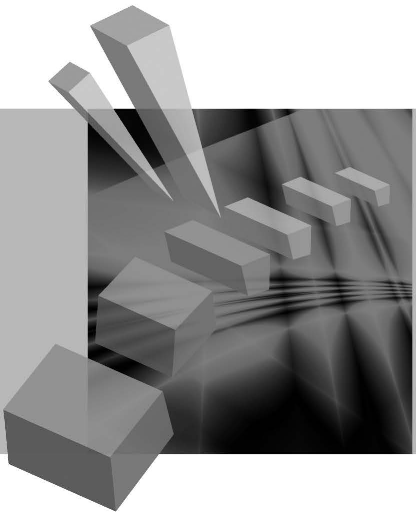
PRACE NAUKOWE
Uniwersytetu Ekonomicznego we Wrocławiu
RESEARCH PAPERS
of Wrocław University of Economics

238

Zastosowania badań operacyjnych Zarządzanie projektami, decyzje finansowe, logistyka



Redaktor naukowy
Ewa Konarzewska-Gubała

Wydawnictwo Uniwersytetu Ekonomicznego we Wrocławiu
Wrocław 2011

Recenzenci: Stefan Grzesiak, Donata Kopańska-Bródka, Wojciech Sikora,
Józef Stawicki, Tomasz Szapiro, Tadeusz Trzaskalik

Redaktor Wydawnictwa: Elżbieta Kożuchowska

Redaktor techniczny: Barbara Łopusiewicz

Korektor: Barbara Cibis

Łamanie: Małgorzata Czupryńska

Projekt okładki: Beata Dębska

Publikacja jest dostępna w Internecie na stronach:
www.ibuk.pl, www.ebscohost.com,
The Central and Eastern European Online Library www.ceeol.com,
a także w adnotowanej bibliografii zagadnień ekonomicznych BazEkon
http://kangur.uek.krakow.pl/bazy_ae/bazekon/nowy/index.php

Informacje o naborze artykułów i zasadach recenzowania znajdują się
na stronie internetowej Wydawnictwa
www.wydawnictwo.ue.wroc.pl

Kopiowanie i powielanie w jakiejkolwiek formie
wymaga pisemnej zgody Wydawcy

© Copyright by Uniwersytet Ekonomiczny we Wrocławiu
Wrocław 2011

ISSN 1899-3192
ISBN 978-83-7695-195-9

Wersja pierwotna: publikacja drukowana

Druk: Drukarnia TOTEM

Spis treści

Summaries

PRACE NAUKOWE UNIWERSYTETU EKONOMICZNEGO WE WROCŁAWIU
RESEARCH PAPERS OF WROCŁAW UNIVERSITY OF ECONOMICS nr 238 • 2011

Zastosowanie badań operacyjnych
Zarządzanie projektami, decyzje finansowe, logistyka ISSN 1899-3192

Tymon Marchwicki, Dorota Kuchta
Technical University in Wrocław

A NEW METHOD OF PROJECT SCHEDULE LEVELLING

Summary: We consider the problem of project schedule levelling and propose a new algorithm, based on the pattern matching algorithms. The activity resource requirements and free resources are considered as two patterns which are to be matched. We explain the idea of the algorithm (the pattern matching algorithms have to be modified in order to be used to solve the problem in question) and prove its computational complexity, which turns out to be O(n^2).

Keywords: RCPSP, resource gaps, complexity, continuous time schedule, project schedule levelling.

1. Introduction

The method described in this paper concerns project schedule levelling. We may imagine a situation in which the whole schedule is known, activities already have their start times assigned, but there is a need to move backward or forward some of the activities, so that the resource constraints stay valid.

The problem can be defined as follows: Let A be the set of already scheduled activities (tasks): $A = \{a_1, a_2, \ldots, a_n\}$, where each activity is characterized as a triple $a_i = <s_i, d_i, q_i>$, $i = 1 \ldots n$, where s_i is the activity start time, d_i is the activity duration, q_i is the amount of resources used by this activity. What is more, we assume that each activity a_i has its list of successors $S(a_i)$ and predecessors $P(a_i)$. These lists, in particular, may be empty. Additionally we assume the resource constraints for the project: $R=\{r_1, r_2, \ldots, r_M\}$. Each r_k has the form of the condition that specifies the amount of resources of the k-th type for the whole project. It is also important that we assume the continuous time: each activity can start at any arbitrary moment of time. We then take one activity a_i from the set A for which there is not enough resources in the period it is planned for (together with all its successors and predecessors), move this activity forward/backward to some arbitrary place t_0 and we ask what the new schedule is, so as a_i and all its successors/predecessors are rescheduled to such time moments that their start times are greater/less than t_0 (or equal to) and the resource constraints are fulfilled. The problem is illustrated in Figure 1.

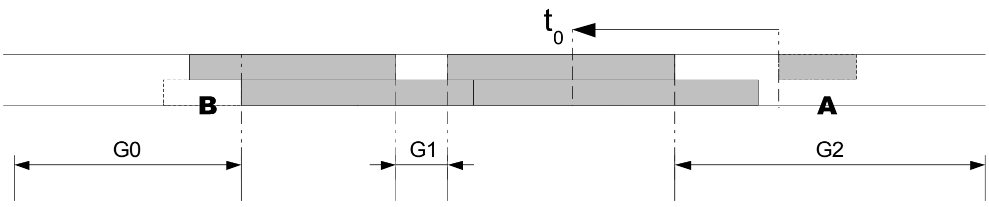


Figure 1. Finding resource gaps.

Source: own study.

Figure 1 shows a situation in which each activity (grey rectangle) is already scheduled and has its fixed position in the timeline and new activity (dashed rectangle) is being inserted. Activities that overlap (i.e. are executed paralelly at the same time as other activities) are drawn in this diagram in different lines. Resource constraint is that no more than two activities can be executed at the same time (let us say that each activity uses 1 unit of resources and we only have 2 units of resources available at each moment of time during the whole project execution). In this representation gaps are empty spaces so as at least one diagram line is empty. The idea of the algorithm showed in Figure 1 can be described in three steps: 1) We have an activity at position "A". 2) We move this activity backward to position t_0 (the movement is symbolised by an arrow). 3) Because we moved activity backward to the place where there is no resource gap, we need to move it to the first place before t_0 that has available resources. Gap 1 (marked as "G1") is too short for the activity, so finally our activity goes to position "B" which belongs to gap 0 ("G0").

In the literature there are various algorithms for determining a feasible schedule in case the resources are limited [e.g. Deckro et al. 1989; Brucker et al. 1999; Gemmil et al. 1999; Chatourou , Haouari 2008). Most of them are heuristics (in the sense that they do not necessarily minimise the project completion time), as the problem in question (with the project completion time as the objective function) is NP-hard. The problem stated in this article is a special case of resource-constrained scheduling problem where the schedule is considered to change during time and all schedule modifications need to be done in an on-line manner (i.e. rescheduling problem). Rescheduling is needed due to different environment disturbances: machine breakdown, processing time variation, job(s) cancellation (as listed in [Cheng et al. 2005]). Rescheduling techniques have been studied in several articles and there are different approaches. An overview of rescheduling techniques can be found in [Vieira et al. 2003] and [Liu, Shih 2009]. As in case of scheduling, many rescheduling methods also incorporate heuristics. The examples can be found in [Aufenanger et al. 2009] or [Cheng et al. 2005]. It is common knowledge that heuristics behave in various ways for various project instances. There are approaches that take advantage of local techniques which reschedule only the subset of tasks that might be affected [Kuster et al. 2007]. There are also techniques that try to build a robust schedule, which means that the schedule quality does not change significantly when a disruption occurs (as defined in [Pfeiffer et al. 2007]). Sometimes the simplest solution such as the allocation of additional resources, switches from one process variant to another or task shift is only needed. The example of such a technique is presented in [Artigues et al. 2003]. Here we would like to present another heuristic with a fairly good computational complexity, based on the pattern matching algorithms. To our best knowledge, no such algorithms have been known so far. In section 2 we present the idea of the method and its basic procedures and in section 3 the proof concerning its computational complexity in the worst case.

2. The algorithm idea

The method consists of two procedures. The first procedure builds a resource profile for a given schedule, the second procedure is a matching procedure that iterates through elements of the resource profile and tries to match the activity to the nearest available gap. We first present this very simple and intuitive procedure for building resource profile for the schedule with continuous time and activities that can start/end at any arbitrary moment of time. We then present a matching procedure.

Procedure used for building the resource profile is based on the “change points” in resource availability. Because different activities have different resource requirements, activities start times and end times (together with project resource constraints change points) determine the points where available resources can possibly change (but they do not have to). We first iterate through all activities and add their start times and end times to the list of “change points”. Other points that should be added here are the points where project resource constraints change. Change points are finally sorted (Figure 2a). The main procedure is based on these change points: each pair of subsequent points is taken one after another to create subsequent time intervals (Figure 2c). To calculate the resource that is available for each time interval we need another helper procedure (see Figure 2b) that iterates through all tasks and checks which of the tasks are executed within this interval. (It checks if the given interval is the subinterval of the activity time range.) Before sorting all the change points (Figure 2a) and creating intervals, the two “special” change points need to be added to this list. These are “boundary change points” that are the artificial change points in minus infinity and plus infinity time moments. They are needed to have the full list of all time intervals where the activity can be put into. Different intervals created in this way represent different time periods where available resources are fixed. New interval means that available resources are (possibly) different from the previous one.

```
Procedure GetChangePoints(activities)
        Let changePoints be the empty list
For each $a_i$ in ativities
        If $a_i$ startTime is not already in changePoints list
                Add $a_i$ startTime to the list
        End if
        If ($a_i$ startTime + $a_i$ duration) is not already in changePoints list
                Add ($a_i$ startTime + $a_i$ duration) to the list
        End if
End for
Add boundary change points to the list (-infinity and +infinity)
Sort changePoints list
Return changePoints
End procedure
```

Figure 2a. Calculating all change points, where resource availability can possibly change

Source: own study.

```
Procedure GetResourceUsage(intervalStart, intervalEnd, activities)
        resourceUsed = 0
For each $a_i$ in activities
                If middle of interval is within $a_i$ time range
                resourceUsed = resourceUsed + $a_i$ resource usage
                End if
        End for
        Return resourceUsed
End procedure
```

Figure 2b. Calculating resource usage of the time interval

Source: own study.

```
Procedure GetResourceProfile(activities)
Let resourceProfile be the empty list of intervals
changePoints = getChangePoints(activities) → figure 2a
For each $c_j$ in changePoints
        intervalStart = $c_j$
        intervalEnd = $c_{j+1}$
resourcesUsed = getResourceUsage(intervalStart, intervalEnd, activities) → figure 2b
Add new interval with intervalStart, intervalEnd & resourcesUsed parameters to the resourceProfile
End for
Return resourceProfile
End procedure
```

Figure 2c. Building resource profile according to the change points determined (Figure 2a) and using resource calculation procedure (Figure 2b)

Source: own study.

Figure 3a presents the result of *getChangePoints* procedure for the schedule from Figure 1. Change points are drawn as dashed lines. Change points are all activities start times and end times extended with +inf and –inf time points and then sorted. We assume that the resource constraint (maximum amount of resources available) is fixed for the whole project duration. Figure 3b presents the resource profile with available resource values calculated for this schedule.

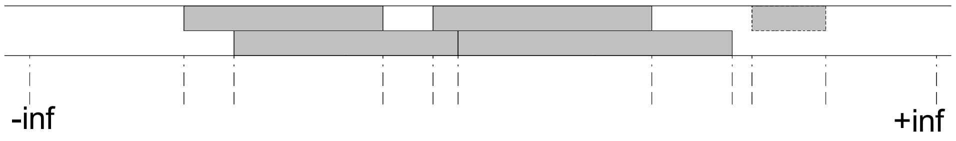


Figure 3a. Change points for the schedule from Figure 1

Source: own study.

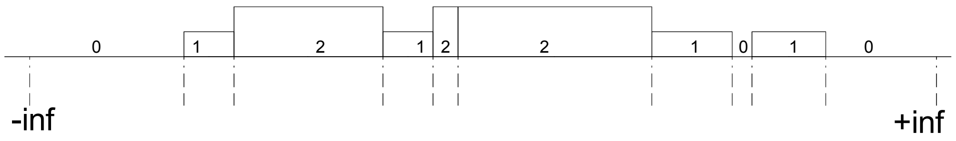


Figure 3b. Resource profile corresponding to the schedule from Figure 1 and change points from Figure 3a

Source: own study.

The first interval (formed from the first boundary change point and the first "ordinary" change point) is always the gap, where all project resources are available. No resources are used here, because there are no activities. The last interval (formed from the last "ordinary" change point and the last boundary change point) is similar, it contains all available resources. The intervals that are in between the first and the last interval may vary in amount of resources used (which implies the amount of free resources, if resource constraint is constant for the whole project), depending on number and type of activities that are executed within this interval (see Figure 3b).

After the resource profile is known, we may apply the matching procedure to match activity that is being moved to the proper resource gap.

We can identify finding resource gap in the resource profile as a problem of pattern matching. There are several such algorithms that has been widely studied in the literature – they concern text pattern matching. These are: *KMP* algorithm (Knuth-Morris-Pratt), Boyer-Moore algorithm, Rabin-Karp algorithm and the algorithm based on constructing finite automats [Cormen et al. 2004]. The best of them (*KMP* and finite automata) algorithms have *O(n)* time complexity where *n* is the text length. We can reduce the problem described in this article to a problem of pattern matching with two symbols only: positive and negative symbols. Positive symbols are those resource profile intervals that have available resources for the activity, negative are those that do not have enough amount of resources. The pattern is the activity being moved and we can imagine that it consists of positive symbols only and has variable length. The length is variable, because the length of the pattern is fixed, but resource profile elements have variable length, so when we try to match our pattern to the profile, every time we have different number of elements that should match. This is presented in Figure 4. We now describe why none of the standard patterns matching algorithms can be applied to our problem

The basic problem here is that the length of the pattern is variable and (depending on the resource elements lengths) the pattern should be compared to different number of elements. That is why we are not able to construct a reasonable automat, because it would then have infinite number of states. We cannot omit this variable length problem not only in a case of automat algorithms, but also with all other standard text-match algorithms mentioned above. There exist methods that match variable length patterns [Rahmann et al. 2006], but they operate on patterns that vary in length counted in number of symbols, but all symbols have fixed "unit" length, which again cannot be applied to the problem considered here.

Actually our problem is much simpler than described methods, although symbol lengths may vary. The simplicity arises from the fact that we have only two symbols and our pattern consists of one ("positive") symbol only.

The matching procedure can be examined for two different cases. The first case is that we are moving our task forward, so the nearest gap is searched in the future. The second case concerns moving task backward and searching gap in the past. For simplicity we present matching procedure for only one case – while moving task forward. Analogue procedure can be applied for moving backward. The changes will be in *reference points* we consider as well as in *membership intervals* (we introduce these definitions shortly). The process goes as follows: 1) We moved activity forward to the new place, which is some point t_0 (which means that activity start time equals t_0). 2) This point does not have to be the right point to put our activity at, because resource constraints may not hold. 3) We start our matching procedure to find the nearest resource gap. Particularly we may find this gap at t_0. It happens when there are available resources in the place we moved our activity to. But in general we should find the gap in the future of t_0 time moment, which means somewhere on the right (compare this to Figure 1, where we have the opposite situation – gap is searched in the past).

To present our method the two basic definitions need to be introduced. We first define the *reference point*. The *reference point* is the point that identifies the activity position. For the case being described (searching gaps in the future) it is convenient to choose the activity start time as a *reference point*. We also need to define the *membership interval*. *Membership interval* is the interval connected with every resource element. When reference point is within this interval, we say that our activity starts at this resource element and from this element we will start our matching procedure. For our case, the *membership interval* is the interval [a; b], which is the range of the resource element (but considered without the right closure, so that subsequent resource elements do not intersect).

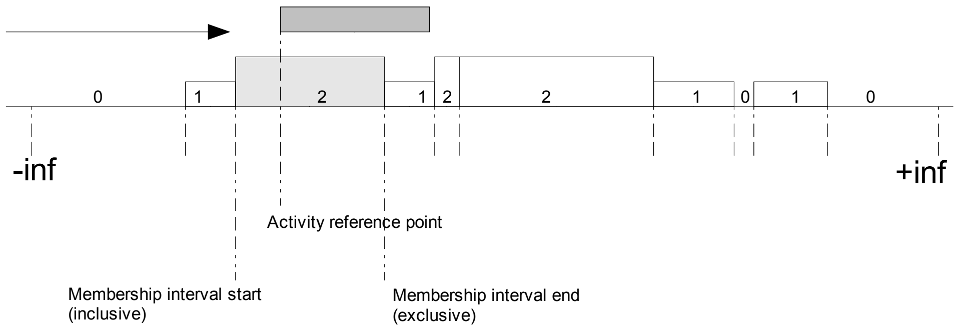

0
1
2
1
2
2
1
0
1
0
-inf
+inf
Activity reference point
Membership interval start (inclusive)
Membership interval end (exclusive)

Figure 4. Matching procedure starts from the modified binary search

Source: own study.

The matching procedure starts from searching the resource profile element we are at. When our activity is being moved, we do not know at which resource element we stopped, so we need to find it (Figure 4). In Figure 4 we are moving forward. Activity is coloured grey and the resource profile element that was found through binary search is coloured light grey. Every time we move activity to some arbitrary point t_0, we first need to search this initial resource element. Once we find it, we can execute the main matching procedure. The initial search procedure is realised as a binary search that searches the proper resource profile element we should begin matching from. Unlike standard binary search that uses single values and >, <, = operators to compare them, our binary search uses *membership intervals* and compares them with the *reference point*. Using operator > in standard binary search is replaced by a simple check if our *reference point* is on the right of the *membership interval*. Operator < is replaced by an opposite check if our *reference point* is on the left. Operator = is replaced with the check if reference point is within the *membership interval*.

After the search is complete, the main procedure can be executed. It starts matching from the resource profile element that was determined in binary search (light grey rectangle in Figure 4). The main procedure examines subsequent shifts to check if activity is matched (Figure 7b). The exception is the first check, when we do not shift activity, but leave it as it is – at the place it was put into – see Figure 5. In this picture we have a situation that activity was first moved somewhere over the resource element indexed with $i = 2$. We first try to match it without shifting/aligning it to the beginning of the resource element, but leave it as it is. It turns out that the first element does not match (symbol "X" over $i = 2$). We then shift activity to index 3 (which is the first element after the element that did not match) aligning it to the beginning of the resource profile element and we check if this position is matched. Although index 3 matches, the next index ($i = 4$) does not match (symbolised by letter "X" over $i = 4$). The main procedure uses helper procedure *IsResourceMatched* to check if current shift matches (Figure 7a). If current shift matches, we have found the right place. If current shift does not match, the index that first did not match is remembered and the main procedure jumps to the next index after this index. It is

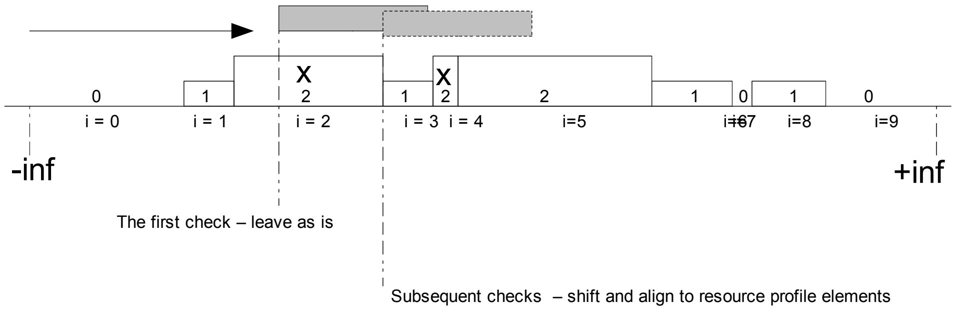

X
X
0
1
2
1
2
2
1
0
1
0
i = 0
i = 1
i = 2
i = 3
i = 4
i=5
i=6 7
i=8
i=9
-inf
+inf
The first check – leave as is
Subsequent checks – shift and align to resource profile elements

Figure 5. Shifts illustration used by *MatchActivity* procedure

Source: own study.

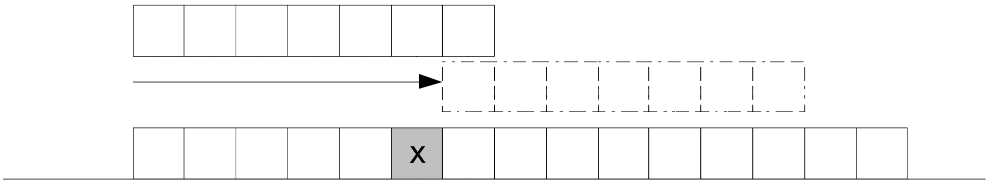


Figure 6. Correct shifts illustration

Source: own study.

```
Procedure IsResourceMatched(a_i, index)
       isMatched = true
       while isMatched & element intersects with activity
If there is not enough amount of resources for this activity in
this capacity element
              isMatched = false
              remember index that first does not match
End if
index = index + 1
       End while
End procedure
```

Figure 7a. Helper matching procedure that checks if a given shifted activity is matched to all the resource elements from the resource profile.

Source: own study.

very important to notice that if not the whole activity is matched and we know the index that first did not match, we can for sure move to the element that is situated just after the capacity element that first did not match, because for sure all shorter shifts will not be correct. Figure 6 depicts this observation. For simplicity and without loose of generality we present a case where all symbols (both from the pattern and symbols we compare our pattern to) have equal length. White rectangles represent *positive symbols* (i.e. that have free resources). Grey rectangle with letter "X" is a *negative symbol* without free resources.

```
Procedure MatchActivity($a_i$)
matched = false
       startIndex = BinarySearchElementIndex() → figure 4
       index = startIndex

       While matched = false
              If index <> startIndex
                     shiftActivity to resurce index
              End if

              matched = IsResourceMatched($a_i$, index) → figure 5

              If matched = false
                     index = index that first not matched + 1
              End if
       End while
       Return $a_i$
End procedure
```

Figure 7b. Matching procedure used for finding resource gaps

Source: own study.

In the next section we discuss the complexity of the algorithm.

3. Algorithm complexity

To calculate the algorithm upper bound execution time (*O-notation,* [Cormen et al. 2004]) as a function of number of activities (n), we analyse each line of the pseudocode presented in the article, associate fixed cost with each line and ask how many times the operation in the line was executed. We than simplify its expression by replacing sums of costs with one fixed cost and consider only the leading term of this expression [Cormen et al. 2004].

In Table 1 sums are indexed from 0 to $2n - 1$, because the list length grows (from 0 to $2n$) after each step and each time the whole list needs to be checked to determine if an element does not already exist in the list. What is more, we assume that the time complexity of the sorting algorithm that we use is logarithmic.

Table 1. Procedure *GetChangePoints*

Line of code	Cost	No. of times
`Let` *`changePoints`* `be the empty list`	c_1	1
`For each` *`aᵢ`* `in` *`ativities`*	c_2	N
`If` *`aᵢ startTime`* `is not already in` *`changePoints`* `list`	c_3	$\sum_{i=0}^{2n-1} i$
`Add` *`aᵢ`* `startTime to the list`	c_4	n
`If (`*`aᵢ startTime`* `+` *`aᵢ duration`*`) is not already in` *`changePoints`* `list`	c_5	$\sum_{i=0}^{2n-1} i$
`Add (`*`aᵢ startTime`* `+` *`aᵢ duration`*`) to the list`	c_6	n
`Add boundary change points to the list (-infinity and +infinity)`	c_7	1
`Sort` *`changePoints`* `list`	$c_8 \cdot n\log n$	1

Source: own study.

The time of the above procedure is then:

$$T_1(n) = c_1 + c_2 n + c_3 \frac{2n-1}{2} n + c_4 n + c_5 \frac{2n-1}{2} n + c_6 n + c_7 + c_8 n \log n$$

$$= (c_3 + c_5) n^2 + c_8 n \log n + (c_2 - \frac{c_3}{2} + c_4 - \frac{c_5}{2} + c_6) n + c_1 + c_7 =$$

$$= an^2 + b \cdot n \log n + cn + d = O(n^2).$$

We perform similar calculations for the second procedure (*GetResourceUsage*). Calculated complexity of this procedure is then $O(n)$.

Table 2. Procedure *GetResourceUsage*

Line of code	Cost	No. of times
`resourceUsed` `= 0`	c_1	1
`For each` *`aᵢ`* `in` *`ativities`*	c_2	n
`If middle of interval is within` *`aᵢ`* `time range`	c_3	n
`resourceUsed` `=` *`resourceUsed`* `+` *`aᵢ`*	c_4	n

Source: own study.

Thus we have $T_2(n) = O(n)$.

Third procedure (*GetResourceProfile*) uses the previous two procedures and we use results calculated above (lines associated with costs c_2 and c_6) to determine procedure's time complexity (Table 3).

Table 3. Procedure *GetResourceProfile*

Line of code	Cost	No. of times
`Let` *`resourceProfile`* `be the empty list of intervals`	c_1	1
`changePoints` `=` *`getChangePoints`*`(`*`activities`*`)`	$c_2 n^2$	1
`For each` c_j `in` *`changePoints`*	c_3	$2n$
`intervalStart` `=` c_j	c_4	$2n$
`intervalEnd` `=` c_{j+1}	c_5	$2n$
`resourcesUsed` `=` *`getResourceUsage`* `(`*`intervalStart`*`, `*`intervalEnd`*`, `*`activities`*`)`	$c_6 n$	$2n$
`Add new interval with` *`intervalStart`*`,` *`intervalEnd`* `&` *`resourcesUsed`* `parameters to the` *`resourceProfile`*	c_7	$2n$

Source: own study.

Thus we have $T_3(n) = O(n^2)$.

For the next procedure (*IsResourceMatched*) we can observe that its execution time (number of times the *while* loop is executed) is dependent on the task length. We denote task length as m (number of resource profile elements used) and we use this value also to determine the execution time of subsequent procedure (*MatchActivity*).

Table 4. Procedure *IsResourceMatched*

Line of code	Cost	No. of times
`isMatched` `= true`	c_1	1
`while` *`isMatched`* `& element intersects with activity`	c_2	m
`If there is not enough amount of resources for this activity in this capacity element`	c_3	m
`isMatched` `= false`	c_4	1
`index that first does not match`	c_5	1
`index` `=` *`index`* `+ 1`	c_6	m

Source: own study.

Thus we have $T_4(m) = O(m)$.

For the last procedure we use modified binary search algorithm, that has logarithmic complexity. This logarithmic time is included in our calculations (Table 5).

Table 5. Procedure *MatchActivity*

Line of code	Cost	No. of times
`matched = false`	c_1	1
`startIndex = BinarySearchElementIndex()`	$c_2 \log n$	1
`index = startIndex`	c_3	1
`While matched = false`	c_4	n/m
`If index <> startIndex`	c_5	n/m
`shiftActivity to resource index`	c_6	1
`matched = IsResourceMatched(`a_i`, index)`	$c_7 m$	n/m
`If matched = false`	c_8	n/m
`index = index that first did not match + 1`	c_9	n/m

Source: own study.

Thus we have $T_5(n) = O(n)$.

The last procedure (*MatchActivity*) is executed recursively for every successor. Maximum number of such successors is n, so the total time complexity for this procedure will be multiplied by n. We denote this time by $T_6(n)$.

Thus we have $T_6(n) = O(n^2)$.

The total execution time for the whole algorithm is the sum of the execution time of two main procedures: *GetResourceProfile* and *MatchActivity*. We calculate this total time as:

$$T(n) = T_3(n) + T_6(n) = O(n^2).$$

The above calculations show that our algorithm total time complexity is $O(n^2)$ in the worst case.

4. Conclusions

The method described here is an algorithm that finds resource gaps in continuous time projects schedule, determining a feasible project schedule. Its pessimistic time complexity is $O(n^2)$, where n is the number of tasks used in our schedule. We assumed one resource constraint for the whole project, but the algorithm could be easily extended to the case when there are various resource constraints in various periods. Further research is needed to find out for which project network types the algorithm proposed here behaves better than $O(n^2)$ and better than other project levelling heuristics known from the literature.

Literature

Artigues C., Michelon P., Reusser S., *Insertion techniques for static and dynamic resource--constrained project scheduling*, "European Journal of Operational Research" 2003, no. 149, p. 249-267.

Aufenanger M., Lipka N., Klopper B., Dangelmaier W., *A knowledge-based Giffer-Thompson heuristic for rescheduling job-shops*, IEEE Symposium on Computational Intelligence in Scheduling, Nashville, USA, 2009, p. 22-28.

Bandelloni M., Tucci M., Rinaldi R., *Optimal resource leveling using non-serial dynamic programming*, "European Journal of Operational Research" 1994, no. 78(2), p. 162-177.

Brucker P., Drexl A., Möhring R., Neumann K., Pesch E., *Resource-constrained project scheduling: Notation, classification, models, and methods*, "European Journal of Operational Research" 1999, no. 112, p. 3-41.

Chatourou H., Haouari M., *A two-stage-priority-rule-based algorithm for robust resource-constrained project scheduling*, "Computers & Industrial Engineering" 2008, no. 55(1), p. 183-194.

Cheng M., Sugi M., Ota J., Yamamoto M., Ito H., Inoue K., *Online job shop rescheduling with reaction-diffusion equation on a graph*, Intelligent Robots and Systems (IROS 2005), Edmonton, Canada, 2005, p. 3219-3224.

Cormen T., Leiserson C., Rivest R., Stein C., *Wprowadzenie do algorytmów*, Wydawnictwa Naukowo-Techniczne, Warszawa 2004.

Deckro R.F., Hebert J.E., *Resource Constrained Project Management*, "OMEGA International Journal of Management Science" 1989, no. 17(1), p. 69-79.

Gemmill D.D., Edwards M.L., *Improving resource-constrained project schedules with look-ahead techniques*, "Project Management Journal" 1999, no. 30(3), p. 44-55;

Kuster J., Jannach D., Friedrich G., *Local rescheduling – a novel approach for efficient response to schedule disruptions*, IEEE Symposium on computational intelligence in scheduling 2007, SCIS, p. 79-86.

Liu S., Shih K., *Construction rescheduling based on a manufacturing rescheduling framework*, "Automation in Construction" 2009, no. 18 (2009), p. 715-723.

Pfeiffer A., Kadar B., Monostori L., *Stability-oriented evaluation of rescheduling strategies by using simulation*, "Computers in Industry" 2007, no. 58, p. 630-643.

Rahman M., Iliopoulos C., Lee I., Mohamed M., Smyth W., *Finding Patterns with Variable Length Gaps or Don't Cares*, "Computing and Combinatorics, Lecture Notes in Computer Science" 2006, Volume 4112, p. 146-155.

Vieira G., Herrmann J., Lin E., *Rescheduling manufacturing systems: a framework of strategies, policies and methods*, "Journal of Scheduling" 2003, no. 6, p. 39-62.

NOWA METODA NIWELACJI HARMONOGRAMU PROJEKTU

Streszczenie: W pracy rozważamy problem bilansowania zasobów w harmonogramu projektu. Proponujemy nowy algorytm, oparty na dopasowywaniu wzorców. Wymagania dotyczące zasobów potrzebnych do wykonania zadania oraz wolne zasoby rozpatrujemy jako wzorce, które należy dopasować w sytuacji modyfikacji harmonogramu. Prezentowany materiał składa się z czterech części. W pierwszej części przedstawiamy matematyczne sformułowanie problemu oraz przegląd literaturowy rozważanych zagadnień. W drugiej części opisujemy ideę algorytmu (metody dopasowania wzorca zmodyfikowane tak, aby rozwiązać postawiony problem).

Opisane zostały dwie podprocedury składające się na algorytm. Pierwsza dotyczy budowy profilu zasobów, druga to właściwa procedura dopasowywania. Metoda znajduje zastosowanie dla harmonogramów z czasem reprezentowanym w sposób ciągły. W trzeciej części dowodzimy, że złożoność obliczeniowa metody wynosi $O(n^2)$ w pesymistycznym przypadku. W ostatniej części prezentujemy wnioski.

Słowa kluczowe: RCPSP, złożoność obliczeniowa, harmonogram z czasem ciągłym, bilansowanie zasobów, szeregowanie.