisfy some desired criteria. For example, in KBS validation, we return the world(s) that cover the largest subsets of $\mathcal{OUT}$. Alternatively, for an explanation system, we would return the world(s) that have the greatest overlap with a user profile containing the set of edges and vertices which they are familiar with. For a survey of other $\mathcal{BEST}$ operators, see [10].

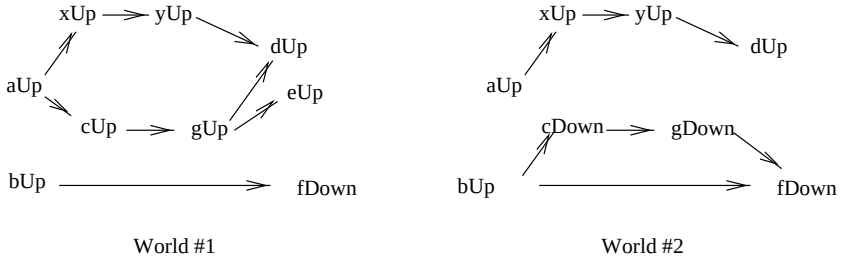


Figure 2: The worlds of Figure 1

More generally, we say that abudction builds worlds $\mathcal{W}$ from consistent, maximal subsets of $\mathcal{P}$ (consistent with respect to $\mathcal{I}$ and maximal with respect to size). These worlds can be built one at a time as required [5] or concurrently [3, 13]. All worlds consistent with the $\mathcal{FACTS}$ may be generated (e.g. consistency-based diagnosis [3, 18]). Alternatively, world generation may be restricted to only the proofs that connected literals the user is interested in (e.g. set-covering diagnosis [7, 16]).

HT4 adopts a set-covering view. The core problem in HT4 is finding $\mathcal{A}_{\mathrm{B}}$. In the *forward sweep*, $\mathcal{A}_{\mathrm{C}}$ is discovered as a side-effect of computing the transitive closure of $\mathcal{IN}$. In the *backwards sweep*, proof generation is then constrained to the transitive closure of $\mathcal{IN}$. As a proof is grown from a member of $\mathcal{OUT}$ back to any $\mathcal{IN}$, several invariants are maintained:

- Proofs can't contain loops.
- Proofs maintain a *forbids* set; i.e. a set of literals that are incompatible with the literals used in the proof. For example, the literals used in $\mathcal{P}_1$ forbid the literals {`aDown, xDown, yDown, dDown`}.
- The upper-most $\mathcal{A}_{\mathrm{C}}$ found along the way is recorded as that proof's *guess*. The union of all the guesses of all the proofs is $\mathcal{A}_{\mathrm{B}}$.

Once $\mathcal{A}_{\mathrm{B}}$ is known then the proofs can be sorted into worlds in the *worlds sweep*. HT4 extracts all the objects $\mathcal{O}$ referenced in $\mathcal{A}_{\mathrm{B}}$. A world-defining environment $\mathcal{ENV}_{\mathrm{i}}$ is created for each combination of objects and their values. In our example, $\mathcal{ENV}_1 = \{\texttt{cUp}\}$ and $\mathcal{ENV}_2 = \{\texttt{cDown}\}$. The worlds sweep is simply two nested loops over each $\mathcal{ENV}_{\mathrm{i}}$ and each $\mathcal{P}_j$. A proof $\mathcal{P}_j$ belongs in world $\mathcal{W}_i$ if its *forbids* set does not intersect the assumptions $\mathcal{ENV}_{\mathrm{i}}$ that define that world.

5 Complexity of HT4

For a theory comprising a directed and-or graph connecting literals $\mathcal{V}$ with $\mathcal{E}$ edges and average fan-in $\mathcal{F}=\frac{|\mathcal{E}|}{|\mathcal{V}|}$, the worst-case complexity of the forwards sweep is $O(|\mathcal{V}|^3)$. If the average size of a proof is X, then worse case backwards sweep is $O(X^{\mathcal{F}})$. The worlds sweep is proportional to the number of proofs and the number of world-defining assumptions; i.e. (i.e. $O(|\mathcal{P}| * |\mathcal{ENV}|)$ $= O(X^{\mathcal{F}}) * |\mathcal{ENV}|)$). In practice, we have found the backwards sweep to the limiting step of HT4 . We therefore explored memoing of the backwards sweep.

In order to retain generality, HT4 must support the general case of full worlds generation. One technique for taming abductive complexity that we cannot apply is some heuristic cull of the search space. This is the essence of DeKleer and Williams' entropy technique [4] where a information gain heuristic is used to contain the search. Not all domains can be culled in this manner. For example, consider the task of resolving a debate between feuding experts. An HT4 validation algorithm returns the world(s) that *covers* (i.e. explains) the greatest percentage of known behaviour. $EXPERT_1$'s theory $\mathcal{T}_1$ is better than $EXPERT_2$'s theory $\mathcal{T}_2$ iff $\mathcal{T}_1^{cover} \gg \mathcal{T}_2^{cover}$. By definition, this process cannot be heuristically culled lest $EXPERT_2$ reject the analysis[1].

6 CLUMP

One way to represent a cache in an and-or graph is to add new edges between vertices whenever an inference path was found between them. Using these new edges, the inference engine can avoid stepping over a space it has searched successfully before. Rather, it can jump over several steps with a single inference. We call such jump edges "clumps" since, in effect, they clump together the processing of many vertices into a single new vertex. We acknowledge that this

[1] "You mean", says $EXPERT_2$, "that there is a chance that my theory is better than $EXPERT_1$'s? Well then, I knew it all along!"

memoing technique is analogous to Kowalski's connection graph technique [8] or Coldwell's tableau resolution theorem prover [2].

CLUMP was a small algorithm that run in between the forwards and backwards sweep. This algorithm searched for vertices which have only a single parent. Such vertices were "clumped" with their parent. For example, in Figure 1, g could be clumped with c. The process repeated itself recursively. Such clumps represent "choice-less" sequences of inferences. If the backwards sweep ever encountered a leaf of a clump, it has no choice except to move to the root of a clump. CLUMP augmented each clump with the set of vertices it contains as well as the union of the *forbids* set of its members. Such sets could be consulted very quickly when testing for looping and constraint violation.

Depending on the topology of the network, clumping can produce a spectacular reduction in the search space. Consider Figure 3.i. All these vertices have only one parent. All of Figure 3.i could be replaced with a one clump. Clumpable vertices occur very frequently. In 870 runs over one large neuroendocrinological model we have studied intensively, 52% on average of the vertices have only one parent within the transitive closure of $\mathcal{IN}$. Such observations lead to considerable interest in CLUMP .

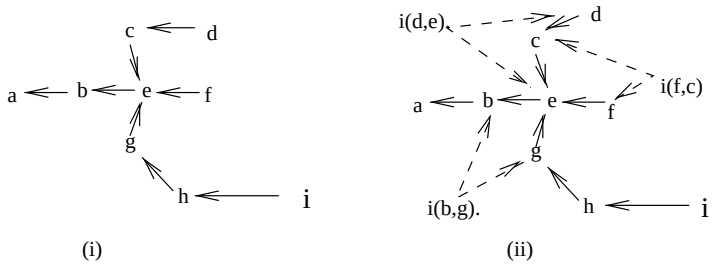


Figure 3: (i) No invariants; (ii) With invariants.

However, because of the invariants, CLUMP ing turned out to be more complicated than expected. Consider Figure 3.ii where $\mathcal{IN} = \{\texttt{f,d,i}\}$ and $\mathcal{OUT} = \{\texttt{a,e}\}$. Note that Figure 3.ii is merely Figure 3.i with some added invariants. If we clumped all of Figure 3.ii into a single clump, we would lose some important features of the search space. For example:

- A proof for i can terminate on e but not on a (due to $\mathcal{I}\{\texttt{b,g}\}$). A clump containing all of {a,b,c,d,e,f,g,h,i} would contain a *forbids* sets that would preclude this proof.
- While a clump containing all of {a,b,c,d,e,f,g,h,i} cannot explain both f *and* d (due to $\mathcal{I}\{\texttt{d,e}\}$ and $\mathcal{I}\{\texttt{f,c}\}$), it can still explain f *or* d.

Clearly, over-zealous clumping can lead to problems. We spent some time searching for a more restrictive "safe" clumping policy which did not constrain the multiple-worlds inferencing or compromise the completeness of HT4 . We have only found one safe clumping policy: do not add members of $\mathcal{IN}$ or $\mathcal{OUT}$ to clumps. In this policy, only {g,h} would be clumped in Figure 3.ii. When implemented, the results were not impressive. In 870 runs over our large neuroendocrinological model, the time to clump added a third to the total runtimes yet only reduced the backwards sweep time by 3%. Hence, CLUMP was abandoned.

7 CACHE

The CACHE algorithm attached a memo to the HT4 routine which explored the parents of a particular vertex in the backwards sweep. If N proofs were found which connected a vertex X to the $\mathcal{IN}$ set, these proofs were cached at X and re-used if ever HT4 returned to X.

CACHE was tested by running HT4 with and without caching. A four-fold speed-up was noted in the execution time of a large model (554 literals, fanout of 2.25). However, when the caching output was compared to the no-caching version, 8% of the generated proofs were different suggesting that CACHE had introduced some incompleteness into the inferencing.

After some investigation, it was realised that the proofs found at a vertex were dependent on the route taken from the output being explored to that vertex. Recall that proof generation is constrained by the *forbids* set built incrementally as a candidate proof grows during the backwards sweep. Proofs generated along different routes have different *forbids* sets. If a proof arrives at a vertex from two different routes, then (potentially) different upstream vertices are usable by that proof (i.e. those that do not conflict with the *forbids* set).

For example, consider the invariant $\mathcal{I}$(c,d) in Figure 4. If $\mathcal{OUT}$ = {a,c} and $\mathcal{IN}$ = {e}, then we will arrive at vertex b twice (once while proving a and once while proving c). If we arrive from a first, CACHE would record that a valid proof from b is {b,d,e}. However, when we subsequently arrive from c, HT4 would find {b,d,e} invalid due to $\mathcal{I}$(c,d).

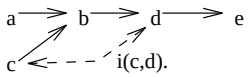


Figure 4: A problematic network topology for CACHE .

Various special cases were built into CACHE in order to cope with problematic topologies such as Figure 4. After a memo was extracted, some post-processing was required before the memo could be safely used (such as testing for contradictory *forbids*). In some cases, the post-processing was as complex as the ignoring the cache and building the proof again (e.g. when handling loops). The size of the CACHE program grew alarmingly. For every special case extension built into CACHE , there was always a network topology that introduced some incompleteness into the reasoning.

Eventually, it was realised that CACHE had to remove all tests of the *forbids* set from proof generation; i.e. defer a test that a proof does not contain incompatible invariants until after the backward sweep was finished. This version of CACHE ran out of memory. Every visited vertex cached every proof it could generate from itself up to any member of inputs. CACHE was hence abandoned.

8 Conclusion

We have argued that (i) abduction is an important framework for KBS systems; (ii) abduction is NP-hard; (iii) we have failed to tame its complexity with memoing. The CACHE experiment tells us that naive memoing over and-or graphs with invariants can lead to incomplete inferences. In the CLUMP experiment, we had to severely constrain clumping to edge subsets which avoid the incompleteness problem. The resulting system ran no faster than the unCLUMP ed version.

Note that CLUMP and CACHE failed for the same fundamental reason. When performing deduction over and-or graphs, memoing works because (loop detection aside) the context of how

we arrived at a literal X is irrelevant to what inferences we can make from X. However, with abduction, the trail of literals that lead us to X constrains the literals we can reach subsequently. Traditional memoing only caches the inferences that can be made from X and ignores the trail that lead us to X. If we add this context trial to the memo, we then have to split the retrieved memo into the smaller memos that were relevant for different context trails. Such "memo-splitting" is really a synonym for the multiple-worlds reasoning performed at the global level by HT4 .

Despite this negative result, we decline to abandon this abductive framework. HT4 is a simple enough procedure to allow us to explore the computational cost of certain common expert system tasks. Recall that we have argued elsewhere that our abductive framework is practical for the types of system we see in contemporary practice [10]. We know that, ultimately, many interesting inference tasks are intractable. However, with HT4 we have the possibility of being able to say precisely where are the computational limits to KBS inferencing.

References

[1] T. Bylander, D. Allemang, M.C. M.C. Tanner, and J.R. Josephson. The Computational Complexity of Abduction. *Artificial Intelligence*, 49:25–60, 1991.

[2] J.M. Coldwell and G. Wrightson. Lemmas and Links in Analystic Tableau. In C. Zhang, J. Debenham, and D. Lukose, editors, *Artificial Intelligence: Sowing the Seeds for the Future; Proceedings of AI '94*, pages 275–282. World Scientific, 1994.

[3] J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence*, 28:163–196, 1986.

[4] J. DeKleer and B.C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32:97–130, 1 1987.

[5] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12:231–272, 1979.

[6] K. Eshghi. A Tractable Class of Abductive Problems. In *IJCAI '93*, volume 1, pages 3–8, 1993.

[7] K. Konoligue. Abduction versus Closure in Causal Theories. *Artificial Intelligence*, 53:255–272, 1992.

[8] R. Kowalski. A Proof Procedure Using Connection Graphs. *Journal of the Association for Computing Machinery*, 22(4):572–595, October 1975.

[9] T.J. Menzies. Applications of Abduction #2: Knowledge Level Modeling. Technical Report TR95-23, Department of Software Development, Monash University, 1995.

[10] T.J. Menzies. Applications of Abduction to Expert Systems. Technical report, Department of Software Development, Monash University, 1995.

[11] T.J. Menzies. *Principles for Generalised Testing of Knowledge Bases*. PhD thesis, University of New South Wales, 1995.

[12] T.J. Menzies and P. Compton. On the Practicality of Abductive Validation. Technical report, Department of Software Development, Monash University, 1995.

[13] T.J. Menzies and W. Gambetta. Exhaustive Abduction: A Practical Model Validation Tool. In *ECAI '94 Workshop on Validation of Knowledge-Based Systems*, 1994.

[14] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[15] P. O'Rourke. Working Notes of the 1990 Spring Symposium on Automated Abduction. Technical Report 90-32, University of California, Irvine, CA., 1990. September 27, 1990.

[16] D. Poole. Normality and Faults in Logic-Based Diagnosis. In *IJCAI '89*, pages 1304–1310, 1989.

[17] A.D. Preece and R. Shinghal. Verifying Knowledge Bases by Anomaly Detection: An Experience Report. In *ECAI '92*, 1992.

[18] R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32:57–96, 1 1987.

[19] B. Selman and H.J. Levesque. Abductive and Default Reasoning: a Computational Core. In *AAAI '90*, pages 343–348, 1990.

[20] D.S. Warren. Memoing for Logic Programs. *Communications of the ACM*, 35:93–101, March 1992.