1. Introduction

Activity scheduling is an important aspect in various domains, like business processes, industrial workflows, distributed systems, and so on. Scheduling the activities efficiently can bring multiple benefits, like minimizing costs, maximizing profits and/or throughput or optimizing the social welfare of the employees. In this paper we consider several constrained time and space activity scheduling problems, for which we present efficient algorithms for computing optimal schedules. Although the considered problems are mostly tackled from a theoretical point of view, they have applications in some of the domains mentioned above, particularly those related to economic activities and computer science.

The rest of this paper is structured as follows. In Sections 2-5 we present novel algorithmic solutions for several activity scheduling problems over time. In Sections 6 and 7 we consider two space scheduling problems, for which we present efficient algorithms for computing optimal schedules. We will introduce into the first pair (value=Smax(0,j-1), moment=i-Lo(j)), we will maintain a deque DQ, into which we will introduce (value, moment) pairs. These pairs will be maintained sorted decreasingly according to the value and increasingly according to the moment. The functions DQ.getFirst() and DQ.getLast() (DQ.removeFirst() and DQ.removeLast()) will be used for retrieving (removing) the first and last pair of (from) the deque (if DQ is not empty). When we compute the values Smax(*,j), we will introduce into DQ the first pair (value=Smax(0,j-1), moment=0). Then, we traverse the moments i=1,...,N, in increasing order. When we reach a moment i, we perform the following actions. While DQ.getFirst().moment<i-Up(j), we call DQ.removeFirst(). Then, we compute Scand=Smax(i-Lo(j),j-1)-SP(i-Lo(j)) and we set Sbest=max(Scand, Sbest). We will have Smax(i, j) = max{Smax(i-1, j), Sbest+SP(i)}. This way, the case Up(j)=N (for all i≤j≤K) can be solved in O(N·K) time.

In order to solve the general case, we will proceed as follows. When we compute the values Smax(*,j), we will maintain a deque DQ, into which we will introduce (value, moment) pairs. These pairs will be maintained sorted decreasingly according to the value and increasingly according to the moment. The functions DQ.getFirst() and DQ.getLast() (DQ.removeFirst() and DQ.removeLast()) will be used for retrieving (removing) the first and last pair of (from) the deque (if DQ is not empty). When we compute the values Smax(*,j), we will introduce into DQ the first pair (value=Smax(0,j-1), moment=0). Then, we traverse the moments i=1,...,N, in increasing order. When we reach a moment i, we perform the following actions. While DQ.getFirst().moment<i-Up(j), we call DQ.removeFirst(). Then, we compute Scand=Smax(i-Lo(j),j-1)-SP(i-Lo(j)) and we set Sbest=max(Scand, Sbest). We will have Smax(i, j) = max{Smax(i-1, j), Sbest+SP(i)}. This way, the case Up(j)=N (for all i≤j≤K) can be solved in O(N·K) time.

3. Constrained Scheduling of K Activities over Time in order to Maximize the Total Utility

This problem is identical to the previous one, except that every activity j (1≤j≤K) must necessarily contain the
special time moment \( p(j) \) \( (p(1)<...<p(K)) \). We notice that the time moments in intervals of the form \([p(j), p(j+1)-1]\) \( (1 \leq j \leq K) \) can be the rightmost time moment only of activity \( j \)'s interval (we consider \( p(K+1) = N+1 \) and \( p(0)=0 \). We will assign \( a(t)=j \) to every time moment \( t \) in the interval \([left(j)=p(j), right(j)=p(j+1)-1]\) (in \( O(N) \) time). Then, we can use dynamic programming and compute \( S_{\text{max}}(i) \) the maximum sum of the utilities if the first \( a(i) \) activities have been scheduled and the rightmost moment of activity \( a(i) \)'s time interval is smaller than or equal to \( i \). We have \( S_{\text{max}}(0) = S_{\text{max}}(1) = 0 \). Then, we will compute all the values \( S_{\text{max}}(i) \) for the time moments \( i \) such that \( a(i) \) is joint, in increasing order of \( j \) \( (j=1,...,K) \). Like before, we compute the partial sums \( S_{\text{max}}(*) \), with which we can evaluate \( S_{\text{max}}(a,j) \) in \( O(1) \) time.

We will first consider the case when all the values \( U(p(j)) \) \( (i.e. \) there are no upper bounds). In this case, when we reach the value \( j \), we will compute the values \( V_{\text{max}}(p(j)) = S_{\text{max}}(p(j)-1)-SP(p(j)) \) and \( V_{\text{max}}(p(j)+1) = \max\{V_{\text{max}}(p(j)-1), S_{\text{max}}(j)-SP(j)\} \). Afterwards, we consider all the values \( S_{\text{max}}(i) \leq j <(j+1) \), in increasing order of \( i \). If \( j = \text{Lo}(j) + 1 \) then \( S_{\text{max}}(i) = -\infty \) \( (\text{if } i = \text{prev}(j)) \) or \( S_{\text{max}}(i) \) \( (\text{if } i> \text{prev}(j)) \). Otherwise, let \( \text{prev}(i) = \min\{\text{Lo}(j), j\} \). If \( i = \text{p}(j) \) then \( S_{\text{max}}(i) = \max\{S_{\text{max}}(\text{prev}(i)), S_{\text{max}}(i-1)\} \). This case can be handled in \( O(N) \) time. If, instead, we have \( U(p(j)) \) \( (\text{for every } 1 \leq j \leq K) \), when considering the values \( p(j) \leq j <(j+1) \), we have: if \( \text{Lo}(j) = j \) \( (\text{if } i = \text{prev}(j)) \) or \( i = \text{Lo}(j) + 1 \) \( (\text{if } i> \text{prev}(j)) \) then \( S_{\text{max}}(i) = -\infty \) \( (\text{if } i = \text{prev}(j)) \) or \( S_{\text{max}}(i) \) \( (\text{if } i> \text{prev}(j)) \). Otherwise, let \( \text{prev}(i) = \min\{\text{Lo}(j), j\} \). If \( i = \text{p}(j) \) then \( S_{\text{max}}(i) = \max\{S_{\text{max}}(\text{prev}(i)), S_{\text{max}}(i-1)\} \). In order to handle the general case, we will proceed as follows. When we compute the values \( S_{\text{max}}(p(j)) \) \( (p(j) \leq j <(j+1)) \), we will maintain a deque \( DQ \), into which we will introduce \( (value, moment) \) pairs. These pairs will be maintained sorted decreasingly according to the value and increasing according to the moment. We will use the functions \( DQ.\text{getFirst}() \), \( DQ.\text{getLast}() \), \( DQ.\text{removeFirst}() \) and \( DQ.\text{removeLast}() \) (defined in the previous section). Based on these functions, we define the function \( DQ.\text{insert}(val, mom) \) as follows: \( (1) \) while \( DQ \) is not empty and \( DQ.\text{getFirst}().value = val \) do \( DQ.\text{removeLast}() \); \( (2) \) add the pair \( (val, mom) \) at the end of \( DQ \). For each position \( j \) \( (i.e. j = a(i)) \), we define \( \text{l}(i) = \max\{i-\text{Up}(j), j\} \) and \( \text{th}(i) = \min\{i-\text{Lo}(j), p(j)\} \). When we reach a new value of \( j \), we empty the deque \( DQ \). Then, we consider all the time moments \( \text{lo}(p(j)) \leq j \leq \text{th}(p(j)) \) and call \( DQ.\text{insert}(value = S_{\text{max}}(i)-SP(i), \text{moment} = t) \). Afterwards, if \( DQ \) is empty then \( S_{\text{max}}(p(j)) = -\infty \); otherwise, we set \( S_{\text{max}}(p(j)) = DQ.\text{getFirst}().value + SP(p(j)) \). For \( p(j) + 1 \leq j \leq p(j+1)-1 \) (in increasing order of \( i \), we perform the following actions: \( (1) \) for every time moment \( t \) with \( \text{th}(i-1) + 1 \leq t \leq \text{th}(p(j)) \) we call \( DQ.\text{insert}(value = S_{\text{max}}(t)-SP(t), \text{moment} = t) \); \( (2) \) while \( DQ \) is not empty and \( DQ.\text{getFirst}().moment < \text{l}(i) \) do \( DQ.\text{removeFirst}() \). If \( DQ \) is empty, then \( S_{\text{max}}(i) = S_{\text{max}}(i-1) \); otherwise, \( S_{\text{max}}(i) = \max\{S_{\text{max}}(i-1), DQ.\text{getFirst}().value\} \). The time complexity in this case is linear \( O(N) \) in an amortized sense.

4. Scheduling the Largest Number of Activities

We consider \( N \) activities. Each activity \( i \) \( (1 \leq i \leq N) \) has a fixed duration \( l(i) \) and must be scheduled during \( l(i) \) consecutive time moments. Moreover, each activity has a special time moment \( p(i) \) which must be included within its scheduled time interval. The activities must be scheduled during non-overlapping time intervals; however, the intervals may “touch” at their endpoints, but must not intersect otherwise. Because of the constraints, it may not be possible to schedule all the activities. Thus, we want to maximize the number of scheduled activities.

We will use a greedy algorithm. First, we sort the activities in increasing order of their special time moments. Thus, we will consider that \( p(1) \leq p(2) \leq ... \leq p(N) \). We will traverse the activities in this order, maintaining a stack \( S \) of the activities which have been scheduled so far (the activities scheduled more recently are closer to the top of the stack). Initially, we schedule the first activity, during the interval \([p(1)-l(1), p(1)]\) (and push the activity together with its interval on the stack). When we reach the activity \( p(i) \geq 2 \), we have \( \{x,y\} \), the interval of the activity at the top of the stack. If \( p(i) \geq y \), then we schedule activity \( i \) in the interval \([u = \max\{p(i)-l(i), y\}, v = u+l(i)]\). Then, we push the activity \( i \) on the stack, together with the interval \([u,v]\) during which it was scheduled. If, instead, we have \( p(i) < y \) and \( l(i) < x \), then we do not schedule the activity \( i \). The time complexity of this algorithm is \( O(N \log(N)) \) for sorting the activities and \( O(N) \) for traversing the activities in the sorted order and scheduling them.

5. Lexicographically Optimal Activity Scheduling

We consider a sequence of \( N \) time moments. For each time moment \( t \) \( (1 \leq N) \), a value \( u(t) \) is known, representing the utility function if no activity is scheduled during a time interval containing \( t \). We have a set of \( K \) activities, each of which consists of \( x \) consecutive time moments. We want to schedule the \( K \) activities during non-overlapping time intervals (i.e. containing disjoint time moments), such that the chronological sequence of utilities of the time moments during which no activity is scheduled is lexicographically minimum. To be more precise, if \( t(1), ..., t(N-K+1) \) are the moments when no activity is scheduled (and \( t(i) < t(i+1) \) for \( 1 \leq i \leq N-K+1 \)), then the sequence \( u(t(1)), ..., u(t(N-K+1)) \) is lexicographically minimum.

A simple solution is the following. We will maintain a counter \( CK \) with the number of already scheduled activities and a counter \( CC \) with the number of saved time moments (initially, \( CK = CC = 0 \)). We will also maintain a counter \( pos \), meaning that all the time moments on the positions \( 1, ..., pos-1 \) have already been considered (they are either part of a scheduled activity or are saved); initially, \( pos = 1 \). While \( CK < K \) and \( CC < N-K \), we will execute the following actions. We will select the next time moment to be saved. This is one of the moments \( pos \),
that we selected the time moment for which u(t) is minimum and, in case of ties, we will choose the smallest such moment t. Let’s assume that we selected the time moment pos+j·x. We will increment CK by j (as j more activities are scheduled in the intervals [pos, pos+x-1], ..., [pos+(j-1)x, pos+j·x-1]), we will increment CC by 1 and we will set pos= pos+j·x. This algorithm can be easily implemented in a time complexity of O(N·K). However, when K is too large, this complexity is not satisfactory. We will reduce the time complexity down to O(N), as follows. We will maintain a double-ended queue (deque) DQ(r) for each value r=0,1,...,x-1. We will gradually introduce in DQ(r) (0≤r≤x-1) the utilities of the time moments t (together with their associated time moments), with t mod x=r. Initially, every deque is empty. Each deque will store utility>u(pos'), for which 1≤r<; 0≤q<; and compute the maximum value TMAX(i,j,p)=max[Amax(i,s,q)+Amax(s+1,j,p-q)] over all the pairs of tuples. TMAX(i,j,p) is a candidate for Amax(i,j,p). The third possibility consists of having both vertices i and j as two vertices of the pth part. The pth part is allowed to have at most emax=B-I edges which are not also edges of the polygon; when (i,j) is an edge of the polygon (j=i+n-1), the pth part may have up to emax=B edges which are not also edges of the polygon. We will consider every value e (1≤e≤emax) and, for each e, we consider every set of e pairs (a1,b1), (a2,b2), ..., (an,bn) with the following properties: ia1≤; ib1≤; ia2≤ib2≤...≤; and compute the maximum value for each value of e and set of e pairs (a1,b1) (1≤e≤emax). We need to consider every set of e numbers q1, q2,..., qe, with the following properties: qe≥0 (1≤e≤e); q1+q2+...+qe=p-1. Then, the value AP(e,(a1,b1), ..., (an,bn)+Amax(a1+1,b1-1, q1)+...+Amax(a1+1,b1-1,qe)) is a candidate value for Amax(i,j,p). We will set Amax(i,j,p) to the maximum of all the candidate values (or -∞ if no candidate value exists). The optimal value of the total area of the K vertex-disjoint parts is max[Amax(i,j,K)] and the time complexity of this approach is O(nm+n=2·B+2·K·mmax+B+2·2).
complexity of the presented algorithm. There are found so far; initially, $\text{MaxAgg}$

$$\text{MaxAgg}=(LJ-LS+1)(CD-CS+1)$$

will add the element $A(LJ, c)$ to the list $\text{List}(c)$. For every element added to a list $\text{List}(c)$, we will also maintain a counter with the number of occurrences of this element in $\text{List}(c)$ (e.g. by using a hash table $\text{HT}(c)$ associated to each column, where the keys are the elements’ values and the values are the number of occurrences of the corresponding key). If, when adding a new element to $\text{List}(c)$, this element has never occurred before in $\text{List}(c)$, then its counter will be set to 1; otherwise, its counter will be incremented by 1. Thus, $\text{List}(c)$ will contain all the distinct elements on the column $c$, between the rows $LS$ and $LJ$. If $|\text{List}(c)|<K$ ($|\text{List}(c)|$ denotes the number of elements in $\text{List}(c)$), we will add no more element to $\text{List}(c)$. Thus, the maximum number of elements in a list $\text{List}(c)$ is bounded by $\min[M, K+1]$. We will now traverse the columns from left to right, maintaining two pointers, $CS$ and $CD$. We initialize $CS=1$ and $CD=0$. We will also maintain a list $L$ with the distinct elements (and a hash table $H$ with their numbers of occurrences) between the rows $LS$ and $LJ$ and the columns $CS$ and $CD$. Initially, $L$ (and $H$) will be empty. At every step $i$ ($i=1, \ldots, N$) we increment $CD$ by 1 and add the elements in $\text{List}(CD)$ to the list $L$; if an element $x$ in $\text{List}(CD)$ was not part of $L$, then we add it to $L$ and set its number of occurrences (in $H$) to 1; otherwise, we increment the number of occurrences of the element $x$ (in $H$). Then, while $|L|>K$, we will perform the following steps: (1) we delete from $L$ the elements $x$ in $\text{List}(CS)$; if the number of occurrences of $x$ (in $H$) is greater than 1, we decrement this number by 1; otherwise, we remove $x$ from $L$ (and from $H$); (2) $CS=CS+1$. At the end of each step, if $CS\leq CD$, then we have a submatrix $B$ with at most $K$ distinct elements, with the upper row $LS$, lower row $LJ$, left column $CS$ and right column $CD$. We will compute the aggregate $\text{Bagg}$ of the utilities of the submatrix $B$ in $O(1)$ time. For the sum aggregate function, we can use 4 prefix sum queries (see [5]) and for the max aggregate function, we can use multidimensional RMQ [5]. If all the utility values are 1, then $\text{Bagg}$ is the area of the submatrix: $\text{Bagg}=(LJ-LS+1)(CD-CS+1)$. We will set $\text{MaxAgg}=$max($\text{MaxAgg}$, $\text{Bagg}$) (where $\text{MaxAgg}$=the maximum aggregate value found so far; initially, $\text{MaxAgg}=0$). Let’s analyze the time complexity of the presented algorithm. There are $O(M^2N)$ insertion operations into the lists $\text{List}(c)$. An insertion can be performed in $O(1)$ time (if we use a normal linked list and a hash table for the number of occurrences and for maintaining pointers to the location of each element $x$ in the list), or in $O(\log(min[M,K]))$ time if we use a balanced tree (both for the list and the number of occurrences). Then, we have $O(M^2N\min[M,K])$ addition and/or removal operations to/from the list $L$. Again, if we use a standard linked list (for $L$) together with a hash table ($H$) for the number of occurrences and for maintaining pointers to the locations of the elements $x$ in $L$, the time complexity per operation is $O(1)$. If we implement $L$ as a balanced tree (which we use both as a “list” and for maintaining the number of occurrences), the time complexity is $O(\log(K))$ (because the list $L$ never contains more than $2K+1$ distinct elements). Thus, the best time complexity that we can achieve with the presented algorithm is $O(M^2N\min[M,K])$.

8. Related Work Activity scheduling problems have been considered in many papers. Problems regarding personnel activity scheduling in multiple domains were considered in [1, 3, 4, 6, 7, 10]. Pedestrian-route and activity scheduling theory and models were presented in [2]. Several greedy and dynamic programming algorithms for data transfer scheduling were presented in [8] and some efficient data structures were developed in [5, 9].

9. Conclusions and Future Work

In this paper we considered several constrained activity scheduling problems in the time and space domains. For each problem we presented novel, efficient algorithmic solutions which compute optimal schedules. As future work, we intend to consider activity scheduling problems with more complex constraints, which will have more direct applications in practical settings.

10. References

[1] Beddoe, G. R., Petrovic, S., “Selecting and Weighting Features Using a Genetic Algorithm in a Case-Based Reasoning Approach to Personnel Rostering”, European Journal of Operational Research, vol. 175, p. 649, 2006.
[2] Hoogendoorn, S. P., Bovy, P. H. L., “Pedestrian route-choice and activity scheduling theory and models”, Transportation Research Part B: Methodological, vol. 38 (2), p. 169, 2004.
[3] Burke, E. K., De Causmaecker, P., Vandenberghe, G., “A Hybrid Tabu Search Algorithm for the Nurse Rostering Problem”, Simulated Evolution and Learning, Springer Lecture Notes in Artificial Intelligence, vol. 1585, p. 187, 1999.
[4] Burke, E. K., Curtois, T., Post, G., Qu, R., Veltman, B., “A Hybrid Heuristic Ordering and Variable Neighbourhood Search for the Nurse Rostering Problem”, Technical Report NOTTCS-TR-2005-3, University of Nottingham, 2005.
[5] Andreica, M. E., Andreica, M. I., Catanciu, N., “Multidimensional Data Structures and Techniques for Efficient Decision Making”, Proc. of the WSEAS Intl. Conf. on Math. and Comp. in Business and Econ., 2009.
[6] Alfieri, A., Kroon, L., van de Veldse, S., “Personnel scheduling in a complex logistic system: a railway application case”, Journal of Intelligent Manufacturing, vol. 18, p. 223, 2007.
[7] Millar, H. H., Kiragu, M., “Cyclic and non-cyclic scheduling of 12 h shift nurses by network programming”, European Journal of Operational Research, vol. 104, p. 582, 1998.
[8] Andreica, M. I., Tapus, N., “High Multiplicity Scheduling of File Transfers with Divisible Sizes on Multiple Classes of Paths”, Proc. of the IEEE Intl. Symp. on Consumer Electronics, p. 516, 2008.
[9] Andreica, M. I., Tapus, N., “Efficient Data Structures for Online QoS-Constrained Data Transfer Scheduling”, Proc. of the IEEE Intl. Symp. on Parallel and Distributed Computing (ISPD), p. 285, 2008.
[10] Andreica, M. I., Andreica, R., Andreica, A., “Minimum Dissatisfaction Personnel Scheduling”, Proc. of the 32nd ARA Congress, p. 459, 2008.