master's thesis

Algorithms for mixed-criticality scheduling with positive and negative time lags

Petr Cincibus

May 2015

Supervisor: Ing. Přemysl Šůcha, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Computer Graphics and Interaction

Acknowledgement

I would like to express my gratitude to my supervisor Ing. Přemysl Šůcha, Ph.D. for ideas which inspired me through the entire work, for willingness to discuss problems whenever I needed and for a friendly attitude.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

In Prague on May 11, 2015

Abstract

Mnoho komplikovaných průmyslových produktů v oblasti letectví, automobilového průmyslu, robotiky nebo zbrojního průmyslu se skládá z více podsystémů řídicích specifickou část systémových funkcí. Tyto podsystémy mohou být sloučeny v rámci jedné výpočetní platformy. V tomto případě se často stává, že některé funkce jsou důležitější než ostatní. V případě přetížení systému se prováděcí časy těchto funkcí prodlužují a je zapotřebí některé méně důležité systémové funkce vynechat. Řešení tohoto problému je plánování s různými stupni kritičnosti. Funkce systému jsou ohodnoceny kritičností podle důležitosti funkce. Plánování s různými stupni kritičnosti je relativně neprozkoumaný problém v oblasti offline plánování. Tato práce navrhuje algoritmy pro plánování s různými stupni kritičnosti s minimálními a maximálními časovými omezeními. Plánované úkoly jsou vykonávány bez přerušení na dedikovaných zdrojích s jednotkovou kapacitou. V této práci byl navržen a implementován jeden heuristický algoritmus a dva algoritmy pro hledání optimálního řešení. Efektivita algoritmů je porovnána se solverem celočíselného lineárního programování a s lazy clause generation SAT solverem.

Klíčová slova

Plánování; Algoritmy;

Abstract

Many complicated industry products in the area of aircraft production, automobile production, robotics or military industry consists of multiple subsystems controlling various parts of system functionalities. These subsystems can be merged into one computational platform. In this situation, it is quite frequent that some system functions are more important then the others. In the case of the system overload when execution times of functions start to prolongate there is a need to skip some unimportant functions. A solution to this problem is mixed-criticality scheduling where functions are evaluated with criticalities based on the function importance. Mixed-criticality scheduling is relatively unexplored problem on the field of offline scheduling. This work propose algorithms for the mixed-criticality scheduling problem with minimal and maximal temporal constraints. Tasks have to be scheduled without preemption on dedicated resources with unit capacity. One heuristic and two exact algorithms were proposed and implemented. Efficiency of the algorithms is compared with an integer linear programming solver and a SAT solver with lazy clause generation.

Keywords

Scheduling; Algorithms; mixed-criticality;

Contents

Abbreviations

RCPSP	Resource-Constrained Project Scheduling Problem
RCPSP/max	Resource-Constrained Project Scheduling Problem with minimal and maximal time-lags
SAT	Boolean Satisfiability Problem
FD	Finite Domain
CNF-SAT	Conjunctive Normal Form Boolean Satisfiability Problem
MDA	Minimal Delaying Alternative

1 Introduction

Scheduling is the process of assigning tasks to resources at particular times with respect to a given objective function. It is widely used in civil engineering, management, manufacturing, supply chain planning etc. A typical example of scheduling are factories where work is assigned to workers, machines or energy resources. In building construction, foundation of house, sanitation, building of walls, roof or wiring are tasks which have to be done and a concrete mixers, cranes, excavators or workers are resources available for tasks completion. In a computer operation system supporting multitasking or multithreading, tasks or threads are scheduled for execution on processor units by a scheduler.

Different objective functions are used in scheduling, for example project makespan, tardiness, weighted tardiness or total completion time. If a company optimizes a production process, then higher effectiveness of production and lower operational costs are criterions of scheduling. A little change in the objective function can cause enormous cash profits due to concurrent benefits.

Tasks are a subject of different constraints. Tasks can behave release dates or deadlines. In a reality a release time represents situations when a material is available from some point in time. A Deadline represents time when work must be done for example a result of a task is a product which must be available for customers or scheduler have to satisfy task deadlines in a real-time operation system. When task have to be processed after another task e.g. house walls have to be build after foundations of house are laid it can be expressed by precedence constraints. If there should be time distance between tasks e.g. walls must be build thirty days after concrete had been laid in houses foundations because of concrete hardening, then temporal constraints are considered.

Scheduling is a combinatorial optimization problem. There is a group of scheduling problems which can be solved by a polynomial algorithm but there are problems where a polynomial algorithm does not exist. These problems are solved by exact algorithms only for smaller input instances because computation may take an unreasonable amount of time. Heuristic algorithms are used for large scale instances. These types of algorithms are searching only in a small part of the search space and that is the reason why heuristics are a lot of faster then exact algorithms but the solution of a heuristic algorithm is not guaranteed to be an optimal solution. Heuristics are designed for specific problem and take full advantage of problem properties.

Meta-heuristics are general algorithms which are not problem-dependent and can be applied on various problems. Meta-heuristics can be seen as frameworks for creating algorithms for a specific problem. Example of meta-heuristics are local search, tabu search, simulated annealing, genetic algorithms, evolutionary algorithms, participle swarm optimization and etc. They can be divided by a number of solutions which are utilized on single solution meta-heuristics and population-based meta-heuristics. Local search is a typical example of a single solution meta-heuristic. On the other hand evolutionary algorithms are typical example of population heuristic utilizing multiple solutions which are all improved using properties of other solutions within population.

If we are interested in finding an optimal solution, then we have to use exact algorithms. Branch and bound is dominant technique used in scheduling for exact algorithms. The algorithm is searching in whole solution space by construction of the search tree. Search tree branching divides search space into disjoint sets. It is called branching. A lower bound can be computed for each search tree branch. If a lower bound has a higher value then the best solution found so far, then branch is discarded from searching. It is called bounding.

Scheduling algorithms can also be divided according to input. If an input is completely defined before execution of the algorithm, then it is called the offline scheduling algorithm. On the other hand an online scheduling algorithm is running continuously and an input is expanding during time. An online scheduling algorithm is used for example in operating systems to schedule tasks for execution in central processing units.

Embedded systems are used in cars, planes, weapons and etc. These devices are very complex consisting of many embedded systems controlling different functionalities. Each system contains a processor unit or a communication bus but these components are often underused. Nowadays, there is a trend to merge more different systems into one computation platform. It has a positive impact on energy consumption, weight, size and cost of a device. Because of that, systems with different importance are sharing computational resources. These systems are called mixed-criticality systems. Each task has assigned criticality. If a task with higher criticality is not completely processed in estimated time, it's processing time can be increased and consequent task with a lower criticality level can be omitted. Consequently, systems with high criticality can be mixed with lower ones e.g. flight-by-wire system of airplanes which is critical for flight share a processor or a bus with lower criticality weapon or communication systems which are critical for a mission but airplane is able to flight without it. In classical systems without mixed-criticalities task completion is assured using large pessimistic estimates of processing times. This approach leads to schedules with low utilization of resources in the run-time. In contrast, mixed criticality systems are using optimistic estimates of processing times for lower criticality levels and pessimistic estimates for higher criticality levels. This approach leads to better utilization of resources in the running time.

1.1 Motivation

Advantages of mixed-criticality scheduling can be shown on a simple example. There are three task which represents messages which will be scheduled for communication on bus in airplane. Transmission of messages is repeating in cycles. We are creating a schedule which will be repeated in each cycle. Task T_1 is fly-by-wire message, T_2 is weapon system message and T_3 is temperature sensor message. If the message transmission fails, message is send again depending on number of allowed repetition.

In classical scheduling without criticalities task processing times must be chosen pessimistically which will sufficiently assure message delivery. Transmission of one fly-by-wire message takes two time units. The message is very important therefore message can be send two times again in case of transmission failure. A weapon system message takes one time unit and can be resend one time. A temperature system message takes one time unit and when message transmission fails it is not send again because it is not so important.

- $p_1 = 6$
- $p_2 = 2$
- $p_3 = 1$

One of possible optimal schedules is illustrated in Figure 1 where schedule makespan is nine units. Schedule makespan can be improved if we use mixed-criticality scheduling

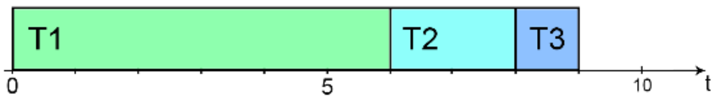


Figure 1 An example of a classical schedule

as we can se in Figure 2. Task is now specified by a pair (χ_i, P_i) where x_i is the task criticality and P_i is a vector of processing times.

- $T_1 = (3, (2, 4, 6))$
- $T_2 = (2, (1, 2))$
- $T_3 = (1, (1))$

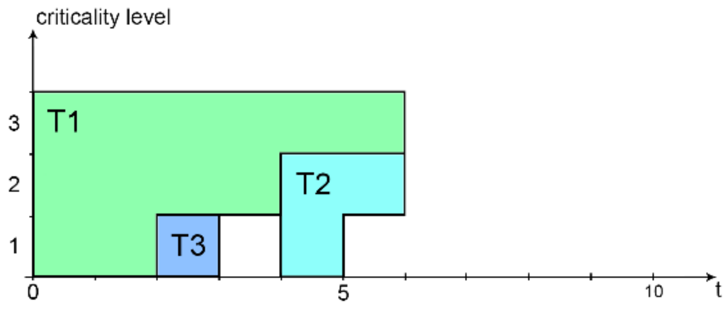


Figure 2 An example of a mixed-criticality schedule

Task T_1 is crucial for existence of an airplane and thus it gets the highest criticality. The Weapon system has lower criticality because it is mission critical but not important for flight and airplane existence except for a threat from another airplane. The temperature sensor message has the lowest criticality. Processing times in lower criticality levels can be chosen more optimistically. The makespan of mixed-criticality schedule is seven units long. It is four units shorter then schedule without criticalities.

The schedule can be executed in many different ways. Execution of the schedule starts with T_1 in criticality level 1. If T_1 is not completed at time 3 then the controlling system is switched into criticality level 2. If T_1 ends before time 4 then the system is switched back into criticality level 1 and execution of T_2 follows. If T_2 is not completed at time 5 then the system is switched into criticality level 2, task T_3 is skipped and T_2 is allowed to be processed until time 7. If T_1 is still running at time 5, then the system is switched into criticality level 3 and tasks T_2, T_3 are skipped. The simplest execution scenario is when every message is successfully sent on first attempt, then the system completely runs in criticality level 1.

One of possible execution examples is in Figure 3.

1.2 Related work

Most of works describing the mixed criticality task model belong to the domain of real-time scheduling. The first author who formalize the mixed-criticality scheduling model was Vestal [1]. Baruah et al. [2] showed that the preemptive mixed-criticality scheduling problem on a mono-processor is NP-hard and proposed the own criticality based priority algorithm for the problem. Kelly et al. [3] proposed an algorithm for

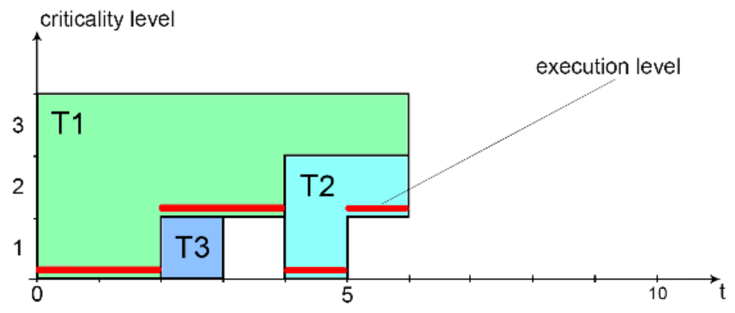


Figure 3 A schedule execution example

preemptive mixed-criticality scheduling on multiple processors. The problem is divided into two phases. Tasks are assigned to processors in the first phase and scheduled in the second phase. The second phase is realized with Audsley's algorithm or the rate monotonic algorithm. The zero slack rate monotonic scheduling was proposed by de Niz et al. [4]. These works are farther to our problem because we consider non-preemptive tasks and offline scheduling. More connected to our problem is work of Hanzálek et al. [5] considering non-preemive task model and offline scheduling. They proved that the non-preemptive time-constrained version of mixed-criticality scheduling on a mono processor is NP-hard in the strong sense by polynomial reduction from the 3-Partition problem. They also formulate an integer linear programming model based on precedence relations and tasks with release dates and deadlines.

Because there are no other works on the field of offline scheduling inspiration for our work comes from different problems. Well suited problem is The Resource-Constrained Project Scheduling Problem with Minimal and Maximal Time-Lags (RCPSP/max). It is frequently used in industry and that is the reason why is well studied in literature. One of our goals is to propose an exact algorithm. Blazewicz [6] proved that RCPSP is NP-hard problem in the strong sense and defined basic notation for scheduling problems. A Survey of works interested in exact algorithms is following. The first work which proposed an exact algorithm for the RCPSP/max and laid important theoretical foundations was published by Bartusch et al. [7]. There is a time oriented algorithm proposed by De Reyck and Herroelen [8] which resolves resource conflicts by adding precedence constraints between conflicting tasks. Andreas Fest et al. proposed a similar algorithm which resolves resource conflicts by dynamic release dates. Dorndorf et al. [9] proposed an algorithm on principle of constraint satisfaction where domains are sets of possible task start times. These domains are pruned by temporal and resource constraints. Another interesting approach in solving resource-constrained project scheduling problem is utilization of SAT solver. Ohrimeko [10] described method connecting SAT solving technique with finite domain solving technique. This was used by Horbach [11] who described reduction of feasibility RCPSP to SAT problem. The problem is then solved by SAT solver using separation of cover clauses which means that SAT solver starts with an incomplete set of constraints and additional constraints are generated during the run-time of the algorithm. He used Minisat solver [12] which is implementation of Davis–Putnam–Logemann–Loveland algorithm with conflict-driven clause learning [13]. Algorithm of Schutt et al. [14] is a combination of constraint satisfaction and sat solver with lazy clause generation where domains of possible start times are represented as boolean formulas. Lazy clause generation is some type of learning of the problem structure. Two possibilities of implementation of global cumulative constraint are discussed in Schutt et al. [15].

Another problems can be also considered for an exact algorithm inspiration . Based on a disjunctive graph model Carlier and Pinson [16] formulated an algorithm for the job shop scheduling problem. A similar idea was used by Brucker [17] in the algorithm for solving one machine scheduling problem with temporal constraints and he extends the algorithm with an advanced immediate selection procedure which fixes more disjunctions in each level of the branch and bound algorithm. A Heuristic algorithm for the RCPSP/max called iterative resource scheduling is proposed by Hanzálek et al. [18]. The Algorithm iteratively schedules tasks in free slots and unschedules the conflicting ones. Another interesting heuristic approach is application of squeaky wheel optimization with a bulldozing procedure proposed by Smith and Pyle [19]. Squeaky wheel optimization is a local search algorithm combined with a priority scheme. Foundations of Squeaky wheel optimization are described by Joslin and Clements [20]. Bulldozing is an effective technic rearranging unfeasible sequences of tasks.

1.3 Contribution and outline

The main contribution of this work are proposals and implementation of three algorithms for mixed-criticality scheduling problem. The first is a heuristic algorithm. It is based on iterative resource scheduling algorithm proposed by Hanzálek and Šůcha [18] combined with the bulldozing procedure [19]. Other two algorithms are exact algorithms finding an optimal solution. The first algorithm is based on fixing order of tasks combined with immediate selection procedure inspired by Brucker et al. [17] . The second one is working in time domain with conflict resolution by precedence constraints inspired by DeRyck et al. [8].

The work is organized as follows. Section 2 is the problem statement with the integer linear programming model. Section 3 is a proposal of the heuristic algorithm. In Section 4, there are presented two exact algorithms. Section 5 summaries computational results. Finally, Section 6 is conclusion of the work.

2 Problem statement

This section defines a basic notation used in this work. We deal with the non-preemptive mixed-criticality scheduling problem with dedicated resources of unit capacity and tasks constrained by maximal time-lags. The objective function is the project duration minimization. The standard notation [6] of the problem is $PSm, 1|temp, mc = \mathcal{L}|C_{max}$. A project is represented by a digraph G = (T,E) where $T = \{T_0, T_1, \ldots, T_{n+1}\}$ is a set of tasks and E is a set of edges representing temporal constraints. Let $R = \{1, 2, ..., m\}$ be a set of m resources. Resources has unit capacity therefore a resource could process only one task at a time. Let $\mathcal{L} \in \mathbb{N}$ be project criticality which is equal to the maximal criticality over all tasks. Task is defined by a 3-tuple of parameters (χ_i, P_i, a_i). Task criticality $\chi_i \in \{1, \ldots, \mathcal{L}\}$ is maximal criticality level of task i. Processing times $P_i \in \mathbb{N}_0^{\mathcal{L}}$ is a vector $(p_i^1, p_i^2, ..., p_i^{\chi_i})$, where c-th entry p_i^c is processing time on criticality level c. Task 0 and $n+1$ are dummy tasks $p_0^x = p_{n+1}^x = 0, \forall x \in \{1, ..., \mathcal{L}\}$. Processing time in certain criticality level is greater or equal to processing time in the previous level (1).

$$p_i^{c-1} \leq p_i^c, \qquad \forall i \in T, \forall c \in \{2, \ldots, \chi_i\} \tag{1}$$

Tasks in our problem has dedicated resources. It means that each task has predetermined one resource for processing. Task resource $a_i \in R$ is a resource where task i must be scheduled. Start time s_i is time when task i starts running on its resource. A schedule $S = (s_0, s_1, ..., s_{n+1})$ is a vector of start times and C_{max} is the schedule makespan.

Temporal constraints or generalized precedence constraints or time lags are constraints defining minimal distance between tasks start times given by inequality

$$s_i + l_{ij} \leq s_j, \qquad \forall (i,j) \in E \tag{2}$$

where l_{ij} is value of an edge $(i,j) \in E$ and defines the minimal distance between task start times. If $l_{ij} \geq 0$ then the time lag is called minimal. It defines the minimal distance between start times of tasks i and j assuming that task i precedes task j. If $l_{ij} < 0$ it is called maximal time lag defining maximal distance between start times of tasks j and i assuming that task j precedes task i. If task i have no temporal constraint pointing to task j it could be interpreted as temporal constraint with value $l_{ij} = -\infty$. Maximal time-lags make problem much more difficult.

Example of temporal constraints is illustrated in Figure 4. Task T_i defines a relative time window for task T_j in which it can be processed.

Task 0 defines the project start and have to be scheduled before all other tasks, i.e. $l_{0,i} = 0, \forall i \in T$. Task $n+1$ defines the project end and have to be scheduled after all other tasks, i.e. $l_{i,n+1} = p_i^{\chi_i}, \forall i \in T$.

Let $D = [d_{ij}]$ be a matrix of the longest paths computed from G, then d_{ij} is the minimal distance between tasks start times given by inequality

$$s_i + d_{ij} \leq s_j, \qquad \forall i, j \in V^2. \tag{3}$$

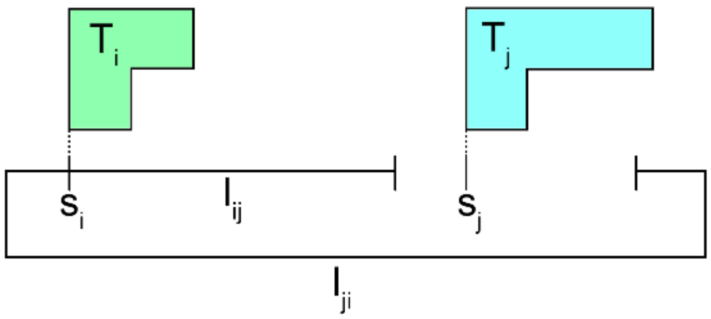


Figure 4 Maximal and minimal time-lag

Let UB be an upper bound of the project duration. If we assume that $s_0 = 0$ then $es_i = d_{0,i}$ is an the earliest start time of task i and $ls_i = UB - d_{i,n+1}$ is the latest start time of task i.

If task i precedes task j then the minimal distance between task start times is equal to processing time in maximal common criticality level of both tasks. Example of processing time in common criticality level is in Figure 5 and 6. The first example in Figure 5 illustrates a situation where task T_1 with higher criticality precedes task T_2 with lower criticality whereas the second example in Figure 6 illustrates the opposite situation.

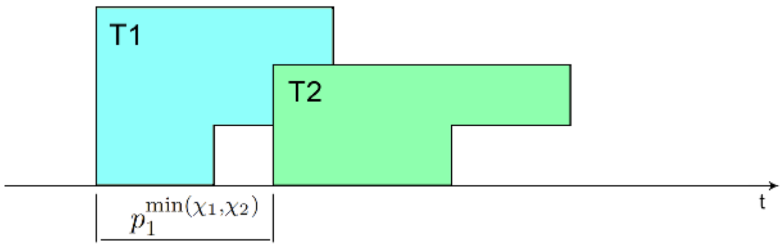


Figure 5 Processing time in common criticality level

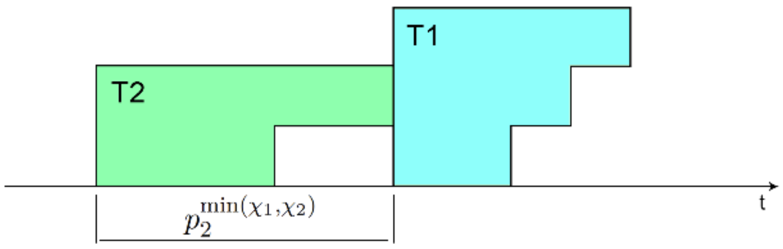


Figure 6 Processing time in common criticality level

Next, we formulate an integer linear programming model. The model is based on precedences of tasks. Objective function (4) is the project duration minimization. Inequality (5) is the temporal constraint. Precedence of tasks i and j is defined by a binary variable x_{ij}. Resource constraints are inequalities (6) and (7). The value of M

is some sufficiently large constant.

$$
\begin{aligned}
&\text{minimize} && C_{max} && && (4)\\
&\text{subject to} && s_i + l_{ij} < s_j, && \forall (i,j) \in E && (5)\\
& && s_j + p_j^{min(\chi_i,\chi_j)} \leq s_i + x_{ij}M, && \forall (i,j) \in T^2; i < j \wedge a_i = a_j && (6)\\
& && s_i + p_i^{min(\chi_i,\chi_j)} \leq s_j + (1 - x_{ij})M, && \forall (i,j) \in T^2; i < j \wedge a_i = a_j && (7)\\
& && s_i + p_i^{\chi_i} \leq C_{max}, && \forall i \in T && (8)\\
&\text{where} && s_i \in \langle 0, M \rangle && \forall i \in T &&\\
& && C_{max} \in \langle 0, M \rangle && &&\\
& && x_{ij} \in \{0,1\} && \forall (i,j) \in T^2; i < j \wedge a_i = a_j &&
\end{aligned}
$$

2.1 Scheduling Example

This section illustrates example of a scheduling instance. A graph of temporal constraints is illustrated in Figure 7. There are minimal and maximal temporal constraints. In Section 2 we defined that graph G contains a set of temporal constraints leading from start task to all other tasks and a set of temporal constraints leading from all tasks to end dummy task in order to guarantee that start task is scheduled before all other tasks and end task is scheduled after all other tasks. For the simplicity, we do not illustrate all of these constraints in figures because some of these constraints are not important and do not change a value of any longest path. The graph contains several negative temporal constraints.

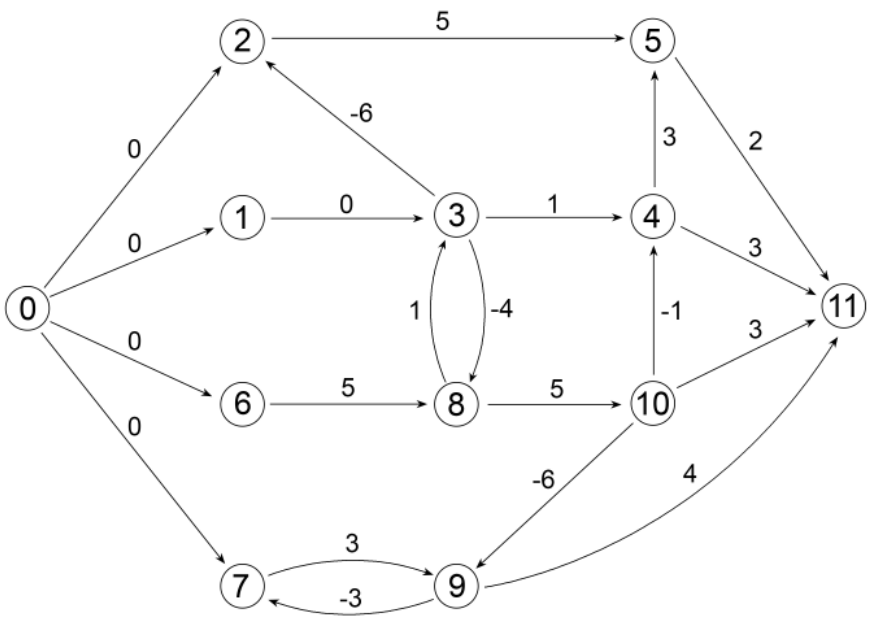


Figure 7 An input graph

There are two cycle structures. The first cycle is formed with tasks T_3 and T_8 and the second one is formed with tasks T_7 and T_9. Instances containing a cycle with positive length are infeasible (see Section 4.1.5). Both cycles in the example have negative or zero length therefore cycles do not force infeasibility. The example contains ten tasks which are assigned to two dedicated resources. Tasks are defined in the list below.

- $T_1 = (4, (1, 2, 3, 4), 1)$
- $T_2 = (2, (2, 5), 1)$
- $T_3 = (3, (1, 2, 3), 1)$
- $T_4 = (1, (3), 1)$
- $T_5 = (2, (1, 2), 1)$
- $T_6 = (2, (3, 5), 2)$
- $T_7 = (1, (2), 2)$
- $T_8 = (4, (1, 2, 4, 6), 2)$
- $T_9 = (1, (4), 2)$
- $T_{10} = (3, (1, 2, 3), 2)$

An optimal solution with makespan $C_{max} = 14$ is illustrated in Figure 8. A schedule for the first resource is in upper part of the figure and a schedule for the second one is in the bottom part of the figure.

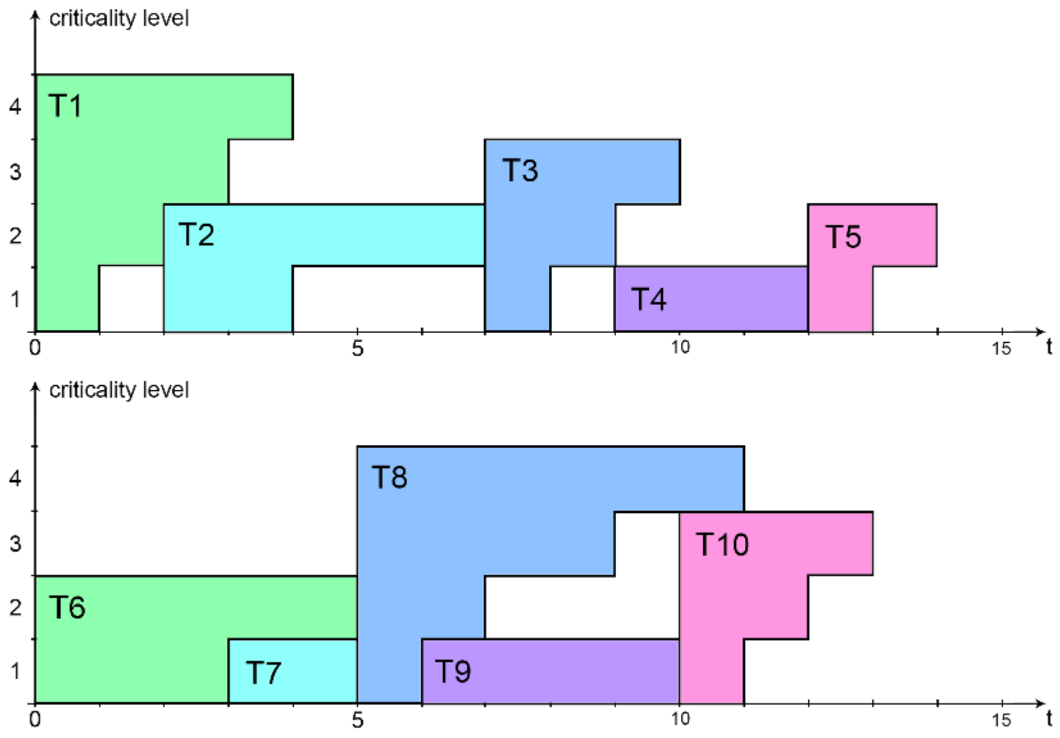


Figure 8 A solution schedule

3 Heuristic algorithm

This chapter propose a heuristic algorithm. The algorithm is a modification of the iterative resource scheduling algorithm proposed by Hanzálek and Šůcha [18] for RCPSP/max. It is a priority based algorithm with an unscheduling step. The first section is an algorithm description followed by an example of a problematic input instance with a solution in form of bulldozing procedure [19]. A proposal of an upper bound and a lower bound is in the end of the chapter.

3.1 Iterative resource scheduling algorithm

The main idea of the algorithm is to schedule tasks into free slots. If there are no free slots then a part of the schedule is canceled and the task is inserted into the new free space.

Algorithm 1 Iterative resource scheduling algorithm

```
1: procedure iterativeResourceScheduling( G, budgetRatio, n)
2:     $D \leftarrow \textsc{longestPaths}(G)$
3:     $LB \leftarrow \textsc{lowerBound}(G)$
4:     $UB \leftarrow \textsc{upperBound}(G)$
5:     $priority_i \leftarrow d_{i,n+1},\ \forall i \in T$
6:     $budget \leftarrow budgetRatio \cdot n$
7:     $C \leftarrow LB$
8:     while $LB \leq UB$ do
9:         $S \leftarrow \textsc{findSchedule}(C, priority, budget, D)$
10:        if $S$ is feasible then
11:            $S_{left} \leftarrow \textsc{ShiftLeft}(S)$
12:            $UB \leftarrow C_{max}(S_{left}) - 1$
13:            $S_{best} \leftarrow S_{left}$
14:        else
15:            $LB \leftarrow C + 1$
16:        end if
17:        $C = ceil((LB + UB)/2)$
18:    end while
19: end procedure
```

The main procedure is illustrated in Algorithm 1. Graph G is the problem instance. Parameter n is a number of tasks without two dummy tasks. First of all, the algorithm computes a matrix of the longest paths D from G by Floyd-Warshall algorithm. Edges of G are temporal constraints therefore the longest path d_{ij} from task i to task j is the minimal distance of start times in a feasible schedule enforced by temporal constraints. Project duration upper bound UB and project duration lower bound LB initialization is described in Section 3.5.

The algorithm is searching for a solution on a principle of an interval bisection. Function $findSchedule$ is finding a solution in a maximal number of steps denoted as *budget*. The number of steps *budget* is computed as $budgetRatio \cdot n$ where *budgetRatio* is an average number of attempts to schedule task. Variable C is an upper bound for $findSchedule$ function which have to find a solution with the makespan less or equal to C. The value of C is selected from interval $\langle LB, UB \rangle$. A task priority is defined as the maximal distance to task $n+1$. Consequently tasks with a low value of latest start time will have a higher priority for scheduling.

If a feasible solution is found, then the schedule is shifted left because the function *findSchedule* creates schedules containing unwanted gaps i.e. gaps which are not enforced by any temporal or resource constraint. Shifted schedule S_{left} is set as a new best schedule S_{best} and a value of an upper bound is updated $UB = C_{max}(S_{left}) - 1$. Minus one is used because an upper bound must be strict therefore the algorithm will not be interested in solutions with makespan value equal to $C_{max}(S_{best})$. If the solution of function $findSchedule$ is not feasible, then the lower bound is updated $LB = C + 1$.

The find schedule upper bound C is initialized with the value of the lower bound $C = LB$ for the first iteration of while loop, and as a result, the algorithm is trying to find the best possible solution in the first algorithm iteration. In next iterations, C is set to the mean value of the upper bound and lower bound. In the end of each algorithm iteration, the value of the lower bound is increased or the value of the upper bound is decreased. It assures algorithm termination because while loop terminates when $UB < LB$.

Function $shiftLeft$ computes schedule S_{left} from schedule S while preserving order of tasks. This could be done by Bellman-Ford algorithm computing the longest paths from task 0 to other tasks in graph G which is extended by precedence constraints representing the order of tasks in S. Then the longest path from task 0 to task i is equal to start time of task i in the new schedule S_{left}.

Algorithm 2 Iterative resource scheduling algorithm

```
procedure findSchedule(C, priority, budget, D)
    $s_i \leftarrow -\infty \; \forall i \in \{1, \dots, n\}$
    $scheduled \leftarrow \{\}$
    while $budget > 0 \wedge |scheduled| < n + 2$ do
        $i \leftarrow argmax_{\forall j \in T : j \notin scheduled}(priority_j)$
        $ES_i \leftarrow \max_{\forall j \in T : j \in scheduled}(s_j + d_{ji})$
        $LS_i \leftarrow \min_{\forall j \in T : j \in scheduled}(s_j - d_{ij})$
        $s_i \leftarrow$ findTimeSlot$(i, ES_i, LS_i)$
        if $s_i$ was found then
            scheduleActivity$(i, s_i, scheduled)$
        else
            scheduleActivityViolently$(i, s_i, scheduled)$
        end if
        $budget \leftarrow budget - 1$
    end while
end procedure
```

Schedule is constructed in function $findSchedule$. There is a priority queue of unscheduled tasks. In each iteration task i is removed from top of the queue. The earliest start time of task i is computed as maximum $s_j + d_{ji}$ for each task j from a set of scheduled tasks as a consequence of Equation (3). The latest start time of task i is bounded by an upper bound C i.e. $LS_i = C - d_{i,n+1}$. Function $findTimeSlot$ is searching for a resource time window for task i in interval $\langle ES_i, LS_i \rangle$. If a time window exists, then task i is scheduled by function *scheduleActivity*. If a time window is not found, then task i is inserted violently into the schedule on position $s_i = s_i^{prev} + 1$ where s_i^{prev} is the last start time of task i. If task i is scheduled for the first time, then it is scheduled to the earliest start time of the task i.e. $s_i = es_i = d_{0,i}$.

3.2 Problematic example

The algorithm described in the previous section is working on principle of task scheduling and unscheduling. There is a possibility that a group of tasks is periodically scheduled and unscheduled until the algorithm budget is exhausted. The algorithm is very efficient, however we found an input instance which was not solved successfully by the original algorithm.

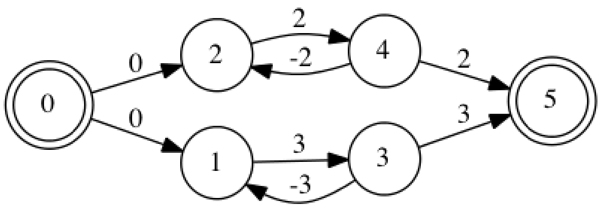
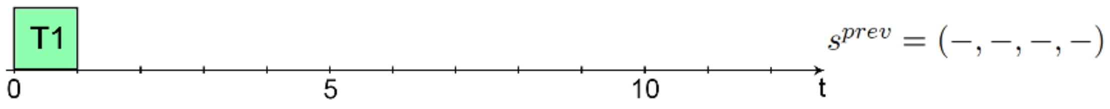


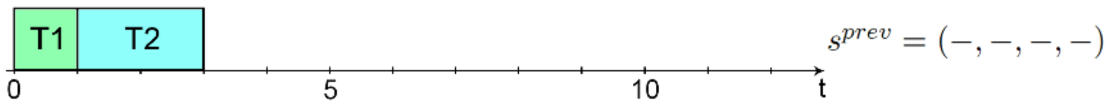
Figure 9 Graph of temporal constraints

For simplicity, example instance has $\mathcal{L} = 1$ and only one resource. The problematic instance is in Figure 9. There are four tasks with processing times $P_1 = (1), P_2 = (2), P_3 = (3), P_4 = (2)$ and two dummy tasks highlighted by double circle. The instance conains two cycle structures T_1, T_3 and T_2, T_4 which are important to make algorithm stucked. The longest path matrix D, computed from the input graph, is

$$D = \begin{pmatrix} 0 & 0 & 0 & 3 & 2 & 6 \\ -\infty & 0 & -\infty & 3 & -\infty & 6 \\ -\infty & -\infty & 0 & -\infty & 2 & 4 \\ -\infty & -3 & -\infty & 0 & -\infty & 3 \\ -\infty & -\infty & -2 & -\infty & 0 & 2 \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{pmatrix}.$$

Task priority is defined by the longest distance to end task therefore the priority queue is filled in this way $queue = (T_0, T_1, T_2, T_3, T_4, T_5)$ where the first element represents the top of the queue with the highest priority.





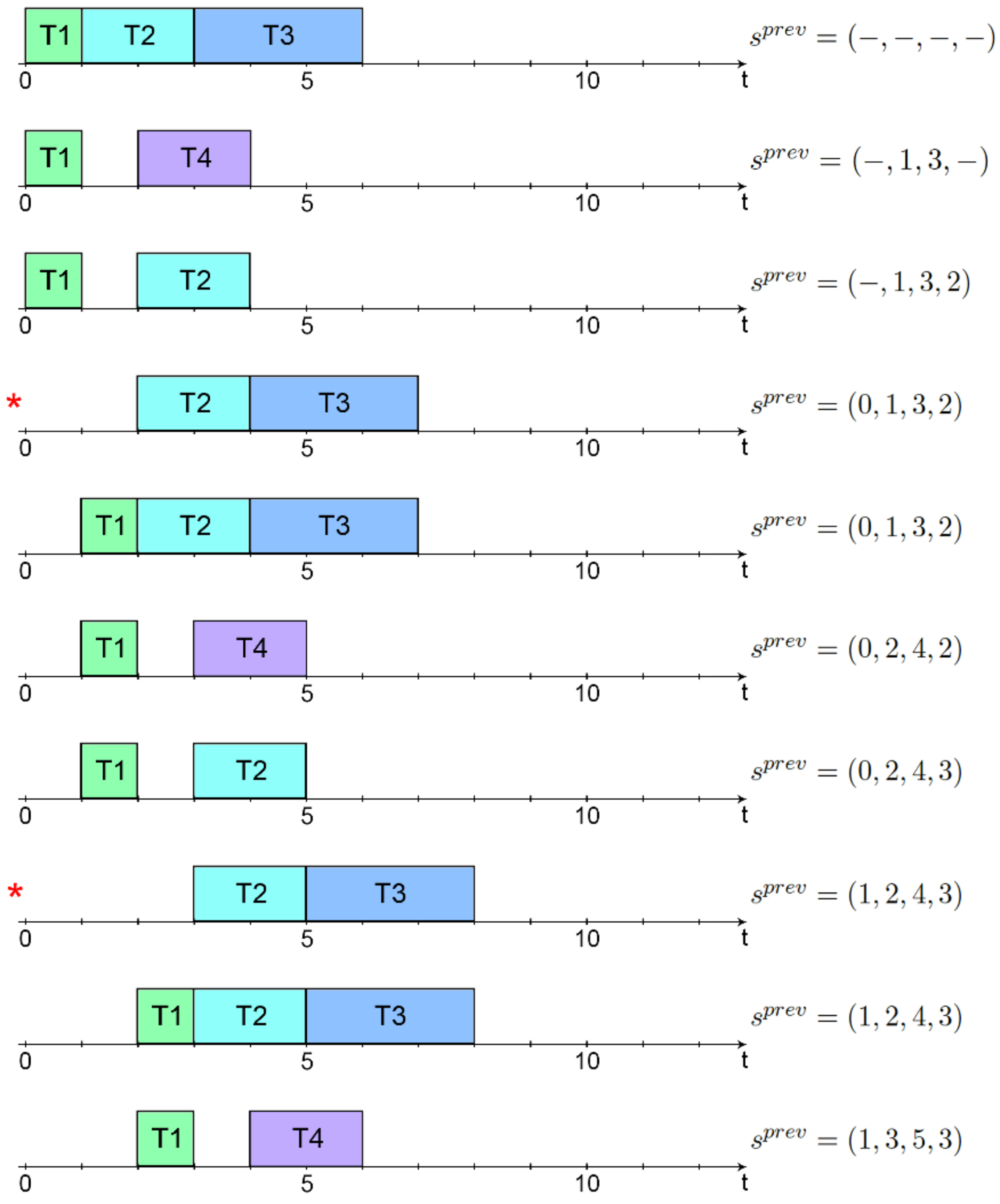


Figure 10 Schedules created by the heuristic algorithm solving the problematic input instance

Schedules created during run of the heuristic algorithm are illustrated in Figure 10. The algorithm schedules tasks T_1, T_2, T_3 with start times $s_1 = 0, s_2 = 1$ and $s_3 = 3$. With respect to temporal constraints, task T_4 have to be scheduled at time 4 but the resource is occupied by T_3. Therefore, T_4 have to be scheduled violently. It is scheduled for the first time and that is the reason why it have to be scheduled in it's earliest start time $s_4 = es_4 = 2$. Tasks T_2 and T_3 are unscheduled because of the resource conflict with newly scheduled task. The highest priority task in the queue is T_2 which have to be scheduled at time 0 according to temporal constraints. Task have to be scheduled violently because the resource is occupied by T_1 at time 0. It is scheduled with $s_2 = s_2^{prev} + 1 = 2$ and conflicting task T_4 is unscheduled. Tasks T_1 and T_3 are scheduled in free slots and T_4 is remaining. Task 4 have to be scheduled violently at time 4 and T_2, T_3 are unscheduled. It is the same situation that happened several steps before and the next steps are in the same manner until the budget of the algorithm is depleted.

The first schedules which are proving that the algorithm gets stuck are marked with a red $*$ label because task start times in the second schedule are incremented by one against start times in the first one. Set of unscheduled tasks is equal in both cases and the second schedule previous start time vector is greater by one against the first schedule previous start time vector. These properties are proving that both schedules are equal with only difference that start times vector and previous start times vector are incremented by one. These properties also holds for successive pairs of schedules.

This situations are repeating until a budget is depleted, then the function $findSchedule$ reports that a feasible solution was not found. Subsequently, function $findSchedule$ is called again with increased upper bound C and the same situation happens again. The algorithm terminates with the no solution found result after several runs. A solution to this problem is proposed in the next section.

3.3 Bulldozing

We have decided to incorporate bulldozing procedure into the the algorithm to overwhelm the problem described in the previous section.

Bulldozing is a procedure dealing with unfeasible schedules by rearranging order of tasks proposed by Tristan and Smith [19]. A group of tasks which are closely related by temporal constraints is pushed to the right by bulldozing. Group of tasks is revolving and changing order of tasks during bulldozing until a feasible order of tasks is found.

3.3.1 The find schedule procedure with bulldozing

Function $findSchedule$ from Algorithm 2 is extended with bulldozing in Algorithm 3. Original latest scheduling time LS_i^{orig} of task i is the latest time when task i could be scheduled regardless of actual start times of other tasks i.e. LS_i^{orig} is not considering the longest path distance constraints in the actual schedule and it is computed only from the original distance to end task $LS_i^{orig} = C - d_{i,n+1}$. Function $findTimeSlot$ is searching in the interval $\langle ES_i, LS_i^{orig}\rangle$ instead of $\langle ES_i, LS_i\rangle$ like in the previous version of the algorithm. This gives a possibility to schedule the task behind the interval defined by temporal constraints. If start time s_i is in the interval $\langle ES_i, LS_i\rangle$, then the task meets all temporal constraints and it is scheduled by the function *scheduleActivity*. If start time s_i is in the interval $(LS_i, LS_i^{orig}\rangle$, then some temporal constraints are unfulfilled and task i have to be scheduled with bulldozing which will also move tasks with unfulfilled temporal constraints. If start time s_i is not found or bulldozing was not successful, then task is scheduled violently as mentioned in Section 3.1.

3.3.2 The bulldozing procedure

Function *scheduleActivityWithBulldozing* depicted in Algorithm 4 schedules task i in s_i which is resource feasible and add all conflicting tasks via temporal constraints to the set B which represents a bulldozer. Tasks from the bulldozer set are scheduled in the while loop. Task i is randomly selected from the bulldozer set. Values of the earliest start time ES_i from temporal constraints, the latest start time LS_i from temporal constraints and the original latest start time LS_i^{orig} are computed. If start time s_i is

Algorithm 3 Iterative resource scheduling algorithm

```
1: procedure findSchedule(C, priority, budget, D)
2:     $s_i \leftarrow -\infty \; \forall i \in \{1, \dots, n\}$
3:     $scheduled \leftarrow \{\}$
4:     while $budget > 0 \; \wedge |scheduled| < n + 2$ do
5:         $i \leftarrow argmax_{\forall j \in T: j \notin scheduled}(priority_j)$
6:         $ES_i \leftarrow \max_{\forall j \in T: j \in scheduled}(s_j + d_{ji})$
7:         $LS_i \leftarrow \min_{\forall j \in T: j \in scheduled}(s_j - d_{ij})$
8:         $LS_i^{orig} \leftarrow C - d_{i,n+1}$
9:         $s_i \leftarrow$ findTimeSlot$(i, ES_i, LS_i^{orig})$
10:        if $s_i \leq LS_i$ then
11:            scheduleActivity$(i, s_i, scheduled)$
12:        else if $s_i \leq LS_i^{orig}$ then
13:            scheduleActivityWithBulldozing$(i, s_i, scheduled, D, budget)$
14:        end if
15:        if $s_i$ not found $\wedge$ bulldozing not successful then
16:            scheduleActivityViolently$(i, s_i, scheduled)$
17:        end if
18:        $budget \leftarrow budget - 1$
19:    end while
20: end procedure
```

Algorithm 4 Buldozing

```
1: procedure scheduleActivityWithBulldozing$(task, start, scheduled, D, budget)$
2:     scheduleActivity$(task, start, scheduled)$
3:     $B \leftarrow$ conflictingActivities$(task)$
4:     while $budget > 0 \wedge B$ is not empty do
5:         $budget \leftarrow budget - 1$
6:         $i \leftarrow$ randomly remove task from $B$
7:         unscheduleActivity(i)
8:         $ES_i \leftarrow \max_{\forall j \in T: j \in scheduled}(s_j + d_{ji})$
9:         $LS_i \leftarrow \min_{\forall j \in T: j \in scheduled}(s_j - d_{ij})$
10:        $LS_i^{orig} \leftarrow C - d_{i,n+1}$
11:        $s_i \leftarrow$ findTimeSlot$(i, ES_i, LS_i^{orig})$
12:        if $s_i$ is not found then
13:            undoBulldozing
14:            return not successful
15:        end if
16:        scheduleActivity$(i, s_i)$
17:        if $s_i > LS_i$ then
18:            $B \leftarrow B \cup$ conflictingActivities$(i)$
19:        end if
20:    end while
21: end procedure
```

not found on the interval $\langle ES_i, LS_i \rangle$, then bulldozing is stopped and all task used in bulldozing are returned back to the original positions i.e. positions where task were scheduled before start of the bulldozing procedure. If start time is found, then task is scheduled at s_i and all temporal constraints conflicting tasks are added to the bulldozer set.

Random task selection is incorporated in order to schedule tasks in different order if bulldozing schedule the same set of tasks multiple times. If the selection of tasks was not random then bulldozing can schedule one set of tasks always in same unfeasible order of start times until the schedule upper bound is reached.

3.4 Problematic example solution

This section shows the solution of the problematic example from Section 3.2 with the original algorithm extended with the bulldozing procedure.

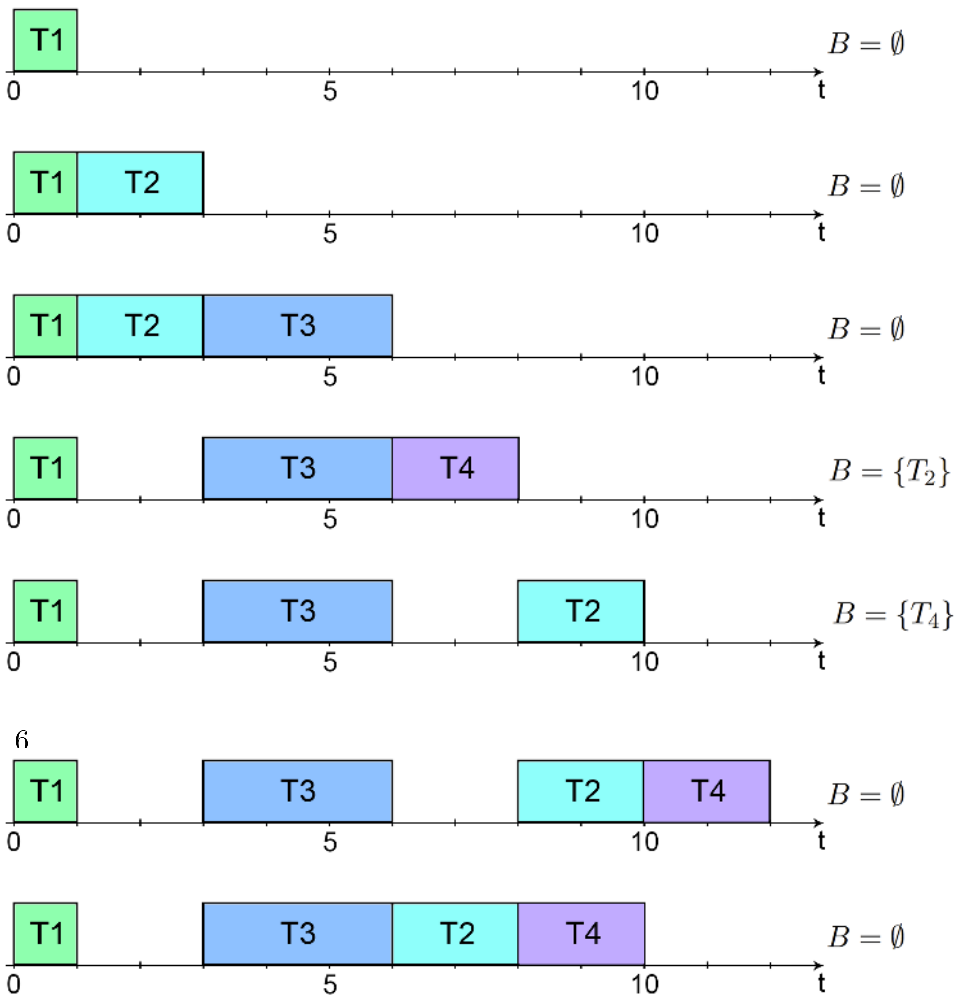


Figure 11 Schedules created by the heuristic algorithm with bulldozing solving the problematic input instance

Schedules created by the heuristic algorithm with the bulldozing procedure solving the problematic input instance are illustrated in Figure 11. Tasks T_1, T_2 and T_3 are scheduled with start times 0, 2, 4 in the same way like in example without bulldozing.

Task T_4 have to be scheduled with $s_4 = 3$ according to temporal constraints but it is not resource feasible therefore bulldozing is started with T_4.

Lets assume that makespan upper bound is 18. Bulldozing schedules T_4 it the first resource feasible time in interval $\langle ES_4, LS_4^{orig} \rangle$ where $ES_4 = 3$ and $LS_4^{orig} = 18-2 = 16$. It is scheduled with $s_4 = 6$ and all task T_2 which is conflicting via temporal constraints is added to the bulldozing set. Consecutively, bulldozing selects T_2 for scheduling. It cannot be scheduled feasibly because resource is occupied at time 4 therefore it is scheduled with $s_2 = 8$ and task T_4 conflicting via temporal constraints is removed from schedule and added to the bulldozing set. Bulldozer selects T_4 for scheduling. The task have to be scheduled at time 10 according to temporal constraints. There is no resource conflict and T_4 is successfully scheduled.

The schedule is left shifted in the end which will move tasks T_2 and T_4 two units to left. The final schedule has makespan $C_{max} = 10$ and it is an optimal solution.

3.4.1 Implemented versions of heuristic algorithm

This section discuss some possible changes in the algorithm design and describes three implemented versions of the heuristic algorithm.

In the first step, the main procedure of the heuristic algorithm depicted in Algorithm 1 computes the longest paths between all tasks in a graph of temporal constraints. Consequently, the algorithm knows the minimal distances of task start times which have to be respected in all feasible schedules. The longest paths are used in the calculation of the earliest scheduling time ES_i and the latest scheduling time LS_i of task i which are calculated considering start times of each scheduled task because the maximal distance to each task is known.

Next, we will discuss determination of the earliest start times and the latest start times based only on temporal constraints not considering the longest path distances between all tasks. Considering only tasks directly connected via temporal constraints, there is a possibility that some directly connected task j is not scheduled therefore it is not considered in computation of the earliest start time of task i. If task j is in some longest path leading from already scheduled task k to task i then start time of task k will not be considered in computation of ES_i and task i will be scheduled in position which will make impossible to schedule task j and when task j becomes to be scheduled, task i have to be unscheduled.

Determination of the earliest start time and the latest start considering the longest paths to all tasks and their start times is more precise than determination based only on temporal constraints leading only from smaller set of tasks. On the other hand, the approach which not considers the longest paths to all tasks have advantage in faster computation because computation of the earliest start time and the latest start times is not enumerating all tasks but only tasks which have a direct edge to task i. We also do not need computation of the longest paths between all tasks. We only need to know the longest paths from dummy start task to all tasks and the longest paths from all tasks to end dummy task. The longest paths from all tasks to end task are important for task priority assignment and the longest paths from all tasks to end task are considered in computation of the latest start time as depicted in Equation (9).

$$LS_i = \min\left(\min_{\forall j \in T:(i,j)\in E}(s_j - l_{ij}), UB - d_{i,n+1}\right) \tag{9}$$

The longest paths from start task to all tasks are considered in computation of the earliest start times as depicted in Equation (10).

$$ES_i = \max(\max_{\forall j \in T:(j,i) \in E}(s_j + l_{ji}), d_{0,i}) \tag{10}$$

Consequently, we do not need Floyd-Warshall algorithm to compute the longest paths between all pairs of tasks and it is replaced by Bellman-Ford algorithm. The first run of Bellman-Ford computes the longest paths from start task to all other tasks. The second run of Bellman-Ford computes the longest paths from all tasks to the end task which is achieved by turning direction of temporal edges into opposite way. Moreover, both runs of Bellman-Ford algorithm can be merged into one run where both values are computed concurrently i.e. when Bellman-Ford select edge (i,j) for update of values then update of the longest path from start is made as $d_{0,j} = \max(d_{0,j}, d_{0,i} + l_{ij})$ and also update of the longest path to end task is made as $d_{i,n+1} = \max(d_{i,n+1}, d_{j,n+1} + l_{ij})$. The replacement of calculation of the earliest start time and latest start time using temporal constraints instead of calculation using the longest paths between all pairs of tasks can lead into better running time especially on sparse graphs with relatively low numbers of temporal constraints.

The second possible change in the algorithm design is depicted in Equation (11). The latest start time is calculated simply as the upper bound minus the maximal distance to the end task. In other words, the latest scheduling time is set to the maximal value which can still lead into a solution with a better value of C_{max} than the best found yet. The latest scheduling time is therefore not considering actual start times of scheduled tasks. The time window for task scheduling is greater than the time window created in version with the latest start time considering start times of scheduled tasks. On the other hand, after successful task scheduling into the time window there can be unfulfilled temporal constraints therefore possible conflicting tasks have to be unscheduled.

$$LS_i = UB - d_{i,n+1} \tag{11}$$

Next, there is descriptions of three versions of the heuristic which were implemented. Proposed versions are combinations of modifications discussed in this section.

Heuristic 1

This version combines calculation of the earliest start time and the latest start time from temporal constraints according to Equations (10) and (9) with bulldozing procedure 3.3. This version is supposed to be able to find a solution in large instances in a reasonable time.

Heuristic 2

The second version of the heuristic algorithm consists of calculation of the earliest start time from temporal constraints according to Equation (10) and the latest start time calculated according to Equation (11). This version of heuristic is also supposed to be a fast algorithm for solving large instances.

Heuristic 3

Floyd-Warshall, original LS_i The third heuristic algorithm version combines calculation of the earliest start time considering the longest paths and current start times of all scheduled tasks as depicted in original proposal in Algorithm 2 and the calculation of the latest start time according to Equation (11). This version is supposed to be slower than the other two vesions but with better ability to find a feasible solution in complicated instances.

3.5 Bounds

For each set of solutions and their evaluations S there exist a lower bound and an upper bound of S. The lower bound is a value smaller or equal to every evaluation of a solution from S. The upper bound is a value greater or equal to every evaluation of a solution from S. If the lower (upper) bound is equal to the minimal (maximal) element of S, then it is called strict. For a set of solutions there could be many algorithms which produce bound values. The ideal algorithm or rule is producing strict bound values. Such algorithm can take unreasonable amount of time and it is with conflict with our requirement for short running computation of bounds. Our goal is to find an algorithm with good ratio between running time and strictness of computed bounds. The bounds proposed in the next two sections are inspired by Brucker et al. [17].

3.5.1 Lower bound

The idea of the lower bound is that a sum of task processing times in a certain criticality level c on a certain resource r is less or equal to resulting schedule makespan. The lower bound can be also seen as transformation of task criticality levels to separate tasks. Then all new tasks which are from the same criticality level and from the same original resource are assigned to same new dedicated resource. The transformed problem is then solved like $PSm, 1|r_j|C_{max}$ problem where $r_j = es_j$.

Tasks are sorted according to increasing earliest start times $es_1 \leq es_2 \leq ... \leq es_n$. A lower bound value is computed for each resource and each criticality level separately. Consequently, there are $|R| \cdot |\mathcal{L}|$ lower bound variables initialized to zero. The first loop is iterating over the sorted task list starting from the first list element. The earliest start time is compared with a value of the lower bound for each criticality level of task i. If a value of the lower bound is greater or equal to the value of the earliest start time then the processing time value is added to the lower bound. It is corresponding to schedule task immediately after the last task. If a value of the lower bound is less than a value of the task earliest start time, then the lower bound is set to $es_i + p_i^c$. It is corresponding to creating a gap in the schedule because task can not be schedule before it's earliest start time. The value of the final lower bound is the maximum value of all partial lower bounds.

Algorithm 5 Lower bound

```
 1: procedure LOWERBOUND($tasks$)
 2:     Sort $tasks$ according to increasing earliest start times.
 3:     $lb_r^c \leftarrow 0 \; \forall r \in R, \forall c \in \{1, .., \mathcal{L}\}$
 4:     for $i \in tasks$ do
 5:         for $c \leftarrow 1$ to $\chi_i$ do
 6:             $lb_{a_i}^c \leftarrow \max(lb_{a_i}^c, es_i) + p_i^c$
 7:         end for
 8:     end for
 9:     $LB \leftarrow \max_{\forall r \in R, \forall c \in \{1..\mathcal{L}\}} lb_r^c$
10: end procedure
```

3.5.2 Upper bound

The upper bound in the Equation (12) is pessimistic but in combination with bulldozing it can be an advantage because bulldozing may need more space to find a feasible order than space created by the strict upper bound.

$$UB = \sum_{\forall i \in T} \max(p_i^{x_i}, \max_{\forall j,k \in T} l_{j,k}) \tag{12}$$

4 Exact algorithms

This chapter proposes two exact algorithms and describes a solution using lazy clause generation SAT solver. The first section proposes disjunctive pairs algorithm which is a modification of the algorithm proposed by Brucker et al. [17] for single machine problem with minimal and maximal time-lags. The algorithm is based on the branch and bound design paradigm fixing precedences between pairs of tasks. Conflict resolution algorithm is proposed in the second section. It is a modification of the algorithm proposed by De Ryck and Herroelen [8] for RCPSP/max realized by the branch and bound design paradigm fixing resource conflict by creation of precedence constraints. The solution using lazy clause generation SAT solver [14] is described in the third section.

4.1 Disjunctive pairs algorithm

The algorithm published by Brucker et al. [17] was proposed for the non-preemtive single machine scheduling problem with minimal and maximal time-lags. This section describes an adaptation of the algorithm to our problem with dedicated resources and mixed-criticality. The algorithm is based on fixing of precedences between pairs of tasks. It is realized in so called disjunctive graph model. The disjunctive graph model is well suited for the representation of a partial solution inside the algorithm. It is described in the next section.

4.1.1 The Disjunctive graph model

A partial solution of the studied scheduling problem can be represented by the disjunctive graph model. It is defined as 3-tuple $G = (T, E, W)$ where

- T is a set of graph vertexes representing tasks.
- E is a set of conjunctive edges. These edges are directed determining the order of tasks. Each edge from vertex i to vertex j has a value l_{ij} defining the minimal distance between task start times i.e. $s_i + l_{ij} \leq s_j$.
- W is a set of disjunctive edges. These edges are undirected. A disjunctive edge also called disjunctive pair represents a pair of tasks with an undecided precedence order. A disjunctive edge may be only between task with the same dedicated resource. Two values w_{ij}, w_{ji} are tied with each disjunctive edge. If a disjunctive edge is transformed to a conjunctive edge defining precedence $T_i > T_j$, then w_{ij} is the value of the new conjunctive edge. If a disjunctive edge is transformed to a conjunctive edge defining precedence $T_j > T_i$, then w_{ji} is the value of the new conjunctive edge. The values w_{ij}, w_{ji} therefore represent processing times of tasks T_i and T_j in common criticality level i.e. time which must be reserved for processing of the first task if tasks are scheduled consecutively.

Transformation of a disjunctive edge into a conjunctive edge is called fixing of a disjunction. A set of conjunctive edges which were created by fixing of a disjunctive edge is called selection. If all disjunctive edges are fixed, then the set of transformed conjunctive edges is called a complete selection.

Next, we will present a scheduling example using a disjunctive graph model. The example consists of nine tasks. Task T_0 is dummy start task and task T_8 is dummy end task. Task definitions are in the list below.

- $T_1 = (2, (1, 2), 1)$
- $T_2 = (2, (2, 4), 1)$
- $T_3 = (1, (3), 1)$
- $T_4 = (2, (2, 4), 2)$
- $T_5 = (1, (2), 2)$
- $T_6 = (1, (2), 2)$
- $T_7 = (2, (1, 2), 2)$

An example of a particular solution is in Figure 12. Conjunctive edges are illustrated with a solid line arrow and disjunctive edges are illustrated with a dotted line. A disjunctive edge is connected with values (w_{ij}, w_{ji}) which are illustrated with the edge. A value of w_{ij} is positioned closer to vertex i and a value of w_{ji} is positioned closer to vertex j. Disjunctive edges are only between tasks sharing the same dedicated resource.

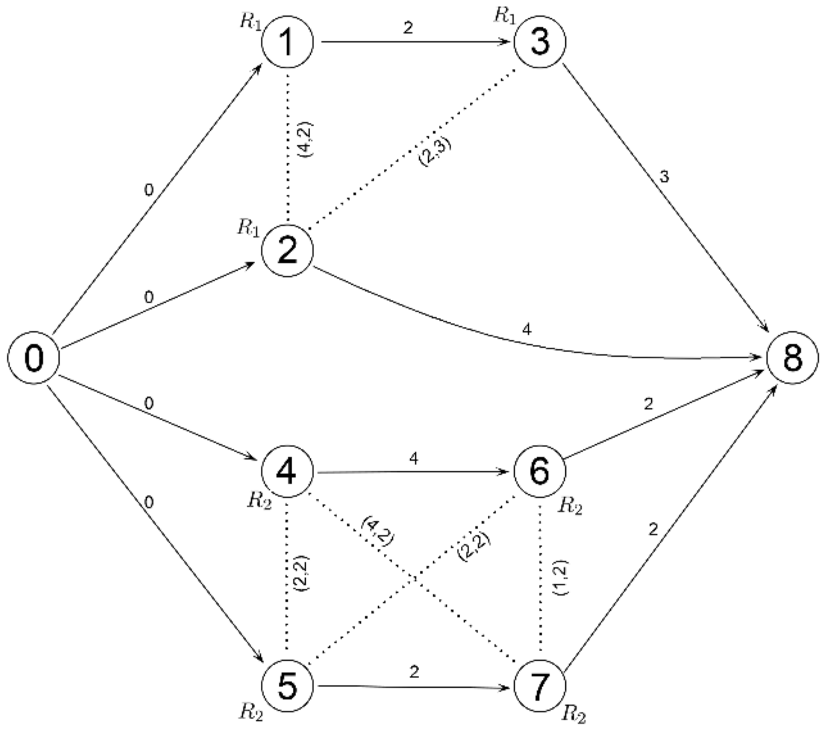


Figure 12 Disjunctive graph

One of the possible solutions is in Figure 13. It is an optimal one. Each disjunctive edge was fixed in one direction and transformed into a conjunctive edge. Makespan of the solution $C_{max} = 7$ because the longest path from start task to end task is equal to seven units.

The schedule corresponding to the solution is illustrated in Figure 14. As we can see, start time of task i is equal to the longest path from start dummy task to task i in the disjunctive graph.

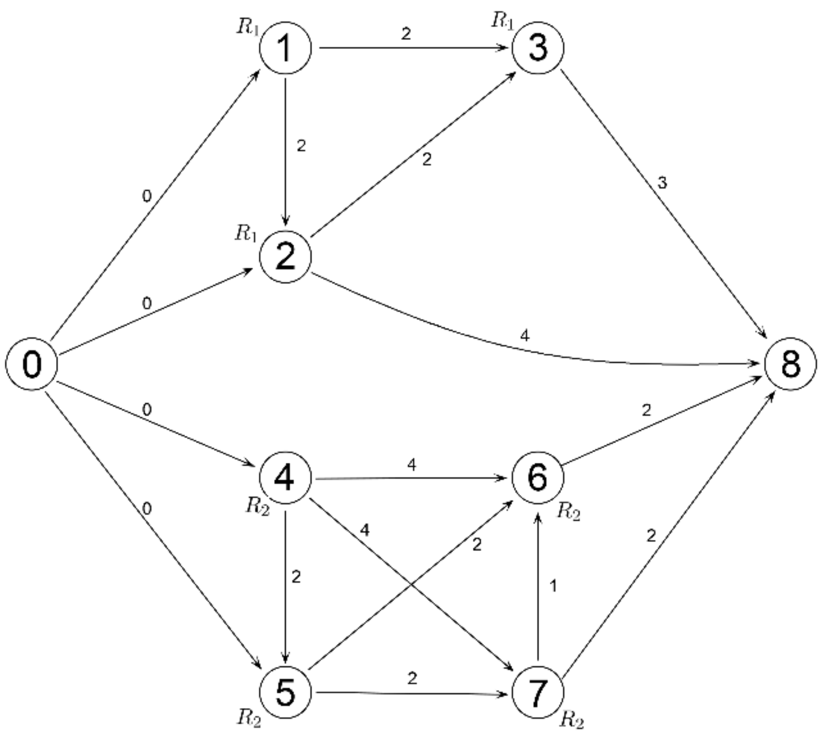


Figure 13 Disjunctive graph with fixed disjunctions

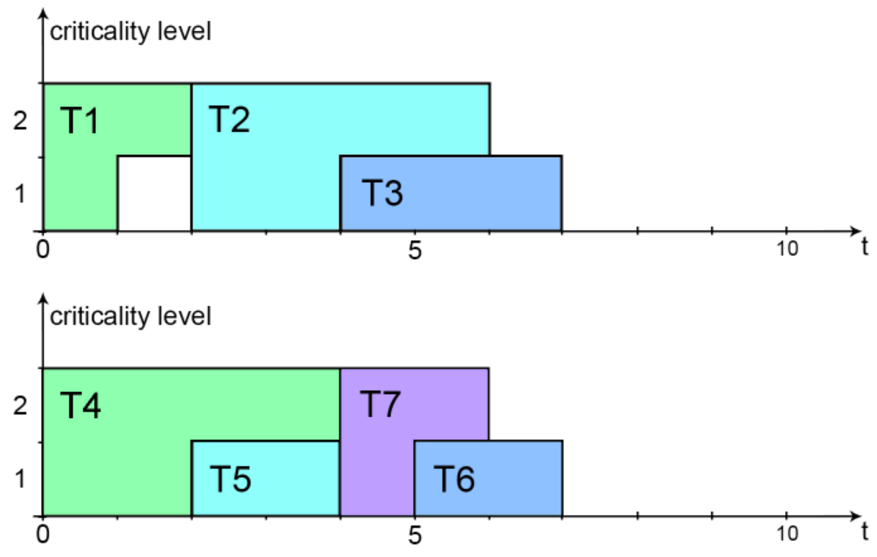


Figure 14 A schedule of the solution from Figure 13

4.1.2 Algorithm desription

In the first step of the algorithm in Algorithm 6, a disjunctive edge is created for each pair of tasks sharing the same resource and not having a conjunctive edge with the value greater then the value of processing time in common criticality level . A matrix of the longest paths is computed from conjunctive edges. A value of the upper bound is computed by a heuristic. If the heuristic does not find a solution, then the upper bound is computed as defined in Equation (12). Successively, procedure *disjunctiveRecursive* is called which realizes branching of the algorithm.

Algorithm 6 Disjunctive pairs algorithm

```
procedure DISJUNCTIVEMAIN($T, E$)
    $W \leftarrow \emptyset$
    for $i, j \in T : ((i,j) \notin E \vee l_{ij} < p_i^{\min(\chi_i,\chi_j)}) \wedge ((j,i) \notin E \vee l_{ji} < p_j^{\min(\chi_i,\chi_j)}) \wedge i < j \wedge a_i = a_j$ do
        $w_{ij} \leftarrow p_i^{\min(\chi_i,\chi_j)}$
        $w_{ji} \leftarrow p_j^{\min(\chi_i,\chi_j)}$
        $W \leftarrow W \cup (i, j, w_{ij}, w_{ji})$
    end for
    $D \leftarrow$ LONGESTPATHS($E$)
    $UB \leftarrow$ UPPERBOUND
    DISJUNCTIVERECURSIVE($T, E, W, D$)
end procedure
```

Algorithm 7 Disjunctive pairs algorithm

```
procedure DISJUNCTIVERECURSIVE($T, E, W$)
    if INFEASIBILITYTEST($T, E, W, D$) proves infeasibility then
        return
    end if
    IMMEDIATESELECTION($T, E, W, D$)
    if |W| = 0 then
        SOLUTIONFOUND(D)
        return
    end if
    $LB \leftarrow$ LOWERBOUND
    if $LB > UB$ then
        return
    end if
    $(i, j, w_{ij}, w_{ji}) \leftarrow$ SELECTBRANCHINGPAIR($W$)
    $D_1 \leftarrow$ ADDEDGETOLONGESTPATHS($D, (i,j), w_{ij}$)
    $D \leftarrow$ ADDEDGETOLONGESTPATHS($D, (j,i), w_{ji}$)
    DISJUNCTIVERECURSIVE($T, E \cup (i,j), W \setminus (i, j, w_{ij}, w_{ji}), D_1$)
    if $UB < -d_{n+1,0}$ then
        $D \leftarrow$ ADDEDGETOLONGESTPATHS($D, (n+1, 0), -UB$)
    end if
    DISJUNCTIVERECURSIVE($T, E \cup (j,i), W \setminus (i, j, w_{ij}, w_{ji}), D$)
end procedure
```

Algorithm 8 Disjunctive pairs algorithm

```
1: procedure SOLUTIONFOUND(D)
2:     if $d_{0,n+1} \leq UB$ then
3:         $bestSolution \leftarrow D$
4:         $UB \leftarrow d_{0,n+1} - 1$
5:     end if
6: end procedure
```

An input of *disjunctiveRecursive* procedure depicted in Algorithm 7 is a disjunctive graph represented by triplet (T, E, W). The disjunctive graph constraints the search space of the current search tree branch. The procedure starts with the infeasibility test described in Section 4.1.5. If the test proves infeasibility, then the branch is pruned.

Immediate selection described in Section 4.1.4 may fix some additional disjunctive edges which drastically reduce a size of the search tree. When immediate selection fixes some disjunctive edge, then the created conjunctive edge is added to the longest path matrix within the immediate selection procedure. If immediate selection leads to a complete selection, then *solutionFound* procedure is called. The lower bound is computed as described in Section 3.5.1. If a value of the lower bound is greater than the upper bound, then the branch is pruned.

A disjunctive pair for branching (i, j) is selected by *selectBranchingPair* procedure. The selected disjunctive edge is attempted to be fixed in both directions which creates two conjunctive edges. A new conjunctive edge have to be added to the longest path matrix. It can be achieved by inserting edge value to the matrix D i.e. $d_{ij} = l_{ij}$ and compute the longest paths by Floyd-Warshall algorithm. Nevertheless, there is a faster solution, which benefits from the fact, that the matrix of the longest paths contained correct values before insertion of the fixed edge. If we have the longest path d_{ab} in graph G and we add edge (c,d) then a value of the longest path d_{ab} is enlarged only if $d_{ac} + l_{cd} + d_{db}$ is greater than d_{ab}. The matrix of the longest paths is recomputed according to Equation (13).

$$d_{ad}^{new} = \max(d_{ad}, d_{ab} + l_{bc} + d_{cd}), \forall a, d \in T \tag{13}$$

This operation takes $\mathcal{O}(n^2)$ time which is less than computation of the longest paths matrix by Floyd-Warshall algorithm in $\mathcal{O}(n^3)$ time. Next, there are two calls of *disjunctiveRecursive*. One call with disjunctive edge fixed in the one direction and the second one in the opposite direction. It corresponds to dividing the current branch of the search tree into two new branches. One of the parameters of the two recursive calls is matrix of the longest paths. A values of the longest paths matrix may change inside the procedure recursion therefore a copy D_1 of the longest paths matrix is created for the first branch and fixed edge (i,j) is inserted into D_1 by *addEdgeToLongestPaths* procedure. The second branch uses original matrix D because there is no need to keep matrix D for further calculations therefore fixed edge (j,i) is inserted straight into original matrix D by *addEdgeToLongestPaths* procedure.

The *solutionFound* procedure depicted in Algorithm 8 simply saves the new solution as the best found if a value of C_{max} is better than a value of the previous one. When a new solution is found then a value of the updated upper bound is inserted into the matrix of the longest paths as edge $(n + 1, 0)$ with a value $-UB$. The longest path matrix update may be done in all levels of the search tree because each level has a separate copy of the longest path matrix. Update necessity of the longest paths matrix with a new upper bound depends on the number of branches already fathomed in a search tree node. Maximally two branches leads from a search tree node. If an update of the upper bound is made in the first branch, then the matrix of the longest paths have to be updated with a new upper bound because it will be used in the second branch. If the upper bound is updated in the second branch, then there is no need for the longest path matrix update because it is not further used.

Obviously, the upper bound update of the longest paths matrices is relatively expensive operation because insertion of an edge to the matrix of the longest paths takes $\mathcal{O}(n^2)$ time and have be done in an each current branch node on the way back to the root node. A depth of the search tree may be up to the number of disjunctive edges and a graph may contains up to $\frac{n(n-1)}{2}$ disjunctive edges maximally. Consequently, an update of an upper bound in all nodes of the current branch may take $\mathcal{O}(n^2 \cdot \frac{n}{2}(n-1)) = \mathcal{O}(n^4)$. On the other hand, in the worst case where G contains the maximum number of disjunctive edges i.e. set of disjunctive edges forms a complete graph, the immediate selection will fix a large number of disjunctive edges thus the branch depth will be considerably smaller. Moreover, if two search tree leaves will produce an upper bound update then the update of the longest paths matrix in nodes which are in common part of the two branches is performed maximally once with the lowest value of the two upper bound updates except for the last common node where an update is performed with the first found upper bound value.

On the other hand, if we consider a situation where an upper bound update occurs in every leaf node of the search tree, then an update of the longest path matrix in each internal search tree node is performed exactly once. There is several time consuming operations in each search tree node: lower bound computation, the longest path matrix copy and the longest path matrix upper bound update. Running time of these operations in each search tree node is $\mathcal{O}(n \log n + n^2 + n^2) = \mathcal{O}(n^2)$.

4.1.3 Branching strategy

A disjunctive pair for branching is selected by procedure *selectBranchingPair* [17]. The procedure uses time windows of possible task start times when a position of second task from disjunctive pair is fixed. More precisely, when start time of task i is fixed with $s_i = 0$, then task k have to be scheduled in the time interval $\langle d_{ik}, -w_{ki}\rangle \cup \langle w_{ik}, -d_{ki}\rangle$. A value of f_i^k defined in Equation (14) is computed as a size of the interval normalized by a sum of processing times in common criticality level.

$$f_i^k = \frac{-d_{ik} - d_{ki} - w_{ik} - w_{ki} + 2}{w_{ik} + w_{ki}} \tag{14}$$

This value is calculated for each disjunctive pair. The median of all f_i^k values is calculated and disjunctive pair (i,k) is selected for branching.

4.1.4 Immediate selection

Immediate selection [17] is a procedure which fixes some disjunctive pairs which are proved to be feasible only in one direction according to Equation (15).

$$d_{ij} > -p_j^{\min(\chi_i,\chi_j)} \wedge a_i = a_j \tag{15}$$

If the longest path from task i to task j leaves no room for processing task j before task i and tasks are assigned to the same dedicated resource, then disjunctive pair have to be fixed in the way that task i precedes task j i.e. a temporal constraint with value $l_{ij} = p_i^{\min(\chi_i \cdot \chi_j)}$ is inserted into G.

4.1.5 Infeasibility test

Two infeasibility tests are used in the algorithm. The first one [17] is based on a detection of cycle with positive length in graph G. If the graph contains positive cycle, then it is not possible to feasibly schedule all tasks. Let T_i and T_j be two arbitrary tasks in T, then following inequalities must hold

$$s_i + d_{ij} \leq s_j \tag{16}$$
$$s_j + d_{ji} \leq s_i \tag{17}$$

it follows

$$s_i + d_{ij} \leq s_j \leq s_i - d_{ji} \tag{18}$$
$$s_i + d_{ij} \leq s_i - d_{ji} \tag{19}$$
$$d_{ij} + d_{ji} \leq 0 \tag{20}$$

A graph G could not include edges which starts and ends in the same task therefore each cycle contains at least two nodes then implication in Equation (21) must holds.

$$\exists i \in T : d_{ii} > 0 \rightarrow \exists j \in T : d_{ij} + d_{ji} > 0 \rightarrow s_i + d_{ij} > s_j \vee s_j + d_{ji} > s_i \tag{21}$$

Existence of positive cycle can be detected by a check of the main diagonal in the matrix of the longest paths therefore it could be done in $\mathcal{O}(n)$ time.

The second feasibility test is modification of the test proposed by Brucker [17]. It is modified for mixed-criticality tasks and dedicated resources . The test is realized as a transformation of our problem to an easier one which can be solved in polynomial time. The transformation is similar to the transformation used in lower bound described in Section 3.5.1. There is created a set of $|R| \cdot |\mathcal{L}|$ dedicated resources and each original resource corresponds to a set of $\mathcal{L}$ new resources each for one criticality level. Each task i is transformed into a set of new tasks with χ_i elements. The new tasks are without criticalities and each new task corresponds to one criticality level of the original task. New task processing time equals to processing time of original task i in the corresponding criticality level. Release time of tasks transformed from task i is equal to the earliest start time of task i and due date is equal to the latest start time of task i. Dedicated resource of new tasks is assigned in accordance with the original resource and the corresponding criticality level.

This transformation is applied on each task in T and let Q be a set of newly created tasks. Set Q is then scheduled like $1|r_j, pmtn|L_{max}$ problem for each new dedicated resource separately. If $L_{max} > 0$ then the branch is proved to be infeasible.

4.2 Conflict resolution algorithm

This section propose a conflict resolution algorithm. The algorithm is a modification of the algorithm proposed by De Ryck and Herroelen [8]. It was originally proposed for RCPSP/max. They proposed several dominance rules which fathom portions of the search tree but these rules are not applicable when resources in our problem have a unit capacity because a unit capacity eliminates the cause of these dominance rules. The algorithm schedules task start times based only on the project network and then resolve resource conflicts by creation of additional precedence constraints.

4.2.1 Algorithm description

The algorithm starts with computation of the longest paths between all tasks using Floyd-Warshall algorithm. Next, there is a preprocessing step. The preprocessing step may add some temporal constraints which tighten solution space and which do not cut off any feasible solution. It is described in Section 4.2.2. An upper bound UB is computed by a heuristic and if the heuristic does not find a feasible solution, then an upper is computed as described in Section 3.5.2. Next, $conflictResolutionRecursive$ procedure is called.

Algorithm 9 Conflict resolution algorithm

```
1: procedure conflictResolution(T,E)
2:     $D \leftarrow$ longestPaths($E$)
3:     preprocessing($T, E, D$)
4:     $UB \leftarrow$ upperBound($T, E, D$)
5:     conflictResolutionRecursive(T,E,D)
6: end procedure
```

Branching of the algorithm is realized in $conflictResolutionRecursive$ procedure. A lower bound is computed as described in Section 3.5.1. If the lower bound is greater than the upper bound, then the branch is pruned because there is no better solution than the best one found yet. Tasks are considered to be scheduled in their earliest start times i.e. $s_i = es_i = d_{0,i}\ \forall i \in T$. Resource conflicts are checked in $findResourceConflict$ procedure. A resource conflict is identified by triplet (t, c, r) where t is the conflict time, c is criticality level where the conflict was found and variable r is a resource where the conflict was found. Time of the resource conflict t is found by checking the schedule from left to right for each resource searching for a time when some resource has more then one task being processed at the same time. Detailed description of $findResourceConflict$ is in Section 4.2.3. If resource conflict time is not found, then the schedule is feasible and procedure $solutionFound$ is called which saves solution and update the upper bound if the new solution is better than the best one found yet.

Procedure $conflictSet$ returns set C of tasks which are in conflict on some resource. The original algorithm [8] resolves resource conflicts by so called minimal delaying alternatives. Minimal delaying alternatives are the minimal sets of tasks which have to be delayed to eliminate a resource conflict in time t. A Minimal Delaying Alternative MDA is delayed by precedence constraints i.e. temporal constraints with a value equal to processing time of source task. Source task i is chosen from set $C \setminus MDA$ and precedence constraints to each element of MDA are created i.e. $E \cup (i, j),\ \forall j \in MDA$ with values $l_{ij} = p_i$ where p_i is processing time of task i in RCPSP/max. Combination of one source task and one minimal delaying alternative is called minimal delaying mode. Branching is based on iteration over all minimal delaying alternatives and all it's delaying modes because we must try to delay all minimal delaying alternatives and try every possible source task of this delay.

In our problem, we have resources with a unit capacity therefore if there is a conflict found on a resource in particular time, then there can remain only one task and other tasks must be delayed. Consequently, $|MDA| = |C| - 1$ and number of minimal delaying modes is equal to size of the conflict set. Precedence constraints from task i to task j are realized as temporal constraints with a value $l_{ij} = p_i^{\min(\chi_i, \chi_j)}$. Branching is then realized as iteration over tasks in the conflict set and iterated task is defined as a source of precedence constraints leading to every other task from the conflict set. An example

Algorithm 10 Conflict resolution algorithm

```
1: procedure conflictResolutionRecursive(T,E,D)
2:     $LB \leftarrow$ lowerBound($T, E, D$)
3:     if $LB > UB$ then
4:         return
5:     end if
6:     $(t, c, r) \leftarrow$ findResourceConflict()
7:     if no resource conflict found then
8:         solutionFound(D)
9:         return
10:    end if
11:    $C \leftarrow$ ConflictSet($t, c, r$)
12:    for $i \in C$ do
13:        $D_{rec} \leftarrow D$
14:        for $j \in C \setminus task$ do
15:            addEdgeToLongestPaths($D_{rec}, (i, j)$)
16:        end for
17:        conflictResolutionRecursive($T, E, D_{rec}$)
18:    end for
19: end procedure
```

of branching is depicted in Figure 15. Tasks T_1, T_2 and T_3 with vectors of processing times

- $P_1 = (2, 4)$
- $P_2 = (1, 2)$
- $P_3 = (2)$

are in conflict on the same dedicated resource in criticality level 1. Conflict is also between tasks T_2 and T_3 in criticality level 2 but only one conflict at a time can be considered. The conflict is resolved by three branches. The first branch has source task T_1 and two precedence constraints are created. The first one leading to T_2 with value $l_{12} = 4$ because common criticality level of T_1 and T_2 is 2 units and processing time of T_1 in criticality level 2 is 4 units. The second one leading to T_3 with value $l_{13} = 2$ because common criticality level of T_1 and T_3 is 1 and processing time of T_1 in criticality level 1 is 2 units. The other two branches with source tasks T_2 and T_3 are created in the same way.

4.2.2 Preprocessing step

In this section we will show how to modify the preprocessing step of the original algorithm [8] for RCPSP/max to our problem with mixed-criticality tasks and dedicated resources with unit capacity. We must define some parameters of RCPSP/max. Let V be a set of non-criticality tasks. Let K be a set of resources and $capacity_k$ be capacity of resource k. Let r_{ik} be a task i requirement on resource k. The preprocessing step in the original algorithm [8] is defined in Equation (22) i.e. the preprocessing step creates an additional temporal constraint if it finds two tasks i, j which cannot be processed in the same time due to resource requirements and task j cannot be completely scheduled before task j due to the longest path d_{ij}.

$$\exists i, j \in V, \exists k \in K : r_{ik} + r_{jk} > capacity_k \ \wedge -p_j < d_{ij} \wedge d_{ij} < p_i \rightarrow d_{ij} = p_i \qquad (22)$$

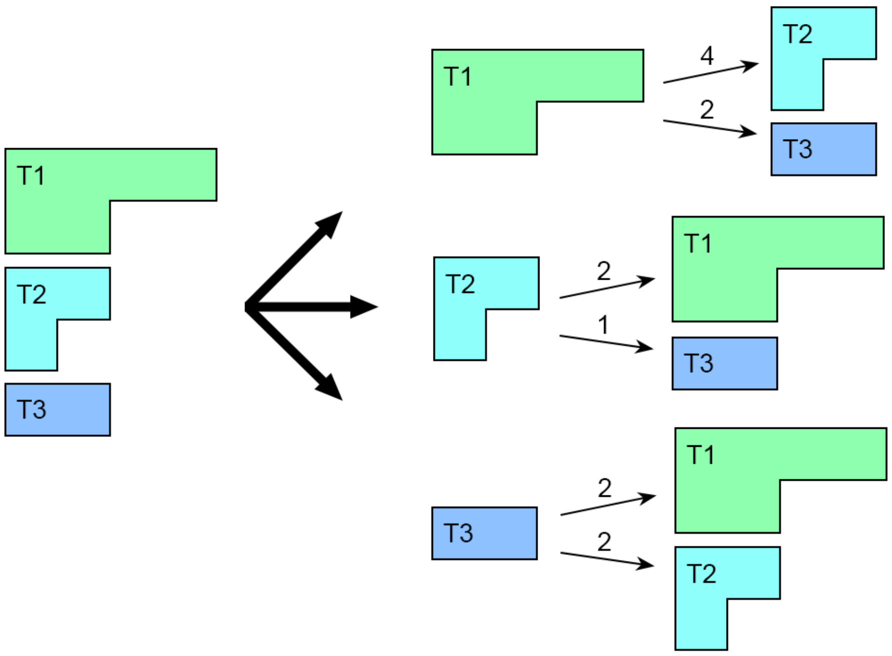


Figure 15 A Conflict resolution algorithm branching scheme

Transformation of Equation (22) to our problem is depicted in Equation (23). Next, we will describe the transformation.

$$\exists i, j \in T : a_i = a_j \ \wedge -p_j^{\min(\chi_i,\chi_j)} < d_{ij} \wedge d_{ij} < p_i^{min \chi_i,\chi_j} \rightarrow d_{ij} = p_i^{min \chi_i,\chi_j} \tag{23}$$

The first condition of Equation (22) holds for every two tasks assigned to the same dedicated resource because we have a unit resource capacity and unit resource requirements therefore $1 + 1 > 1$ and we can substitute this condition by a condition $a_i = a_j$. The second condition of Equation (22) identifies a situation when task j cannot be scheduled before task i because temporal constraints restrict start times in the way that task j can not be completely processed before start time of task i. Transformed for mixed-criticality tasks, the second condition is $-p_j^{\min(\chi_i,\chi_j)} < d_{ij}$. The third condition in Equation (22) warants that the new precedence constraint will be beneficial i.e. that the new precedence constraint will enlarge a value of the longest path d_{ij}. The third condition transformed for mixed-criticality tasks looks like this $d_{ij} < p_i^{min \chi_i,\chi_j}$. These three conditions are corresponding to immediate selection described in Section 4.1.4. The immediate selection 4.1.4 is proposed for disjunctive pairs and if we replace disjunctive pairs with pairs of tasks i, j where $d_{ij} < p_i^{\min(\chi_i,\chi_j)}$, then we are able to realize the preprocessing step by immediate selection procedure because remaining conditions from Equation (23) $a_i = a_j$ and $-p_j^{\min(\chi_i,\chi_j)} < d_{ij}$ are already realized by immediate selection.

4.2.3 Find resource conflict procedure

Procedure *findResourceConflict* is in Algorithm 11. The procedure finds time of a resource conflict if it exists. It is based on principle of resource reservation. Tasks are sorted according to increasing start times in the first step of the procedure. The first loop iterates over sorted tasks. Task i is task selected by the first loop. Next, there is a loop iterating over all criticality levels c of task i. If resource a_i is reserved at time s_i in criticality level c than conflict is found. Otherwise the dedicated resource of task i is reserved in criticality level c until task i completion time in criticality level c i.e. it is reserved until time equal to $s_i + p_i^c$. The procedure was proposed in the way that running time of the procedure is not dependent on a size of start time values.

Algorithm 11 Conflict resolution algorithm

```
1: procedure findResourceConflict
2:     Sort tasks according to increasing start times.
3:     $reservation_{cr} \leftarrow 0 \;\; \forall c \in \{1,..,\mathcal{L}\}, \forall r \in R$
4:     for $i \in tasks$ do
5:         for $c \leftarrow 1$ to $\chi_i$ do
6:             if $reservation_{ca_i} > s_i$ then
7:                 return resource conflict found at $(s_i, c, a_i)$
8:             end if
9:             $reservation_{ca_i} \leftarrow s_i + p_i^c$
10:        end for
11:    end for
12:    return no resource conflict found
13: end procedure
```

4.3 SAT solver with lazy clause generation

In this section we will describe a basic principle of the algorithm proposed by Schutt et al. [14] for solving RCPSP/max with SAT solver using lazy clause generation. We will also describe G12 framework and in the end of the section, we will describe how we used the algorithm in our problem.

4.3.1 Algorithm principle description

Possible start times of task are represented as finite domains. A finite domain is a set of integer numbers representing possible values of some variable. In our case, values of domains are possible start times of tasks. Variables are subjects of constraints which represents relations between variables, therefore domains are pruned by these constraints.

A domain of integers can be transformed into Conjunctive Normal Form Boolean Satisfiability (CNF-SAT) formula [14]. Let x be a variable with domain $DOM_{init}(x) = [l \dots u]$. There is created $2(u-l)+1$ boolean variables. Boolean variables $[\![x \leq l]\!], [\![x \leq l+1]\!], \dots, [\![x \leq u-1]\!]$ and variables $[\![x = l]\!], [\![x = l+1]\!], \dots [\![x = u]\!]$. A boolean formula representing domain is then created as conjunction of these literals. If formula contains literals $[\![x \leq k]\!], [\![x \leq k+1]\!], \dots, [\![x \leq u]\!]$ and does not contains literal $[\![x \leq k-1]\!]$ it means that k is the greatest possible value in domain $DOM(x)$ i.e. $k = u_{current}$. If formula contains literals $\neg[\![x \leq l]\!], \neg[\![x \leq l+1]\!], \dots, \neg[\![x \leq k-1]\!]$ and does not contains literal $\neg[\![x \leq k]\!]$ it means that k is the smallest possible value in domain $DOM(x)$ i.e $k = l_{current}$. If a formula contains literal $\neg[\![x = k]\!]$ it means the $k \notin DOM(x)$. There exists assignments which are not consistent, therefore clauses ensuring consistency are also added to the SAT representation.

Constraints are transformed into a set of propagators which reduce domains of variables. If some propagator reduces a domain of a variable then the propagator generates a clause which explains the propagation and the clause is added to the SAT representation [10]. This process where boolean representation of constraints is generated when it is really needed is called lazy clause generation. Another feature of this approach is no-good learning. The no-good learning uses an implication graph where nodes represents decided literals and edges represents the cause of literal assignment. When a conflict in assignment is found, then a clause explaining the conflict is generated from the implication graph and the clause is added to the SAT representation.

4.3.2 G12 framework

The algorithm is implemented in G12 combinatorial optimization framework [21] [22] created by National ICT Australia. The framework decomposes optimization process into three separate layers. An optimization problem is specified as a set of constraints and variables in the first layer. The second layer [23] transforms problem model to the representation of a specific solver or combination of solvers. The third layer is realized by a set of solvers. The language used in the first layer is called Zinc [24] or it's subset called MiniZinc. It is declarative language for specifying solver independent constraints. The second layer is realized by language called Cadmium which is transforming Minizinc problem definition into a solver specific representation. The third layer is implemented in Mercury language. There are many solvers [27] which can be used in the third layer for example a finite domain constraint solver, an integer linear programming solver or a SAT solver.

4.3.3 Solution description

We specified the mixed-criticality scheduling problem in Minizinc language and used already implemented SAT algorithm with lazy clause generation which is part of G12 framework as one of standard solvers called LazyFD solver. We also implemented program which transforms data from generator of instances (described in Section 5.2) to the Minizinc data format. The program also starts the execution of the solver.

5 Computational results

Computational results of the heuristic algorithm and exact algorithms are presented in this chapter. Moreover, there is a description of the process of instances generation and description of implementation.

5.1 Implementation and testing enviroment

The algorithms and generator were implemented in C# programming language. We have decided to use Mono framework which is an open source implementation of Microsoft's .NET framework capable of running on different operating systems including Linux or Android. We intended to use Mono framework under OS X operating system. In the course of implementation we found out that CPLEX library for OS X does not support C# interface therefore we migrated our solution to Windows operating system and we used Microsoft's .NET framework.

Testing was performed on a computer equipped with 2.13 GHz Intel Core 2 Duo processor and 4 GB of operational memory.

5.2 Data generator

Up to our knowledge, there is no published mixed-criticality generator or a public test set for the mixed-criticality scheduling problem therefore we decided to implement a generator which transforms instances generated by ProGenMax generator [25] to mixed-criticality instances.

5.2.1 ProGenMax generator

ProGenMax is generator created by Schwindt [25] capable of generating instances for several different scheduling problems including RCPSP/max. We adjust the generator to generate instances of RCPSP/max where all resources has unit capacity and tasks are assigned only to one resource with a unit resource requirement.

5.2.2 Mixed-criticality generator

Mixed-criticality generator transforms instances generated by ProGenMax generator into mixed-criticality instances. Instances generated by ProGenMax generator already have graph of temporal constraints, tasks with processing times and each task has one dedicated resource therefore mixed criticality generator have to assign task criticalities and processing times for each criticality level.

Project criticality $\mathcal{L}$ is defined as constant by user before start of generation. The generator selects criticality χ_i for task i randomly from set $\{1, ..., \mathcal{L}\}$. There were implemented two ways of generation of processing time vector. The first approach is called shortening generation approach. Shortening generation approach assigns original processing time p_i^{orig} generated by ProGenMax for RCPSP/max task i to processing time in the maximal criticality level i.e. $p_i^{\chi_i} = p_i^{orig}$. Processing times of other criticality

		$budgetRatio = 1.2$		$budgetRatio = 2$		$budgetRatio = 10$	
Instance	C_{max}^{opt}	C_{max}	t $[s]$	C_{max}	t $[s]$	C_{max}	t $[s]$
h01t10c2	258	NF	0.01	262	0.02	262	0.09
h02t10c6	184	NF	<0.01	263	<0.01	263	<0.01
h03t20c2	419	NF	<0.01	534	<0.01	534	<0.01
h04t20c6	404	NF	<0.01	414	<0.01	414	<0.01
h05t40c2	889	NF	<0.01	1006	<0.01	1006	<0.01
h06t40c6	874	893	<0.01	893	<0.01	893	<0.01
h07t70c2	1860	NF	<0.01	2287	<0.01	2287	0.02
h08t70c6	1875	NF	<0.01	2046	<0.01	2046	0.10
h09t100c2	-	NF	<0.01	3096	<0.01	3096	0.03
h10t100c6	-	NF	<0.01	NF	<0.01	NF	0.02
h11t200c2	-	NF	0.01	5661	0.01	5522	0.01
h12t200c6	-	NF	0.01	NF	0.05	5469	0.01
h13t400c2	-	12067	0.01	12067	0.01	12067	0.01
h14t400c6	-	11976	0.05	11976	0.01	11976	0.01
h15t700c2	-	NF	0.05	NF	0.08	NF	0.32
h16t700c6	-	22533	0.04	22533	0.04	22533	0.03
h17t1000c2	-	NF	0.11	27723	0.09	29676	0.09
h18t1000c6	-	NF	0.12	NF	0.12	NF	0.13

Table 1 Computations results of heuristic 1

levels are then generated iteratively according to Equation (24) for each criticality level $c \in \{\chi_i - 1, ..., 1\}$.

$$p_i^c = floor(\frac{p_i^{c+1}}{rand}) \tag{24}$$

Number $rand$ is randomly generated on interval $\langle 1, randMax \rangle$ where $randMax$ is a user defined constant.

The second approach is elongate generation approach which assigns original processing time p_i^{orig} to processing time in the lowest criticality level i.e. $p_i^1 = p_i^{orig}$ and processing times of higher criticality levels are generated according to Equation (25) iteratively for each criticality level $c \in \{2, ..., \chi_i\}$.

$$p_i^{c+1} = ceil(p_i^c \cdot rand) \tag{25}$$

Shortening generation approach creates instances which have a feasible solution more often than instances created by elongate generation approach. For testing purposes, it is good to know an optimal solution of instance therefore our generator has an option to compute an optimal solution of a generated instance by CPLEX solver.

5.3 Heuristic algorithm results

This section discuss computational results of three versions of the heuristic algorithm. We will use naming convention heuristic 1, heuristic 2 and heuristic 3 as defined in Section 3.4.1.

Heuristics were tested on a set of eighteen instances of different size. Parameters of the instance are encoded in the instance name with pattern $hxtxcx$ where the first x is a number of the instance, the second x is a number of task and the third x is the maximal criticality level therefore an instance with name $h18t1000c6$ has number of

		budgetRatio = 1.2		*budgetRatio* = 2		*budgetRatio* = 10	
Instance	C_{max}^{opt}	C_{max}	t [s]	C_{max}	t [s]	C_{max}	t [s]
h01t10c2	258	NF	0.01	262	0.01	262	0.03
h02t10c6	184	NF	<0.01	263	<0.01	247	<0.01
h03t20c2	419	NF	<0.01	534	<0.01	501	<0.01
h04t20c6	404	NF	<0.01	414	<0.01	414	<0.01
h05t40c2	889	NF	<0.01	1006	<0.01	942	<0.01
h06t40c6	874	893	<0.01	893	<0.01	893	<0.01
h07t70c2	1860	NF	<0.01	2239	<0.01	2126	<0.01
h08t70c6	1875	NF	<0.01	2046	<0.01	2046	0.01
h09t100c2	-	NF	<0.01	NF	<0.01	3282	0.05
h10t100c6	-	NF	<0.01	NF	0.01	NF	0.04
h11t200c2	-	NF	0.01	5492	0.02	5492	0.09
h12t200c6	-	NF	0.01	NF	0.04	611	0.01
h13t400c2	-	12067	0.01	12067	0.01	12067	0.01
h14t400c6	-	11976	0.01	11976	0.01	11976	0.01
h15t700c2	-	NF	0.11	16730	0.19	16730	0.79
h16t700c6	-	22533	0.04	22533	0.04	22533	0.04
h17t1000c2	-	NF	0.2	NF	0.41	29647	0.34
h18t1000c6	-	NF	0.12	NF	0.12	NF	0.12

Table 2 Computational results of heuristic 2

instance equal to 18, it consists of one thousand tasks and maximal criticality level is equal to 6. Each instance has two dedicated resources.

Computational results of heuristic 1, heuristic 2 and heuristic 3 are in Tables 1, 2 and 3 respectively. Second columns presents a value of an optimal solution computed by CPLEX solver. Optimal solutions were computed for instances of size up to seventy tasks. Optimal values of other instances are not not known. Each heuristic version was tested with three different values of budgetRatio. If a heuristic does not found a solution then abbreviation NF is placed into the table cell. All three heuristics were not able to solve most of instances with *budgetRatio* = 1.2. When a value of *budgetRatio* was increased to 2, then heuristics found a feasible solution in most of the instances. When a value of *budgetRatio* was increased to 10, then number of unsolved instances by heuristic 1 and heuristic 2 was only slightly better than number of unsolved instances with *budgetRatio* = 2 but number of unsolved instances by heuristic 3 decreases from four unsolved instances to only one unsolved instance.

Table 4 is comparison of all three versions of the heuristic algorithm with *budgetRatio* = 10. The greatest number of feasible solutions was found by heuristic 3. It was unable to solve only one instance. Heuristic 2 was unable to find solution for two instances and heuristic 1 was unable to solve three instances. The best value of C_{max} was found by heuristic 3 in most of the cases. Heuristic 2 found the best value of C_{max} in three cases.

Running times of heuristic 1 and 2 are significantly better than running time of heuristic 3. This difference is caused by computation of the longest paths between all tasks and by calculation of the earliest start time and the latest start time based on the longest paths to all scheduled tasks as mentioned in Section 3.4.1. Heuristic 1 and 2 are capable of solving very large instances but we did not generate larger instances than one thousand tasks because of running time of ProGenMax generator. Generation of one thousand task instance set lasted more than one hour. Running time of heuristic

		$budgetRatio = 1.2$		$budgetRatio = 2$		$budgetRatio = 10$	
Instance	C_{max}^{opt}	C_{max}	$t\ [s]$	C_{max}	$t\ [s]$	C_{max}	$t\ [s]$
h01t10c2	258	NF	0.01	NF	<0.01	262	0.01
h02t10c6	184	244	<0.01	244	<0.01	244	<0.01
h03t20c2	419	NF	<0.01	450	<0.01	450	<0.01
h04t20c6	404	NF	0.01	414	<0.01	414	<0.01
h05t40c2	889	942	<0.01	942	<0.01	942	0.01
h06t40c6	874	893	<0.01	893	<0.01	874	<0.01
h07t70c2	1860	NF	0.01	2114	0.02	2114	0.05
h08t70c6	1875	NF	0.01	2017	0.02	2017	0.06
h09t100c2	-	NF	0.04	NF	0.06	2819	0.16
h10t100c6	-	NF	0.04	NF	0.05	2755	0.19
h11t200c2	-	NF	0.27	NF	0.32	5685	0.58
h12t200c6	-	NF	0.27	5532	0.31	5157	0.43
h13t400c2	-	12067	1.96	12067	2.06	12067	2.96
h14t400c6	-	11976	1.93	11976	2.03	11976	2.81
h15t700c2	-	NF	9.77	17072	10.91	17072	14.93
h16t700c6	-	22533	9.91	22533	10.28	22533	13.22
h17t1000c2	-	NF	27.75	26764	30.31	26764	38.62
h18t1000c6	-	NF	27.74	NF	31.48	NF	37.31

Table 3 Computational results of heuristic 3

1 is less than half a second for each instance and running time of heuristic 2 is less than one second for each instance. Heuristic 3 running time was under one minute for each instance. It is still a fast heuristic algorithm even though the other two heuristic algorighms are faster because instances of size about one thousand are considered to be large and running time under one minute is good result.

5.4 Exact algorithms results

This section presents computational results of exact algorithms. Algorithms were tested on a set consisting of seventeen instances. Parameters of instances are encoded into instance name with pattern $etxcx$ where the first x is number of tasks and the second x is the maximal number of criticality levels. For example, an instance with name $et50c6$ contains fifty tasks and the maximal criticality level is equal to six. Number of resources is fixed to two resources in all instances and we use two values of the maximal criticality level. A running time limit for algorithm execution time on one instance was set to ten minutes. If an algorithm was stopped due to time limit, then a value of running time is replaced by a dash character.

Two implemented exact algorithms were tested together with integer linear programming solver CPLEX and with SAT/FD lazy clause generation solver which was described in Section 4.3. Computational results of exact algorithms are depicted in Table 5. Conflict resolution algorithm was better than disjunctive pairs algorithm. It was able to solve instances with 54 tasks. Disjunctive pairs algorithm was able to solve instances up to size of 46 tasks. An instance with name et50c2 was solved by Disjunctive pairs algorithm in short time because it has no feasible solution which was revealed by positive cycle test in the first steps of the algorithm.

An integer linear programming solver and SAT solver with lazy clause generation

		Heuristic 1		Heuristic 2		Heuristic 3	
Instance	C_{max}^{opt}	C_{max}	t $[s]$	C_{max}	t $[s]$	C_{max}	t $[s]$
h01t10c2	258	262	0.09	262	0.03	262	0.01
h02t10c6	184	263	<0.01	247	<0.01	**244**	<0.01
h03t20c2	419	534	<0.01	501	<0.01	**450**	<0.01
h04t20c6	404	414	<0.01	414	<0.01	414	<0.01
h05t40c2	889	1006	<0.01	942	<0.01	942	0.01
h06t40c6	874	893	<0.01	893	<0.01	**874**	<0.01
h07t70c2	1860	2287	0.02	2126	<0.01	**2114**	0.05
h08t70c6	1875	2046	0.1	2046	0.01	**2017**	0.06
h09t100c2	-	3096	0.03	3282	0.05	**2819**	0.16
h10t100c6	-	NF	0.02	NF	0.04	**2755**	0.19
h11t200c2	-	5522	0.01	**5492**	0.09	5685	0.58
h12t200c6	-	5469	0.01	6111	0.01	**5157**	0.43
h13t400c2	-	12067	0.01	12067	0.01	12067	2.96
h14t400c6	-	11976	0.01	11976	0.01	11976	2.81
h15t700c2	-	NF	0.32	**16730**	0.79	17072	14.93
h16t700c6	-	22533	0.03	22533	0.04	22533	13.22
h17t1000c2	-	29676	0.09	**29647**	0.34	26764	38.62
h18t1000c6	-	NF	0.13	NF	0.12	NF	37.31

Table 4 A comparison of heuristic algorithm versions

achieved better computational results than our algorithms. Conflict resolution algorithm was able to solve instance et54c6 in 38 seconds whereas the integer linear programming solver and the SAT solver were able to solve the instance under six seconds.

The instance with eighty tasks was additionally created in order to compare the integer linear programming solver and the SAT solver. The instance show that the SAT solver is able to achieve better results than the integer linear programming solver on instances with more tasks. Our algorithms were outperformed by the integer linear programming solver and the SAT solver. CPLEX solver is probably the most advanced commercial integer linear programming solver and techniques used by this solver are not published in the scientific community. Approach based on SAT/FD lazy clause generation solver is state of the art in RCPSP/max. We proved that SAT solver is a powerful approach also in the mixed-criticality scheduling problem.

Instance	Disjunctive pairs alg. t $[s]$	Conflict resolution alg. t $[s]$	ILP t $[s]$	SAT/FD t $[s]$
et26c2	0.15	0.11	0.37	1.62
et26c6	0.01	0.01	0.2	1.64
et30c2	0.44	0.08	0.19	1.22
et30c6	0.01	<0.01	0.3	0.98
et34c2	0.11	0.04	0.24	1.47
et34c6	0.23	0.03	0.34	0.83
et38c2	1.24	0.09	0.47	0.92
et38c6	0.13	0.09	0.42	1.71
et42c2	0.07	0.82	0.69	1.68
et42c6	0.02	0.01	0.48	1.36
et46c2	2.37	2.57	0.71	2.07
et46c6	-	22.93	1.08	2.3
et50c2	-	41.65	2.71	3.37
et50c6	0.32	0.17	2.96	1.98
et54c2	-	14.44	1.46	2.46
et54c6	-	37.48	5.88	5.26
et80c6	-	-	377.89	80.44

Table 5 Computational results of exact algorithms

6 Conclusion

This work propose offline algorithms for the mixed-criticality scheduling problem with positive and negative time lags and non-preemtive tasks with dedicated resources. Up to our knowledge, these are the first algorithms in this area. An inspiration for our algorithms comes from different scheduling problems. We proposed one heuristic algorithm which is a modification of the Iterative resource scheduling algorithm proposed by Hanzálek and Šůcha [18] for RCPSP/max. It was implemented in three versions differing in some algorithm aspects described in detail in Section 3.4.1. We showed a problematic example of the approach used in the first heuristic version and incorporate bulldozing procedure which was proposed by Smith [19] for Squeaky wheel optimization of RCPSP/max. Bulldozing effectively solved the problem illustrated in the problematic example. The first and the second heuristic versions turns out to be very quick algorithms solving really big instances. One thousand task input instances were solved under one second and possibly larger instances can be considered. The third version of the heuristic algorithm is slower solving one thousand task instances under one minute but with a greater ability to find a feasible solution in complicated instances.

We proposed two exact algorithms based on a branch and bound design paradigm. The first one is Disjunctive pairs algorithm. It is a modification of the algorithm proposed by Brucker et al. [17] for the single-machine scheduling problem with minimal and maximal time lags. Conflict resolution algorithm is the second exact algorithm which is inspired by the algorithm proposed by DeRyck et al. [8] for RCPSP/max. We were also interested in approach using representation of scheduling problem as finite domain and transformation of finite domain to boolean satisfiability problem which is the state of the art approach in RCPSP/max problem [14]. Transformed problem is then solved with SAT solver. This solution is not incorporated in contributions of our work because we formulated the mixed-criticality scheduling problem in Minizinc modeling language but the transformation to boolean satisfiability problem and the solver are already implemented in optimization framework G12 [22] which we were using.

Testing instances were generated by ProGenMax generator [25] as instances of the project scheduling problem with minimal and maximal time lags and then transformed to mixed-criticality instances by our generator.

Computational results were compared with integer linear programming solver CPLEX. Disjunctive pairs algorithm were able to successfully find an optimal solution on instances of size above forty tasks. Conflict resolution algorithm results were better. It was able to find an optimal solution for instances of size above fifty tasks. Our algorithms were outperformed by CPLEX solver and by the SAT/FD solver. The best results were achieved with SAT/FD solver which outperformed CPLEX solver on bigger instances.

Bibliography

[1] S. Vestal. “Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance”. In: *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International.* Dec. 2007, pp. 239–243. DOI: 10.1109/RTSS.2007.47.

[2] Vincenzo Bonifaci Sanjoy Baruah. “Scheduling Real-Time Mixed-Criticality Jobs”. In: *IEEE Transactions* 61 (8 Aug. 2012).

[3] O.R. Kelly, H. Aydin, and Baoxian Zhao. “On Partitioned Scheduling of Fixed-Priority Mixed-Criticality Task Sets”. In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on.* Nov. 2011, pp. 1051–1059. DOI: 10.1109/TrustCom.2011.144.

[4] D. de Niz, K. Lakshmanan, and R. Rajkumar. “On the Scheduling of Mixed-Criticality Real-Time Task Sets”. In: *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE.* Dec. 2009, pp. 291–300. DOI: 10.1109/RTSS.2009.46.

[5] Zdeněk Hanzálek, Tomáš Tunys, and Přemysl Šůcha. “Non-preemptive mixed-criticality match-up scheduling problem”. In: ().

[6] J. Blazewicz, J.K. Lenstra, and A.H.G.Rinnooy Kan. “Scheduling subject to resource constraints: classification and complexity”. In: *Discrete Applied Mathematics* 5.1 (1983), pp. 11–24. ISSN: 0166-218X.

[7] M. Bartusch, R.H. Möhring, and F.J. Radermacher. “Scheduling project networks with resource constraints and time windows”. English. In: *Annals of Operations Research* 16.1 (1988), pp. 199–240. ISSN: 0254-5330. DOI: 10.1007/BF02283745. URL: http://dx.doi.org/10.1007/BF02283745.

[8] Bert De Reyck and Willy Herroelen. “A branch-and-bound procedure for the resource-constrained project scheduling problem with generalized precedence relations”. In: *European Journal of Operational Research* 111 (1998), pp. 152–174.

[9] Toan Phan-Huy Ulrich Dorndorf Erwin Pesch. “A time-oriented branch-and-bound algorithm for resource-constrained project scheduling with generalised precedence constraints”. In: *Management Science* (Oct. 2000).

[10] Olga Ohrimenko, PeterJ. Stuckey, and Michael Codish. “Propagation via lazy clause generation”. English. In: *Constraints* 14.3 (2009), pp. 357–391. ISSN: 1383-7133. DOI: 10.1007/s10601-008-9064-x. URL: http://dx.doi.org/10.1007/s10601-008-9064-x.

[11] Andrei Horbach. “A Boolean satisfiability to the resource-constrained project scheduling problem”. English. In: *Annals of Operations Research* 181.1 (2010), pp. 89–107. ISSN: 0254-5330. DOI: 10.1007/s10479-010-0693-2.

[12] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. English. In: *Theory and Applications of Satisfiability Testing.* Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-20851-8. DOI: 10.1007/978-3-540-24605-3_37. URL: http://dx.doi.org/10.1007/978-3-540-24605-3_37.

[13] João P. Marques Silva, Inês Lynce, and Sharad Malik. "Conflict-Driven Clause Learning SAT Solvers." In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Sept. 16, 2009, pp. 131–153. ISBN: 978-1-58603-929-5. URL: `http://dblp.uni-trier.de/db/series/faia/faia185.html#SilvaLM09`.

[14] Peter J. Stuckey Andresas Schutt Thibaut Feydy. "Solving RCPSP/max by lazy clause generation". In: *Journal of Scheduling* 16 (2013).

[15] Andreas Schutt et al. "Why Cumulative Decomposition Is Not as Bad as It Sounds". English. In: *Principles and Practice of Constraint Programming - CP 2009*. Ed. by IanP. Gent. Vol. 5732. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 746–761. ISBN: 978-3-642-04243-0. DOI: `10.1007/978-3-642-04244-7_58`. URL: `http://dx.doi.org/10.1007/978-3-642-04244-7_58`.

[16] Pinson E. Carlier J. "An algorithm for solving the job-shop problem". In: *Management Science* 30.2 (Feb. 1989).

[17] Peter Brucker, Thomas Hilbig, and Johann Hurink. "A branch and bound algorithm for a single-machine scheduling problem with positive and negative time-lags". In: *Discrete Applied Mathematics* 94.1–3 (1999). Proceedings of the Third International Conference on Graphs and Optimization GO-III, pp. 77–99. ISSN: 0166-218X. DOI: `http://dx.doi.org/10.1016/S0166-218X(99)00015-3`. URL: `http://www.sciencedirect.com/science/article/pii/S0166218X99000153`.

[18] Zdeněk Hanzálek and Přemysl Šůcha. "Time symmetry of project scheduling with rime windows and take-give resources". In: *Proc. of the 4th Multidisciplinary International Scheduling Conf.: Theory and Applications (MISTA)* (Aug. 2009).

[19] Tristan B. Smith. "An effective algorithm for project scheduling with arbitrary temporal constraints". In: *In: Proceedings of the 19 th National Conference on Artificial Intelligence. (2004*. 2004, pp. 544–549.

[20] David E. Joslin and David P. Clements. *"Squeaky Wheel" Optimization*. 1999.

[21] Mark Wallace. "G12 - Towards the Separation of Problem Modelling and Problem Solving". English. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Ed. by Willem-Jan van Hoeve and JohnN. Hooker. Vol. 5547. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 8–10. ISBN: 978-3-642-01928-9. DOI: `10.1007/978-3-642-01929-6_2`. URL: `http://dx.doi.org/10.1007/978-3-642-01929-6_2`.

[22] PeterJ. Stuckey et al. "The G12 Project: Mapping Solver Independent Models to Efficient Solutions". English. In: *Logic Programming*. Ed. by Maurizio Gabbrielli and Gopal Gupta. Vol. 3668. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 9–13. ISBN: 978-3-540-29208-1. DOI: `10.1007/11562931_3`. URL: `http://dx.doi.org/10.1007/11562931_3`.

[23] Reza Rafeh et al. "From Zinc to Design Model". English. In: *Practical Aspects of Declarative Languages*. Ed. by Michael Hanus. Vol. 4354. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 215–229. ISBN: 978-3-540-69608-7. DOI: `10.1007/978-3-540-69611-7_14`. URL: `http://dx.doi.org/10.1007/978-3-540-69611-7_14`.

[24] Kim Marriott et al. "The Design of the Zinc Modelling Language". In: *Constraints* 13.3 (Sept. 2008), pp. 229–267. ISSN: 1383-7133. DOI: `10.1007/s10601-008-9041-4`. URL: `http://dx.doi.org/10.1007/s10601-008-9041-4`.

[25] Christoph Schwindt and Christoph Schwindt. *ProGen/max: A New Problem Generator for Different Resource-Constrained Project Scheduling Problems with Minimal and Maximal Time Lags*. Tech. rep. 1995.

[26] Andreas Fest et al. *Resource-Constrained Project Scheduling With Branching Scheme Based On Dynamic Release Dates*. 1999.

[27] Ralph Becket et al. "The Many Roads Leading to Rome: Solving Zinc Models by Various Solvers." In: *ModRef'08: 7th International Workshop on Constraint Modelling and Reformulation*. Sydney Australia, Sept. 2008.

[28] S. Baruah and S. Vestal. "Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications". In: *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*. July 2008, pp. 147–155. DOI: `10.1109/ECRTS.2008.26`.