Concurrent Reasoning with Inference Graphs

Daniel R. Schlegel and Stuart C. Shapiro

Department of Computer Science and Engineering
University at Buffalo, Buffalo NY 14260, USA
{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 [3] 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 [10]. 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 [17], 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 [7]), 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 [24].

We present *inference graphs* [16], 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 [21]. 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 [19], and focused reasoning in addition to the backward inference used in resolution. Natural deduction also allows

M. Croitoru et al. (Eds.): GKR 2013, LNAI 8323, pp. 138–164, 2013.

formulas generated during inference to be retained in the KB for later re-use, whereas refutation techniques always start by assuming the negation of the formula to be derived, making intermediate derivations useless for later reasoning tasks. 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 [23] which allow queries to be performed upon a graph, not within it. More similar to our approach is that of Truth Maintenance Systems [4] (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 [4,8,12], RETE networks [5], and Active Connection Graphs [13], and implemented in Clojure [6], 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 implements introduction and elimination rules to determine if itself or its consequents are true or negated.

In Sect. 2 we review propositional graphs and introduce an initial example, followed by our introduction to inference graphs in Sect. 3. Section 4 explains how the inference graphs are implemented in a concurrent processing system. In Sect. 5 we present a second illustrative example. We evaluate the implementation in Sect. 6 and finally conclude with Sect. 7.

2 Propositional Graphs

Propositional graphs in the tradition of the SNePS family [20] are graphs in which every well-formed expression in the knowledge base, including individual constants, functional terms, atomic formulas, and 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] An exclamation mark, "`!`", is appended to the label if the proposition is true in the KB. No two nodes represent syntactically identical expressions; rather, if there are

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

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 logic which do not include variables or quantifiers.[2] Specifically, we have implemented introduction and elimination rules for the set-oriented connectives `andor`, and `thresh` [18], and the elimination rules for numerical entailment. The `andor` connective, written (`andor` $(i\ j)\ p_1 \ldots p_n$), $0 \leq i \leq j \leq n$, is true when at least i and at most j of $p_1 \ldots p_n$ are true (that is, an `andor` may be introduced when those conditions are met). It generalizes `and` $(i = j = n)$, `or` $(i = 1, j = n)$, `nand` $(i = 0, j = n-1)$, `nor` $(i = j = 0, n > 1)$, `xor` $(i = j = 1)$, and `not` $(i = j = 0, n = 1)$. For the purposes of `andor`-elimination, each of $p_1 \ldots p_n$ may be treated as an antecedent or a consequent, since when any j formulas in $p_1 \ldots p_n$ are known to be true (the antecedents), the remaining formulas (the consequents) can be inferred to be negated, and when any $n - i$ arguments are known to be false, the remaining arguments can be inferred to be true. For example with `xor`, 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`, and written (`thresh` $(i\ j)\ p_1 \ldots p_n$), $0 \leq i \leq j \leq n$, is true when either fewer than i or more than j of $p_1 \ldots p_n$ are true. The thresh connective is mainly used for equivalence (`iff`), when $i = 1$ and $j = n - 1$. As with `andor`, for the purposes of `thresh`-elimination, each of $p_1 \ldots p_n$ may be treated as an antecedent or a consequent. Numerical entailment is a generalized entailment connective, written `(=>` i `(setof` $a_1 \ldots a_n$`) (setof` $c_1 \ldots c_m$`))` meaning if at least i of the antecedents, $a_1 \ldots a_n$, are true then all of the consequents, $c_1 \ldots c_m$, are true. The initial example 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 $i = n$ [20].

In our initial example we will use two rules, one using an and-entailment and one using an or-entailment: `(if (setof a b c) d)`, meaning that whenever `a`, `b`, and `c` are true then `d` is true; and `(v=> (setof d e) f)`, meaning that whenever `d` or `e` is true then `f` is true. In Fig. 1 `wft1` represents the and-entailment `(if (setof a b c) 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 `(v=> (setof d e) 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 (and one is shown in Section 5), we will use this rather simplistic one in our initial example 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.

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

[2] The system has been designed to be extended to a logic with quantified variables in the future.

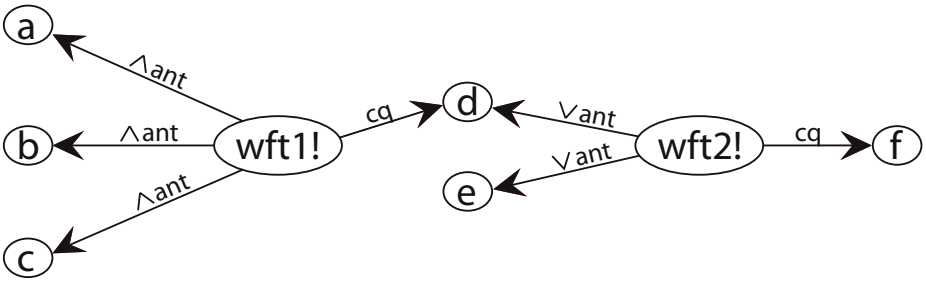

Fig. 1. Propositional graph for the assertions that if a, b, and c are true, then d is true, and if d or e are true, then f is true

cq arcs backward 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. Channels 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 Fig. 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. Often we will speak of a channel being open or closed, where that is intended to mean that the valve in the channel is open or closed.

[3] u-channels and u-infer messages were previously called y-channels and Y-INFER messages in [14,16].

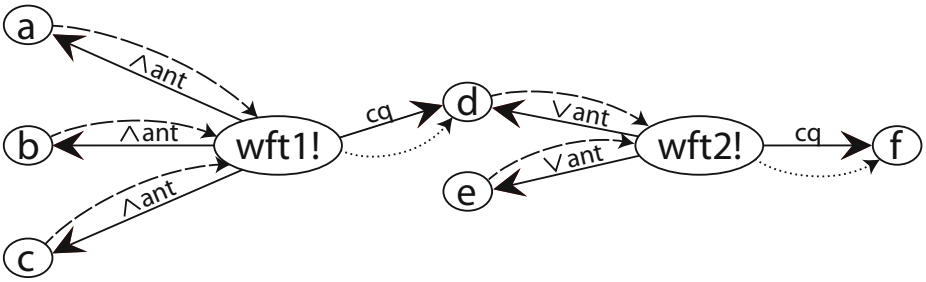

Fig. 2. The same propositional graph from Fig. 1 with the appropriate channels added. Channels represented by dashed lines are i-channels and are drawn from antecedents to rule nodes. Channels represented by dotted lines are u-channels and are drawn from rule nodes to consequents.

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 backward 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:

$$< origin, support, type, assertStatus?, fwdInfer?, priority >$$

where $origin$ 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 [8,11]); $type$ is the type of the message (such as `i-infer` or `backward-infer`, described in detail in the next several subsections); $fwdInfer?$ states whether this message is part of a forward-inference task; $assertStatus?$ 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 Sect. 4). We define the different types of

messages below, while the actions the receipt of those messages cause are discussed in Secs. 3.2 and 3.3.

i-infer. When a node is found to be true or false, an `i-infer` message reflecting this new assertional status is submitted to its outgoing i-channels. These messages are sent from antecedents to rule nodes. An `i-infer` message contains a support set which consists of 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. Rule nodes which have just learned enough about their antecedents to fire send `u-infer` messages to each of their consequents, informing them of what their 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. When it is necessary for the system to determine whether a node can be derived, `backward-infer` messages are used to open channels to rules that node may be the consequent of, or, if the node is a rule node, to open channels of antecedents to derive the rule. These messages set up a backward inference operation by passing backward through channels and opening any valves they reach. In a backward inference operation, these messages are generally sent recursively backward through the network until valves are reached which have waiting `i-infer` or `u-infer` messages. 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 to know the assertional status of some of its antecedents.[5] `cancel-infer` messages are sent from some node backward through the graph. These messages are generally sent recursively backward through the network to prevent unnecessary inference. 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 task.

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

[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.

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

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 an `i-infer` message arrives at a rule node the message is translated into *Rule Use Information*, or RUI [2]. 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. When a rule fires, new `u-infer` messages are sent to the rule's consequents, informing them whether they should be asserted or negated.

Figure 3 shows the process of deriving `f` in our initial example. We assume backward inference has been initiated, opening all the valves in the graph. First, in Fig. 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 Fig. 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 (Fig. 3c). In Fig. 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 Fig. 3e, `f` is asserted, and inference is complete.

3.3 Inference Segments

The inference graph is divided into *inference segments* (henceforth, *segments*). A segment represents the operation – from receipt of a message to sending new ones – which occurs in a node. Valves delimit segments, as seen in Fig. 4. The operation of a node

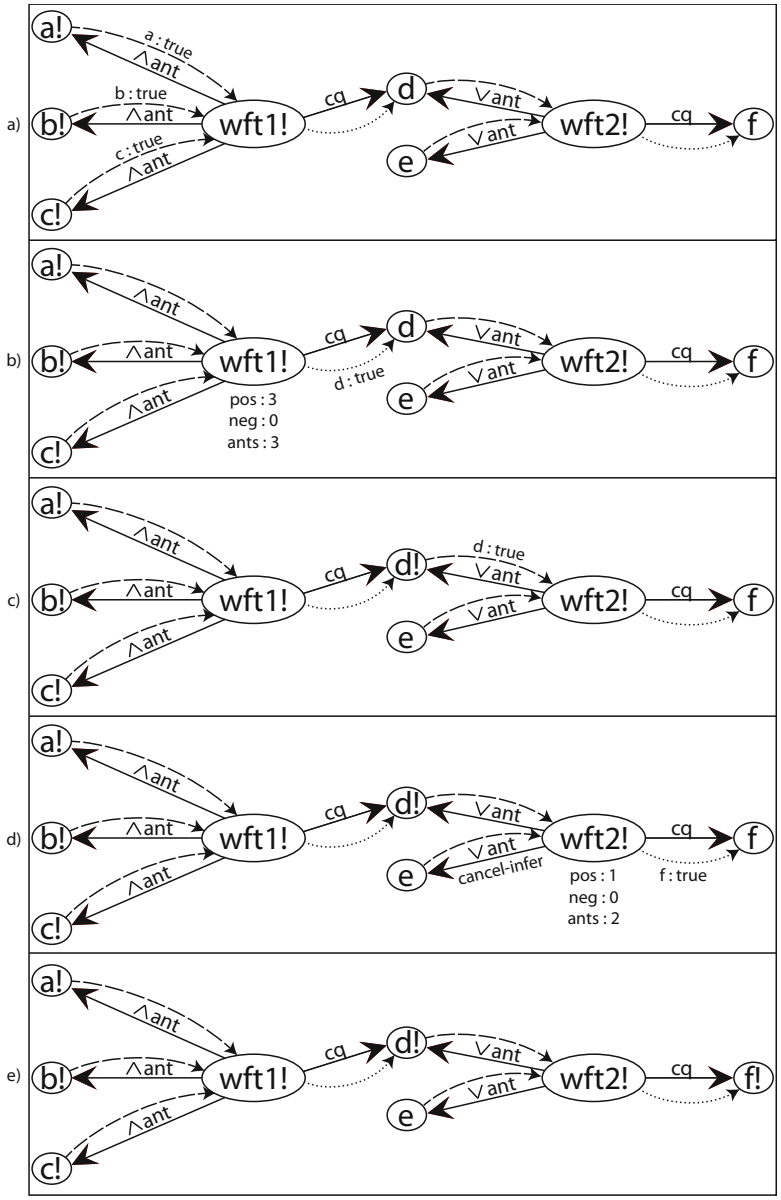

Fig. 3. a) Messages are passed from `a`, `b`, and `c` to `wft1`. b) `wft1` combines the messages from `a`, `b`, and `c` to find that it has 3 positive antecedents, of a total of 3. The and-entailment can fire, so it sends a message through its u-channel informing its consequent, `d`, that it has been 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.

is different depending on the type of message that node is currently processing. The collection of operations performed within an inference segment is what we call the segment's *inference function*.

Only a rule node can ever receive an `i-infer` message, since `i-infer` messages only flow across i-channels, built from rule antecedents to the rule itself. When a rule node receives an `i-infer` message, the rule performs the operation discussed in Sec. 3.2. On the other hand, any node may receive a `u-infer` message. When a `u-infer` message is processed by a node, the node asserts itself to be true or negated (depending on the content of the `u-infer` message), and sends new `i-infer` messages through any of its outgoing i-channels to inform any rule nodes the node is an antecedent of of it's new assertional status.

When a `backward-infer` message is processed by a node, all of that node's incoming channels which do not originate at the source of the `backward-infer` message are opened. This act causes messages waiting at the now-opened valves to flow forward. In addition, `backward-infer` messages are sent backward along each of the newly-opened channels which did not have `i-` or `u-infer` messages waiting at their valves.

Messages which cancel inference – `cancel-infer` messages – are processed somewhat similarly to `backward-infer` messages, but with the opposite effect. All of the node's incoming channels which do not originate at the source of the `cancel-infer` message are closed (if they're not already), as long as the node's outgoing channels are all closed. `cancel-infer` messages are sent backward along each of the newly-closed channels.

The last type of message, `unassert` messages, are processed by unasserting the node, and sending new `unassert` messages along all outgoing channels which terminate at nodes which either have the unasserted formula in each of their support sets, or are the consequent of a rule which no longer has the appropriate number of positive or negative antecedents. `unassert` messages do not wait at valves, since they are of critical importance to maintaining the consistency of the KB.

It's important to note that none of these operations ever requires a node to build new channels. Channels are built when nodes are added to the graph, and all such channels which can be built are built at that time. When a node is made true or negated by a user, `i-infer` messages are automatically sent out each of its i-channels. If a new i-channel is created by the addition of a new rule to the KB, an appropriate `i-infer` message is automatically submitted to that channel if the origin is true or negated.

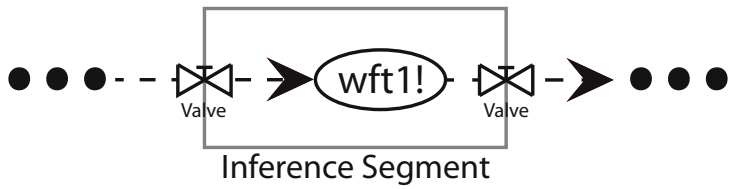

Fig. 4. A single inference segment is shown in the gray bounding box

4 Concurrent Reasoning

The inference graph's structure lends itself naturally to concurrent inference. It is clear in our initial 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, we aim to execute the inference functions of multiple segments at the same time, and do so in an efficient manner through the use of scheduling heuristics. A *task* is the application of a segment's inference function to a message. Tasks are created whenever a message passes the boundary between segments. For `i-infer` and `u-infer` messages, that's when they pass through a valve, or for `unassert` messages when they ignore a valve. For the `backward-infer` and `cancel-infer` messages it's when they cross a valve backward. When tasks are created they enter a global prioritized queue, where the priority of the task is the priority of the message. The task which causes a message to cross between segments is responsible for creating the new task for that message and adding it to the queue. When the task is executed, the appropriate operation is performed as described above, and any newly generated messages allow the process to repeat.

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

[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.

at once due to hardware limitations, we must prioritize their execution, remove tasks which we know are no longer necessary, and prevent the creation of unnecessary tasks. Therefore,

1. tasks for relaying newly derived information using segments to the right are executed before those to the left,
2. once a node is known to be true or false, all tasks still attempting to derive it are canceled, as long as their results are not needed elsewhere, and all channels pointing to it which may still derive it are closed.
3. once a rule fires, all tasks for potential antecedents of that rule still attempting to satisfy it are canceled, as long as their results are not needed elsewhere, and all channels from antecedents which may still satisfy it are closed.

Together, these three 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 Fig. 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 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 An Illustrative Example

The initial example we have discussed thus far was useful for gaining an initial understanding of the inference graphs, but it is not sophisticated enough to show several interesting properties of the graphs. The following example shows how the inference graph can be used for rule introduction, deriving negations, the use of ground predicates, and the set-oriented logical connectives such as `andor`, `xor`, and `iff` discussed in Sect. 2.

Inspired by L. Frank Baum's *The Wonderful Wizard of Oz* [1], we consider a scene in which Dorothy and her friends are being chased by Kalidas – monsters with the head of a tiger and the body of a bear. In the world of Oz, whenever Toto becomes scared, Dorothy carries him, and that's the only time she carries him. If Toto walks, he is not carried, and if he is carried, he does not walk. Toto becomes scared when Dorothy is being chased. Since Dorothy has only two hands, she is capable of either carrying the Scarecrow (who is large, and requires both hands), or between 1 and 2 of the following items: Toto, her full basket, and the Tin Woodman's oil can. In our example, the Tin Woodman is carrying his own oil can. Only one of Dorothy, the Scarecrow, or the Tin Woodman can carry the oil can.

The relevant parts of this scene are represented below in their logical forms.

```
;;; Dorothy can either carry the scarecrow,
;;;  or carry one or two objects from the list:
;;;   her full basket, Toto, oil can.
(xor (Carries Dorothy Scarecrow)
     (andor (1 2) (Carries Dorothy FullBasket)
                  (Carries Dorothy Toto)
                  (Carries Dorothy OilCan)))

;;; Either Dorothy, the Tin Woodman, or the Scarecrow
;;;    carry the Oil Can.
(xor (Carries Dorothy OilCan)
     (Carries TinWoodman OilCan)
     (Carries Scarecrow OilCan))

;;; Either Dorothy carries Toto, or Toto walks.
(xor (Carries Dorothy Toto) (Walks Toto))

;;; Dorothy carries Toto if and only if Toto is scared.
(iff (Scare Toto) (Carries Dorothy Toto))

;;; Toto gets scared if Dorothy is being chased.
(if (Chase Dorothy) (Scare Toto))

;;; The Tin Woodman is carrying his Oil Can.
(Carries TinWoodman OilCan)
```

```
;;; Dorothy is being chased.
(Chase Dorothy)
```

We can then wonder, "Is Dorothy carrying the Scarecrow?" According to the rules of these connectives, as discussed in Section 2, we should be able to derive that this is not the case. This can be derived by hand as follows:

```
1) Since (if (Chase Dorothy) (Scare Toto))
     and (Chase Dorothy),
   infer (Scare Toto)
                          by Implication Elimination.
2) Since (iff (Carries Dorothy Toto) (Scare Toto))
     and (Scare Toto)
   infer (Carries Dorothy Toto)
                          by Equivalence Elimination.
3) Since (xor (Carries Scarecrow OilCan)
              (Carries Dorothy OilCan)
              (Carries TinWoodman OilCan))
     and (Carries TinWoodman OilCan)
   infer (not (Carries Dorothy OilCan))
                                  by Xor Elimination.
4) Since (Carries Dorothy Toto)
     and (not (Carries Dorothy OilCan))
   infer (andor (1 2) (Carries Dorothy FullBasket)
                      (Carries Dorothy OilCan)
                      (Carries Dorothy Toto))
                               by Andor Introduction.
5) Since (xor
            (andor (1 2) (Carries Dorothy FullBasket)
                         (Carries Dorothy OilCan)
                         (Carries Dorothy Toto))
            (Carries Dorothy Scarecrow))
     and (andor (1 2) (Carries Dorothy FullBasket)
                      (Carries Dorothy OilCan)
                      (Carries Dorothy Toto))
   infer (not (Carries Dorothy Scarecrow))
                                    by Xor Elimination.
```

The inference graphs are able to reach the same conclusion by applying these same rules of inference. The inference graph for the above example is displayed in Fig. 5. Since this example makes use of ground predicate logic (instead of only proposition symbols, as used in the initial example), we have had to define the arc labels used to

identify the arguments of those formulas in the graph.[7] The `Carries` relation has two arguments, `carrier` and `carried`, where the `carrier` is the entity carrying the `carried` object. The `Walks` relation has only the `selfMover` argument – the entity doing the walking. Both the `Chase` and `Scare` relations have only one argument. The `Chase` relation has a `theme` – the one being chased – and the `Scare` relation has an `experiencer` – the one who becomes scared. In order to make it clear in the graph what the act the `theme` or `experiencer` is involved in, we add an additional arc labeled `act` pointing to a node whose symbol is the relation name.[8]

In Fig. 5, `wft6` represents the proposition that either Dorothy carries the Scarecrow (`wft1`), or `wft5`, that Dorothy carries between 1 and 2 of the items: the oil can (`wft2`), her full basket (`wft3`), and Toto (`wft4`). `wft2`, `wft8` and ,`wft9` respectively represent the propositions that Dorothy, the Scarecrow, and the Tin Woodman carry the oil can, and `wft10` is the exclusive disjunction of those formulas. The exclusive-or rule node `wft14` represents the proposition that either Toto walks (`wft13`) or Toto is carried by Dorothy (`wft4`). `wft12` expresses that Dorothy carries Toto (`wft4`) if and only if Toto is scared (`wft11`). Finally, the relation expressing that if Dorothy is being chased (`wft15`), then Toto is scared (`wft13`) is represented by `wft16`. Notice that, as indicated by the appended "`!`", `wft6`, `wft9`, `wft10`, `wft12`, `wft14`, `wft15`, and `wft16`, are asserted, but none of the other wfts are.

Channels have been drawn on the graph as described earlier. Since `andor` and `thresh` do not have pre-defined antecedents and consequents (as the various types of entailment do), each of the arguments in one of these rules must, in the inference graph, have an i-channel drawn from the argument to the rule, and a u-channel drawn from the rule to the argument. This way each argument can inform the rule when it has a new assertional status, and the rule can inform each argument about a new assertional status it should adopt based on the rule firing.

For the purposes of this example, we will make two assumptions: first that there are two processors being used (*a* and *b*), and second that any two tasks which begin on the two CPUs simultaneously, end at the same time as well. Figure 6 shows the first set of processing steps used in the derivation. Processing steps in this figure are labeled one through five, with "a" and "b" appended to the label where necessary to denote the CPU in use for ease of reference to the diagram. The step labels are placed at the nodes, since a task stretches from valve-to-valve, encompassing a single node. We'll discuss the inference process as if the nodes themselves are added to the task queue for easier reading, when what we really mean is that tasks created for the inference process of a node, applied to a message, are added to the task queue.

The steps illustrated in Fig. 6 consist mostly of backward inference. The backward inference begins at the query, `wft1`, and continues until some channel is opened which contains an `i-infer` or `u-infer` message. In this example, this happens first at `wft10`, in step 5b of the figure. Listed below are the details of the processing which

[7] A KR formalism can be seen simultaneously as graph-, frame-, and logic-based. In our system, we define caseframes which have a number of slots, corresponding to argument positions in the logical view. The names of those slots are what decide the arc labels. See [15] for more details.

[8] The relations used here are based upon entries in the Unified Verb Index [22].

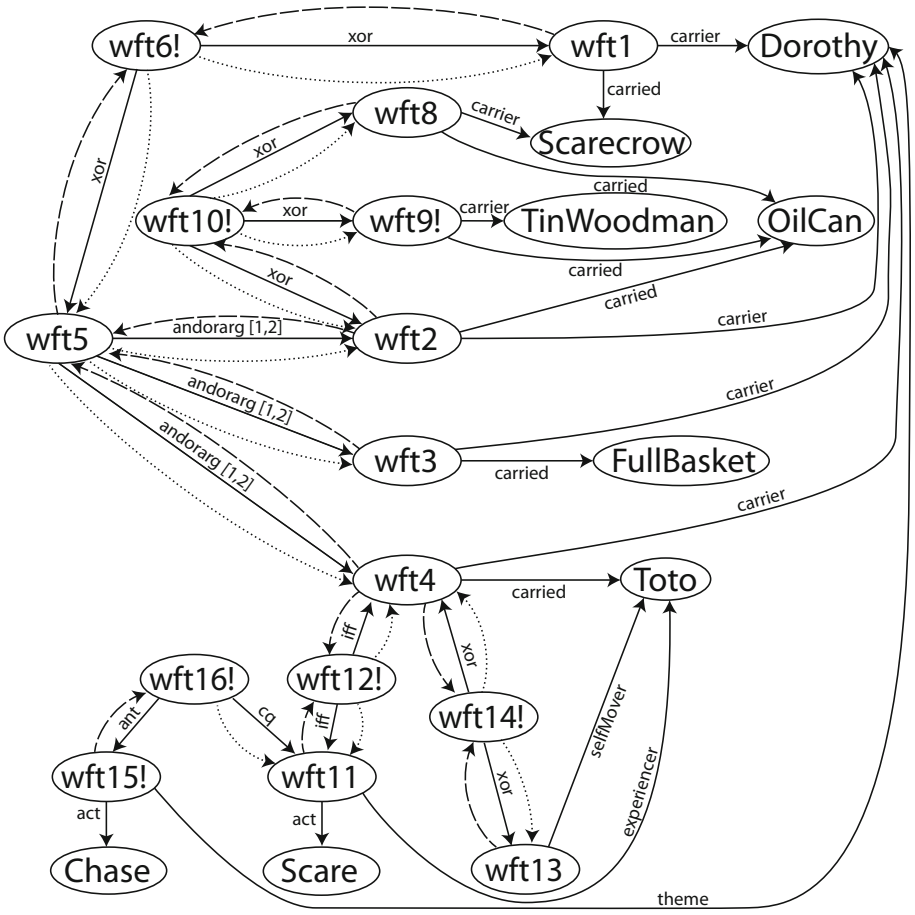

Fig. 5. The inference graph intended to mean that either Dorothy can carry the Scarecrow, or she can carry one or two of the following: Toto, her full basket, and the Tin Woodman's oil can; only one of the Scarecrow, the Tin Woodman, or Dorothy can carry the oil can; Dorothy carries Toto when, and only when, Toto is scared; Toto is either carried by Dorothy, or walks; if Dorothy is being chased, then Toto is scared; Dorothy is being chased; and the Tin Woodman is carrying his oil can

occurs during each step shown in the figure, along with the contents of the task queue. Tasks in the task queue are displayed in the following format:

$$\texttt{<wftSrc} - X \rightarrow \texttt{wftDest>}$$

where `wftSrc` is the source of the message which caused the creation of the task, `wftDest` is the node the task is operating within, and X is one of i,u,b, or c standing for the type of message the task processes, `i-infer`, `u-infer`, `backward-infer`, or `cancel-infer`. The `unassert` message type is not used in this example, and therefore was not listed.

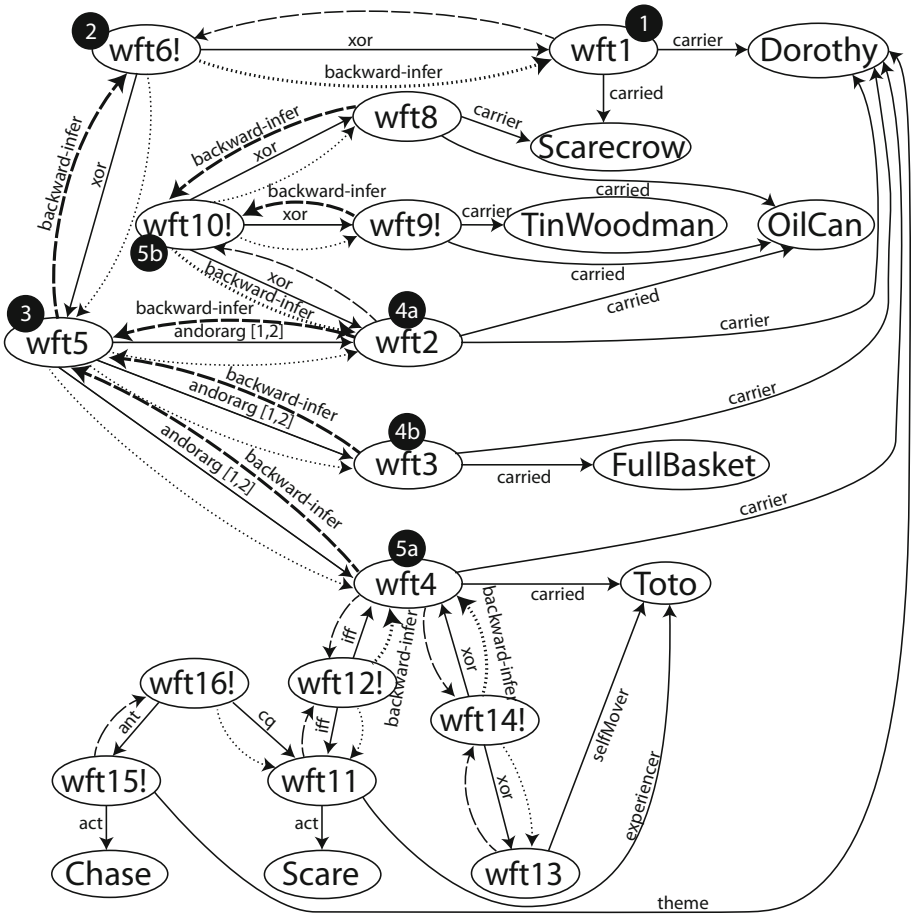

Fig. 6. The first five steps of inference when attempting to derive whether Dorothy is carrying the Scarecrow. Channels with a heavier weight and slightly larger arrows have had their channels opened through backward inference. Two processors are assumed to be used – a and b – and for this reason some steps in the graph have "a" or "b" appended to them. In these five steps, `backward-infer` messages flow backward through the graph until the first channel is reached with messages which will flow forward: the fact that `wft9` is asserted will flow to `wft10` since the i-channel connecting them has just been opened through backward inference.

1 `wft1` sends `backward-infer` message to `wft6!`;
opens the channel from `wft6!` to `wft1`.
task queue <wft1 $-b\rightarrow$ wft6!>
2 `wft6!` sends `backward-infer` message to `wft5`;
opens the channel from `wft5` to `wft6!`.
task queue <wft6! $-b\rightarrow$ wft5>
3 `wft5` sends `backward-infer` messages to `wft2`, `wft3`, and `wft4`;
opens the channels from `wft2`, `wft3`, and `wft4` to `wft5`.

task queue <wft5 $-b \rightarrow$ wft2>, <wft5 $-b \rightarrow$ wft3>, <wft5 $-b \rightarrow$ wft4>
4a wft2 sends backward-infer message to wft10!;
opens the channel from wft10! to wft4.
4b wft3 has no channels to open.
task queue <wft5 $-b \rightarrow$ wft4>, <wft2 $-b \rightarrow$ wft10!>
5a wft4 sends backward-infer messages to wft12! and wft14!;
opens the channels from wft12! and wft14! to wft4.
5b wft10! sends backward-infer messages to wft8 and wft9!;
opens the channels from wft8 and wft9! to wft10!.
Since wft9! is asserted, there is an i-infer message already waiting in the channel from wft9! to wft10! with higher priority than any backward inference tasks. That i-infer message is moved across the valve, and a new task is created for it – causing wft10! to be added to the front of the queue again. Since there was an i-infer message waiting at the opened valve from wft9! to wft10!, the backward-infer task just queued to occur in wft9! is canceled, as it is unnecessary.
task queue <wft9! $-i \rightarrow$ wft10!>, <wft4 $-b \rightarrow$ wft12!>,
<wft4 $-b \rightarrow$ wft14!>, <wft10! $-b \rightarrow$ wft8>

Remember that no backward-infer messages are sent to nodes which are already part of the derivation. For example, wft10 does not send a backward-infer message back to wft2 since wft2 is already part of the current derivation. This prevents eventual unnecessary derivations.

Figure 7 shows the next series of inference steps. In these steps the truth of wft9 is used to infer the negation of wft2, that Dorothy is not carrying the oil can (corresponding to step 3 in the earlier manual derivation), and relays this information to wft5. Backward inference continues from wft14 and wft12 back to wft13 and wft15, respectively. This is, again, the point where an i-infer message is ready to flow forward, this time from wft15 to wft16. Below we have once again described these processing steps in detail.

6a wft10! receives i-infer message from wft9!;
derives that both wft2 and wft8 are negated, by the rules of xor;
sends u-infer messages to both wft2 and wft8 telling them they are negated (of which, only the message to wft2 will pass through an open channel);
cancels any inference in progress or queued to derive wft8, since it is the only antecedent still attempting to satisfy wft10!.
6b wft12! sends backward-infer message to wft11;
opens the channel from wft11 to wft12!.
task queue <wft10! $-c \rightarrow$ wft8>, <wft10! $-u \rightarrow$ wft2>,
<wft4 $-b \rightarrow$ wft14!>, <wft12! $-b \rightarrow$ wft11>
7a wft8 has no channels to close.
7b wft2 receives u-infer message from wft10!;
asserts that it itself is negated (wft17!);
sends an i-infer message along the channel to wft5, telling wft5 that wft2 has been derived to be false.

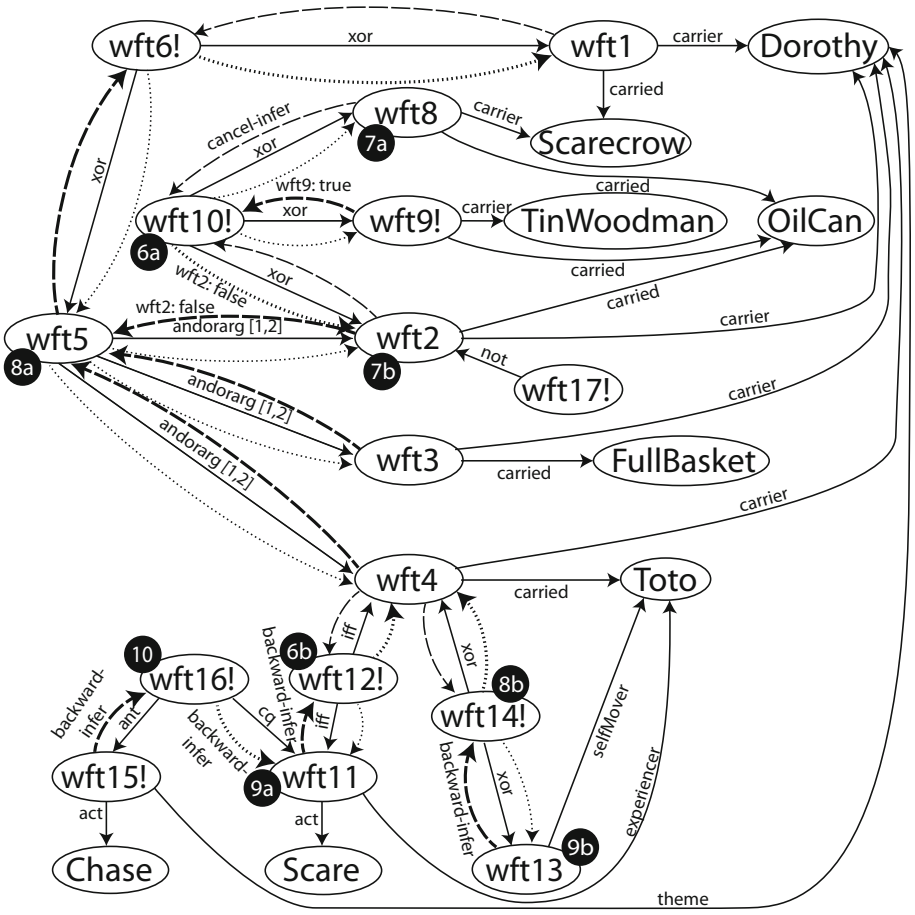

Fig. 7. Steps six through ten of the attempted derivation of whether Dorothy is carrying the Scarecrow. In steps 6b, 8b, 9a, 9b, and 10 backward inference is performed until the `i-infer` message indicating `wft15` is true might flow forward across it's i-channel to `wft16`. Additionally, `wft10` receives an `i-infer` message about the truth of `wft9` (step 6a), which derives that `wft2` is false through xor-elimination. `wft2` then reports this (step 7b) to `wft5`, which records this information (step 8a), but cannot yet do anything else.

task queue <wft2 $-i \rightarrow$ wft5>, <wft4 $-b \rightarrow$ wft14!>, <wft12! $-b \rightarrow$ wft11>

8a `wft5` receives `i-infer` message from (the negated) `wft2`. Since `wft5` requires more information to determine if between 1 and 2 of its arguments are true, no more can be done.

8b `wft14!` sends `backward-infer` message to `wft13`; opens the channel from `wft13` to `wft14!`.

task queue <wft12! $-b \rightarrow$ wft11>, <wft14! $-b \rightarrow$ wft13>

9a wft11 sends backward-infer message to wft16!;
opens the channel from wft16! to wft11.
9b wft13 has no channels to open.
task queue <wft11 $-b \rightarrow$ wft16!>
10 wft16! sends backward-infer messages to wft15!;
opens the channel from wft15! to wft16!. Since wft15! is asserted, there is an i-infer message waiting in the channel from wft15! to wft16!, which flows forward creating a new task to operate on wft16!, added at the front of the queue.
task queue <wft15! $-i \rightarrow$ wft16!>

Figure 8 illustrates the conclusion of the derivation that Dorothy is not carrying the Scarecrow. In this final series of inference steps, it is derived that Toto is scared (wft11, step 1 from the manual derivation), that Dorothy carries Toto (wft4, step 2 from the manual derivation), that Dorothy carries one or two of the items from the list of: her full basket, the oil can, and Toto (wft5, step 4 from the manual derivation), and that Dorothy does not carry the Scarecrow (wft18, step 5 from the manual derivation). The final set of inference steps are below.

11 wft16! receives i-infer message from wft15!;
derives wft11 by the rules of entailment elimination;
sends u-infer message to wft11 telling it that wft11 is true.
task queue <wft16! $-u \rightarrow$ wft11>
12 wft11 receives u-infer message from wft16!;
asserts itself;
sends i-infer message to wft12! telling it that wft11! has been derived.
task queue <wft11! $-i \rightarrow$ wft12!>
13 wft12! receives i-infer message from wft11!;
derives wft4 by the rules of equivalence elimination;
sends u-infer message to wft4 telling it that wft4 is true.
task queue <wft12! $-u \rightarrow$ wft4>
14 wft4 receives the u-infer message from wft12!;
asserts itself;
sends i-infer message to wft5 telling it that wft4! has been derived;
sends a cancel-infer message to wft14!.
task queue <wft4! $-c \rightarrow$ wft14!>, <wft4! $-i \rightarrow$ wft5>
15a wft14! receives cancel-infer message from wft4!;
sends a cancel-infer message to wft13.
15b wft5 receives the i-infer message from wft4!;
derives itself through andor introduction, since between 1 and 2 of its antecedents must be true;
sends an i-infer message to wft6!. sends a cancel-infer message to wft3.
task queue <wft5! $-c \rightarrow$ wft3>, <wft14! $-c \rightarrow$ wft13>,
<wft5! $-i \rightarrow$ wft6!>
16a wft3 has no channels to close.

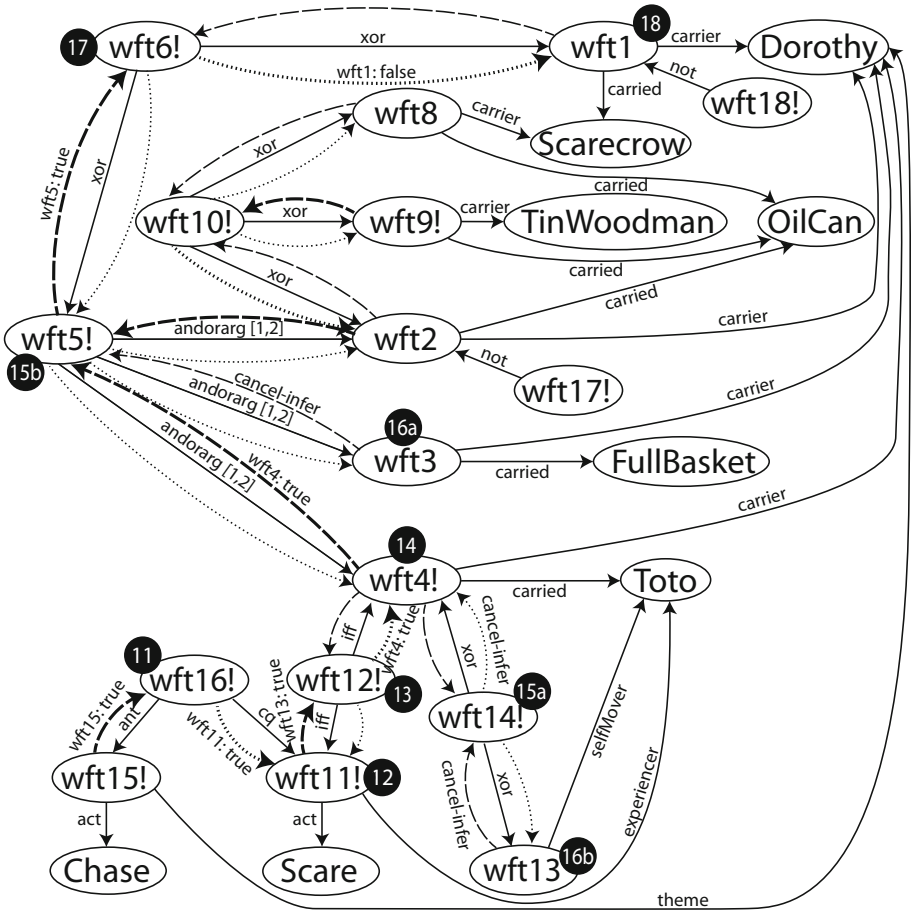

Fig. 8. The conclusion of the derivation that Dorothy is not, in fact, carrying the Scarecrow. Since `wft15` is true (the fact that Dorothy is being chased), it is reported to `wft16` (step 11), which fires and sends `u-infer` messages to its consequent – `wft11`. `wft11`, in step 12, reports its new truth value to `wft12` which fires (step 13), since it is an if-and-only-if, and sends a `u-infer` message to `wft4`. Since `wft4` is now true, it cancels other inference attempting to derive it and reports its truth to `wft5` (step 14). Simultaneously, `wft14` continues canceling inference (step 15a), and `wft5` is found to be true, since it required between 1 and 2 of its antecedents to hold, and that is now the case (step 15b). Unnecessary inference attempting to derive `wft5` and `wft14` are canceled in steps 16a and b. Now, in step 17, `wft6` receives the message that `wft5` is true (that is, Dorothy is carrying 1 or 2 items), and because it is exclusive or, it determines that `wft1` is false - Dorothy is not carrying the Scarecrow. Step 18 asserts this fact, and reports it to the user.

16b `wft13` has no channels to close.
task queue `<wft5! $-i\rightarrow$ wft6!>`
17 `wft6!` receives `i-infer` message from `wft5!`;
derives the negation of `wft1` by the rules of `xor` elimination;
sends `u-infer` message to `wft1` telling it that `wft1` is false.
task queue `<wft6! $-u\rightarrow$ wft1>`
18 `wft1` receives `u-infer` message from `wft6!`;
asserts the negation of itself (`wft18!`);
informs the user that the negation of the query is true.
task queue *empty*

Inference is now complete, and any channels which remain open can be closed.

6 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.

6.1 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

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

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.[9]

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 Sect. 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).

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

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

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 Fig. 9), 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

[9] 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."

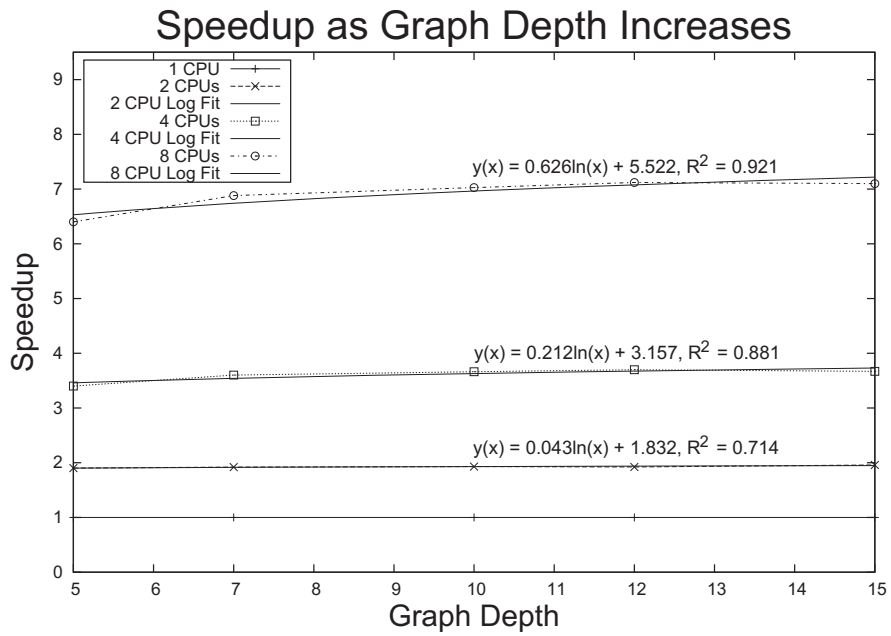

Fig. 9. Speedup of the and-entailment test, shown in relation to the depth of the inference graph. As the depth of the graph increases, speedup increases slowly but logarithmically with each number of CPUs tested. A branching factor of 2 was used in all tests.

a chain of nodes, and the use of more processors can provide no possible improvement in computation time. In fact, as shown in Fig. 10, 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.[10] There may be a slight performance impact as the branching factor increases, because the RUI computations happening in the rule nodes rely on shared 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.

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 adding more processing

[10] 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.

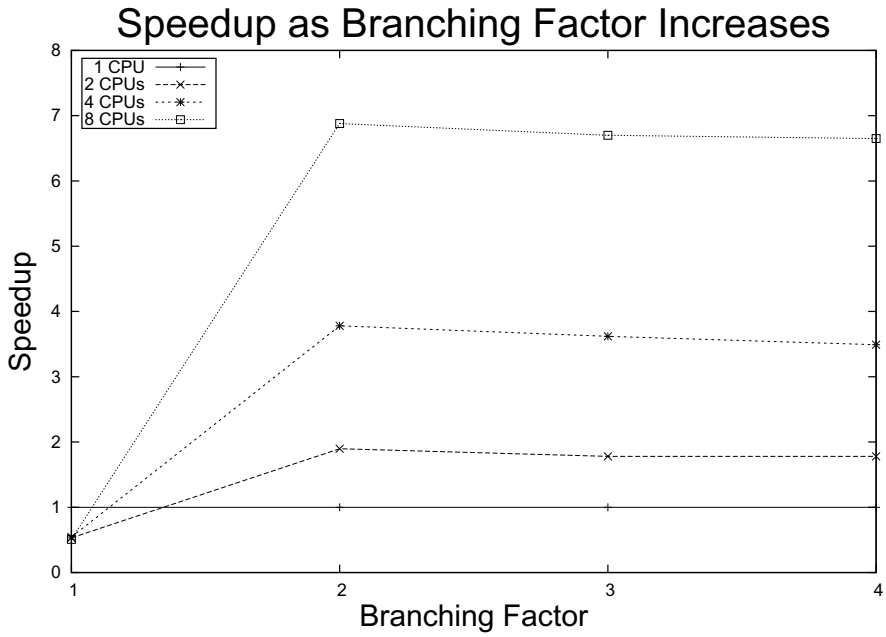

Fig. 10. Speedup of the and-entailment test, shown in relation to the branching factor of the inference graph. A depth of 7 was used in all tests, to keep the number of nodes reasonable.

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 Sect. 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.

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

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 4. The same experiment as Table 3 replacing the improvements discussed in Sect. 4.1, with a LIFO queue. Our results in Table 3 are 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 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 Sect. 6.1.

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

6.2 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 Figs. 9 and 10.

7 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.

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

1. Baum, L.F.: The Wonderful Wizard of Oz. G. M. Hill (1900)
2. Choi, J., Shapiro, S.C.: Efficient implementation of non-standard connectives and quantifiers in deductive reasoning systems. In: Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, pp. 381–390. IEEE Computer Society Press, Los Alamitos (1992)
3. Dixon, M., de Kleer, J.: Massively parallel assumption-based truth maintenance. In: Reinfrank, M., Ginsberg, M.L., de Kleer, J., Sandewall, E. (eds.) Non-Monotonic Reasoning 1988. LNCS, vol. 346, pp. 131–142. Springer, Heidelberg (1988)
4. Doyle, J.: A truth maintenance system. Artificial Intelligence 19, 231–272 (1979)
5. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19, 17–37 (1982)
6. Hickey, R.: The Clojure programming language. In: Proceedings of the 2008 Symposium on Dynamic Languages. ACM, New York (2008)
7. Huang, S.S., Green, T.J., Loo, B.T.: Datalog and emerging applications: an interactive tutorial. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, pp. 1213–1216. ACM, New York (2011)
8. de Kleer, J.: Problem solving with the ATMS. Artificial Intelligence 28(2), 197–224 (1986)
9. Lehmann, F. (ed.): Semantic Networks in Artificial Intelligence. Pergamon Press, Oxford (1992)
10. Lendaris, G.G.: Representing conceptual graphs for parallel processing. In: Conceptual Graphs Workshop (1988)
11. Martins, J.P., Shapiro, S.C.: A model for belief revision. Artificial Intelligence 35, 25–79 (1988)

12. McAllester, D.: Truth maintenance. In: Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI 1990), Boston, MA, pp. 1109–1116 (1990)
13. McKay, D.P., Shapiro, S.C.: Using active connection graphs for reasoning with recursive rules. In: Proceedings of the Seventh International Joint Conference on Artificial Intelligence, pp. 368–374. Morgan Kaufmann, Los Altos (1981)
14. Schlegel, D.R.: Concurrent inference graphs (doctoral consortium abstract). In: Proceedings of the Twenty-Seventh AAAI Conference (AAAI 2013), pp. 1680–1681 (2013)
15. Schlegel, D.R., Shapiro, S.C.: Visually interacting with a knowledge base using frames, logic, and propositional graphs. In: Croitoru, M., Rudolph, S., Wilson, N., Howse, J., Corby, O. (eds.) GKR 2011. LNCS, vol. 7205, pp. 188–207. Springer, Heidelberg (2012)
16. Schlegel, D.R., Shapiro, S.C.: Concurrent reasoning with inference graphs (student abstract). In: Proceedings of the Twenty-Seventh AAAI Conference (AAAI 2013), pp. 1637–1638 (2013)
17. Shapiro, E.: The family of concurrent logic programming languages. ACM Comput. Surv. 21(3), 413–510 (1989)
18. Shapiro, S.C.: Set-oriented logical connectives: Syntax and semantics. In: Lin, F., Sattler, U., Truszczynski, M. (eds.) Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2010), pp. 593–595. AAAI Press, Menlo Park (2010)
19. Shapiro, S.C., Martins, J.P., McKay, D.P.: Bi-directional inference. In: Proceedings of the Fourth Annual Conference of the Cognitive Science Society, pp. 90–93. The Program in Cognitive Science of The University of Chicago and The University of Michigan, Ann Arbor, MI (1982)
20. Shapiro, S.C., Rapaport, W.J.: The SNePS family. Computers & Mathematics with Applications 23(2-5), 243–275 (1992), reprinted in [9, pp. 243–275]
21. The Joint Task Force on Computing Curricula, Association for Computing Machinery, IEEE-Computer Society: Computer Science Curricula 2013 (2013)
22. University of Colorodo: Unified verb index (2012), `http://verbs.colorado.edu/verb-index/index.php`
23. Wachter, M., Haenni, R.: Propositional DAGs: a new graph-based language for representing boolean functions. In: KR 2006, 10th International Conference on Principles of Knowledge Representation and Reasoning, pp. 277–285. AAAI Press, U.K. (2006)
24. Yan, F., Xu, N., Qi, Y.: Parallel inference for latent dirichlet allocation on graphics processing units. In: Proceedings of NIPS, pp. 2134–2142 (2009)