Managing Personal Tasks with Time Constraints and Preferences

Ioannis Refanidis

University of Macedonia, Dept. of Applied Informatics
Egnatia str. 156, 54006, Thessaloniki, Greece
yrefanid@uom.gr

Abstract

This paper treats the problem of managing personal tasks, through an adaptation of the Squeaky Wheel Optimization (SWO) framework, enhanced with powerful heuristics and full constraint propagation. The problem involves preemptive and non-preemptive tasks, with extra constraints imposed on the sizes of and the distances between the parts of each preemptive task. Travelling times are imposed by the alternative localization possibilities of each task. Ordering constraints are imposed by the producer-consumer relations between tasks. The user may have preferences regarding scheduling options of single tasks or pairs of tasks. Higher degree time constraints and preferences are supported as well. SWO allows for fast scheduling and rescheduling. Several heuristics are proposed to estimate the difficulty to schedule each task and to compensate with the degree of the user's satisfaction. Experimental results show that this approach is remarkably effective and efficient.

Introduction

Modern electronic organizers, such as MS-Outlook 2007, Google Calendar and Yahoo Calendar, do not provide for automatic scheduling of users' tasks. Users have to manually place their tasks into the calendar, as well as to arrange meetings with others. These applications provide various functionalities to assist the user to put her tasks into the calendar, detect conflicts, merge calendars, assign tasks to other users, arrange meetings, share and publish her calendar. The need for intelligent assistance to schedule a user's tasks has already been identified (Refanidis, McCluskey and Dimopoulos, 2004). It is a general impression that most of the effort in developing new editions of office applications is towards personal time management simplification. However, current status still remains behind automatic scheduling abilities (not to speak about planning) being incorporated into these programs.

We consider managing personal tasks as a variation of disjunctive preemptive scheduling (Le Pape and Baptiste, 1996) with setup times. First of all, no two tasks can overlap in time. Some tasks, such as writing a paper, may be interruptible, with special constraints imposed on the way they can split; other tasks, such as giving a lecture, are not. Each task is characterized by a set of location references, i.e. alternative places where the user should be in order to perform the task; additional time (setup time) is required to travel between these locations. Various forms of constraints between tasks, including ordering and proximity constraints are allowed. Similarly, various types of preferences, e.g. unary preferences regarding the various time-windows to schedule a task or binary preferences regarding the relative positions of pairs of tasks can be defined.

This paper treats the problem of scheduling in time and space tasks with arbitrary time constraints and preferences using an adaptation of the incomplete heuristic search framework Squeaky Wheel Optimization (Joslin and Clements, 1999). We propose general and adaptable heuristics for scheduling the tasks, with the ability to trade-off between efficiency, i.e. the ability to find a schedule quickly, and effectiveness, i.e. the quality of the found schedule (if any) wrt user preferences. Moreover, we propose an incremental scheduling schema, which preserves existing schedule as far as possible, when new tasks arrive or the current schedule becomes invalid for external reasons. The overall approach is tested experimentally through a large set of artificially created problem instances and is shown to be remarkably efficient and effective. In this paper we do not treat the underlying planning problem, we do not consider scheduling with resources and we do not cope with the problem of arranging meetings.

The rest of the paper is structured as follows: The next section formally defines the problem. Then we outline the SWO algorithm. The next two sections treat the problem of scheduling with arbitrary constraints, without and with time preferences respectively. Next we modify SWO to support incremental scheduling, present empirical results and, finally, conclude the paper and pose future directions.

Problem formulation

Time is considered discrete. Indeed, most electronic organizers use a minimum time slot, usually 30 mins. So, time is considered a non-negative integer, with zero denoting the current time. We have a set T of N tasks, $T=\{T_1, T_2, \ldots, T_N\}$. Each task $T_i \in T$ is characterized by several attributes, as described in the following paragraphs.

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Each task has duration, denoted with dur_i [2]. All tasks are considered interruptible, i.e. they can split into parts that can be scheduled separately. With the decision variable p_i we denote the number of parts in which the i-th task has been split, where $p_i \geq 1$. With T_{ij} we denote the j-th part of the i-th task, $1 \leq j \leq p_i$.

For each T_{ij} we introduce the decision variables t_{ij} and dur_{ij}, denoting the start time and the duration of this part respectively. The sum of the durations of all parts of a task must equal its total duration (C1).

For each task T_i, the maximum and minimum allowed duration for its parts, $smax_i$ and $smin_i$ (C2), as well as the minimum allowed temporal distance between every pair of its parts, $dmin_i$ (C3), are given. Depending on the values of $smax_i$ and $smin_i$ and the overall duration of the task dur_i, implicit constraints are imposed on p_i. For example, if $smin_i > dur_i/2$, then $p_i=1$, so task T_i is non-interruptible.

Each task i has its given domain D_i, consisting of a set of intervals within which all of its parts have to be scheduled (we do not consider tasks with infinite horizon of execution): $D_i=[a_{i1},b_{i1}]\cup[a_{i2},b_{i2}]\cup\ldots\cup[a_{iF_i},b_{iF_i}]$, where F_i is the number of intervals of D_i (C4). Obviously, $a_{ij}+smin_i \leq b_{ij}$ as well as $b_{ij} < a_{ij+1}$ must hold for each $1 \leq i \leq N$, $1 \leq j \leq F_i$.

We are given a set of M locations, $Loc=\{L_1, L_2, \ldots, L_M\}$ and a two dimensional matrix $Dist$ (not necessarily symmetric) with their temporal distances (non-negative integers). Each task T_i has its own spatial references, $Loc_i \subseteq Loc$, denoting alternative places where the user should be in order to execute each part of the task (the user has not to execute all the parts of a task in the same location). With the decision variable $l_{ij} \in Loc_i$ we denote the particular position where T_{ij} will be executed (C5, C6).

For any subset of tasks $S \subseteq T$, a constraint c over these tasks may be defined, thus determining the valid ways to schedule the tasks of the set. Constraints refer only to time, not to location references; however the role of the locations in deciding when to schedule a task is important, since the decision to schedule a task at a specific location may affect the domain of other tasks. Each constraint c is defined by one procedure and one function:

- $propagate_c(S)$: Given a set of partially instantiated tasks S, $propagate_c(S)$ applies the constraint $c(S)$ on the domains of these tasks.
- $eval_c(S)$: Given a set of partially instantiated tasks S, $eval_c(S)$ returns $\bot$ if this partial assignment violates the constraint $c(S)$, otherwise it returns $\top$ (C7).

A typical example is the ordering constraint, denoted with $\prec(T_i,T_j)$, meaning that no part of the j-th task can start its execution until all parts of the i-th task have finished their execution. In this case, $propagate_{\prec}(T_i,T_j)$ applies bounds consistency (Van Hentenryck et al., 1998) to the domains of T_i and T_j, whereas $eval_{\prec}(T_i,T_j)$ checks whether the current partial assignment satisfies the constraint. Higher order constraints can be defined in a similar way.

[2] In the following we use the notation x_i to abbreviate $T_i.x$, where x is any attribute of the task structure. In case of multiple subscripts, e.g. x_{ij}, the first one, i.e. i, indicates the task.

Finally, time preferences over sets of tasks are also allowed. A preference v over a set of tasks S is defined as a function $v : \prod_{T_i \in S} D_i \rightarrow \Re$, i.e. function v maps each combination of the domains of the tasks of S in a real number. For example, a unary preference could evaluate the possible time-windows when a task could be scheduled, whereas a binary preference could evaluate the relative positions in time of two tasks.

So, after these definitions, the problem of managing personal tasks can be formulated as follows:

Given a set of tasks T with their attributes, a set of constraints C and a set of preferences V, find appropriate values for the decision variables p_i, t_{ij}, dur_{ij}, l_{ij}, where $1 \leq i \leq N$, $1 \leq j \leq p_i$ such as to maximize the expression:

$$\sum_{v_k \in V} v_k(S_k) \quad (1)$$

subject to the following constraints:

C1: $\forall i,\ 1 \leq i \leq N:\ \sum_{j=1}^{p_i} dur_{ij} = dur_i$.

C2: $\forall i,j,\ 1 \leq i \leq N,\ 1 \leq j \leq p_i : smin_i \leq dur_{ij} \leq smax_i$.

C3: $\forall i,j,k,\ 1 \leq i \leq N,\ 1 \leq j \leq p_i,\ 1 \leq k \leq p_i : j \neq k \Rightarrow$
$t_{ij} \geq t_{ik}+dur_{ik}+dmin_i \vee t_{ik} \geq t_{ij}+dur_{ij}+dmin_i$

C4: $\forall i,j,\ 1 \leq i \leq N,\ 1 \leq j \leq p_i\ \exists k,\ 1 \leq k \leq F_i : a_{ik} \leq t_{ij} \leq b_{ik}-dur_{ij}$.

C5: $\forall i,j,\ 1 \leq i \leq N,\ 1 \leq j \leq p_i : l_{ij} \in Loc_i$.

C6: $\forall i,j,m,n,\ 1 \leq i \leq N,\ 1 \leq j \leq p_i,\ 1 \leq m \leq N,\ 1 \leq n \leq p_m : i \neq m \vee j \neq n$
$\Rightarrow t_{ij}+dur_{ij}+Dist(l_{ij},l_{mn}) \leq t_{mn} \vee t_{mn}+dur_{mn}+Dist(l_{mn},l_{ij}) \leq t_{ij}$

C7: $\forall c(S) \in C,\ eval_c(S) = \top$.

Squeaky Wheel Optimization

Squeaky Wheel Optimization (SWO) is a general optimization framework that can be adapted to several constraint satisfaction problems. The core of SWO is a Construct/Analyze/Prioritize cycle, as shown in Figure 1(a). Constraint variables are ordered in a priority queue in decreasing order of an initial estimate of the difficulty to assign a value to each one of them. A solution is constructed by a greedy algorithm, taking decisions in the order determined by the priority queue. This solution is then analyzed to find those constraint variables that were the “trouble makers”. The priorities of the “trouble makers” are increased, causing the greedy constructor to deal with them sooner in the next iteration. This cycle repeats until a termination condition occurs.

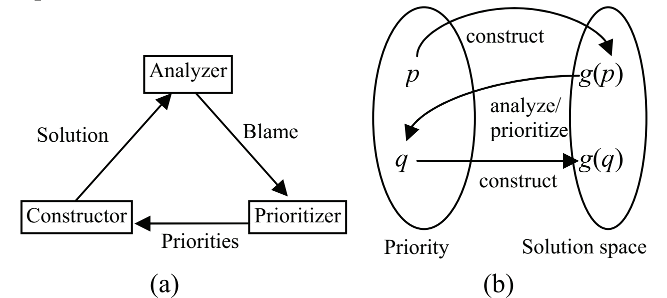

Figure 1. (a) The Construct/Analyze/Prioritize cycle. (b) Coupled search spaces.

SWO is a fast but incomplete (in the general case) search procedure. As shown in Figure 1(b), SWO searches in two coupled spaces: The priority space and the solution space. The greedy construction algorithm defines a function g from the priority queues to the solutions, i.e. for each ordering p of the tasks a schedule $g(p)$ is defined. However, function g may be neither surjective nor injective; so, many feasible solutions may not correspond to any ordering of the tasks in the queue.

Managing Personal Tasks without Preferences

This section presents our adaptation of SWO to the personal time management problem, with arbitrary constraints but without preferences. We present the initialization of the priority queue, the greedy construction algorithm and finally the queue reordering strategy.

Initialization of the Priority Queue

We employed a simple priority queue, with the priority of each task simply being its position in the queue. The priority queue is initialized by taking into account the difficulty to schedule each task by its own. We defined two heuristic metrics to measure this difficulty. The first one takes into account the net size of the domain of each task.

Definition 1 (net size): The net size $net(D)$ of a domain D consisting of a set of intervals is defined as the sum of the widths of these intervals.

For example, for a task T_i having domain $D_i = [0,12] \cup [20, 35]$, we have $net(D_i)=(12-0)+(35-20)=27$.

Definition 2 (metric m_1): Metric m_1 of a task T_i is defined as the ratio between the total duration of the task and the net size of its domain, i.e.:

$$m_1(T_i)=dur_i/net(D_i)$$

Suppose that task T_i has duration $dur_i=15$, then $m_1(T_i)=15/27=0.55$. Note that intervals whose widths are smaller than $smin_i$ are removed from D_i. Metric m_2 takes into account the minimum distance $dist_i$ between any two parts of each interruptible task T_i.

Definition 3 (makespan of a task): We define as $makespan(T_i)$ of a task T_i the distance between the start time of the earliest scheduled part of the task and the end time of the latest scheduled part of the task, i.e.:

$$makespan(T_i)=max_j(t_{ij}+dur_{ij})-min_k(t_{ik})$$

Definition 4 (width of a domain): The width $width(D_i)$ of a domain D_i is defined as the distance between the left end of the leftmost interval and the right end of the rightmost interval, i.e.:

$$width(D_i)=max_j B_{ij}-min_k A_{ik}$$

In the running example we have $width(D_i)=35-0=35$.

Definition 5 (metric m_2): Metric m_2 of a task T_i is defined as the ratio between the minimum possible makespan of the task and the width of its domain, i.e.:

$$m_2(T_i)=min(makespan(T_i))/width(D_i)$$

For the task of the running example, suppose that $smax_i=6$, $smin_i=3$ and $dmin_i=4$. Since $smax_i<dur_i$, this task has to be executed in parts. The minimum number of parts is $min_parts_i=\lceil dur_i/smax_i \rceil=3$. This implies that $dmin_i$ will appear at least two times within the makespan of T_i, so a lower bound for the makespan of T_i can be computed as $15+2\cdot4=23$. So, $m_2(T_i)= 23/35=0.657$.

Definition 6 (difficulty of a task): The difficulty of a task T_i is defined as the maximum between m_1 and m_2, i.e.:

$$diff(T_i)=max(m_1(T_i), m_2(T_i))$$

In our example, $diff(T_i)= \max(0.55, 0.657)=0.657$.

Non-interruptible tasks are evaluated only by m_1, since for these tasks $m_1 \geq m_2$ always hold. Note also that both metrics are admissible, so in case any one of them has a value greater than 1, the task cannot be scheduled.

To conclude, the priority queue is initialized as follows:

INITIALIZATION OF THE PRIORITY QUEUE

1. Apply full constraint propagation for all constraints.
2. Compute the difficulties of all tasks.
3. Place the tasks in the priority queue in decreasing order of their difficulty.

Greedy construction algorithm

Tasks are scheduled one after the other in specific start times, in the order imposed by the priority queue. To simplify our presentation, we suppose that the order of the tasks in the priority queue coincides with their numbering, i.e. task T_i is in the i-th position of the priority queue. Now suppose that tasks T_1 to T_{i-1} have already been scheduled and next is the case for T_i.

While scheduling tasks T_1 to T_{i-1}, several parts of the domains of the remaining N-i+1 tasks may have been removed, due to the propagation of the various constraints. In order to schedule T_i, we consider its current domain, as well as the current domains of the remaining tasks. This is done as follows:

Let's first suppose that all tasks are non-interruptible; we generalize later for interruptible tasks. So, for each location $l_i \in Loc_i$ where T_i could take place and for each possible start time (dependent on the location l_i) where T_i could be scheduled, we compute the impact on the difficulty to schedule the remaining tasks. This requires two steps: First, propagating all the constraints to the domains of the remaining tasks. Second, computing for each one of the remaining tasks the difficulty to schedule it. Note that while propagating the constraints, each one of the remaining tasks is considered to be executed in the closest to l_i available location, i.e. there is no need to examine all the possible locations for the remaining tasks. Then, the combination $\langle l_i,t_i \rangle$ for the current task that results in the lower overall difficulty to schedule the remaining tasks is adopted (in case of ties, earlier start times are preferred):

Definition 7 (Overall difficulty): The overall difficulty to schedule a set of tasks S, denoted with $overall(S)$, is defined as the product of their individual difficulties:

$$overall(S)=\prod_{T_i\in S} diff(T_i)$$

Example 8: Suppose for example that we have three non-interruptible tasks, A, B and C in this order in the priority queue, with durations 2, 1 and 3 respectively, locations $Loc_A=\{X,Y\}$, $Loc_B=\{X\}$, $Loc_C=\{Y\}$, $Dist(X,Y)=Dist(Y,X)=2$, the ordering constraint $\prec(A,B)$ and their current domains shown in Figure 2.

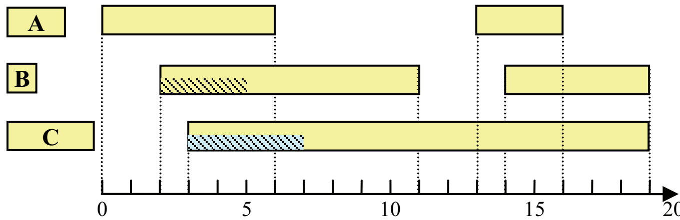

Figure 2. An example of scheduling tasks. Colored areas denote the current domains of the tasks, before scheduling task A. Shaded areas are excluded from their domains after placing task A at t_A=3 and l_A=X.

There are seven possible start times to schedule A, i.e. $t_A\in\{0,1,2,3,4,13,14\}$, and two possible locations, i.e. X and Y. Each combination of start time and location has to be evaluated and the best one will be selected. Let's take for example the case $t_A=3$ and $l_A=X$. By applying bounds consistency we get $t_B\geq 5$. Moreover, since A and C cannot coincide and must have a distance of at least 2, the interval between 3 and 7 is removed from C's domain. Difficulties are computed as $m_1(B)=1/11$ and $m_1(C)=3/12$ respectively. The overall difficulty is then $overall(\{B,C\})=0.0227$. After testing all the alternative possibilities to schedule A, we conclude that the best case is for $t_A=0$ and $l_A=X$, with $overall(\{B,C\})=0.014286$.

In case the difficulty to schedule some of the remaining tasks for a particular option to schedule the current task exceeds 1, this option is never adopted, irrelevant to its score. However, if no feasible schedule for the current task can be found, the current task is considered a trouble maker and is promoted in the priority queue, as explained later in this section. The process continues from the new position of the current task.

Interruptible tasks need special treatment when they are current: Parts are scheduled one after the other, as if they were non-interruptible. However, for each part an additional decision has to be taken, concerning its duration. So, for each possible part duration $dur_{ij}\in[smin_i, smax_i]$ and for each possible time/location combination when/where this part could be scheduled, the overall difficulty to schedule the remaining duration of the current task as well as the remaining tasks is computed. Then, the best combination of part duration/time-window/location is selected. In case of ties, higher durations are preferred. This process continues until either the whole duration of the interruptible task has been scheduled, or we fail to schedule some part. In the latter case the current task is considered a trouble maker and is promoted in the priority queue.

Computing the valid part durations: Depending on the values of dur_i, $smax_i$ and $smin_i$, several durations for dur_{ij} may not be allowed. For example, if $dur_i=12$, $smax_i=8$ and $smin_i=5$, it is not possible to schedule a part of duration 8, since in this case the next part to be scheduled should have a maximum duration of 4, which violates the constraint of $smin_i$. This situation does not necessarily indicate bad problem definition; indeed, dur_i might be the remaining duration to be scheduled, after already having scheduled several parts of the current task. Another more strange situation is when $dur_i=6$, $smax_i=6$ and $smin_i=3$. In this case the only valid part durations are 3 and 6.

Identifying the valid part durations to be scheduled for an interruptible task is crucial, since it does not only preserve computation time, but it also prevents us from unnecessary failures that would be detected later. In order to compute the valid durations for the next part to be scheduled, we use the following approach:

Given dur_i, $smin_i$ and $smax_i$, with dur_i being the remaining duration to be scheduled of task T_i, any duration dur_{ij}, such that $smin_i\leq dur_{ij}\leq smax_i$ is allowed for the next part, if for $L=\lfloor(dur_i - dur_{ij})/smin_i\rfloor$, the following condition exists:

$$L\cdot smin_i\leq dur_i-dur_{ij}\leq L\cdot smax_i \qquad (2)$$

In the first example, with $dur_i=12$, $smax_i=8$ and $smin_i=5$, if $dur_{ij}=8$ then $L=\lfloor(12-8)/5\rfloor=0$, so condition (2) does not hold. For $dur_{ij}=7$ we have $L=\lfloor(12-7)/5\rfloor=1$ and (2) holds with equality to the left. Similarly, for $dur_{ij}=5$ we have $L=\lfloor(12-5)/5\rfloor=1$ and (2) holds with inequality to both directions. In the second example, with $dur_i=6$, $smax_i=6$ and $smin_i=3$, for $dur_{ij}=6$ we have $L=\lfloor(6-6)/3\rfloor=0$ and (2) holds with equality to both directions. Similarly, for $dur_{ij}=3$ we have $L=\lfloor(6-3)/3\rfloor=1$ and (2) holds with equality to the left. Finally, for all other values of dur_{ij} we have $L=\lfloor(6-5)/3\rfloor=0$ and the right inequality of (2) does not hold.

Lemma 9: If part durations are chosen according to (2), then the whole duration of T_i can be covered.

Proof: Suppose we are given dur_i, $smin_i$ and $smax_i$, with dur_i being the remaining duration of task T_i to be scheduled, and we select dur_{ij} such as condition (2) holds. Suppose that $dur_i-dur_{ij}=L\cdot smin_i+K$, with $0\leq K\leq L\cdot(smax_i-smin_i)$. Let $K_1=K$ div $(smax_i-smin_i)$ and $K_2=K$ mod $(smax_i-smin_i)$, where div and mod denote integer division and modulo respectively. The relation between K, K_1 and K_2 is $K=K_1\cdot(smax_i-smin_i)+K_2$. So, one possibility to schedule the remaining dur_i-dur_{ij} time units is to schedule K_1 parts of duration $smax_i$, one part of duration $smin_i+K_2$ and $L-K_1-1$ parts of duration $smin_i$. In this case we have scheduled $(L-K_1-1)\cdot smin_i+smin_i+K_2+K_1\cdot smax_i=L\cdot smin_i+K_1\cdot(smax_i-smin_i) + K_2 = L\cdot smin_i+K=dur_i-dur_{ij}$.

Lemma 9 follows by applying inductively the above one step proof until the whole duration of T_i is scheduled. □

Lemma 10: Taking into account the minimum and maximum part durations, $smin_i$ and $smax_i$ respectively, the maximum and minimum possible number of parts in which an interruptible task with duration dur_i can be split are $\lfloor dur_i/smin_i \rfloor$ and $\lceil dur_i/smax_i \rceil$ respectively.

Proof: The proof is similar to that of Lemma 9. □

Reordering and Memorization

Reordering of the tasks in the priority queue occurs each time the greedy construction algorithm fails to schedule the current task in a way such that, after having applied full constraint propagation, the difficulty of every one of the remaining tasks does not exceed 1. In this case, the current task is considered a trouble maker and is promoted in the priority queue. Two issues arise: How far to promote the current task and how to avoid cycles.

Concerning promotion, there are several approaches. The simplest one is to promote the trouble maker task by one position. This approach is easy to implement, but may perform unnecessary computations in case the previous task was totally irrelevant to the current one. In our implementation of SWO we adopted a greedy approach for promotion, which after thorough experimentation, has been proved significantly efficient: The trouble maker task is promoted in the first position of the priority queue, with the intermediate tasks being shifted one position backwards. This strategy is quite powerful for two reasons: First, it causes drastic reordering of the priority queue, thus increasing the probability to escape local optima; and second, it minimizes the probability to enter into loops, i.e. to revisit the same ordering in the priority queue.

More elaborated approaches to promote the trouble maker could exploit dynamic information generated while scheduling the tasks in the current order. For example, the trouble maker could be promoted just before the closest preceding task which was responsible for canceling at least one scheduling possibility of the trouble maker; or just before all tasks that removed some scheduling possibilities. Such strategies however require extensive bookkeeping, whereas the strategy we eventually adopted is simple enough and performs well.

Irrelevant to what promotion strategy is adopted, it is always useful to partition the tasks into clusters of related tasks, using the transitive closure of the constraint graph. Suppose for example the case that the only constraint between tasks is C6, i.e. the constraint that no two tasks can overlap in the final schedule. Then morning tasks may form a cluster, evening tasks another one and weekend tasks a third one. The identification of such clusters is crucial, since the scheduling problem can be solved separately for each cluster. Of course, the existence of a task whose domain spans over morning, evening and weekends would merge all clusters.

Especially for the constraint C6, two tasks are potentially overlapping (so they are connected in the constraint graph) according to the following definition:

Definition 11 (potentially overlapping tasks): Two tasks T_i and T_j are potentially overlapping if there is a pair of locations $l_i \in Loc_i$ and $l_j \in Loc_j$, as well as a pair of domain intervals $[A_{im},B_{im}] \in D_i$ and $[A_{jn},B_{jn}] \in D_j$, such that either $B_{im}+Dist(l_i,l_j)>A_{jn}$ or $B_{jn}+Dist(l_j,l_i)>A_{im}$ hold.

Definition 11 captures the case that scheduling one task may cancel some scheduling possibility of the other task due to the time needed by the user to travel between the two locations. Note that this cancellation might not concern the domain intervals themselves, but some possible location references for these intervals.

Finally, in order to avoid cycles, memorization of no-good prefixes of the priority queue is required. Then, each time a promotion occurs, the new ordering of the priority queue should be checked against the memorized no-good prefixes. In case a match is detected, either further reordering should take place, or the problem should be claimed unsolvable. With our adopted greedy promotion strategy, loops in the priority queue occur rarely. However, in case of a detected loop, we proceed with promoting the tasks following the trouble maker. If no such tasks occur, or if they lead to orders in the priority queue that have been already checked, the problem is claimed unsolvable.

Managing Personal Tasks with Preferences

Time preferences can naturally be incorporated into the SWO framework. Our approach allows the user to express any kind of time preference between subsets of tasks, provided that he can express an estimate about the final value of each preference as a function of the involved tasks' domains. More formally:

Definition 12 (Time Preference): A time preference v over a set of tasks S is a real valued function $v:\prod_{T_i \in S} D_i \rightarrow \mathbb{R}$, higher values being preferred.

Definition 12 states that in order to define a time preference, we need to provide a function that maps any combination of domains of the involved tasks to a real number. This function will be used, together with the difficulty heuristics, while scheduling the tasks of the priority queue. Definition 12 works both for fully instantiated and for partially instantiated sets of tasks. For a fully instantiated set of tasks, D_i's will refer to the actual time-window(s) where each task T_i has been scheduled. For partially instantiated tasks, several approaches could be adopted in order to map (the combination of) their domains into a real value; for example, the maximum possible value of the preference wrt the current domains is one such possibility. Note finally that function v usually takes into account other constant parameters of the involved tasks, such as their duration, $smin_i$, $smax_i$ etc.

Example 13 (unary time preference): Suppose a unary preference v_i over a task T_i, with $D_i=[0,10]$ and $dur_i=2$. We may define v_i by assigning values to the various time slots of D_i and then sum the values of the slots where T_i actually has been scheduled, divided by dur_i. For example, suppose

that we assign value t+1 to each unit time slot starting at t, then the value received from executing task T_i at time t is, $v(t)=\sum_{t}^{t+dur_i-1}(t+1)/dur_i$. So if T_i will be scheduled at t_i=3, the received value is $v_i(t_i=3)=(4+5)/2=4.5$.

Suppose now that T_i has not yet been scheduled and that its domain has reduced to $D_i'=[0,4]\cup[7,10]$. What could be an estimate about the final value obtained by v_i? The simplest approach is to consider the maximum possible value of $v_i(t)$ for the various time windows where T_i could be scheduled. In this case this happens for t=8, so: $v_i(D_i')=max_t v_i(t)=v_i(8)=9.5$ □

Example 14 (binary time preference): Suppose two non-interruptible tasks T_i and T_j and a binary preference over them defined as $v_{ij}(D_i,D_j)=max_{ti\in Di,tj\in Dj}(t_j-t_i-dur_i,\ t_i-t_j-dur_j)$. In other words, v_{ij} is defined as the maximum possible distance between the two tasks in the final schedule. Suppose for example that $D_i=[0,5]\cup[20,30]$, $D_j=[10,16]$, dur_i=3 and dur_j=4. The maximum possible distance between the two tasks is achieved if T_i is scheduled at 27 and T_j at 10 and the received value is $v_{ij}(D_i',D_j')=13$.

In case of interruptible tasks a similar approach can be adopted, just by trying to maximize the minimum distance between any pair of parts of the two tasks. Finally, a similar approach can be used in case we are trying to minimize the temporal distance between two tasks: Replace *max* with *min* and vice-versa, and try to maximize the negated minimum distance. □

Incorporating time preferences in the SWO framework is quite natural. Preferences can be taken into account together with the difficulty to schedule the set of the remaining tasks S. Indeed, for each task to be scheduled, the following ratio has to be minimized:

$$\frac{\left(overall(S)\right)^a}{\left(\sum_{v_k\in V} v_k\right)^b} \quad (3)$$

Non-negative parameters a and b in (3) are used as relative weights between the objective to receive a feasible schedule quickly and the objective to receive a high-value schedule. As the ratio a/b becomes higher, priority is given to receive a feasible schedule quickly. As the ratio a/b becomes smaller, priority is given to receive a good schedule. In the extreme case where $a/b\to\infty$, preferences are ignored, whereas as the ratio a/b tends to zero, the choices of SWO are based solely on the preferences.

Different values of the ratio a/b lead to different scheduling decisions during the application of the SWO algorithm. So, changing the ratio a/b is a way to alter its behavior. A simple policy for changing this ratio is the following: Start with a low value for a/b. In case of failing to find a schedule, increase a/b and start again. In case of success, decrease a/b, thus trying to find a better schedule. Note that each time the algorithm restarts with a new value for the ratio a/b, the previous set of no-goods is deleted.

Example 15: In Example 8, suppose that the following unary time preferences are given:

$$v_A(t)=\sum_{t}^{t+1}(t+1)$$

$$v_B(t)=\begin{cases}6, \text{if } t<12\\ 3, \text{if } t\geq 12\end{cases}$$

$$v_C(t)=\sum_{t}^{t+2}2t$$

Moreover, we have a binary distance time preference between A and B of the form $v_{AB}(t_A,t_B)=max(t_B-t_A-dur_A,\ t_A-t_B-dur_B)$, i.e. tasks A and B should be scheduled as far to each other as possible. We assume also that $a=b=1$. Let us evaluate the possibility to schedule A at t_A=3 and $l_A=X$. For this case we have already computed the difficulty to schedule the remaining tasks, which is 0.0227. Now we will compute also the preferences. As for task A we have $v_A(t=3)=9$. As for task B, the maximum of $v_B(t)$ over its remaining domain (not shaded in Figure 2) is 6. Finally, as for task C we have $max_t v_c(t)=v_c(16)=2\cdot(16+17+18)=102$. Finally, as for the binary time preference v_{AB} we have $v_{AB}(t_A,t_B)=max(t_B-t_A-dur_A,\ t_A-t_B-dur_B)=max(t_B-3-2,\ 3-t_B-dur_B)=13$. So, the ratio of (4) becomes:

$$\frac{0{,}0227}{9+6+102+13}=1{,}75\cdot 10^{-4}$$

Doing the same calculations for the various scheduling possibilities of A results that the best case is to schedule task A at t=0, and $l_A=X$ with an overall score equal to:

$$\frac{0{,}014286}{3+6+102+16}=1{,}12\cdot 10^{-4} \quad \square$$

Other types of preferences, beyond time preferences, could be defined. For example, one could express location preferences, or, even more complicated, mix time and location preferences, e.g. preferring location l_1 if the task is scheduled during the five working days of a week, or location l_2 if the task is scheduled during the weekend. Various engineering issues arise concerning these types of preferences, such as how to define them in a user-friendly way, as well as to incorporate them within the SWO framework; these issues constitute future research challenges.

Incremental scheduling

Scheduling personal time is a continuous process. Time passes quickly, early tasks are executed and new tasks arrive. So, the scheduling problem must be solved repeatedly each time a new set of tasks arrives. An incremental scheduling approach would be beneficial for both efficiency and usability reasons. Indeed, each time a set of new tasks arrives, the current schedule should be taken as a basis while trying to schedule the new tasks, thus saving computation time and minimizing the user disturbance from frequent changes in her personal plan. SWO is very suitable for incremental scheduling, as the following paragraphs illustrate.

In order to adapt SWO to an incremental scheduling approach, we apply four modifications:

- The tasks of the priority queue are divided into two sets, the already scheduled *old tasks* and the *new tasks*, for which scheduling is pending.
- New tasks are initially placed at the end of the priority queue, in decreasing order of their difficulty.
- In order to schedule each one of the old tasks, the greedy construction algorithm considers only the part of the remaining tasks until the last old one.
- In case new tasks are involved in user preferences, it is an option for the user whether incremental scheduling or scheduling from scratch will be applied.

Suppose for example that the current priority queue order is $\langle A,B,C,D \rangle$, with a schedule already found for the four tasks, when the new tasks *E* and *F* arrive. The new tasks are initially placed at the end of the priority queue, so the new priority queue is $\langle A,B,C,D,E,F \rangle$. When applying SWO for first time with the new set of tasks, the scheduling decisions for *A*, *B*, *C* and *D* do not change, since the greedy construction algorithm does not take into account tasks *E* and *F* when computing the overall difficulty. Suppose now that task E is unable to schedule in this order, so it is promoted and the new priority queue order becomes $\langle E,A,B,C,D,F \rangle$. Now, *E* is scheduled first thus affecting scheduling decisions for the old tasks; on the contrary, task *F*, which is beyond the last old task, is not taken into account while scheduling the old tasks; however, *F* is taken into account while scheduling the new task *E*, i.e. the difficulty to schedule *F* is considered by the greedy construction algorithm when deciding when to schedule *E*.

An issue arising in incremental scheduling is the introduction of new preferences along with the new tasks. In this case, incremental scheduling might result in a lower quality schedule compared to the schedule that would result by scheduling from scratch. Several engineering solutions could be adopted in this case, such as never perform incremental scheduling in case of new preferences, or perform both incremental and non-incremental scheduling and let the user decide.

Another issue that arises in incremental scheduling concerns the executed tasks. If a task is executed and thus removed from the priority queue, it is possible (although least probable) that if SWO is applied to the remaining tasks, their schedule might change. To avoid such unnecessary changes, we do not reapply SWO when tasks are executed (except if manually asked by the user) but only when new tasks arrive. However, this issue may arise even when scheduling new tasks. In the aforementioned example, suppose that when the new tasks E and F arrive, task B has already been executed. In this case, the new ordering $\langle A,C,D,E,F \rangle$, with *D* being the last old task, does not ensure that the schedule for *A*, *C* and *D* will be identical to the schedule that had resulted by the prefix $\langle A,B,C,D \rangle$. Nevertheless, this is a marginal situation that arises rarely, with changes being minor and not in the immediate future, so it is expected not to be annoying; otherwise, engineering solutions such as locking the proximate tasks could be adopted.

Experimental Results

We implemented our SWO algorithm in C++ and tested it in a set of artificially created problems. We considered tasks with durations uniformly selected from the interval [4, 12]. The 60% of the tasks with durations greater than 5 were interruptible. In this case, we considered *dmin*=8, *smin*=2 and *smax* uniformly selected from the interval [3, *dur* - 2]. We considered three locations, randomly assigned to the tasks, with symmetric mutual distances 3, 4 and 5. We chose three alternative configurations for the *a*/*b* ratio: 1/5, 1 and 5. Starting from the 1/5 configuration, SWO performed *N* iterations at maximum in each case, where *N* is the number of tasks. Between each pair of tasks T_i and T_j, i<j, an ordering constraint of the form $<(T_i,T_j)$ was imposed, with probability 1/*N*. We randomly assigned unary preference functions over the domains of all tasks, of the following types: constant, linear ascending, linear descending, step ascending and step descending. All preferences were normalized in the interval [0,1], so an upper bound for the value received by a schedule is the number of the tasks. We set a deadline of 500 time units for all problems and created random non-compact domains for all tasks. We created seven sets of problems, with 15, 30, 35, 40, 45, 50 and 55 tasks per problem in each set respectively and 10 problems per set in all cases. All problems were generated randomly (a problem generator is supplied along with the test set for cross-validation).

To compare with, we developed an alternative approach to solve the same problems, based on the constraint-logic programming platform (CLP) ECLiPSe[3]. We modeled the problems using a combination of normal and reified constraints and we employed a complete branch-and-bound search procedure with the most constrained variable selection heuristic and the domain splitting method to solve them. In order to optimize the CLP program, we introduced additional constraints in the model, so as to avoid generating redundant solutions (e.g. $dur_{ij} \geq dur_{i,j+1}$ for all $1 \leq i \leq N$, $1 \leq j \leq p_i$). We set a time limit of 2 mins for the branch-and-bound algorithm to report the best possible solution within this time limit.

All experiments were run on an MS-Windows based AMD Turion, 64 bit, 2 GHz dual core machine, with 2GB memory. Table 1 summarizes the results[4]. Problem ids are of the form XX-Y, where XX corresponds to the number of tasks in the problem. Problems with 55 tasks are not reported, since none of them was solved by either solver.

For each problem columns *iter*, *time* and *value* for SWO report the number of iterations performed by SWO, the time needed to solve the problem (in secs) and the value of the solution respectively. Column *value* for CLP reports the best solution found by the CLP program; if no value is reported, no solution was found within the time limit.

It is interesting to observe the behavior of SWO. In the

[3] http://eclipse.crosscoreop.com/.

[4] The source code and the data of our experiments are available from the author's web-page, http://eos.uom.gr/~yrefanid.

#	SWO			CLP	#	SWO			CLP	#	SWO			CLP
	iter	time	value	value		iter	time	value			iter	time	value	time
15-1	1	0.14	12.95	11.34	35-1	1	0.625	28.80	27.85	45-1	1	0.64	37.42	35.89
15-2	1	0.125	12.25	10.81	35-2	1	0.375	29.165	×	45-2	2	2.031	33.97	×
15-3	1	0.15	13.71	11.95	35-3	1	0.797	27.842	×	45-3	21	16.17	35.44	×
15-4	1	0.188	11.57	10.88	35-4	1	0.547	26.64	24.11	45-4	43	29.54	33.02	×
15-5	1	0.125	12.64	12.57	35-5	2	1.032	25.15	25.01	45-5	66	51.79	30.83	×
15-6	1	0.125	14.3	13.88	35-6	1	0.718	26.12	23.14	45-6	1	0.656	32.70	×
15-7	1	0.125	13.08	12.22	35-7	1	0.484	29.28	27.47	45-7	1	0.828	32.40	33.71
15-8	1	0.187	11.46	9.79	35-8	1	0.578	25.71	26.89	45-8	110	74.032	31.79	×
15-9	1	0.141	11.44	8.29	35-9	2	0.906	23.74	21.20	45-9	1	0.797	35.79	×
15-10	1	0.172	12.07	9.37	35-10	1	0.406	30.70	×	45-10	1	0.75	32.78	×
30-1	1	0.368	24.17	21.55	40-1	9	5.36	24.72	×	50-1	26	20.157	42.04	×
30-2	1	0.399	24.69	19.68	40-2	1	1.002	23.48	×	50-2	150	141	×	×
30-3	1	0.386	25.61	21.81	40-3	1	0.711	33.57	×	50-3	106	104	×	×
30-4	1	0.316	27.13	25.10	40-4	1	0.75	31.46	29.21	50-4	150	141	×	×
30-5	1	0.413	23.89	21.00	40-5	1	0.619	28.05	29.14	50-5	2	1.46	34.25	×
30-6	1	0.433	28.09	24.14	40-6	3	2.1	29.46	27.03	50-6	36	24.96	38.32	×
30-7	1	0.339	23.8	18.10	40-7	2	1.28	33.13	×	50-7	2	1.328	32.59	×
30-8	1	0.415	24.06	20.73	40-8	1	0.691	29.72	26.78	50-8	24	16.26	34.70	×
30-9	1	0.447	23.42	18.95	40-9	1	0.556	33.03	×	50-9	150	127	×	×
30-10	1	0.461	22.04	21.00	40-10	3	2.23	30.28	25.33	50-10	24	17.5	37.46	×

Table 1: Experimental results for SWO and comparison to a complete CLP branch-and-bound search procedure.

easier problems (up to 35 tasks), all problems were solved in just one or two iterations. For harder problems with 40 tasks, SWO needs more iterations in many cases, but it does not increase the ratio *a*/*b*. For even harder problems, i.e. 45 tasks or more, significantly more iterations are needed, whereas now the ratio a/b is frequently increased to the values 1 or 5 (e.g. problems 45-5, 45-8).

Comparing to the complete CLP search procedure, we have two comments: First, SWO solved all problems solved by CLP in the specified time limit. In many cases, CLP was unable to find a solution, whereas SWO found one in just a single iteration. This demonstrates the power of the difficulty measures and the reordering strategy employed in SWO. However it is interesting that SWO solved many of the really difficult problems, as those with 45 tasks. This makes it reliable for using in real-world applications. Finally, concerning the value of the schedules returned by the two algorithms, SWO found better schedules in all but three cases (35-8, 40-5 and 45-7)

Conclusions and Future Work

This paper treats the problem of automatically scheduling a user's tasks, while trying to maximize her satisfaction. New types of tasks and constraints are introduced, powerful heuristics are devised and an SWO incomplete search procedure with full constraint propagation is employed to solve the problem. Experimental results demonstrate the effectiveness and the efficiency of this approach.

This work constitutes a first step in coping with the more general problem of managing personal tasks. There are many directions to extend this work, such as:

- introduction of new types of tasks, constraints and preferences,
- coordination between different users,
- enhancement of tasks with preconditions and effects and employment of a plan engine in order to automatically introduce tasks into the user's agenda,
- maintenance of the user status,
- automated accomplishment of tasks using web services.

In particular, we see our system as a module of highly-integrated electronic assistants, such as CALO (Myers and Yorke-Smith, 2005).

One of our immediate plans is to integrate the work presented in this paper with existing calendar applications in order to provide additional functionality to the users. Several issues concerning usability and knowledge engineering arise. Finally, we are investigating new types of constraints, preferences and heuristics, as well as the possibility to combine the heuristics with a complete search procedure, such as dynamic backtracking (Ginsberg 1993), in order to take the best of each side.

References

Ginsberg, M.L., 1993. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25-46.

Joslin, D.E., Clements D.P.: “Squeaky Wheel” Optimization. J. of Artificial Intelligence Res, vol. 10 (1999), 375-397.

Le Pape, C., Baptiste, Ph.: Constraint Propagation Techniques for Disjunctive Scheduling: The Preemptive Case. 12th European Conf. on Artificial Intelligence, 1996.

Myers K. and Yorke-Smith N. A Cognitive Framework for Delegation to an Assistive User Agent. Proc. of AAAI 2005 Fall Symp. on Mixed-Initiative Problem Solving Assistants.

Refanidis, I., McCluskey, T.L., Dimopoulos, Y.: Planning Services for Individuals: A New Challenge for the Planning Community. Workshop on Connecting Planning Theory with Practice, Whistler, British Columbia, Canada, 2004.

Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation and evaluation of the constraint language cc(fd), J.of Logic Programming, 37 (1998), pp. 139-164.