Rating of Discrimination Networks for Rule-based Systems

Fabian Ohler, Kai Schwarz, Karl-Heinz Krempels and Christoph Terwelp
Informatik 5, Information Systems and Databases, RWTH Aachen University, 52072 Aachen, Germany

Keywords: Rule-based System, Discrimination Network, Rating.

Abstract: The amount of information stored in a digital form grows on a daily basis but is mostly only understandable by humans, not machines. A way to enable machines to understand this information is using a representation suitable for further processing, e. g. frames for fact declaration in a Rule-based System. Rule-based Systems heavily rely on Discrimination Networks to store intermediate results to speed up the rule processing cycles. As these Discrimination Networks have a very complex structure it is important to be able to optimize them or to choose one out of many Discrimination Networks based on its structural efficiency. Therefore, we present a rating mechanism for Discrimination Networks structures and their efficiencies. The ratings are based on a normalised representation of Discrimination Network structures and change frequency estimations of the facts in the working memory and are used for comparison of different Discrimination Networks regarding processing costs.

1 INTRODUCTION

This paper presents a rating function for Discrimination Networks (DNs) in Rule-based Systems (RBSs). It elaborates the need for information about the fact base of the RBS and introduces a normalised form for DNs. This normalised form allows for a rating of the DN disregarding implementation details. Using these ratings it is possible to evaluate the efficiency of DNs, measure optimization attempts and improve the overall performance of the RBS without the need for benchmarking.

The paper is organized as follows: 2 explains why a new rating approach is worthwhile. 3 introduces existing rating approaches for DNs. The new rating algorithm is developed in 4. Finally, conclusions are given in 5.

2 MOTIVATION

The different existing construction algorithms (e.g. TREAT (Miranker, 1987), Gator (Hanson and Hasan, 1993)) are suited for different use cases. Therefore the resulting DNs yield varying runtime and memory costs while maintaining the same semantics. In most cases it is desirable to minimize runtime and memory usage as much as possible. To achieve this goal a way to identify the DN which best suits the given environment is necessary.

As DNs are complex and large an automated comparison of DNs is required. The idea is to rate a DN's runtime and memory usage in order to compare different DNs. Benefits include enabling to find the 'best' DN for a given environment as well as measuring the pay-off of optimization attempts.

The composition of the facts in the working memory contribute largely to the DNs performance regarding runtime and memory usage. An approach for rating will have to take this into account and rate a DN in the context of the composition of the given working memory. Therefore a rating only gives information about the performance of a DN with a working memory that satisfies the statistical information used to rate. If a different working memory is used with the DN it is possible that the rating no longer gives an accurate estimate of the performance.

3 STATE OF THE ART

There already are some approaches to compare DNs, mostly used to show that a new construction algorithm is an improvement.

Benchmark

TREAT (Miranker, 1987) uses benchmarks like 'monkeys and bananas' (Brownston et al., 1985)

Ohler F., Schwarz K., Krempels K. and Terwelp C..
Rating of Discrimination Networks for Rule-based Systems.
DOI: 10.5220/0004634900320042
In *Proceedings of the 2nd International Conference on Data Technologies and Applications* (DATA-2013), pages 32-42
ISBN: 978-989-8565-67-9

and 'waltz' (Winston, 1992) for an OPS5 (Forgy, 1981) rule based system. These benchmarks test a construction algorithm by letting it construct a DN and then benchmarking it. This approach often leads to construction algorithms being optimized for the existing benchmarks. But because the performance depends on the composition of the working set and the possible compositions are nearly endless, a construction algorithm can yield very good results in a benchmark and still be outperformed when constructing a DN in an actual application.
Another possibility is to actually benchmark the possible DNs with the real working memory and take the best one. While being very accurate these benchmarks need a lot of time and resources. Therefore, benchmarks are of limited use when trying to rate a DN efficiently.

Cost Function
Gator (Hanson and Hasan, 1993) (Hanson, 1993) uses a thorough cost function for single rules using statistical information about the composition of the working memory as well as the selectivity of filters to predict the runtime and memory usage. It even considers the size of facts in memory and how many memory pages have to be touched to apply changes to the working memory.
While an efficient approach the cost function only rates a single rule, not the whole DN. It ignores the benefits of shared nodes on the one hand. On the other hand negated condition elements can not be rated (Hanson, 1993). Additionally the cost function was developed having implementation of databases in mind, so rating a DN not implemented on a database can lead to deviating results.

4 APPROACH

The considerations taken in the last section suggest providing a universal cost function for rating DNs. Such a cost function is introduced in this section. To rate the DN, at first its structure is normalised in a very general way representing optimizations concerning the network's structure as detailed as possible. For each normalised component cost functions are given. The costs of all network components can be used to rate any DN structure.

4.1 Normalisation

The construction algorithms introduced differ not only in the resulting network structure but also in internal mechanisms regarding the RBS. To rate DNs in a normalised manner the following simplifying assumptions are made:

4.1.1 Alpha Nodes

An alpha node filters for one attribute of a fact only. Alpha nodes filtering for more than one attribute are split up and represented as a chain of alpha nodes as they are semantically equivalent. If an alpha node is connected to other alpha nodes only and does not have a negated input (see 4.1.3) the internal memory is omitted (virtual alpha node). This approach prevents memory overhead by splitting up alpha nodes.

Facts in alpha nodes are stored for optimal performance regarding joining and selection.

4.1.2 Beta Nodes

For every input a beta node has a list containing the other nodes connected to its inputs in the order they are supposed to be joined to keep the intermediate results as small as possible. Variant I beta nodes include the negated inputs in their lists, variant II beta nodes don't (variants explained in 4.1.3). This lists can be created based on estimates regarding fact correlation or in a simplified way by ordering the nodes according to estimated size.

The storage of the fact tuples in a beta node is optimised for the beta nodes connected to its output to allow for efficient selection on relevant facts or their slots similar to alpha nodes.

To delete facts or fact tuples in beta nodes the following common optimisation is used: If a fact is deleted in one of the nodes connected to an input, the fact tuples resulting from that fact are deleted and the fact is propagated to successor nodes as it entered the node, meaning it is not joined. This optimisation is not used for nodes connected negatively.

4.1.3 Negated Condition Elements

A lot of RBSs allow elements of a rule's condition to be negated (negated condition elements). There are several ways to represent these in DNs. In the obvious variant a beta node combines a positive set of facts with a negated set of facts by propagating the set of facts in the positive set that have no counterpart (regarding the join) in the negated set. Both input nodes continue to be usual (positive) nodes, but one of the edges is marked as negated. Because no joined fact tuples are produced and passed on, such nodes can be implemented in the alpha network too: Two alpha nodes are connected by a negated edge. The 'one input rule' in the alpha network is diluted, alpha nodes have exactly one positive input and an arbitrary number of negated inputs. Negated condition elements can thus be realized in the alpha and beta network.

If a negated edge connects the output of x with an input of y, we call x a negatively connected node (NCN) of y.

The possible implementations of negations are now discussed briefly. As above we start in the beta network.

Variant I in the Beta Network
The already mentioned method of implementation in a beta node shall be explained in more detail. A beta node with positively connected nodes (PCNs) and NCNs joins with the PCNs and filters using the NCNs. If a new fact tuple reaches the node from a PCN it is joined with the other PCNs. The result is stored in the node's memory only if no matching fact tuples can be found in any of the NCNs. Deleting a fact tuple in a PCN deletes the fact tuples resulting from that fact. A new fact tuples reaching the node on a NCN is joined with the current result set and matching fact tuples are deleted. If a fact tuple is deleted in a NCN, matching fact tuples are searched for in the PCNs or their joins. Fact tuples found are filtered by NCNs and added to the result set.

Variant II in the Beta Network
The second method tries to reduce the comparatively high effort of the joins needed in a variant I beta node if a fact tuple is deleted in a NCN. For that purpose all join results of the PCNs are saved instead of only saving a filtered set. To every fact tuple a counting field is added for every NCN. The counting field holds the number of matching fact tuples in the corresponding NCN. Only the fact tuples with zeros in their counting fields are relevant to successor nodes. If a new fact tuple reaches the node from a PCN, it is joined with the other PCNs and added to the result set. The counting fields are filled with the sizes of the particular joins with every NCN. Deleting a fact tuple in a PCN deletes the fact tuples resulting from that fact. A new fact tuple reaching the node from a NCN is joined with the result set and the counting fields of the matching fact tuples are increased by one. One of the counting fields raising from zero triggers a propagation of the corresponding fact tuple to successor nodes as deleted. If a fact tuple is deleted in a NCN, it is joined with the result set and the counting fields of the matching fact tuples are decreased by one. A counting field dropping to zero has the corresponding fact tuple propagated as new to successor nodes. The rule based system Jamocha (Jamocha, 2006) uses nodes of this kind.

Negation in the Alpha Network
As mentioned above, negation can also be implemented in the alpha network. The implementation is similar to the second variant in the beta network adding counting fields for the NCNs. If a new fact reaches the node, it is joined with the NCNs and saves the join size in the counting fields. Only if no matching facts are found, the fact is passed on to successor nodes. Deleting a fact in the node propagates the fact to successor nodes only if the counting field is zero. A new fact reaching a NCN is joined with the node and the counting fields of the fact tuples in the join result are increased by one. One of the counting fields raising from zero triggers a propagation of the corresponding fact to successor nodes as deleted. If a fact in a NCN is deleted, it is joined with the node and the counting fields of the fact tuples in the join result are decreased by one. A counting field dropping to zero has the corresponding fact propagated as new to successor nodes. Gator (Hanson and Hasan, 1993) (Hanson, 1993) uses negation in this way.

The methods mentioned above are certainly not the only ways to implement negation, but one possibility to negate in the alpha network and two fundamentally different variants to negate in the beta network with dissimilar pros and cons have been presented. It has been clarified that a negated condition element does not change the nodes themselves, but how facts from the corresponding nodes are handled. The possibility to negate in the alpha network already offers potential for network optimization.

On the other hand the task to estimate the costs for a negation in a network has become more difficult than just estimating the costs of nodes concerning joins. Here the costs arising from a negation are to be added to the costs of the adjacent nodes. However to estimate the cost in a proper way, both mechanisms have to be rateable. If it is not known how a network implements negations, it is rated as if it uses variant I.

According to the considerations above, the constraints for rateable networks can be summarised as follows: Each rule condition has at least one non-negated condition element and different negated condition elements don't have shared bound variables. Alpha nodes with negated edges always have an internal memory.

4.1.4 Processing Tokens

Upon creation of a new fact f (by a rule for example) it is encapsulated into a + token. This token is injected into the root node to be processed by the DN. Analog a deleted fact is injected using a - token. The following section explains the steps necessary to process these facts when a token reaches a node. In the following ⋈ always has the semantic of the corresponding beta node.

+ token reaches alpha node If the fact f does not pass the test set it is discarded. Else the node stores the fact inside its memory (if applicable) and passes the + token on to the following nodes.
f is joined with the memory of any alpha NCN and stores the size of the joins in the matching counting fields. Only if all fields contain zeros, i. e. there are no facts matching f in alpha NCN the + token is passed on to the following nodes.
For follow-up beta nodes the fact is encapsulated into a +Temporary Result (TR) token first.
As a NCN the node joins f with the memories of the connected nodes and updates their counting fields. The corresponding counting fields are incremented by one if the join yields a result. If the increment changes the value from zero a - token or -TR token for f has to be created and passed on from the connected node.

- token reaches alpha node. If the fact f does not pass the test set it is discarded. Else the node deletes the fact from its memory (if applicable) and passes the - token on to the following nodes. The - token is only passed on to successor nodes if all (possible) counting fields are zero.
For follow-up beta nodes the fact is first encapsulated into a -TR token.
As a NCN the node joins f with the memories of the connected nodes and updates their counting fields. The corresponding counting fields are decremented by one if the join result yields a result. If the decrement changes the value to zero a + token or +TR token for f has to be created and passed on from the connected node.

+TR token reaches beta node. When a +TR token reaches a node from a NCN one has to distinguish: Variant I beta nodes join the fact with their result sets, delete the resulting tuples from its result set and propagate this to the follow-up nodes. Variant II beta nodes join the fact with their result sets and increment the counting fields of the resulting tuples by one. If a counting fields is incremented from zero, a -TR token is passed on to the following nodes.
When a +TR token reaches a node from a PCN one has to distinguish again: Variant I beta nodes (or beta nodes without NCNs) have sorted lists of all their inputs excluding the input the +TR token arrived from. The nodes then join the token with the inputs iterating the list (TR⋈input) or subtract the join for NCNs (TR-TR⋉input). In each iteration the result is stored in TR. The final result is stored in the internal memory and passed on to the following nodes.
Variant II beta nodes have lists of their PCNs excluding the input the TR token arrived from. The node then joins the token with the inputs iterating the list (TR⋈input). In each iteration the result is stored in TR and the final result is stored in the internal memory. Additionally the size of the join of each result with each NCN is stored in the appropriate counting field. Only results whose counting fields are zero are passed on to the follow-up nodes.

-TR token reaches beta node. A -TR token reaching the node from a NCN is handled similar to a +TR token from a PCN. Additionally one has to distinguish: Variant I beta nodes have sorted lists of all inputs excluding the inputs the -TR token arrived from. The node then joins the token with the inputs iterating the list (TR⋈input) or subtracts the join for NCNs (TR-TR⋉input). TR is then filtered by the input the token originated from (TR-TR⋉input). The result is then stored in the internal memory and passed on to following nodes.

4.2 Rating

The DNs shall be rated using a cost analysis. In this context costs are considered to be memory usage and runtime. The normalisation allows a uniform rating. A detailed rating is done based on statistical values about filters, joins and relations. If the facts are stored in a relational database, a lot of the values used are available for internal use already.

The better the statistical values match the real values, the more precise the DN can be rated. But even without these values, statements on DNs can be made. DNs can be compared e.g. using general mean values for the missing values or by rating all conceivable scenarios.

For the sake of brevity nodes are tagged corresponding to 1.

Hereafter is explained, which statistical values are used to rate a DN and how the different node types can be rated.

4.2.1 Statistical Values the Rating Bases On

The naming partially follows Gator (Hanson and

Table 1: Node tags.

x^{α}	is an alpha node
$x^{-\alpha}$	is an alpha node in a negated context
x^{β}	is a beta node
$x^{-\beta}$	is a beta node in a negated context
x^{+}	is an alpha or beta node in a non-negated context
x^{-}	is an alpha or beta node in a negated context

Hasan, 1993), (Hanson, 1993).

$|U(x^{\alpha})|$ $U(x^{\alpha})$ is the set of facts contained in node x^{α} including the ones filtered out by NCNs. $|U(x^{\alpha})|$ is the size of this set and is a statistical input value for alpha nodes. X is the set of facts belonging to node x as it reaches successor nodes, thus already filtered. Estimates for $|X|$ will be given.

JSF(x,y)**.** Estimated size of the join in relation to the joined nodes' fact set sizes, so this value describes the selectivity of the join (Join Selectivity Factor). It is an expected value for:

$$\frac{|X \bowtie Y|}{|X| \cdot |Y|}\ . \tag{1}$$

$\mathrm{Sel}(x^{\alpha})$**.** The selectivity of a node is the ratio between accepted and rejected facts.

$F_i(x^{\alpha})$**.** The frequency of + tokens reaching and changing the node x. F_i'(x^{α}) is the frequency of + tokens reaching the node x which lead to a propagation to follow-up nodes – this value is calculated as necessary.

$F_d(x^{\alpha})$**.** The frequency of - tokens reaching and changing the node x. $F_d(x^{\alpha})$ is the frequency of - tokens reaching the node x which lead to a propagation to follow-up nodes – this value is calculated as necessary.

$T(x)$**.** Tuple Size (T) in node x. For the sake of simplicity facts in alpha nodes are treated to be of the same size ($T(a^{\alpha})= 1$). A way to calculate this value is given for beta nodes.

$TPP(x)$**.** The number of Tuples per Page (TPP) for facts in node x. Caused by the simplification of fact sizes the TPP is inversely proportional to the tuple size T.

Remark Concerning the Join Selectivity Factor. Let x and y be nodes with non-empty internal memory, thus $|X| > 0$ and $|Y| > 0$. Adding or removing a fact from X emerges the node x' with the result set X'. The size of the join $X \bowtie Y$ is expected to change by some $\mu \in \mathbb{R}_+$. An estimated value for this can be calculated using the $JSF(x,y)$.

$$JSF(x,y) = \frac{|X' \bowtie Y|}{|X'| \cdot |Y|} \tag{2a}$$

$$= \frac{|X \bowtie Y| \pm \mu}{(|X| \pm 1) \cdot |Y|} \tag{2b}$$

$$= \frac{JSF(x,y) \pm \frac{\mu}{|X| \cdot |Y|}}{1 \pm \frac{1}{|X|}} \tag{2c}$$

$$\Leftrightarrow \frac{JSF(x,y)}{1 \pm \frac{1}{|X|}}\left(1 \pm \frac{1}{|X|} - 1\right) = \frac{\pm\mu}{(|X| \pm 1)\,|Y|} \tag{2d}$$

$$\Leftrightarrow \frac{JSF(x,y)}{|X| \pm 1} = \frac{\mu}{(|X| \pm 1)\,|Y|} \tag{2e}$$

$$\Leftrightarrow JSF(x,y) \cdot |y| = \mu \tag{2f}$$

Until now, the Join Selectivity Factor (JSF) has been considered for node pairs only. For a precise size estimation of joins with several inputs JSFs influenced by conditional probabilities of intermediate results and further inputs would be necessary. To keep the amount of input values small, the calculations are simplified in the following way: The JSF for two inputs is a good mean for the JSF between these two inputs as well as for the JSF between each of these inputs and all intermediate results arising during the processing of the join list in the node.

Preliminary Considerations about Join Costs. At this point the problem of join cost estimation is discussed. Without considering implementation details it is hard to determine the right criteria for the estimation. Thus primarily the costs of a join in a network using an optimal implementation are to be examined. As mentioned in the normalized form description, nodes are optimized for successor nodes. Hence for the costs of the join $L \bowtie R$ mainly the size of the result, $S := |L| \cdot JSF(l,r) \cdot |R|$, is to be considered. The size of this set is a main characteristic of the join and can not be changed by optimisations in the join implementation. To factor the size of facts in this set into the resulting costs accesses to memory pages are chosen as basis for the rating.
This approach can also be found in Gator (Hanson, 1993). There an algorithm applied by Cárdenas (Cárdenas, 1975) is used to calculate the amount of memory pages touched for datasets (uniformly) distributed across m memory pages when selecting k datasets as follows:

$$C(m,k) = m(1 - (1 - 1/m)^k) \tag{3}$$

A derivation can be found in Yao (Yao, 1977), where the formula is considered to be faulty as it reflects a

combination with repetition instead of a *combination without repetition*, but Yao's corrected version only allows integer numbers of touched memory pages. The error described is insignificant for large TPP values and is to be put up with to allow for non-integer estimated numbers of touched memory pages without interpolation. According to Yao alternative formulas are by far more complicated, but can be used to replace the Cárdenas formula as needed to determine more accurate results.
The number of resulting datasets is $k := S$.
Most nodes are connected to multiple successor nodes and their facts are usually joined with different slots, so datasets can not be stored optimized for selection of a particular slot. Thus to assume a uniform distribution of the datasets will presumably cause only a minor deviation for most of the nodes. Nodes that store their data sets optimized in this way cause less runtime costs and can be considered separately. In the interest of clarity this is not done here.
Using the simplified knowledge about the number of facts / fact tuples of a node r fitting on a memory page, the number of memory pages needed for r can be determined:

$$m(\mathrm{r}) := \lceil |\mathrm{R}| / \mathrm{TPP}(\mathrm{r}) \rceil \tag{4}$$

$$m(\mathrm{U}(R)) := C\left(\left\lceil \frac{|\mathrm{U}(R)|}{\mathrm{TPP}(\mathrm{r})} \right\rceil, m(\mathrm{r})\right) \tag{5}$$

The costs to join a set of p facts / fact tuples from node l on node r will be given for subsequent calculations by the following formula:

$$\mathrm{JC}_{\mathrm{l},\mathrm{r}}(p) := C(m(\mathrm{r}), p \cdot \mathrm{JSF}(\mathrm{l},\mathrm{r}) \cdot |\mathrm{R}|) \tag{6}$$

4.2.2 Root Node

This node is necessary and therefore present in every network. Its outer structure is unique. Thus its needless to factor this node into the rating.
However statistical values about the set of facts in the root node, the number of facts per memory page and the frequency of +/- tokens reaching the root node and changing it, are important to derive values for successor nodes.

4.2.3 Alpha Node

An alpha node causes memory and runtime costs in the network. Memory costs are involved only if the node has an internal memory.

Memory. To determine the memory costs of alpha node x^α with internal memory, let $N(\mathrm{x}^\alpha)$ be the set of alpha NCNs of x^α. Neglecting the counting fields the memory costs of x^α are given by $|\mathrm{U}(\mathrm{x}^\alpha)|$. Depending on the implementation and size of the facts, the memory cost impact of counting fields varies. For now a counting field shall increase the size of a fact by 15%.

Result. The memory costs for alpha node results in

$$|\mathrm{U}(\mathrm{x}^\alpha)| \left(1 + 0,15 \cdot |N(\mathrm{x}^\alpha)|\right) \tag{7}$$

memory units.

Runtime. Below an attempt is made to give an estimation for runtime costs for x^α regarding new facts:

$$\mathrm{F_i}(\mathrm{x}^\alpha)\Bigg(\mathrm{InsC}_{\mathrm{x}^\alpha} + \sum_{\mathrm{y}^{-\alpha} \in N(\mathrm{x}^\alpha)} \mathrm{JoinC}_{\mathrm{x}^\alpha}(\mathrm{y}^{-\alpha}) + \sum_{\mathrm{y}^\alpha \in N(\mathrm{x}^{-\alpha})} \mathrm{JoinC}_{\mathrm{x}^{-\alpha}}(\mathrm{y}^\alpha)\Bigg) \tag{8}$$

$\mathrm{InsC}_{\mathrm{x}^\alpha}$ are the costs to filter and insert a fact into the internal memory. These are 1 for applying the filter if x^α does not have an internal memory. Otherwise the costs are increased by 1 for storing the fact in the node appropriately, resulting in 2 runtime units.
If there are alpha NCNs for x^α, upon inserting a fact into x^α it is joined with their facts. Let $\mathrm{N}(\mathrm{x}^\alpha)$ be the set of alpha NCNs of x^α. $\mathrm{JoinC}_{\mathrm{x}^\alpha}(\mathrm{y}^{-\alpha})$ are the join costs for the new fact in x^α with the NCN $\mathrm{y}^{-\alpha}$. These are approximated by $\mathrm{JC}_{\mathrm{x}^\alpha,\mathrm{U}(\mathrm{y}^{-\alpha})}(1)$.
As a NCN for other alpha nodes, $\mathrm{x}^{-\alpha}$ joins any inserted facts with the memories of its connected nodes. Let $\mathrm{N}(\mathrm{x}^{-\alpha})$ be the set of nodes $\mathrm{x}^{-\alpha}$ is connected to. $\mathrm{JoinC}_{\mathrm{x}^{-\alpha}}(\mathrm{y}^\alpha)$ are the join costs for the new fact in $\mathrm{x}^{-\alpha}$ with the connected node y^α. These are approximated by $\mathrm{JC}_{\mathrm{x}^{-\alpha},\mathrm{U}(\mathrm{y}^\alpha)}(1)$.
In the same manner an estimate for costs resulting from deleting a fact is given:

$$\mathrm{F_d}(\mathrm{x}^\alpha)\Bigg(\mathrm{DelC}_{\mathrm{x}^\alpha} + \mathrm{CheckCounters} + \sum_{\mathrm{y}^\alpha \in N(\mathrm{x}^{-\alpha})} \mathrm{JoinC}_{\mathrm{x}^{-\alpha}}(\mathrm{y}^\alpha)\Bigg) \tag{9}$$

Here $\mathrm{DelC}_{\mathrm{x}^\alpha}$ are the costs to filter and delete the fact from the internal memory. These are 1 for applying the filter if x^α does not have an internal memory. Otherwise the costs are increased by 1 for deleting the fact, resulting in 2 runtime units.
If there are alpha NCNs for x^α, it has to check the

fact's counting fields before propagating it in the network. For this action 1 runtime unit is assessed.
As a NCN for other alpha nodes, $x^{-\alpha}$ joins any deleted facts with the memories of its connected nodes. Let $N(x^{-\alpha})$ be the set of nodes $x^{-\alpha}$ is connected to. $\mathrm{JoinC}_{x^{-\alpha}}(y^{\alpha})$ are the join costs for the fact to be deleted in $x^{-\alpha}$ with the connected node y^{α}. These are approximated by $JC_{x^{-\alpha},U(y^{\alpha})}(1)$.

Result. The runtime costs for an alpha node x^{α} without internal memory are

$$F_i(x^{\alpha}) + F_d(x^{\alpha}) \tag{10}$$

runtime units, those for an alpha node x^{α} with internal memory are

$$\begin{aligned} F_i(x^{\alpha}) & \left(2 + \sum_{y^{\alpha} \in N(x^{-\alpha})} JC_{x^{-\alpha},U(y^{\alpha})}(1)\right) \\ & + F_d(x^{\alpha}) \left(2 \right. \\ & \left. + \sum_{y^{\alpha} \in N(x^{-\alpha})} JC_{x^{-\alpha},U(y^{\alpha})}(1)\right) \end{aligned} \tag{11}$$

runtime units if x^{α} does not have any NCNs and

$$\begin{aligned} F_i(x^{\alpha}) & \left(2 + \sum_{y^{-\alpha} \in N(x^{\alpha})} JC_{x^{\alpha},U(y^{-\alpha})}(1) \right. \\ & \left. + \sum_{y^{\alpha} \in N(x^{-\alpha})} JC_{x^{-\alpha},U(y^{\alpha})}(1)\right) \\ & + F_d(x^{\alpha}) \left(3 \right. \\ & \left. + \sum_{y^{\alpha} \in N(x^{-\alpha})} JC_{x^{-\alpha},U(y^{\alpha})}(1)\right) \end{aligned} \tag{12}$$

runtime units if x^{α} has NCNs.

Further Considerations The set of facts propagated by x^{α} is filtered by its NCNs ad can be described by $U(x^{\alpha}) - \bigcup_{y^{-\alpha} \in N(x^{\alpha})} U(x^{\alpha}) \ltimes y^{-\alpha}$. As only those facts are propagated appearing in none of the joins, but as facts can easily appear in several joins, the set of facts propagated can not just be estimated (similar to the reverse triangle inequality) by

$$\begin{aligned} |U(x^{\alpha})| & - \sum_{y^{-\alpha} \in N(x^{\alpha})} \left|U(x^{\alpha}) \ltimes y^{-\alpha}\right| \\ & \leq \left| U(x^{\alpha}) - \bigcup_{y^{-\alpha} \in N(x^{\alpha})} U(x^{\alpha}) \ltimes y^{-\alpha} \right| . \end{aligned} \tag{13}$$

Following thoughts will elaborate how the number of propagated facts can be estimated. The sum of the counting fields of $U(x^{\alpha})$ regarding $y^{-\alpha}$ can be expected to be $|U(x^{\alpha}) \bowtie y^{-\alpha}|$. The estimate for a counting field of a fact in $U(x^{\alpha})$ is therefore

$$\frac{|U(x^{\alpha}) \bowtie y^{-\alpha}|}{|U(x^{\alpha})|} = \left|y^{-\alpha}\right| \cdot JSF(U(x^{\alpha}), y^{-\alpha}) . \tag{14}$$

The expected probability that a fact in $U(x^{\alpha})$ matches a fact in $y^{-\alpha}$ is therefore $JSF(U(x^{\alpha}, b^{-\alpha}))$. The probability that a fact in $U(x^{\alpha})$ has no matching fact in $y^{-\alpha}$ can be calculated with

$$\left(1 - JSF(U(x^{\alpha}), y^{-\alpha})\right)^{|y^{-\alpha}|} . \tag{15}$$

An estimate for the number of facts in $U(x^{\alpha})$ which have zero in all counting fields is given by

$$\begin{aligned} |x^{\alpha}| = |U(x^{\alpha})| \cdot \prod_{y^{-\alpha} \in N(x^{\alpha})} & \left(1 \right. \\ & \left. - JSF(U(x^{\alpha}), y^{-\alpha})\right)^{|y^{-\alpha}|} . \end{aligned} \tag{16}$$

Hereby an estimate for the number of propagated facts has been found. Changes in alpha NCNs can lead to changes in the counting fields and therefore to +/- (TR) tokens to be propagated.
An estimate for the number of (TR) tokens is valuable for further considerations. Let $z^{-\alpha}$ be an alpha NCN. Further let $N(x^{\alpha})$ be the set of NCNs of x^{α}, so $z^{-\alpha} \in N(x^{\alpha})$. Referencing (16) the number of facts in $U(x^{\alpha})$ with zeros in all counting fields when a fact is added to $z^{-\alpha}$ can be estimated with:

$$\begin{aligned} & \left(1 - JSF(U(x^{\alpha}), z^{-\alpha})\right) |U(x^{\alpha})| \\ & \cdot \prod_{y^{-\alpha} \in N(x^{\alpha})} \left(1 - JSF(U(x^{\alpha}), y^{-\alpha})\right)^{|y^{-\alpha}|} \end{aligned} \tag{17}$$

The expectancy for the number of generated - (TR) tokens after inserting a fact in $z^{-\alpha}$ can be described as the difference between (16) and (17):

$$JSF(U(x^{\alpha}), z^{-\alpha}) \, |x^{\alpha}| \tag{18}$$

The expectancy for the number of generated + (TR) tokens after deleting a fact in $z^{-\alpha}$ can be calculated analogue:

$$\frac{JSF(U(x^{\alpha}), z^{-\alpha})}{1 - JSF(U(x^{\alpha}), z^{-\alpha})} |x^{\alpha}| \tag{19}$$

With these thoughts an expectancy for the number of generated +/- (TR) tokens can be given:

$$\begin{aligned} & F_i{'}(x^{\alpha}) \\ & \quad = F_i(x^{\alpha}) \frac{|x^{\alpha}|}{|U(x^{\alpha})|} \\ & \qquad + \sum_{y^{-\alpha} \in N(x^{\alpha})} F_d(y^{-\alpha}) \, |x^{\alpha}| \frac{JSF(U(x^{\alpha}), y^{-\alpha})}{1 - JSF(U(x^{\alpha}), y^{-\alpha})} \end{aligned} \tag{20}$$

$$\mathrm{F_d}'(\mathrm{x}^\alpha) = \mathrm{F_d}(\mathrm{x}^\alpha)\frac{|\mathrm{x}^\alpha|}{|\mathrm{U}(\mathrm{x}^\alpha)|} + \sum_{\mathrm{y}^{-\alpha}\in N(\mathrm{x}^\alpha)} \mathrm{F_i}(\mathrm{y}^{-\alpha})\,|\mathrm{x}^\alpha|\,\mathrm{JSF}(\mathrm{U}(\mathrm{x}^\alpha),\mathrm{y}^{-\alpha}) \tag{21}$$

Now one can estimate the frequency of +/- tokens that reach and change x^α from the PCN w^α as follows:

$$\mathrm{F_i}(\mathrm{x}^\alpha) = \mathrm{Sel}(\mathrm{x}^\alpha)\,\mathrm{F_i}(\mathrm{w}^\alpha) \tag{22a}$$

$$\mathrm{F_d}(\mathrm{x}^\alpha) = \mathrm{Sel}(\mathrm{x}^\alpha)\,\mathrm{F_d}(\mathrm{w}^\alpha) \tag{22b}$$

Lacking a expectancy for facts in a node one can estimate this analogue:

$$|\mathrm{U}(\mathrm{x}^\alpha)| = \mathrm{Sel}(\mathrm{x}^\alpha)\,|\mathrm{w}^\alpha| \tag{23}$$

4.2.4 Beta Node

A beta node always causes memory and runtime costs. Let $\mathrm{E}(\mathrm{x}^\beta)$ be the (multi-)set of the nodes connected to the inputs of x^β. Further let $\mathrm{E}(\mathrm{x}^\beta) = \mathrm{P}(\mathrm{x}^\beta) \cup \mathrm{N}(\mathrm{x}^\beta)$ with $\mathrm{N}(\mathrm{x}^\beta)$ being the set of NCNs and $\mathrm{P}(\mathrm{x}^\beta)$ the set of PCNs.

Preliminary. When a new fact reaches a beta node it has to be joined with the facts of all other PCNs. Below an estimate for the size of this join will be developed. The expectancy for the input y of node x^β is $\mathrm{JoinSize}_{\mathrm{x}^\beta}(\mathrm{y})$.
For nodes lacking an edge in the join graph the JSF values are always one. Let x^β have n inputs. For the input y it has a sorted join list of all inputs excluding y. This join list consecutively numbers the inputs e_i of x^β beginning with index 2. e_1 always is y. The JSF values are marked with the indices of the corresponding inputs for the sake of clarity

$$\mathrm{JoinSize}_{\mathrm{x}^\beta}(\mathrm{y}^+) = \prod_{k=2}^{n} |\mathrm{e_k}| \cdot \mathrm{JSF}_{\mathrm{y}^+}(\mathrm{e}_{k-1},\mathrm{e}_k) \quad \text{with } \mathrm{e}_1 = \mathrm{y}^+ \tag{24}$$

Memory. $\left|\mathrm{U}(\mathrm{x}^\beta)\right|$ is to be calculated. The mean of all join sizes will give an estimate.

$$\left|\mathrm{U}(\mathrm{x}^\beta)\right| = \frac{1}{\left|P(\mathrm{x}^\beta)\right|}\sum_{\mathrm{y}^+\in P(\mathrm{x}^\beta)} \left|\mathrm{y}^+\right| \mathrm{JoinSize}_{\mathrm{x}^\beta}(\mathrm{y}^+) \tag{25}$$

This estimate is further filtered by NCNs, see (16).

$$\left|\mathrm{x}^\beta\right| = \left|\mathrm{U}(\mathrm{x}^\beta)\right| \prod_{\mathrm{y}^-\in N(\mathrm{y}^\beta)} (1-\mathrm{JSF}(\mathrm{U}(\mathrm{x}^\beta),\mathrm{y}^-))^{|\mathrm{y}^-|} \tag{26}$$

The size of the fact tuple equals the sum of the tuples joined. Given the size the number of tuples per memory page can be calculated.

$$\mathrm{T}(\mathrm{x}^\beta) = \sum_{\mathrm{y}^+\in P(\mathrm{x}^\beta)} \mathrm{T}(\mathrm{y}^+) \tag{27a}$$

$$\mathrm{TPP}(\mathrm{x}^\beta) = \mathrm{TPP}(\mathrm{root})/\mathrm{T}(\mathrm{x}^\beta) \tag{27b}$$

Result. The memory costs of a beta node without NCNs and a variant I beta node (see 4.1.3) is given by

$$\left|\mathrm{x}^\beta\right| T(\mathrm{x}^\beta)\ . \tag{28}$$

In a variant II beta node the counting fields further increase the memory costs:

$$\left|\mathrm{U}(\mathrm{x}^\beta)\right| \left(T(\mathrm{x}^\beta) + 0,15 \cdot \left|N(\mathrm{x}^\beta)\right|\right) \tag{29}$$

Runtime. The frequency of +/-(TR) tokens being passed on from beta nodes is needed to estimate the runtime costs. The frequency can be estimated similar to the frequency in the alpha network. As no estimate about the bundling of facts in TR tokens can be made without a distribution function the assumption is made that a TR token only encapsulates one fact tuple. Given this assumption the frequency can simply be multiplied with the expectancy for the number of generated fact tuples.
For -TR tokens reaching a beta node from a PCN the optimization described in 4.1.2 is used. Therefore only the fact to be deleted is considered and not the deleted elements. This optimization can not be used for NCNs.

$$\begin{aligned}\mathrm{F_i}'(\mathrm{x}^\beta) &= \frac{\left|\mathrm{x}^\beta\right|}{\left|\mathrm{U}(\mathrm{x}^\beta)\right|}\sum_{\mathrm{y}^+\in P(\mathrm{x}^\beta)} \mathrm{F_i}'(\mathrm{y}^+)\cdot \mathrm{JoinSize}_{\mathrm{x}^\beta}(\mathrm{y}^+) \\ &\quad + \sum_{\mathrm{y}^-\in N(\mathrm{x}^\beta)} \mathrm{F_d}'(\mathrm{y}^-)\left|\mathrm{x}^\beta\right| \frac{\mathrm{JSF}(\mathrm{U}(\mathrm{x}^\beta),\mathrm{y}^-)}{1-\mathrm{JSF}(\mathrm{U}(\mathrm{x}^\beta),\mathrm{y}^-)}\end{aligned} \tag{30a}$$

$$\mathrm{F_d}'(\mathrm{x}^\beta) = \sum_{\mathrm{y}^+\in P(\mathrm{x}^\beta)} \mathrm{F_d}'(\mathrm{y}^+) + \sum_{\mathrm{y}^-\in N(\mathrm{x}^\beta)} \mathrm{F_i}'(\mathrm{y}^-)\left|\mathrm{x}^\beta\right| \mathrm{JSF}(\mathrm{U}(\mathrm{x}^\beta),\mathrm{y}^-) \tag{30b}$$

The formula for $\mathrm{JoinSize}_{\mathrm{x}^\beta}(\mathrm{y})$ (24) only considers PCNs of x^β.

Algorithm 1: costPosInsVarI$_{x^\beta}(y^+)$: Expected costs for handling a +TR token from a PCN of a variant I beta node or one without NCNs.

```
input  : beta node $x^\beta$, input $y^+$ of $x^\beta$
output: expected costs
begin
    costs $\leftarrow 0$
    size $\leftarrow 1$
    $r_1 \leftarrow y^+$
    while not empty(joinList ) do
        // $R_k$ is right join operand
        $r_k \leftarrow$ pop(joinList )
        costs $\leftarrow$ costs $+ JC_{r_{k-1},r_k}$ (size)
        if $r_k$ is positive then                // i.e. $r_k^+$
            size $\leftarrow$ size $\cdot JSF_{y^+}(r_{k-1}, r_k^+) \cdot |r_k^+|$
        else                                     // i.e. $r_k^-$
            size $\leftarrow$
            $\left(\sqrt[M_{x^\beta}(r_k^-)]{\frac{|X|}{|U(X)|}}\right)^{\text{weighting}_{x^\beta}(y^+,r_k^-)}$
    return costs
```

Algorithm 2: costNegDelVarI$_{x^\beta}(y^+)$: Expected costs for handling a -TR token from a NCN of a variant I beta node.

```
input  : beta node $x^\beta$, input $y^-$ of $x^\beta$
output: expected costs
begin
    costs $\leftarrow 0$
    size $\leftarrow 1$
    $r_1 \leftarrow y^-$
    while not empty(joinList ) do
        // $R_k$ is right join operand
        $r_k \leftarrow$ pop(joinList )
        costs $\leftarrow$ costs $+ JC_{r_{k-1},r_k}$ (size)
        if $r_k$ is positive then                // i.e. $r_k^+$
            size $\leftarrow$ size $\cdot JSF_{y^-}(r_{k-1}, r_k^+) \cdot |r_k^+|$
        else                                     // i.e. $r_k^-$
            size $\leftarrow$
            $\left(\sqrt[M_{x^\beta}(r_k^-)]{\frac{|X|}{|U(X)|}}\right)^{\text{weighting}_{x^\beta}(y^-,r_k^-)}$
    costs $\leftarrow$ costs $+ JC_{x^\beta,y^-}$ (size)   // final join
    return costs
```

The runtime costs can be described with:

$$\sum_{y^+ \in P(x^\beta)} F_i'(y^+) \cdot \text{costPosInput}_{x^\beta}(y^+) + F_d'(y^+) \cdot \text{costPosDel}_{x^\beta}(y^+) + \sum_{y^- \in N(x^\beta)} F_i'(y^-) \cdot \text{costNegInput}_{x^\beta}(y^-) + F_d'(y^-) \cdot \text{costNegDel}_{x^\beta}(y^-) \tag{31}$$

The costs vary depending on whether a variant I or variant II beta node is used.

Variant I Beta Nodes (or without NCNs). The costs for handling a -TR token from a PCN (costPosDel) are: As the result set has to be searched and the matching entries have to be deleted, the costPosDel are:

$$\text{costPosDel}_{x^\beta}(y^+) = m(x^\beta) + C(m(x^\beta), \text{JoinSize}_{x^\beta}(y^+)) \tag{32}$$

The costs for handling a +TR token from a NCN (costNegIns) can be approximated by $JC_{y^-,x^\beta}(1)$, as the corresponding join can be performed efficiently for NCNs.

The costs for handling a +TR token from a PCN (costPosIns) can be approximated by 1.

The costs for handling a -TR token from a NCN (costNegDel) can be approximated by 2 for variant I beta nodes. For beta nodes without NCNs, no algorithm is necessary. 2 and 1 are identical except for the additional line in 2 and the positive / negative signs marking the input node y and the node l. They have only been separated for the sake of clarity.

A way to identify the weighting of a node is still needed. Why the weighting is important was discussed in 4.2.3: Fact tuples that originated from the join of PCNs of a node can be culled by several NCNs. This fact prevents using

$$\left|U(x^\beta)\right| - \sum_{y^- \in N(x^\beta)} \left|y^-\right| JSF(y^-, U(x^\beta)) \left|U(x^\beta)\right|$$

to estimate $\left|x^\beta\right|$.

From now on this multiple culling is called overlap – as the filters from the different NCNs overlap. The following holds when the overlap is maximal, i. e. all culled fact tuples are filtered by all NCNs:

$$\max\left\{\left|y^-\right| JSF(y^-, U(x^\beta)) \left|U(x^\beta)\right| \mid y^- \in N(x^\beta)\right\} = \left|\bigcup_{y^- \in N(x^\beta)} y^- \bowtie U(x^\beta)\right| . \tag{33}$$

While the overlap is minimal, i. e. every culled fact tuple is filtered by only one NCN,

$$\sum_{y^- \in N(x^\beta)} \left|y^-\right| JSF(y^-, U(x^\beta)) \left|U(x^\beta)\right| = \left|\bigcup_{y^- \in N(x^\beta)} y^- \bowtie U(x^\beta)\right| \tag{34}$$

holds. As in general the data is not sufficient to calculate the overlap correctly, the mean between the minimal and maximal overlap will be used.

M_{e} is the mean expectancy for the number of matching fact tuples in the NCNs in the join list of e (using its order). The indices of the nodes d_k^- match the indices of the join list of e. $\mathrm{d}^+_{j \to k-1}$ is the last PCN with an index lower than k.

$$M_{\mathrm{x}^\beta}(\mathrm{e}) = \frac{1}{2} \cdot \left(\max_{\mathrm{d}_k^- \in N(\mathrm{x}^\beta)} \left|\mathrm{d}_k^-\right| \mathrm{JSF}_{\mathrm{e}}(\mathrm{d}^+_{j\to k-1}, \mathrm{d}_k^-) + \sum_{\mathrm{d}_k^- \in N(\mathrm{x}^\beta)} \left|\mathrm{d}_k^-\right| \mathrm{JSF}_{\mathrm{e}}(\mathrm{d}^+_{j\to k-1}, \mathrm{d}_k^-) \right) \tag{35}$$

The $\texttt{weighting}_{\mathrm{x}^\beta}(\mathrm{e}, \mathrm{z}^-)$ gives an estimate for the number of matching fact tuples in the NCN z^- after handling all joins in the join list of e until reaching z^-.

$$\texttt{weighting}_{\mathrm{x}^\beta}(\mathrm{e}, \mathrm{z}_k^-) = M_{\mathrm{x}^\beta}(\mathrm{e}) \cdot \frac{\left|\mathrm{z}_k^-\right| \mathrm{JSF}_{\mathrm{e}}(\mathrm{d}^+_{j\to k-1}, \mathrm{z}_k^-)}{\sum_{\mathrm{d}_k^- \in N(\mathrm{x}^\beta)} \left|\mathrm{d}_k^-\right| \mathrm{JSF}_{\mathrm{e}}(\mathrm{d}^+_{j\to k-1}, \mathrm{d}_k^-)} \tag{36}$$

Now we can deduce

$$\begin{aligned} \frac{|\mathrm{X}|}{|\mathrm{U}(X)|} &= \left(\sqrt[M_{\mathrm{x}^\beta}(\mathrm{e}_k)]{\frac{|\mathrm{X}|}{|\mathrm{U}(X)|}} \right)^{M_{\mathrm{x}^\beta}(\mathrm{e}_k)} \\ &= \left(\sqrt[M_{\mathrm{x}^\beta}(\mathrm{e}_k)]{\frac{|\mathrm{X}|}{|\mathrm{U}(X)|}} \right)^{\sum_{\mathrm{e}_k^- \in N(\mathrm{x}^\beta)} \texttt{weighting}_{\mathrm{x}^\beta}(\mathrm{y}, \mathrm{e}_k^-)} \\ &= \prod_{\mathrm{e}_k^- \in N(\mathrm{x}^\beta)} \left(\sqrt[M_{\mathrm{x}^\beta}(\mathrm{e}_k)]{\frac{|\mathrm{X}|}{|\mathrm{U}(X)|}} \right)^{\texttt{weighting}_{\mathrm{x}^\beta}(\mathrm{y}, \mathrm{e}_k^-)} . \end{aligned} \tag{37}$$

Now a function for estimating the size of the join considering NCNs can be given:

$$\mathrm{JoinSize}_{\mathrm{x}^\beta}(\mathrm{y}) = \prod_{k=2}^{n} \begin{cases} \left|\mathrm{e}_k^+\right| \cdot \mathrm{JSF}_{\mathrm{y}}(\mathrm{d}^+_{j\to k-1}, \mathrm{e}_k^+) & \text{for } \mathrm{e}_k^+ \\ \left(\sqrt[M_{\mathrm{x}^\beta}(\mathrm{e}_k)]{\frac{|\mathrm{X}|}{|\mathrm{U}(X)|}} \right)^{\texttt{weighting}_{\mathrm{x}^\beta}(\mathrm{y}, \mathrm{e}_k^-)} & \text{otherwise} \end{cases} \tag{38}$$

For variant I beta nodes or beta nodes without NCNs the runtime costs are:

$$\begin{aligned} &\sum_{\mathrm{y}^+ \in P(\mathrm{x}^\beta)} \mathrm{F_i}'(\mathrm{y}^+) \cdot \mathrm{costPosInsVarI}_{\mathrm{x}^\beta}(\mathrm{y}^+) + \mathrm{F_d}'(\mathrm{y}^+) \\ &\cdot \left(m(\mathrm{x}^\beta) + C(m(\mathrm{x}^\beta), \mathrm{JoinSize}_{\mathrm{x}^\beta}(\mathrm{y}^+)) \right) + \sum_{\mathrm{y}^- \in N(\mathrm{x}^\beta)} \mathrm{F_i}'(\mathrm{y}^-) \\ &\cdot \mathrm{JC}_{\mathrm{y}^-, \mathrm{x}^\beta}(1) + \mathrm{F_d}'(\mathrm{y}^-) \cdot \mathrm{costNegDelVarI}_{\mathrm{x}^\beta}(\mathrm{y}^-) \end{aligned} \tag{39}$$

Variant II Beta Node. The costs for handling a -TR token from a PCN (costPosDel) can be approximated by $m(\mathrm{U}(\mathrm{x}^\beta)) + C(m(\mathrm{U}(\mathrm{x}^\beta)), \mathrm{JoinSize}_{\mathrm{x}^\beta}(\mathrm{y}^+))$ runtime units, because the result set has to be searched and the matching fact tuples have to be updated.

The costs for handling a +TR token from a NCN (costNegIns) can be approximated by $2 \cdot \mathrm{JC}_{\mathrm{b}^-, \mathrm{U}(\mathrm{a}^\beta)}(1)$ runtime units because the fact has to be joined with the result set and the counting fields have to be updated.

3 calculates the costs for handling a +TR token from a PCN (costPosIns) and 4 the costs for handling a -TR token from a NCN (costNegDel).

Algorithm 3: $\mathrm{costPosInsVarII}_{\mathrm{x}^\beta}(\mathrm{y}^+)$: Estimated costs for handling a +TR token from a PCN of a variant II beta node.

```
input  : beta node $\mathrm{x}^\beta$, input $\mathrm{y}^+$ of $\mathrm{x}^\beta$
output: estimated costs
begin
    costs $\leftarrow 0$
    size $\leftarrow 1$
    $\mathrm{r}_1^+ \leftarrow \mathrm{y}^+$
    while not empty(positiveJoinList) do
        // R is right join operand
        $\mathrm{r}_k^+ \leftarrow$ pop(positiveJoinList )
        costs $\leftarrow$ costs $+ \mathrm{JC}_{\mathrm{r}^+_{k-1}, \mathrm{r}^+_k}$(size)
        size $\leftarrow$ size $\cdot \mathrm{JSF}_{\mathrm{y}^+}(\mathrm{r}^+_{k-1}, \mathrm{r}^+_k) \cdot \left|\mathrm{r}^+_k\right|$
    while not empty(negativeInputList) do
        $\mathrm{n}^- \leftarrow$ pop(negativeInputList )
        costs $\leftarrow$ costs $+ \mathrm{JC}_{\mathrm{U}(\mathrm{x}^\beta), \mathrm{n}^-}$(size)    // join
    // save results and counting fields
    costs $\leftarrow$ costs $+$ size
    return costs
```

Algorithm 4: $\mathrm{costNegDelVarII}_{\mathrm{x}^\beta}(\mathrm{y}^-)$: Estimated costs for handling a -TR token from a NCN of a variant II beta node.

```
input  : beta node $\mathrm{x}^\beta$, input $\mathrm{y}^-$ of $\mathrm{x}^\beta$
output: estimated costs
begin
    costs $\leftarrow 0$
    size $\leftarrow 1$
    $\mathrm{r}_1 \leftarrow \mathrm{y}^-$
    while not empty(positiveJoinList ) do
        // R is right join operand
        $\mathrm{r}_k^+ \leftarrow$ pop(positiveJoinList )
        costs $\leftarrow$ costs $+ \mathrm{JC}_{\mathrm{r}_{k-1}, \mathrm{r}^+_k}$(size)
        size $\leftarrow$ size $\cdot \mathrm{JSF}_{\mathrm{y}^-}(\mathrm{r}_{k-1}, \mathrm{r}^+_k) \cdot \left|\mathrm{r}^+_k\right|$
    // decrement counting fields
    costs $\leftarrow$ costs $+$ size
    return costs
```

So the runtime costs of a variant II beta node are:

$$\begin{aligned}\sum_{y^+ \in P(x^\beta)} & \mathrm{F_i}'(y^+) \cdot \mathrm{costPosInsVarII}_{x^\beta}(y^+) \\ & + \mathrm{F_d}'(y^+) \Big(m(\mathrm{U}(x^\beta)) + C(m(\mathrm{U}(x^\beta)), \mathrm{JoinSize}_{x^\beta}(y^+)) \Big) \\ & + \sum_{y^- \in N(x^\beta)} \mathrm{F_i}'(y^-) \cdot 2 \cdot \mathrm{JC}_{y^-,\mathrm{U}(x^\beta)}(1) \\ & + \mathrm{F_d}'(y^-) \cdot \mathrm{costNegDelVarII}_{x^\beta}(y^-)\end{aligned} \tag{40}$$

4.2.5 Terminal Node

The runtime and memory costs of a terminal node can be omitted as it only needs to set a flag if and only if the connected beta node x^β has fact tuples in its internal memory.

5 CONCLUSIONS

In this paper a general structure for DNs has been developed allowing for a consistent rating. For the components of DNs in this structure cost functions have been worked out. By rating the normalised DN every DN can be rated.

In the course of the rating simplifications and estimations had to be made for several reasons. These include the estimates for means of the overlap or the simplified fact sizes. Both quantities could have been declared as necessary input parameters, but the improvement of the cost estimates don't seem to compensate the additional expenses for the one using the rating function. The same holds for the severe simplification regarding the JSFs. To force the specification of all values means increasing the amount of base values needed vehemently forfeiting the abstract character of the algorithm.

On the other hand all simplifications made can be replaced by the correct values with little effort allowing for a more precise rating.

REFERENCES

Brant, D., Grose, T., Lofaso, B., and Miranker, D. (1991). Effects of Database Size on Rule System Performance:Five Case Studies. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)*.

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming expert systems in OPS5: an introduction to rule-based programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Cárdenas, A. F. (1975). Analysis and performance of inverted data base structures. *Commun. ACM*, 18(5):253–263.

Forgy, C. L. (1981). OPS5 User's Manual. Technical report, Department of Computer Science, Carnegie-Mellon University.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37.

Hanson, E. N. (1993). Gator: A Discrimination Network Structure for Active Database Rule Condition Matching. Technical report, University of Florida.

Hanson, E. N. and Hasan, M. S. (1993). Gator: An Optimized Discrimination Network for Active Database Rule Condition Testing. Technical report, University of Florida.

Jamocha (2006). Jamocha Project Page. http://www.jamocha.org, http://sourceforge.net/projects/jamocha.

Miranker, D. P. (1987). TREAT: A Better Match Algorithm for AI Production Systems; Long Version. Technical report, University of Texas at Austin, Austin, TX, USA.

Winston, P. H. (1992). *Artificial intelligence*. Addison-Wesley.

Yao, S. B. (1977). Approximating block accesses in database organizations. *Commun. ACM*, 20(4):260–261.