Proc. GKR 2013

Concurrent Reasoning with Inference Graphs

Daniel R. Schlegel and **Stuart C. Shapiro**
Department of Computer Science and Engineering
University at Buffalo
<drschleg,shapiro>@buffalo.edu

Abstract

Since their popularity began to rise in the mid-2000s there has been significant growth in the number of multi-core and multi-processor computers available. Knowledge representation systems using logical inference have been slow to embrace this new technology. We present the concept of inference graphs, a natural deduction inference system which scales well on multi-core and multi-processor machines. Inference graphs enhance propositional graphs by treating propositional nodes as tasks which can be scheduled to operate upon messages sent between nodes via the arcs that already exist as part of the propositional graph representation. The use of scheduling heuristics within a prioritized message passing architecture allows inference graphs to perform very well in forward, backward, bi-directional, and focused reasoning. Tests demonstrate the usefulness of our scheduling heuristics, and show significant speedup in both best case and worst case inference scenarios as the number of processors increases.

1 Introduction

Since at least the early 1980s there has been an effort to parallelize algorithms for logical reasoning. Prior to the rise of the multi-core desktop computer, this meant massively parallel algorithms such as that of [Dixon and de Kleer, 1988] on the (now defunct) Thinking Machines Corporation's Connection Machine, or using specialized parallel hardware which could be added to an otherwise serial machine, as in [Lendaris, 1988]. Parallel logic programming systems designed during that same period were less attached to a particular parallel architecture, but parallelizing Prolog (the usual goal) is a very complex problem [Shapiro, 1989], largely because there is no persistent underlying representation of the relationships between predicates. Parallel Datalog has been more successful (and has seen a recent resurgence in popularity [Huang *et al.*, 2011]), but is a much less expressive subset of Prolog. Recent work in parallel inference using statistical techniques has returned to large scale parallelism using GPUs, but while GPUs are good at statistical calculations, they do not do logical inference well [Yan *et al.*, 2009].

We present *inference graphs* [Schlegel and Shapiro, 2013], a graph-based natural deduction inference system which lives within a KR system, and is capable of taking advantage of multiple cores and/or processors using concurrent processing techniques rather than parallelism [The Joint Task Force on Computing Curricula *et al.*, 2013]. We chose to use natural deduction inference, despite the existence of very well performing refutation based theorem provers, because our system is designed to be able to perform forward inference, bi-directional inference [Shapiro *et al.*, 1982], and focused reasoning in addition to the backwards inference used in resolution. Natural deduction also allows formulas generated during inference to be retained in the KB for later re-use, whereas refutation techniques always reason about the negation of the formula to be derived, making intermediate derivations invalid. In addition, our system is designed to allow formulas to be disbelieved, and to propagate that disbelief to dependent formulas. We believe inference graphs are the only concurrent inference system with all these capabilities.

Inference graphs are, we believe, unique among logical inference systems in that the graph representation of the KB is the same structure used for inference. Because of this, the inference system needn't worry about maintaining synchronicity between the data contained in the inference graph, and the data within the KB. This contrasts with systems such as [Wachter and Haenni, 2006] which allow queries to be performed upon a graph, not within it. More similar to our approach is that of Truth Maintenance Systems [Doyle, 1979] (TMSes), which provide a representation of knowledge based on justifications for truthfulness. The TMS infers within itself the truth status of nodes based on justifications, but the justifications themselves must be provided by an external inference engine, and the truth status must then be reported back to the inference engine upon request.

Drawing influences from multiple types of TMS [Doyle, 1979; de Kleer, 1986; McAllester, 1990], RETE networks [Forgy, 1982], and Active Connection Graphs [McKay and Shapiro, 1981], and implemented in Clojure [Hickey, 2008], inference graphs are an extension of propositional graphs allowing messages about assertional status and inference control to flow through the graph. The existing arcs within propositional graphs are enhanced to carry messages from antecedents to rule nodes, and from rule nodes to consequents. A rule node in an inference graph combines messages and

carries out introduction and elimination rules to determine if itself or its consequents are true or negated.

In Section 2 we review propositional graphs and introduce a running example, followed by our introduction to inference graphs in Section 3. Section 4 explains how the inference graphs are implemented in a concurrent processing system. We evaluate the implementation in Section 5 and finally conclude with Section 6.

2 Propositional Graphs

Propositional graphs in the tradition of the SNePS family [Shapiro and Rapaport, 1992] are graphs in which every well-formed expression in the knowledge base, including individual constants, functional terms, atomic formulas, or non-atomic formulas (which we will refer to as "rules"), is represented by a node in the graph. A rule is represented in the graph as a node for the rule itself (henceforth, a *rule node*), nodes for the argument formulas, and arcs emanating from the rule node, terminating at the argument nodes. Arcs are labeled with an indication of the role the argument plays in the rule, itself. Every node is labeled with an identifier. Nodes representing individual constants, proposition symbols, function symbols, or relation symbols are labeled with the symbol itself. Nodes representing functional terms or non-atomic formulas are labeled `wft`*i*, for some integer, *i*.[1] No two nodes represent syntactically identical expressions; rather, if there are multiple occurrences of one subexpression in one or more other expressions, the same node is used in all cases.

In this paper, we will limit our discussion to inference over formulas of Propositional Logic.[2] Specifically, we have implemented introduction and elimination rules for the set-oriented connectives `andor`, and `thresh` [Shapiro, 2010], and the elimination rules for numerical entailment. The `andor` connective, written (`andor` $(i\ j)$ $p_1...p_n$), $0 \leq i \leq j \leq n$, is true when at least i and at most j of $p_1...p_n$ are true (that is, an `andor` may be introduced when those conditions are met). It generalizes `and`, `or`, `nand`, `nor`, `xor`, and `not`. For the purposes of `andor`-elimination, each of $p_1...p_n$ may be treated as an antecedent or a consequent, since when any j formulas in $p_1...p_n$ are known to be true (the antecedents), the remaining formulas (the consequents) can be inferred to be negated. For example with `xor`, for which $i = j = 1$, a single true formula causes the rest to become negated, and if all but one are found to be negated, the remaining one can be inferred to be true. The `thresh` connective, the negation of `andor`, is mainly used for equivalence (`iff`). Numerical entailment is a generalized entailment connective, written $\{a_1...a_n\} i \to \{c_1...c_m\}$ where at least i of the antecedents, $a_1...a_n$, must be true for all of the consequents, $c_1...c_m$, to be true. The examples and evaluations in this paper will make exclusive use of two special cases of numerical entailment – or-entailment, where $i = 1$, and and-entailment, where $i = n$ [Shapiro and Rapaport, 1992].

[1] "`wft`" rather than "`wff`" for reasons that needn't concern us in this paper.

[2] Though the system is designed to be extended to a logic with quantified variables in the future.

For a running example, we will use two rules, one using an and-entailment and one using an or-entailment: `{a, b, c}` $\wedge\!\!\to$ `d`, meaning that whenever `a`, `b`, and `c` are true then `d` is true; and `{d, e}` $\vee\!\!\to$ `f`, meaning that whenever `d` or `e` is true then `f` is true. In Figure 1 `wft1` represents the and-entailment `{a, b, c}` $\wedge\!\!\to$ `d`. The identifier of the rule node `wft1` is followed by an exclamation point, "!", to indicate that the rule (well-formed formula) is asserted – taken to be true. The antecedents, `a`, `b`, and `c` are connected to `wft1` by arcs labeled with `∧ant`, indicating they are antecedents of an and-entailment. An arc labeled `cq` points from `wft1` to `d`, indicating `d` is the consequent of the rule. `wft2` represents the or-entailment `{d, e}` $\vee\!\!\to$ `f`. It is assembled in a similar way to `wft1`, but the antecedents are labeled with `∨ant`, indicating this rule is an or-entailment. While much more complex examples are possible, we stick to this rather simplistic one since it illustrates the concepts presented in this paper without burdening the reader with the details of the, perhaps less familiar, `andor` and `thresh` connectives.

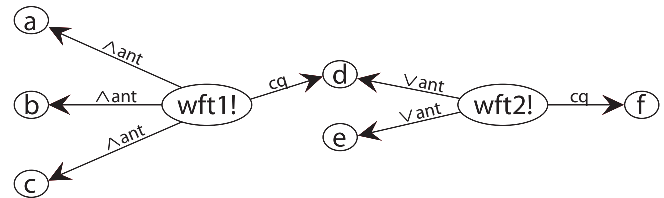

Figure 1: Propositional graph for the assertions that if `a`, `b`, and `c` are true, then `d` is true, and if `d` or `e` are true, then `f` is true.

The propositional graphs are fully indexed, meaning that not only can the node at the end of an arc be accessed from the node at the beginning of the arc, but the node at the beginning of the arc can be accessed from the node at the end of the arc. For example, it is possible to find all the rule nodes in which `d` is a consequent by following `cq` arcs backwards from node `d` (We will refer to this as following the reverse `cq` arc.), and it is possible to find all the or-entailment rule nodes in which `d` is an antecedent by following reverse `∨ant` arcs from node `d`.

3 Inference Graphs

Inference graphs are an extension of propositional graphs to allow deductive reasoning to be performed in a concurrent processing system. Unlike many KR systems which have different structures for representation and inference, inference graphs serve as both the representation and the inference mechanism.

To create an inference graph, certain arcs and certain reverse arcs in the propositional graph are augmented with *channels* through which information can flow. Channels come in two forms. The first type, *i-channels*, are added to the reverse antecedent arcs. These are called i-channels since they carry messages reporting that "I am true" or "I am negated" from the antecedent node to the rule node. Chan-

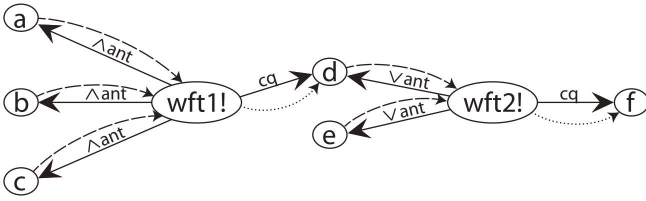

Figure 2: The same propositional graph from Figure 1 with the appropriate channels added. Channels represented by dashed lines are i-channels and are drawn from antecedents to rule nodes. Channels represented by dotted lines are u-channels and are drawn from rule nodes to consequents.

nels are also added to the consequent arcs, called *u-channels*[3], since they carry messages to the consequents which report that "you are true" or "you are negated." Rules are connected by shared subexpressions, as `wft1` and `wft2` are connected by the node `d`.

Figure 2 shows the propositional graph of Figure 1 with the appropriate channels illustrated. i-channels (dashed arcs) are drawn from `a`, `b`, and `c` to `wft1`, and from `d`, and `e` to `wft2`. These channels allow the antecedents to report to a rule node when it (or its negation) has been derived. u-channels (dotted arcs) are drawn from `wft1` to `d` and from `wft2` to `f` so that rule nodes can report to consequents that they have been derived.

Each channel contains a valve. Valves enable or prevent the flow of messages forward through the graph's channels. When a valve is closed, any new messages which arrive at it are added to a waiting set. When a valve opens, messages waiting behind it are sent through.

Inference graphs are capable of forward, backward, bi-directional, and focused inference. In forward inference, messages flow forward through the graph, ignoring any valves they reach. In backward inference, messages are sent backwards through incoming channels to open valves, and hence other messages about assertions are allowed to flow forward only along the appropriate paths. Normally when backward inference completes, the valves are closed. Focused inference can be accomplished by allowing those valves to remain open, thus allowing messages reporting new assertions to flow through them to answer previous queries. Bi-directional inference can begin backward inference tasks to determine if an unasserted rule node reached by a forward inference message can be derived.

3.1 Messages

Messages of several types are transmitted through the inference graph's channels, serving two purposes: relaying newly derived information, and controlling the inference process. A message can be used to relay the information that its origin has been asserted or negated (an I-INFER message), that its destination should now be asserted or negated (U-INFER), or that its origin has either just become unasserted or is no longer sufficiently supported (UNASSERT). These messages all flow forward through the graph. Other messages flow backward through the graph controlling inference by affecting the channels: BACKWARD-INFER messages open them, and CANCEL-INFER messages close them. Messages are defined as follows:

$$< orig, support, type, astatus?, finfer?, priority >$$

where $orig$ is the node which produced the message; $support$ is the set of support which allowed the derivation producing the message (for ATMS-style belief revision [de Kleer, 1986; Martins and Shapiro, 1988]); $type$ is the type of the message (such as I-INFER or BACKWARD-INFER, described in detail in the next several subsections); $finfer?$ states whether this message is part of a forward-inference task; $astatus?$ is whether the message is reporting about a newly true or newly false node; and $priority$ is used in scheduling the consumption of messages (discussed further in Section 4).

I-INFER

I-INFER messages originate at rule nodes which have been newly derived as either true or false, and are sent to rule nodes in which the originating node is an antecedent. An I-INFER message contains a support set which contains every node used in deriving the originator of the message. These messages optionally can be flagged as part of a forward inference operation, in which case they treat any closed valves they reach as if they were open. The priority of an I-INFER message is one more than that of the message that caused the change in assertional status of the originator. Section 4 will discuss why this is important.

U-INFER

U-INFER messages originate at rule nodes which have just learned enough about their antecedents to fire. They inform the target node what its new assertional status is – either true or false.[4] As with I-INFER messages, U-INFER messages contain a support set, can be flagged as being part of a forward inference operation, and have a priority one greater than the message that preceded it.

BACKWARD-INFER

BACKWARD-INFER messages are generated by rules which need to infer the assertional status of some or all of their antecedents in order to decide the assertional status of one or more of its consequents. These messages set up a backward inference operation by passing backward through channels and opening any valves they reach. The priority of these messages is lower than any inference tasks which may take place. This allows any messages waiting at the valves to flow forward immediately and begin inferring new formulas towards the goal of the backward inference operation.

CANCEL-INFER

Inference can be canceled either in whole by the user or in part by a rule which determines that it no longer needs

[3]u-channels and U-INFER messages were previously called y-channels and Y-INFER messages in [Schlegel and Shapiro, 2013; Schlegel, 2013].

[4]For example a rule representing the exclusive or of `a` and `b` could tell `b` that it is true when it learns that `a` is false, or could tell `b` that it is false when it learns that `a` is true.

to know the assertional status of some of its antecedents.[5] CANCEL-INFER messages are sent from some node backward through the graph. A node relays the message backward if it has been initiated by the inference system itself, or if the message has been received on all outgoing channels. These messages cause valves to close as they pass through, and cancel any BACKWARD-INFER messages scheduled to re-open those same valves, halting inference. CANCEL-INFER messages always have a priority higher than any inference tasks.

UNASSERT
Each formula which has been derived in the graph has one or more sets of support. For a formula to remain asserted, at least one of its support sets must have all of its formulas asserted as well. When a formula is unasserted by a human (or, eventually, by belief revision), a message must be sent forward through the graph to recursively unassert any formulas which have the unasserted formula in each of their support sets, or is the consequent of a rule which no longer has the appropriate number of positive or negative antecedents. UNASSERT messages have top priority, effectively pausing all other inference, to help protect the consistency of the knowledge base.

3.2 Rule Node Inference

Inference operations take place in the rule nodes. When a message arrives at a rule node the message is translated into *Rule Use Information*, or RUI [Choi and Shapiro, 1992]. A RUI is defined as:

$$< pos, neg, flaggedNS, support >$$

where pos and neg are the number of known true (“positive”) and negated (“negative”) antecedents of the rule, respectively; the $flaggedNS$ is the *flagged node set*, which contains a mapping from each antecedent with a known truth value to its truth value, and $support$ is the set of support, the set of formulas which must be true for the assertional statuses in the $flaggedNS$ to hold.

All RUIs created at a node are cached. When a new one is made, it is combined with any already existing ones – pos and neg from the RUIs are added and the set of support and flagged node set are combined. The output of the combination process is a set of new RUIs created since the message arrived at the node. The pos and neg portions of the RUIs in this set are used to determine if the rule node's inference rules can fire. A disadvantage of this approach is that some rules are difficult, but not impossible, to implement, such as negation introduction and proof by cases. For us, the advantages in capability outweigh the difficulties of implementation. If the RUI created from a message already exists in the cache, no work is done. This prevents re-derivations, and can cut cycles.

Figure 3 shows the process of deriving `f` in our running example. We assume backward inference has been initiated, opening all the valves in the graph. First, in Figure 3a, messages about the truth of `a`, `b`, and `c` flow through i-channels to `wft1`. Since `wft1` is and-entailment, each of its antecedents must be true for it to fire. Since they are, in Figure 3b the message that `d` is true flows through `wft1`'s u-channel. `d` becomes asserted and reports its new status through its i-channel (Figure 3c). In Figure 3d, `wft2` receives this information, and since it is an or-entailment rule and requires only a single antecedent to be true for it to fire, it reports to its consequents that they are now true, and cancels inference in `e`. Finally, in Figure 3e, `f` is asserted, and inference is complete.

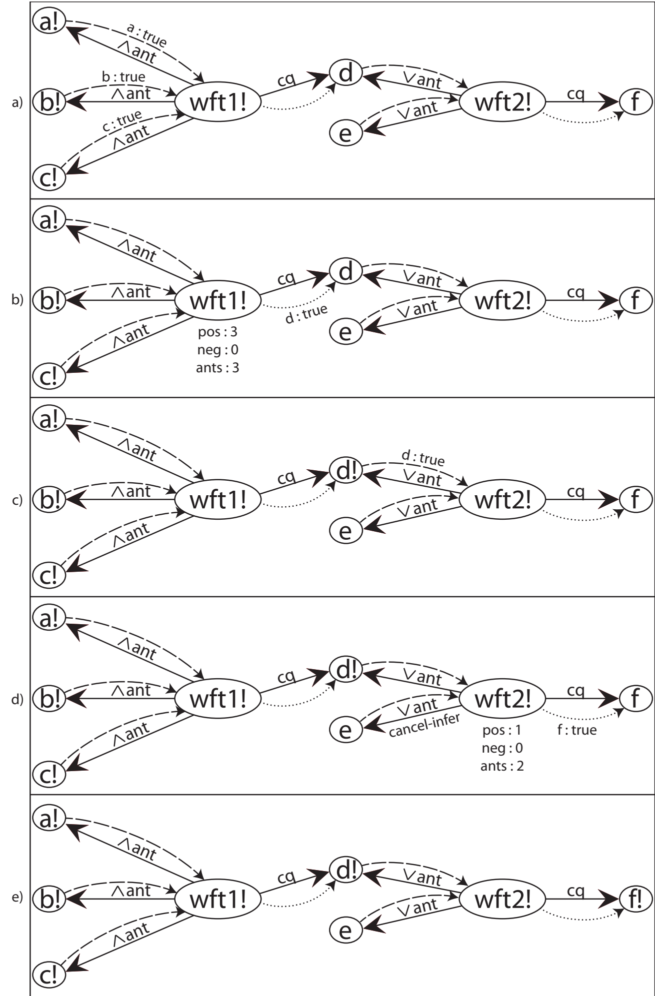

Figure 3: a) Messages are passed from `a`, `b`, and `c` to `wft1`. b) `wft1` combines the messages from `a`, `b`, and `c` to find that it has 3 positive antecedents, of a total of 3. The and-entailment can fire, so it sends a message through its u-channel informing its consequent, `d`, that it has been derived. c) `d` receives the message that it is asserted and sends messages through its i-channel. d) `wft2` receives the message that `d` is asserted. Only one true antecedent is necessary for or-entailment elimination, so it sends a message through its u-channels that its consequent, `f`, is now derived. It also cancels any inference in its other antecedents by sending a CANCEL-INFER message to `e`. e) `f` is derived.

[5] We recognize that this can, in some cases, prevent the system from deriving a contradiction automatically.

4 Concurrent Reasoning

The inference graph's structure lends itself naturally to concurrent inference. It is clear in our running example that if inference were required to derive `a`, `b`, and `c` that each of those inference tasks could be running concurrently. After each of `a`, `b`, and `c` were asserted, messages would be sent to `wft1`, as in our example. The RUIs generated from the messages would then need to be combined. Since there is shared state (the RUI cache) we perform the combination of RUIs synchronously using Clojure's Software Transactional Memory, guaranteeing we don't "lose" results. We need not concern ourselves with the actual order in which the RUIs are combined, since the operation is commutative, meaning we don't need to maintain a queue of changes to the RUI cache.

In addition to the RUI cache, we must also update the set of asserted formulas whenever a formula is newly derived. We use the same technique as above for this, again recognizing that the order in which formulas are asserted is not important. It is possible to perform these assertions in a thread outside of the inference task, but tests show no advantage in doing so[6], especially in the case where we have as many (or more) inference threads as CPUs.

In order to perform inference concurrently, the inference graph is divided into *inference segments* (henceforth, *segments*). A segment represents the inference operation – from receipt of a message to sending new ones – which occurs in a node. Valves delimit segments, as seen in Figure 4. When a message passes through an open valve a new *task* is created – the application of the segment's inference function to the message. When tasks are created they enter a global prioritized queue, where the priority of the task is the priority of the message. When the task is executed, inference is performed as described above, and any newly generated messages are sent toward its outgoing valves for the process to repeat.

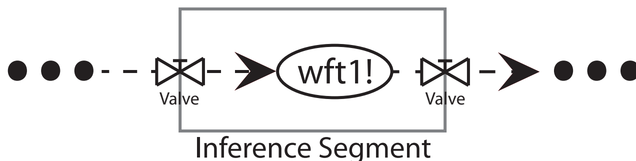

Figure 4: A single inference segment is shown in the gray bounding box.

4.1 Scheduling Heuristics

The goal of any inference system is to infer the knowledge requested by the user. If we arrange an inference graph so that a user's request (in backward inference) is on the right, and channels flow from left to right wherever possible (the graph may contain cycles), we can see this goal as trying to get messages from the left side of the graph to the right side of the graph. We, of course, want to do this as quickly as possible.

Every inference operation begins processing messages some number of levels to the left of the query node. Since there are a limited number of tasks which can be running at once due to hardware limitations, we must prioritize their execution, and remove tasks which we know are no longer necessary. Therefore,

1. tasks for relaying newly derived information using segments to the right are executed before those to the left, and
2. once a node is known to be true or false, all tasks attempting to derive it (left of it in the graph) are canceled, as long as their results are not needed elsewhere.

Together, these two heuristics ensure that messages reach the query as quickly as possible, and time is not wasted deriving unnecessary formulas. The priorities of the messages (and hence, tasks) allow us to reach these goals. All UNASSERT messages have the highest priority, followed by all CANCEL-INFER messages. Then come I-INFER and U-INFER messages. BACKWARD-INFER messages have the lowest priority. As I-INFER and U-INFER messages flow to the right, they get higher priority, but their priorities remain lower than that of CANCEL-INFER messages. In forward inference, I-INFER and U-INFER messages to the right in the graph always have higher priority than those to the left, since the messages all begin flowing from a common point. In backward inference, the priorities of BACKWARD-INFER, and I-INFER, and U-INFER messages work together to derive a query formula as quickly as possible: since BACKWARD-INFER messages are of the lowest priority, those I-INFER and U-INFER messages waiting at valves which are nearest to the query formula begin flowing forward before valves further away are opened. This, combined with the increasing priority of I-INFER and U-INFER messages ensure efficient derivation. In short, the closest possible path to the query formula is always attempted first in backward inference.

The usefulness of CANCEL-INFER can be seen if you consider what would happen in the example in Figure 3 if `e` were also asserted. Remember that the CANCEL-INFER messages close valves in channels they passes through, and are passed backward further when a node has received the same number of cancellation messages as it has outgoing channels. In this example, backward inference messages would reach `wft2`, then `d` and `e`. The message that `e` is asserted would flow forward through `e`'s i-channel to `wft2`, which would in turn both send a message resulting in the assertion of `f`, and cancel inference going on further left in the graph, cutting off `wft1`'s inference since it is now unnecessary.

The design of the system therefore ensures that the tasks executing at any time are the ones closest to deriving the goal, and tasks which will not result in useful information towards deriving the goal are cleaned up. Additionally, since

[6]This can actually reduce performance by causing introduction rules to fire more than once. Consider two I-INFER messages arriving at an unasserted rule node nearly simultaneously. Assume the first message triggers the rule's introduction. The rule requests, asynchronously, to be asserted, then sends appropriate messages through the its i- and u-channels. The second message arrives, but the node's assertion request has not completed yet, causing duplication of work. This can happen (rarely) in the current concurrency model, but is far more likely when it's uncertain when the actual assertion will occur.

nodes "push" messages forward through the graph instead of "pulling" from other nodes, it is not possible to have tasks running waiting for the results of other rule nodes' tasks. Thus, deadlocks are impossible, and bottlenecks can only occur when multiple threads are making additions to shared state simultaneously.

5 Evaluation

The massively parallel logical inference systems of the 1980s and 90s often assigned each processor a single formula or rule to be concerned with. This resulted in limits on the size of the KB (bounded by the number of processors), and many processors sitting idle during any given inference process. Our technique dynamically assigns tasks to threads only when they have work to do, meaning that processors are not sitting idle so long as there are as many tasks available as there are processors.

In evaluating the performance of the inference graph, we are mostly concerned with the speedup achieved as more processors are used in inference. While overall processing time is also important, if speedup is roughly linear with the number of processors, that will show that the architecture and heuristics discussed scale well. We will look at the performance of our system in both backward and forward inference. The other two types of inference – bi-directional inference, and focused reasoning – are hybrids of forward and backward inference, and have performance characteristics between the two.

5.1 Backward Inference

To evaluate the performance of the inference graph in backward inference, we generated graphs of chaining entailments. Each entailment had bf antecedents, where bf is the branching factor, and a single consequent. Each consequent was the consequent of exactly one rule, and each antecedent was the consequent of another rule, up to a depth of d entailment rules. Exactly one consequent, cq, was not the antecedent of another rule. Therefore there were bf^d entailment rules, and $2 * bf^d - 1$ antecedents/consequents. Each of the bf^d leaf nodes were asserted. We tested the ability of the system to backchain on and derive cq when the entailments used were both and-entailment and or-entailment. Backward inference is the most resource intensive type of inference the inference graphs can perform, and most fully utilizes the scheduling heuristics developed in this paper.

In the first test we used and-entailment, meaning for each implication to derive its consequent both its antecedents had to be true. Since we backchained on cq, this meant every node in the graph would have to become asserted. This is the worst case scenario for entailment. The timings we observed are presented in Table 1.[7]

In increasing the number of usable CPUs from 1 to 2, we achieve nearly double the performance. As we increase further, there is still a benefit, but the advantage begins to drop off. The primary reason for this is that 1023 formulas must be asserted to the KB, and this requires maintenance of shared state. Only one thread can modify the shared state at a time, and so we handle it synchronously (as explained in Section 4). We found 100 iterations of asserting 1023 formulas which had already been built, as in the case of this test, took approximately 7500ms, regardless of the number of CPUs used. Excluding this from each of the times in Table 1 reveals a close-to-halving trend every time the number of CPUs is doubled, as would be expected (See Table 2).

CPUs	Inference Time (ms)	Speedup
1	37822	1.00
2	23101	1.64
4	15871	2.39
8	11818	3.20

Table 1: Inference times using 1, 2, 4, and 8 CPUs for 100 iterations of and-entailment in an inference graph with $d = 10$ and $bf = 2$ in which all 1023 rule nodes must be used to infer the result.

CPUs	Inference−Assert Time (ms)	Speedup
1	30322	1.00
2	15601	1.94
4	8371	3.62
8	4318	7.02

Table 2: The results from Table 1, excluding the time (7500ms) for assertions.

We then tried to determine whether the depth or branching factor of the graph has any effect on speedup as the number of processors increases. We first ran an experiment to judge the impact of graph depth. We ran the experiment on graphs of five different depths, ranging from 5 to 15 (32 leaves, to 32,768 leaves), while maintaining $bf = 2$ (see Figure 5), and found that as graph depth is increased, speedup increases very slowly. Since this increase must be bounded (we have no reason to believe a more-than-doubling speedup is possible), we have fit logarithmic trendlines to the 2, 4, and 8 CPU data, and found R^2 values to suggest a strong fit.

To find out if branching factor affects speedup, we chose $d = 7$ (128 leaves, 127 rule nodes), and varied the branching factor from 1 to 4. When $bf = 1$, the graph is simply a chain of nodes, and the use of more processors can provide no possible improvement in computation time. In fact, as shown in Figure 6, throwing more processors at the problem makes things worse. Fortunately, this is a rather contrived use of the inference graph. At branching factors 2-4, the graph performs as expected, with the branching factor increase having little effect on performance.[8] There may be a slight performance impact as the branching factor increases, because the RUI computations happening in the rule nodes rely on shared

[7] All tests were performed on a Dell Poweredge 1950 server with dual quad-core Intel Xeon X5365 processors (no Hyper-Threading) and 32GB RAM. Each test was performed twice, with the second result being the one used here. The first run was only to allow the JVM to "warm up."

[8] Because of the graph size explosion as branching factor increases, it was not possible to collect enough data to perform a quantitative analysis, only a qualitative one.

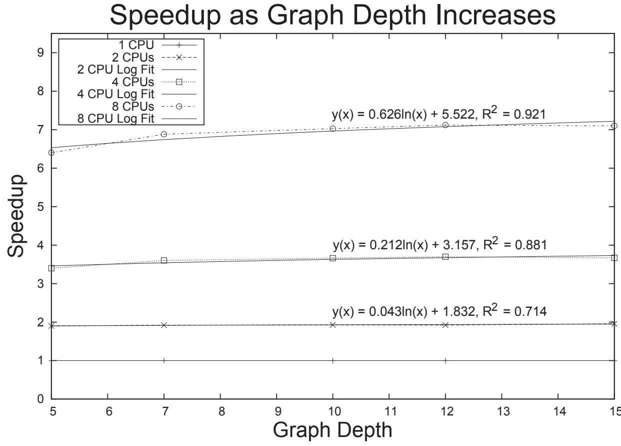

Figure 5: Speedup of the and-entailment test, shown in relation to the depth of the inference graph. As the depth of the graph increases, speedup increases slowly but logarithmically with each number of CPUs tested. A branching factor of 2 was used in all tests.

state, but that performance hit only occurs when two tasks attempt to modify the RUI set of a single node simultaneously – an increasingly unlikely event as we consider larger graphs.

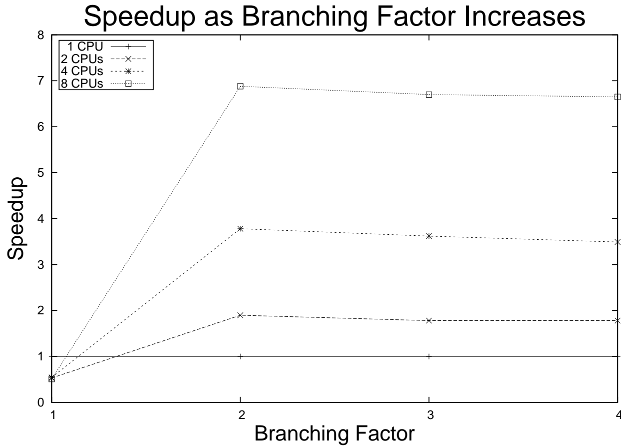

Figure 6: Speedup of the and-entailment test, shown in relation to the branching factor of the inference graph. A depth of 7 was used in all tests, to keep the number of nodes reasonable.

In our second test we used the same KB from the first ($d = 10$, $bf = 2$), except each and-entailment rule was swapped for or-entailment. Whereas the earlier test required every consequent in the KB to be derived, this test shows the best case of entailment - only a single leaf must be found to be asserted to allow the chaining causing cq to become true.

The improvement in or-entailment processing times as we add more CPUs (see Table 3) is not as dramatic since the inference operation performed once a chain of valves from an asserted formula to cq are open cannot be accelerated by

CPUs	Time (ms)	Avg. Rules Fired	Speedup
1	1021	10	1.00
2	657	19	1.55
4	498	38	2.05
8	525	67	1.94

Table 3: Inference times, and number of rule nodes used, using 1, 2, 4, and 8 CPUs for 100 iterations of or-entailment in an inference graph ($d = 10$, $bf = 2$) in which there are many paths through the network (all of length 10) which could be used to infer the result.

adding more processing cores – that process is inherently sequential. By timing forward inference through the graph, we found that approximately 340ms in each of the times in Table 3 is attributed to the relevant inference task. The improvement we see increasing from 1 to 2, and 2 to 4 CPUs is because the BACKWARD-INFER and CANCEL-INFER messages spread throughout the network and can be sped up through concurrency. During the periods where backward inference has already begun, and CANCEL-INFER messages are not being sent, the extra CPUs were working on deriving formulas relevant to the current query, but in the end unnecessary – as seen in the number of rule nodes fired in Table 3. The time required to assert these extra derivations begins to outpace the improvement gained through concurrency when we reach 8 CPUs in this task. These extra derivations may be used in future inference tasks, though, without re-derivation, so the resources are not wasted. As only a chain of rules are required, altering branching factor or depth has no effect on speedup.

The difference in computation times between the or-entailment and and-entailment experiments are largely due to the scheduling heuristics described in Section 4.1. Without the scheduling heuristics, backward inference tasks continue to get executed even once messages start flowing forward from the leaves. Additionally, more rule nodes fire than is necessary, even in the single processor case. We ran the or-entailment test again without these heuristics using a FIFO queue, and found the inference took just as long as in Table 1. We then tested a LIFO queue since it has some of the characteristics of our prioritization scheme (see Table 4), and found our prioritization scheme to be nearly 10x faster.

CPUs	Time (ms)	Avg. Rules Fired	Speedup
1	12722	14	1.00
2	7327	35	1.77
4	4965	54	2.56
8	3628	103	3.51

Table 4: The same experiment as Table 3 replacing the improvements discussed in Section 4.1, with a LIFO queue. Our results in Table 3 are nearly 10x faster.

5.2 Forward Inference

To evaluate the performance of the inference graph in forward inference, we again generated graphs of chaining entailments. Each entailment had a single antecedent, and 2 consequents.

Each consequent was the consequent of exactly one rule, and each antecedent was the consequent of another rule, up to a depth of 10 entailment rules. Exactly one antecedent, ant, the "root", was not the consequent of another rule. There were 1024 consequents which were not antecedents of other rules, the leaves. We tested the ability of the system to derive the leaves when ant was asserted with forward inference.

Since all inference in our graphs is essentially forward inference (modulo additional message passing to manage the valves), and we're deriving every leaf node, we expect to see similar results to the and-entailment case of backward inference, and we do, as shown in Table 5. The similarity between these results and Table 2 shows the relatively small impact of sending BACKWARD-INFER messages, and dealing with the shared state in the RUIs. Excluding the assertion time as discussed earlier, we again show a near doubling of speedup as more processors are added to the system. In fact, altering the branching factor and depth also result in speedups very similar to those from Figures 5 and 6.

CPUs	Inference−Assert Time (ms)	Speedup
1	30548	1.00
2	15821	1.93
4	8234	3.71
8	4123	7.41

Table 5: Inference times using 1, 2, 4, and 8 CPUs for 100 iterations of forward inference in an inference graph of depth 10 and branching factor 2 in which all 1024 leaf nodes are derived. Each result excludes 7500ms for assertions, as discussed in Section 5.1.

6 Conclusions

Inference graphs are an extension of propositional graphs capable of performing natural deduction using forward, backward, bi-directional, and focused reasoning within a concurrent processing system. Inference graphs add channels to propositional graphs, built along the already existing edges. Channels carry prioritized messages through the graph for performing and controlling inference. The priorities of messages influence the order in which tasks are executed – ensuring that the inference tasks which can derive the user's query most quickly are executed first, and irrelevant inference tasks are canceled. The heuristics developed in this paper for prioritizing and scheduling the execution of inference tasks improve performance in backward inference with or-entailment nearly 10x over just using LIFO queues, and 20-40x over FIFO queues. In and-entailment using backward inference, and in forward inference, our system shows a near linear performance improvement with the number of processors (ignoring the intrinsically sequential portions of inference), regardless of the depth or branching factor of the graph.

7 Acknowledgments

This work has been supported by a Multidisciplinary University Research Initiative (MURI) grant (Number W911NF-09-1-0392) for Unified Research on Network-based Hard/Soft Information Fusion, issued by the US Army Research Office (ARO) under the program management of Dr. John Lavery. We gratefully appreciate this support.

References

[Choi and Shapiro, 1992] Joongmin Choi and Stuart C. Shapiro. Efficient implementation of non-standard connectives and quantifiers in deductive reasoning systems. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, pages 381–390. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[de Kleer, 1986] Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28(2):197–224, 1986.

[Dixon and de Kleer, 1988] Michael Dixon and Johan de Kleer. Massively parallel assumption-based truth maintenance. In *Non-Monotonic Reasoning*, pages 131–142. Springer-Verlag, 1988.

[Doyle, 1979] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 19:231–272, 1979.

[Forgy, 1982] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[Hickey, 2008] Rich Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. ACM New York, NY, USA, 2008.

[Huang *et al.*, 2011] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1213–1216, New York, NY, USA, 2011. ACM.

[Lehmann, 1992] Fritz Lehmann, editor. *Semantic Networks in Artificial Intelligence*. Pergamon Press, Oxford, 1992.

[Lendaris, 1988] George G. Lendaris. Representing conceptual graphs for parallel processing. In *Conceptual Graphs Workshop*, 1988.

[Martins and Shapiro, 1988] João P. Martins and Stuart C. Shapiro. A model for belief revision. *Artificial Intelligence*, 35:25–79, 1988.

[McAllester, 1990] David McAllester. Truth maintenance. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI '90)*, pages 1109–1116, Boston, MA, 1990.

[McKay and Shapiro, 1981] Donald P. McKay and Stuart C. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 368–374, Los Altos, CA, 1981. Morgan Kaufmann.

[Schlegel and Shapiro, 2013] Daniel R. Schlegel and Stuart C. Shapiro. Concurrent reasoning with inference graphs (student abstract). In *Proceedings of the Twenty-Seventh AAAI Conference (AAAI-13)*, 2013. (In Press).

[Schlegel, 2013] Daniel R. Schlegel. Concurrent inference graphs (doctoral consortium abstract). In *Proceedings of*

the Twenty-Seventh AAAI Conference (AAAI-13), 2013. (In Press).

[Shapiro and Rapaport, 1992] Stuart C. Shapiro and William J. Rapaport. The SNePS family. *Computers & Mathematics with Applications*, 23(2–5):243–275, January–March 1992. Reprinted in [Lehmann, 1992, pp. 243–275].

[Shapiro *et al.*, 1982] Stuart C. Shapiro, João P. Martins, and Donald P. McKay. Bi-directional inference. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pages 90–93, Ann Arbor, MI, 1982. The Program in Cognitive Science of The University of Chicago and The University of Michigan.

[Shapiro, 1989] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, September 1989.

[Shapiro, 2010] Stuart C. Shapiro. Set-oriented logical connectives: Syntax and semantics. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (KR2010)*, pages 593–595, Menlo Park, CA, 2010. AAAI Press.

[The Joint Task Force on Computing Curricula *et al.*, 2013] The Joint Task Force on Computing Curricula, Association for Computing Machinery, and IEEE-Computer Society. *Computer Science Curricula 2013*. 2013.

[Wachter and Haenni, 2006] Michael Wachter and Rolf Haenni. Propositional DAGs: a new graph-based language for representing boolean functions. In *KR06, 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 277–285. U.K., AAAI Press, 2006.

[Yan *et al.*, 2009] F. Yan, N. Xu, and Y. Qi. Parallel inference for latent dirichlet allocation on graphics processing units. In *Proceedings of NIPS*, pages 2134–2142, 2009.