Learning Others' Calendars

Akiva Leffert

June 2006
CMU-CS-06-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the Senior Honors Thesis in the School of Computer Science.

The work in this report is based within the context of the RADAR project, supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), or the Department of Interior-National Business Center (DOI-NBC).

Keywords: calendar learning, meeting scheduling

Abstract

This work develops a method for aiding the process of meeting scheduling through learning about the meetings in the calendars of other users. We assume users do not share their entire calendars. This makes it difficult to determine the exact state of another user's calendar and represent it using a traditional calendar. We solve this problem by representing another agent's calendar as a probability distribution of possible meeting types and present an algorithm called *LOC* (*L*earning *O*thers' *C*alendars) for learning these distributions based on responses to meeting requests. We then present a modification to *LOC* which uses this information to guide the process of selecting time slots to decrease the number of messages sent during the meeting negotiation process. We implemented these algorithms and ran experiments to test them. We found they successfully learned others' calendars and the second version sent fewer messages than a system which did not leverage the learning information. This shows that calendar learning can aid the scheduling process. Our work integrates into the CMRadar project.

1 Introduction

In many organizations, people need to schedule meetings with each other. Meeting scheduling is typically done over email and can be a significant cognitive burden on computer users. It involves weighing multiple options, keeping track of communication, finding mutually agreeable time slots from a potentially large set, and possibly moving existing meetings. People typically find this both difficult and tedious. Hence, it is no surprise that there have been a number of attempts to automate the process. These efforts have primarily had two goals: to learn user preferences and to increase schedule quality. Examples of user preferences include a preference for afternoons over mornings and a preference for continuous blocks of meetings versus more spread out meetings with breaks.

One effort to learn user preferences was CAP[3], which used a rule induction system based on user feedback to suggest to users what times to select when adding meetings to their calendars. Sen and Durfee[5] tried to solve the problem of scheduling meetings by focusing on the negotiation protocol and tweaking the parameters to the protocol they developed. Garrido and Sycara[2] used a negotiation algorithm based on gradually moving away from the user's preferred times. More recently, both the CMRadar[4] and CALO[1] projects are working toward the goal of a fully featured learning agent for handling meeting scheduling.

We consider the meeting scheduling problem in the context of a distributed environment. This is different from how meetings are scheduled using typical commercial systems like Microsoft Exchange Server. In that environment, every calendar is on a central server and users explicitly share their calendar information with others. Our assumptions are more general. We do not assume that other users are using any particular piece of software or indeed any software at all. In this way we minimize software incompatibilities while providing for the more general case. Our approach also facilitates privacy since there is no central server with every users calendar. Additionally, we also aid privacy by assuming users do not transmit their entire calendar when scheduling meetings. In fact, we introduce a way to transmit less calendar information and still schedule high quality meetings. For the aforementioned reasons, we use the distributed approach to meeting scheduling.

It is obviously easier to schedule a meeting if we know the calendars of the meeting attendees exactly as we can find time slots where all of the attendees are free without even asking them. However, as stated, we are not operating in an environment where we have perfect access to that information. Therefore we will attempt to learn the calendars of other users. This allows us to behave as if we have good information about the calendars of others while maintaining a distributed setting and allowing agents to keep their information private when they choose.

Most attempts to automate meeting scheduling have involved the use of negotiation. Each user can be represented by an agent. An agent could be a completely autonomous piece of software, a program that works with a certain amount of user input, or a user not aided by automation at all. When a meeting needs to be scheduled among a set of users, their agents negotiate the meeting time on their behalf. Typically, the negotiation involves one agent, the meeting initiator, proposing times to the attendee agents and receiving responses from the attendees signifying some sort of acceptance or rejection. A simplified version of this general approach works as follows:

```
while no intersection in proposals:
    initiator proposes times to the attendees
    attendees respond about times to the initiator
```

Automating meeting negotiation effectively is very challenging. An automated negotiation agent needs to be able to effectively schedule meetings with many different types of agents. Each agent is available at different times and must be able to handle different negotiation strategies e.g., some agents are more willing to compromise their time preferences than others. Each agent can have different preferences about things like time of day or meeting density. The process of learning the preferences of an agent's user is difficult. Some users may not have agents at all. As such, verbose and frequent communications, which may be adequate when sent to an automated agent can be aggravating to humans that have to respond to them. Additionally, the problem we consider is incremental. That is, calendars change gradually as new meetings are introduced so we cannot just find a globally optimal schedule. This results in a tradeoff between local optimization and future planning.

In this report, we show how an agent can learn a model of another agent's calendar by introducing an algorithm LOC (*L*earning *O*thers' *C*alendars). We show how it can use the models it learns of other agents to schedule meetings more effectively. In particular, we show in simulation, how an agent learning online can exploit as it learns to increase its user's satisfaction with scheduling outcomes. Our work is implemented as part of the CMRadar project and integrates into the CMRadar code base.

The basic organization of this report is as follows. In Section 2, we formalize the problem definition and discuss representing calendars using probability distributions. In Section 3, we introduce an algorithm for learning the calendars of other agents and present a set of experiments demonstrating its efficacy. In Section 4, we present an algorithm which uses this method to increase the quality of scheduled meetings according to a user's utility metric and give the results of experiments using this method. Finally, in Section 5, we present our conclusions.

2 Representation

In this section we discuss representing the meeting scheduling process. First, in Subsection 2.1 we introduce the notation we use for the remainder of this thesis. Then we present a formalization of the meeting scheduling problem using this notation. In Subsection 2.2 we discuss the problem inherent in representing another user's calendar and introduce the idea of a probabilistic calendar as a solution.

2.1 Problem Definition

This subsection presents our notational conventions in a formal manner. We then present the problem of scheduling meetings using these notational conventions. This gives us a formalism to use for this document.

2.1.1 Notation

We now define our environment and notational conventions.

- There is a set of agents A. These are the people between whom meetings are scheduled. They can be human or completely or partially automated agents representing humans. This might be the people in an office or a set of professors and students.

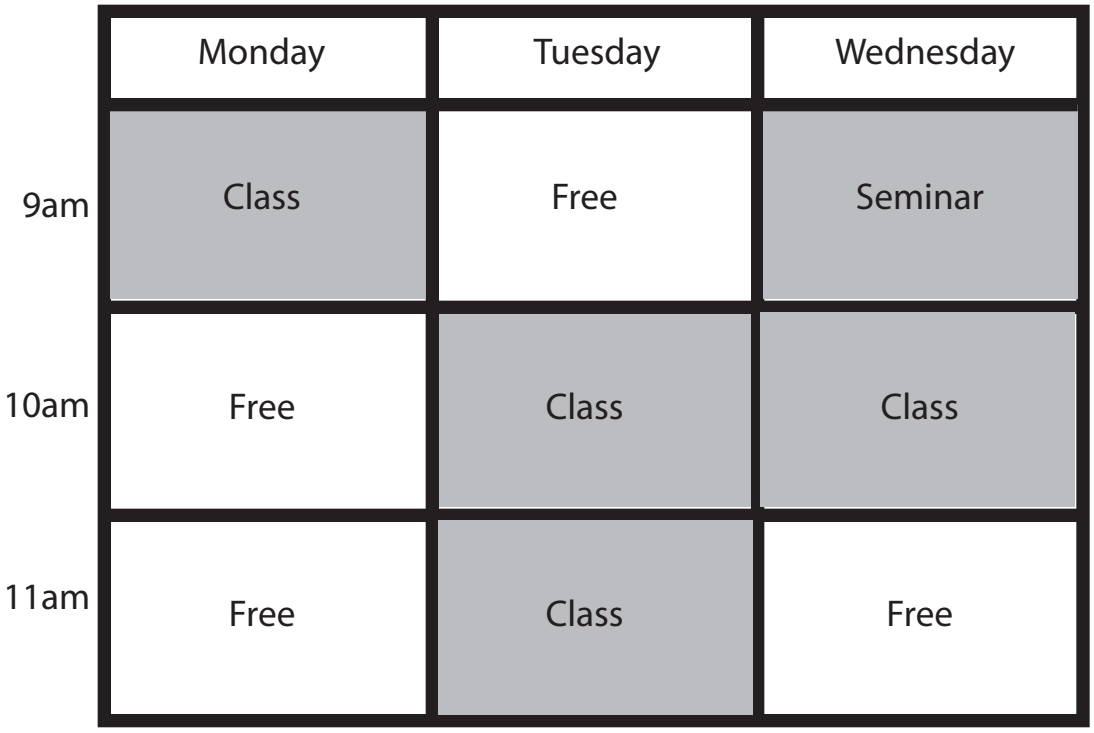

Figure 1: Example Calendar

- A meeting m is a triple: (a_0, A_0, t) of an initiator agent a_0, a set of agents $A_0 \subseteq A$ representing the attendees of the meeting, and a meeting type t. The initiator agent is responsible for scheduling the meeting. If a meeting has to be canceled or moved it is the initiator's job to reschedule it.

- A meeting type t is an element from a set of meeting types T. As we will discuss later, learning about the types of other users' meetings can aid the scheduling process. A sample set of meeting types, which we will use in our examples later is $\{class, conference, seminar\}$.

- A time slot d is a date indicating the start of the time slot. That is, it contains an hour, day, year or any other relevant information for fixing a moment in time. We assume all time slots are of fixed size and start at regular times. For example, we will later consider time slots as being an hour long and starting on the hour.

- A calendar c is a set of time slots. Each time slot can have a meeting attached to it. This indicates that the calendar contains a meeting scheduled at that time slot. Although we do not do it, it would be simple to extend this definition such that a meeting spanned multiple time slots. Additionally, making the time slot size sufficiently small, for example the Planck time, would allow for meetings of arbitrary start time and length. An example calendar with three time slots and three days appears in Figure 1.

- Each agent $a \in A$ has a corresponding calendar c_a. This calendar represents the meetings in the calendar of agent a.

- A meeting response r is an element from a set R of possible responses in a negotiation. An example of a response would be an affirmation signifying that a proposed time is acceptable. Another type of response might indicate that an agent would be willing to schedule a meeting at the proposed time, but would prefer not to.

2.1.2 Problem Formalization

Thus, we restate the scheduling problem as follows: To schedule a meeting m with attendees A_0, the initiator agent a_0 sends a set of possible slots for the meeting to each attendee in A_0. These slots signify the time slots at which a_0 is proposing to have the meeting. The other agents then send a response r for each time slot. If there is a time slot the agents agree on then a_0 sends a confirmation message to the agents. Otherwise it sends a new set of time slots to the other agents. This process continues until an acceptable time is found. This communication process continues until a slot is agreed upon. This process is an example of meeting negotiation and is the formalism for negotiation we use for the remainder of this document.

2.2 Calendar Representation

We want to learn the calendars of other agents and use this information to improve the scheduling process. With this information we can ask about times that are likely to be free or favorable and use this to cut down the messages sent between agents. We need some way to represent this information for our agent's use. In this subsection, we describe the problems with representing others' calendars as we would our own. We then present the idea of a probabilistic calendar as a solution to this problem.

2.2.1 Limitations

We can use the information in the calendars of others even if we do not accurately know what those calendars actually look like. For example, we might be able to learn that another agent tends to schedule meetings in the afternoon. Therefore, we know that they are more likely to be busy if we try to propose a meeting in the afternoon. This is the case even if they have not signified this for certain by responding negatively to a request at a specific afternoon time.

In the previous subsection we described each calendar as a set of time slots that may have meetings attached if the agent associated with the calendar has a meeting at that time. An agent can use this as a model for the calendar of its user, as the agent knows the calendar of its user as ground truth. However, the calendars of other agents change. Other agents can schedule meetings between when our agent last contacted them. Indeed, it is rarely the case that we even have a complete picture of another person's calendar captured at a single point in time and we make the assumption that calendars are not, in general, public. People tend not to want to share their entire calendar with other people. As such, they tend to give a limited subset of their times during a round of meeting negotiation. If we tried to use a traditional calendar to represent this we only be able to indicate information we have for certain, but we can make use of other information. Therefore, this is not a useful representation.

2.2.2 Probabilistic Calendars

Given the inconclusive nature of the information we can collect about the calendars of other agents, we propose representing the calendars of other users with probability distributions. Our agent a_0 represents the calendar of each other agent $a \in A$ as a probabilistic calendar $p_{a_0,a}$. A probabilistic

	Monday		Tuesday		Wednesday	
9am	Class	40%	Class	40%	Class	40%
	Activity	40%	Activity	40%	Activity	40%
	Free	20%	Free	20%	Free	20%
10am	Class	10%	Class	86%	Class	05%
	Activity	80%	Activity	8%	Activity	05%
	Free	10%	Free	6%	Free	90%
11am	Class	13%	Class	05%	Class	33%
	Activity	44%	Activity	05%	Activity	33%
	Free	43%	Free	90%	Free	34%

Figure 2: Example Probabilistic Calendar

calendar is a set of time slots each paired with a probability distribution over type of meeting at that time slot. For example, an element j of $p_{a,i}$ would represent that agent a believes that during time slot d_j there is a thirty percent chance that user i has a meeting of type t_0, a forty percent chance it is of type t_1, and thirty percent chance that time slot is empty for user i if the slot is paired with the described distribution. This representation provides more flexibility than a static representation while also being able to represent meetings we know about for certain. We initialize each distribution over responses to a uniform distribution. An example probabilistic calendar for three days with three time slots appears in Figure 2. Contrast it with Figure 1, a traditional calendar. Keep in mind that this probabilistic calendar is what one agent believes is the calendar of another user, but may not accurately reflect that calendar.

3 Learning

Since we are modeling the calendars of other agents with probability distributions, we must have a way to learn these probability distributions in order for this representation to be useful. We present

an algorithm *LOC* (*L*earning *O*thers' *C*alendars) to do this. At the very high level, this algorithm works by examining the messages sent during meeting negotiations. If an agent typically responds affirmatively when a meeting is requested at a certain slot then the agent is probably free at that time and analogously for negative responses. Once we have presented this algorithm, we then present a series of experiments. The first demonstrates the general efficacy of *LOC*. The second tests it under more difficult learning conditions.

3.1 Learning From Negotiations

To learn others' calendars we keep track of the responses to meeting requests. If an agent accepts when we propose a meeting at a certain time several weeks in a row, it is likely the agent is normally free at that time. This works analogously for negative responses, providing information on when other agents are not free. Simply keeping track of this data allows us to build up a reasonable idea of when another agent is free.

We extend this basic idea so we can also learn the types of these meetings. For each type of meeting, we assume that there is a fixed probability distribution that models the responses the agent makes when a time containing an existing meeting of that type is requested. For instance, a professor is unlikely to accept a new meeting at a time when he/she normally teaches a class, but may occasionally accept a new meeting if he/she had been planning to attend a seminar at that time. We codify this intuition into a probably distribution for that response type. In this paper, we assume that for each meeting type we know the probability of receiving each possible type of response. In practice, this information could be gathered by observing user behavior. Figures 1 and 2 show examples of these distributions with a set of meeting types $T = \{conference, class, seminar\}$ and set of response types $R = \{yes, no, hesitant - yes\}$

Formally: Let R be the set of possible meeting request responses and T be the set of all meeting types. For each $t \in T$ we assume we have a probability distribution over the responses $r \in R$. We let r_t denote the probability of an agent making response r when it receives a request for a meeting at a time when it has a meeting of type. Additionally, for notational convenience, we denote an empty time slot as containing a meeting of type e and we let e_t be the response probability distribution for empty time slots.

For example, let $R = \{yes, no\}$ and $T = \{t_0, t_1\}$ where t_0 is something of moderate importance, and t_1 is something of high importance. A likely distribution for this would have a high probability of yes for no meeting, a moderate probability of yes for meetings of type t_1, and a low probability for those of type t_2. We can compare these distributions to the observed distribution of responses to learn the types of meetings in other users' calendars.

3.2 Belief Updating

We now present our belief update method for an agent a after a response r to a request for a meeting at time slot d from agent u.

1. Take the element from $p_{a,u}$ for time d. As previously mentioned, if we have learned nothing about this time yet, we use a uniform distribution.

2. Treat this element as a vector of weights $\overrightarrow{w}_n$ where $w_{n,j}$ is the current belief of the agent that u has a meeting of type j at d. This includes an element for the empty meeting.

3. Define a vector $w_{n+1,j} = w_{n,j} \cdot r_j$. Divide each $w_{n+1,j}$ by $\Sigma_{t \in T} w_{n+1,t}$. This normalized $\overrightarrow{w_{n+1}}$ is then used as an updated version of the beliefs in place of w_n.

Basically, this updates the weight for a type according to the likelihood of the observed response given a meeting of that type and then normalizes the results.

We now give a general example of applying this update method. We return to our example where $R = \{yes, no\}$ and $T = \{t_0, t_1\}$. Assume we are an agent a_0 and have received a response of yes in response to a request for a meeting at time slot d with agent a. We look up the current probability distribution for d in $p_{a_0,a}$. After our updating, the probability of $none$ rises significantly since we got a positive response. The probability of t_0 stays about the same since meetings of that type are somewhat likely to be bumped or skipped. The probability of t_1 drops since it is unlikely someone would want to schedule another meeting over a meeting of type t_1.

3.3 Agent Behavior

```
initiate-meeting(m)
    possible-slots = pick-random-free-slots
    foreach attendee a of m
        send-proposal(a, m, possible-slots)

handle-response(m, responses)
    foreach possible-slot s in responses
        update-beliefs(s, response-for(s, responses))
    if contains-possible-meeting(responses)
        foreach attendee a of m
            send-confirmation(a, m)
    else
      initiate-meeting(m)
```

Figure 3: Agent Algorithm

A brief version of the *LOC* algorithm for our learning agent is given in Figure 3, a more complete version is found in Appendix A. The algorithm for our learning agent has two routines: `initiate-meeting` and `handle-response`. The `initiate-meeting` routine is given a meeting and tries to schedule it by coming up with a list of time slots. It then sends these slots to the other attendees of the meeting. The `handle-response` routine takes a meeting and a set of responses from the other agents about the proposed time slots. First it updates the beliefs about the type of meetings in others calendars based on the responses we get. Then it checks if there is a slot they all agree on. If there is, it confirms it, otherwise it tries again with a different set of time slots.

	yes	$hesitant - yes$	no
$class$	0.30	0.30	0.40
$conference$	0.10	0.30	0.60
$seminar$	0.05	0.10	0.85
$none$	0.80	0.10	0.10

Table 1: Learning Experiment Response Distributions

3.4 Experiments

Now that we have described our algorithm for learning, we must show that this algorithm actually works. We ran a series of experiments testing aspects of this algorithm. The first experiment tested that *LOC* actually learned another user's calendar and tested a parameter to the learning function. The second experiment tested the algorithm under more constrained circumstances.

3.4.1 Learning Experiment

We ran two parallel scenarios to test the *LOC* algorithm. These are grouped together as the Learning Experiment. These experiments were meant to simulate the typical behavior of a student and the meeting types and response distribution we use in the experiments reflect that assumption. Both shared the following setup.

1. The set of meeting types T was $\{class, seminar, conference\}$ along with the empty type $none$, signifying no meeting.

2. The set of response types $R = yes, no, hesitant-yes$. yes and no indicate a positive or negative response respectively. A $hesitant - yes$ indicates that the responder would be willing to schedule a meeting at that time but would prefer not to if possible. We modified the *LOC* algorithm described in the previous subsection slightly to allow for this type of response. If, from the list of responses to a request we got both a yes response and a $hesitant - yes$ response for two different slots, we would schedule the yes response. Otherwise, we would schedule the $hesitant-yes$ response. The response distribution table used in the experiment is shown in in Table 1.

3. There were three agents: the $learner$, the $responder$, the $schedulee$. The $learner$, was the agent running *LOC* and was the initiator for all the meetings. The $responder$ was an agent programmed to simply respond to requests according to the appropriate response distribution based on its calendar. The $schedulee$ was the party with whom all of the meetings in the $responder$'s calendar were scheduled. It was necessary to have this so that the the meetings in the $responder$'s calendar would not be with the $learner$ which would allow the $learner$ to start out knowing the types of the meetings in the calendar of the $responder$.

4. Calendars ranged over one work week, Monday - Friday, with hour long time slots starting at 8am and going hourly until 6pm. This gives fifty time slots in a week. The $learner$ started with an empty calendar. The $responder$ started with a randomly generated calendar. This was generated by assigning each type of meeting, including $none$, a random likelihood and

then, for each time slot picking a type from that distribution. The $schedulee$ had the same calendar as the $responder$ as adding additional meetings to the $schedulee$'s calendar would not have had an effect on the interaction we were looking at, that between the $learner$ and the $responder$.

5. To simulate the process of learning from week to week, we scheduled five meetings at a time and then restored the calendars to their original state. This state was meant to indicate the regular meetings in that user's calendar. Each reset denoted one week. We then ran this through forty weeks.

The `initiate-meeting` function picks some subset of the set of the set of free time slots to propose. The number of time slots it picks is a free parameter. The value of this parameter represents how big our messages are. Consider, that when scheduling a meeting with a human it may be worthwhile to propose fewer slots than when scheduling a meeting with an automatic agent. This can reduce the burden on the human. For a program, message overhead may be less significant. Thus, to represent the effects of this tradeoff between message size and amount learned, we ran two similar experiments based on the previously described setup. In the first, we set this parameter to five time slots per proposal, in the second we set this parameter to ten slots per proposal.

3.4.2 Learning Experiment Results

Figure 4 was generated by running the previously described experiments ten times each and averaging the results of the ten runs. Remember, we have two different versions, one which proposes five time slots at a time and one which proposes ten. The ten slot condition is the solid line, the five slot condition is the dotted line. Each line shows the correct number of meeting types learned by the $learner$ about the meetings in the $responder$'s calendar after each week against the number of weeks passed. We consider the first ten weeks the most important as that is a realistic amount of time over which meetings are scheduled with a static base calendar. The other thirty weeks are to show what happens in the optimal case where we have traffic over a longer period.

The first thing to note is that as time goes on, the number of correct meetings increases. The second thing to note is that the ten slots condition correctly learned the types of half of the calendar meetings after just three weeks. This shows not only that *LOC* does its job, but that it can do it quickly. The ten slot condition leveled off rapidly after it hit forty correct time slots, whereas the five slot condition had a gradual increase. It seems reasonable at that point to cut down on the messages sent in the ten slot condition, since asking about more slots does not provide much more information. This is because we have already learned pretty much everything. Later on, we will introduce a version of *LOC* which has a similar effect to cutting the number of messages.

3.4.3 Constrained Response Experiment

We wanted to test the performance of *LOC* when the response distributions for each meeting type were less distinct, that is, when it was more difficult to determine which meeting had which type. To do this, we adjusted the response distributions from those in Table 1 to a more homogenous, constrained set of distributions. This new distribution set is found in Table 2. When running this

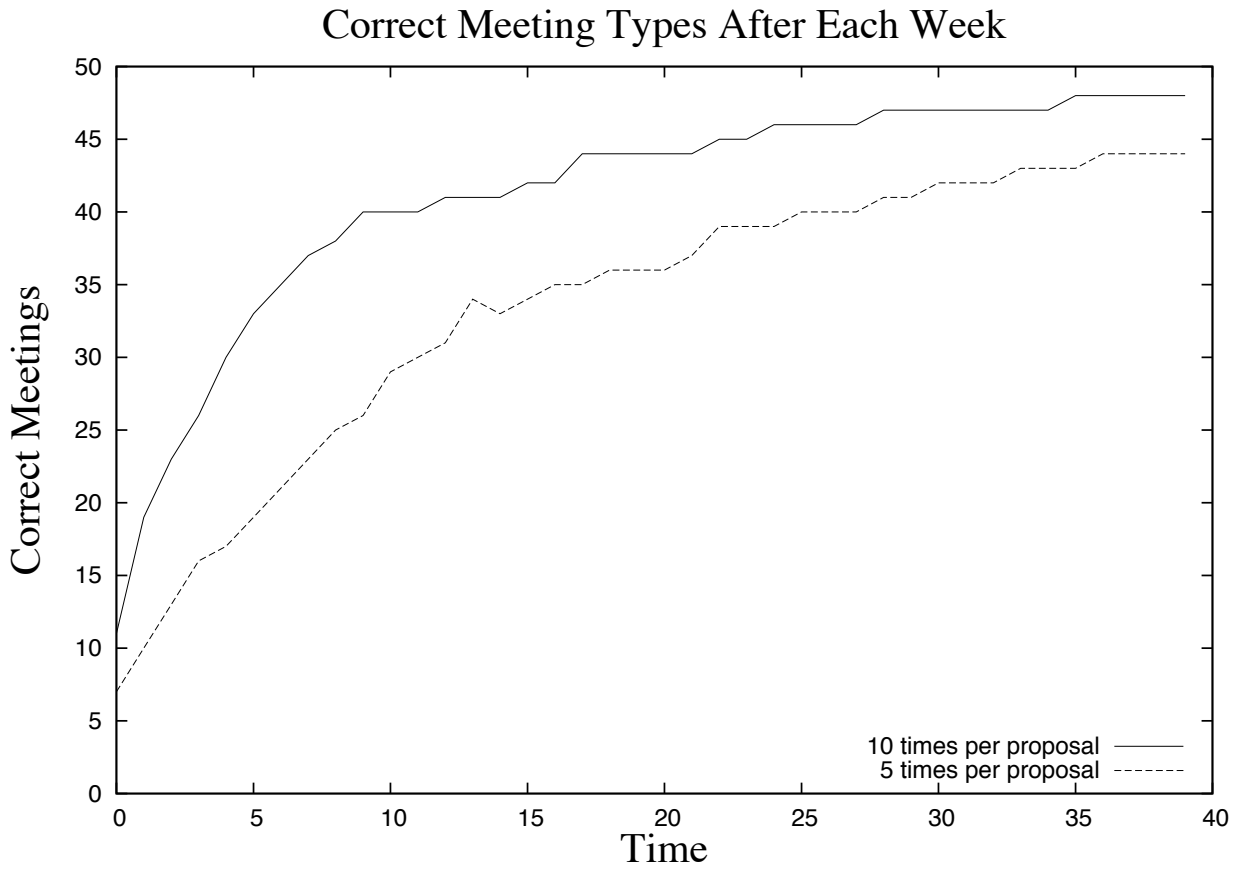

Figure 4: Learning Experiment Results

experiment, each meeting proposal contained ten time slots. All other aspects of this experiment are the same as the previous experiment.

3.4.4 Constrained Response Experiment Results

The results of this experiment are in Figure 5. They show that *LOC* works under constrained circumstances. However, not surprisingly, it does not perform as well as in the previous experiment, where the distributions were more distinct. We cite this result as a demonstration that even if we do not predict very well how people will respond, as long as there is some difference, *LOC* will converge on the correct calendar.

4 Improving Scheduling

We have demonstrated the *LOC* algorithm for learning calendars. We introduced this algorithm in order to use the information it learns to improve the scheduling process. In this chapter we present a way of doing this. First, we formalize our conception of quality meetings with a numeric utility function. Then we describe a modification to the *LOC* algorithm which uses the learned data to

	yes	$hesitant - yes$	no
$class$	0.20	0.45	0.35
$conference$	0.30	0.30	0.40
$seminar$	0.10	0.25	0.65
$none$	0.60	0.20	0.20

Table 2: Constrained Response Experiment Distributions

decrease the number of messages sent in negotiation while scheduling meetings with high utility. Lastly, we present an experiment succesfully demonstrating this.

4.1 Utility

How do we define the concept of meeting quality? We do this by assuming that users have a utility function that provides a mapping from time slots to utilities. For convenience, we assume that utilities are in the interval $[0, 1]$ where 0 is minimal utility and 1 is maximal utility. For example, by having a function that gives a low utility for early morning times we can represent a person who does not like to meet in the morning. The version of *LOC* previously described generated times randomly, this results in a mostly uniform distribution of scheduled meetings. In the presence of a utility function that does not value all times equally, this doesn't really make sense. As such, we augment each agent a with a known utility function u_a which takes a time slot and returns the utility of scheduling a meeting at that time.

4.2 Modified *LOC* Algorithm

We have a modified version of *LOC* which uses this utility function and what we have learned about the other users' calendars to schedule meetings with high utility. Additionally, it uses fewer messages than a method that picks times without using calendar learning. This is done using a simple modification. In the definition of `initiate-meeting` instead of picking times randomly, we order the times. This ordering is accomplished by assigning each slot d a priority based on the utility of that time and the learned probability pr that a meeting request at that time will receive an affirmative response. The exact priority is given by the following formula:

$$p = \alpha u_a(d) * \beta pr$$

In this case α and β are parameters describing the relative weight of each portion of the utility.

Using this ordering is not always ideal. First, at the beginning, when we do not know anything about other agents' calendars the probability of acceptance component does not provide any information. Once we gain this information, it can possibly change as people adjust their calendars from week to week as classes end or obligations change. Additionally, this could open up times with high utility or fill up times at which we have previously had success scheduling meetings.

As such, rather than following this ordering every time we try to schedule a meeting, we probabilistically choose either to use this ordering, that is, to exploit our information, or to explore randomly in the hopes of obtaining useful information for later. Furthermore, as time goes on

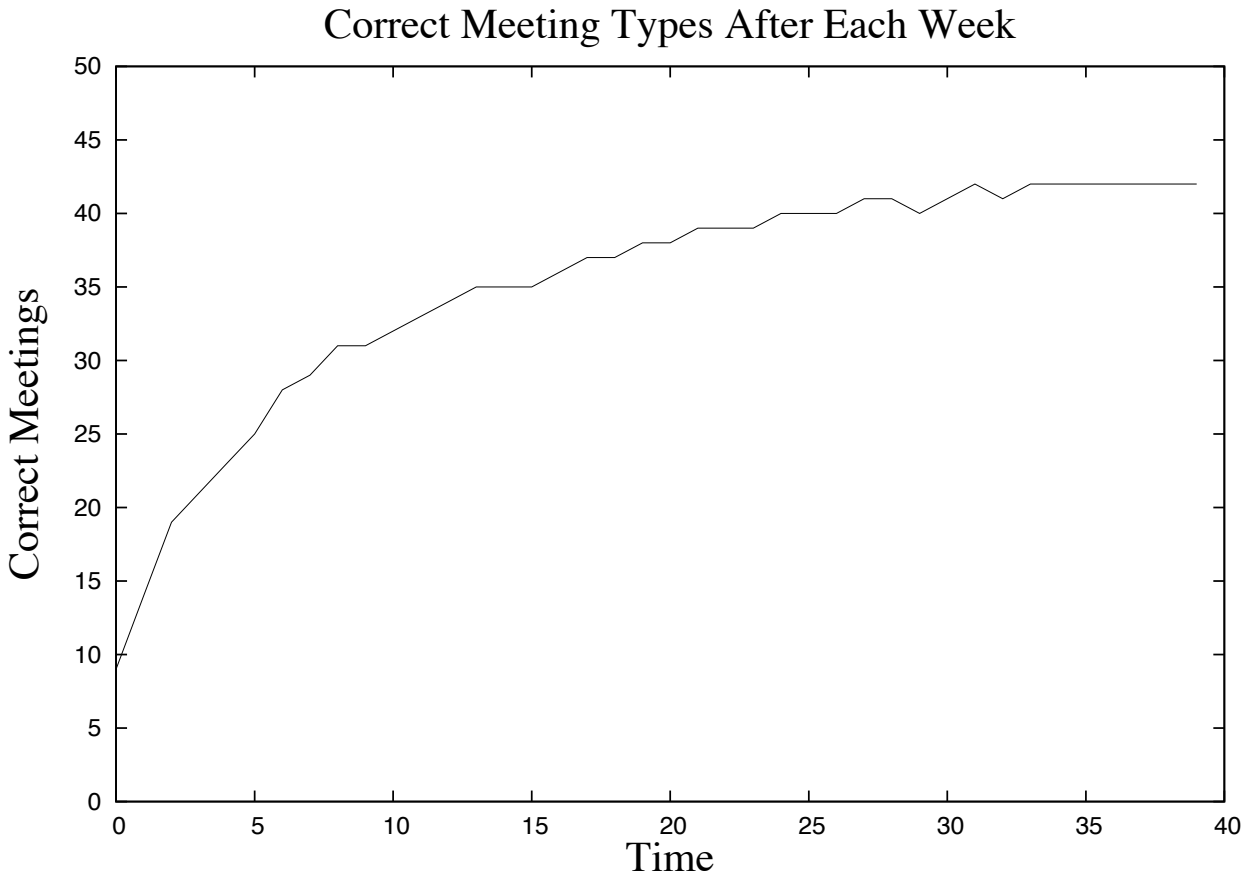

Figure 5: Constrained Response Experiment Results

and our picture of other agents calendars get better we explore less often, until we hit a minimum level at which point we explore with the constant probability of .1. The threshold is there as otherwise, any decreasing function would eventually stop exploring entirely even in the face of calendar changes. This introduces a time dependency t into our decision making process. We specify this decision making process in Figure 6. The process starts at $t = 0$ with high probability of exploring and gradually explores less by using a falloff proportional to $1/(1 + t)$ to give the probability that we should explore. That this is a proportional falloff and not an exact function is implemented by the weight `w` in the decision process specification. This function falls as t increases, meaning we exploit increasingly often until we hit the exploitation threshold.

4.3 Slot Ordering Experiment

In this subsection, we present an experiment testing this modified version of the *LOC* algorithm. We first explain how we model the general utility function in our system. We then describe the experimental setup which is similar to previous experiments. Finally, we present the results of this experiment, showing that we can successfully use calendar information to cut down on communication while keeping utility high.

```
r = rand(0, 1)
if r < max(.1, 1 / (1 + w * t))
    explore
else
    exploit
```

Figure 6: Exploitation vs Exploration Control Process

4.3.1 Utility Modeling

We previously described utility as a mapping from time slot to utility value. We model this by dividing each day into three sections - morning, afternoon, and evening. Morning is from 8am to 12pm. Afternoon is from 12pm to 3pm. Evening is from 3pm to 6pm. We assume each user then provides a utility weight in the interval $[0, 1]$ for each of these time ranges. However, as other work on learning user preferences develops, such systems could be used to learn these weights rather than assuming they are specified by the user. This was done to simplify our model for the experiment, there is nothing about our algorithm that requires we use this simplification.

4.3.2 Experimental Setup

We ran an experiment to show that we can use this algorithm to get a utility comparable to a system which orders slots entirely by user preference while also sending fewer messages than such a system. To demonstrate this we ran a simulation in two slightly different conditions. Both used the same setup as the previous experiments except for the following modification and additions:

1. For the first condition we use the slot ordering method described previously. We call this the learning condition. The second condition uses this algorithm, except that it has $\beta = 0$, causing it to ignore the calendar information we have learned. We call this the no learning condition. The learning condition uses $\beta = 1.0$. All other aspects are the same between the two conditions.

2. We set $\alpha = 1.0$. This causes each part of the ordering to function to be weighed equally in the case of the learning condition.

3. We set `w`, the weight of the falloff for exploring versus exploiting to 0.3. This was determined empirically as a number which leveled out the falloff.

4. We ask about three time slots at a time. The primary reason for this is that we are no longer trying to learn about the calendar for the sake of learning. Rather, we are learning about the calendar in order to leverage this information to improve the quality of scheduling. As such, sending enough messages so that we can learn everything about the calendar with high probability is not that useful. Instead, care more about the slots with high utility and only if those are not available do we move onto less desirable slots. Thus, we show that using our algorithm we can schedule well with less information.

5. We used the meeting response distribution used in experiment 1 as given in Figure 1.

6. Unlike in previous experiments, some meetings shift around the calendar of the $responder$ between each week. Specifically, we chose three slots per week. Three was chosen as a number where the calendar would be mostly stable, but still show recognizable change. This shifting has two purposes. First, it makes the experiment more realistic - it is likely in the real world that some meetings shift from week to week. Second, a changing calendar makes the exploration/exploitation tradeoff more paramount. In the case where meeting times do not change, once the learner component finds a set of free slots with high utility it can just ask about those times each week. Thus, after a few weeks, there is no utility to be gained by exploring. By moving some meetings each week, we raise the utility gained by exploring, while also using a more realistic setting.

4.3.3 Results

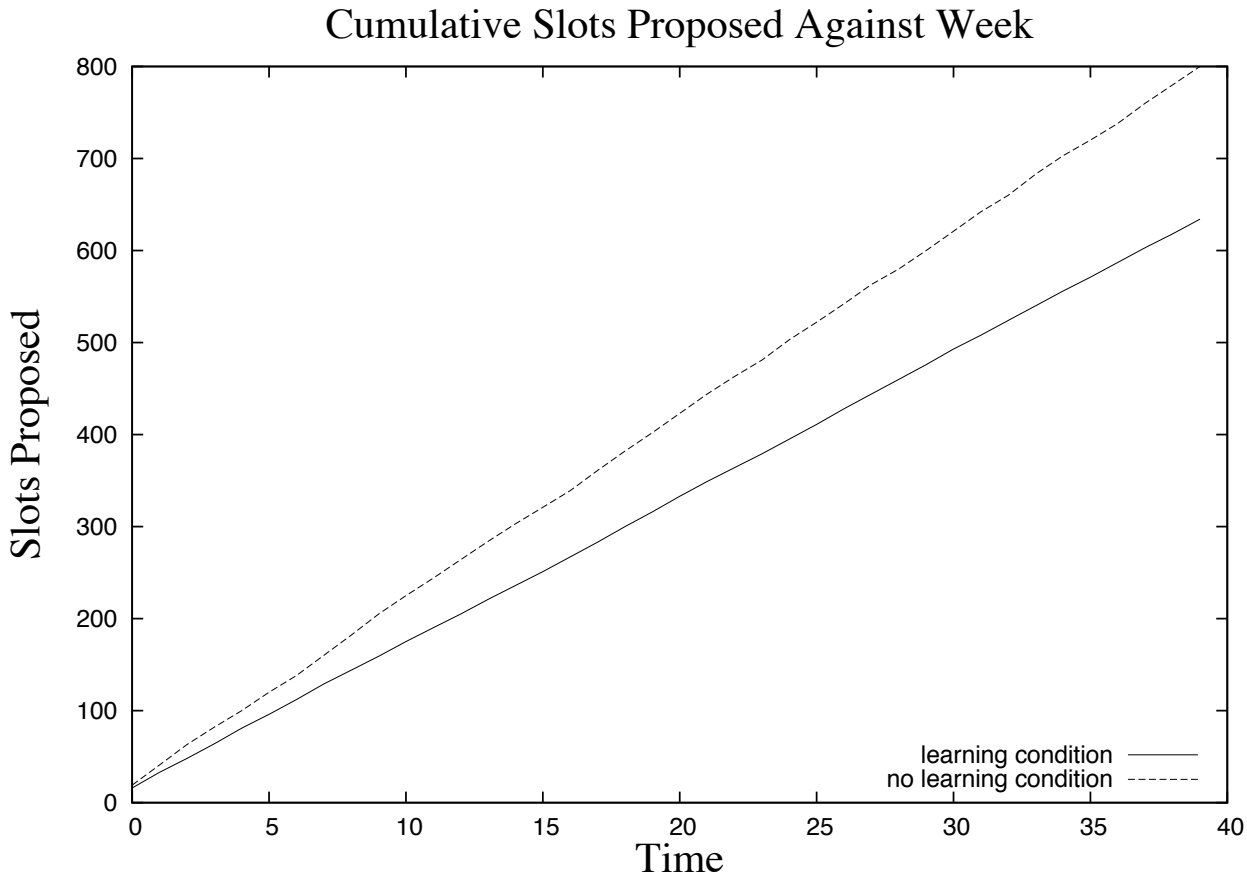

Figure 7: Order Weighting Experiment Slots Proposed Results

A plot of the number of time slots proposed over time in both the learning and no learning conditions is given in Figure 7. The learning condition is displayed with a solid line, the no learning condition with a dotted line. The graph shows that the number of times proposed was consistently

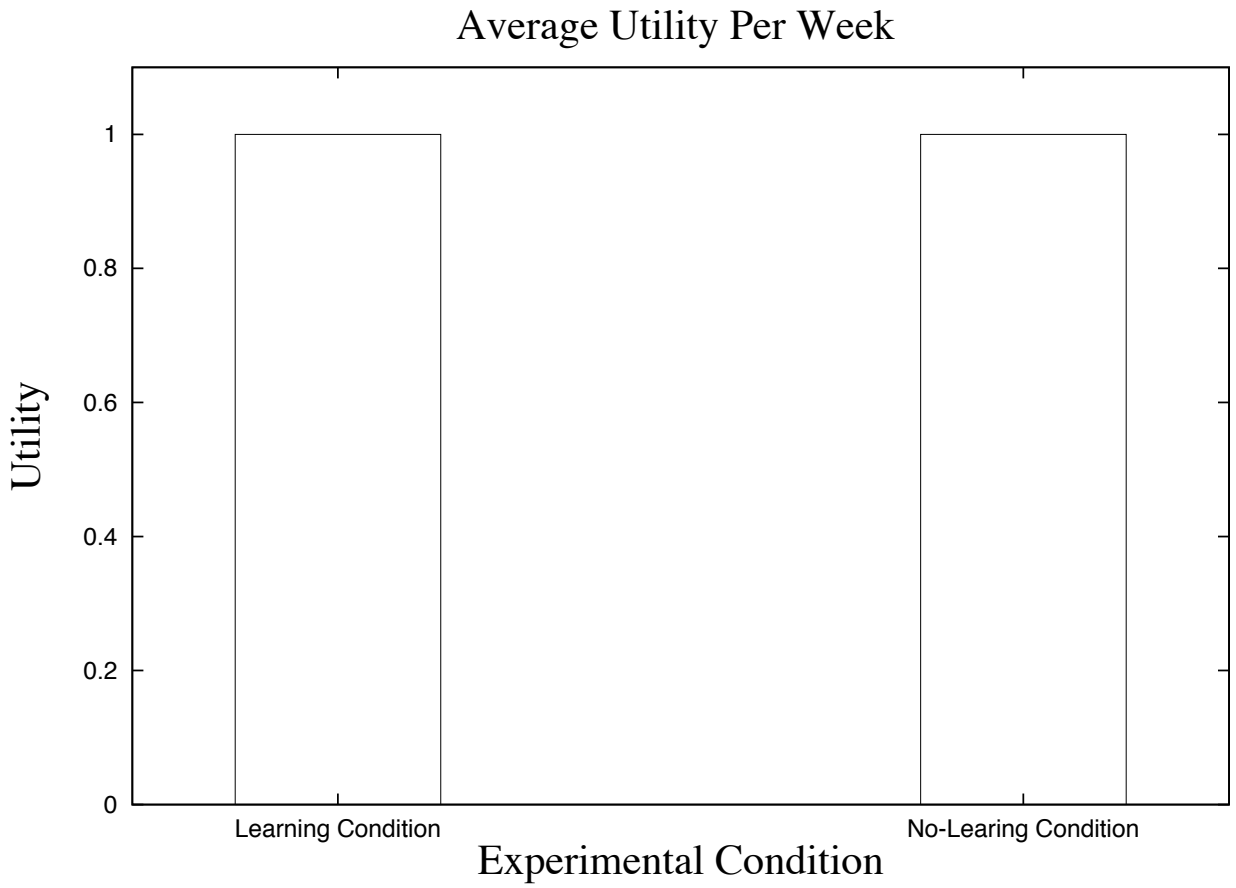

Figure 8: Order Weighting Experiment Utility Results

lower in the learning condition and the disparity between the two conditions became more extreme as time went on. Clearly, using the learned information made a big difference. Figure 8 shows the mean utility of the experimental condition and the control condition. The maximum value was 1.0, which both achieved. Simply sorting slots by utility in this situation where one-third of the slots have the maximum utility was enough to maximize utility as it is unlikely that all of these slots are full. The two have comparable mean utility, yet our algorithm, which adjusts its ordering based on the calendar learning, sends fewer messages. More specifically, by the end of the experiment it had proposed one hundred sixty-six fewer slots, which averages out to over four per week. Within the ten week period, as previously mentioned, we consider especially relevant, this works out to forty fewer slots proposed.

5 Conclusion

We set out to improve the quality of automatic meeting scheduling. This process is tedious. Furthermore, it is complicated due to the inaccuracies in information we receive and the sparsity of information available in general. To deal with these inaccuracies, we decided to represent the calendars of other users using a probability distribution for every time slot. This let us handle the fact

that others' calendars can change without our agent's knowledge. By making a simple assumption about response patterns based on meeting type, we were able to learn a probability distribution for each time slot in others' calendars. This algorithm used the responses to meeting proposals. For each time slot in a response we updated the beliefs about that time slot in the responder's calendar based on the response using a Bayesian update method. Not only did this system succeed at learning the meetings of others, but it was able to learn them within just a few weeks of negotiations.

Once we had learned the meetings of others calendars. We then set out to use the learned calendar information to decrease the number of messages sent. This allows us to give out less information while creating less communication overhead in negotiation. This was used to improve the meeting negotiation process by decreasing the number of messages sent while scheduling meetings in slots with high utility. We compared this to a version which did not use calendar learning when coming up with time for proposals. Our algorithm outperformed it significantly. Thus, we have shown that calendar learning can aid meeting negotiation.

Our system integrates into the work of the CMRadar project[4]. It is simple and straightforward to implement. Our part of the CMRadar code base is approximately two thousand lines of Java code along with assorted scripts and data files used to run the simulations.

References

[1] Pauline Berry, Melinda Gervasio, Tomas Uribe, Karen Myers, and Ken Nitz. A personalized calendar assistant. In *In the AAAI Spring Symposium Series, March*, 2004.

[2] Leonardo Garrido and Katia Sycara. Multi-agent meeting scheduling: Preliminary experimental results. In *Proceedings of the First International Conference on Multi-Agent Systems*, 1995.

[3] Tom M. Mitchell, Rich Caruana, Dayne Freitag, John McDermott, and David Zabowski. Experience with a learning personal assistant. *Communications of the ACM*, 37(7):80–91, 1994.

[4] Pragnesh Jay Modi, Manuela Veloso, Stephen F. Smith, and Jean Oh. Cmradar: A personal assistant agent for calendar management. In *6th International Workshop on Agent-Oriented Information Systems*, 2004.

[5] Sandip Sen and Edmund Durfee. A formal study of distributed meeting scheduling. *Group Decision and Negotiation*, 7:265–289, 1998.

```
initiate-meeting(m)
    possible-slots = shuffle(free-times)
    negotiation-info[m] = possible-slots
    propose-times(m)

propose-times(m)
    possible-slots = negotiation-info[m]
    current-slots = take(TIMES-PER-ROUND,
                         possible-slots)
    negotiation-info[m] = drop(TIMES-PER-ROUND,
                               possible-slots)
    foreach attendee a of m
        send-proposal(a, m, possible-slots)

handle-response(m, responses)
    foreach possible-slot s in responses
        update-beliefs(s, response-for(s, responses))
    slot = best-time(responses)
    if slot = none
        propose-times(m)
    else
        foreach attendee a of m
            send-confirmation(a, m, slot)
```

Figure 9: Full Version of *LOC* Algorithm

A *LOC* Algorithm

The following definitions are used in the definition of the learner algorithm in Figure 9:

- `free-times` is a list of the free time slots in the agent's calendar.
- `shuffle` a function which takes a list and randomly orders it.
- `take` is a function which takes an integer n and a list and returns the first n elements of the list.
- `drop` is a function which takes an integer n and a list and returns all elements of the list after the first n.
- `TIMES-PER-ROUND` which is a constant parameter indicating the number of time slots the agents asks about at one time.
- `send-proposal` is a function which takes an agent, a meeting, and a set of time slots and sends a message to the agent proposing those times for the meeting.

- `update-beliefs` is a function which given a slot s and response r updates the beliefs about slot s by applying the belief updating algorithm described in section 3.2.
- `response-for` is a function which takes a slot and a set of responses and gets the response for that slot.
- `best-time` is a function which takes a list of pairs of time slots and responses and finds a time slot with an affirmative response. If there is none it returns `none`.
- `send-confirmation` is a function which takes an agent, a meeting, and a time slot and confirms that the meeting should be scheduled at that time slot.