Information Elicitation in Scheduling Problems

Ulaş Bardak

August 2007

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
www.lti.cs.cmu.edu

Thesis Committee

Jaime Carbonell, Chair
Eugene Fink
Stephen F. Smith
Sven Koenig, University of Southern California

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in Language and Information Technologies

Copyright © 2007 Ulaş Bardak

ABSTRACT

When we work on a practical scheduling task, we usually do not have complete knowledge of the related resources and constraints. For example, when scheduling a conference, we may not know the exact sizes of available rooms or equipment needs of some speakers. The task of constructing a schedule based on incomplete data gives rise to several related problems, including the representation of uncertainty, efficient search for schedules, and elicitation of additional data that help to reduce uncertainty.

In this thesis we introduce a new information elicitation approach aiming to resolve uncertainty in order to increase quality of optimization while keeping the number of questions the user has to answer to a minimum. The approach differs from other approaches in terms of working with a continuous domain with a large number of uncertain variables, not having a need for bootstrapping, tight integration with the optimization process and integration of multiple approaches to elicitation.

The approach estimates the potential impact of asking a question on the schedule quality, based on available information such as stated user preferences or information about the uncertainty itself such as a probability distribution of typical values. The elicitation approach unifies three different elicitation algorithms which we also developed as a part of this work. The unified elicitor is a part of a scheduling system that supports the use of incomplete data. This work has been part of the RADAR project (www.radar.cmu.edu) at Carnegie Mellon University, which is aimed at building an intelligent system for assisting an office manager.

The purpose of this thesis is to introduce a new elicitation algorithm and to confirm the following hypothesis:

Using the presented elicitation procedure we can pick questions that improve the quality of produced schedules significantly better than picking questions randomly or by using simple heuristics.

We present evaluation results in the conference scheduling domain using different problem sizes, even though theoretically our approach is generalizable to optimization under uncertainty in any other domain. We show that we perform better than random picking of questions and simple heuristics and the difference becomes more prominent as the problem size increases.

ACKNOWLEDGEMENTS

This work would not have been possible without the support from my thesis committee members Jaime Carbonell, Eugene Fink, Stephen Smith and Sven Koenig who lent me their guidance and support and I owe them my deepest gratitude. In particular I would like to thank Eugene Fink with whom I have spent countless hours discussing various aspects of the algorithms used in this work and who has always been there to give me advice and answer my questions, not just on algorithm design but also on writing.

I would like to also thank current and past members of the RADAR Space/Time Group who worked with me in making my elicitation system a part of the Space/Time module and creating the whole Space/Time module: Konstantin Salomatin who worked on the backend, Matt Jennings and Greg Jorstad who built the optimization mechanism, Andrew Faulring and Brandon Rothrock who worked on the GUI, and Steve Gardiner who worked on building input files, and Blaze Iliev who proved himself an expert at evaluating and testing software. Throughout the implementation and testing of the elicitation many undergraduate students working part time for RADAR Space/Time group helped making the process go smoother. In particular, I would like to thank Franklin Ho, Chris Martens, Vijay Prakash, Nisheeth Sharma, Sonny Uppal, and Thuc Vu who have worked closely with me.

I would like to thank my officemates and friends Betty Cheng, Vasco Pedro, and Yanjun Qi for not kicking me out with all the noisy meetings I had with other members of RADAR and answering my random questions. I also want to thank Jahanzeb Sherwani, who with Betty and Vasco, worked with me on our start-up project, Mindkin for all his support.

Last but not least, I would like to thank my family who were there for me to calm me down when I felt like the work was starting to overwhelm me. My father, Ali, who visited me every year; my mother, Nazmiye, whose wisdom has been crucial for me throughout my life; my sister, Nazal, who gave me healthy eating tips; and my grandmother, Emetullah, whose love always made me have a positive outlook on the world, kept me focused and sane. I also want to thank Pınar for all her great support and understanding.

Table Of Contents

Abstract ..... *ii*

Acknowledgements ..... *iii*

1. Introduction *1*

1.1. Motivation **1**

1.2. Overview of the Results **1**

1.3. Overview of the Approach **2**

1.3.1. DOMAIN 2

1.3.2. ARCHITECTURE 3

1.3.3. KEY CONTRIBUTIONS 4

1.4. Related Work **4**

1.5. Thesis Outline **7**

2. Representation *8*

2.1. Scheduling Problem **8**

2.2. Resources and Constraints **9**

2.2.1. ROOMS 9

2.2.2. EVENTS 10

2.2.3. SCHEDULE 12

2.3. Schedule Quality **13**

2.4. Uncertain Resources **15**

2.5. Uncertain Preferences **16**

2.6. Utility Function **16**

2.7. Inference Rules **21**

2.7.1. RULE SYNTAX 22

2.7.2. PRIORITIES OF PROPERTY VALUES 23

2.7.3. RULE APPLICABILITY 24

2.7.4. APPLICATION PRIORITY 24

2.7.5. APPLICATION LOOP 25

3. Search For Schedules *27*

3.1. Search Algorithm **27**

3.2. Extensions **37**

3.3. Experiments **38**

4. Elicitation of Additional Data *40*

4.1. Elicitation Problem **40**

4.2. Estimate of Question Utilities **40**

4.3. Search for Optimal Questions **44**

4.4. Integrated Elicitor 48

4.5. Evaluation 51

4.5.1. METHODOLOGY 51
4.5.2. LARGEST WORLD STATE 53
4.5.3. LARGE WORLD STATE 59
4.5.4. MEDIUM WORLD STATE 64
4.5.5. SMALL WORLD STATE 69

5. Vendor Elicitation 74

5.1. Vendor Order Problem 74

5.2. Representation of Partial Knowledge 74

5.3. Search for Optimal Orders 76

5.4. Elicitation of Vendor Data 77

5.5. Experiments 79

6. Conclusions 80

6.1. Limitations 80

6.2. Future Work 81

7. Bibliography 82

Table of Figures

FIGURE 1: COMPARISON OF OUR ELICITATION APPROACH (SOLID) WITH RANDOM PICKING OF QUESTIONS (DASHED) FOR CONFERENCE SCHEDULING PROBLEM WITH 3300 POTENTIAL QUESTIONS. 2
FIGURE 2: ARCHITECTURE OF THE SCHEDULING SYSTEM. 3
FIGURE 3: EXAMPLE SCHEDULE. 9
FIGURE 4: EXAMPLE OF A ROOM HIERARCHY. 10
FIGURE 5: EXAMPLE OF A PARTIAL SCHEDULE. 12
FIGURE 6: EXAMPLE OF A PREFERENCE FUNCTION, WHICH SHOWS THE DEPENDENCY OF THE ASSIGNMENT QUALITY ON THE ROOM SIZE FOR THE DEMO. 13
FIGURE 7: EXAMPLE SCHEDULE. 14
FIGURE 8: PROBABILITY DENSITY FUNCTION FOR UNCERTAIN ROOM SIZE, WHICH IS BETWEEN 500 AND 750 WITH 0.75 PROBABILITY, AND BETWEEN 1000 AND 1250 WITH 0.25 PROBABILITY. 15
FIGURE 9: ENCODING OF UNCERTAIN VALUES AND FUNCTIONS. WE USE UNCERTAIN VALUES TO REPRESENT ROOM PROPERTIES, EVENT IMPORTANCES, AND PREFERENCE WEIGHTS, AND UNCERTAIN FUNCTIONS TO REPRESENT PREFERENCES. 17
FIGURE 10: EXAMPLES OF UNCERTAIN PREFERENCE FUNCTIONS; THE FIRST FUNCTION INCLUDES UNCERTAIN QUALITY VALUES, WHEREAS THE SECOND IS ENCODED BY TWO DIFFERENT FUNCTIONS, WITH PROBABILITIES OF 0.75 AND 0.25. 18
FIGURE 11: COMPUTING THE MATHEMATICAL EXPECTATION OF A FULLY CERTAIN PIECEWISE-LINEAR FUNCTION APPLIED TO AN UNCERTAIN PROPERTY VALUE. 19
FIGURE 12: COMPUTING THE MATHEMATICAL EXPECTATION OF AN UNCERTAIN VALUE, REPRESENTED BY A COLLECTION OF UNIFORM DISTRIBUTIONS AND THEIR PROBABILITIES. 20
FIGURE 13. COMPUTING THE MATHEMATICAL EXPECTATION OF AN UNCERTAIN PREFERENCE FUNCTION APPLIED TO AN UNCERTAIN PROPERTY VALUE. 20
FIGURE 14: RESULT OF APPLYING THE EXAMPLE RULE WITH AN EXPECTED ATTENDANCE OF 10 FOR THE EVENT. 21
FIGURE 15. EXAMPLE INFERENCE RULES FOR ROOMS (A), AND EVENTS (B). 23
FIGURE 16: CALCULATION OF THE APPLICATION PRIORITY OF AN INFERENCE RULE TO A ROOM. 25
FIGURE 17: APPLICATION LOOP FOR INFERENCE RULES. 26
FIGURE 18: SEARCHING FOR A HIGH-QUALITY SCHEDULE. 27
FIGURE 19: MAIN PROCEDURES OF THE ALGORITHM GIVEN IN FIGURES 19–25. 28
FIGURE 20: MAIN VARIABLES IN THE PROCEDURES GIVEN IN FIGURES 4–10 29
FIGURE 21: INITIALIZATION PROCEDURES OF THE SCHEDULING ALGORITHM. 30
FIGURE 22: COMPUTING THE REWARD-SCORE DIFFERENCE BETWEEN A NEW ROOM AND THE OLD ROOM OF A GIVEN EVENT. 31
FIGURE 23: COMPUTING THE REWARD-SCORE DIFFERENCES RELATED TO THE START TIME, DURATION, AND END TIME OF A GIVEN EVENT. 32
FIGURE 24: EVALUATION OF CANDIDATE TIME SLOTS FOR A GIVEN EVENT, WHERE EACH SLOT IS DEFINED BY ITS START TIME AND DURATION. 33
FIGURE 25: CHECKING THE AVAILABILITY OF A ROOM, AND IDENTIFYING THE EARLIEST AVAILABLE TIME SLOT IN A ROOM AFTER A GIVEN TIME. 34
FIGURE 26: CHANGING AN EVENT'S ASSIGNMENT, WHICH INVOLVES REMOVAL OF THE CONFLICTING EVENTS AND RE-COMPUTATION OF THE RELATED REWARDS. 35
FIGURE 27: TOP-LEVEL SEARCH PROCEDURE, WHICH RESCHEDULES ONE EVENT AT A TIME, UNTIL REACHING A LOCAL MAXIMUM OR HITTING THE TIME LIMIT. 36
FIGURE 28: COMPARISON OF MANUAL AND AUTOMATIC SCHEDULING. WE GIVE THE RESULTS FOR SMALL PROBLEMS (5 ROOMS AND 32 EVENTS), MEDIUM PROBLEMS (9 ROOMS AND 62 EVENTS), AND LARGE PROBLEMS (13 ROOMS AND 84 EVENTS). WE SHOW THE QUALITY OF MANUAL SCHEDULES BY GREY BARS, AND THE RESULTS OF AUTOMATIC SCHEDULING BY WHITE BARS. 38
FIGURE 29: DEPENDENCY OF THE SCHEDULE QUALITY ON THE RUNNING TIME. WE SHOW THE RESULTS OF SCHEDULING WITH FULLY CERTAIN KNOWLEDGE (SOLID LINE) AND UNCERTAIN KNOWLEDGE (DASHED LINE); BOTH PROBLEMS INCLUDE 13 ROOMS AND 84 EVENTS. 39
FIGURE 30: COMPUTING THE IMPACT OF AN UNCERTAIN ROOM PROPERTY ON THE STANDARD DEVIATION OF THE OVERALL SCHEDULE QUALITY. NOTE THAT THIS COMPUTATION USES THE REWARD PROCEDURE, GIVEN IN FIGURE 33. 42
FIGURE 31: IMPACT OF AN UNCERTAIN IMPORTANCE ON THE SCHEDULE QUALITY. 43

FIGURE 32: COMPUTING THE IMPACT OF AN UNCERTAIN RANGE OF ACCEPTABLE VALUES ON THE STANDARD DEVIATION OF THE SCHEDULE QUALITY. 43
FIGURE 33: COMPUTING THE REWARD FOR SATISFYING A GIVEN PREFERENCE. 44
FIGURE 34: EXAMPLE RELATIVE RANKING OF TWO UNCERTAIN VARIABLES BY SEARCH ELICITOR. CIRCLES DENOTE STEP NUMBER AND WE REFINE THE RANGE OF POSSIBLE UTILITY VALUES AT EACH STEP USING THE OPTIMIZER. AFTER STEP 5, THE MINIMUM POSSIBLE UTILITY OF ASKING ABOUT THE SIZE OF A ROOM BECOMES HIGHER THAN THE MAXIMUM POSSIBLE UTILITY FOR ASKING ABOUT THE NUMBER OF PROJECTORS. THEREFORE, WE RANK THE QUESTION ABOUT UNCERTAIN ROOM SIZE HIGHER THAN THE QUESTION ABOUT UNCERTAIN NUMBER OF PROJECTORS. 46
FIGURE 35: SEARCH ELICITOR ALGORITHM FOR HANDLING ALL UNCERTAIN VARIABLES EXCEPT PREFERENCE FUNCTIONS. 47
FIGURE 36: HELPER FUNCTION FOR SEARCH ELICITOR ALGORITHM. 47
FIGURE 37: SEARCH ELICITOR ALGORITHM FOR HANDLING UNCERTAIN PREFERENCE FUNCTIONS. 48
FIGURE 38: RULE-BASED ELICITOR ALGORITHM FOR CHOOSING SUPPLEMENTAL QUESTIONS ON ROOM ATTRIBUTES. 48
FIGURE 39: STEPS OF THE INTEGRATED ELICITOR. 50
FIGURE 40: EVALUATION PROCEDURE. 52
FIGURE 41: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING HEURISTIC AND RULE-BASED ELICITORS ON WORLD STATE WITH 3300 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. 54
FIGURE 42: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING SEARCH AND RULE-BASED ELICITORS ON WORLD STATE WITH 3300 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE RERUN THE ELICITATION AFTER WE ANSWER EACH BATCH OF 20 QUESTIONS. 55
FIGURE 43: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING SEARCH, HEURISTIC, AND RULE-BASED ELICITORS TOGETHER ON WORLD STATE WITH 3300 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE RERUN THE ELICITATION AFTER WE ANSWER EACH BATCH OF 20 QUESTIONS. 56
FIGURE 44: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING RULE-BASED ELICITOR AND RANDOM QUESTION SELECTION ON WORLD STATE WITH 3300 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE ALSO COMPARE ALL SYSTEMS WITH ONE ANOTHER. 57
FIGURE 45: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING HEURISTIC AND RULE-BASED ELICITORS ON WORLD STATE WITH 1000 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. 59
FIGURE 46: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING SEARCH AND RULE-BASED ELICITORS ON WORLD STATE WITH 1000 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE RERUN THE ELICITATION AFTER WE ANSWER EACH BATCH OF 20 QUESTIONS. 60
FIGURE 47: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING SEARCH, HEURISTIC, AND RULE-BASED ELICITORS TOGETHER ON WORLD STATE WITH 1000 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE RERUN THE ELICITATION AFTER WE ANSWER EACH BATCH OF 20 QUESTIONS. 61
FIGURE 48: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING RULE-BASED ELICITOR AND RANDOM QUESTION SELECTION ON WORLD STATE WITH 1000 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE ALSO COMPARE ALL SYSTEMS WITH ONE ANOTHER. 62
FIGURE 49: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING HEURISTIC AND RULE-BASED ELICITORS ON WORLD STATE WITH 500 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. 64

FIGURE 50: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING SEARCH AND RULE-BASED ELICITORS ON WORLD STATE WITH 500 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE RERUN THE ELICITATION AFTER WE ANSWER EACH BATCH OF 20 QUESTIONS. 65
FIGURE 51: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING SEARCH, HEURISTIC, AND RULE-BASED ELICITORS TOGETHER ON WORLD STATE WITH 500 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE RERUN THE ELICITATION AFTER WE ANSWER EACH BATCH OF 20 QUESTIONS. 66
FIGURE 52: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING RULE-BASED ELICITOR AND RANDOM QUESTION SELECTION ON WORLD STATE WITH 500 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE ALSO COMPARE ALL SYSTEMS WITH ONE ANOTHER. 67
FIGURE 53: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING HEURISTIC AND RULE-BASED ELICITORS ON WORLD STATE WITH 100 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. 69
FIGURE 54: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING SEARCH AND RULE-BASED ELICITORS ON WORLD STATE WITH 100 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE RERUN THE ELICITATION AFTER WE ANSWER EACH BATCH OF 20 QUESTIONS. 70
FIGURE 55: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING SEARCH, HEURISTIC, AND RULE-BASED ELICITORS TOGETHER ON WORLD STATE WITH 100 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE RERUN THE ELICITATION AFTER WE ANSWER EACH BATCH OF 20 QUESTIONS. 71
FIGURE 56: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING RULE-BASED ELICITOR AND RANDOM QUESTION SELECTION ON WORLD STATE WITH 100 UNKNOWNS. DASHED LINES SHOW THE SCHEDULE QUALITY AS ESTIMATED, WHEREAS SOLID LINES SHOW THE ACTUAL SCHEDULE QUALITY. WE ALSO COMPARE ALL SYSTEMS WITH ONE ANOTHER. 72
FIGURE 57: RESOURCE FUNCTIONS FOR A SESSION THAT REQUIRES TABLES. 75
FIGURE 58: EXAMPLE COST PENALTY FUNCTION. 76
FIGURE 59: COMPUTATION OF THE LIST OF BEST VENDOR ORDERS TO PLACE. 77
FIGURE 60: VENDOR ELICITATION ALGORITHM. 78
FIGURE 61: DEPENDENCY OF THE SCHEDULE QUALITY ON THE NUMBER OF ANSWERED QUESTIONS USING VENDOR ELICITOR. RED LINES SHOW THE RANDOM SELECTION, WHEREAS BLUE LINES SHOW THE VENDOR ELICITOR. 79

Table of Tables

TABLE 1: TYPES OF REQUESTS TO THE USER. THE SYSTEM MAY ASK THE USER TO FIND OUT MORE INFORMATION ABOUT AVAILABLE RESOURCES AND SCHEDULING CONSTRAINTS, AND TO SCHEDULE SOME EVENTS MANUALLY 3
TABLE 2: AVAILABLE ROOMS AND THEIR PROPERTIES. 8
TABLE 3: EVENTS AND RELATED CONSTRAINTS. 8
TABLE 4: EXAMPLES OF DISTANCE CONSTRAINTS: WE CAN SPECIFY MAXIMAL AND MINIMAL ALLOWED DISTANCES (IN FEET) BETWEEN EVENTS. 11
TABLE 5: EXAMPLES OF RELATIVE-TIME CONSTRAINTS. 11
TABLE 6: SCHEDULING PREFERENCES; WE SPECIFY PREFERENCE FUNCTIONS USING A THREE-POINT SCHEME, WHERE THE "MIN" VALUE OF THE ARGUMENT CORRESPONDS TO THE −5.0 QUALITY VALUE, "GOOD" GIVES THE QUALITY OF 0.0, AND "BEST" GIVES THE QUALITY OF 1.0. 14
TABLE 7: BEAN AUDITORIUM WITH PRIORITIZED PROPERTIES. 23
TABLE 8: AVAILABLE ROOMS AND THEIR PROPERTIES. 41
TABLE 9: EVENTS AND RELATED CONSTRAINTS. 41
TABLE 10: LIST OF WORLD STATES WE USE FOR EVALUATION. 52
TABLE 11: TWO TAILED T-TEST APPLICATION COMPARING DIFFERENT ELICITATION SYSTEMS FOR WORLD STATE WITH 3300 UNKNOWNS. 58
TABLE 12: PERCENTAGE OF THE GENERATED QUESTIONS THAT WE HAD TO ANSWER IN ORDER TO ACHIEVE 85% OF THE FULLY CERTAIN SCHEDULE QUALITY FOR EACH SYSTEM FOR WORLD STATE WITH 3300 UNKNOWNS. 58
TABLE 13: TWO TAILED T-TEST APPLICATION COMPARING DIFFERENT ELICITATION SYSTEMS FOR WORLD STATE WITH 1000 UNKNOWNS. 63
TABLE 14: PERCENTAGE OF THE GENERATED QUESTIONS THAT WE HAD TO ANSWER IN ORDER TO ACHIEVE 85% OF THE FULLY CERTAIN SCHEDULE QUALITY FOR EACH SYSTEM FOR WORLD STATE WITH 1000 UNKNOWNS. 63
TABLE 15: TWO TAILED T-TEST APPLICATION COMPARING DIFFERENT ELICITATION SYSTEMS FOR WORLD STATE WITH 500 UNKNOWNS. 68
TABLE 16: PERCENTAGE OF THE GENERATED QUESTIONS THAT WE HAD TO ANSWER IN ORDER TO ACHIEVE 85% OF THE FULLY CERTAIN SCHEDULE QUALITY FOR EACH SYSTEM FOR WORLD STATE WITH 500 UNKNOWNS. 68
TABLE 17: TWO TAILED T-TEST APPLICATION COMPARING DIFFERENT ELICITATION SYSTEMS FOR WORLD STATE WITH 100 UNKNOWNS. 73
TABLE 18: PERCENTAGE OF THE GENERATED QUESTIONS THAT WE HAD TO ANSWER IN ORDER TO ACHIEVE 85% OF THE FULLY CERTAIN SCHEDULE QUALITY FOR EACH SYSTEM FOR WORLD STATE WITH 100 UNKNOWNS. 73
TABLE 19: A SUBSET OF THE SERVICES WE REPRESENT IN THE SYSTEM. 74
TABLE 20: TYPES OF REQUESTS ABOUT VENDORS TO THE USER. THE SYSTEM MAY ASK THE USER TO FIND OUT MORE INFORMATION ABOUT AVAILABLE VENDORS, AND RESOURCE ITEMS OFFERED BY THOSE VENDORS. 78

1. INTRODUCTION

1.1. Motivation

When we work on a practical scheduling task, we usually do not have complete knowledge of the related resources and constraints. For example, when scheduling a conference, we may not know the exact sizes of available rooms or equipment needs of some speakers. The task of constructing a schedule based on incomplete data gives rise to several related problems, including the representation of uncertainty, efficient search for schedules, and elicitation of additional data that help to reduce uncertainty.

Uncertain information has a negative effect on the quality of schedules produced by an optimizer as the optimizer is forced to effectively "guess" the missing information, most of the time erring on the side of safety and not utilizing the available resources to their full extent. Resolving this uncertainty by asking questions on all possible uncertain pieces of information is impractical and in most cases impossible as no human user would be willing to answer thousands upon thousands of questions which may be typical in a practical scheduling task. Furthermore, some questions may not make sense to ask. For example, if we are trying to assign a hotel room to a guest who stated no preferences and there are no non-smoking rooms available, asking the guest's smoking preference is not very useful. Likewise, when trying to find matches on a social networking application we would not consider a question about a person's unknown height to be important, if all potential matches for that person stated that they do not care about height. An effective elicitation method would produce an ordering of potential questions maximizing the increase in quality of the assignment as each question is answered.

Although researchers have long realized the importance of uncertain information in scheduling and optimization problems, the related work has been limited [Sahinidis, 2004; Bidot, 2005]. Researchers have developed several domain-specific systems for optimization based on incomplete data [Chajewska et al., 1998; Averbakh, 2001; Lodwick et al., 2001; Moore, 2002; Balasubramanian and Grossmann, 2003; Lin et al., 2004]; however, they have not studied a general problem of scheduling under uncertainty.

We have investigated the problem of elicitation to aid in scheduling based on uncertain information about available resources and user preferences. The previous techniques have turned out inapplicable to this problem, and we have developed a new mechanism for improving scheduling under uncertainty.

1.2. Overview of the Results

We evaluate our approach to elicitation in the conference scheduling and vendor order domains. We show that our approach satisfies our research hypothesis and in particular it produces a ranking of questions that is significantly better at improving the quality of optimized schedules than the rankings produced by simple heuristics or random picking of questions. For example, in Figure 1 we show the evaluation results comparing

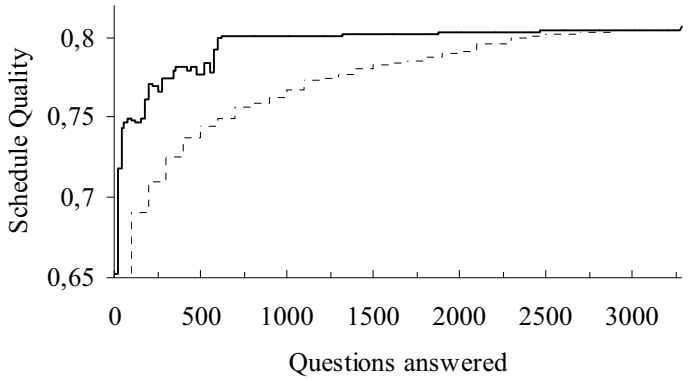


Figure 1: Comparison of our elicitation approach (solid) with random picking of questions (dashed) for conference scheduling problem with 3300 potential questions.

our system to the random picking of questions for a conference scheduling problem with 3300 possible questions to ask to the human user. In order to achieve 85% of the fully certain schedule quality, our elicitation approach needs the human user to answer only 17.5% of the questions while random picking of questions needs 45%. This difference becomes even more prominent when we look at the number of questions we need to achieve 95% of the fully certain schedule quality; our approach only needs 18% of the questions while random picking requires 62%. In vendor orders domain, our approach needs only 5% of the questions to be answered in order to achieve fully certain schedule quality while random picking needs more than 90% of the questions.

1.3. Overview of the Approach

We introduce the domain we chose and architecture we use for our approach and explain each component of the overall system. We also state the key contributions of our research.

1.3.1. DOMAIN

We use an academic conference as our main development and evaluation domain. Rooms are our resources and we need to assign them to sessions. We try to produce a schedule of high quality based on known constraints and preferences.

Rooms have a set of physical properties such as room size, seating capacity, and so on as well as properties relating to resources placed in the rooms such as microphones and projectors. We also keep information about distances between rooms.

Each event has an importance and specifies preferences and constraints relating to the room we may assign it to.

Most importantly, we allow having uncertain information about room properties, session importances, and session preferences and constraints.

1.3.2. ARCHITECTURE

The scheduling architecture consists of the components shown in Figure 2. We briefly describe the role of these components. We have given a more detailed description of these components in the chapters on the representation of uncertainty (Chapter 2), and scheduling based on uncertain knowledge (Chapter 3).

World model: This component maintains the description of a scheduling scenario, which includes the information about resources, constraints, and current schedule. It keeps a persistent copy of the world model on disk, and a fast-access copy in memory. We cover the representation in Chapter 2

Scorer: The scoring module is a fast procedure for evaluating schedule quality, which computes the expected quality of each assignment. The system uses it for automatic scheduling and for feedback during manual scheduling. We cover the scorer as a module for the scheduler in Chapter 3.

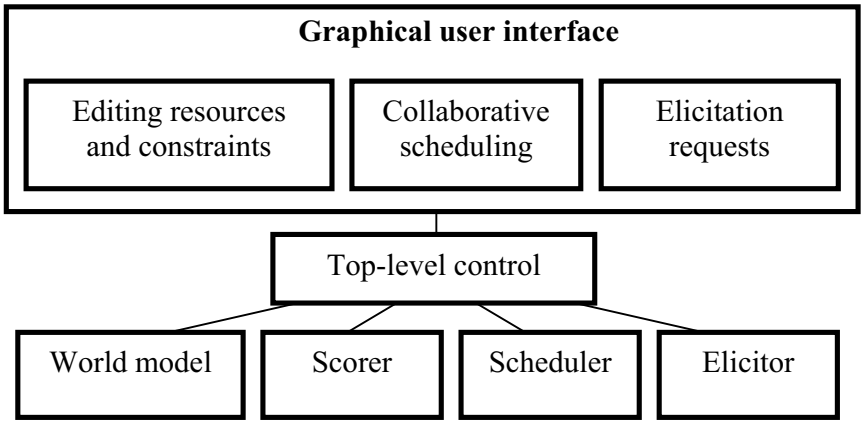


Figure 2: Architecture of the scheduling system.

- Provide the exact value for an uncertain room property.
  Example: Find out the size of the conference room.
- Provide the exact value for an uncertain request importance.
  Example: Find out the importance of the demo.
- Provide the exact specification for a set of acceptable values for start time, end time, duration, or room property in an event description.
  Example: Find out the acceptable duration of the demo.
- Provide the exact specification for a set of preferred values for start time, end time, duration, or room property in an event description.
  Example: Find out the preferred room size for the discussion.
- Select a room and time for an event.
  Example: Select a time slot for the workshop.

Table 1: Types of requests to the user. The system may ask the user to find out more information about available resources and scheduling constraints, and to schedule some events manually.

Scheduler: The scheduling module uses the description of rooms and events, and searches for a schedule of high expected quality. The search algorithm is based on hill-climbing and it does not guarantee optimality. If we apply this algorithm to construct a new schedule, it begins with the initially empty schedule and gradually improves it. If we use it to repair a schedule after changing resources or constraints, it starts with the old schedule. At each step, it either assigns a slot to some unscheduled event, or moves some scheduled event to a better slot. It continues the search until it cannot find further improvements, or until reaching a time limit. We cover the scheduler in Chapter 3.

Elicitor: The algorithm behind this module is the main focus of this dissertation.. The elicitation module determines whether the scheduler needs manual help, and generates respective requests to the user. We list the request types and give example requests in Table 1.

For each potential request, the elicitor analyzes the expected schedule improvement due to the user's help, and the expected human effort of addressing this request; it evaluates the utility of a request by the difference between the expected improvement and the required effort. The system selects the requests with positive utility, ranks them from the highest to the lowest utility, and displays them in this order. We cover the elicitation process in Chapter 4.

Top-level control: This module coordinates the invocation of the other modules, and it also routes data among them. Currently, it uses simple control procedures; we are now working on a more intelligent version, which will include heuristics and learning mechanisms for making the best use of the search algorithm, and for improving its co-ordination with the manual scheduling.

Interface: The graphical user interface consists of three main screens, as shown in Figure 2. The first screen is for editing the description of rooms and events, the second is for constructing and repairing conference schedules, and the third is for keeping track of the requests generated by the elicitor.

1.3.3. KEY CONTRIBUTIONS

In this work, we have investigated a novel approach to information elicitation and this has led to three main contributions:

- As a part of the elicitation process, we use a fast heuristic computation of the expected utility of potential questions to determine initial question rankings. We explain this process in detail in Section 4.2.
- We use B* search [Berliner, 1979] for refining question rankings. Section 4.3 covers this process.
- We employ a synergy of domain-independent and domain-specific elicitation techniques under a unified elicitation approach. The end result is our Integrated Elicitor which we cover in Section 4.4.

1.4. Related Work

Elicitation of missing knowledge is important in a wide range of domains, from recommender applications on the web [Burke, 1999; Stolze and Ströbel, 2003], to

understanding preferences of auction participants [Chen and Pu, 2004]. We review some of these applications and discuss their relation to our problem.

Burke's [1996] FindME systems use domain knowledge to provide the users with assisted browsing when the users cannot specify the attributes of the object the users are trying to retrieve. The process involves two steps: an initial query and the subsequent quick modifications or tweaks to the results by the user. The tradeoffs, category boundaries, and search strategies within a given domain have to be known, and the user may need to do a long series of attribute tweaks [Burke *et al.*, 1997]. [Burke, 1999; Stolze and Ströbel, 2003] apply this method to different domains, such as video rental and general-purpose shopping. This technique is impractical for our problem because the amount of tweaking needed in a domain with continuous variables would render the problem intractable, and we would need to provide an impractical amount of initial knowledge.

Gajos and Weld [2005] employ preference elicitation to find the best interface design for the user. Their method involves example critiquing and asking actual elicitation questions; however, it is limited to binary queries and to the elicitation of preference weights.

Pu [2003b] gives guidelines and empirical results for designing a good interface for preference elicitation. Linden [1997], Torrens [2003], and Pu and Faltings [2000; 2002; 2003a] employ example critiquing to assist the user in finding the best airline tickets. In all cases, the user needs to answer a lot of questions for each flight selection, which makes the underlying technique impractical for our problem, which involves a large number of selections. Faltings states that, for each travel planning task, thirty examples needed to be shown at each step of tweaking [Faltings *et al.*, 2004]. Even compounding critiques together to form more complex critiques at each step [McCarthy, 2005] fails in our domain. This is because, although they may reduce the amount of critiquing the user has to provide, they cannot deal with continuous preference functions.

Stolze [2003; 2004] introduces a different kind of preference elicitation. He suggests that users are more comfortable in providing the intended use for the desired object rather than desired attributes. Once the search returns the top matches, the user can fine-tune attributes to get different matches. This methodology was used commercially by AOL's PersonaLogic system, Active Buyer Guide, and PurchaseSource systems [Stolze and Rjaibi, 2001]. Stolze and Ströbel [2001] also go over a method for building a decision tree in order to decide which questions to ask.

Boutilier [1997; 2003c] addresses a problem similar to our research: doing elicitation in order to improve a solution to an optimization problem. He uses Ceteris Paribus Nets to represent preferences where conditional dependence and independence can be contained under a ceteris paribus interpretation [Boutilier *et al.*, 2003a, Boutilier *et al.*, 2004a], as well as generalized additive independence models [Braziunas and Boutilier, 2005]. Boutilier deals with linear utility constraints and allows uncertain preferences, which is similar to our approach; he decides which question to ask by considering the possible reduction in maximum regret in general purpose domains [Wang and Boutilier, 2003; Boutilier *et al.*, 2005] and specific domains, such as resource allocation in computing systems [Boutilier *et al.*, 2003b; Patrascu *et al.*, 2005]. His approach is limited to elicitation in problems with discrete variables. For example, the resource allocation

problem involves eliciting the desired amount of memory out of an explicit set of possible values.

Sometimes, user preferences can be approximated through preferences of other "similar" users. A new user expresses preferences for a set of items, and the system finds other users who have similar ratings. The challenge here is picking the right set of preferences to ask the new users to provide. For example, some approaches try to identify items that the user will most likely have a preference about, or items that would potentially give most value to the recommender system [Rashid *et al.*, 2002]. Boutilier used expected value of information in order to choose which questions to ask [Boutilier and Zemel, 2003; Boutilier *et al.*, 2003d]. This "collaborative filtering" approach has been used for making recommendations to the users on Internet news [Resnick *et al.*, 1994], videos [Hill *et al.*, 1995], music [Shardanand and Maes, 1995], and products provided by eBay and Amazon [Schafer *et al.*, 2001]. This approach requires users to rank related items; the sheer number of possible "items" as well as failing to account for the fine-grained detail in user preferences makes this approach impractical for our problem.

Burke [2000b] uses similarity-based heuristics to make recommendations based on past user ratings of available options. These heuristics include Euclidean distance [Ha and Haddawy, 1998] and probabilistic distance [Ha and Haddawy, 2003]. This collaborative filtering approach requires a lot of knowledge engineering. A lot of past ratings are required though, which creates problems in domains like ours. Burke [2000a] also noted a possibility to integrate collaborative filtering with the knowledge-based approach; however, it would not solve our problem either since our domain would require an impractically large knowledge base.

Another approach using similarity between utility models of different users is producing clusters of known utility functions [Chajewska *et al.*, 1998]. When a new user's utility function needs to be elicited, the system asks questions that allow finding the related cluster. The shortcomings of this approach are similar to the previous approach by Burke. A big database of utility models is needed for effective clustering. Furthermore, as the utility models can widely differ and are very complex in our domain, we are very likely to end up with lots of clusters leading to quite a few questions needed to differentiate between them.

Preference elicitation is also important in reducing the number of queries in combinatorial auctions, where the system needs to figure out which bundle of items the user would be most satisfied with [Smith *et al.*, 2002; Boutilier *et al.*, 2004b; Sandholm and Boutilier, 2006]. Common variations on this kind of elicitation include incremental auctions [Kress and Boutilier, 2004] and using value queries [Zinkevich *et al.*, 2003; Blum *et al.*, 2004].

The existing research in the area has been successful in finding and improving methods for doing preference elicitation in various scenarios; however, in these scenarios either an optimizer is not the target "client" for the elicitation or the elicitation domain is assumed to be simpler than our domain. We do not make an assumption about the variables in our domain being discrete or the number of possible variations in the preferences. We also use B* search [Berliner, 1979], which was originally developed for playing games such as chess or backgammon against human opponents, into the elicitation process. Furthermore, our elicitation system is tightly connected with the

optimizer and our approach's main objective is getting the most important information for improving optimization.

1.5. Thesis Outline

In this thesis, we first introduce the domain we use for applying our elicitation approach: Planning an academic conference. We cover the main objects in this domain including rooms, events and the concept of a "schedule" which provides the means for assigning events to time slots in rooms. We explain how we measure the quality of a schedule and introduce how we represent uncertainty which may exist in our knowledge about rooms and events. We give information on inference rules which the system uses in generating common sense assumptions when information is unavailable. We then introduce the optimization algorithm which we use to produce a schedule.

As expected, we dedicate most of the document to details of the elicitation algorithm. We introduce the elicitation problem in the conference planning domain and the three different kinds of elicitors we use in order to improve the quality of schedules produced by the optimization algorithm. We also give, in detail, the evaluation methodology we follow and the results of our evaluation using different scenarios. We introduce another domain, placing vendor orders for sessions in a conference and evaluate our approach in this domain.

2. REPRESENTATION

2.1. Scheduling Problem

We begin with an example of a conference scenario, and use it to illustrate the representation of resources and constraints. Suppose that we need to assign rooms to events at a small one-day conference, which starts at 11:00am and ends at 4:30pm, and that we can use three rooms: Bean Auditorium, Wean 100, which is a classroom, and Wean 250, which is a conference room (Table 2). These rooms host other events on the same day, and they are available for the conference only at the following times:

Bean Auditorium:	11:00am–1:30pm and 3:30pm–4:30pm.
Wean 100:	11:00am–2:30pm.
Wean 250:	12:00pm–4:30pm.

We describe each room by a set of properties; in this example, we consider three properties:

Size:	Room area in square feet.
Mikes:	Number of microphones.
Stations:	Maximal number of demo stations that can be set up in the room.

The conference includes five events: demonstration, discussion, tutorial, workshop, and conference-committee meeting (Table 3). For each event, the conference committee specifies its importance, as well as constraints and preferences on its time and room properties. We construct a schedule by assigning a room and time slot to every event; we give an example schedule in Figure 3.

	Bean Auditorium	Wean100	Wean 250
Size	1200	700	500
Stations	10	5	5
Mikes	5	1	2

Table 2: Available rooms and their properties.

	Demo	Discussion	Tutorial	Committee	Workshop
Importance	50	30	75	10	50
Start time	Any	Any	11am	3pm–4pm	Any
End time	Any	Any	1pm	3pm-4pm	Any
Duration	≥60	≥30	≥30	≥15	≥60
Room size	≥600	≥200	≥400	≥400	≥600
Stations	≥5	Any	Any	Any	Any
Mikes	Any	≥2	≥1	Any	≥1

Table 3: Events and related constraints.

<table>
<tr><th></th><th>Bean Auditorium</th><th>Wean 100</th><th>Wean 250</th></tr>
<tr><td>11:00</td><td rowspan="5">Demo</td><td>Tutorial</td><td>Unavailable</td></tr>
<tr><td>11:30</td><td rowspan="4">Workshop</td><td rowspan="3"></td></tr>
<tr><td>12:00</td></tr>
<tr><td>12:30</td></tr>
<tr><td>1:00</td></tr>
<tr><td>1:30</td><td rowspan="4">Unavailable</td><td rowspan="5">Unavailable</td></tr>
<tr><td>2:00</td></tr>
<tr><td>2:30</td></tr>
<tr><td>3:00</td><td rowspan="3">Discussion</td></tr>
<tr><td>3:30</td><td rowspan="2">Committee meeting</td></tr>
<tr><td>4:00</td></tr>
</table>

Figure 3: Example schedule.

We define constraints on acceptable schedules by limiting appropriate start times, end times, durations, and room properties for each event. For example, we may specify that the committee meeting starts and finishes between 3:00pm and 4:00pm, has an acceptable duration of 15 minutes or more, and the minimal acceptable room size for it is 400 square feet. In Table 3, we give example constraints for all five events; note that the schedule in Figure 3 satisfies these constraints.

2.2. Resources and Constraints

We now describe the representation of available resources, scheduling requirements, and specific schedules. We use *room* objects to represent resources; *event* objects to represent events and constraints; and *assignment* objects to represent selection of a room and time slot for each event.

2.2.1. ROOMS

The main resource represented in the current system is rooms, which may be in multiple buildings. The system also keeps data about other relevant resources related to rooms, such as portable equipment or services.

We represent a room by a name and a list of properties, such as its size, number of microphones, and the building containing it. The system allows the user to define an arbitrary list of room properties, where each property is either numeric or nominal; for instance, the size of a room is a number, whereas the building that contains a room is a nominal value. In Table 2, we give an example of three rooms, represented by three properties.

We also specify distances between rooms, which represent some measure of the difficulty of getting from one room to another. The distances may not be in feet; for example, we may measure distances in walking minutes.

For each room, we specify its availability for the conference, represented by a collection of time intervals. For instance, Bean Auditorium in the example is available for two intervals: 11:00 am – 1:30 pm and 3:30 pm – 4:30 pm.

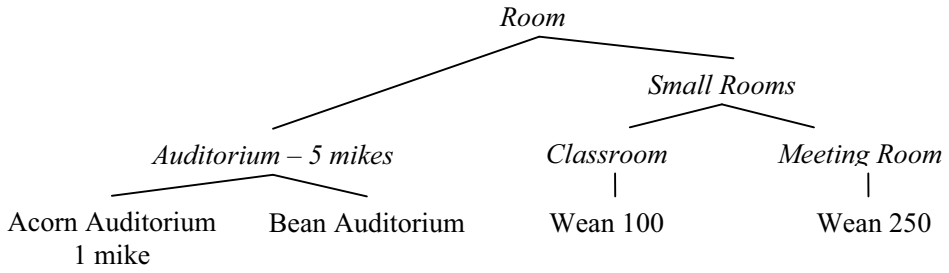


Figure 4: Example of a room hierarchy.

We arrange rooms into a tree-structured hierarchy, where the top node corresponds to the set of all rooms, other nonleaf nodes are specific room types, and the leaves are individual rooms. In Figure 4, we give an example of a room hierarchy with four room types: auditoriums, small rooms, classrooms, and meeting rooms.

We use the hierarchy to specify default property values, which are inherited by descendant nodes. For example, consider the hierarchy in Figure 4. The default number of microphones for the auditoriums is 5, and this value is inherited by any auditorium without a specified number of microphones. Bean Auditorium inherits this value, whereas Acorn Auditorium has an explicitly specified value, which differs from the default. We also use the hierarchy to help us deal with uncertainty about room properties. If a given room property is not known, the system can estimate a value from nodes higher in the hierarchy. For example, if we define small rooms to be between 500 and 800 square feet, the system would estimate the size of a meeting room to be in that range in the absence of an explicit room size.

2.2.2. EVENTS

We next describe the representation of conference events, such as workshops and demo sessions, and related scheduling constraints. The representation of an event includes its name, importance, list of hard constraints on scheduling the event, and list of soft preferences.

We represent an event importance by a positive integer; the higher this value, the greater the importance. For instance, the importance of the workshop in the motivating example is 50, whereas the importance of the committee meeting is 10 (Table 3); intuitively, it means that finding a good time slot for the workshop is five times more important than that for the committee meeting.

When specifying an event, we can impose hard constraints on its start time, end time duration, and properties of a room allocated for this event, as well as its time and position with respect to other events. We define constraints by sets of acceptable values for each of these parameters. We give an example of such constraints in Table 3; for instance, the committee meeting requires a room of size at least 400 square feet, and it has to start no earlier than 3 pm and end no later than 4 pm, and continue for at least 15 minutes.

- **Room-property constraints:** For each room property, we can define acceptable values, which limit the rooms that can be used for the event. For a numeric

property, we define acceptable values by a list of nonoverlapping intervals; for a nominal property, we define a list of acceptable values.

- **Distance constraints:** We may constrain the maximal and minimal allowed distance of an event from other events. For example, as shown in Table 4, the demo needs to be at least 100 and at most 300 feet from the tutorial session. Note that these constraints involve pairs of events, which makes them different from room-property constraints.
- **Temporal constraints:** We may define two types of temporal constraints. The constraints on the start time, end time, and duration of an event, defined by a set of acceptable intervals are first type of temporal constraints. For example, the tutorial in the motivating example should start no earlier than 11 am, end no later than 1pm, and its duration should be at least 30 minutes (Table 3). Constraints on the relative start times of events are the second type of temporal constraints. We may specify that the difference between the start times of two events, or start time of one event and end time of another, should be within a certain interval. For example, we may specify that the discussion should be immediately before the tutorial, or that the demo should start between 2 and 3 hours before the discussion and should not overlap with the committee meeting (Table 5).

First Event	Second Event	Min. Distance	Max. Distance
Demo	Tutorial	100	300
Demo	Committee	150	250
Demo	Workshop	Any	100
Discussion	Committee	Any	100
Discussion	Workshop	100	150
Tutorial	Committee	Any	200

Table 4: Examples of distance constraints: We can specify maximal and minimal allowed distances (in feet) between events.

First Event	Second Event	Constraint
Demo	Discussion	2–3 hours before
Discussion	Tutorial	Immediately before
Workshop	Demo	Immediately after
Demo	Committee	Non-overlap

Table 5: Examples of relative-time constraints.

2.2.3. SCHEDULE

When the system builds a schedule, it must satisfy all hard constraints; for instance, if the constraints are as shown in Table 5, it must not consider a schedule where the discussion is after the tutorial.

To build a schedule, we need to assign a specific room and time slot to each event. We represent this assignment by four variables: a pointer to the event, a pointer to a room, a start time, and a duration. Alternatively, we can decide that an event is not a part of the schedule, which is also considered an assignment. We call such an event *rejected*, and represent it internally by setting the related room pointer to NIL.

A *partial schedule* is a set of assignments, where different assignments correspond to different events, and some events do not have assignments. A *full schedule* is a set of assignments that includes exactly one assignment for each event. For example, the schedule in Figure 5 is a partial schedule, where we have assigned rooms for the committee meeting, tutorial, and discussion, and decided that we do not include demo into the conference, but we have not yet made a decision for the workshop.

Note that we distinguish unassigned and rejected events. An event is rejected if we have decided that it is not a part of the conference, whereas an event is unassigned if we have not yet made any decision. When the system builds a new schedule, it can temporarily mark some events as unassigned; however, the final schedule must not have unassigned events.

Also note that we always have an option of rejecting an event, regardless of its constraints, and a schedule with rejected events is valid. Thus, we can build a valid schedule by simply rejecting all events, regardless of their constraints; however, its quality would be very poor.

<table>
<tr><th></th><th>Bean Auditorium</th><th>Wean 100</th><th>Wean 250</th></tr>
<tr><td>11:00</td><td rowspan="5"></td><td rowspan="2">Tutorial</td><td>Unavailable</td></tr>
<tr><td>11:30</td><td rowspan="4"></td></tr>
<tr><td>12:00</td><td rowspan="5"></td></tr>
<tr><td>12:30</td></tr>
<tr><td>01:00</td></tr>
<tr><td>01:30</td><td rowspan="4">Unavailable</td></tr>
<tr><td>02:00</td><td rowspan="4">Unavailable</td></tr>
<tr><td>02:30</td><td rowspan="3">Discussion</td></tr>
<tr><td>03:00</td></tr>
<tr><td>03:30</td><td rowspan="2">Committee Meeting</td></tr>
<tr><td>04:00</td></tr>
<tr><td colspan="4">Rejected: Demo
Unassigned: Workshop</td></tr>
</table>

Figure 5: Example of a partial schedule.

2.3. Schedule Quality

We next define schedule quality, based on the notion of preferences, which represent soft constraints. We measure quality on the scale from −*penalty* to 1.0, where *penalty* is a nonnegative real value that represents the penalty for the worst possible schedule. Intuitively, the zero quality corresponds to satisfactory assignments, negative quality values represent poor assignments, and positive values mean unusually good assignments.

For each event, we specify preferences that represent the desirable selection of a room and time slot. We represent a specific preference by a piecewise-linear function that shows the dependency of the assignment quality on its start time, end time, duration, some room property, or relative distance or time with respect to other events. The domain of this function is the set of acceptable values for the related property, and the range is the quality values between −*penalty* and 1.0.

In Figure 6, we show an example preference, which determines the dependency of the assignment quality on the room size; in this example, the minimal acceptable room size is 600, and *penalty* is 5.0, which means that the function range is from −5.0 to 1.0.

Note that this function is monotonically increasing, and we can specify it by three values of the room size, which correspond to the segment endpoints: the minimal acceptable size (600), which corresponds to the quality of −penalty; the satisfactory size (1000), which corresponds to the zero quality; and the minimal value of the "perfect" size (1200), which corresponds to the quality of 1.0. Although the system allows arbitrary functions, we often use this three-point scheme, which matches the human intuition for many preferences. In Table 6, we give example preferences for the conference described earlier; in this example, we use this scheme to describe functions with the range from −5.0 to 1.0.

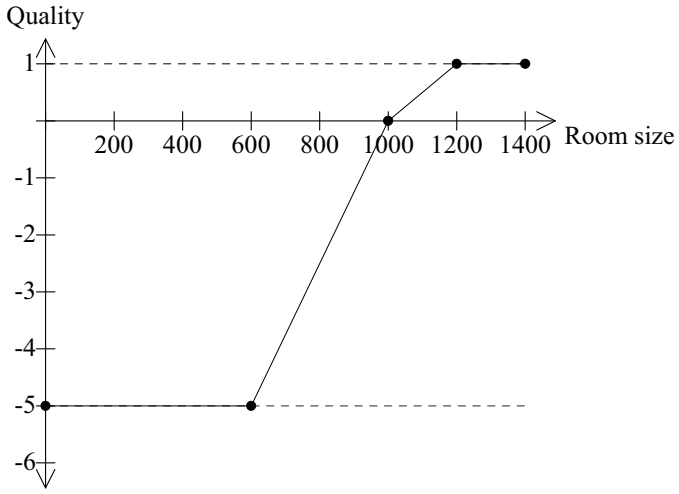


Figure 6: Example of a preference function, which shows the dependency of the assignment quality on the room size for the demo.

For each preference, we specify its weight, which is a positive integer that shows its relative importance compared to other preferences. In the example, the most important event is the committee meeting. The discussion session and the workshop are only half as important, and the tutorial and the demo session have very low importance (see Table 6).

If we reject an event, then this assignment has the worst possible quality, which is *–penalty*. If an event has a room and time slot, we instantiate the respective start time, duration, and room properties into the event's preference functions, and take the weighted sum of their values. If an event has k preference functions, their values are $p_1,\ldots, p_k$, and their weights are $w_1,\ldots, w_k$, then the assignment quality is

$$(w_1 \cdot p_1 + \ldots + w_k \cdot p_k) / (w_1 + \ldots + w_k). \quad (2.1)$$

		Demo	Discussion	Tutorial	Committee	Workshop
Importance		1	5	1	10	5
Duration	Min	60	30	30	15	60
	Good	120	60	45	30	75
	Best	150	90	60	60	120
Room size	Min	600	200	400	400	600
	Good	1000	400	600	600	800
	Best	1200	600	800	800	1000
Stations	Min	5		0		
	Good	10	Any	1	Any	Any
	Best	15		2		
Mikes	Min		2	1		1
	Good	Any	3	1	Any	1
	Best		4	2		1

Table 6: Scheduling preferences; we specify preference functions using a three-point scheme, where the "min" value of the argument corresponds to the −5.0 quality value, "good" gives the quality of 0.0, and "best" gives the quality of 1.0.

	Acorn Auditorium	*Wean 100*	*Wean 250*
11:00	Demo	Tutorial	***Unavailable***
11:30			
12:00			
12:30		Workshop	
1:00			
1:30	***Unavailable***		
2:00			
2:30		***Unavailable***	
3:00			Discussion
3:30	Committee meeting		
4:00			

Figure 7: Example schedule**.**

The overall schedule quality is the weighted sum of the quality values for individual assignments. That is, if a schedule includes n assignments, their quality values are $Qual_1,\ldots, Qual_n$, and their importances are $imp_1,\ldots, imp_n$, then the overall schedule quality is

(2.2) $$(imp_1 \cdot Qual_1 + \ldots + imp_n \cdot Qual_n) \,/\, (imp_1 + \ldots + imp_n).$$

For example, if we use the preferences in Table 6 and the schedule is as shown in Figure 7, then the quality of the time slot for the demo is 0.75, for the discussion is 0.75, for the tutorial is 0.62, for the committee meeting is 1.00, and for the workshop is −0.17, and the overall schedule quality is 0.50.

We use an optimization algorithm that uses the description of rooms and events, and searches for a high-quality schedule (see Chapter 3). The algorithm is based on hill-climbing and it does not guarantee optimality.

2.4. Uncertain Resources

When scheduling a conference, we may have incomplete data about resources, event importances, and preferences; for instance, we may not know the exact size of the conference room, or the relative importance of the demo and discussion. We represent uncertain values of room properties, event importances, and preference weights by probability density functions, approximated by collections of uniform distributions. Specifically, we encode an uncertain value by a set of disjoint intervals that may contain it, with a probability assigned to each interval; the sum of these probabilities is 1.0.

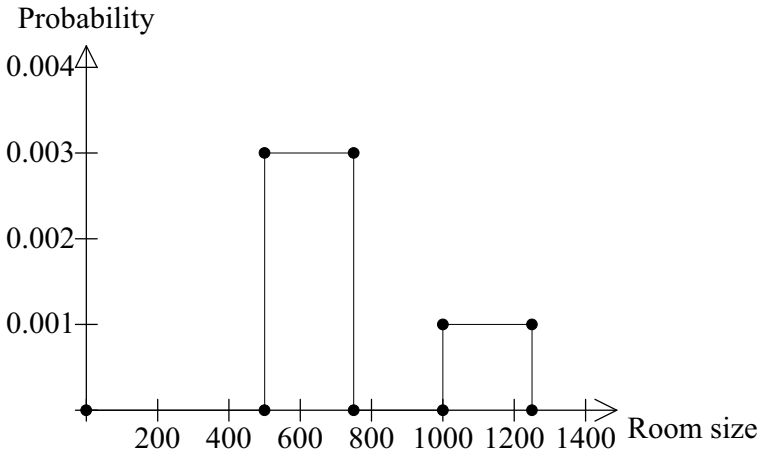


Figure 8: Probability density function for uncertain room size, which is between 500 and 750 with 0.75 probability, and between 1000 and 1250 with 0.25 probability.

In Figure 9(a), we summarize this encoding and give the related constraints on probabilities and endpoints of intervals.

For example, suppose that the exact size of the conference room is uncertain. Recent measurements suggest that it is between 500 and 750, whereas old records show that it is between 1000 and 1250. If we trust the measurements more than the old records, but not completely, we may assume that the size is between 500 and 750 with 0.75 probability, and between 1000 and 1250 with 0.25 probability; in Figure 8, we show the corresponding probability density function.

2.5. Uncertain Preferences

The representation of uncertain preferences is based on the combination of piecewise-linear functions with uncertain values. Specifically, we represent a preference by a piecewise-linear function that may have uncertain *y*-coordinates. For example, suppose that we need to encode a preference for a room size. Suppose further that the minimal allowed size is 600, and that 1200 is definitely enough, but we are uncertain about sizes between 600 and 1200. We believe that 800 may be an acceptable size, but there is a risk that it would be barely enough. We also believe that the size of 1000 should make the attendees perfectly happy, but there is a chance that some attendees would prefer a larger room. We may represent it by the function in Figure 10(a), where the quality for the size of 800 is an uncertain value between −5.0 and 0.0, and the quality for the size of 1000 is an uncertain value between 0.0 and 1.0.

We also allow specifying an uncertain preference by multiple functions and their probabilities. For example, suppose that some conference event requires at least 600 square feet, and the description of the event indicates that the optimal room size is 800, but a member of the conference committee has told us that the appropriate size is 1200. If we trust the committee member more than the description, but not completely, we may assume that the description is correct with probability 0.25. We then represent the size preference by two different piecewise-linear functions, with the probabilities of 0.75 and 0.25, as shown in Figure 10(b).

The developed system allows the use of uncertain quality values and multiple piecewise-linear functions at the same time; that is, we may specify several piecewise-linear functions with their probabilities, and use uncertain *y*-coordinates in each function.

2.6. Utility Function

If the description of rooms and events includes uncertainty, the schedule utility depends on the mathematical expectation of the quality and on the standard deviation of the expected quality. When the system constructs a schedule, it keeps track of the expected quality of candidate schedules. The quality computation is based on the assumption that all probability distributions are independent. If some of them are dependent, the computation does not give the exact expected quality, but it usually provides a good approximation.

(a) Representation of an uncertain value.

$prob_1$: from min_1 to max_1
$prob_2$: from min_2 to max_2
…
$prob_m$: from min_m to max_m

We describe an uncertain value by multiple intervals and respective probabilities, and we specify each interval by its minimal and maximal value. The intervals do not overlap, and the sum of the probabilities is 1.0, which means that we impose the following constraints on the related values:

$$min_1 \le max_1 \le min_2 \le max_2 \le \ldots \le min_m \le max_m$$
$$prob_1 + prob_2 + \ldots + prob_m = 1.0$$

(b) Representation of an uncertain function.

$prob_1$: (x_{11}, y_{11}), (x_{12}, y_{12}), …
$prob_2$: (x_{21}, y_{21}), (x_{22}, y_{22}), …
…
$prob_m$: (x_{m1}, y_{m1}), (x_{m2}, y_{m2}), …

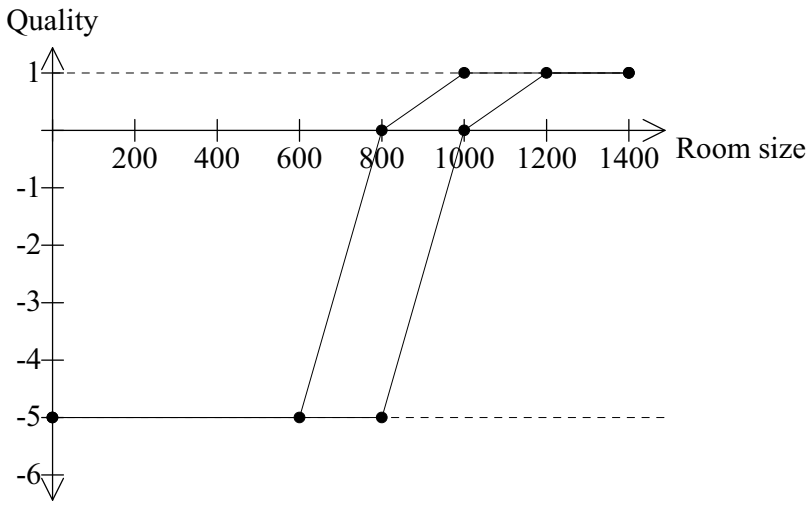
We describe an uncertain function by multiple piecewise-linear functions and respective probabilities. The description of each piecewise-linear function is a list of segment endpoints sorted by x-coordinate. The x-coordinate of a point must be a specific number, whereas its y-coordinate may be either a number or an uncertain value. For each piecewise-linear function, the x-coordinates of its endpoints are distinct:

$$x_{11} < x_{12} < \ldots$$
$$x_{21} < x_{22} < \ldots$$
$$\ldots$$
$$x_{m1} < x_{m2} < \ldots$$

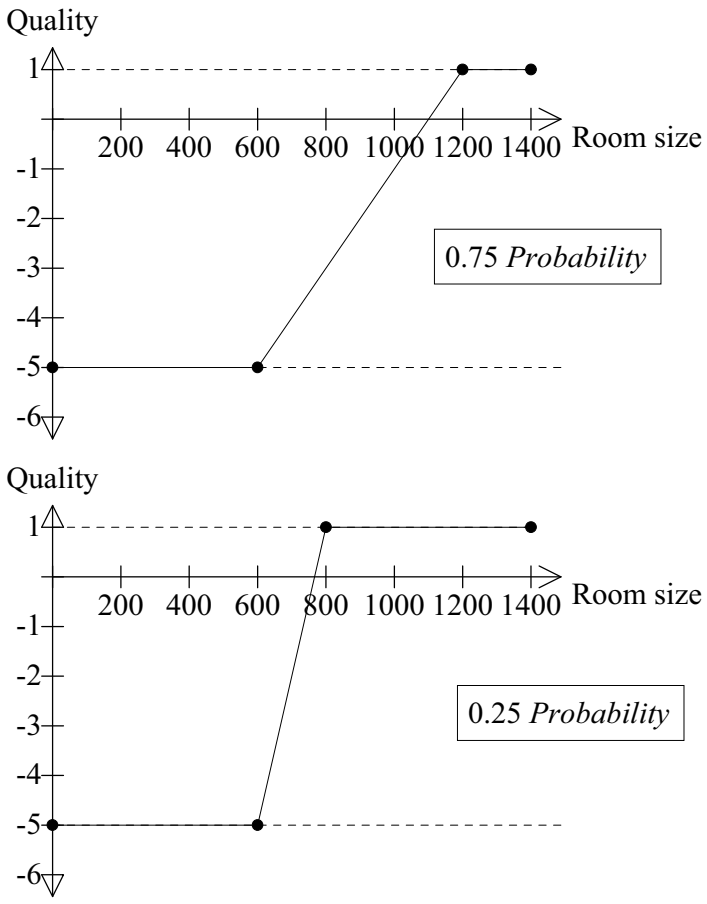
Furthermore, the probabilities that correspond to different piecewise-linear functions sum to 1.0:

$$prob_1 + prob_2 + \ldots + prob_m = 1.0$$

Figure 9: Encoding of uncertain values and functions. We use uncertain values to represent room properties, event importances, and preference weights, and uncertain functions to represent preferences**.**



(a) Piecewise-linear function with uncertain *y*-values.



(b) Two piecewise-linear functions and their probabilities.

Figure 10: Examples of uncertain preference functions; the first function includes uncertain quality values, whereas the second is encoded by two different functions, with probabilities of 0.75 and 0.25.

The algorithm inputs a fully certain preference function, represented by the coordinates of its endpoints, $x_1 \ldots x_l$ and $y_1 \ldots y_l$; and an uncertain value, represented by vectors $prob_1 \ldots prob_m$, $min_1 \ldots min_m$, and $max_1 \ldots max_m$, as shown in Figure 9(a). It returns the mathematical expectation of applying the certain preference function to the uncertain value. We assume that all intervals of the uncertain value are in the domain of the preference function, that is, $x_1 \le min_1 \le max_m \le x_l$. If the uncertain value does not satisfy this assumption, then it violates the related hard constraint, and the system does not apply the preference function to this value.

The algorithm consists of three procedures: PREF-FOR-CERTAIN, which determines the value of a piecewise-linear function for a fully certain argument, PREF-FOR-UNIFORM, which determines the expected value of a piecewise-linear function for an uncertain argument represented by a single uniform distribution, and EXPECTED-CERTAIN-PREF, which determines the expected value of a piecewise-linear function for an uncertain argument represented by multiple uniform distributions and their probabilities.

Input to PREF-FOR-CERTAIN is a piecewise-linear function, represented by vectors $x_1 \ldots x_l$ and $y_1 \ldots y_l$, and a fully certain argument *arg*. Output is the value of the function for this argument.

PREF-FOR-CERTAIN (x, y, l; *arg*)
Find index a such that either $arg = x_a$ or $x_{a-1} < arg < x_a$
if $arg = x_a$ **then return** y_a
return $(y_a \cdot arg - y_{a-1} \cdot arg + y_{a-1} \cdot x_a - y_a \cdot x_{a-1}) / (x_a - x_{a-1})$

Input to PREF-FOR-UNIFORM is a piecewise-linear function, represented by vectors $x_1 \ldots x_l$ and $y_1 \ldots y_l$, and a uniform-distribution argument, represented by the endpoints of the distribution, *min-arg* and *max-arg*. Output is the expected value of the function for this uncertain argument.

PREF-FOR-UNIFORM (x, y, l; *min-arg*, *max-arg*)
if *min-arg* = *max-arg* **then return** PREF-FOR-CERTAIN(x, y, l; *min-arg*)
Find index a such that either $min\text{-}arg = x_a$ or $x_a < min\text{-}arg < x_{a+1}$
Find index b such that either $max\text{-}arg = x_b$ or $x_{b-1} < max\text{-}arg < x_b$
min-y = PREF-FOR-CERTAIN (x, y, l; *min-arg*)
max-y = PREF-FOR-CERTAIN (x, y, l; *max-arg*)
if $a = b - 1$ **then return** (*min-y* + *max-y*) / 2
$sum = (x_{a+1} - min\text{-}arg) \cdot (min\text{-}y + y_{a+1}) / 2$
for $j = a + 1$ **to** $b - 2$ **do**
    $sum = sum + (x_{j+1} - x_j) \cdot (y_j + y_{j+1}) / 2$
$sum = sum + (max\text{-}arg - x_{b-1}) \cdot (y_{b-1} + max\text{-}y) / 2$
return *sum* / (*max-arg* − *min-arg*)

Input to EXPECTED-CERTAIN-PREF is a piecewise-linear function, represented by vectors $x_1 \ldots x_l$ and $y_1 \ldots y_l$, and an uncertain argument, represented by vectors $prob_1 \ldots prob_m$, $min_1 \ldots min_m$, and $max_1 \ldots max_m$, as shown in Figure 9(a). Output is the expected value of the function for the uncertain argument.

EXPECTED-CERTAIN-PREF (x, y, l; *prob*, *min*, *max*, m)
mean = 0
for $i = 1$ **to** m **do**
    $mean = mean + prob_i \cdot$ PREF-FOR-UNIFORM (x, y, l; min_i, max_i)
return *mean*

Figure 11: Computing the mathematical expectation of a fully certain piecewise-linear function applied to an uncertain property value.

The algorithm inputs an uncertain value, represented by the vectors $prob_1 \ldots prob_m$, $min_1 \ldots min_m$, and $max_1 \ldots max_m$, as shown in Figure 5(a). It returns the mathematical expectation of this uncertain value.

```
EXPECTED-UNCERTAIN-VALUE (prob, min, max, m)
mean = 0
for i 1 to m do
        mean = mean + prob_i · (min_i + max_i) / 2
return mean
```

Figure 12: Computing the mathematical expectation of an uncertain value, represented by a collection of uniform distributions and their probabilities.

The algorithm inputs an uncertain preference function, represented by a collection of functions, $pref_1 \ldots pref_m$, and their probabilities, $prob_1 \ldots prob_m$. Every $pref_i$ is a piecewise-linear function, which may include uncertain y-coordinates, as shown in Figure 9 (b). It also inputs an uncertain value *val*, represented by a collection of intervals and their probabilities, as shown in Figure 9(a). It returns the expected value of applying the uncertain preference function to the uncertain value.

```
EXPECTED-UNCERTAIN-PREF (prob, pref, m; val)
mean = 0
for i = 1 to m do
        for every uncertain y-coordinate in the representation of pref_i do
                Call EXPECTED-UNCERTAIN-VALUE to find the expected value of the uncertain y
                Replace the uncertain y in pref_i with its expected value
                Call EXPECTED-CERTAIN-PREF (see Fig. 7) to find E(pref_i(val)),
                        which is the expected value of applying the resulting fully certain function pref_i
                        to the uncertain value val
                mean = mean + prob_i · E(pref_i(val))
return mean
```

Figure 13. Computing the mathematical expectation of an uncertain preference function applied to an uncertain property value.

If a schedule violates some hard constraint with a nonzero probability, the overall schedule quality is $-penalty$ regardless of the other constraints. If the schedule satisfies all hard constraints, the system computes the expected quality of each assignment. To estimate the assignment quality for a specific event, it determines the expected values of the related preference functions, $E(p_1),\ldots, E(p_k)$, as well as the expected values of their weights, $E(w_1),\ldots, E(w_k)$, and uses them to compute the expected quality of the assignment, which is

$$\frac{E(w_1)\cdot E(p_1)+E(w_2)\cdot E(p_2)+\ldots+E(w_k)\cdot E(p_k)}{E(w_1)+E(w_2)+\ldots+E(w_k)}. \tag{2.3}$$

The procedure in Figure 12 finds the mean of a probability-density function represented by a collection of uniform distributions. We use it to determine the expected values of uncertain preference weights, $E(w_1),\ldots, E(w_k)$. The procedure in Figure 11 gives the expected value of a fully certain preference function applied to an uncertain argument, and the procedure in Figure 13 determines the expected value of an uncertain preference function. We use these procedures to compute the expected preference values, $E(p_1),\ldots, E(p_k)$.

The system uses the expected quality of assignments, along with the expected values of event importances, to compute the expected quality of the overall schedule,

$$\frac{\mathrm{E}(imp_1)\cdot\mathrm{E}(Qual_1)+\mathrm{E}(imp_2)\cdot\mathrm{E}(Qual_2)+\ldots+\mathrm{E}(imp_n)\cdot\mathrm{E}(Qual_k)}{\mathrm{E}(imp_1)+\mathrm{E}(imp_2)+\ldots+\mathrm{E}(imp_n)}. \tag{2.4}$$

2.7. Inference Rules

In the presence of uncertain or missing information, the system may be able to resolve some of the uncertainty through inference based on the available information. For example, a speaker may state that she expects to have an attendance of 45 people. If all we know about a room is its size in square feet then we cannot directly reason if a given room size in square feet satisfies the attendance preference for the event. The system needs a mechanism to convert the expected attendance into a size preference in square feet. An example inference rule that can be used in this case is:

> *If we do not know the room-size preference, but know the number of attendees, assume that the minimal acceptable size is ten square feet per person, a capacity of twenty square feet per person is good, and the most preferred capacity is thirty square feet per person* (*Figure 14*).

We use inference rules for reducing the uncertainty in room properties such as the size of a room as well as in event preferences such as in the example above.

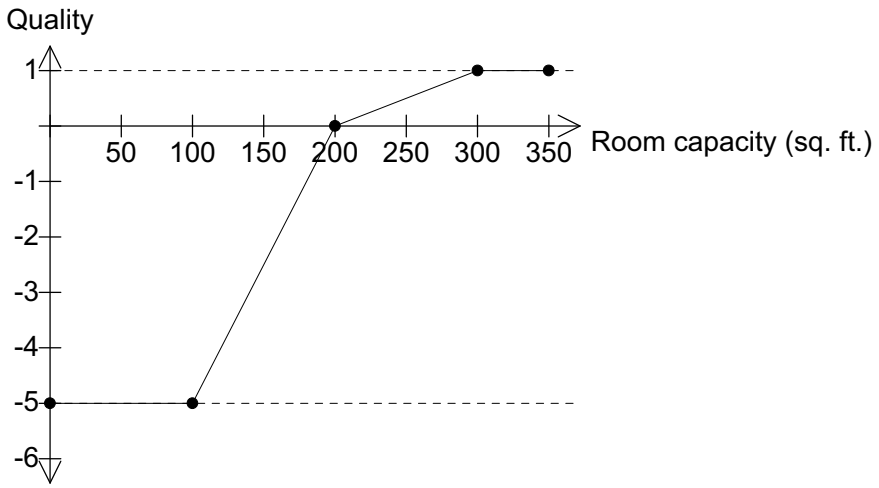


Figure 14: Result of applying the example rule with an expected attendance of 10 for the event.

2.7.1. RULE SYNTAX

An inference rule consists of five parts: a name, priority, applicability conditions, variables to be used by the rule, a target, and effects which set the value of the target. We use two sets of rules: *room rules* which affect rooms and *event rules* which affect events. The priority of a rule shows its "importance" and helps to resolve conflicts; a greater number means higher importance. The applicability conditions determine when the system uses the rule. For room rules, these conditions include a list of room types, as well as a list of room properties and respective value sets. For event rules, we include a list of event types and a list of room properties with value sets. A value set may have a list of nonoverlapping intervals or a special marker specifying that the room property must be absent or present. The marker has values SPECIFIED and UNSPECIFIED. For example in Section 2.7, we give an example of an event rule where we state that the *Expected Attendance* must be SPECIFIED and *Size Preference* must be UNSPECIFIED (Figure 15).

The list of variables includes the names of the known room properties used in computing default values. Each of these properties must be specified as either SPECIFIED or a list of intervals in applicability conditions. In the example, we compute the effects of the rule based on the room size preference, so the variables include only *Size Preference* in Figure 15(a).

The effects list is a list of room properties, event properties or preferences and related defaults. Each element includes a property name and its default value, which may be a specific value or an uncertain value expressed by a set of intervals with probabilities. The rules cannot have effects on any properties used in the rule's conditions; also, different elements of the list must refer to different properties. If an effect is a specific value, it may be a number, or any arithmetic or C# function based on properties listed as variables. In our example, the effect sets the size preference based on the expected attendance of the event Figure 15(b).

If the rule effect is a probability distribution, we represent it as a list of intervals, where each interval includes a probability, lower-end value, and upper-end value. The probability of an interval is a number, and the sum of these probabilities must be 1.0. Each lower-end and upper-end value may be either a number or a linear combination of properties listed as variables. For example, in Figure 15(a), we are inferring number of mikes in an auditorium when it is not specified. The rule states that with 30% probability auditoriums have only one microphone, and with 70% probability the number of microphones is given as a function of the room size.

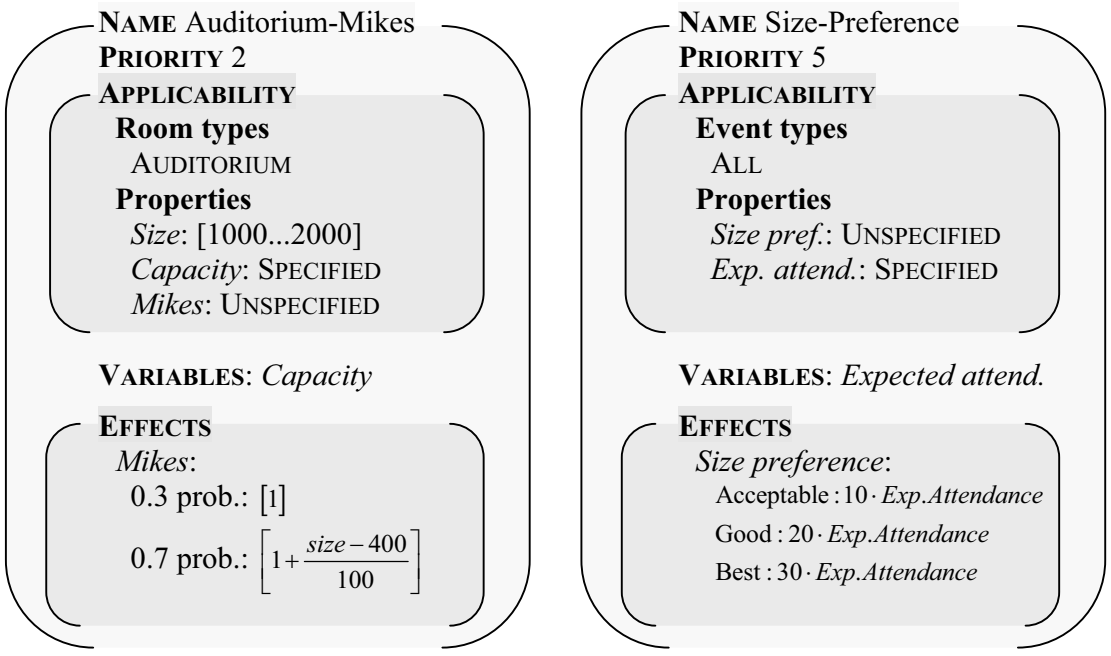


Figure 15: Example inference rules for rooms (a) and events (b).

Property	Value	Priority
Room type	AUDITORIUM	-
Size	1200	+∞
Mikes	UNSPECIFIED	-
Other-Capacity	150	+∞
Reception-Capacity	100	2
Banquet-Capacity	[100…150]	0

Table 7: Bean Auditorium with prioritized properties.

2.7.2. PRIORITIES OF PROPERTY VALUES

We next describe an algorithm for applying inference rules. Note that rules may conflict with each other and they may also conflict with values inherited from a parent (Figure 4 on page 10) and with explicitly specified values. To resolve these conflicts, the system assigns a priority to each known value, and compares it with priorities of conflicting rules.

For each known property value of each room, properties of events, and event preferences, the system keeps a priority, which is a natural number; note that the system does not keep priority of values marked to be explicitly unknown. Intuitively, this priority shows the reliability of the system's knowledge. For example, consider Bean Auditorium from Table 2. We add three new properties: Reception-Capacity, which is the number of people that can fit into the room when used for a reception, Banquet-Capacity, which is

the capacity when used for a banquet, and Other-Capacity, which is the capacity for other events (Table 7).

The knowledge of *Size* and *Capacity* in this example is very reliable (+∞), the knowledge of *Reception-Capacity* is unreliable (2), and the knowledge of *Banquet-Capacity* is even less reliable (0). If the user explicitly specifies a property, its priority is +∞. If a room inherits a property from its parent in the hierarchy, then the system marks it as INHERITED without a priority, and then uses the priority of inherited values specified as a parameter to the system; in the current implementation, the priority of all inherited values is the same.

2.7.3. RULE APPLICABILITY

A default rule is applicable to a specific room if it satisfies all of the following conditions.

- The type of the room is among the types specified in the rule.
- For every room property specified as SPECIFIED in the applicability conditions, the related value in the room specification is either a numeric value or a set of ranges of values with probabilities.
- For every room property marked as UNSPECIFIED in the applicability conditions, the related value in the room specification is UNSPECIFIED.

For every room property specified by a list of numeric intervals in the applicability conditions, the value in the room specification is either a numeric value in this range or a set of ranges of values that are completely contained in this range. In other words, the probability that this value is in the specified range must be 100%. For example, if a rule has an applicability condition stating that the banquet capacity of a room has to be between 75 and 200, Bean Auditorium (Table 7) would satisfy this condition. As another example, if the condition states that the banquet capacity has to be between 150 and 200, the rule is not applied to the Bean Auditorium.

2.7.4. APPLICATION PRIORITY

When the system applies a rule to a specific room, it computes the application priority; intuitively, this priority shows the reliability of the related default values. The application priority is the minimum among the rule's priority and the priorities of all known room properties that are listed in the rule's applicability conditions. Note that the system ignores the UNSPECIFIED properties when computing the priority. For example, suppose that we apply the “Auditorium-Mikes” rule (Figure 15a) to Bean Auditorium. The only known property in the applicability conditions is *Capacity*, and the priority for this property in Bean Auditorium is +∞, and the priority of the rule itself is 2. The application priority is the minimum of these three values, which is 2. We give the algorithm for calculating the application priority of an inference rule applied to a room in Figure 16.

The procedure inputs an inference rule, *rule*, and a room, *room*, and returns the application priority

```
APPLICATION-PRIORITY-ROOM(rule, room)

Min-Priority = PRIORITY-OF(rule)
for each property in APPLICABILITY-CONDITIONS-OF(rule)
        if property ≠ UNSPECIFIED
                Property-Name = GET-NAME(property)
                Room-Property = GET-PROPERTY(Room, Property-Name)
                Prop-Priority = PRIORITY-OF(Room-Property)
                 if (Min-Priority > Prop-Priority) then Min-Priority = Prop-Priority
return Min-Priority
```

Figure 16: Calculation of the application priority of an inference rule to a room.

When a rule is applicable to a room, the system may replace some of the room's property values with new values generated by the rule. The new values used in the replacement are called applicable effects. An effect is applicable if it satisfies the following conditions.

- This value is among the effects in the rule specification.
- The old value in the room specification is marked as UNSPECIFIED or the old value's priority is strictly smaller than the application priority.

2.7.5. APPLICATION LOOP

The "application loop" involves applying all rules to a given room or event. The system sorts the rules in decreasing order of their priority, and then applies them one by one. When it finds a rule with applicable defaults, it replaces the related values in the room or event specification with the new values, and sets the priority of these values to the application priority. For example, if it applies the "Auditorium-Mikes" rule to the Bean Auditorium, it sets the Mikes value of Bean Auditorium to be 1 with 30% probability and 9 with 70% probability, and the related priority to 2.

If the system changes some property during its pass through all rules, then it goes back to the beginning of the application loop and makes a second pass through all rules. If the second pass also changes some property, the system makes a third pass, and so on. We present the pseudo code in Figure 17.

If the total number of rules is R, and the number of room properties is P, then the system can make at most $P \cdot R + 1$ passes through the loop for applying room rules. This is because a minimum of one rule can be applied on each pass which would, at a minimum, change one room property. One extra pass through the loop is required to determine no rules are applicable which results in $P \cdot R + 1$ passes.

```
APPLICATION-LOOP()

Event-Rule-List = SORT-EVENT-RULES()
Room-Rule-List = SORT-ROOM-RULES()
for each event in all-events
        repeat
                event-changed = FALSE
                for each rule in Event-Rule-List do
                        event-before-application = event
                        event = APPLY-RULE(rule, event)
                        if (event ≠ event-before-application) event-changed = TRUE
        until(!event-changed)
for each room in all-rooms
        repeat
                room-changed = FALSE
                for each rule in Room-Rule-List
                        room-before-application = room
                        room = APPLY-RULE(rule, room)
                        if (room != room-before-application) room-changed = TRUE
        until(!room-changed)
```

Figure 17: Application loop for inference rules.

3. SEARCH FOR SCHEDULES

We now describe an algorithm for constructing a schedule based on uncertain knowledge of resources and constraints, which is also used as a subroutine of the elicitor. We explain the representation of uncertain facts, present the search for a schedule, and give empirical results on its effectiveness.

3.1. Search Algorithm

The purpose of search is to construct a schedule with a high expected quality; that is, we use the expected quality as the utility function. The system begins with the empty schedule and gradually improves it; at each step, it either assigns a slot to some unscheduled event, or moves some scheduled event to a better slot.

In Figure 18, we give the main steps of the hill-climbing search algorithm, which processes the events in decreasing order of their expected importances. When processing an event, it evaluates every assignment consistent with the event's constraints, and selects the assignment that gives the greatest utility increase. After processing all events, the algorithm returns to the beginning of the sorted list of events and repeats the processing. It stops when the last iteration through all events has not led to any improvements, or when it has reached a time limit.

```
SCHEDULER()
Sorted-List = Sort all events in order of decreasing importance
for each event in all-events
        rooms[event] = Get all acceptable rooms for event
        timeslots[event] = Get all acceptable time slots for event
sched = Create empty schedule
improvement = 0
old-utility = Minimum possible utility
repeat
        for i = 0 to COUNT(all-events)
                event = Sorted-List[i];
                for each room in rooms[event]
                        for each timeslot in timeslots[event]
                                backup-schedule = sched
                                overlapping-events = Events overlapping with event in sched
                                Remove overlapping-events from sched
                                dist-events = Get events with unacceptable distance from event in sched
                                Remove distance-events from sched
                                rel-events = Events with unaccep. start times relative to event  in sched
                                Remove relative-events from sched
                                new-utility = Calculate utility of sched
                                if (new-utility ≤ old-utility)
                                        schedule = backup-schedule
                                else
                                        improvement = new-utility – old-utility
                                        old-utility = new-utility
until (improvement == 0 or time limit is reached)
```

Figure 18: Searching for a high-quality schedule.

We next present a more detailed description of this search algorithm. We list its main variables in Figure 20, show its main procedures and calls between them in Figure 19, and give pseudo-code for these procedures in Figures 19–25. Note that the algorithm includes a mechanism for caching intermediate results of the assignment-quality computation, which allows fast evaluation of candidate assignments. This mechanism is essential for efficiency as the quality computation is the most time-consuming part of the algorithm.

We use two global variables, accessible from all procedures: the set of all conference events, denoted *all-events*, and the set of all available rooms, denoted *all-rooms*. In addition, the top-level procedure, which is called SCHEDULER (Figure 27), inputs four parameters that control the search: the beginning and end times of the conference, the discrete time step used in scheduling, and the limit on the search time. When the algorithm constructs the schedule, it only considers start times and durations divisible by the given time step. For instance, if this step is thirty minutes, then all scheduled events start and end on half hour.

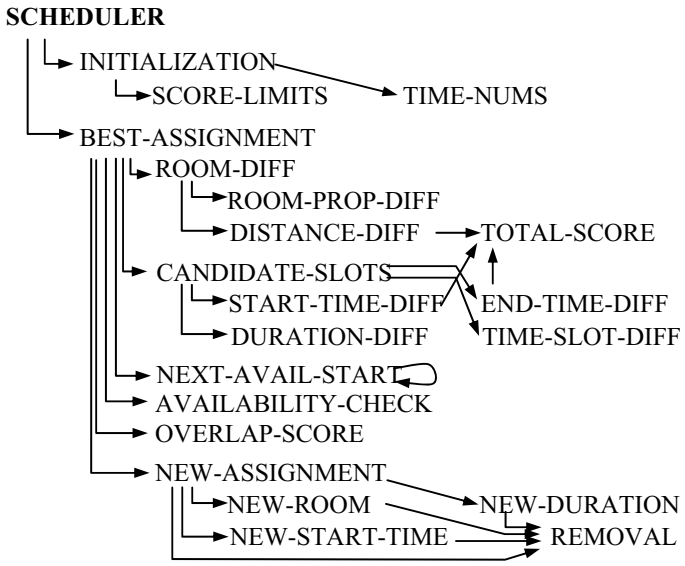


Figure 19: Main procedures of the algorithm given in Figures 19–25.

(a) Global variables
We use two global variables, accessible from all procedures:

All-Events	set of all conference events
All-Rooms	set of all available rooms

We index all events by their place in the schedule, which allows fast retrieval of the events in a given room that overlap a given time slot.

(b) Event structure
We represent a conference event by a data structure that includes its importance, constraints and preferences, place in the current schedule, and intermediate results of related computations. We use the following fields of event in the pseudo-code:

imp[*event*]	expected importance of the event
min-start[*event*]	minimal acceptable start time
max-start[*event*]	maximal acceptable start time
min-dur[*event*]	minimal acceptable duration
max-dur[*event*]	maximal acceptable duration
min-start-num[*event*]	min-start converted to discrete time steps
max-start-num[*event*]	max-start converted to discrete time steps
min-dur-num[*event*]	min-dur converted to discrete time steps
max-dur-num[*event*]	max-dur converted to discrete time steps
room[*event*]	room of the event in the current schedule
start[*event*]	current start time of the event
dur[*event*]	current duration of the event
num-prefs[*event*]	total number of the event's preferences
room-score-limit[*event*]	upper limit on the possible sum of rewards for satisfying the room-property and distance preferences
start-score-limit[*event*]	upper limit on the possible sum of rewards for satisfying the start-time preferences
dur-score-limit[*event*]	upper limit on the possible reward for satisfying the duration preference
room-score[*event*]	sum of the current rewards for satisfying the room-property and distance preferences
start-score[*event*]	sum of the rewards for the start-time preferences
dur-score[*event*]	reward for the duration preference

(c) Search parameters
We use four parameters to control the search algorithm, which are inputs of the top-level procedure, called scheduler (Figure 27):

conf-start	time of the conference beginning; events cannot start before this time
conf-end	time of the conference end; events cannot end after this time
step	discrete time step used in scheduling; all start times and durations must be divisible by it
run-time-limit	limit on the overall search time

(d) Local arrays
When the algorithm computes the quality of candidate assignments for a given event, it uses five arrays for caching intermediate results:

room-diffs	differences between the quality of new candidate rooms and that of the event's current room
start-diffs	diff. between the quality of new candidate start times and that of the event's current start time
dur-diffs	diff. between the quality of new candidate durations and that of the event's current duration
end-diffs	diff. between the quality of new candidate end times and that of the event's current end time
slot-diffs	differences between the quality of new candidate time slots and that of the event's current time slot; each candidate slot is defined by its start time and duration

Figure 20: Main variables in the procedures given in Figures 4–10

This procedure inputs an event, the beginning and end times of the conference, and the time step used in scheduling. It converts the acceptable start times and durations of the given event into the respective numbers of time steps. For example, if the conference begins at 11am, the step is 30 minutes, and the range of acceptable times is "1pm...3pm," it converts this range into "4...8."

TIME-NUMS(*event*, *conf-start*, *conf-end*, *step*)
min-start = MAX(*min-start*[*event*], *conf-start*)
min-start-num[*event*] = ⌈(*min-start* − *conf-start*) / *step*⌉
max-start = MIN(*max-start*[*event*], *conf-end* − *min-dur*[*event*])
max-start-num[*event*] = ⌊(*max-start* − *conf-start*) / *step*⌋
min-dur-num[*event*] = ⌈*min-dur*[*event*] / *step*⌉
max-dur = MIN(*max-dur*[*event*], *conf-end* − *conf-start*)
max-dur-num[*event*] = ⌊*max-dur* / *step*⌋

For a given event, the procedure determines the upper limits on the possible rewards for satisfying room related preferences, start-time preferences, and duration preferences. For instance, if an event includes five room preferences, four start-time preferences, and one duration preference, then the respective limits are 0.5, 0.4, and 0.1.

SCORE-LIMITS(*event*)
num-room = COUNT(*room-property-preferences*[*event*]) + COUNT(*distance-preferences*[*event*])
num-start = COUNT(*start-time-preferences*[*event*]) + COUNT(*relative-start-preferences*[*event*])
num-dur = COUNT(*duration-preferences*[*event*])
num-prefs[*event*] = *num-room* + *num-start* + *num-dur*
room-score-limit[*event*] = *num-room* / *num-prefs*[*event*]
start-score-limit[*event*] = *num-start* / *num-prefs*[*event*]
dur-score-limit[*event*] = *num-dur* / *num-prefs*[*event*]

The initialization procedure inputs the beginning and end times of the conference, and the time step used in scheduling. It converts the acceptable start times and durations of all events into the respective numbers of time steps, determines the upper limits on the possible rewards, creates the initial empty schedule by setting the rooms of all events to NIL, and sorts the events by importance.

INITIALIZATION(*conf-start*, *conf-end*, *step*)
for each *event* in *All-Events*
    TIME-NUMS(*event*, *conf-start*, *conf-end*, *step*); SCORE-LIMITS(*event*)
for each *event* in *All-Events*
    room[*event*] = NIL
    room-score[*event*] = 0; *start-score*[*event*] = 0; *dur-score*[*event*] = 0
for each *event* in *All-Events*
    imp[*event*] = GET-EXPECTED-IMPORTANCE(*event*)
SORT-IN-DECREASING-IMPORTANCE-ORDER(*All-Events*)

Figure 21: Initialization procedures of the scheduling algorithm.

The procedure determines the total reward score of an event.

```
TOTAL-SCORE(event)
return imp[event] · (room-score[event] + start-score[event] + dur-score[event])
```

For a given event, the procedure finds the difference between the quality of a new room and that of the event's old room.

```
ROOM-PROP-DIFF(event, new-room)
unscaled-diff = 0
all-room-property-preferences = GET-ALL-ROOM-PROPERTY-PREFERENCES(event)
for each preference in all-room-property-preferences
        if PREFERENCE-IS-UNACCEPTABLE-IN-ROOM(preference, new-room) then return NIL
        new-reward = GET-EXPECTED-REWARD(new-room, preference)
        old-reward = GET-EXPECTED-REWARD(room[event], preference)
        unscaled-diff = unscaled-diff + new-reward − old-reward
return imp[event] · unscaled-diff / num-prefs[event]
```

The procedure finds the difference between the distance rewards for placing a given event into a new room and those for its old room.

```
DISTANCE-DIFF(event, new-room)
dist-diff = 0
all-room-distance-preferences = GET-ALL-ROOM-DISTANCE-PREFERENCES(event)
for each preference in all-room-distance-preferences
        other-event = GET-OTHER-EVENT-IN-PREFERENCE(preference, event)
        if DISTANCE-PREFERENCE-IS-UNACCEPTABLE(preference, new-room, room[event])
                then dist-diff = dist-diff − TOTAL-SCORE(other-event)
        else
                new-reward = GET-EXPECTED-REWARD(preference, new-room, room[other-event])
                old-reward = GET-EXPECTED-REWARD(preference, room[event], room[other-event])
                dist-diff = dist-diff + imp[event]· (new-reward − old-reward) / num-prefs[event]
other-events = GET-EVENTS-WITH-DISTANCE-PREFERENCE-TO(event)
for each other-event in other-events
        if DISTANCE- IS-UNACCEPTABLE(room[other-event], new-room)
                then dist-diff = dist-diff − TOTAL-SCORE(other-event)
        else
                new-reward = GET-EXPECTED-DISTANCE-REWARD(room[other-event], new-room)
                old-reward = GET-EXPECTED-DISTANCE-REWARD(room[other-event], room[event])
                dist-diff = dist-diff + imp[other-event] · (new-reward − old-reward) / num-prefs[other-event]
return dist-diff
```

The procedure evaluates the reward for placing an event into a given new room. If the properties of this room are unacceptable, it returns NIL. If the room quality is so low that its use would worsen the schedule regardless of the time-slot selection, it also returns NIL. Else, it returns the difference of the room-related reward scores between this room and the event's old room.

```
ROOM-DIFF(event, new-room)
prop-diff = ROOM-PROP-DIFF(event, new-room)
if prop-diff = NIL then return NIL
dist-diff = DISTANCE-DIFF(event, new-room)
if dist-diff = NIL then return NIL
slot-diff-limit = imp[event] · (start-score-limit[event] + dur-score-limit[event]
                − start-score[event] − dur-score[event])
if prop-diff + dist-diff + slot-diff-limit ≤ 0 then return NIL
return prop-diff + dist-diff
```

Figure 22: Computing the reward-score difference between a new room and the old room of a given event.

The procedure finds the difference between the rewards related to a new start time of an event and those related to its old start time.

```
START-TIME-DIFF(event, new-start)
if IS-UNACCEPTABLE-START(new-start, event) then return NIL
new-reward = GET-EXPECTED-START-TIME-REWARD(new-start, event)
old-reward = GET-EXPECTED-START-TIME-REWARD(start[event], event)
start-diff=imp[event] · (new-reward − old-reward)/num-prefs[event]
for each relative-start-time-pref in relative-start-time-preferences[event]
        other-event = GET-OTHER-EVENT-IN-PREFERENCE(preference, event)
        if START-TIME-IS-UNACCEPTABLE(new-start, other-event)
                then start-diff = start-diff − TOTAL-SCORE(other-event)
        else
                new-reward = GET-EXPECTED-START-TIME-REWARD(new-start, time[other-event])
                old-reward = GET-EXPECTED-START-TIME-REWARD(start[event], time[other-event])
                start-diff = start-diff + imp[event] · (new-reward − old-reward) / num-prefs[event]
for each other-event in GET-EVENTS-WITH-RELATIVE-START-TIME-PREFERENCE(event)
        if IS-UNACCEPTABLE-START(new-start, event, other-event)
                then start-diff = start-diff − TOTAL-SCORE(other-event)
        else
                new-reward = GET-EXPECTED-REWARD(new-start, event, other-event)
                old-reward = GET-EXPECTED-REWARD(start[event], event, other-event)
               start-diff = start-diff + imp-other[event] · (new-reward − old-reward) / num-prefs[other-event]
return start-diff
```

The procedure finds the difference between the reward for a new duration of an event and that for its old duration.

```
DURATION-DIFF(event, new-dur)
if IS-UNACCEPTABLE-DURATION(new-dur, event) then return NIL
new-reward = GET-EXPECTED-REWARD(new-dur, event)
old-reward  = GET-EXPECTED-REWARD(dur[event], event)
return imp[event] · (new-reward − old-reward) / num-prefs[event]
```

For a given event, the procedure finds the difference between the relative-time rewards of other events with respect to its new end time and those with respect to its old end time.

```
END-TIME-DIFF(event, new-end)
old-end = start[event] + dur[event]
end-diff = 0
for each other-event in GET-EVENTS-WITH-RELATIVE-START-TIME-PREFERENCE(event, end-time[event])
        if  IS-UNACCEPTABLE-START-TIME(new-end, other-event)
                then end-diff = end-diff − TOTAL-SCORE(other-event)
                else
                        new-reward = GET-EXPECTED-REWARD(new-end, other-event)
                        old-reward = GET-EXPECTED-REWARD(old-end, other-event)
                        end-diff += imp-other[event]  · (new-reward − old-reward) / num-prefs[other-event]
return end-diff
```

The procedure inputs an event and its new place in the schedule, and computes the total reward of the events that overlap with this place.

```
OVERLAP-SCORE(event, new-room, new-start, new-dur)
score = 0
for each other-event in GET-OVERLAPPING-EVENTS(event)
        score = score + TOTAL-SCORE[other-event]
return score
```

Figure 23: Computing the reward-score differences related to the start time, duration, and end time of a given event.

The procedure inputs an event and three reward-score differences between its new candidate slot and its old slot. The first difference is for the start-time preferences, the second is for the duration preferences, and the third is for the relative-time preferences of the other events with respect to the end time of the given event. It checks if the new slot is sufficiently good. If the slot's quality is so low that its use would worsen the schedule regardless of the room selection, the procedure returns NIL; else, it returns the difference of the time-related reward scores between this new slot and the old slot.

```
TIME-SLOT-DIFF(event, start-diff, dur-diff, end-diff)
if start-diff == NIL or dur-diff == NIL or end-diff == NIL then return NIL
slot-diff = start-diff + dur-diff + end-diff
room-diff-limit = imp[event] · (room-score-limit[event] − room-score[event])
if slot-diff + room-diff-limit ≤ 0 then return NIL
return slot-diff
```

The procedure inputs an event, the beginning and end times of the conference, and the time step used in scheduling. It evaluates the quality of all potential time slots for this event; each slot is defined by its start time and duration. It returns the two-dimensional array *slot-diffs*, indexed by start times and durations; for each slot, it shows the difference between the quality of this slot and that of the event's old slot. If a time slot is unacceptable, the procedure marks it by NIL. If the slot is acceptable, but contains a smaller sub-slot with the same or higher quality, the procedure also marks it by NIL, which prevents the use of unnecessarily long slots. For example, if the 9am–11am slot is acceptable, but its 9am–10am sub-slot has the same quality, the procedure marks the 9am–11am slot by NIL.

```
CANDIDATE-SLOTS(event, conf-start, conf-end, step)
for start-num from min-start-num[event] to max-start-num[event]
        new-start = conf-start + start-num · step
        start-diffs[start-num] = START-TIME-DIFF(event, new-start)
for dur-num from min-dur-num[event] to max-dur-num[event]
        new-dur = dur-num · step
        dur-diffs[dur-num] = DURATION-DIFF(event, new-dur)
conf-end-num = ⌊(conf-end − conf-start) / step⌋
min-end-num = min-start-num[event] + min-dur-num[event]
max-end-num = MIN(max-start-num[event] + max-dur-num[event], conf-end-num)
for end-num from min-end-num to max-end-num
        new-end = conf-start + end-num · step
        end-diffs[start-num] = END-TIME-DIFF(event, new-end)
for start-num from min-start-num[event] to max-start-num[event]
        if start-diffs[start-num] ≠ NIL then best-slot-diff = NIL
        for dur-num from min-dur-num[event] to MIN(max-dur-num[event],  conf-end-num − start-
num)
                slot-diff = TIME-SLOT-DIFF(event, start-diffs[start-num], dur-diffs[dur-num],
                                end-diffs[start-num + dur-num])
                if slot-diff = NIL or (best-slot-diff ≠ NIL and best-slot-diff ≥ slot-diff)
                        then slot-diffs[start-num, dur-num] = NIL
                else best-slot-diff = slot-diff, slot-diffs[start-num, dur-num] = slot-diff
return slot-diffs
```

Figure 24: Evaluation of candidate time slots for a given event, where each slot is defined by its start time and duration.

The procedure inputs a room, the start time and duration of a time slot, represented by the respective time-step numbers, the beginning time of the conference, and the time step.

It checks if the room is available for the conference during a given time slot, and returns TRUE if it is available.

```
AVAILABILITY-CHECK(room, start-num, dur-num, conf-start, step)
start = conf-start + time-num · step
end = start + dur-num · step
for each availability-interval in GET-ALL-AVAILABLE-INTERVALS(room)
    if start[availability-interval] ≥ start and end[availability-interval] ≤ end
        return TRUE
return FALSE
```

The procedure inputs a room, the start time and duration of a time slot, represented by the respective time-step numbers, the beginning time of the conference, and the time step. If the room is available for the given time slot, the procedure returns the input start time. If not, it returns the earliest start time after the input start time that allows using the room for the specified duration. If we cannot use the room for the specified duration at any later time, it returns NIL.

```
NEXT-AVAIL-START(room, start-num, dur-num, conf-start, step)
start = conf-start + start-num · step;  end = start + dur-num · step
room-end = GET-ENDING(GET-LATEST-AVAILABLE-INTERVAL(room))
if end > room-end then return NIL
earliest-interval = GET-EARLIEST-AVAILABLE-INTERVAL- ENDING-BEFORE(room, end)
room-start = GET-START(earliest-interval)
if start ≥ room-start then return start-num
interval-start-num = ⌜(room-start − conf-start) / step⌝
return NEXT-AVAIL-START(room, interval-start-num,  dur-num, conf-start, step)
```

Figure 25: Checking the availability of a room, and identifying the earliest available time slot in a room after a given time.

The procedure removes an event from the schedule, adjusting the reward scores of the other events that have distance or start-time preferences with respect to that event. The representation of each event includes pointers to the other-event preferences affected by this event, allowing fast retrieval of the related events.

```
REMOVAL(event)
room[event] = NIL; room-score[event] = 0; start-score[event] = 0; dur-score[event] = 0
for each other-event in GET-EVENTS-WITH-DISTANCE-PREFERENCE(event)
        ADJUST-DISTANCE-REWARD(other-event)
for each other-event in GET-EVENTS-WITH-START-TIME-PREFERENCE(event)
        ADJUST-START-TIME-REWARD(other-event)
```

The procedure moves an event to a new room, removes the events whose distances to this event have become unacceptable, and re-computes the rewards for the related distance preferences.

```
NEW-ROOM(event, new-room)
room[event] = new-room
for each preference in GET-ALL-ROOM-DISTANCE-PREFERENCES(event)
        other-event = GET-OTHER-EVENT-IN-PREFERENCE(preference, event)
        if IS-UNACCEPTABLE-DISTANCE(event, other-event) then REMOVAL(other-event)
for each other-event in GET-EVENTS-WITH-RELATIVE-DISTANCE-PREFERENCE(event)
        if IS-UNACCEPTABLE-DISTANCE(other-event, event) then REMOVAL(other-event)
                else RECOMPUTE-DISTANCE-REWARD(other-event, event)
RECOMPUTE-ROOM-SCORE(event, room-score[event])
```

The procedure changes the start time of an event, and removes the other events that violate the related time constraints.

```
NEW-START-TIME(event, new-start)
start[event] = new-start
for each preference in GET-ALL-RELATIVE-START-PREFERENCES(event)
        other-event = GET-OTHER-EVENT-IN-PREFERENCE(preference, event)
        if IS-UNACCEPTABLE-START-TIME(start[event], other-event) then REMOVAL(other-event)
for each other-event in GET-EVENTS-WITH-RELATIVE-START-TIME-PREFERENCE(event, start[event])
        if IS-UNACCEPTABLE-START-TIME(start[event], other-event) then REMOVAL(other-event)
                else RECOMPUTE-START-TIME-REWARD(other-event, event)
RECOMPUTE-START-SCORE(event, start-score[event])
```

The procedure changes an event's duration, and removes the other events that violate the related time constraints.

```
NEW-DURATION(event, new-dur, old-end)
dur[event] = new-dur
RECOMPUTE-DUR-SCORE(event, dur-score[event])
if start[event] + dur[event] = old-end then return
for each other-event in GET-EVENTS-WITH-RELATIVE-START-TIME-PREFERENCE(event, end[event])
        if IS-UNACCEPTABLE-START-TIME(end[event], other-event) then REMOVAL(other-event)
                else RECOMPUTE-START-SCORE(other-event, start-score[other-event])
```

The procedure moves an event to a given new place in the schedule, removes the events that conflict with this new assignment, and re-computes the related rewards.

```
NEW-ASSIGNMENT(event, new-room, new-start, new-dur)
for each other-event in GET-OVERLAPPING-EVENTS(event)
        REMOVAL(other-event)
old-end = start[event] + dur[event]
NEW-ROOM(event, new-room)
NEW-START-TIME(event, new-start)
NEW-DURATION(event, new-dur, old-end)
```

Figure 26: Changing an event's assignment, which involves removal of the conflicting events and re-computation of the related rewards.

The procedure inputs an event, the beginning and end times of the conference, and the time step used in scheduling. It finds the best new place in the schedule for the given event, and then moves the event to this place. If the event has already been in its best place, it returns FALSE.

```
BEST-ASSIGNMENT(event, conf-start, conf-end, step)
for each room in All-Rooms
        room-diffs[room] = ROOM-DIFF(event, room)
slot-diffs = CANDIDATE-SLOTS(event, conf-start, conf-end, step)
best-asst-diff = 0
for each room in All-Rooms
        if room-diffs[room] ≠ NIL then
                start-num= NEXT-AVAIL-START(room, min-start-num[event],  min-dur-num[event],
                                        conf-start, step)
        while start-num ≠ NIL and start-num ≤ max-start-num[event]
                if start-diffs[start-num] ≠ NIL then dur-num = min-dur-num[event]
                while dur-num ≤ max-dur-num[event]
                  and AVAILABILITY-CHECK(room, start-num, dur-num, conf-start, step)
                        if slot-diffs[start-num, dur-num] ≠ NIL then
                                asst-diff = rooms-diffs[room] + slot-diffs[start-num, dur-num]
                                  − OVERLAP-SCORE(event, room, conf-start + start-num · step, dur-num · step)
                        if asst-diff > best-asst-diff then
                                best-asst-diff = asst-diff
                                best-room = room
                                best-start-num = start-num
                                best-dur-num = dur-num
                                dur-num = dur-num + 1
                        start-num = NEXT-AVAIL-START(room, start-num + 1, min-dur-num[event], conf-start,
                                        step)
if best-asst-diff = 0 then return FALSE
best-start = conf-start + best-start-num · step
best-dur = best-dur-num · step
NEW-ASSIGNMENT(event, best-room, best-start, best-dur)
return TRUE
```

The top-level scheduling procedure inputs the beginning and end times of the conference, the time step used in scheduling, and the limit on the search time. It begins with the empty schedule and searches for local improvements; at each step, it improves the assignment of one event. It stops after either reaching the time limit or iterating through all events without funding any improvements.

```
SCHEDULER(conf-start, conf-end, step, run-time-limit)
INITIALIZATION(conf-start, conf-end, step)
num-events = COUNT-EVENTS(All-Events)
num-unchanged = 0
while(not LIMIT-REACHED(run-time-limit))
        for each event in SORT-IN-DECREASING-IMPORTANCE(All-Events)
                change = BEST-ASSIGNMENT(event, conf-start, conf-end, step)
                if change then num-unchanged = 0
                        else num-unchanged = num-unchanged + 1
                if num-unchanged = num-events then return
```

Figure 27: Top-level search procedure, which reschedules one event at a time, until reaching a local maximum or hitting the time limit.

3.2. Extensions

We now outline some techniques for improving the search efficiency; we have implemented these techniques and used them in the experiments in Section 3.3.

Expected rewards: If the description of rooms and events includes uncertainty, the procedures in Figure 22 and Figure 23 compute the mathematical expectations of rewards. We have given algorithms for fast computation of expected rewards in Section 2.3.

Event indexing: We index the events by their place in the current schedule, that is, by room and time slot, which allows fast retrieval of the events that occupy a given room during a given time interval. In particular, it allows fast identification of the events that conflict with a newly scheduled event.

Constraint pointers: The representation of each event includes pointers to the distance constraints and relative-time constraints of the other events affected by this event. When the system moves an event, it uses these pointers to identify the affected events and re-computes their rewards.

Room availability: For every room, we represent its availability for the conference by a sorted list of nonoverlapping time intervals; this representation allows fast checking whether the room is available for a given time slot.

End times: The system supports constraints and preferences for the end times of events, in addition to constraints for start times, durations, and room properties. For instance, we may specify that the workshop should end after the demo and before 3pm. These constraints require a modification to the evaluation of time slots in the CANDIDATE-SLOTS procedure (Figure 24), as well as adding the re-computation of end-time rewards to REMOVAL, NEW-START-TIME, and NEW-DURATION (Figure 25).

Preference weights: The description of preferences may include their weights, which show the relative importance of each preference. For example, we may indicate that the size of a room for the workshop is twice more important than the preferred time and duration of the workshop. The system computes the reward for an assignment as the weighted sum of preference rewards; that is, if an event has k preferences, their weights are $w_1,\ldots, w_k$, and the respective rewards are $r_1,\ldots, r_k$, then the assignment quality is $(w_1 \cdot r_1 + \ldots + w_k \cdot r_k) / (w_1 + \ldots + w_k)$. The use of weights requires modifications to the computation of reward limits in SCORE-LIMITS (Figure 21), as well as to the reward computations in the ROOM-PROP-DIFF and DISTANCE-DIFF procedures in Figure 22, and the START-TIME-DIFF, DURATION-DIFF, and END-TIME-DIFF procedures in Figure 23.

Multi-day schedule: If a conference continues for several days, we specify its beginning and end times for each day, and the system marks all rooms as unavailable outside of the specified "business hours."

Initial schedule: The system can start its search from a given initial schedule rather than from the empty schedule. We use this option to repair an old schedule after changes in the availability of rooms and related resources. We also use it if the user builds a manual schedule and then applies the system to finalize it. This also provides the elicitor with a set of assignments it can use in generating questions.

The user can optionally impose a penalty on rescheduling of events, which prevents the system from making changes that would give only an insignificant improvement. This preserves the initial schedule as much as possible which in turn makes elicitation results more accurate. This is because elicitation takes into account the

schedule as it was when elicitation was ran. It also helps making the system more practical as in real life there is a cost for moving an event from an already scheduled room to another.

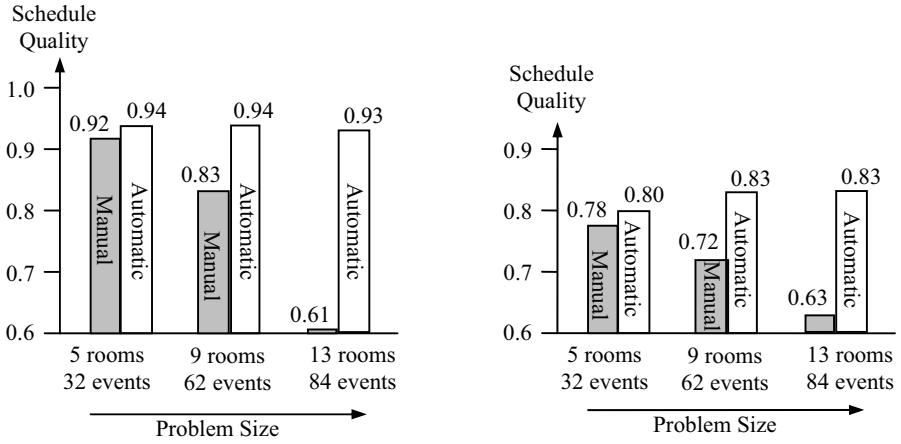
Locked assignments: The user can "lock" some events in the manually selected places, and apply the system to find assignments for the other events. This option requires a modification to the top-level SCHEDULER procedure (Figure 27); specifically, SCHEDULER should skip the locked events in its main loop, thus ensuring that they remain in their original places.

3.3. Experiments

As the optimization is not the main focus of this work we only provide evaluation results for reference. We have applied the developed system to several scheduling problems, and compared the quality of the automatically constructed schedules with the results of manual scheduling. These problems involve the scheduling of four-day conferences, with the time discretized to fifteen-minute steps. Every room has fifteen properties, and every event has between fifteen and twenty constraints and preferences.

We have used a 2.4-GHz Xeon computer, and set the time limit to ten seconds. On the other hand, we have not imposed any time limit on manual scheduling; most subjects have spent five to ten minutes on small scheduling problems, and ten to twenty minutes on large problems. In Figure 28, we summarize the results of these experiments, which show that the system has outperformed the human subjects.

We have also evaluated the dependency of the quality of automatically constructed schedules on the search time, and we show the results in Figure 29. If the knowledge is fully certain, the system constructs a schedule in about three seconds. If the knowledge is uncertain, it needs about nine seconds because it spends more time for computing the expected quality of candidate assignments.



(a) Experiments with fully certain knowledge.

(b) Experiments with uncertain knowledge.

Figure 28: Comparison of manual and automatic scheduling. We give the results for small problems (5 rooms and 32 events), medium problems (9 rooms and 62 events), and large problems (13 rooms and 84 events). We show the quality of manual schedules by grey bars, and the results of automatic scheduling by white bars.

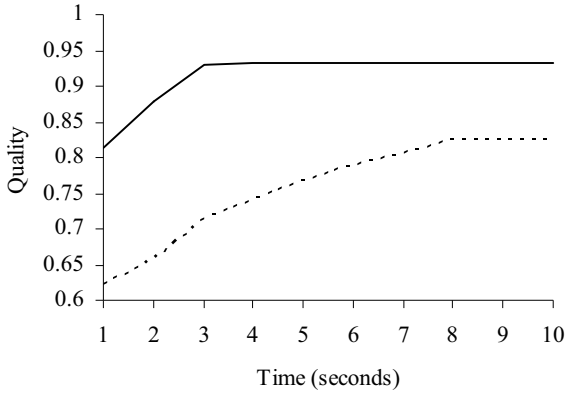


Figure 29: Dependency of the schedule quality on the running time. We show the results of scheduling with fully certain knowledge (solid line) and uncertain knowledge (dashed line); both problems include 13 rooms and 84 events.

4. ELICITATION OF ADDITIONAL DATA

4.1. Elicitation Problem

When considering candidate schedules in search for a schedule, the system computes not only the expected quality of candidate schedules, but also the standard deviation of the expected quality. If this standard deviation is high, the true schedule quality may turn out much lower than the expected quality. For example, if the properties of Wean 100 are uncertain, and the schedule is as shown in Figure 2 (page 3), then we risk placing the workshop and tutorial into a low-quality room.

The system may reduce uncertainty by asking the human administrator to provide more accurate data. For instance, it may ask to measure the size of Wean 100 and check the number of microphones in the room. We list the types of possible questions in Table 1; every question corresponds to an uncertain value, and the number of possible questions equals the number of uncertain values.

The human effort involved in providing answers may vary from question to question. For example, checking the number of microphones is easier than measuring the room size. We represent this effort by question costs; that is, we assign different costs to different questions by encoding manually, and subtract the costs of all answered questions from the final schedule quality. Costs cannot be uncertain.

The purpose of the elicitation procedure is to identify a small number of critical questions, which help to improve the schedule without incurring a high elicitation cost. There are three smaller modules which make up the overall elicitation approach which we call the unified elicitor. The **heuristic elicitor** (Section 4.2) generates an initial list of questions. The questions are then passed to the **rule-based elicitor** (Section 4.3) which may tag on more questions. The **search elicitor** (Section 4.3) takes the final list and can re-rank or remove questions.

4.2. Estimate of Question Utilities

The system estimates a question utility by the impact of the respective uncertain value on the standard deviation of the schedule quality. To determine this impact, in the heuristic elicitor it replaces all other uncertain parameters by their mathematical expectations, and then computes the standard deviation of the schedule quality. We show this computation for an uncertain room property in Figure 30, for an uncertain event importance in Figure 31, and for an uncertain range of acceptable values in Figure 32. We do not show the computation for an uncertain range of preferred values because it is similar to the computation for uncertain acceptable values in Figure 32. In search elicitor we use the optimizer in order to get a more accurate value for the impact of each question.

For instance, consider the example in Chapter II repeated in Figure 2 (page 3), and suppose that the Wean 250 has a size between 500 and 750, the importance of the demo is between 40 and 60, its minimal acceptable duration is between 60 and 90, and all other resources and preferences are fully certain, as shown in Table 2 and Table 3 Then, the impact of the size of Wean 250 on the standard deviation of the schedule quality is 0.00027, the impact of the demo importance is 0.026, and the impact of the minimal

acceptable duration of the demo is zero. Therefore, finding out about demo importance is the most important to ask about.

	Bean Auditorium	Wean100	Wean 250
Size	1200	700	500–750
Stations	10	5	5
Mikes	5	1	2

Table 8: Available rooms and their properties.

	Demo	Discussion	Tutorial	Committee	Workshop
Importance	40-60	30	75	10	50
Start time	Any	Any	11am	3pm–4pm	Any
End time	Any	Any	1pm	3pm–4pm	Any
Duration	≥60–90	≥30	≥30	≥15	≥60
Room size	≥600	≥200	≥400	≥400	≥600
Stations	≥5	Any	Any	Any	Any
Mikes	Any	≥2	≥1	Any	≥1

Table 9: Events and related constraints.

The procedure inputs the lowest and highest possible values of an uncertain room property, *low-p* and *high-p*; its minimal and maximal acceptable values, *min-ac* and *max-ac*; and its minimal and maximal preferred values, *min-pref* and *max-pref*. Note that *min-ac* ≤ *min-pref* ≤ *max-pref* ≤ *max-ac*; furthermore, for a valid schedule, *min-ac* ≤ *low-p* < *high-p* ≤ *max-ac*. The procedure returns the standard deviation of the reward for satisfying the preference for the given room property.

It uses the following local variables:

low-r, *high-r*	reward values for *low-prop* and *high-prop*
p-small, p-large	probability that the property value is below *min-pref* / above *max-pref*
exp-r	mathematical expectation of the reward value
exp-sqr-r	mathematical expectation of the squared reward value

```
LOCAL-IMPACT(low-p, high-p, min-ac, max-ac, min-pref, max-pref)
if (min-pref ≤ low-p < high-p ≤ max-pref) then return 0
low-r = REWARD(low-p, min-ac, max-ac, min-pref, max-pref)
high-r = REWARD(high-p, min-ac, max-ac, min-pref, max-pref)
if (min-ac ≤ low-p < high-p < min-pref) then return (high-r − low-r) / (2 · √3)
if (max-pref < low-p < high-p ≤ max-ac) then return (low-r − high-r) / (2 · √3)
if min-pref ≤ low-p then p-small = 0 else p-small = (min-pref − low-p) / (high-p − low-p)
if  high-p ≤ max-pref then p-large = 0 else p-large = (high-p − max-pref) / (high-p − low-p)
exp-r = p-small · (low-r + 1) / 2 + p-large · (high-r + 1) / 2 + (1 − p-small − p-large)
exp-sqr-r = p-small · (low-r^2 + low-r + 1) / 3 + p-large · (high-r^2 + high-r + 1) / 3
            + (1 − p-small − p-large)
return (exp-sqr-r − exp-r^2)^0.5
```

The procedure inputs the lowest and highest possible values of an uncertain room property, *low-prop* and *high-prop*; a list of the events scheduled in the room with this property, *events*; and the sum of the importances of all conference events, *sum-imps*. It returns the impact of the uncertain room property on the standard deviation of the schedule quality.

It uses the following local variables:

std-r	standard deviation of the related reward for one event
total-sqr-std	weighted sum of the squared standard deviations for all events

```
PROPERTY-IMPACT(low-prop, high-prop, events, sum-imps)
for each event in events
    k = COUNT(preferences[event])
    imp = importance[event]
    min-ac = GET-MINIMAL-ACCEPTABLE-VALUE-OF-PROPERTY()
    max-ac = GET-MAXIMAL-ACCEPTABLE-VALUE-OF-PROPERTY()
    min-pref = GET-MINIMAL-PREFERRED-VALUE-OF-PROPERTY()
    max-pref = GET-MAXIMAL-PREFERRED-VALUE-OF-PROPERTY()
    std-r = LOCAL-IMPACT(low-prop, high-prop, min-ac, max-ac, min-pref, max-pref)
    total-sqr-std = total-sqr-std + (imp · std-r / k)^2
return total-sqr-std^0.5 / sum-imps
```

Figure 30: Computing the impact of an uncertain room property on the standard deviation of the overall schedule quality. Note that this computation uses the REWARD procedure, given in Figure 33.

The procedure inputs an uncertain event importance, represented by its lowest and highest possible values, *low-imp* and *high-imp*; the quality of the respective event assignment, *qual*; and the sum of the importances of all conference events, *sum-imps*.

It returns the impact of the uncertain importance on the standard deviation of the schedule quality. The computation of this impact is an approximation, based on the assumption that *low-imp* and *high-imp* are significantly smaller than *sum-imps*.

IMPORTANCE-IMPACT(*low-imp*, *high-imp*, *qual*, *sum-imps*)
std-imp = (*high-imp* − *low-imp*) / (2 · $\sqrt{3}$)
return *qual* · *std-imp* / *sum-imps*

Figure 31: Impact of an uncertain importance on the schedule quality.

The procedure inputs an uncertain range of acceptable values, represented by the lowest and highest possible values of its left endpoint, *l-min-ac* and *h-min-ac*, and the lowest and highest possible values of its right endpoint, *l-max-ac* and *h-max-ac*.

It also inputs the respective range of preferred values, represented by its endpoints, *min-p* and *max-p*; the actual value of the respective property in the current schedule, *prop*; the number of preferences in the respective event, *k*; the importance of this event, *imp*; and the sum of the importances of all conference events, *sum-imps*.

Note that *l-max-ac* ≤ *min-p* ≤ *max-p* ≤ *h-max-ac*; furthermore, for a valid schedule, *h-max-ac* ≤ *prop* ≤ *l-min-ac*.

The procedure returns the impact of the acceptable-value uncertainty on the standard deviation of the schedule quality.

It uses the following local variables:

exp-r	mathematical expectation of the reward value
exp-sqr-r	mathematical expectation of the squared reward value

CONST-IMPACT(*l-min-ac*, *h-min-ac*, *l-max-ac*, *h-max-ac*, *min-p*, *max-p*, *prop, k, imp, sum-imps*)
if *min-p*≤ *prop* ≤ *max-p* **then return** 0
if *prop* < *min-p* **then**
    exp-r = 1 − ((*min-p* − *prop*) / (*h-min-ac* − *l-min-ac*)) ·
        (ln(*min-p* − *l-min-ac*) − ln(*min-pref* − *h-min-ac*))
    exp-sqr-r = 2 · *exp-r* − 1 + (*min-p* − *prop*)^2 / ((*min-p* − *l-min-ac*) · (*min-p* − *h-min-ac*))
else *exp-r* = 1 − ((*prop* − *max-pref*) / (*h-max-ac* − *l-max-ac*))·
        (ln(*h-max-ac* − *max-p*) − ln(*l-max-ac* − *max-p*))
    exp-sqr-r = 2 · *exp-r* − 1 + (*prop* − *max-p*)^2 / ((*h-max-ac* − *max-p*) · (*l-max-ac* − *max-p*))
return *imp* · ((*exp-sqr-r* − *exp-r*^{2})$^{0.5}$ / *k*) / *sum-imps*

Figure 32: Computing the impact of an uncertain range of acceptable values on the standard deviation of the schedule quality.

The procedure inputs a property value, *prop*; the minimal and maximal acceptable values for this property, *min-ac* and *max-ac*; and minimal and maximal preferred values, *min-pref* and *max-pref*. Note that *min-ac* ≤ *min-pref* ≤ *max-pref* ≤ *max-ac*. If the property is within the acceptable interval, the procedure returns the respective reward value; else, it returns NIL.

REWARD(*prop*, *min-ac*, *max-ac*, *min-pref*, *max-pref*)
if *min-ac* ≤ *prop* < *min-pref* **then return** (*prop* − *min-ac*) / (*min-pref* – *min-ac*)
if *min-pref* ≤ *prop* ≤ *max-pref* **then return** 1
if *max-pref* < *prop* ≤ *max-ac* **then return** (*max-ac* − *prop*) / (*max-ac* − *max-pref*)
return NIL

Figure 33: Computing the reward for satisfying a given preference.

The elicitation procedure estimates the utility of all candidate questions, prunes the questions whose estimated utilities are no greater than their costs, and then selects a given number of questions with the greatest difference between the utility and cost. It then uses best-first search (Figure 35) to select the most important among the remaining candidate questions.

4.3. Search for Optimal Questions

The problem we are addressing is the identification of critical uncertain information. We consider the case of a conference where we need to assign sessions to locations. Each session has a set of requirements together with their respective importance levels and each location has a set of properties. These pieces of information may be uncertain. The final goal is to create a schedule of high quality and reduce the uncertainty in the final schedule. We propose an algorithm which would create questions leading to a significant schedule improvements at a low cost. We assume that all the variables are independent, which is a simplification.

We calculate the overall utility in terms of utility of assigning each conference session to a certain location. We add session utilities, weighing each session by its importance;

$$Utility = \sum_{s \in Sessions} \left(weight(s) \cdot \sum_{p \in Pref(s)} weight(p) \cdot utility(p, s) \right). \tag{4.1}$$

The utility is a real value between predefined minimum and maximum utility values. In our system, we use -5 and 1 for these values. For each preference we calculate the utility by plugging the relevant location attribute into the relevant preference function. 1 denotes maximum possible utility and -5 denotes that the organizer of the event would be just as happy without an assignment. We also allow handling of cases where if hard preferences are violated, the whole assignment becomes unacceptable. If the happiness for *any* such preference of a session is violated, we assume that the utility for that whole

session is –6 capturing the fact that having this assignment is worse than having no assignment. The weights are scaling factors such that the overall utility itself ends up being in the range –6 and 1.

Uncertainty can exist in the system in weights, in preference functions and in location attributes. All of the uncertain variables, except for preference functions, are composed of a set of value ranges and probabilities. For each of these variables we define the mean and standard deviation to be

$$\bar{x} = \sum p_i \cdot x_i \,, \tag{4.2}$$

$$\sigma_x = \sqrt{\sum p_i \cdot \left(x_i - \bar{x}\right)^2} \,. \tag{4.3}$$

The information about the available locations, preferences of session organizers and attributes of the organizers form what we call the *world state*. We provide this world state as an input to the optimizer.

We show the types of questions the system can generate in Table 1 on page 3. Note that the system generates questions based on question classes, which means that, if a new element is added to an existing class, for example a new room attribute called "number of available chairs", the system can readily pick up that attribute and generate related questions. The system also allows more specialized questions. For example we can define a question template which depends on a specific attribute rather than a class of attributes.

The heuristic elicitor ranks each uncertain variable based on the standard deviation of overall utility due to that uncertainty. While we calculate the standard deviation due to a particular variable, we average all the other variables which may factor in. For example, in calculation for an uncertain room capacity preference if the actual room capacity of the assigned room is uncertain as well, we average the room capacity. We also weigh the standard deviation by the importance of the session and the particular preference,

$$StandardDeviation(x) = \sum_{s \in sessions} weight(s) \cdot weight(x) \cdot \sigma_{utility(s,x)} \,. \tag{4.4}$$

After the system makes the ranking, we eliminate all uncertain variables that get a score less than a preset threshold. We then apply the search elicitor. There are two versions of the search elicitor, the first version deals with all uncertain variables except preference functions, whereas the second version deals with uncertain preference functions. The search elicitor produces a weight for each uncertain variable. If this weight is zero the variable is not worth asking about.

The search elicitor ranks potential questions using a non-adversarial variant of B* search [Berliner, 1979].To evaluate each node in the search space, we use the optimizer directly giving us a more accurate ranking than estimation we use for heuristic elicitor. When evaluating a potential question, the search starts out with looking at the utility at extreme points of the complete uncertain value range. If the maximum possible utility is below a certain threshold we stop and not ask the question. If the minimum possible

utility is above a certain threshold we stop as well and record this utility value as the new question weight. If either of these conditions is not true, then we split the complete region of possible values into two equally likely regions and repeat the process for each of those regions. We can perform this search in parallel looking at the complete list of questions given to the search elicitor. In this case we would need to make sure the minimum possible utility is higher than the maximum possible utility of others. We show an example of this ranking process in Figure 34 with two potential questions.

Even though optimization is very central to the search elicitor, the search elicitor is domain-independent and it can work with any optimizer. We give the pseudo-code for both versions of the search elicitor in Figure 35, Figure 36 and Figure 37.

Sometimes less questions than the user can potentially answer get generated. In these cases, we use the rule-based elicitor to generate more questions. The rule-based elicitor infuses domain knowledge into elicitation and it uses hand coded rules to generate supplemental questions about room properties. In the conference scheduling domain, the rule-based elicitor takes into account the sessions assigned to each room with uncertain properties as well as the importance of the sessions and the requesters when ranking potential questions. We also allow the user to specify some attributes to have a bigger weight than the others. For example, in the implemented system, we weigh the attributes about a room's capacity heavier than the other attributes. Compiling rules for each domain is a relatively simple task given general knowledge about the domain. For the implemented system we spent less than a day for creating rules even though we had no previous expertise and we expect this to be the same for other domains. We give the pseudo-code for the rule-based elicitor in Figure 38.

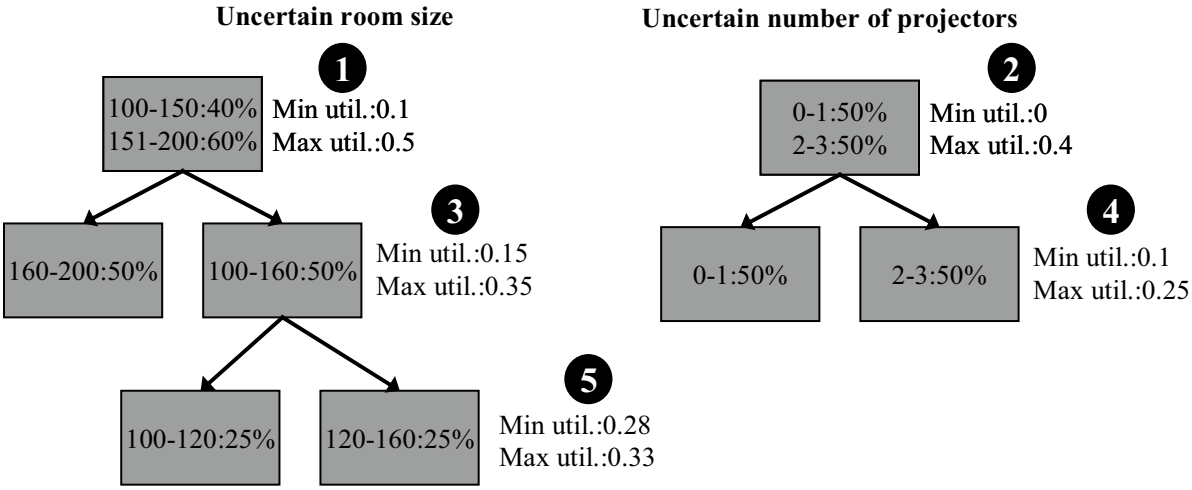


Figure 34: Example relative ranking of two uncertain variables by search elicitor. Circles denote step number and we refine the range of possible utility values at each step using the optimizer. After step 5, the minimum possible utility of asking about the size of a room becomes higher than the maximum possible utility for asking about the number of projectors. Therefore, we rank the question about uncertain room size higher than the question about uncertain number of projectors.

The procedure inputs a preference function with domain x, *prop*; the cost of asking questions, *cost*; acceptable difference between question cost and potential gain, *epsilon*; and maximum number of iterations, *maxSteps*.

```
QUESTION-EVALUATION(pref, cost, epsilon, maxSteps)
d_l = GET-UTILITY-INCREASE(x_min[pref], pref)
d_r = GET-UTILITY-INCREASE(x_max[pref], pref)
interval = CREATE-NEW-INTERVAL(x_min[pref], x_max[pref], d_l, d_r, 1)
highSum = d_high[interval]
lowSum = d_low[interval]
pq = CREATE-PRIORITY-QUEUE(index on diff)
INSERT(pq,interval)
while(true)
        if (highSum ≤ (1 + epsilon) · cost) return 0
        if (lowSum ≥ (1 - epsilon) · cost) return lowSum;
        if (size[pq] ≥ maxSteps) return smallest-positive-integer;
        interval = POP(pq)
        highSum = highSum – d_high[interval] · prob[interval]
        lowSum = lowSum – d_low[interval] · prob[interval]
        x_mid = GET-POINT-AT-MID-PROBABILITY(pref, x_l[interval], x_r[interval])
        delta_mid = GET-UTILITY-INCREASE(x_mid, pref)
        interval_l = CREATE-INTERVAL(x_l[interval], x_mid, d_l[interval], d_mid, 0.5 · prob[interval])
        interval_r = CREATE-INTERVAL(x_mid, x_r[interval], d_mid, d_l[interval], 0.5 · prob[interval])
        highSum = highSum + d_high[interval_l] · prob[interval_l] + d_high[interval_r] · prob[interval_r]
        lowSum = lowSum + d_low[interval_l] · prob[interval_l] + d_low[interval_r] · prob[interval_r]
        INSERT(pq,interval_l)
        INSERT(pq,interval_r)
```

Figure 35: Search elicitor algorithm for handling all uncertain variables except preference functions.

The procedure inputs the left and right boundaries of the interval, x_l and x_r; the change in utility on left half, $delta_l$; change in utility on right half, $delta_r$; and interval probability, *prob*. The output is an interval.

```
CREATE-NEW-INTERVAL(x_l, x_r, delta_l, delta_r, prob)
i = new interval
x_l [i] = x_l
x_r [i] = x_r
delta_l [i] = delta_l
delta_r [i] = delta_r
delta_high [i] = MAX(delta_l [i], delta_r [i])
delta_low [i] = MIN(delta_l [i], delta_r [i])
probability[i] = prob
diff[i] = (delta_high[i] – delta_low[i]) ] · probability[i]
```

Figure 36: Helper function for search elicitor algorithm.

The procedure inputs an array of preference functions with probabilities *prob*, *pref*; the cost of asking questions, *cost*; and the required difference between question cost and potential gain, *delta*. The output is a Boolean.

```
QUESTION-EVALUATION(pref[], cost, delta)
SORT-ASCENDING(pref on prob)
accumulatedGains = 0
for each prefFunction in pref
        certainWorldstate = replace(pref in worldstate with prefFunction)
        accumulatedGains = accumulatedGains +
                GET-UTILITY-INCREASE(worldstate, newWorldstate) · prob[prefFunction]
        if (accumulatedGains > cost + delta) return true
return false
```

Figure 37: Search elicitor algorithm for handling uncertain preference functions.

The procedure inputs an array of rooms, *rooms*; and a hashtable of uncertain attributes with key *room*, *attrib*. The output is a ranked list of attributes to ask questions on.

```
SUPPLEMENTAL-QUESTIONS(pref[], cost, delta)
for each room in rooms
        questionList = new question array
        roomWeight = 1
        for each event in room
                requesterWeight = GET-REQUESTER-WEIGHT(event)
                eventWeight = GET-EVENT-WEIGHT(event)
                roomWeight = roomWeight + requesterWeight · eventWeight
        for each attribute in attrib[room]
                attributeWeight = GET-ATTRIBUTE-WEIGHT(attribute)
                questionWeight = roomWeight · attributeWeight
                questionList = INSERT(attribute with questionWeight)
SORT-DESCENDING(questionList on questionWeight)
return questionList
```

Figure 38: Rule-based elicitor algorithm for choosing supplemental questions on room attributes.

4.4. Integrated Elicitor

The Integrated Elicitor combines the following three components:

- **Search Elicitor**: The search elicitor uses best-first search to identify high-utility questions. For each question, it considers possible answers, and evaluates the utility of each answer. Search elicitor takes a list of potential questions as an input which can be generated by either the heuristic or rule-based elicitors or both.

- **Heuristic Elicitor**: The heuristic elicitor evaluates the probabilistic impact of each uncertain value on the schedule score, and uses these impacts as utility estimates. This corresponds to the *first filter* mentioned in the previous section.
- **Rule-based elicitor**: The rule-based elicitor uses simple heuristics to evaluate question utilities; it is much less accurate than the other two algorithms. The rule-based elicitor provides the system with more questions when less than what can be answered at each step is generated by the other elicitors. These supplementary questions use the algorithm in Figure 38.

4.4.1. SEARCH ELICITOR

Search Elicitor inputs a set of candidate questions, and evaluates each of them separately. It first calls Optimizer to construct a schedule, and then uses this schedule for evaluating questions. This initial optimization is essential; the use of Search Elicitor with an unoptimized schedule does not give useful results.

When evaluating a question, Search Elicitor begins with rough lower and upper bounds on its utility, and then gradually narrows these bounds during its search. We use four parameters that determine when to terminate the search:

- *Per-question time*: The time limit for evaluating one question. When Elicitor reaches this limit, it terminates the evaluation of the current question.
- *Low utility*: If the utility of a question falls below this value, Elicitor rejects it; this value must be nonnegative. When the upper utility bound becomes no larger than the low-utility value, Elicitor terminates the evaluation of the current question.
- *High utility*: If the utility of a question is above this value, Elicitor considers it important; this value must be no smaller than the low importance. When the lower utility bound becomes no smaller than the high-utility value, Elicitor terminates the evaluation of the current question. We define this value empirically.
- *Utility ratio*: The upper limit on the ratio of the upper utility bound to the lower bound that represents an "accurate" estimate; this value must be strictly greater than 1.0. When the ratio of the upper bound to the lower bound becomes no larger than the utility ratio, Elicitor terminates the evaluation of the current question. We define this value empirically.

We also use a parameter that controls the invocation of Improver, the sub-module used in the "*GetUtilityIncrease*" procedure in Figure 35. *Improvement time* sets the time limit given to Improver, when Search Elicitor calls it to improve the schedule for a specific answer.

After evaluating all questions, Search Elicitor sorts them in decreasing order of importance. The sorted list of questions consists of three parts.

- *Beginning:* The beginning of the list includes all "unknown" values, in the same order as in the input of Search Elicitor. Thus, we consider "unknown" values most important to get answers for.
- *Middle:* The middle of the list includes all "important" questions, in the same order as in the input of Search Elicitor.
- *End*: The end of the list includes all other questions that have not been rejected, in the decreasing order of their lower utility bounds.

4.4.2. INTEGRATION OF ELICITORS

The Integrated Elicitor algorithm combines Search Elicitor, Heuristic Elicitor, and Rule-based elicitor (Figure 39). It uses the following parameter, in addition to the five parameters for Search Elicitor:

- *Input size for Search Elicitor*: This is the number of questions evaluated by Search Elicitor. We use it to control the trade-off between the accuracy and speed of Integrated Elicitor.

The main steps of the integrated elicitation are as follows.

1. Invoke Heuristic Elicitor, which identifies all uncertain values that affect the schedule utility, and sorts them in the decreasing order of their estimated utilities.
2. Invoke Rule-based elicitor to rank the questions that are not yet in the list, and append them to the end of the list, in the order determined by Rule-based elicitor.
3. Invoke Search Elicitor to re-evaluate the efficiency of the most important questions.

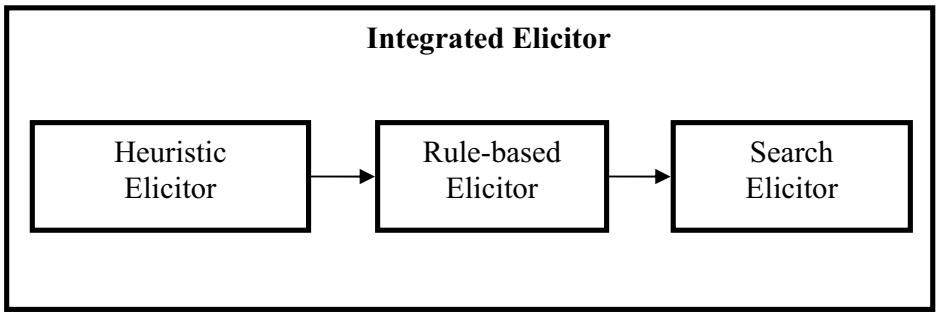


Figure 39: Steps of the Integrated Elicitor.

4.5. Evaluation

The uncertainty in a world state almost always reduces the quality of the schedule produced by the optimizer. We evaluate the improvement in the optimized schedule quality as questions are answered. An effective elicitation procedure should reduce the uncertainty in a world state, by asking a small number of questions, to a level such that the quality of the produced schedule is close to the quality from a fully certain world state.

For this main evaluation section for elicitation, we use problems from the conference scheduling domain. In Section 5.5 we provide supplemental evaluation results in the vendor orders domain to show general applicability of our approach.

4.5.1. METHODOLOGY

When evaluating the effectiveness of the generated questions, we start out by choosing a world state with uncertainty and the corresponding fully certain world state. We run the desired elicitation procedure and generate possible questions. Then, we answer a preset number of the top questions and run optimization using the new world state. We evaluate the system answering a batch of questions at each step instead of just one as this saves time. This also makes the test more realistic because in real life, users may prefer answering questions as a batch instead of just answering a single question, and re-running the system. We merge the resulting schedule with the fully certain world state and evaluate the quality of the schedule with full certainty. We show this process in Figure 40.

We generate the world states we use for this evaluation by first manually creating a fully certain world state. Then, for each problem size, we randomly pick a different number of room properties to be uncertain. We keep the number of sessions the same while we adjust the number of rooms. For smaller problem sizes, we have a smaller number of rooms. We use manually picked ranges to substitute for fully certain values. These ranges almost always contain the certain value. In order to make the experiments reproducible, we use and store a random seed in the picking of variables we make uncertain.

We keep a copy of the world state files and all the related evaluation material online. They are all contained in the zip archive at http://www.cs.cmu.edu/~eugene/Radar/Tests/ulas.zip. There are description files that describe the contents of each folder. The input files are named using a standardized naming convention. The names are in the form ***rR-rqQ-sS.stp*** where R stands for number of rooms, Q stands for number of uncertain room properties, and S stands for the random seed that we use.

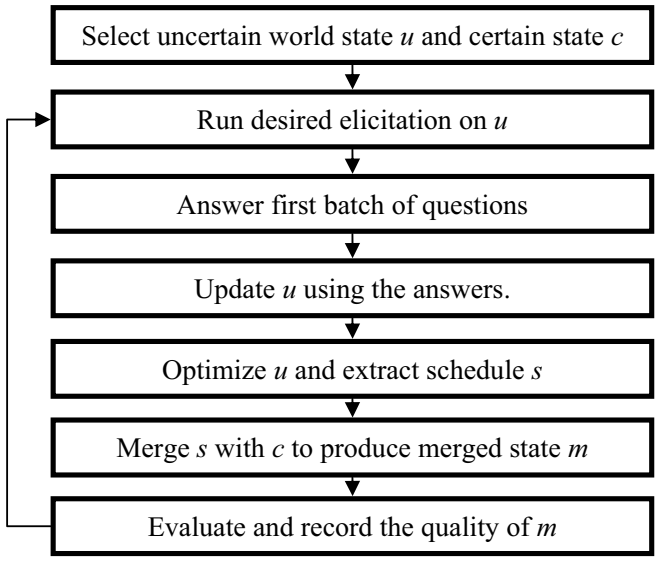


Figure 40: Evaluation procedure.

We select the uncertain world state from a pool of four different world states with different sizes. We list these world states in Table 10. For each world state, we use five different elicitation systems:

1. Heuristic and rule-based elicitors together
   We vary the number of questions we answer as a batch. We answer 10, 20, 50, or 100 questions at a time.
2. Search and rule-based elicitors together
   We vary the number of questions the search elicitor considers. It considers the 10, 20, 50, or 100 top questions. We answer 20 questions at a time.
3. Heuristic, search and rule-based elicitors together (full system)
   We vary the number of questions the search elicitor considers. It considers the 10, 20, 50, or 100 top questions. We answer 20 questions at a time.
4. Rule-based elicitor
5. Random ordering of questions
   We average 10 runs where we pick which question to answer randomly.

World state name	Number of rooms	Number of sessions	Length of conference	Number of possible questions
Largest World state	88	84	4 days	~3300
Large World state	50	84	4 days	1000
Medium World state	20	84	4 days	500
Small World state	10	84	4 days	100

Table 10: List of world states we use for evaluation.

For each scenario and for each elicitation system, we determine the change in the schedule quality as questions are answered. We plot both the actual quality and the estimated quality. The actual quality is the schedule quality we achieve when we evaluate the schedule using fully certain knowledge. The estimated schedule quality reflects the optimizer's estimate of the quality in the presence of uncertainty.

We also plot all the systems together for comparison in terms of both the schedule quality and the percentage remaining loss in schedule quality due to uncertainty as we answer each generated question:

$$RemainingLossInQuality = \frac{\left(FullyCertainQuality - CurrenQuality\right)}{\left(FullyCertainQuality - FullyUncertainQuality\right)}. \tag{4.5}$$

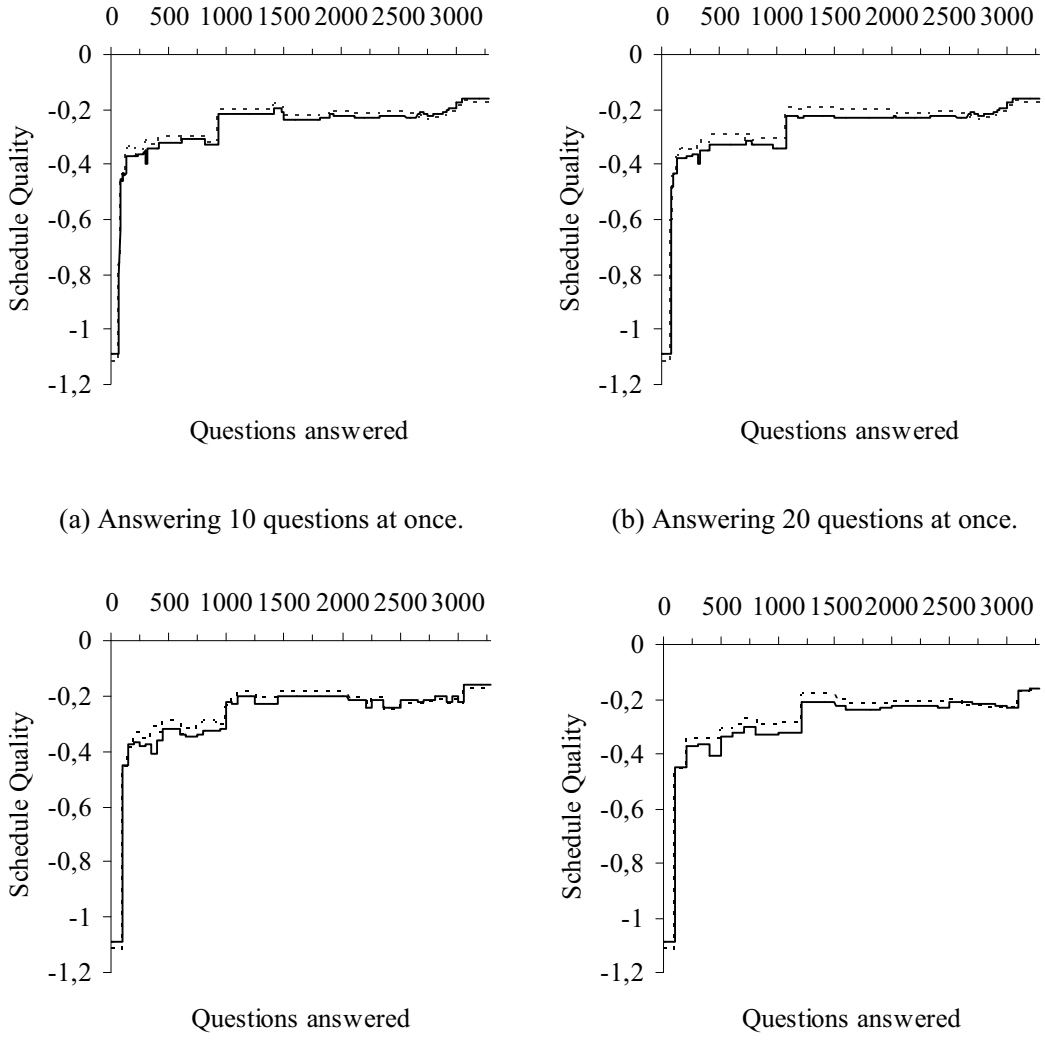
We apply a two tailed T-test in order to verify that the full elicitation system achieves a significantly better quality than each of the other systems. We use the standard formula for a two-tailed T-test

$$t = \frac{\left(\overline{X} - \mu\right)}{\left(S/\sqrt{n}\right)}. \tag{4.6}$$

We subtract the remaining loss for each of the five systems from the remaining loss for the full system after each question is answered. This gives us about as many observations as there are questions (n). Our primary hypothesis is that the mean (μ) of all observations is zero and we test at a 99% confidence level.

4.5.2. LARGEST WORLD STATE

We show the results of using the heuristic and rule-based elicitors in Figure 41. It takes the heuristic elicitor roughly 1000 questions in order to achieve a schedule quality very close to that of a fully certain schedule when we answer 20 or 50 questions at a time. This figure is slightly better when we answer 10 questions at a time and slightly worse when we answer 100 at a time. The 1000 question figure, corresponds to roughly 30% of the possible questions.



(a) Answering 10 questions at once.

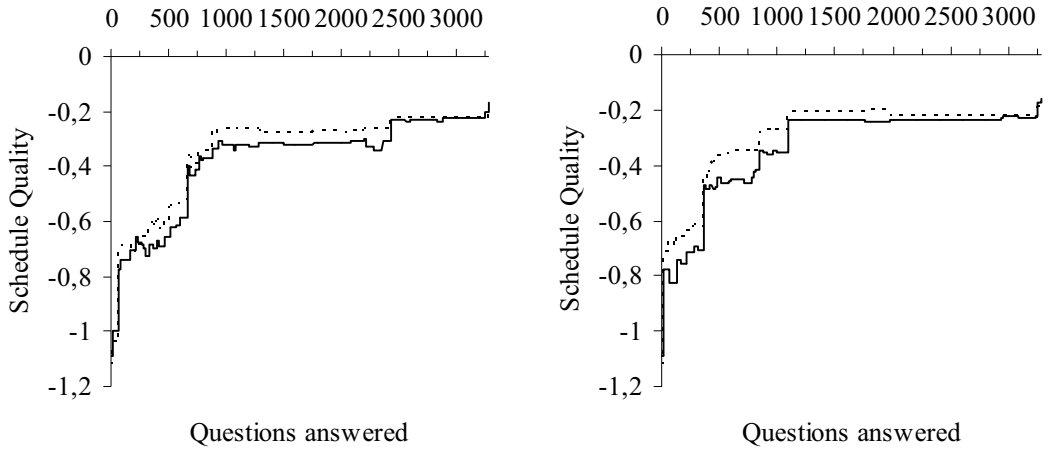
(b) Answering 20 questions at once.

(c) Answering 50 questions at once.

(d) Answering 100 questions at once.

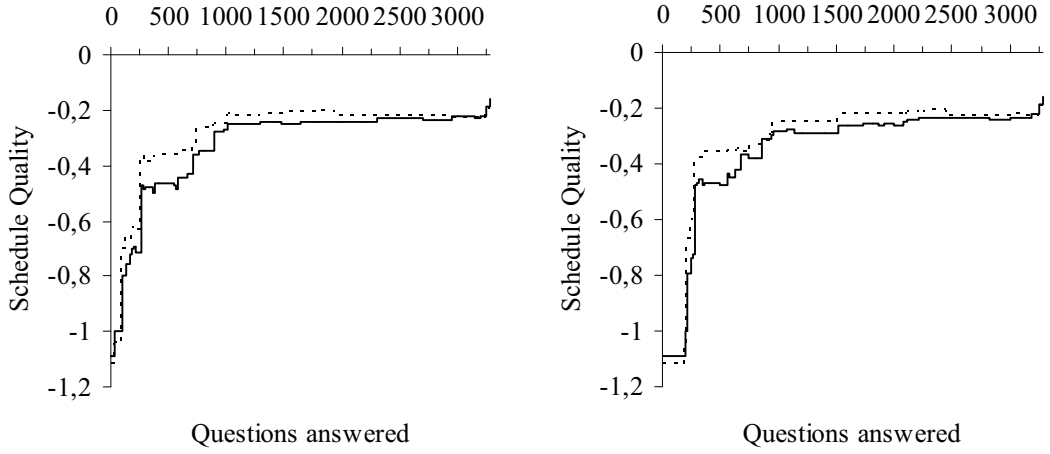
Figure 41: Dependency of the schedule quality on the number of answered questions using heuristic and rule-based elicitors on world state with 3300 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality.

We then evaluate the elicitation system with the search and rule-based elicitors. We show the results in Figure 42. As the search elicitor is meant to enhance the list of questions it is given instead of generating questions of its own, the search elicitor does not perform very well when given questions sorted based on simple heuristics used by the rule-based elicitor. Even when the search considers 100 questions at a time, the schedule quality does not reach the quality of a fully certain schedule until the end.



(a) Search considers top 10 questions.

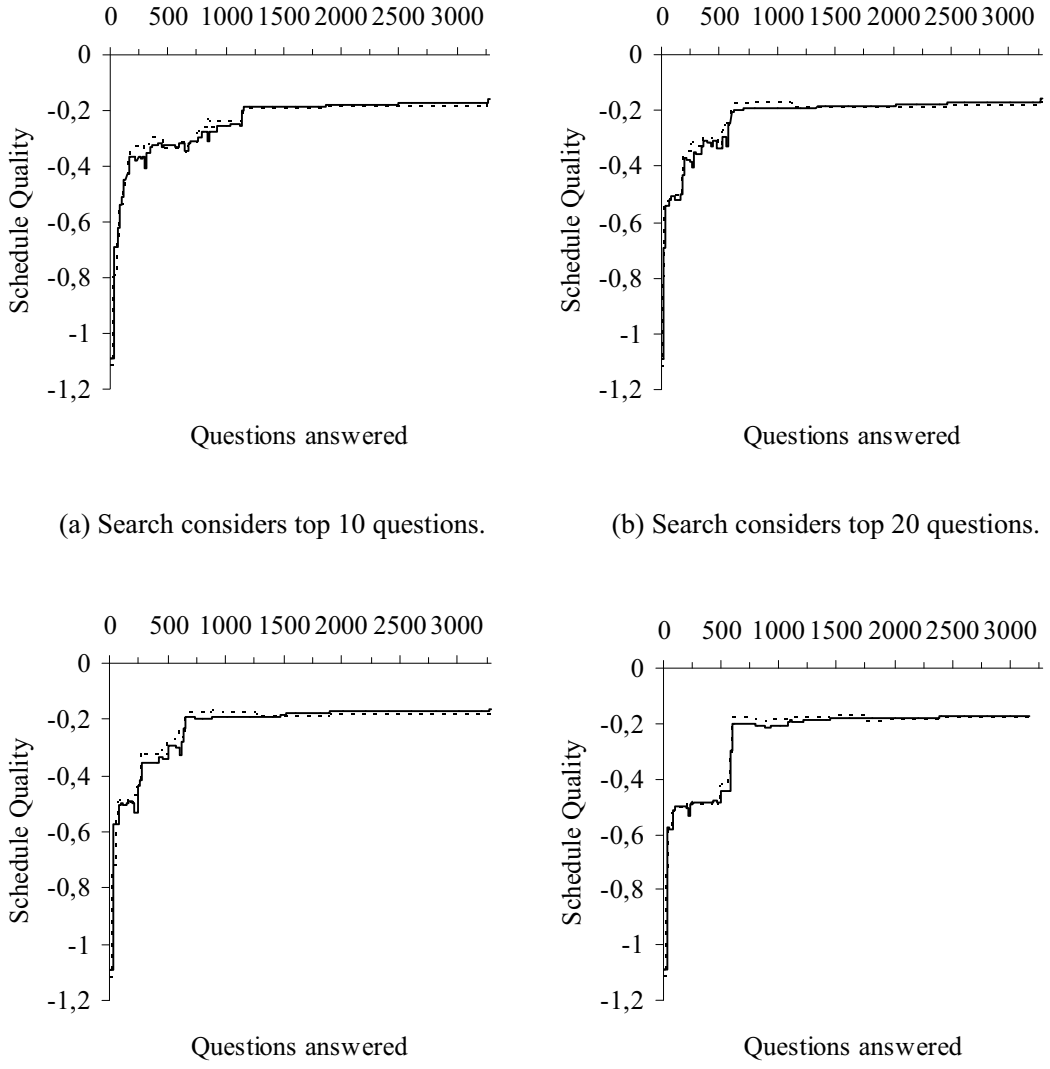
(b) Search considers top 20 questions.



(c) Search considers top 50 questions.

(d) Search considers top 100 questions.

Figure 42: Dependency of the schedule quality on the number of answered questions using search and rule-based elicitors on world state with 3300 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We rerun the elicitation after we answer each batch of 20 questions.



(a) Search considers top 10 questions.

(b) Search considers top 20 questions.

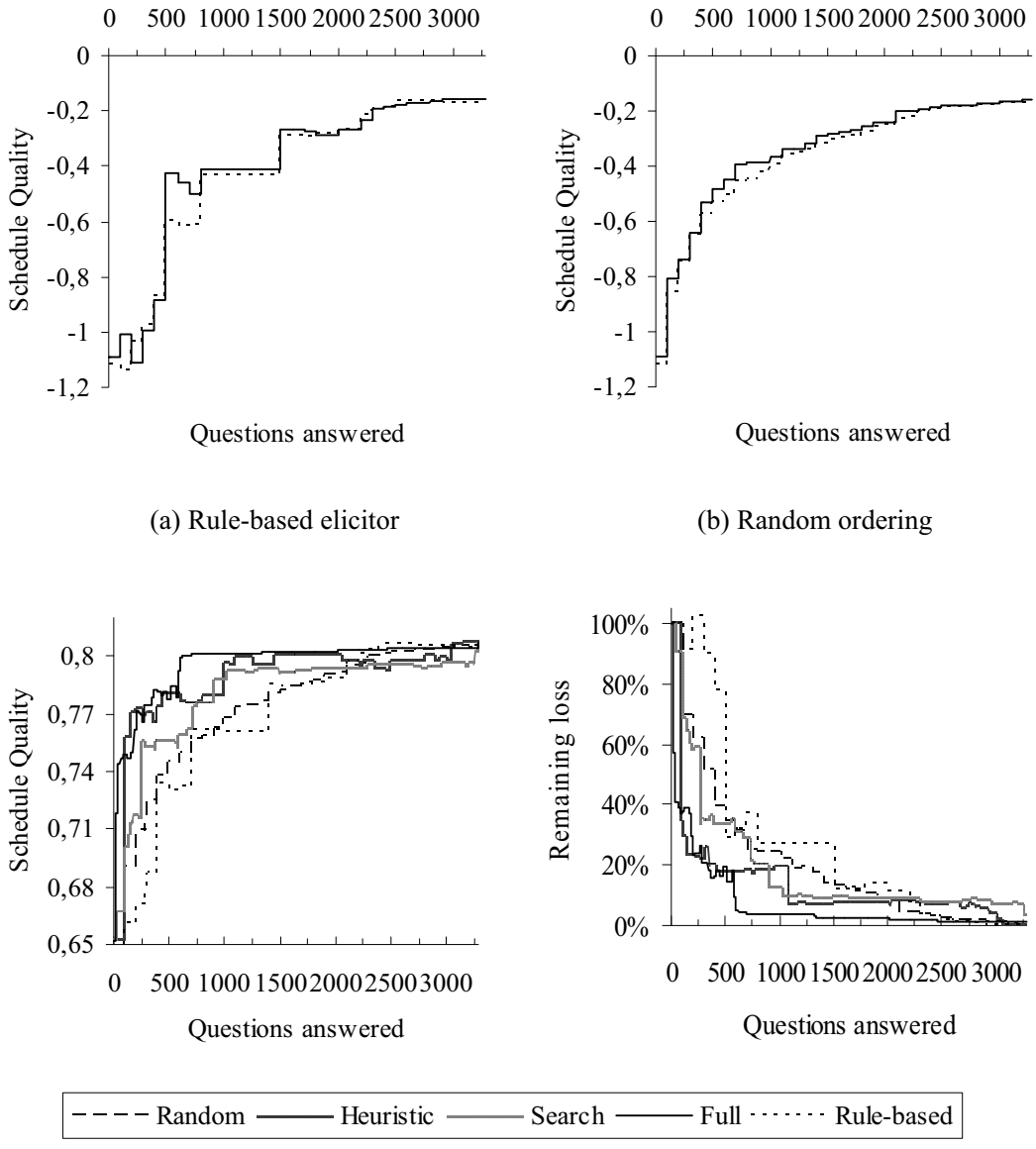
(c) Search considers top 50 questions.

(d) Search considers top 100 questions.

Figure 43: Dependency of the schedule quality on the number of answered questions using search, heuristic, and rule-based elicitors together on world state with 3300 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We rerun the elicitation after we answer each batch of 20 questions.

The search elicitor has a much better performance when used in conjunction with the heuristic and rule-based elicitors. The heuristic elicitor provides the search elicitor with a much stronger list of questions (possibly augmented by rule-based elicitor questions) than the rule-based elicitor. The search elicitor further improves this list. We show the results in Figure 43. Except for the case where the search elicitor only considers 10 questions, this elicitation method reaches the maximum score at around 500 questions out of a possible 3300. This corresponds to less than 15 percent of the possible questions.

Finally, we compare the best of each previous elicitation approach with each other, random selection of questions, and the rule-based elicitor. The rule-based elicitor performs about as well as random selection followed by the search elicitor and heuristic elicitor. Full system performs the best as we show in Figure 44.



(a) Rule-based elicitor

(b) Random ordering

(c) Comparison of all systems

(d) Remaining loss for all systems

Figure 44: Dependency of the schedule quality on the number of answered questions using rule-based elicitor and random question selection on world state with 3300 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We also compare all systems with one another.

Compared system	Observed mean ($\overline{X}$)	Observed std. dev.(S)	Test statistic (t)
Random	-0.126	0.132	-12.291
Heuristic	-0.056	0.086	-8.358
Search	-0.108	0.093	-14.851
Rule-based	-0.190	0.219	-11.142

Table 11: Two tailed T-test application comparing different elicitation systems for world state with 3300 unknowns.

Compared system	% of questions
Random	45%
Heuristic	33%
Search	26%
Rule-based	44%
Full	17.5%

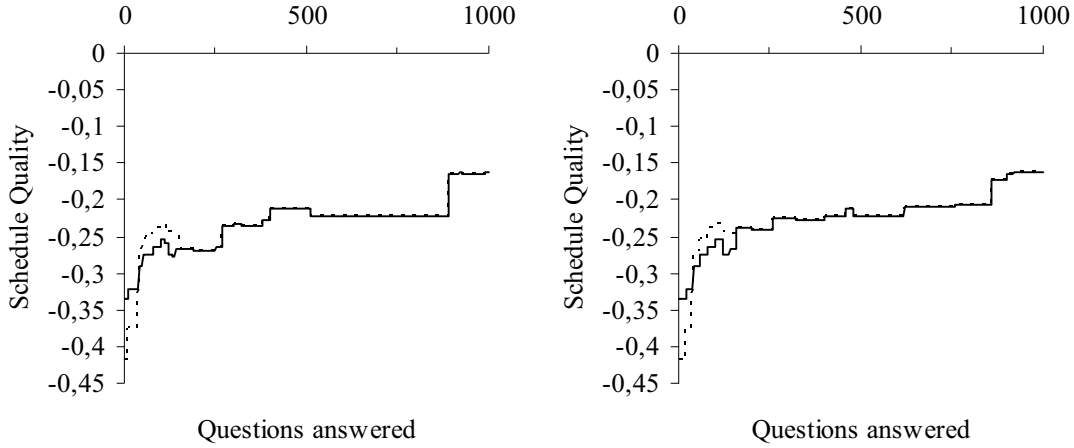
Table 12: Percentage of the generated questions that we had to answer in order to achieve 85% of the fully certain schedule quality for each system for world state with 3300 unknowns.

We show the results of our hypothesis testing in Table 11. According to the test statistic values, we reject the primary hypothesis and we accept the alternate hypothesis stating that the full elicitation system achieves a significantly different quality than each of the other systems. In particular, based on the large negative t statistic values, we can say that the full system will have a lower remaining error on average than the other systems with a 99% confidence.

We also tabulate the percentage of the generated questions that we had to answer in order to achieve 85% of the fully certain schedule quality for each system in Table 12. We see that rule-based elicitor and random picking questions perform on par with one another. The search elicitor has a slightly better performance than the heuristic elicitor while the full system requires only 17.5% of the questions to be answered before reaching 85% of the fully certain schedule quality.

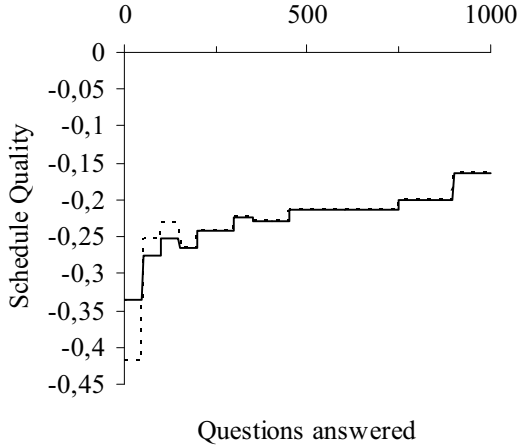
4.5.3. LARGE WORLD STATE

We show the results of using the heuristic and rule-based elicitors in Figure 45. It takes the heuristic elicitor roughly 500 questions in order to achieve a schedule quality that is within 25% of that of a fully certain schedule when we answer 20 or 50 questions at a time. The actual quality is not reached until much later. This figure is slightly better when we answer 10 questions at a time and much worse when we answer 100 at a time. The 500 question figure, corresponds to 50% of the possible questions.

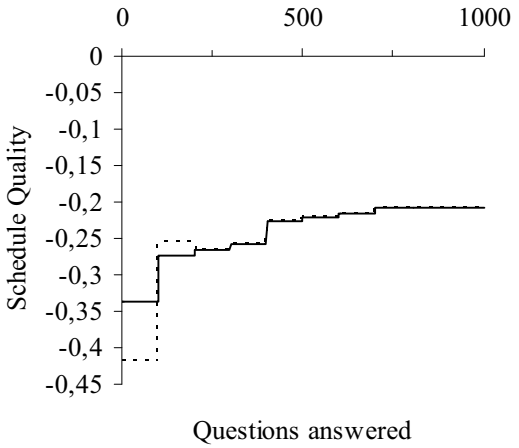


(a) Answering 10 questions at once.

(b) Answering 20 questions at once.



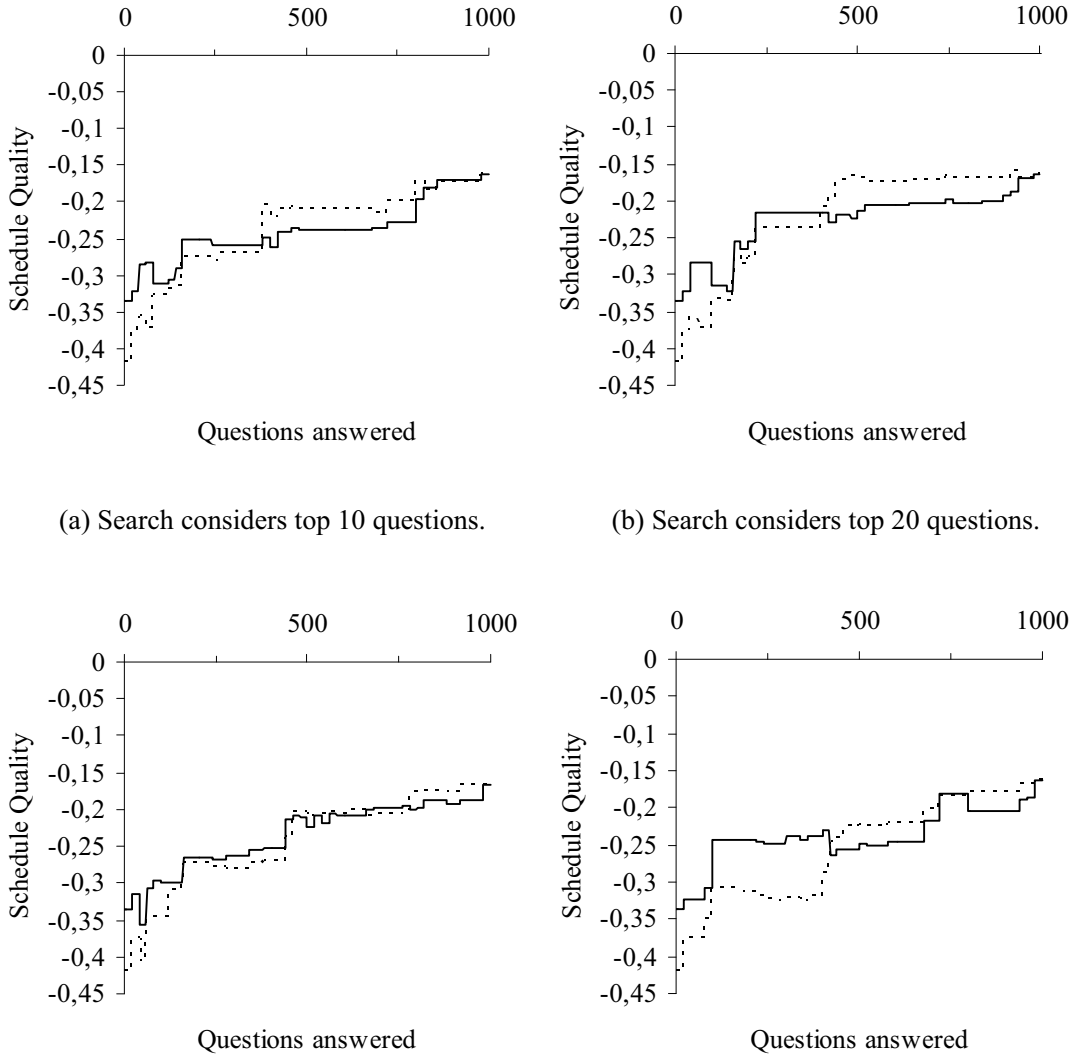
(c) Answering 50 questions at once.



(d) Answering 100 questions at once.

Figure 45: Dependency of the schedule quality on the number of answered questions using heuristic and rule-based elicitors on world state with 1000 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality.

We then evaluate the elicitation system with the search and rule-based elicitors. We show the results in Figure 46. As expected, the search elicitor does not perform very well when used just in conjunction with the rule-based elicitor. The graphs exhibit dips in quality as more questions are answered which we attribute to the optimizer's quirks in dealing with more information in the presence of resource scarcity.



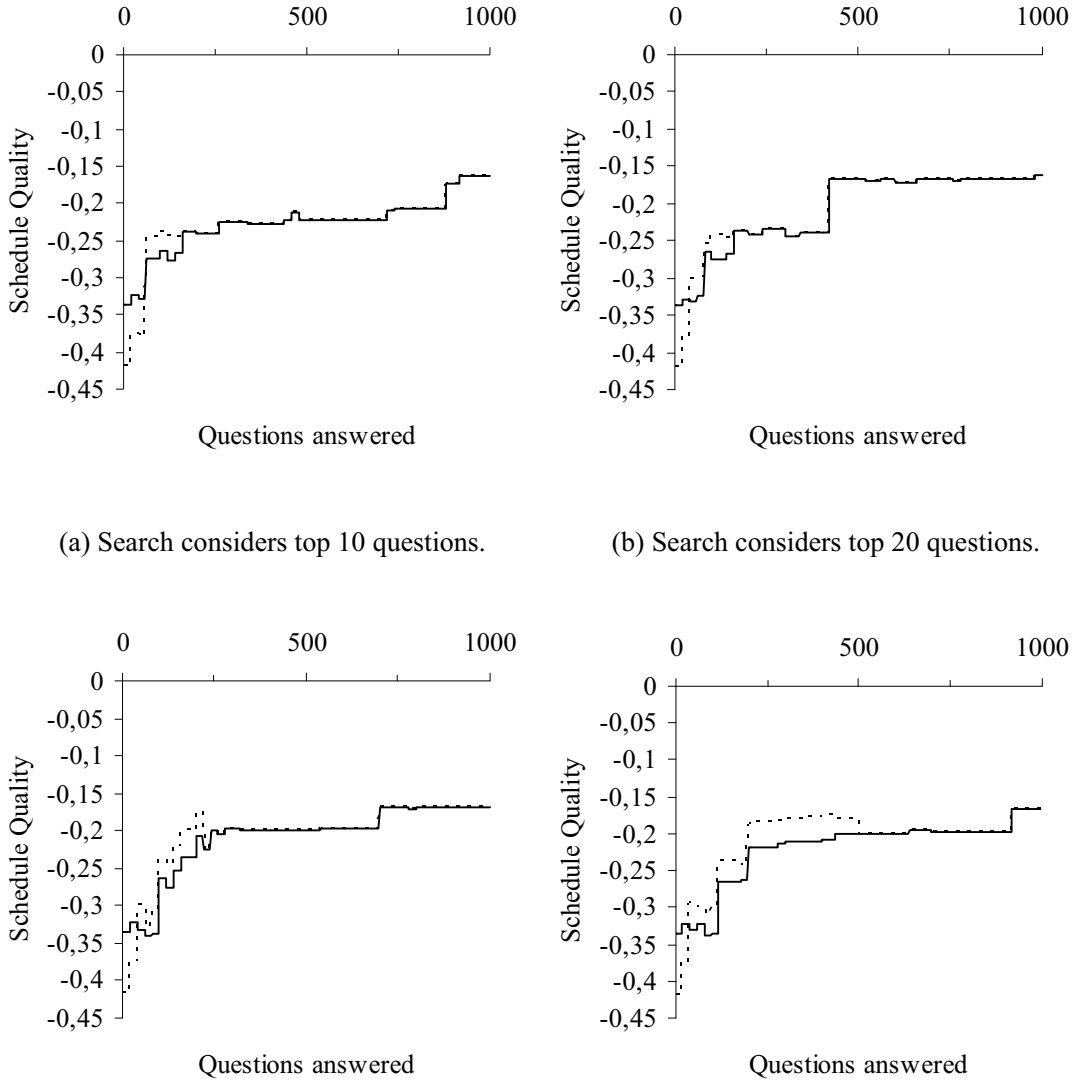
(a) Search considers top 10 questions.

(b) Search considers top 20 questions.

(c) Search considers top 50 questions.

(d) Search considers top 100 questions.

Figure 46: Dependency of the schedule quality on the number of answered questions using search and rule-based elicitors on world state with 1000 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We rerun the elicitation after we answer each batch of 20 questions.



(a) Search considers top 10 questions.

(b) Search considers top 20 questions.

(c) Search considers top 50 questions.

(d) Search considers top 100 questions.

Figure 47: Dependency of the schedule quality on the number of answered questions using search, heuristic, and rule-based elicitors together on world state with 1000 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We rerun the elicitation after we answer each batch of 20 questions.

The search elicitor again has a much better performance when used in conjunction with the heuristic and rule-based elicitors together. We show the results in Figure 46. When the search elicitor considers the top 20 questions, this elicitation method reaches the maximum score at around 400 questions out of a possible 1000. This corresponds to 40 percent of the possible questions. When the search elicitor considers 50 or a 100 questions, it comes within 25% of a fully certain schedule's quality with about 250 questions which corresponds to 25% of the possible questions.

Finally, we compare the best of each previous elicitation approach with each other, random selection of questions, and the rule-based elicitor. The rule-based elicitor performs about as well as random selection followed by the search elicitor and heuristic elicitor. Full system performs the best as we show in Figure 48.

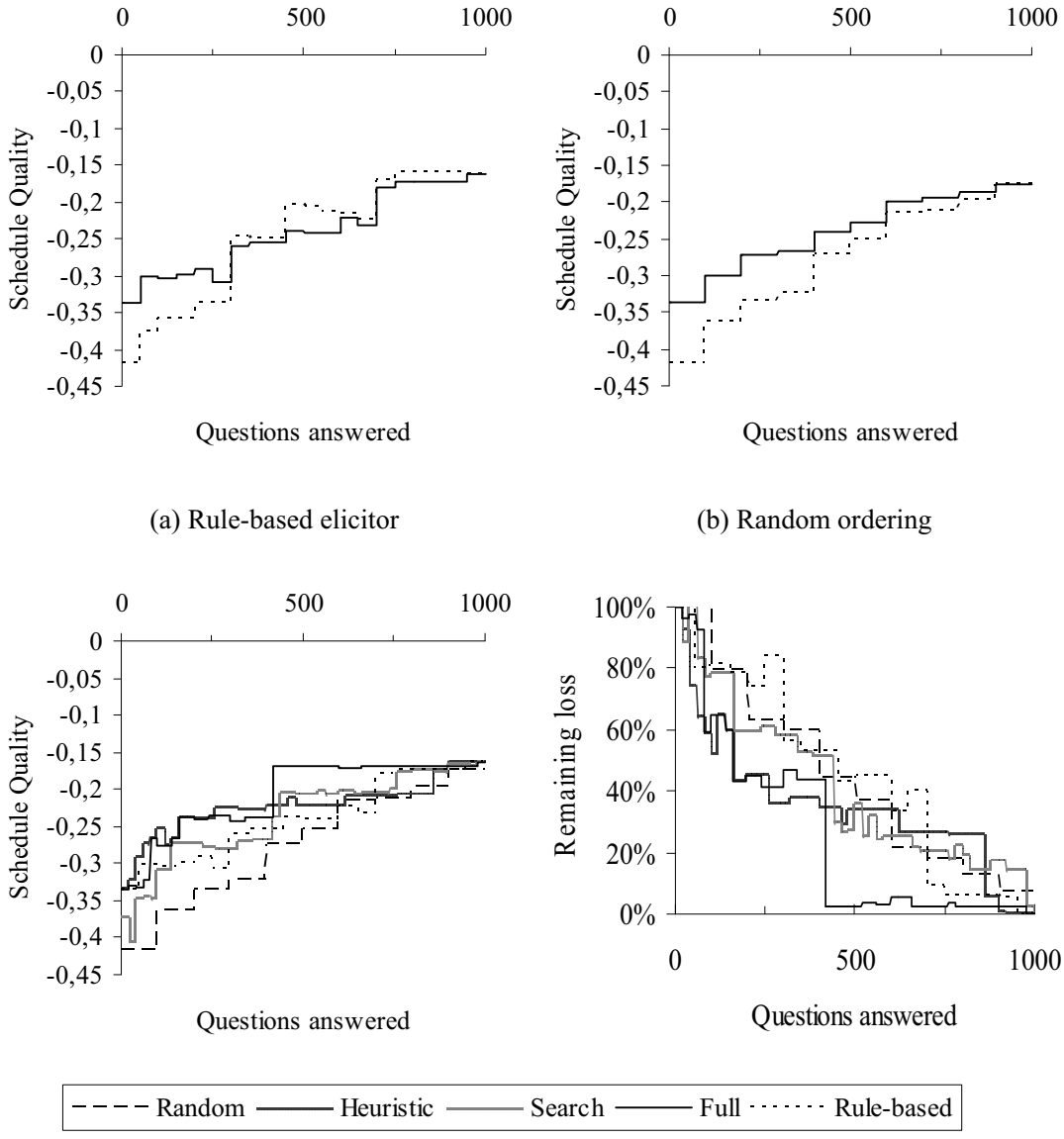


Figure 48: Dependency of the schedule quality on the number of answered questions using rule-based elicitor and random question selection on world state with 1000 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We also compare all systems with one another.

Compared System	Observed Mean ($\overline{X}$)	Observed std. dev.(S)	Test statistic (t)
Random	-0.184	0.123	-10.592
Heuristic	-0.090	0.167	-3.831
Search	-0.156	0.092	-12.023
Rule-based	-0.189	0.181	-7.372

Table 13: Two tailed T-test application comparing different elicitation systems for world state with 1000 unknowns.

Compared system	% of questions
Random	89%
Heuristic	86%
Search	83%
Rule-based	70%
Full	42%

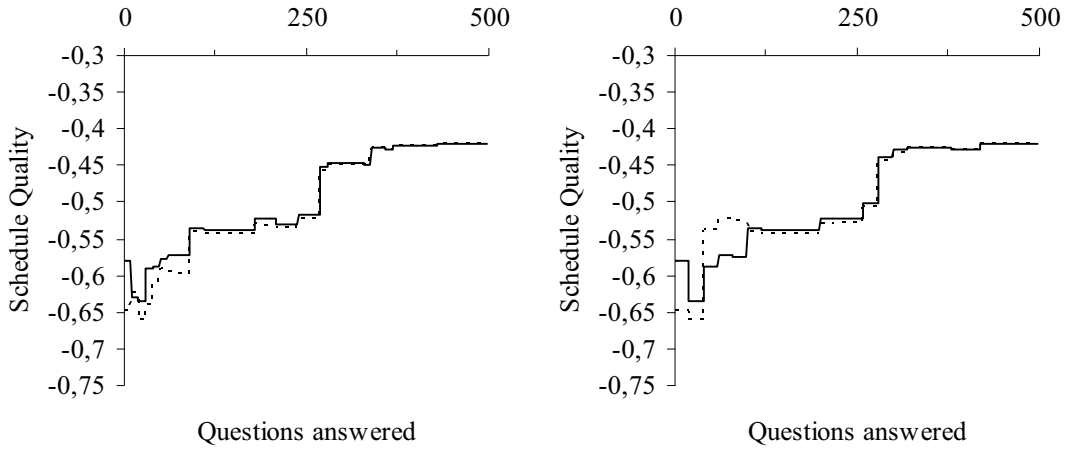
Table 14: Percentage of the generated questions that we had to answer in order to achieve 85% of the fully certain schedule quality for each system for world state with 1000 unknowns.

We show the results of our hypothesis testing in Table 13. According to the test statistic values, we reject the primary hypothesis and we accept the alternate hypothesis stating that the full elicitation system achieves a significantly different quality than each of the other systems. In particular, based on the large negative t statistic values, we can say that the full system will have a lower remaining error on average than the other systems with a 99% confidence.

We also tabulate the percentage of the generated questions that we had to answer in order to achieve 85% of the fully certain schedule quality for each system in Table 14. We see that heuristic elicitor, search elicitor and random picking questions perform on par with one another. The rule based elicitor has a slightly better performance than the others while the full system requires only 42% of the questions to be answered before reaching 85% of the fully certain schedule quality.

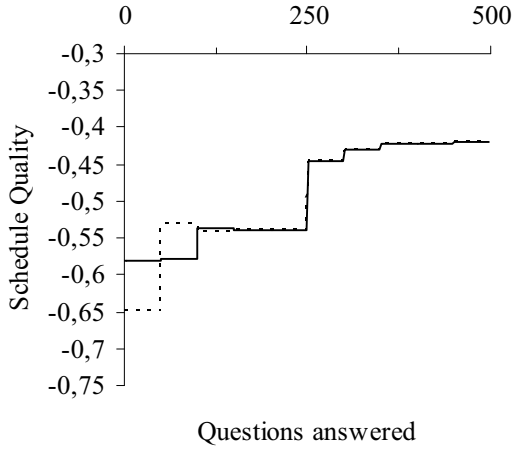
4.5.4. MEDIUM WORLD STATE

We show the results of using the heuristic and rule-based elicitors in Figure 49. It takes the heuristic elicitor roughly 250 questions in order to achieve a schedule quality very close to that of a fully certain schedule when we answer 20 or 50 questions at a time. This figure is slightly better when we answer 10 questions at a time and slightly worse when we answer 100 at a time. The 250 question figure, corresponds to roughly 50% of the possible questions.

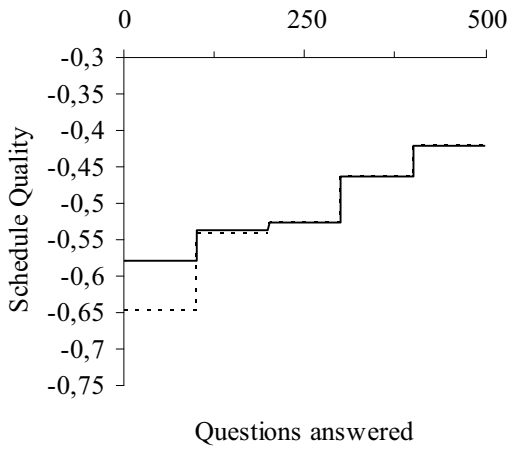


(a) Answering 10 questions at once.

(b) Answering 20 questions at once.



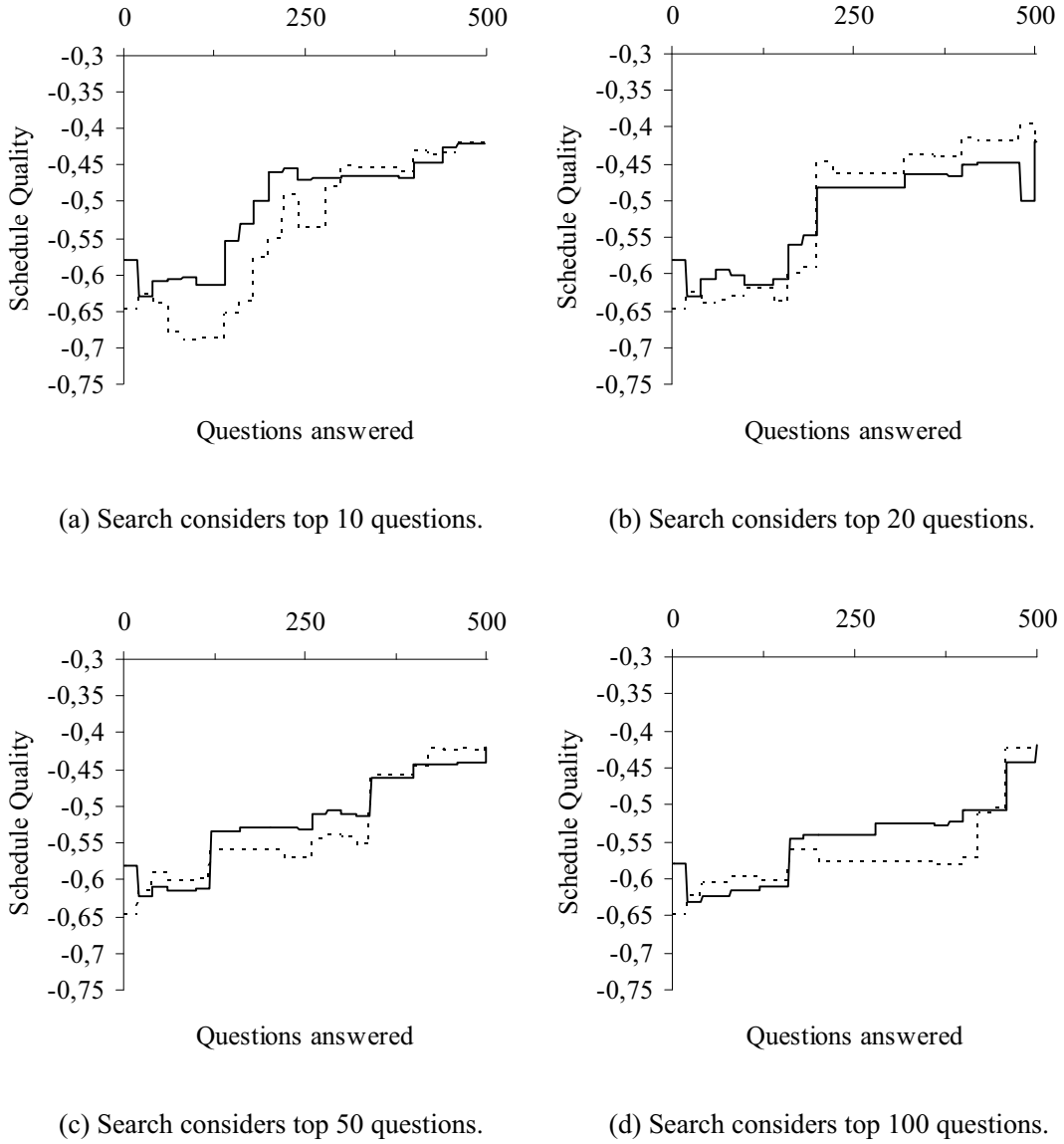
(c) Answering 50 questions at once.



(d) Answering 100 questions at once.

Figure 49: Dependency of the schedule quality on the number of answered questions using heuristic and rule-based elicitors on world state with 500 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality.

We then evaluate the elicitation system with the search and rule-based elicitors. We show the results in Figure 50. The search elicitor reaches the peak quality value for the first time around the 50% mark as well. The graphs exhibit dips in quality as more questions are answered which we attribute to the optimizer's quirks in dealing with more information in the presence of resource scarcity.



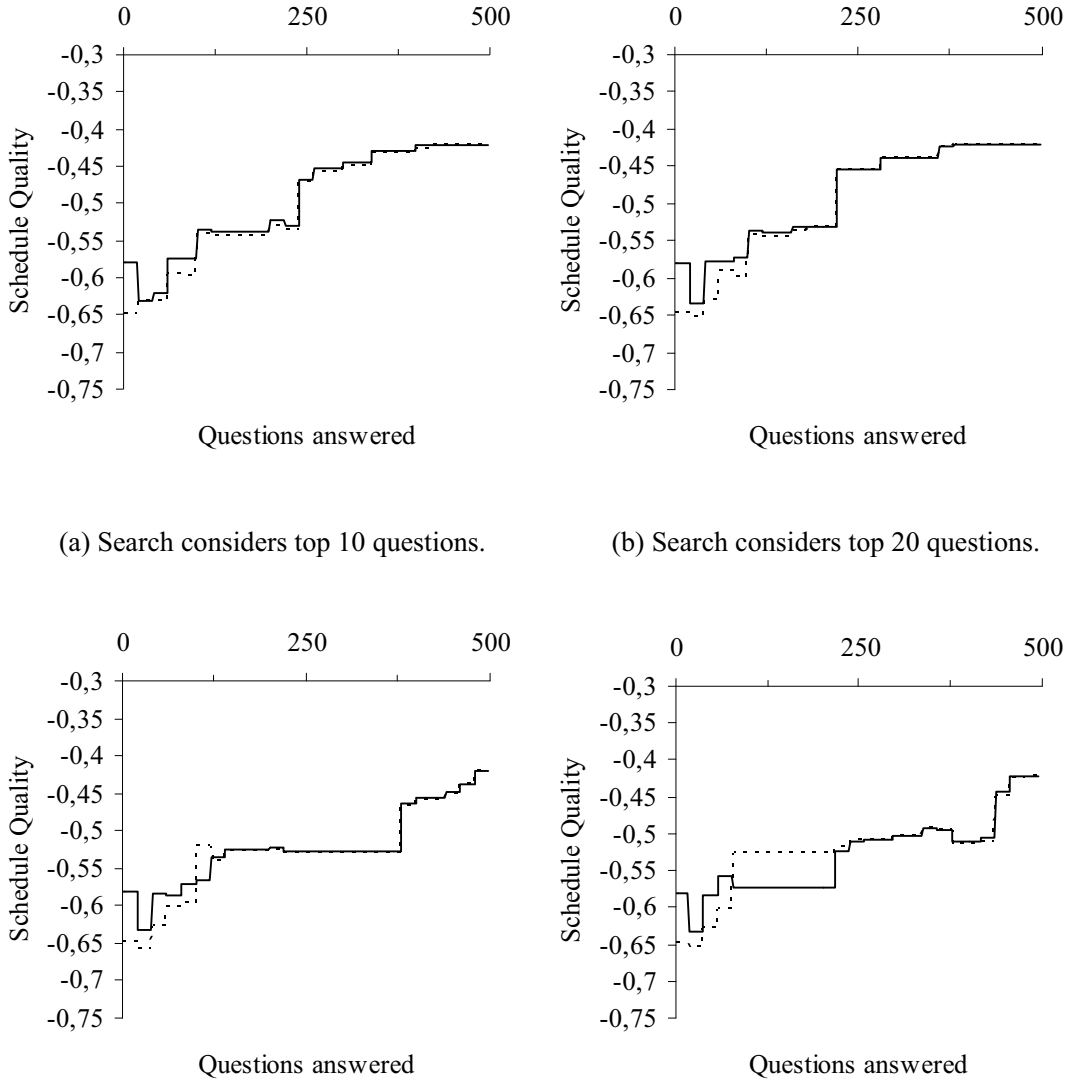
(a) Search considers top 10 questions.

(b) Search considers top 20 questions.

(c) Search considers top 50 questions.

(d) Search considers top 100 questions.

Figure 50: Dependency of the schedule quality on the number of answered questions using search and rule-based elicitors on world state with 500 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We rerun the elicitation after we answer each batch of 20 questions.



(a) Search considers top 10 questions.

(b) Search considers top 20 questions.

(c) Search considers top 50 questions.

(d) Search considers top 100 questions.

Figure 51: Dependency of the schedule quality on the number of answered questions using search, heuristic, and rule-based elicitors together on world state with 500 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We rerun the elicitation after we answer each batch of 20 questions.

With the number of rooms diminishing and number of sessions staying the same, the search elicitor does not have a much better performance when used in conjunction with the heuristic and rule-based elicitors together. We show the results in Figure 51. When the search elicitor considers the top 10, 20, or 100 questions, this elicitation method reaches the maximum score at around 250 questions out of a possible 500. This corresponds to 50 percent of the possible questions.

Finally, we compare the best of each previous elicitation approach with each other, random selection of questions, and the rule-based elicitor. The rule-based elicitor performs about as well as the full system followed by the heuristic elicitor and search elicitors. Random system performs the worst as we show in Figure 52.

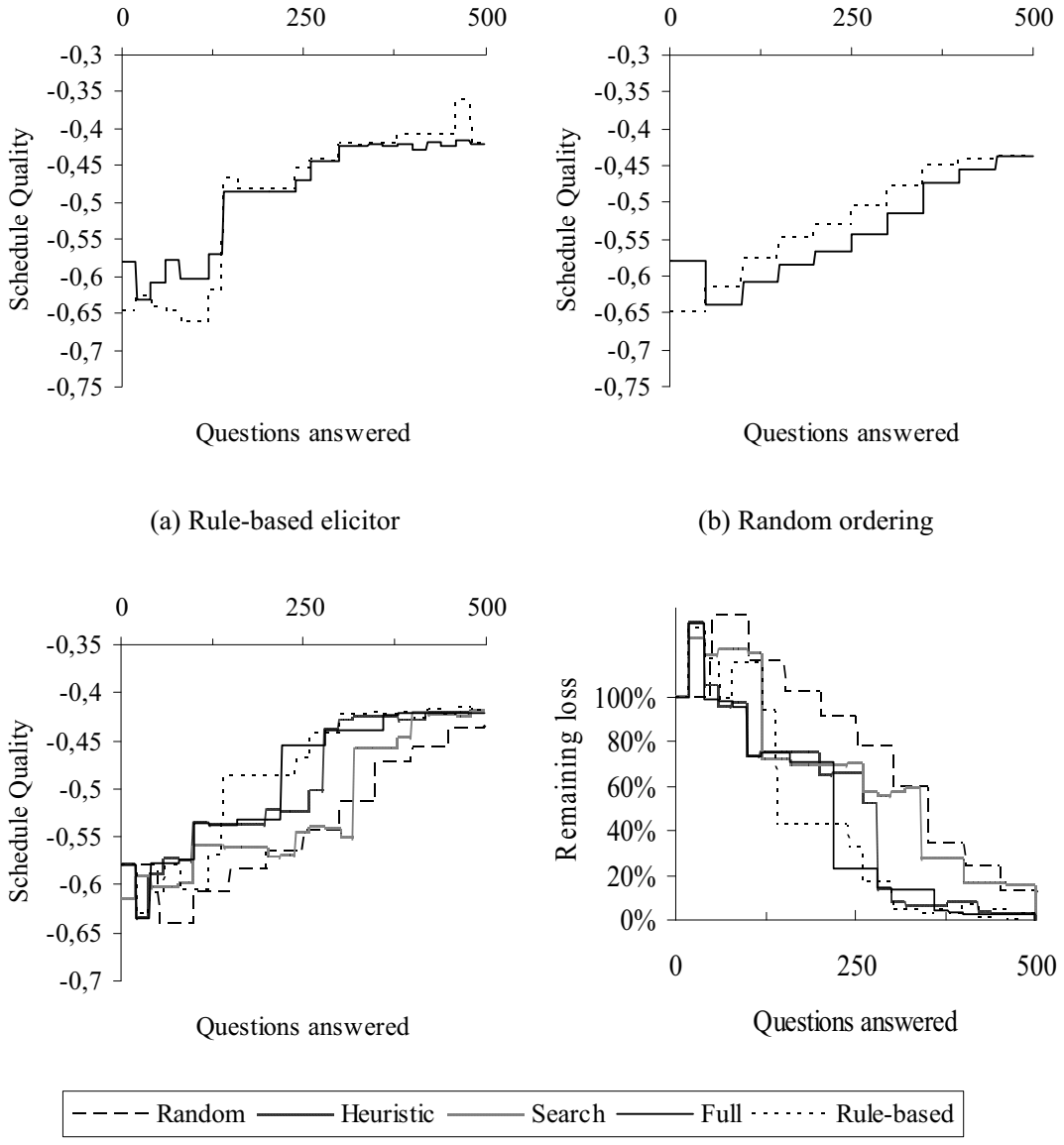


Figure 52: Dependency of the schedule quality on the number of answered questions using rule-based elicitor and random question selection on world state with 500 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We also compare all systems with one another.

Compared System	Observed Mean ($\overline{X}$)	Observed std. dev.(S)	Test statistic (t)
Random	-0.237	0.165	-7.085
Heuristic	-0.033	0.102	-1.640
Search	-0.146	0.137	-5.322
Rule-based	0.009	0.131	0.330

Table 15: Two tailed T-test application comparing different elicitation systems for world state with 500 unknowns.

Compared system	% of questions
Random	90%
Heuristic	57%
Search	81%
Rule-based	59.5%
Full	56%

Table 16: Percentage of the generated questions that we had to answer in order to achieve 85% of the fully certain schedule quality for each system for world state with 500 unknowns.

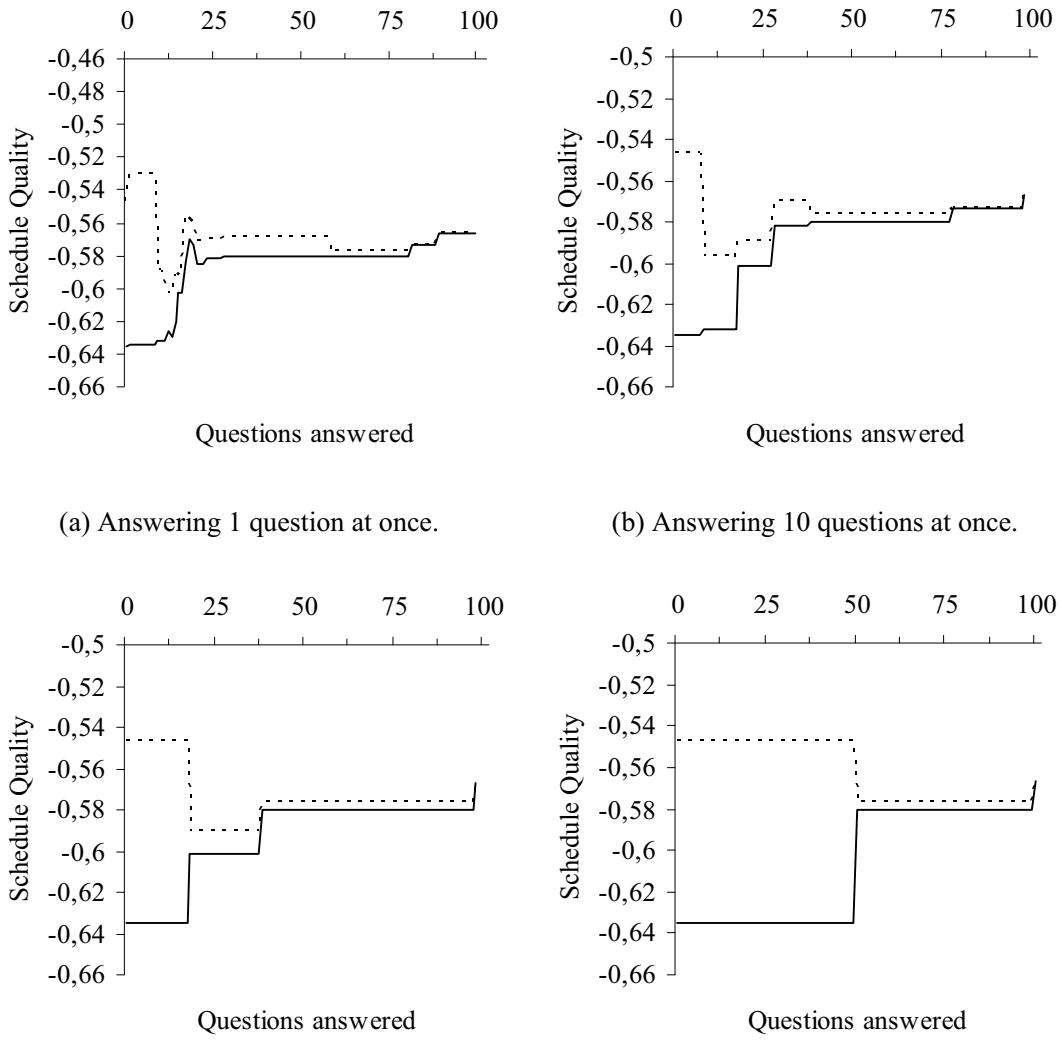
We show the results of our hypothesis testing in Table 15. According to the test statistic values, we reject the primary hypothesis for random, and search elicitors and we accept the alternate hypothesis stating that the full elicitation system achieves a significantly different quality than each of the other systems. In particular, based on the large negative t statistic values, we can say that the full system will have a lower remaining error on average than the other systems with a 99% confidence.

The rule-based and heuristic elicitors however, have test statistics within the critical range which means we accept the primary hypothesis stating those elicitors, which are based on simple heuristics and simple standard deviation, do as well on this world state as the complete system. We believe that the primary reason this happens is because of the limited options the optimizer has as a result of resource scarcity. Furthermore, as fewer variables are present, simple heuristics can do well and the heuristics used in the rule-based elicitor seem to be a good match for solving this particular problem.

We also tabulate the percentage of the generated questions that we had to answer in order to achieve 85% of the fully certain schedule quality for each system in Table 16. We see that search elicitor and random picking questions perform on par with one another. The heuristic and rule based elicitors exhibit similar performance to the full system however, the full system still requires the least number of questions reaching 85% of the fully certain schedule quality with 56% of the questions answered.

4.5.5. SMALL WORLD STATE

We show the results of using the heuristic and rule-based elicitors in Figure 53. It takes the heuristic elicitor roughly 38 questions in order to achieve a schedule quality very close to that of a fully certain schedule when we answer 20 or 50 questions at a time. This figure is around 22 when we answer 1 or 10 questions at a time. The 22 question figure, corresponds to 22% of the possible questions.



(a) Answering 1 question at once.

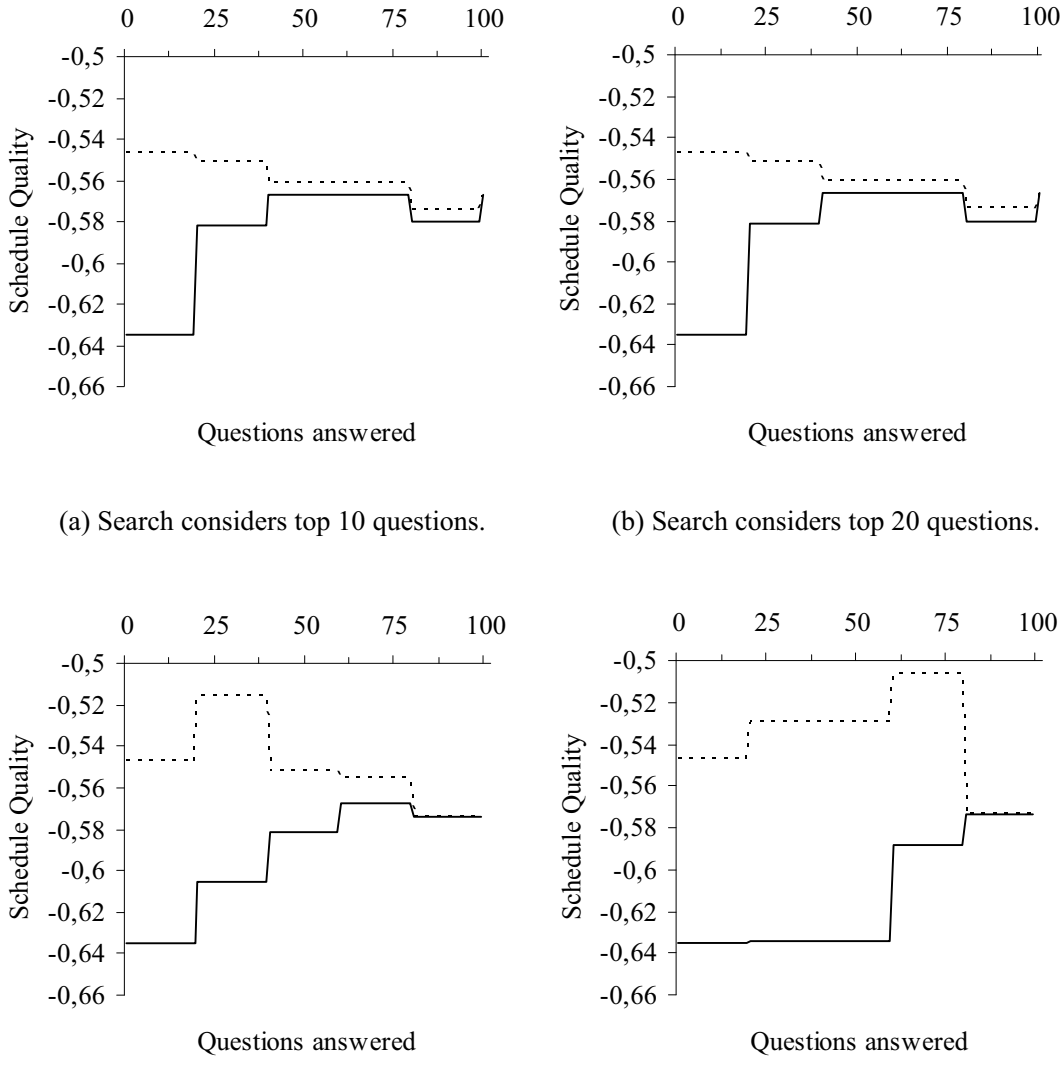
(b) Answering 10 questions at once.

(c) Answering 20 questions at once.

(d) Answering 50 questions at once.

Figure 53: Dependency of the schedule quality on the number of answered questions using heuristic and rule-based elicitors on world state with 100 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality.

We then evaluate the elicitation system with the search and rule-based elicitors. We show the results in Figure 54. The search elicitor reaches the peak quality value for the first time around the 38% mark. The graphs exhibit dips in quality as more questions are answered which we attribute to the optimizer's quirks in dealing with more information in the presence of resource scarcity.



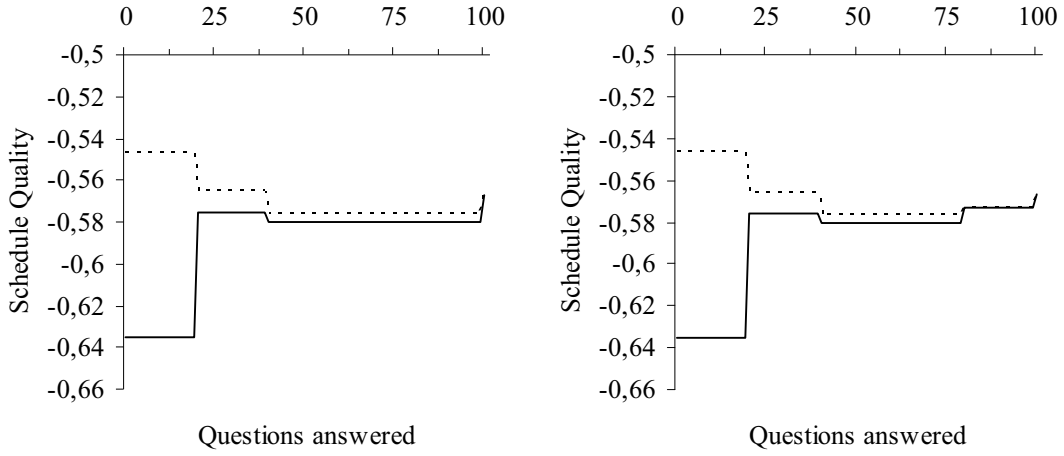
(a) Search considers top 10 questions.

(b) Search considers top 20 questions.

(c) Search considers top 50 questions.

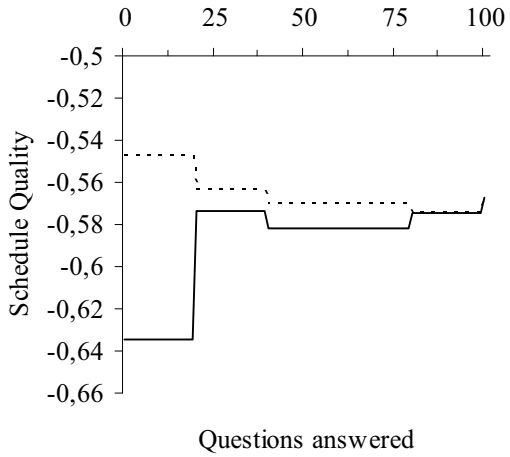
(d) Search considers top 100 questions.

Figure 54: Dependency of the schedule quality on the number of answered questions using search and rule-based elicitors on world state with 100 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We rerun the elicitation after we answer each batch of 20 questions.

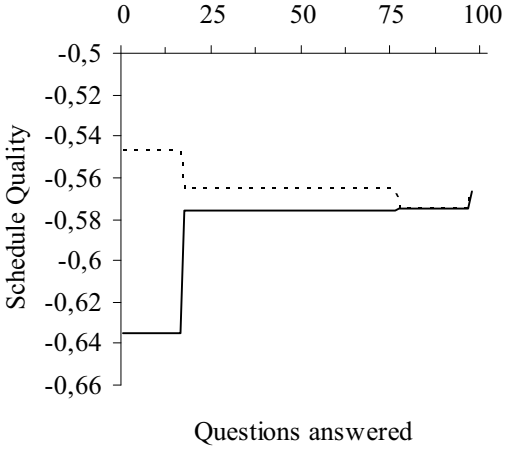


(a) Search considers top 10 questions.

(b) Search considers top 20 questions.



(c) Search considers top 50 questions.



(d) Search considers top 100 questions.

Figure 55: Dependency of the schedule quality on the number of answered questions using search, heuristic, and rule-based elicitors together on world state with 100 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We rerun the elicitation after we answer each batch of 20 questions.

With the number of rooms diminishing even further and number of sessions staying the same, the search elicitor does not have a much better performance when used in conjunction with the heuristic and rule-based elicitors together. We show the results in Figure 55. The unified elicitor reaches the maximum score at around 25 questions out of a possible 100. This corresponds to 25 percent of the possible questions.

Finally, we compare the best of each previous elicitation approach with each other, random selection of questions, and the rule-based elicitor. All systems except for random picking of questions perform very close to one another. Full system performs slightly better and the random system performs the worst as we show in Figure 56.

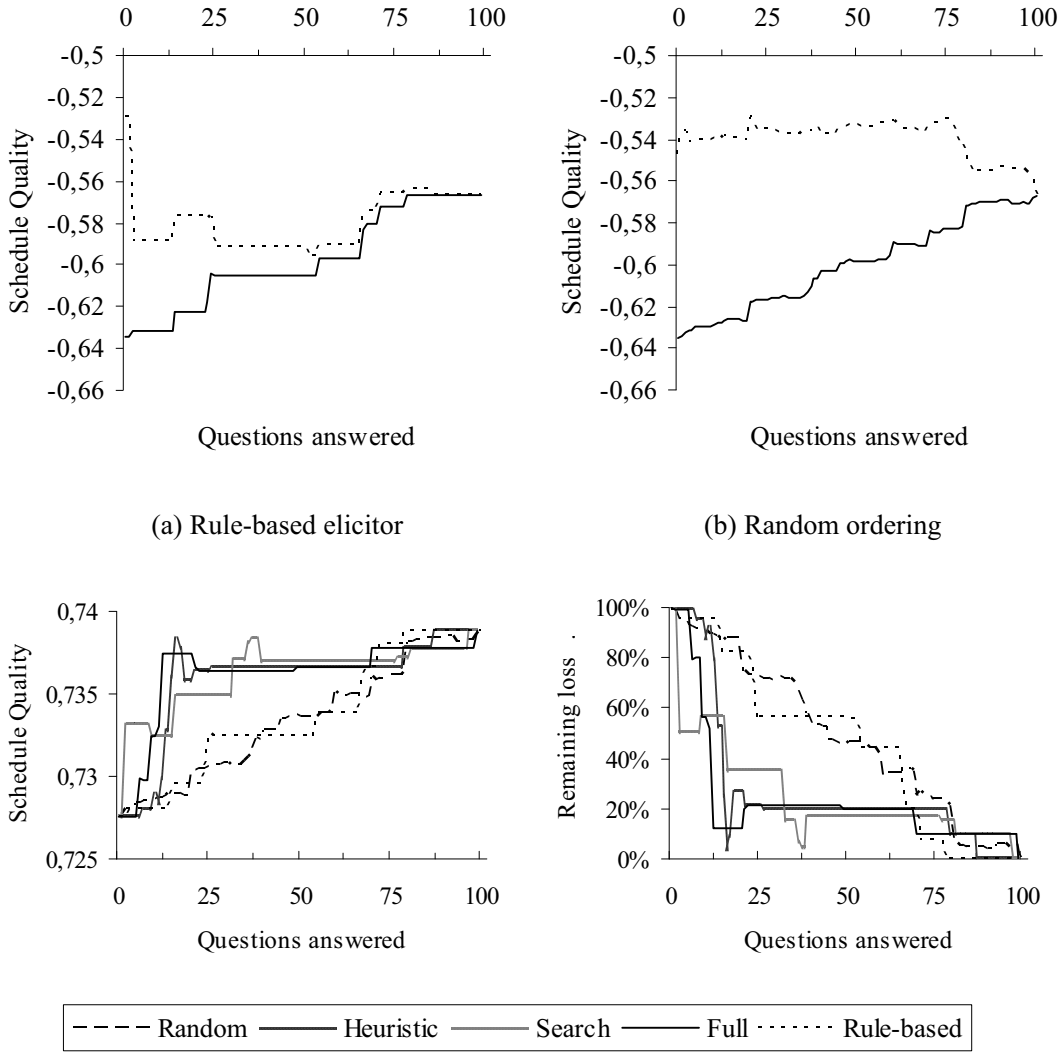


Figure 56: Dependency of the schedule quality on the number of answered questions using rule-based elicitor and random question selection on world state with 100 unknowns. Dashed lines show the schedule quality as estimated, whereas solid lines show the actual schedule quality. We also compare all systems with one another.

Compared system	Observed Mean ($\overline{X}$)	Observed std. dev.(S)	Test statistic (t)
Random	-0.323	0.835	-3.868
Heuristic	0.136	1.177	1.153
Search	-0.004	0.162	-0.239
Rule-based	-0.208	0.256	-8.156

Table 17: Two tailed T-test application comparing different elicitation systems for world state with 100 unknowns.

Compared system	% of questions
Random	80%
Heuristic	16%
Search	35%
Rule-based	70.5%
Full	13%

Table 18: Percentage of the generated questions that we had to answer in order to achieve 85% of the fully certain schedule quality for each system for world state with 100 unknowns.

We show the results of our hypothesis testing in Table 17. According to the test statistic values, we reject the primary hypothesis for only the random elicitor, accepting the alternate hypothesis stating that the full elicitation system achieves a significantly different quality than the random system. In particular, based on the large negative t statistic values, we can say that the full system will have a lower remaining error on average than the random system and the rule-based elicitor with a 99% confidence.

The remaining two systems however, has a test statistic well within the acceptable limit which means we accept the primary hypothesis stating those systems do as well on this world state as the complete system. We believe that the primary reason this happens is because of the limited options the optimizer has as a result of resource scarcity. The difference between the quality of a fully certain world state and the starting uncertain one is very little.

We also tabulate the percentage of the generated questions that we had to answer in order to achieve 85% of the fully certain schedule quality for each system in Table 18. We see that rule-based elicitor and random picking questions perform on par with one another. The search elicitor requires 35% of the questions to be answered while the heuristic elicitor exhibit similar performance to the full system. However, the full system still requires the least number of questions reaching 85% of the fully certain schedule quality with only 13% of the questions answered.

Considering all of the world states we have used for evaluation, we can see that the unified system outperforms the others considerably when the world state is large and the optimizer has a lot of choices. As the optimizer's possible choices become smaller and smaller, the unified system's performance is matched by the other simpler systems excluding the random picking of questions.

5. VENDOR ELICITATION

We explore applying our approach to information elicitation to the vendor order problem in order to show the generalizability of our approach. We treat this domain as a secondary domain and therefore we do not evaluate the results as exhaustively as we do for the conference domain.

5.1. Vendor Order Problem

In a typical conference scenario, sessions can require services that external vendors provide such as mobile equipment not found in rooms or food deliveries. We use a vendor optimizer in order to find a placement of vendor orders given a conference schedule and needs of individual sessions. However, similar to the case with scheduling rooms, uncertainty can exist making the optimizer perform worse.

Uncertainty can exist in different parts of vendor information. The system may not have a complete list of vendors, we may not know all the items a vendor provides, and we may not know the prices of all the items available from all vendors.

We need an elicitation method which would make use of user preferences, any partial knowledge about vendors as well as any penalties related to spending money.

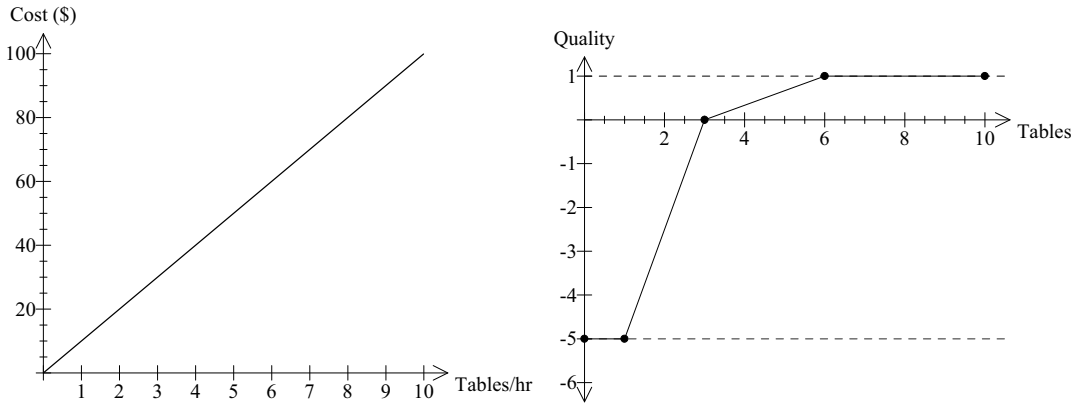
5.2. Representation of Partial Knowledge

The basic objects we represent in the system related to vendor orders are *services*, *resource functions, resource items*, *vendors*, and the *cost penalty function*.

Services are broad categories representing a specific session need. For example, *floral*, *tables* and *meals* are services. A session can request one or more of these services. We show a table of sample services and their explanations in Table 19.

Service	Explanation
Floral	Flower arrangements
Table	Tables needed for reception desks
Meal	Any sort of food ordered for sessions
Security	Security personnel for big sessions
PC Laptop	Laptop personal computers
Apple Laptop	Apple based laptops
Computer	Any computer

Table 19: A subset of the services we represent in the system.



(a) Cost function. (b) Utility function.

Figure 57: Resource functions for a session that requires tables.

Each session has a set of *resource functions*. Each resource function represents a service the session requires. A resource function is made up of two functions: A cost function which shows the dependency of the total order cost on the number of ordered items, and a utility function which shows the dependency of the increase in the overall quality increase on the number of ordered items. We represent both functions as piecewise-linear functions. If multiple vendors offer items that can provide the same service at different prices, the cost function for this service would reflect the cheapest alternative. We show an example resource function in Figure 57 for a session that requires tables. The cost function is completely linear and each table costs $10 to rent per hour. The utility function states that the session requires at least three tables and six tables and higher would get the highest quality.

Resource items are actual items that provide one or more services. For example, a *rose arrangement* would provide a *floral* service and *vegetarian meal* would provide a *meal* service. A *Dell laptop* would satisfy both the *laptop pc* and *computer* services. Items may also have different cost types. For example, a vendor may charge per item (e.g. a florist) or per item per hour (e.g. a vendor renting laptops). We assume that the list of resource items is complete, that is the system knows about all the possible items and respective services before it starts. However, this does not mean that we know about all the items a given vendor provides.

Each *vendor* in the system has a list of resource items and respective prices. The price can be a numerical value, or a marker stating that the vendor does not offer this item or that we do not know whether or not the vendor offers this item (and therefore the price for the item). We make the following assumptions about items a vendor offers:

- If an item type is available from a vendor, the vendor can provide an unlimited supply of such items.
- The unit price does not depend on the order size, which means that vendors do not give discounts or charge premiums for large orders.
- We allow unknown costs, but we do not allow uncertain costs specified by probability distributions.

The system also has a flag which we use for marking whether or not the list of vendors is complete. An incomplete list would mean that there are vendors the system does not yet know about.

There is a cost associated with each vendor order and if we place too many orders we end up spending too much money. Therefore, we have a penalty in the system which captures this concept and makes sure the system takes into account costs of placing orders. The cost penalty function shows the dependence of this penalty on the amount of money spent on vendor orders. The cost function is a piecewise-linear function that represents the dependency between the amount of all expenses and the penalty for spending this money. It must be fully certain and monotonically increasing, although it may not be strictly increasing. The penalty is in the same range as the schedule quality and is subtracted from it. We show an example cost penalty function in Figure 58.

5.3. Search for Optimal Orders

The module we use for searching for optimal vendor orders to place is called *resource optimizer*. The resource optimizer starts with a list of resource functions and outputs a list of suggested orders. It ultimately lets the human user decide which suggestions to follow.

The resource optimizer repeatedly selects the best order to place and determines whether that order improves the overall schedule score. The algorithm works in two passes to account for the possibility of running into local maxima. First it evaluates all possible orders and determines the maximum attainable score. Then it starts from the beginning again, keeping track the placed orders, and stops when it reaches the predetermined score.

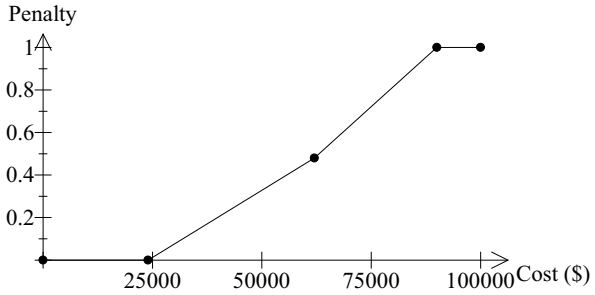


Figure 58: Example cost penalty function.

The algorithm inputs the resource functions for all sessions and items, represented by the vectors $rf_1 \ldots rf_m$, and the initial cost that we are starting from. It returns a prioritized list of suggested orders.

```
BEST-ORDERS-SEARCH (rf, initialCost)
pq = new priority queue
for each resource in rf
        Add the first incremental order to pq
currentQuality =  0
bestQuality = 0
currentCost = initialCost
while(!ISEMPTY(pq))
        Pop an incremental order from the priority queue and store its quality and cost.
        If there is another incremental order for this request/resource, push it on the priority queue.
        currentQuality += quality
         currentCost += cost
        costAdjustedQuality = currentQuality – GET-PENALTY-FOR-COST(currentCost)
        bestQuality = MAX(bestQuality, costAdjustedQuality)
for each resource in rf
        Add the first incremental order to pq
currentQuality =  0
bestQuality = 0
currentCost = initialCost
while(!ISEMPTY(pq) and costAdjustedQuality < bestQuality)
        Pop an incremental order from the priority queue and store its quality and cost.
        If there is another incremental order for this request/resource, push it on the priority queue.
        currentQuality += quality
        currentCost += cost
  costAdjustedQuality = currentQuality - GET-PENALTY-FOR-COST(currentCost)
```

Figure 59: Computation of the list of best vendor orders to place.

To select the best order to place, the orders are inserted into a priority queue. We determine the priority of a given order according to the following heuristic: We first compare on "schedule quality increase per cost increase" (which we call *quotient*), then on cost increase, and finally on request name and resource name in order to break ties. Since it is possible for the quotient to be infinite (if cost increase is zero), we treat all incremental orders with infinite quotient as being better than any incremental orders with finite quotients. We show the steps of the algorithm in Figure 59.

5.4. Elicitation of Vendor Data

Uncertainty in the vendor information means the vendor optimizer may not be able to place orders from vendors which may offer an item for a cheaper price. Even worse, the vendor optimizer may not be able to place an order which would satisfy a certain service because it does not have information about any vendors that may provide a resource item

satisfying that service. We need an effective elicitation mechanism which would generate an ordered (in terms potential usefulness) list of questions on uncertainties in vendor data.

We can ask the user to find out information about the items each vendor provides, the prices of those items and whether or not the list of vendors the system knows about is complete. We show a list of request types in Table 20.

• Provide the exact value for the price of a resource item from a given vendor. *Example:* Find out the price of ordering a Dell laptop from Shadyside Computers.
• Provide the exact value for the price of a set of resource items from a given vendor that may satisfy a certain service. *Example:* Find out the price of ordering a Dell laptop, IBM laptop, or Sony laptop from Shadyside Computers.
• Provide whether or not a vendor offers a certain resource item and the price. *Example:* Find out whether Oakland Florists offers rose arrangements; if found, please enter the price.
• Provide whether or not a vendor offers a set of resource items that may satisfy a certain service and the price. *Example:* Find out whether Oakland Florists offers rose arrangements, spring bouquets, or potted arrangements; if found, please enter the price.
• Verify if the list of vendors is complete. *Example:* Find out whether any available vendors are not yet in the system; if you find such vendors, please enter their names.

Table 20: Types of requests about vendors to the user. The system may ask the user to find out more information about available vendors, and resource items offered by those vendors.

The algorithm returns a prioritized list of suggested orders.

```
VENDOR-ELICITATION()
currentCost = GET-STARTING-COST()
startingPenalty = GET-PENALTY-FOR-COST(startingCost)
allServices = GET-ALL-SERVICES()
for each service in allServices
        items = GET-ITEMS-SATISFYING-SERVICE(service)
        for each item in items
                costFunctions[service] += costFunction[item]
        unifiedCostFn[service] = Unify costFunctions[service] by picking highest cost increase per item
        bestScore = -1
        for each point in unifiedCostFn[service]
                score = GET-PENALTY-FOR-COST(startingCost + cost[point])
                if (score > bestScore) bestScore = score
        score[service] = bestScore
sortedServices = Sort services in allServices based on each service's score
return sortedServices
```

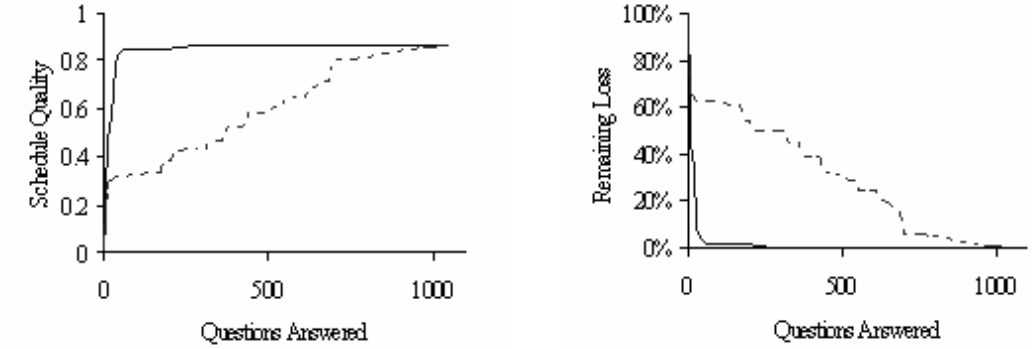
Figure 60: Vendor elicitation algorithm.

The vendor information elicitation algorithm starts out by enumerating all of the services and the corresponding resource functions the system knows about. It ranks the services based on the potential increase in penalty due to costs as more items are ordered. We show the vendor elicitation algorithm in Figure 60.

5.5. Experiments

We evaluate the vendor elicitor based on the information about vendors and the relevant vendor orders placed for the conference setting used in Section 3.1. We compare the results to just selecting questions to answer at random.

There are 13 vendors and 146 items each vendor might offer. We allow the system to have the price of one randomly selected item for each service category. The total number of possible questions is about one thousand (1050 to be exact) including questions about the price for each item, whether or not a given vendor offers a given item and so on. We observe the change in the quality of the produced schedules after each question is answered and show the results in Figure 61 (a). We calculate the remaining loss in quality in the same way as in the previous sections and show the graph in Figure 61 (b). We see that the vendor elicitor reaches the schedule quality of a fully certain world state after we answer about only 5% of the questions. For random elicitation, this figure is around 90%.



(a) Comparison of two systems

(b) Remaining loss for two systems

Figure 61: Dependency of the schedule quality on the number of answered questions using vendor elicitor. Dashed lines show the random selection, whereas solid lines show the vendor elicitor.

6. CONCLUSIONS

In this thesis, we presented a novel approach to elicitation for improving the quality of optimization under uncertainty. There are three main contributions we made:

- **Fast computation of expected impact for potential questions**
  The heuristic elicitor algorithm involves considering the standard deviation in schedule quality due to possible answers of a question in order to determine that question's priority. This provides us with a very quick way of estimating question importances in a more reliable way than simple heuristics.

- **Use of the optimizer as a part of B* search for refining the order of questions**
  The search elicitor algorithm involves using a very tight integration with the optimizer in deciding on the order of questions. Instead of looking at just the standard deviation, the search elicitor actually uses quick runs of the optimizer as a way to evaluate nodes in the search space to get a more realistic estimate of the impact of answering a given question.

- **Unifying different elicitation strategies**
  We unify the heuristic, rule-based and search elicitors producing a system which aims to do better than each of the individual elicitation strategies. The heuristic and rule-based elicitor outputs are refined by the search elicitor producing a more effective list of questions.

All of these contributions are applicable not just in the domains that we have applied them to but to any resource allocation problem with uncertainty. Even though primarily our work has been implemented within the RADAR architecture, the elicitation approach makes no domain specific assumptions with only the exception of the rule based elicitor which we do not claim as a contribution by itself.

We also evaluated each of the implemented elicitation systems using different problem sizes in the academic conference planning domain and in the vendor orders domain. We found that the full system helps optimizer produce a schedule of equivalent quality to a fully certain world state when less than 18% of possible questions are answered. In the vendor orders domain this figure was around 5%.

6.1. Limitations

Our approach seems to do much better as the number of possible questions and the difference in quality between the schedules produced from the initial uncertain world state and the final fully certain world state increases. Accordingly, one limitation of the approach is that if the number of possible questions is small, a heuristic approach much like the Rule-based elicitor or using simple standard deviation like the Heuristic Elicitor can achieve results as good as the unified system.

Another limitation of the approach is that the costs cannot be learnt automatically. That is, it is not possible for the system to observe how long a certain question or a class of questions (for example, questions about room size) generally take to answer and adjust the costs of such questions accordingly.

6.2. Future Work

We plan to design and implement a number of extensions to the approach that will complement the existing modules. We present them here in the order we would like to work on them.

The first extension we would like to work on is modeling variable cost in the system. By looking at the previously posed questions and time taken to answer them, we can have our system dynamically adjust the cost of asking questions. This would mean more accurate question weights and would help improve the prioritization of questions.

Another extension which would be very useful in a practical scenario is considering the possibility of partial uncertainty reduction instead of the complete elimination of uncertainty. A user may be able to answer a question giving a tighter range of values than before and this may still help us in producing a better schedule. Ranking such questions higher than the ones which need complete or almost complete certainty may improve the effectiveness of the elicitation system.

Even though we have made the assumption that the uncertain variables we ask questions about are completely independent, in some domains this may not be true. One possible future work direction is exploring domains where this assumption does not hold and seeing how this impacts the effectiveness of our algorithm. We believe that this may lead to certain tweaks to the algorithm which would make it more resistant to problems due to the violation of this assumption.

Another extension involves learning from past elicitation results. Given the ranking of different kinds of uncertainty and various properties of the world state we may be able to derive some "common sense" rules. The system can then use these rules to generate certain questions automatically without having to go through the other elicitation modules. For example, it may turn out to be the case that whenever there is an Auditorium with an uncertain number of microphones with the possibility of no microphones existing in the room we always generate a question about the microphones in that auditorium.

As a final extension, we want to improve our approach by considering how answering one question can make answering another one easier or harder and how in some cases grouping questions together can make sense. For example, if we are already asking the user to measure the size of a room, then asking them to count the number of projectors in the room at the same time would make sense. However, this would also mean the projector question could be taking the slot of another question which may potentially be more important.

7. BIBLIOGRAPHY

[Averbakh, 2001] Igor C. Averbakh. On the complexity of a class of combinatorial optimization problems with uncertainty. *Mathematical Programming*, 90(2), pages 263–272, 2001.

[Balasubramanian and Grossmann, 2003] Jayanth Balasubramanian and Ignacio E. Grossmann. Scheduling optimization under uncertainty: An alternative approach. *Computers and Chemical Engineering*, 27(4), pages 469–490, 2003.

[Berliner, 1979] Hans J. Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1), pages 23-40, 1979.

[Blum *et al.*, 2004] Avrim Blum, Jeffrey Jackson, Tuomas Sandholm, and Martin A. Zinkevich. Preference elicitaion and query learning. *Journal of Machine Learning Research*, 5, pages 649–667, 2004.

[Borrajo and Veloso, 1993] Daniel Borrajo, and Manuela Veloso. Bounded explanation and inductive refinement for acquiring control knowledge. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, pages 21–27, 1993.

[Borrajo and Veloso, 1994] Daniel Borrajo, and Manuela Veloso. Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning*, pages 64–82, 1994.

[Borrajo and Veloso, 1997] Daniel Borrajo, and Manuela Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *Artificial Intelligence Review Journal Special Issue on Lazy Learning*, 11, (1–5), pages 371–405, 1997.

[Boutilier *et al.*, 2003a] Craig Boutilier, Ronen Brafman, Carmel Domshlak, Holger H. Hoos, and David Poole. CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. In Martha E. Pollack editor, *Journal of Artificial Intelligence Research*. 21. 2003a.

[Boutilier *et al.*, 2004a] Craig Boutilier, Ronen Brafman, Carmel Domshlak, Holger H. Hoos, and David Poole. Preference-based constraint optimization with CP-nets. *Computational Intelligence*, 20, (2), pages 137–157, 2004a.

[Boutilier *et al.*, 1997] Craig Boutilier, Ronen Brafman, Chris Geib, and David Poole. A constraint-based approach to preference elicitation and decision making. In *Proceedings of the AAAI Spring Symposium on Qualitative Decision Theory*, pages 19–28, 1997.

[Boutilier *et al.*, 2003b] Craig Boutilier, Rajarshi Das, Jeffrey O. Kephart, Gerald Tesauro, and William E. Walsh. Cooperative negotiation in autonomic systems using incremental utility elicitation. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, pages 89–97, 2003b.

[Boutilier *et al.*, 2003c] Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. Constraint-based optimization with the minimax decision criterion. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 168–182, 2003c.

[Boutilier *et al.*, 2005] Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. Regret-based utility elicitation in constraint-based decision problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 929–934, 2005.

[Boutilier *et al.*, 2004b] Craig Boutilier, Tuomas Sandholm, and Rob Shields. Eliciting bid taker non-price preferences in (combinatorial) auctions. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 204–211, 2004b.

[Boutilier and Zemel, 2003] Craig Boutilier, and Richard S. Zemel. Online queries for collaborative filtering. In *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, pages n/a, 2003.

[Boutilier *et al.*, 2003d] Craig Boutilier, Richard S. Zemel, and Benjamin Marlin. Active collaborative filtering. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, pages 98–106, 2003d.

[Braziunas and Boutilier, 2005] Darius Braziunas, and Craig Boutilier. Local utility elicitation in (GAI) models. In *Proceedings of the Proceedings of the Twentyfirst Conference on Uncertainty in Artificial Intelligence*, pages 42–49, 2005.

[Burke, 1999] Robin D. Burke. The Wasabi Personal Shopper: A case-based recommender system. In *Proceedings of the Eleventh National Conference on Innovative Applications of Artificial Intelligence*, pages 844–849, 1999.

[Burke, 2000a] Robin D. Burke. Knowledge-based recommender systems. In Allen Kent editor, *Encyclopedia of Library and Information Systems*. 69, Supplement 32. CRC Press, New York, NY. 2000a.

[Burke, 2000b] Robin D. Burke. Semantic ratings and heuristic similarity for collaborative filtering. In *Proceedings of the AAAI Workshop on Knowledge-Based Electronic Markets*, pages 14–20, 2000b.

[Burke *et al.*, 1996] Robin D. Burke, Kristian J. Hammond, and Benjamin C. Young. Knowledge-based navigation of complex information spaces. In *Proceedings of the The Proceedings of The Thirteenth National Conference on Artificial Intelligence*, pages 462–468, 1996.

[Burke *et al.*, 1997] Robin D. Burke, Kristian J. Hammond, and Benjamin C. Young. The FindMe approach to assisted browsing. *IEEE Expert*, 12, (4), pages 32–40, 1997.

[Chajewska *et al.*, 1998] Urszula Chajewska, Lise Getoor, Joseph Normal, and Yuval Shahar. Utility elicitation as a classification problem. In *Proceedings of the Fourtheenth Conference on Uncertainty in Artificial Intelligence*, pages 79–88, 1998.

[Chen and Pu, 2004] Li Chen, and Pearl Pu. Survey of preference elicitation methods *Technical Report No. IC/200467, Swiss Federal Institute of Technology in Lausanne*, pages 1–23, 2004.

[Faltings *et al.*, 2004] Boi Faltings, Pearl Pu, Marc Torrens, and Paolo Viappiani. Designing example-critiquing interaction. In *Proceedings of the Ninth International Conference on Intelligent User Interfaces*, pages 22–29, 2004.

[Gajos and Weld, 2005] Krzysztof Gajos, and Daniel S. Weld. Preference elicitation for interface optimization. In *Proceedings of the Eighteenth Annual ACM Symposium on User interface Software and Technology*, pages 173–182, 2005.

[Ha and Haddawy, 1998] Vu Ha, and Peter Haddawy. Toward case-based preference elicitation: Similarity measures on preference structures. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 193–201, 1998.

[Ha and Haddawy, 2003] Vu Ha, and Peter Haddawy. Similarity of personal preferences: Theoretical foundations and empirical analysis. *Artificial Intelligence*, 146, (2), pages 149–173, 2003.

[Hill *et al.*, 1995] Will Hill, Larry Stead, Mark Rosenstein, and George Furnas. Recommending and evaluating choices in a virtual community of use. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 194–201, 1995.

[Kress and Boutilier, 2004] Alexander Kress, and Craig Boutilier. A study of limited-precision, incremental elicitation in auctions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, pages 1344–1345, 2004.

[Lin *et al*., 2004] Xiaoxia Lin, Stacy L. Janak, and Christodoulos A. Floudas. A new robust optimization approach for scheduling under uncertainty: Bounded uncertainty. *Computers and Chemical Engineering*, 28(6), pages 1069–1085, 2004.

[Linden *et al.*, 1997] Greg Linden, Steve Hanks, and Neal Lesh. Interactive assessment of user preference Models: The Automated Travel Assistant. In *Proceedings of the Sixth Conference on User Modeling*, pages 67–78, 1997.

[Lodwick *et al*., 2001] Weldon A. Lodwick, Arnold Neumaier, and Francis Newman. Optimization under uncertainty: Methods and applications in radiation therapy. In *Proceedings of the Tenth IEEE International Conference on Fuzzy Systems*, pages 1219–1222, 2001.

[McCarthy *et al.*, 2005] Kevin McCarthy, James Reilly, Lorraine McGinty, and Barry Smyth. Experiments in dynamic critiquing. In *Proceedings of the Tenth International Conference on Intelligent User Interfaces*, pages 175–182, 2005.

[Minton, 1988] Steven Minton. Learning search control knowledge: An explanation-based approach. Kluwer Academic Publishers, Boston, MA. 1988.

[Moore, 2002] Frank W. Moore. A methodology for missile countermeasures optimization under uncertainty. *Evolutionary Computation*, 10(2), pages 129–149, 2002.

[Patrascu *et al.*, 2005] Relu Patrascu, Craig Boutilier, Rajarshi Das, Jeffrey O. Kephart, Gerald Tesauro, and William E. Walsh. New approaches to optimization and utility elicitation in autonomic computing. In *Proceedings of the National Conference on Artificial Intelligence*, pages 140–145, 2005.

[Perez and Carbonell, 1994] M. Alicia Perez, and Jaime Carbonell. Control knowledge to improve plan quality. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 323–328, 1994.

[Perez and Etzioni, 1992] M. Alicia Perez, and Oren Etzioni. DYNAMIC: A new role for training problems in EBL. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 367-372, 1992.

[Pu and Faltings, 2000] Pearl Pu, and Boi Faltings. Enriching buyers' experiences: the SmartClient approach. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 289–296, 2000.

[Pu and Faltings, 2002] Pearl Pu, and Boi Faltings. Personalized navigation of heterogeneous product spaces using SmartClient. In *Proceedings of the Seventh International Conference on Intelligent User Interfaces*, pages 212–213, 2002.

[Pu *et al.*, 2003a] Pearl Pu, Boi Faltings, and Marc Torrens. User-involved preference elicitation. In *workshop notes, workshop on Configuration, the Eighteenth International Joint Conference on Artificial Intelligence*, 2003a.

[Pu *et al.*, 2003b] Pearl Pu, Pratyush Kumar, and Boi Faltings. User-involved tradeoff analysis in configuration tasks. In *workshop notes, the Third International Workshop on User-Interaction in Constraint Satisfaction, Ninth International Conference on Principles and Practice of Constraint Programming*, 2003b.

[Rashid *et al.*, 2002] Al Mamunur Rashid, Istvan Albert, Dan Cosley, Shyong K. Lam, Sean M. McNee, Joseph A. Konstan, and John Riedl. Getting to know you: Learning new user preferences in recommender systems. In *Proceedings of the International Conference on Intelligent User Interfaces*, pages 127–134, 2002.

[Resnick *et al.*, 1994] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 175–186, 1994.

[Sahinidis, 2004] Nikolaos V. Sahinidis. Optimization under uncertainty: State-of-the-art and opportunities. *Computers and Chemical Engineering*, 28(6), pages 971–983, 2004.

[Sandholm and Boutilier, 2006] Tuomas Sandholm, and Craig Boutilier. Preference Elicitation in Combinatorial Auctions. In Peter Cramton, Yoav Shoham and Richard Steinberg editor, *Combinatorial Auctions*. The MIT Press, Cambridge, MA. 2006.

[Schafer *et al.*, 2001] Ben J. Schafer, Joseph A. Konstan, and John Riedl. E-commerce recommender applications. *Data Mining and Knowledge Discovery* 5, (1/2), pages 115–153, 2001.

[Shardanand and Maes, 1995] Upendra Shardanand, and Pattie Maes. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 210–217, 1995.

[Smith *et al.*, 2002] Trey Smith, Tuomas Sandholm, and Reid Simmons. Constructing and clearing combinatorial exchanges using preference elicitation. In *Proceedings of the AAAI-02 Workshop on Preferences in AI and CP: Symbolic Approaches*, pages 87–93, 2002.

[Stolze and Nart, 2004] Markus Stolze, and Fabian Nart. Well-integrated needs-oriented recommender components regarded as helpful. In Elizabeth Dykstra-Erickson and Manfred Tscheligi editor, *Extended abstracts of the 2004 Conference on Human Factors in Computing Systems*. 1. ACM Press, Vienna, Austria. 2004.

[Stolze and Rjaibi, 2001] Markus Stolze, and Walid Rjaibi. Towards scalable scoring for preference-based Item recommendation. *IEEE Data Engineering Bulletin*, 24, (3), pages 42–49, 2001.

[Stolze and Ströbel, 2001] Markus Stolze, and Michael Ströbel. Utility-based decision tree optimization: A framework for adaptive interviewing. In *Proceedings of the Eighth International Conference on User Modeling*, pages 105–116, 2001.

[Stolze and Ströbel, 2003] Markus Stolze, and Michael Ströbel. Dealing with learning in eCommerce product navigation and decision support: the Teaching Salesman Problem. In *Proceedings of the Second Interdiciplinary World Congress on Mass Customization and Personalization*, pages n/a, 2003.

[Torrens *et al.*, 2003] Marc Torrens, Patrick Herzog, Loic Samson, and Boi Faltings. Electronic commerce technologies: reality: a scalable intelligent travel planner. In

Proceedings of the 2003 ACM Symposium on Applied Computing, pages 623–630, 2003.

[Veloso *et al.*, 1995] Manuela Veloso, Jaime Carbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7, (1), pages 81–120, 1995.

[Wang and Boutilier, 2003] Tianhan Wang, and Craig Boutilier. Incremental utility elicitation with the minimax regret decision criterion. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 309–316, 2003.

[Zinkevich *et al.*, 2003] Martin A. Zinkevich, Avrim Blum, and Tuomas Sandholm. On polynomial-time preference elicitation with value queries. In *Proceedings of the ACM Conference on Electronic Commerce*, pages 176–185, 2003.