Design and Organisation of Autonomous Systems

Resource Allocation in Distributed Perception Networks

Sietse Dijkstra
sdijkstr@science.uva.nl

Felix Hageloh
fhageloh@science.uva.nl

Max Pfingsthorn
mpfingst@science.uva.nl

Koen van de Sande
ksande@science.uva.nl

February 3, 2005

Abstract

We introduce a resource allocation algorithm for a multi agent-based data fusion network. This network is called a Distributed Perception network in literature. The resource allocation algorithm is based on the cumulative absolute entropy change in the beliefs at the root of the data fusion network. Our algorithm can be efficiently computed locally at the agents, based on a matrix encoding of a Bayesian network. Only very small matrices have to be sent among the agents. Based on just these matrices an agent that is faced with a resource conflict can make an informed decision about which network to connect to.

1 Introduction

Pavlin et al [Pav04] introduced a multi agent-based approach to data fusion. They use a collection of agents called a *Distributed Perception Network* (DPN), as opposed to a centralized approach to data fusion typically used.

Agents in a DPN in general can provide a *service* to other agents upon request. Fusion agents may depend on the services of other agents since they fuse together information. Services are beliefs about some concept variable; the beliefs form a probability distribution over the variable. The fusion agents are able to reason about their own beliefs, given the input received from other agents, by means of an internal Bayesian network.

The internal Bayesian network is constrained to have only a single root node which corresponds to the service provided by the agent. Every leaf node of the Bayesian network should correspond to a service needed. Apart from these two constraints the form of the Bayesian network can be arbitrarily complex.

Beliefs from other agents needed by a fusion agent are received on-demand. Only when a request is made for the service provided by an agent, it will try to obtain the services it depends on. If it cannot find an agent willing to provide the service, it will assume an uniform distribution to represent a complete lack of knowledge about that service. When the agent receives an updated belief for one of its dependencies, it will update its own belief accordingly, see De Oude et al [Oud05] for a detailed explanation.

Besides fusion agents, there are also sensor agents which just provide a service, for example by quering a physical sensor. These readings of the physical sensor are then turned into a belief which can be provided to another agent.

The DPN networks currently implemented only support binary variables, because currently the emphasis is on beliefs about the existence of a certain concept, e.g. there is a 90% certainty that there is a fire. DPNs are more general than the binary case and can be applied to any variables which take only a finite number of values.

A DPN is a distributed respresentation of a Bayesian network; the correspondence is shown in [Oud05, Pav04].

While the agents in a DPN are looking for others which are able to provide the service they need, it can come to conflicts when multiple agents try to connect to the same agent. This might be prevented by careful design of the DPN agents and their roles, but in general, we would like the DPN to be as flexible and therefore as powerful as possible. If we have a DPN with a lot of fusion agents and a limited number of sensor agents, then we do not want the DPN to break down if there is a conflict over a sensor agent. If there is such a resource conflict, then we need to resolve it. In section 2 we describe situations in which resource conflicts can occur and how they can be addressed.

In figure 1 an example of a resource conflict occuring between two DPNs is shown. We have two main concepts that we can test for, *Fire* and *Gas Leak*. This presence of these concepts cause other events to occur, like that *smoke* or *heat* is detected in the case of fire. Both DPNs end up needing the *CO* sensor for their causal model (e.g. *Fire* causes *smoke*, which causes *CO* to be detectable, which in turn causes the *CO* sensor to detect that gas). This constitutes a resource

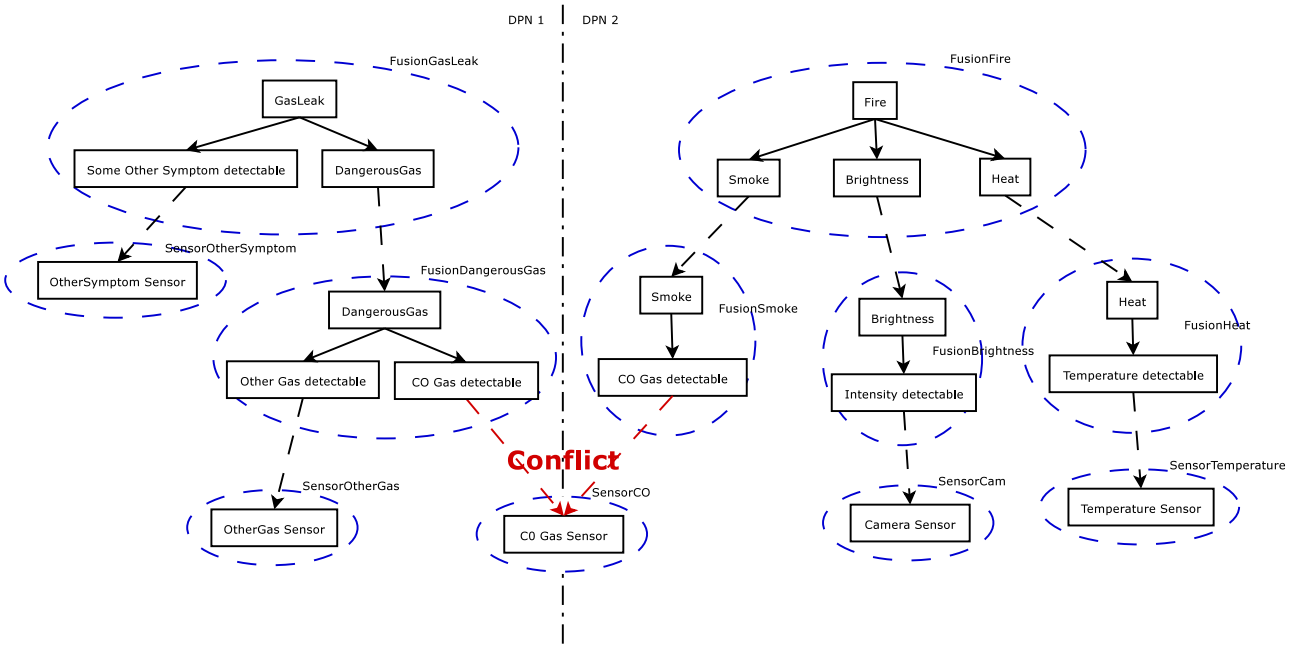


Figure 1: An example of a resource conflict that can occur between two Distributed Perception Networks. These networks were implemented as a test case.

conflict introduced before.

Nunnink et al [Nun05] suggest a solution to the resource conflict by allocating the resource to the DPN based on a change in entropy. The goal of this paper is to demonstrate how their solution can work in the existing DPN framework and to suggest novel ideas for optimizing this approach to resource allocation. In section 3 we discuss their approach in more detail. In section 4 we describe the architecture of our implementation of this approach and finally in section 5 we evaluate the computational cost of this approach.

2 Scenarios

To solve a resource allocation problem in a DPN, there are several possible prototypical scenarios we need to be aware of. These scenarios describe possible combinations of factors that can influence an algorithm to optimally allocate the resource in question.

For some types of sensors however, we may not need sophisticated techniques to allocate it to exactly one parent as the sensor may not depend on settable parameters. This would be true for a sensor which senses only one property of its environment, for example a simple illumination sensor which only says if there is light or not. Actually, for this sensor, it would be optimal to send its output to multiple connected agents to maximize efficiency. This kind of behavior will be termed 'shareable' in this article and does not require any processing at all on the part of resource allocation techniques.

As a general approach for those sensors that need resource allocation, we will use a measure of priority acquired from the requesting agents to find out which one of them needs the service of this resource the most at a certain time.

There are several complications to this approach due to the distributed nature of the DPN. Message sending and computation of the priority measure will take different amounts of time. Hence, it will take some finite time to gather all necessary data to make the decision on allocation. To optimally assign the resource in question in the least time possible, different algorithms have to be applied according to the nature of the resource.

Of importance are how fast a resource can change its parameters (e.g. which gas to sense in a gas sensor, which temperature threshold in a temperature sensor, which way to look with a pan-tilt camera, etc) and how fast the sensor can supply a sensible inference from the gathered data (e.g. a temperature sensor is fast to know if the measured temperature is above or below a set threshold, but a gas sensor might need multiple measurements to get a stable reading). We investigate four cases highlighting the possible combinations of these two variables.

2.1 Case A: Temperature Sensor

A temperature sensor is a simple device and we assume it takes negligible time to measure the current temperature. Also, setting a threshold in the agent software is fast. In this setting, the resource allocation problem boils down to a simple question of whether to reassign the resource to a competing agent or not if the sensor is already connected. Theoretically it makes sense to use resource allocation with the tempature sensor,

but in practice the sensor can serve multiple DPNs if it switches parameters quick enough. This effectively makes this type of sensor 'shareable'.

2.2 Case B: Gas Sensor

A gas sensor is also fast when setting parameters, like a threshold value or the kind of gas to sense, but presumably slow when sensing. This might be because the sensor needs to do some filtering, like averaging, before a stable reading is obtained. In such a setting, it makes sense to do resource allocation. We need to identify an algorithm that solves the problem sufficiently and with proper respect of the time constraints.

Since this kind of sensor needs some finite time to supply a value, we cannot reassign the sensor while it is sensing. Therefore, we need to know when the sensor is done sensing in order to delay the reassignment. However, we can already do some processing (like calculating priority measures) while the sensor is waiting for the new data to make the decision easier later. This can be done by the following algorithm:

- If the sensor is not yet connected, connect immediately.
- If the sensor is already connected, enqueue the incoming request and ask for the priority measure from all agents in the queue to make sure we have recent priorities from all of them.
- When a priority measure is returned, reorder the queue according to the priority.
- Once the sensor is done with a sensing interval, reassign the sensor to the first agent in the queue.

Using a priority queue to store the requests for this sensor will effectively guarantee that the sensor will always send its data to the agent that needs it the most.

2.3 Case C: Pan-Tilt Camera (with fast processing)

The Pan-Tilt Camera will be rather slow when setting its parameters, e.g. changing the direction we want the camera to point to. On the other hand, due to only limited, maybe even hardware based, signal processing (e.g. for color based fire detection with a thermal camera), data will be availably quickly after the parameters are set. This results in a different kind of time contraint than the previous case. Actually, we can approach this problem in two different ways:

1. A sensor must send at least one output to a connected agent.
2. A sensor may be reassigned before it has sent the first output to the connected agent, e.g. while it is still rotating the camera.

In the first approach, the algorithm will be the same as in Case B. In the second approach, we do not need to wait for the sensor to finish so we can reassign the sensor immediately after a priority measure is returned, if needed. Care has to be taken if the second approach is chosen, since it potentially introduces deadlock situations.

2.4 Case D: SMS Cellphone Question Sensor

This sensor receives a question as a parameter and the user of the cellphone will then answer the question (usually a yes/no question) and thus supply the sensor value. Obviously, answering a question takes a long time when compared to the temperature sensor in Case A. Also, setting the parameter takes quite long, as the person answering the question will have to read and understand it first. Thus, we are left with the last and probably most interesting setting of them all: Both reading the sensor data and setting the parameters take a long time.

This time, unlike in Case C, we do not have a choice whether or not we can interrupt the sensor while setting the parameters. Once we have connected to an agent and that one has posed a question, we have to follow through and answer that question, otherwise, the person will not be efficient in answering any query. Since this situation is the same as the first approach in the previous Case C, we can again solve it with the algorithm of Case B.

3 Theory

3.1 Impact measure

The main problem is to find a priority measure for the resource allocation, which can be computed efficiently. Nunnink [Nun05] proposes the use of Shannon entropy as a metric and the cumulative absolute change in entropy as a measure for which DPN will benefit the most from the allocation of the resource. Shannon entropy is given by:

$$\mathcal{H}(P(X)) = -\sum_{x_i \in X} P(x_i) \log P(x_i) \qquad (1)$$

From this we can define the absolute change in entropy:

$$\Delta\mathcal{H}(P(H)) = |\mathcal{H}(P(H|\mathcal{E})) - \mathcal{H}(P(H|\mathcal{E} \cup S))| \qquad (2)$$

Where H is the DPN's hypothesis variable, $\mathcal{E}$ is the set of all current evidence, S is the new evidence that will be obtained through allocation of the resource.

The cumulative change in entropy $\sum \Delta\mathcal{H}$ is then obtained by summing over all possible instantiations of the sensor node S. This is necessary because we do not know beforehand what the input of the sensor will

be. Depending on the input the hypothesis change on the root node will differ so we need to account for all input possibilities.

From (2) we can see that we need to know the conditional probability of the root hypothesis given all the evidence with and without the sensor input to compute $\Delta\mathcal{H}$. This information has to be made available to the sensor agent in order to be able to decide which DPN to connect to. Note that the output of the root node at the root agent is exactly $P(H, \mathcal{E})$. From this we can compute $P(H|\mathcal{E})$ using Bayes rule as follows:

$$P(H = h_i|\mathcal{E}) = \frac{P(H = h_i, \mathcal{E})}{\sum_{j=1}^{m} P(H = h_j, \mathcal{E})} \tag{3}$$

Where m is the number of hypothesis states, $H = \{h_1, h_2, \ldots, h_m\}$.

However, this still leaves the problem how to obtain $P(H, S)$ and how to send it to the sensor agent. A straight forward way would be to use classical belief propagation methods for Bayesian networks. This means treating the distributed Bayesian network of the DPN as single monolithic Bayesian network and using methods such as $\lambda\pi$ propagation or junction tree to compute $P(H, S)$. However, both methods are NP-hard [Coo90] and we also would need several propagation cycles to get the result. For each possible sensor instantiation we have to do one propagation and this for every DPN that is requesting the sensor. Since we want to keep resource allocation as inexpensive as possible, this might not be a good option.

3.2 Parametric approach

A potentially more efficient approach using matrices is suggested in [Nun05]. It is based on the observation that any joint probability in a Bayesian network can be expressed as a linear function of any model parameter. By treating the input of the sensor agent S as soft evidence it becomes a model parameter, so in a monolithic DPN we could express $P(H, \mathcal{E} \cup S)$ as function of S in matrix form as follows:

$$P(H, \mathcal{E} \cup S) = \begin{pmatrix} c_{1,1} & \cdots & c_{1,k} \\ \vdots & \ddots & \vdots \\ c_{m,1} & \cdots & c_{m,k} \end{pmatrix} \cdot P(S) \tag{4}$$

Where k is the number of possible sensor states, and m is the number of hypothesis states of the root node.

However, in a DPN we do not have the entire monolithic Bayesian network available and every agent can only compute the joint probability given its own local network. Yet we know that the root node of an agent is a leaf node of its parent agent. Hence, in a chain of agents $A_1 \ldots A_n$ we can express the joint probability of the root node R_j of agent A_j as a function of its predecessor's root node R_{j+1} in the same way. This is shown in [Nun05]:

$$\begin{aligned} P(R_j, \mathcal{E}) &= \begin{pmatrix} c_{1,1} & \cdots & c_{1,k} \\ \vdots & \ddots & \vdots \\ c_{m,1} & \cdots & c_{m,k} \end{pmatrix} \cdot P(R_{j+1}) \\ &= \mathbf{C}_{j+1}^{j} \cdot P(R_{j+1}) \end{aligned} \tag{5}$$

Thus the linear transformation matrix $\mathbf{C}$ is essentially an encoding of the local network inside agent A_j in terms if its root node. For the resource allocation problem we can now define such a chain of agents from the root agent all the way down to the sensor agent and encode each local network in terms of the overlapping node R_j in such a matrix. Moreover, it was shown in [Nun05] that all those local matrices can be combined using simple matrix multiplication as follows:

$$\mathbf{C}_{n}^{j} = \mathbf{C}_{j+1}^{j} \cdot \mathbf{E}_{j+2}^{j+1} \ldots \mathbf{E}_{j+n}^{j+n-1} = \mathbf{M} \tag{6}$$

Where $\mathbf{E}$ can be obtained by dividing each row in $\mathbf{C}$ by its corresponding component of the prior vector of the root node R_j.

The resulting matrix $\mathbf{M}$ hence encodes the complete mapping from sensor input all the way to the root hypothesis and can thus be used to compute $P(H, \mathcal{E} \cup S)$ and $P(H, \mathcal{E})$!

3.3 Ways to compute the local matrix

This leaves open the last problem, namely how to compute those local $\mathbf{C}$ and $\mathbf{E}$ matrices. From its definition in (5) we can see that for example the first column of $\mathbf{C}$ namely $(c_1 1, \ldots, c_m 1)$ has to be equal to $P(R_j, \mathcal{E})$ for $P(R_{j+1}) = (1; 0; \ldots; 0)^T$. Hence, we can obtain all coefficients by setting $P(R_{j+1})$ to convenient values and acquiring $P(R_j, \mathcal{E})$ through standard propagation methods (like junction tree etc) [Nun05].

As mentioned before, standard belief propagation in a Bayesian network can be very expensive, so we would like to consider possible speedups for these local matrix computations. In [Nun05] a speedup is described, which is based on the observation that the coefficients in the $\mathbf{C}$ matrix, i.e. the encoding of the local network, depend on network parameters and the evidence coming in from the leaf nodes. The only thing that changes in the network is the evidence while the network parameters stay constant. So it seems that we could do some precomputation and only fill in the evidence of the available nodes when the $\mathbf{C}$ matrix is requested. As described before, when computing the $\mathbf{C}$ matrix we express the joint probability at the root node (the "output") as a function of soft evidence coming in from a single leaf node. If we express the joint probability as a function of *all* the leaf nodes, the resulting linear transformation matrix $\mathbf{Y}$ now only depends on the network parameters and is thus constant. When now a request for an encoding of the network as a function of one specific leaf node comes in we just have to multiply the evidence from all the remaining

leaf nodes. This can be more efficient than computing the **C** matrix form scratch (using propagation) every time.

3.4 Precomputing coefficient matrix Y

We can compute **Y** during the initialization phase of an agent in a similar way as we can compute **C** using standard propagation. If we look at an example network with one root node R and two leaf nodes A and B with binary states, meaning a concept can be either *true* or *false* [Nun05], the joint probability $P(R = r, \mathcal{E})$ would be computed using the coefficients of **Y** as follows:

$$\begin{aligned} P(r, \mathcal{E}) =& y_1 P(a)P(b) + y_2 P(\bar{a})P(b) + \\ & y_3 P(a)P(\bar{b}) + y_4 P(\bar{a})P(\bar{b}) \end{aligned} \tag{7}$$

Where $P(a)$ means the probability of A being true and $P(\bar{a})$ the probability of A being false. From this we can see that again we can compute the coefficients by initializing the (soft) evidence of the leaf nodes to convenient values, namely their extreme cases like $P(a) = 1$ or $P(a) = 0$. From (7) we can see that if we set $P(a) = 1$, $P(\bar{a}) = 0$, $P(b) = 1$ and $P(\bar{b}) = 0$ we get $P(r, \mathcal{E}) = y_1$. Again using standard propagation to obtain $P(r, \mathcal{E})$ and repeating this for all possible combinations of extreme evidence input we can compute all entries of the **Y** matrix. To conceptualize this we might say that since we encode the network as a matrix leaving all the evidence as a variable we just have to take care of the extreme cases of the evidence and the linear transformation will do the rest. From this definition of $y_1 \dots y_n$ we can see that the labeling of y is arbitrary as it depends on the order of evidence state combinations we chose. In our example y_1 corresponds to $P(a)P(b)$, but it might have as well been chosen to correspond to $P(\bar{a})P(b)$. This is important because we need to later fill in the evidence of the non-variable leaf nodes and we have to make sure that it gets multiplied with the correct coefficient in the **Y** matrix.

3.5 Computing the local matrix using Y

To continue with our example, if we now were to receive a request for the **C** matrix with A being the variable node, we can fill in the evidence about B as follows:

$$\begin{aligned} P(r, \mathcal{E}) =& [y_1 P(b) + y_3 P(\bar{b})]P(a) + \\ & [y_2 P(b) + y_4 P(\bar{b})]P(\bar{a}) \end{aligned} \tag{8}$$

The sums inside the square brackets represent one row of the **C** matrix. We can see again that the chosen ordering of the y's is important. This equation will give us one row of the **C** matrix as each row corresponds to a different state of the root hypothesis R. As we can see all products that contained $P(a)$ in (7) are summed to form one column of the **C** matrix. Likewise all products that contained $P(\bar{a})$ are summed, forming the second column of **C**. However, given just the **Y** matrix we have no way of knowing which entry y_i appeared in a product with $P(a)$ and which appeared in a product with $P(\bar{a})$ when we originally computed the **Y** matrix. As an implementation trick we can use a state generator to compute **Y**, which produces all possible soft evidence combinations of leaf concepts in a standard way, as for example using a binary counter. The set of evidence for our example would then look like this:

$$\mathcal{E} = \begin{bmatrix} (P(a) = 1, P(\bar{a}) = 0) & (P(b) = 1, P(\bar{b}) = 0) \\ (P(a) = 1, P(\bar{a}) = 0) & (P(b) = 0, P(\bar{b}) = 1) \\ (P(a) = 0, P(\bar{a}) = 1) & (P(b) = 1, P(\bar{b}) = 0) \\ (P(a) = 0, P(\bar{a}) = 1) & (P(b) = 0, P(\bar{b}) = 1) \end{bmatrix} \tag{9}$$

When we later have to compute the **C** matrix we use the same generator to produce all state combinations. Then we need to fill in the actual evidence from the corresponding leaf nodes and multiply them for each row in (9) (see (8)). This way we generate the exact same products as when computing **Y**. Putting them in a row vector $\overrightarrow{ev}$ and multiplying it with the first row of **Y** will replicate equation (8). However those products will still contain $P(A)$ which is now a free variable and has no instantiated soft evidence. Moreover, we want only the terms containing one of the two states, $P(a)$ or $P(\bar{a})$, being summed. As another trick we can first instantiate $P(A)$ as $P(a) = 1$ and $P(\bar{a}) = 0$. This way all products containing $P(\bar{a})$ will be zero and effectively only products containing $P(a)$ will be summed. Thus we can produce the first column of our **C** matrix through calculating $\mathbf{Y} \cdot ev$. The second column can then be obtained by producing another row vector where we set $P(a) = 0$ and $P(\bar{a}) = 1$, in the same way. In the non binary case we would have to repeat this for each state of A, always setting the probability for only one state to 1 and the rest to 0. This way we produce a row vector of multiplied soft evidence $\overrightarrow{ev_i}$ for each state of A. For each $\overrightarrow{ev_i}$ multiplied with **Y** we get one column of the **C** matrix. Hence we can combine all $\overrightarrow{ev_i}$'s in an Matrix **Ev** and obtain **C** with a single matrix multiplication, namely $\mathbf{C} = \mathbf{Y} \cdot \mathbf{Ev}$.

3.6 Computing the resource impact using matrix M

Our sensor agent will now eventually end up with an **M** matrix for each requesting DPN which encodes the complete mapping from the sensor input to the root hypothesis of the corresponding DPN. It now needs to compute the cumulative absolute entropy change for each DPN using this **M** matrix. Note that **M** matrix multiplied with the sensor input will give us $P(R, \mathcal{E} \cup S)$. If we set the (soft) evidence of the sensor to the uniform distribution[1], we have the case where there

[1] In the binary case this means setting them to 0.5.

is no input from the sensor, which will then give us $P(R, \mathcal{E})$. Then using Bayes rule we can compute the corresponding joint probabilities as described earlier (3). Repeating this for all possible instantiations of the sensor and using the definition of entropy (1) we can finally compute $\sum \Delta\mathcal{H}$.

4 Architecture

In the current implementation every agent in the DPN consists of several plug-ins. These plug-ins can communicate by exchanging messages through the so-called blackboard; this blackboard service is provided by Cougaar [Cougaar]. In the rest of our discussion we will say that messages are sent from one plug-in to another, while in fact this is done indirectly through the blackboard. By design the only plug-in allowed to send and receive inter-agent messages is the communication engine (CE).

All agents initially set up their communication through the Contract Net Protocol [Smi80], which consists of initially sending a *CallForProposal* if an agent wants a service. One or more agents that are able to provide the service can respond with a *Bid*. The requesting agent will pick one of the agents it has received a bid from and finally send a *FusionContract* to signal that they are committed to each other.

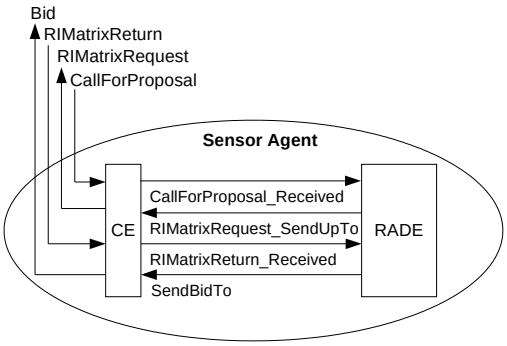


Figure 2: Messages sent and received by the sensor agent are indicated by arrows. Squares indicate plug-ins, while the entire circle denotes the agent. CE is the communication engine plug-in, RADE is the resource allocation decision engine plug-in.

A sensor agent who is going to run into resource allocation problems is initially waiting for *CallForProposal* messages (see figure 2). This proposal is forwarded to another plug-in, the resource allocation decision engine (RADE), which decides whether the agent should respond with a bid. If the agent is available or can be shared by multiple agents, then the RADE plug-in will send a *SendBidTo* message to the communication engine so that it can respond to the proposal with a *Bid* message. If the agent is currently in use and cannot be used by multiple agents at the same time, then it will want to request a resource impact matrix (abbreviated as RIMatrix) from the DPN attempting to connect, so that it can make an informed decision about which DPN to provide its services to.

The resource impact matrix request will first be sent from the RADE plug-in to the communication engine as a *RIMatrixRequest_SendUpTo* message, which in turn will send a *RIMatrixRequest* to the higher level agent. The request messages will contain the name of the sensor agent who initiated the request and a request number, which together uniquely identify the request and thus form a key. This key can be associated with routing information.

Since the *RIMatrixRequest* message is an inter-agent message, it is received by the communication engine (CE) of the higher level fusion agent (see figure 3). The communication engine adds the name of the lower level agent where the request came from to the message and forwards the request to a plug-in called the resource allocation engine (RAE); this messages is called *RIMatrixRequest_Received*. Using the information contained in the message the resource allocation engine can construct a routing table which can be used to trace the path back down again later on.

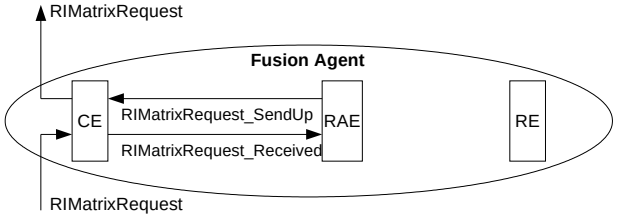


Figure 3: Messages going up towards the root node through a fusion agent are indicated by arrows. Squares indicate plug-ins; CE is the communication engine, RAE is the resource allocation engine. The reasoning engine (RE) is not used on the way up.

After the routing information has been stored, the resource allocation engine forwards the request further up towards the root of the DPN network by sending a *RIMatrixRequest_SendUp* to the communication engine, which sends this to its parent as a *RIMatrixRequest*. At the parent the same process as in figure 3 is performed, unless it is the root of the DPN network. If it is the root of the DPN network then this is detected in the communication engine upon receiving a *RIMatrixRequest_SendUp* message, which is then turned into a *RIMatrixReturn* message with an empty resource impact matrix. When the communication engine receives such a message, it forwards it to the resource allocation engine (see figure 4). This plug-in stores the matrix in the routing table and requests the matrix encoding of the local Bayesian network from the reasoning engine (RE) by issueing a *LocalMatrix_Request*. This message contains a flag whether this agent is the DPN root[2]; if

[2]The agent is the DPN root if the matrix received from the

it is, then the plug-in will return the C matrix, otherwise it will return the E matrix.

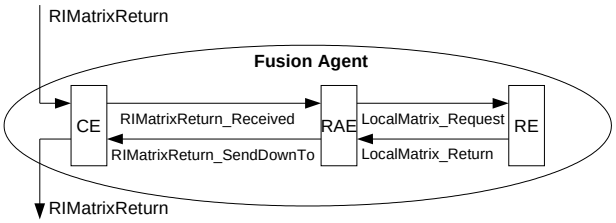


Figure 4: Messages going down from the root node to the sensor agent through a fusion agent are indicated by arrows. Squares indicate plug-ins, the shorthands are the same as in figure 3.

When the resource allocation engine receives the matrix encoding of the Bayesian network, it will combine this matrix with the matrix received from the parent, which was stored in the routing table, by multiplying them together. The resulting matrix is sent down to the lower level agent where the request came from using the information stored in the routing table. Again, this is done indirectly by first sending a *RIMatrixReturn_SendDownTo* to the communication engine which turns this into a *RIMatrixReturn* message. This procedure is the same for all fusion agents in the network.

Eventually the *RIMatrixReturn* message will reach the sensor agent where the request originated (see figure 2). The communication engine will forward the matrix to the resource allocation decision engine, which then needs to make a decision based on the matrices it has received. In the next section we will discuss possible decision policies.

In our description of our architecture we have simplified the names of messages and plug-ins somewhat. All intra-agent messages have a *_INM* suffix while all inter-agent messages have a *_EXM* suffix. Existing plug-ins which needed extension where subclassed and received a *RAE* (*R*esource *A*llocation *E*nabled) prefix to their original name. Sensor agent plug-ins are different from fusion agent plug-ins and receive a *SA* prefix. For example, the sensor agent communication engine was called SACommunicationEngine and our subclass of this plug-in was called RAESACommunicationEngine.

4.1 Resource allocation decision policies

We implemented three different allocation policies, 'shareable', 'random', and 'resourceimpact'. The policy is a configurable parameter for the sensor agent.

The 'shareable' policy means that we completely circumvent the resource allocation and answer to any *CallForProposal* message we receive with a bid. This is used for sensors like the temperature sensor in Case A of section 2.

parent is empty

The 'random' policy implements random assignment, meaning the sensor is reassigned with a given probability. This is the simplest solution to the resource allocation problem and can be used to test the system.

The 'resourceimpact' policy uses our archictecture from the previous sections to request resource impact matrices and reassigns the sensor agent based on a priority measure calculated from these matrices.

After surveying all the possible scenarios that may occur in section 2, it seems clear that we really only have to implement one algorithm to make the decision about resource allocation for this policy. This algorithm should use a priority queue to keep track of requesting agents and update their priority measure as frequently as possible. The priority measures are calculated from the resource impact matrices using the cumulative absolute entropy change, as described in section 3.

The complete algorithm is as follows:

- If the sensor is not yet connected, connect immediately to the requesting agent.
- If the sensor is already connected to an agent, enqueue the incoming request and ask all agents in the queue for their resource impact matrix.
- When a resource impact matrix is returned, compute the cumulative absolute entropy change and reorder the priority queue accordingly.
- When the sensor finishes a sensing cycle, reassign if the agent with the highest priority is not the currently connected agent.

At a later stage, the algorithm should also take an argument whether or not to wait for the sensor to finish sensing before reassigning the sensor. This is not yet implemented due to time constraints. For now, the sensor always gets to finish its sensing once it starts: it cannot be interrupted.

5 Complexity

In section 3 two ways to compute the local encoding of the Bayesian network (the **C** matrix) were discussed. Precomputing the coefficients using the **Y** matrix was introduced as a possible speedup. Now we will compare the two methods and check if that is indeed the case.

The computational complexity of using precomputed coefficients can be determined in a straight forward manner. Of course we only consider the number of operations (multiplications) at run time and not during the initialization (precomputation) phase. We count the number of multiplications only as they are the most time consuming.

First we need to fill in the evidence for all possible evidence combinations of the leaf nodes and then multiply for each combination. Assuming that all leaf

nodes have the same number of possible states (s), the number of state combination is s^n where n is the number of leaf nodes of the local Bayesian network. Then for each state combination we have to perform n multiplications for the probability from each leaf node. This process has to be repeated for all possible instantiations of the variable leaf node, which is also s. So the total number of multiplications is:

$$s \cdot s^n \cdot n = s^{n+1} \cdot n \tag{10}$$

Furthermore, we have to do the matrix multiplication of the coefficient matrix $\mathbf{Y}$ with the combined multiplied evidence matrix $\mathbf{Ev}$. If m is the number of root node hypothesis states, then $\mathbf{Y}$ is an $m \times s^n$ matrix. The dimensions of $\mathbf{Ev}$ are $s^n \times s$ which leads to $h \cdot s^n \cdot s$ multiplications for $\mathbf{Y} \cdot \mathbf{Ev}$. So the total number of multiplications to compute $\mathbf{C}$ or $\mathbf{E}$(!) is

$$s^{n+1} \cdot n + h \cdot s^n \cdot s = s^{n+1} \cdot (n + h) \tag{11}$$

This shows that the time complexity of this method is of $\mathcal{O}(c^n)$ for the number of leaf nodes n (where c is a constant).

Calculating the number of multiplications for standart propagation is less straight forward. Many different algorithms exist, but in this paper we shall only consider the $\lambda\pi$ algorithm. First of all we can observe that the worst case time complexity of $\lambda\pi$ is $\mathcal{O}(s^n)$ where n is the number variable nodes. However, the actual number multiplications heavily depends on the "shape" of the particular Bayesian network. The DPN already restricts the Bayesian network to have only one root node. Moreover, we will consider here the number of multiplications only for three example classes/shapes of Bayesian networks (5) to give a general comparison. A more detailed comparison is not possible at this point due to time constraints and has to be left to further research.

Examples

First we will consider a network as given in figure 5(a). Each node only has one child node so that they effectively build a chain. Assuming that each node is a variable node the number of multiplications to compute the joint probability at the root node can be shown to be:

$$(m - 1) \cdot s^2 \tag{12}$$

Where s is again the number of states for each node (assuming they are all the same) and m is the number of (variable) nodes. To compute the $\mathbf{C}$ matrix we have to do s inferences of this type for each state of the variable leaf node. So the total number of multiplications for this example is:

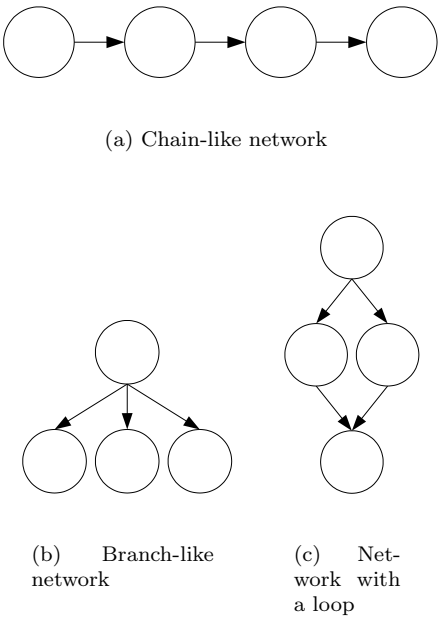
$$s \cdot ((m - 1) \cdot s^2) = (m - 1) \cdot s^3 \tag{13}$$

For the next example we consider a network with only one layer of leaf nodes and one root node given in 5(b). In this case the number of multiplications for the joint probability at the root node is:

$$b \cdot (s + s^2) \tag{14}$$

Where b is the number of branches of the root node. In this case this is the same as the number of leaf nodes, so we can replace b with n. Also we have to repeat this for the number of states s for the variable leaf node, which gives a total number of multiplications of:

$$s \cdot n(s + s^2) = n(s^2 + s^3) \tag{15}$$



(a) Chain-like network

(b) Branch-like network

(c) Network with a loop

Figure 5: Different Bayesian network topologies.

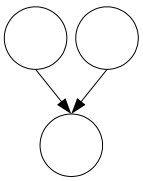


Figure 6: Node with multiple parents.

The last example contains a loop and is depicted in figure 5(c). This is a rather complex case and can only be solved using $\lambda\pi$ messages with a trick (see [Xia02]). Basically the loops have to be cut open resulting in several graphs, one for each possible state of the branching node. If k is the number of loops and the number of states s for each branching node is assumed to be the same, this will lead to s^k graphs. For each graph we then use the normal $\lambda\pi$ algorithm. The total number of multiplications then depends again on the shape of the resulting splitted graphs. In the best case this will

be a chain as in figure 6 with only two nodes. However, the resulting graph can more complicated as depicted in figure 6. A node having multiple parents greatly increases the number of multiplications for $\lambda\pi$. It can be shown to be:

$$s + s^2 + \cdots + s^p + s^{p+1} \tag{16}$$

Moreover, we also have to repeat this for the number of states s for the variable leaf node which leads to $s^2 + \ldots s^{p+2}$ multiplications. So the total number of multiplications is (17) in the best case, or (18) in the worst case.

$$s^k \cdot (2-1) \cdot s^3 = s^{k+3} \tag{17}$$

$$s^k \cdot (s^2 + \ldots s^{p+2}) \tag{18}$$

As a conclusion we can say that there are some cases where standard propagation is more efficient to compute the $\mathbf{C}$ matrix. Also we have to consider that other methods of propagation exist that are more efficient than $\lambda\pi$ for certain classes of Bayesian networks and might thus also outperform the precomputation method. However, the advantage of the precomputation method is that is applicable to any type of Bayesian network and its complexity only depends on the number of leaf nodes. It is insensitive to parameters like branching factors or the number of loops. Thus, it can be much faster on Bayesian networks that have a complex internal structure but only a small number of leaf nodes. One can imagine that we could work out some kind of heuristic that includes all these factors (number of leaf nodes, number of levels, branching factor and loops) in order to determine when which method is more efficient. If this is done in an efficient way it could constitute a big speedup for resource allocation in a DPN. Clearly more research is needed in order to determine a useful heuristic for this.

6 Conclusion

We have analyzed different scenarios to see when it makes sense to do resource allocation in a DPN. When a sensor can be configured quickly and provide sensing information quickly, then it does not make sense to do resource allocation because the time it will take to make an informed decision will be longer than the time it would take to just get a sensor value. If the sensor setup time is long or the sensor needs to accumulate data first before it can provide a sensor value, then it does make sense to do resource allocation.

We have introduced a decision algorithm based on a priority measure which can be used for sensors which cannot be interrupted while sensing. Future extensions to this algorithms to support sensor interruption are fairly simple, though it will make sense then to look at operating system scheduling algorithms, especially the ones which support pre-emption.

As a priority measure we have investigated the cumulative absolute entropy change at the root of the DPN network if the resource is assigned. We calculate this measure from the so-called resource impact matrix. This resource impact matrix is constructed through matrix multiplication from local matrices in which the Bayesian network of an agent is encoded. This local matrix can be efficiently computed if some precomputation is done. The main advantage of this approach is that only only small matrices need to be passed around and most computation can be done locally. Also, only agents along the path from the sensor agent to the root agent need to be involved in computation, as opposed to traditional propagation algorithms where all agents are involved. Another disadvantage of traditional propagation algorithms is that calculations have to be done during the propagation: nothing can be precomputed. In our approach the agent could perform precomputations when it is idle.

References

[Coo90] G. F. Cooper The computational complexity of probabilistic inference using Bayesian belief networks models approach Artificial Intelligence, Volume 42, Issue 2-3 1990

[Cougaar] Aaron Helsinger, Todd Wright. Cougaar: A Robust Configurable Multi Agent Platform. submitted to *IEEE Aerospace Conference 2005*, BBN Technologies, Cambridge, MA, USA, 2005

[Nun05] Jan Nunnink and Gregor Pavlin. A Probabilistic Approach to Resource Allocation in Distributed Fusion Systems. Informatics Institute, UvA, 2005

[Oud05] Patrick de Oude, Brammert Ottens, and Gregor Pavlin. Information fusion with distributed probabilistic networks. Informatics Institute, UvA, 2005

[Pav04] Gregor Pavlin, Marinus Maris and Jan Nunnink. An Agent Based Approach to Distributed Data and Information Fusion. Informatics Institute, UvA, 2004

[Smi80] R. Smith. The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver. IEEE Transactions on Computers C-29(12):1104-1113, 1980

[Xia02] Yang Xiang. Probabilistic Reasoning in Multiagent Systems - a graphical models approach. Cambrige University Press, 2002