Scheduling with Uncertain Resources: Search for a Near-Optimal Solution

Eugene Fink
e.fink@cs.cmu.edu

P. Matthew Jennings
mattj@cs.cmu.edu

Ulas Bardak
cyprus@cs.cmu.edu

Jean Oh
jeanoh@cs.cmu.edu

Stephen F. Smith
sfs@cs.cmu.edu

Jaime G. Carbonell
jgc@cs.cmu.edu

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

***Abstract*—We describe a system for scheduling a conference based on incomplete information about available resources and scheduling constraints. We explain the representation of uncertain knowledge, describe a local-search algorithm for generating near-optimal schedules, and give empirical results of automatic scheduling under uncertainty.**

I. Introduction

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.

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 scheduling a conference based on uncertain information about available resources and conference events. The previous techniques have turned out inapplicable to this problem, and we have developed a new mechanism for scheduling under uncertainty. This work has been part of the RADAR project (www.radar.cs.cmu.edu) at Carnegie Mellon University, which is aimed at building an intelligent system for assisting an office manager. We have described initial results of this work in three earlier papers; specifically, we have explained the representation of uncertainty [Bardak *et al.*, 2006a], automatic elicitation of additional data that help to reduce uncertainty [Bardak *et al.*, 2006b], and collaboration between the scheduling system and human user [Fink *et al.*, 2006].

We now describe an algorithm for constructing a schedule based on uncertain knowledge of resources and constraints. We explain the representation of uncertain facts (Sections II–IV), present the search for a near-optimal schedule (Sections V and VI), and give empirical results on its effectiveness (Section VII).

II. Example

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: auditorium, classroom, and conference room (Figure 1). These rooms host other events, and they are available for the conference only at the following times:

Auditorium: 11:00am–1:30pm and 3:30pm–4:30pm.
Classroom: 11:00am–2:30pm.
Conference room: 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.

We also specify distances between rooms in feet; we assume that the auditorium and classroom are next to each other, whereas the conference room is in another building. In Figure 1, we show the properties of each room and the distances between rooms.

The conference includes five events: demonstration, discussion, tutorial, workshop, and committee meeting (Table 1). For each event, we specify its importance, as well as related constraints and preferences.

We define constraints by limiting appropriate start times, durations, and room properties. For example, we may indicate that an acceptable start time for the tutorial is 1:00pm or

The manuscript was received on March 30, 2006. The described work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010.

earlier, an acceptable duration is 30 minutes or more, and an acceptable room size is 400 square feet or more.

In addition, we define constraints for distances between events and for relative start times of events with respect to other events. For instance, we may specify that the workshop must be in the same room as the tutorial, and that it must start shortly after the tutorial, because many participants plan to attend both events. We may also indicate that the tutorial and workshop must be near the demo, which will allow their attendees to see the demo during the breaks.

We may also select preferred values for start times, durations, room properties, distances, and relative start times, which are subsets of acceptable values. For example, we may specify that the preferred start time for the tutorial is 11:00am, preferred duration is 60 minutes, and preferred room size is 600 square feet or more. We may further indicate that the preferred distance from the workshop to the demo is 100 feet or less, and the preferred start time for the workshop is 30 minutes after the end of the tutorial. In Table 1, we give constraints and preferences for all events.

We construct a schedule by assigning a room and time slot to every event. For instance, the schedule in Figure 2 satisfies all constraints and most preferences given in Table 1.

III. Representation

We now explain the representation of resources and scheduling requirements [Bardak *et al*., 2006a].

Rooms: We represent resources by a set of available rooms; the description of a room includes its name and a list of numeric properties (see Figure 1). For each room, we define its property values and distances to other rooms, as well as its availability, represented by a set of time intervals.

Events: The description of an event includes its name, importance, and related constraints and preferences (see Table 1). The importance is a positive integer; the constraints are ranges of acceptable values for start time, duration, room properties, distances, and relative start times; and the preferences are ranges of preferred values, which must be sub-ranges of the respective acceptable values. Thus, when specifying an event, we may include a range of acceptable values and a sub-range of preferred values for each of the following parameters:

- Start time and duration
- Every room property
- For every other event, the distance from the specified event to the location of the other event
- For every other event, the time difference between the start of the specified event and the start of the other event
- For every other event, the time difference between the start of the specified event and the end of the other event

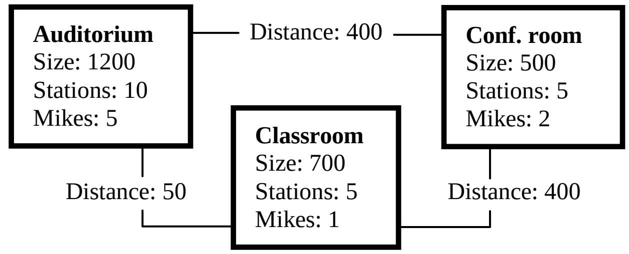


Figure 1. Available rooms, their properties, and distances.

		Demo	Discu-ssion	Tuto-rial	Com-mittee	Work-shop
Importance		5	3	8	1	5
Start time	Acceptable Preferred	Any	Any	≤1pm 11am	≥3pm 3:30pm	Any
Dura-tion	Acceptable Preferred	≥60 150	≥30 90	≥30 60	≥30 60	≥60 120
Room size	Acceptable Preferred	≥600 ≥1200	≥200 ≥600	≥400 ≥600	≥400 ≥800	≥600 ≥1000
Stat-ions	Acceptable Preferred	≥5 ≥10	Any	Any ≥2	Any	Any
Mikes	Acceptable Preferred	Any	≥2 ≥4	≥1 ≥2	Any	≥1 ≥1

(a) Constraints on start times, durations, and room properties.

Event	Parameter	With Respect To	Acceptable	Preferred
Demo	Start Time	Tutorial's Start	[−30..30]	0
Work-shop	Distance	Demo's Room	≤200	≤100
	Distance	Tutorial's Room	0	0
	Start Time	Tutorial's End	[0..60]	30

(b) Constraints on distances and relative start times.

Table 1. Conference events and related constraints and preferences.

	Auditorium	Classroom	Conf. room
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 2. Schedule for the conference scenario in Tables 1 and 2.

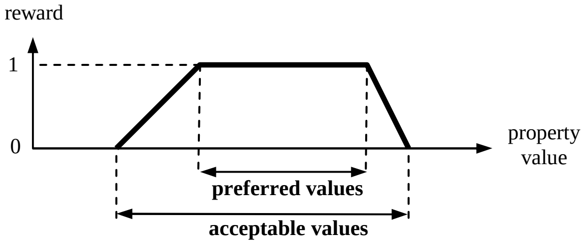


Figure 3. Reward for satisfying a preference.

```
Sort the events in the decreasing order of expected importances.
For every event:
    Create a list of the rooms and time slots that are consistent
        with the ranges of acceptable values for this event.
Create the empty schedule; that is, mark all events as unscheduled.
Repeat until finding no improvements or reaching a time limit:
    For every event, in the order of decreasing importances:
        For every room and time slot consistent with this event:
            Move the event into this room and time slot.
            If some other events overlap with this event,
                then remove them from the schedule.
            If the distances from the moved event
                to some other events are unacceptable,
                then remove these other events.
            If the relative times of the moved event
                w.r.t. some other events are unacceptable,
                then remove these other events.
            Re-compute the schedule utility.
            If the new utility is no greater than the old utility,
                then undo the related schedule changes.
```

Figure 4: Search for a high-quality schedule.

Uncertainty: When scheduling a conference, we may have incomplete information about resources, event importances, constraints, and preferences. We represent an uncertain value as an interval, encoded by the minimal and maximal possible values. For example, we may specify that the size of the conference room is between 500 and 750, the importance of the demo is between 4 and 6, and the minimal acceptable duration of the demo is between 60 and 90.

Schedule: To build a schedule, the system assigns a room and time slot to each event. It represents this assignment by the event name, room name, start time, and duration. Alternatively, it can decide that an event is not part of the schedule, which is also considered an assignment; the system represents this assignment by setting its room to NIL. Note that assignments must not overlap, that is, the system cannot assign two events to the same room at the same time.

IV. Schedule quality

We measure schedule quality on the scale from 0.0 to 1.0; higher values correspond to better schedules. The quality of a specific assignment depends on how well the selected room and time slot match the preferred values. If the start time, duration, some room property, distance to another event, or time with respect to another event is outside the acceptable range, then the assignment quality is zero, regardless of the other constraints. If we decide that an event is not part of the schedule, the quality of its assignment is also zero.

If an assignment satisfies all hard constraints, we determine the rewards for satisfying the related preferences. If a start time, duration, room property, distance to another event, or time with respect to another event is within the preferred range of values, then the respective reward is 1.0. If it is outside the preferred range, the reward depends on its distance from this range; specifically, the reward linearly decreases with the distance from the preferred values, as shown in Figure 3. If an event has a distance or relative-time preference with respect to another event that is left unscheduled, we consider this preference satisfied, and the respective reward is 1.0. If the event has k preferences, and the respective rewards are $r_1,\ldots, r_k$, then the assignment quality is $(r_1 + \ldots + r_k) / k$.

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

$$\left(imp_1 \cdot Qual_1 + \ldots + imp_n \cdot Qual_n\right) / \left(imp_1 + \ldots + imp_n\right).$$

For example, if we use the preferences in Table 1, and the schedule is as shown in Figure 2, then the quality of the time slot for the demo is 1.0, for the discussion is 0.75, for the tutorial is 0.8, for the committee meeting is 1.0, and for the workshop is 0.91, and the overall schedule quality is 0.87.

If the description of rooms and events includes uncertainty, the system computes the mathematical expectation of schedule quality. It determines the expected quality of individual assignments, $E(Qual_1),\ldots, E(Qual_n)$, as well as the expected values of their importances, $E(imp_1),\ldots, E(imp_n)$, and uses them to compute the expected quality of the schedule, which is

$$\left(E(imp_1) \cdot E(Qual_1) + \ldots + E(imp_n) \cdot E(Qual_n)\right) / \left(E(imp_1) + \ldots + E(imp_n)\right).$$

We have given an algorithm for fast computation of this expected quality in the paper on the representation of uncertainty [Bardak *et al.*, 2006a].

For instance, consider the example in Section II, and suppose that the conference-room size is between 500 and 750, the demo importance is between 4 and 6, the minimal acceptable duration of the demo is between 60 and 90, and all other resources and constraints are fully certain, as shown in Tables 1 and 2. Then, the expected quality of the schedule in Figure 2 is 0.88.

V. 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 4, we give the main steps of the hill-climbing search algorithm, which processes the events in the 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.

We next present a more detailed description of this search algorithm. We list its main variables in Figure 5, show its main procedures and calls between them in Figure 6, and give pseudocode for these procedures in Figures 7–13. 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 because 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 13), 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.

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

Expected rewards: If the description of rooms and events includes uncertainty, the procedures in Figures 8 and 9 compute the mathematical expectations of rewards. We have given algorithms for fast computation of expected rewards in the paper on representing uncertainty [Bardak *et al*., 2006a].

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 non-overlapping time intervals; this representation allows fast checking whether the room is available for a given time slot.

VI. EXTENSIONS

We outline several extensions to the described algorithm; we have implemented these extensions and used them in the experiments of Section VII.

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 11), as well as adding the re-computation of end-time rewards to REMOVAL, NEW-START-TIME, and NEW-DURATION (Figure 12).

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 $w_1,\dots, w_k$, and the respective rewards are $r_1,\dots, r_k$, then the assignment quality is $(w_1 \cdot r_1 + \dots + w_k \cdot r_k) / (w_1 + \dots + w_k)$. The use of weights requires modifications to the computation of reward limits in SCORE-LIMITS (Figure 7), as well as to the reward computations in the ROOM-PROP-DIFF and DISTANCE-DIFF procedures in Figure 8, and the START-TIME-DIFF, DURATION-DIFF, and END-TIME-DIFF procedures in Figure 9.

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 [Fink *et al*., 2006]. 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.

Locked assignments: The user can "lock" some events in the manually selected places, and apply the system to find assignments for the other events [Fink *et al*., 2006]. This option requires a modification to the top-level SCHEDULER procedure (Figure 13); specifically, SCHEDULER should skip the locked events in its main loop, thus ensuring that they remain in their original places.

(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 pseudocode:

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 13):

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	differences between the quality of new candidate start times and that of the event's current start time
dur-diffs	differences between the quality of new candidate durations and that of the event's current duration
end-diffs	differences 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 5: Main variables in the procedures given in Figures 7–13.

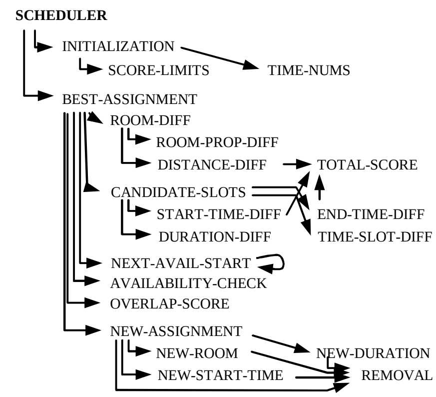


Figure 6: Main procedures of the algorithm given in Figures 7–13.

The 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*)
let *num-room* be the number of *event*'s preferences
  for room properties and distances,
 num-start be the number of *event*'s preferences
  for the start time and relative start times, and
 num-dur be the number of *event*'s duration preferences
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 every *event* in *All-Events* **do**
 TIME-NUMS(*event*, *conf-start*, *conf-end*, *step*); SCORE-LIMITS(*event*)
for every *event* in *All-Events* **do**
 room[*event*] = NIL
 room-score[*event*] = 0; *start-score*[*event*] = 0; *dur-score*[*event*] = 0
for every *event* in *All-Events* **do**
 compute its expected importance and set *imp*[*event*] to this value
sort *All-Events* in the decreasing order of their expected importances

Figure 7: 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
for every room-property preference of event do
  if this property of room is unacceptable then return NIL
  let new-reward be the expected reward for this property in room,
    and old-reward be the expected reward in room[event]
  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
for every distance preference in event do
  let other-event be the related other event in the preference
  if distance from new-room to room[other-event] is unacceptable
    then dist-diff = dist-diff − TOTAL-SCORE(other-event)
    else let new-reward be the expected reward for the distance
                from new-room to room[other-event]
              and old-reward be the expected reward for the distance
                from room[event] to room[other-event]
            dist-diff = dist-diff + imp[event]
              · (new-reward − old-reward) / num-prefs[event]
for every other-event that has a distance preference w.r.t. event do
  if distance from room[other-event] to new-room is unacceptable
    then dist-diff = dist-diff − TOTAL-SCORE(other-event)
    else let new-reward be the expected reward for the distance
                from room[other-event] to new-room,
              and old-reward be the expected reward for the distance
                from room[other-event] to 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 8: Computing the reward-score difference between a new room and the old room of a given event. If the representation of rooms and events includes uncertainty, this computation relies on the algorithms for computing the mathematical expectation of preference values, described in the paper on the representation of uncertainty [Bardak *et al*., 2006a].

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 new-start is an unacceptable start time for event then return NIL
let new-reward be the expected start-time reward for new-start,
  and old-reward be the expected reward for start[event]
start-diff=imp[event] · (new-reward − old-reward)/num-prefs[event]
for every relative start-time preference in event do
  let other-event be the related other event in the preference
  if new-start is unacceptable w.r.t. the time of other-event
    then start-diff = start-diff − TOTAL-SCORE(other-event)
    else let new-reward be the expected reward for
                new-start w.r.t. the time of other-event
              and old-reward be the expected reward for
                start[event] w.r.t. the time of other-event
            start-diff = start-diff + imp[event]
              · (new-reward − old-reward) / num-prefs[event]
for every other-event that has a relative start-time preference
    with respect to the start time of event do
  if its relative start time w.r.t. new-start is unacceptable
    then start-diff = start-diff − TOTAL-SCORE(other-event)
    else let new-reward be the expected reward for
                its relative start time w.r.t. new-start
              and old-reward be the expected reward for
                its relative start time w.r.t. start[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 new-dur is an unacceptable duration for event then return NIL
let new-reward be the expected reward for new-dur,
  and old-reward be the expected reward for dur[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 w.r.t. its new end time and those w.r.t. its old end time.

```
END-TIME-DIFF(event, new-end)
old-end = start[event] + dur[event]
end-diff = 0
for every other-event that has a relative start-time preference
    with respect to the end time of event do
  if its relative start time w.r.t. new-end is unacceptable
    then end-diff = end-diff − TOTAL-SCORE(other-event)
    else let new-reward be the expected reward for
                its relative start time w.r.t. new-end
              and old-reward be the expected reward w.r.t. old-end
            end-diff = 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 every other-event that overlaps with the new place of event do
  score = score + TOTAL-SCORE[other-event]
return score
```

Figure 9: Computing the reward-score differences related to the start time, duration, and end time of a given event.

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*
search for the availability interval, in the sorted list of *room*'s
  availability intervals, that includes both *start* and *end*
if such an interval is found **then return** TRUE; **else 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*
let *room-end* be the ending time of *room*'s latest availability interval
if *end* > *room-end* **then return** NIL
identify the earliest *room*'s availability interval
  whose ending time is no earlier than *end*
let *interval-start* be the beginning time of this interval
if *start* ≥ *interval-start* **then return** *start-num*
interval-start-num = ⌈(*interval-start* − *conf-start*) / *step*⌉
return NEXT-AVAIL-START(*room*, *interval-start-num*,
    dur-num, *conf-start*, *step*)

Figure 10: Checking the availability of a room, and identifying the earliest available time slot in a room after a given time.

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* = *min-start-num*[*event*] **to** *max-start-num*[*event*] **do**
  new-start = *conf-start* + *start-num* · *step*
  start-diffs[*start-num*] = START-TIME-DIFF(*event*, *new-start*)
for *dur-num* = *min-dur-num*[*event*] **to** *max-dur-num*[*event*] **do**
  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* = *min-end-num* **to** *max-end-num* **do**
  new-end = *conf-start* + *end-num* · *step*
  end-diffs[*start-num*] = END-TIME-DIFF(*event*, *new-end*)
for *start-num* = *min-start-num*[*event*] **to** *max-start-num*[*event*] **do**
  if *start-diffs*[*start-num*] ≠ NIL
  then *best-slot-diff* = NIL
    for *dur-num* = *min-dur-num*[*event*]
      to min(*max-dur-num*[*event*],
        conf-end-num − *start-num*) **do**
     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 11: Evaluation of candidate time slots for a given event, where each slot is defined by its start time and duration.

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

```
REMOVAL(event)
room[event] = NIL
room-score[event] = 0; start-score[event] = 0; dur-score[event] = 0
for every other-event that has a distance preference w.r.t. event do
   adjust other-event's reward score for distances
for every other-event that has a start-time preference w.r.t. event do
   adjust other-event's reward score for relative start times
```

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 every distance preference in event do
   let other-event be the related other event in the preference
   if this distance is now unacceptable then REMOVAL(other-event)
for every other-event that has a distance preference w.r.t. event do
   if this distance is now unacceptable then REMOVAL(other-event)
   else re-compute other-event's reward score for the new distance
re-compute the value of 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 every preference on relative start time in event do
   let other-event be the related other event in the preference
   if the start time of event w.r.t. other-event is unacceptable
      then REMOVAL(other-event)
for every other-event that has a start-time preference
      w.r.t. the start time of event do
   if its start time is now unacceptable, then REMOVAL(other-event)
   else re-compute other-event's score for the relative start time
re-compute the value of 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
re-compute the value of dur-score[event]
if start[event] + dur[event] = old-end then return
for every other-event that has a start-time preference
      w.r.t. the end time of event do
   if its start time is now unacceptable then REMOVAL(other-event)
   else re-compute other-event's score for the relative start time
```

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 every other-event that overlaps with the new place of event do
   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 12: 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 every room in All-Rooms do
   room-diffs[room] = ROOM-DIFF(event, room)
slot-diffs = CANDIDATE-SLOTS(event, conf-start, conf-end, step)
best-asst-diff = 0
for every room in All-Rooms do
   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] do
         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) do
                  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)
let num-events be the number of events in All-Events
num-unchanged = 0
while the search time is smaller than run-time-limit do
   for every event in All-Events,
         in the order of decreasing importances do
      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 13: Top-level search procedure, which reschedules one event at a time, until reaching a local maximum or hitting the time limit.

VII. Experiments

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 14, we summarize the results of these experiments, which show that the system has outperformed the human subjects.

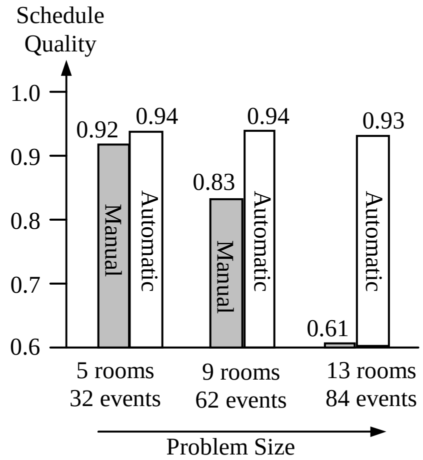
We have also evaluated the dependency of the quality of automatically constructed schedules on the search time, and we show the results in Figure 15. If the knowledge is fully certain, the system constructs a near-optimal schedule in about three seconds. If the knowledge is uncertain, it needs about nine seconds because it spends more time for computing the expected quality of candidate assignments.

VIII. Concluding remarks

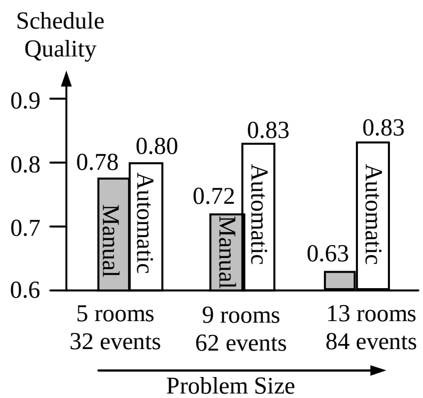
We have described a scheduling algorithm that accounts for uncertainty in resources and constraints. The experiments have confirmed that it quickly solves large-scale problems, and that the resulting schedules are better than manual solutions. We are now working on an extended system, which will support more flexible utility functions, optimize the use of portable equipment related to the scheduled events, and analyze the trade-offs involved in renting additional rooms and equipment.

Acknowledgments

We are grateful to Konstantin Salomatin, Greg Jorstad, and Daniel Cheng for their help in developing the representation of uncertain knowledge. We thank Chris R. Martens, Jason Knichel, Vijay Prakash, and Sung-joo Lim for their work on evaluating the scheduling system. We also thank Aaron Steinfeld and Matt Lahut for their help in applying the system to real-world scheduling problems.



(a) Experiments with fully certain knowledge.



(b) Experiments with uncertain knowledge.

Figure 14: 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.

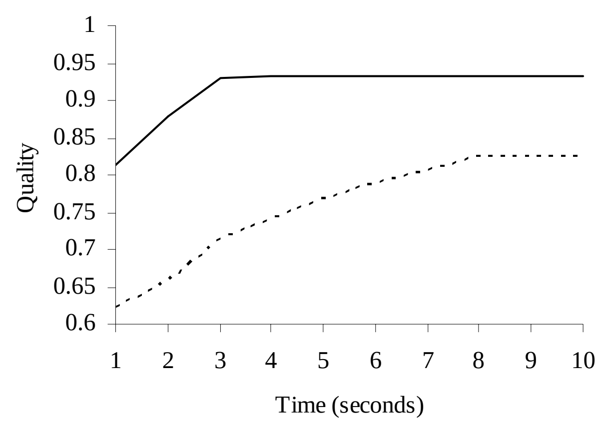


Figure 15. Dependency of the schedule quality on the running time. We show the results of scheduling with fully certain knowledge (dashed line) and uncertain knowledge (solid line); both problems include 13 rooms and 84 events.

References

[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.

[Bardak *et al.*, 2006a] Ulas Bardak, Eugene Fink, and Jaime G. Carbonell. Scheduling with uncertain resources: Representation and utility function. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 2006.

[Bardak *et al.*, 2006b] Ulas Bardak, Eugene Fink, Chris R. Martens, and Jaime G. Carbonell. Scheduling with uncertain resources: Elicitation of additional data. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 2006.

[Bidot, 2005] Julien Bidot. A general framework integrating techniques for scheduling under uncertainty. PhD Thesis, Institut National Polytechnique de Toulouse, 2005.

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

[Fink *et al.*, 2006] Eugene Fink, Ulas Bardak, Brandon Rothrock, and Jaime G. Carbonell. Scheduling with uncertain resources: Collaboration with the user. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 2006.

[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.

[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.

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

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