Flexible Services and Manufacturing Journal (2020) 32:913–948
https://doi.org/10.1007/s10696-019-09359-2

Scheduling approach for on-site jobs of service providers

Tibor Holczinger[1] · **Olivér Ősz**[2] · **Máté Hegyháti**[2]

Published online: 25 May 2019

Abstract

Nowadays the successful operation of a company is unimaginable without fast and reliable communication. As a result, so-called Communication Service Providers play an important role in today's business life. Their orders have to be carried out promptly and dependably, let them be requests for new installations, modifications, or maintenance tasks. These orders have to be performed at different locations and they often have deadlines or strict starting times. Violating such a timing requirement usually implies penalties. In this paper, scheduling problems arising at a Hungarian service provider are examined. At this company, orders are decomposed into smaller tasks, which can be performed by specially trained personnel. Transportation of these specialists contributes a lot to the costs and to the complexity of their scheduling, as well. The goal is to minimize the overall cost of satisfying all orders within the given time horizon with the available assets of the company. The proposed approach relies on the S-graph framework, which has been applied to various production scheduling problems in the literature. In addition to an unambiguous and sound S-graph model of the examined problem, slight modifications of the scheduling algorithms for cost minimization, and new bounding methods have been developed. Several of such bounds have been provided and tested for performance and scalability over a large number of generated examples. The sensitivity of the approach for certain problem features has also been examined.

Keywords S-graph · Scheduling · Time constrained jobs · Service provider

✉ Tibor Holczinger
holczinger.tibor@uni-pen.hu

[1] Department of Applied Informatics, University of Pannonia, Zrínyi u. 18., Nagykanizsa 8800, Hungary

[2] Department of Information Technology, Széchenyi István University, Egyetem tér 1., Győr 9026, Hungary

1 Introduction

Scheduling problems appear almost everywhere from the management of public transportation through CPU time allocation to the chemical production industry. Although these problems share the basic goals of scheduling, i.e. assigning resources and time intervals to tasks, their characteristics have significant differences, and as a result, the applied approaches are very diverse. The specific case of the operation of Communication Service Provider teams has not yet been investigated directly in the literature, however, the problem shows strong similarities with production scheduling and vehicle routing problems (VRP). In the examined problem, specialists of a company are organized into small teams with a car and have to carry out jobs at various locations. The orders may have various timing constraints, such as a deadline or an earliest starting time. Each order consists of several non-preemptive tasks that have arbitrary dependencies between them and may require the expertise of different teams. The general goal is to minimize the overall cost of a workshift which entails transportation costs, liquidated damages, and operation costs.

It can be said that the examined problem is an integration of batch process scheduling and VRP as both problem classes are special cases of it: If all the orders arose at the same location, the problem would be equivalent to a general batch process scheduling problem. On the other hand, in the special case of single-stage orders, the problem would be a VRP with maximum traveling distance, service time (Juan et al. 2009) and time window constraints. There is a number of published approaches for both batch process scheduling (Hegyháti and Friedler 2010) and VRP (Braekers et al. 2016), that could serve as a basis for solving the problem under investigation. In this paper, the scheduling of Communication Service Provider teams is considered as a generalization of the batch process scheduling problem and addressed with an extended approach from that field.

The VRP formulation, as a generalization of the Traveling Salesman Problem, was introduced by Dantzig and Ramser (1959). The original VRP aim is to generate routes for identical vehicles to fulfill the demands of a set of customers. Each customer must be visited exactly once by one vehicle and each vehicle starts from the depot and arrives back at the depot. From the point of view of the paper, the most important variant of the VRP is the VRP with Time Windows (VRPTW) (Bräysy and Gendreau 2005a, b; Gendreau and Tarantilis 2010), where each customer must be visited in a defined time interval. Because of the NP hard nature of the VRP, most of the state-of-the-art methods use heuristics (Reil et al. 2018) or metaheuristics (Harzi and Krichen 2017).

Batch process scheduling gained the interest of both chemical engineers and optimization experts in the 90s and since then the presented approaches have been developed by those groups (Méndez et al. 2006). In its simplest form, the goal in a batch process scheduling problem is to find the best assignment of units to non-preemptive tasks satisfying dependencies and storage related constraints while minimizing the total production time, i.e., makespan. The effort of researchers in the last few decades had two main directions: developing more and

more efficient algorithms for the problem classes already tackled, and to address more and more general, integrated problems. Most of the related papers formulate the problem as a Mixed Integer Linear Programming (MILP) model and apply a general purpose MILP solver such as CPLEX, Gurobi or COIN-OR to solve them. Although these models are rather flexible to extensions and many of them have low CPU needs, the modeling techniques used to improve solution performance have some flaws. Some of the developed approaches may lead to suboptimal solutions as reported by Shaik and Floudas (2009), or even to schedules containing infeasible cross-transfers as shown by Ferrer-Nadal et al. (2008) and Hegyháti et al. (2009). Another well-researched technique is the application of directed graph based models and specialized Branch-and-Bound (B&B) algorithms. The S-graph (Sanmartí et al. 2002) relies on such tools and has been developed with acceleration techniques and generalizations for the last 3 decades. Alternative graphs (D'Ariano et al. 2007) apply a similar mathematical model and branching strategy. The third notable direction in batch process scheduling is based on the state space enumeration of the production facility by either Linear Priced Timed Automata (Schoppmeyer et al. 2012) or Timed Place Petri Nets (Ghaeli et al. 2005).

There are numerous papers in literature focusing on solving transportation problems with various techniques, such as stochastic programming (Larsen et al. 2014), constraint programming (Masoud et al. 2011) or alternative graph representation (Samà et al. 2017). Stochastic programming approaches are used in cases where the precise values of some parameters are not known in advance because of the uncertainty of the traffic (Larsen et al. 2014), or robust solutions are needed for production scheduling problems (Urgo and Váncza 2018). In VRPs, changeover times are usually sequence-dependent, which is also frequent in production scheduling. For example, Guinet (1993) proposes a heuristic extension of the Hungarian method to solve a VRP problem, and Ruiz and Andrés-Romano (2011) shows a MIP model with dispatching heuristics.

In this paper, an extension and application of the S-graph framework is presented to solve the scheduling of Communication Service Provider teams, motivated by an ongoing R&D project with a Hungarian service provider. The framework was originally developed to solve makespan minimization problems for batch processes (Sanmartí et al. 2002). Later it was successfully applied to solve large-scale industrial problems, as well (Adonyi et al. 2008). Transportation, which is a key element of the investigated problem, has been tackled with the S-graph framework to solve train scheduling problems (Adonyi et al. 2010), and the scheduling of transporter robots in Automated Wet-etch Stations (Hegyháti et al. 2014). Many of the later developments and extensions of the framework provide the basis for the approach applied in this paper. A very important aspect of batch scheduling problems is the storage policy of the process. The S-graph extensions developed to tackle the so-called Zero Wait (ZW) and Limited Wait (LW) storage policies (Hegyháti et al. 2011) will be crucial in modeling the strict timing constraints of the orders of the service provider.

The proposed approach entails a slight modification of formerly published S-graph algorithms, a sound S-graph representation of service provider scheduling

problems and several newly developed bounding procedures. The innovative aspect of these developments is to provide an integrated method to tackle both transportation and local scheduling aspects with time windows and complex cost evaluations. The proposed approach can help the planning department of the mentioned company or other companies in the same business to systematically minimize costs, which often inherently maximizes client satisfaction as delay costs can be dominant in the objective.

In Sect. 2, the exact definition of the problem is given, then Sect. 3 provides a brief introduction to the S-graph framework which is needed as the basis of the proposed approach detailed in Sect. 4. The method was tested on random-generated instances, the results are given in Sect. 5. Sections 6 and 7 present the potentials of the proposed approach beyond the presented problem class and summarize the results, respectively. To facilitate readability and the replication of the results, a nomenclature (Appendix 1), a formal summary of the proposed model (Appendix 2) and a pseudo-code for the approach (Appendix 3) is provided in the Appendix.

2 Problem definition

The orders of a service provider company are not arbitrary in structure but follow some predefined templates. The set of orders and templates are denoted by $\mathcal{O}$ and $\mathcal{T}$, respectively. For each order $O \in \mathcal{O}$, $T_O \in \mathcal{T}$ denotes the template for that order. Each template $T \in \mathcal{T}$ entails a set of uninterruptible tasks, $\mathcal{I}_T$, and the mandatory dependencies between them: $D_T \subset \mathcal{I}_T \times \mathcal{I}_T$. If $(I, I') \in D_T$, task I' can only start to be performed after task I has already been completed.

The orders have to be carried out at various locations. The set of possible locations is denoted by $\mathcal{L}$, and it always contains the depot, L^D, where cars and specialists reside between shifts. All tasks of an order $O \in \mathcal{O}$ have to be performed at the same location which is denoted by L_O, where $L_O \in \mathcal{L}$.

At the current stage of our research it is assumed that specialists have already been assigned to cars. Thus, instead of listing competencies and providing the relations between tasks to competencies, competencies to specialists, and specialists to cars, problem description can be simplified by defining the cars that have a suitable specialist assigned for a given task. Let $\mathcal{C}$ denote the set of these cars, and $\mathcal{C}_I \subseteq \mathcal{C}$ denote the cars that have suitable personnel to carry out task I.

As usual, timing constraints provide the core of the problem. None of the cars can start a journey before the start of the shift, denoted by $t^{shiftstart}$, and similarly, all of the cars must arrive back at the depot (L^D) by the end of the shift, $t^{shiftend}$. These constraints are strict, i.e., they cannot be violated in any circumstances.

The time needed for a car to travel from one location to another is given as an input parameter, denoted by $t^{travel}(L_1, L_2)\ \forall L_1, L_2 \in \mathcal{L}$. Note that $t^{travel}(L_1, L_2)$ may differ from $t^{travel}(L_2, L_1)$, however, $t^{travel}(L, L) = 0$ for all $L \in \mathcal{L}$. Besides transportation, other important time consuming activities are the tasks themselves. $t^{perform}(I)$ denotes the time needed to perform a task that is not dependent on the specialist, thus it does not depend on the car, either.

The starting and finishing of orders can have additional timing constraints. From the practical point of view, timing restrictions of each order belong to one of the following types:

'**time window**' Means that the order has to be performed within a time interval. The order cannot start earlier as defined but the finish time can be violated.

'**deadline**' Means that the order has to be finished before a given time point. This constraint can be violated.

'**exact start**' Means that the order has to be started at a given time point. This constraint can be violated and a 15-min delay is allowed without penalty.

From the modeling point of view, these restrictions overlap with each other, therefore the input parameters are assumed to be preprocessed and presented in the following coherent form, which covers all of the above cases:

- Each order $O \in \mathcal{O}$ has a $t^{startafter}(O)$ parameter, which denotes the earliest time when the execution of the first task of the order can be started. This constraint is strict, i.e., it must not be violated by the schedule. If no other value is provided for O, $t^{shiftstart}$ or 0 can serve as a default.
- Each order $O \in \mathcal{O}$ has a deadline, $t^{deadline}(O)$, by which all of the tasks of O must be finished. This constraint is soft, i.e., it can be violated, however, tardiness is punished by penalties. If no other value is provided, $t^{shiftend}$ serves as a default.
- There is a subset of orders $\mathcal{O}^{scheduled} \subseteq \mathcal{O}$, where for each $O \in \mathcal{O}^{scheduled}$, if the first task starts more than 15 min later than $t^{startafter}(O)$, penalties must be paid similarly to violating deadlines.

Next to timing, several constraints must be met for operating the cars, as well. A car $C \in \mathcal{C}$ may not travel a larger distance than its limit denoted by $d^{max}(C)$. Distance between two locations are given by the $d^{travel}(L_1, L_2)$ parameter. It is also assumed that the specialists assigned to the same car must stay together, i.e., part of the crew cannot leave for another location and leave a member performing a task behind. Moreover, specialists in the same car cannot perform separate tasks in parallel at the same location.

The goal is to assign a car to each task and plan the travel schedule of each car while minimizing costs. The overall cost consists of several components:

- If a car is used for at least one task during the shift, a fix cost of $c^{fix}(C)$ must be paid.
- For each car, transportation cost must be paid, which is proportional to the distance traveled by that car. The coefficient is denoted by $c^{travel}(C)$.

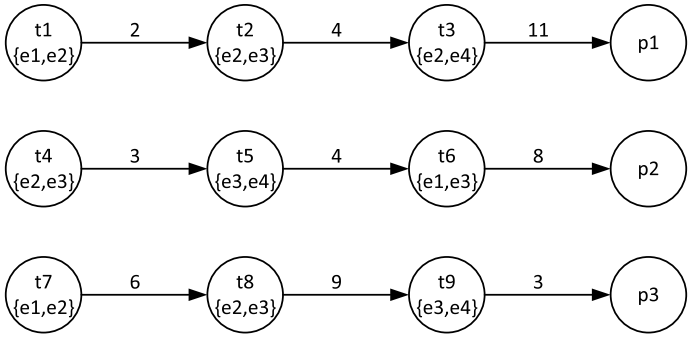

Fig. 1 Example: recipe-graph

- Performing task I costs $c^{perform}(I)$. Note that this cost does not depend on the assigned car, thus, it is just a constant term, not influenced by the scheduling decisions made.
- If a deadline or the 15-min starting window of a scheduled order is missed, a penalty is paid, which is proportional by the tardiness. The coefficient depends on the order and it is denoted by $c^{penalty}(O)$.

3 S-graph framework

The S-graph framework has been developed by Sanmartí et al. (2002) for production scheduling. The framework has two pillars: the directed graph-based mathematical model and the algorithms that operate on that model. In S-graphs nodes denote tasks and products and each arc defines a timing constraint between the connected nodes. There are two types of arcs in an S-graph: recipe-arcs, which express production related constraints given in the input and schedule-arcs, which express the sequencing decisions made by the optimization algorithms. Based on the arcs present in them, two types of S-graphs have special significance in the solution process: the recipe- and the schedule-graph.

A recipe-graph belongs to the recipe, which entails the network of tasks to be performed. Figure 1 represents the production sequence of three products where each production consists of three consecutive steps. Nodes $p1$, $p2$, and $p3$ denote the products (product-nodes) and nodes $t1$–$t9$ denote the tasks (task-nodes). The arcs between the task-nodes represent the production order and the arcs from task-nodes to product-nodes denote the completion of the corresponding products. Each arc is a recipe-arc and denotes a timing constraint between the starting times of the two connected tasks or between the starting time of the task and the completion time of the corresponding product. The weight of these arcs is the processing time of the tasks from which they start.

The sets below each task-node represent the set of units, resources that can perform the task, e.g., task $t1$ can be carried out either in equipment unit $e1$ or $e2$.

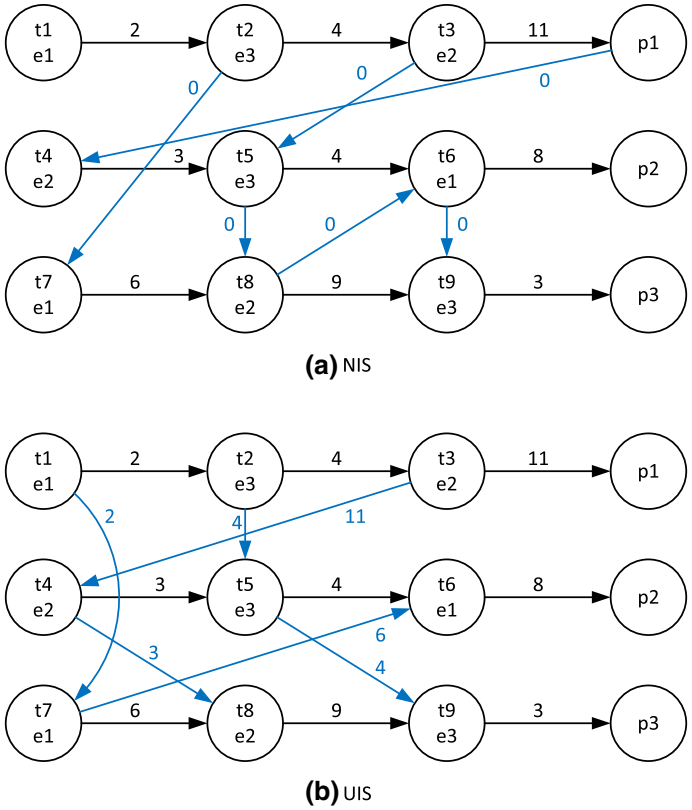

Fig. 2 Example: schedule-graphs

A schedule-graph belongs to a fully scheduled problem, i.e., it represents a solution to the scheduling problem. The two schedule-graphs in Fig. 2 show schedules of the recipe given in Fig. 1. There are two differences between a schedule-graph and a recipe-graph:

- The sets of units/resources have been replaced by the unit/resource chosen by the algorithm to perform that task.
- Some additional arcs expressing the activity order of resources are inserted into the graph.

All these additional arcs are schedule-arcs and the way they are inserted into the graph depends on the availability of storage for intermediate materials in the plant.

If there are no storage units available, the arcs must start from the consecutive nodes of the source node in the recipe, shown as Case (a) in Fig. 2. In this case, the weight of each arc is 0. As an example, the same sequencing of $e1$ is expressed by

the 0 weighted schedule-arc between tasks $t2$ and $t7$ because $e1$ performs task $t1$ first then it loads its material into $e3$ and starts to work on task $t7$.

However, if there is sufficient storage available, the schedule-arc is inserted between the two tasks that have been assigned to the same unit and the weight of the arc is the processing time, as shown as Case (b) in Fig. 2. For example, resource $e1$ performs task $t1$ first then it starts to work on task $t7$, thus a 2 weighted arc is inserted between task $t1$ and task $t7$.

In this paper, storage is assumed to be always available, thus the so-called UIS (Unlimited Intermediate Storage) schedule-arcs of Case (b) will be used.

As presented so far, the weight of the arcs in the S-graph framework were scalar numbers which express a lower bound on the timing of the connected nodes. Many industrial applications, however, require timing constraints that are of the 'less or equal' type. Most commonly, unstable intermediate materials can have a limit on their storage time. To address these types of restrictions, the S-graph model has been extended with interval weighted recipe-arcs (Hegyháti et al. 2011). In this extended S-graph model, if an arc has the weight of (t^l, t^u), then the difference between the timing of the connected nodes should be at least t^l and at most t^u. These type of arcs can be addressed by the S-graph algorithms in different ways, among which the most efficient is a combinatorial technique (Hegyháti 2015).

Interval-weighted arcs in the S-graph framework are called LW-arcs to refer to the Limited Wait storage policy of the intermediate materials where they are most often applied. If the lower and upper bound of an arc are of the same value, i.e., the weight of the arc is in the form of (t, t), the arc is often called a ZW-arc, again, referring to the Zero-Wait storage policy. ZW-arcs express a fixed timing difference between the connected nodes, which can often be used in various situations. In this work all LW-arcs are ZW-arcs. Last but not least, (t, ∞) weighted arcs represent the original lower bound type of arcs of the S-graph framework and thus they are often called UW-arcs (Unlimited Wait arcs).

In this paper, interval-weighted arcs are applied to model the problem at hand. To have a more transparent and easy to understand graphical representation, several graphical notations will be used throughout the rest of the paper as illustrated in Fig. 3. ZW-arcs will have a square head instead of arrow heads and instead of the interval (t, t) only t will be indicated. Similarly, on UW-arcs only t is indicated instead of having (t, ∞) along with an arrow head. A special case for both arcs is when $t = 0$, then no weight is indicated at all.

A very common problem parameter for batch process scheduling is the so-called changeover time, which is the time needed for a unit to change from one task to another. Usually this time is needed to perform some kind of operation on the unit to make it ready for the upcoming task, e.g., cleaning, changing settings. In the literature, changeover times can have different forms. In some cases, the time is unit specific, i.e., the changeover time depends on the assigned units as well, not only the tasks. Changeover time is called sequence dependent if it depends on the order of the two tasks as well.

In this paper, traveling times will be modeled as sequence dependent, non-unit specific changeover times denoted by $T^{chot}(n_1, n_2)$. This type of changeover time can easily be included in the mathematical model by increasing the weight of the

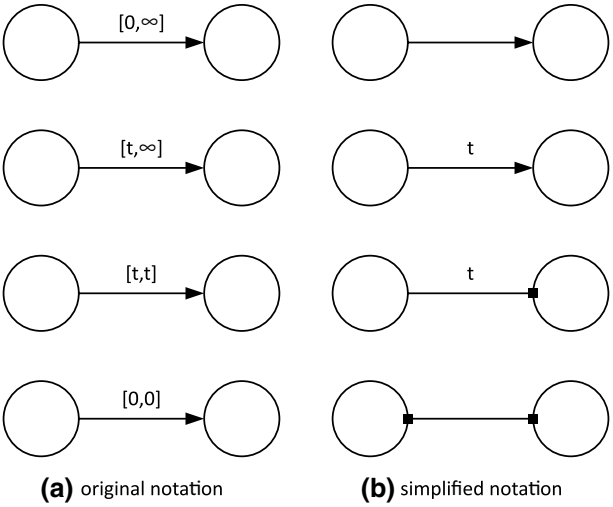

Fig. 3 Graphical notation for UW and ZW-arcs

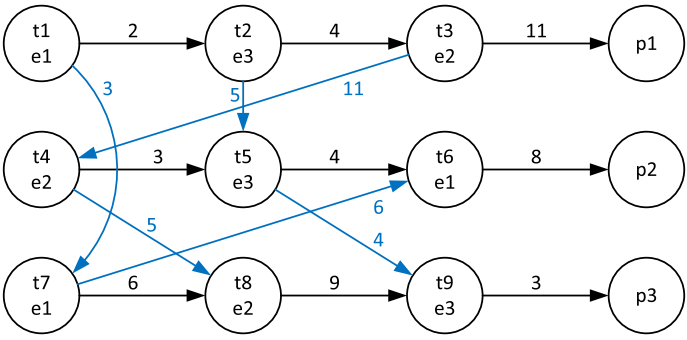

Fig. 4 UIS schedule-graph from Fig. 2 with changeover times added

$(n_1, n_2, T^{proc}(n_1))$ arc by $T^{chot}(n_1, n_2)$, therefore the new schedule-arcs will have the form: $(n_1, n_2, T^{proc}(n_1) + T^{chot}(n_1, n_2))$. An example is shown in Fig. 4 where the same UIS schedule is shown as in Fig. 2, however, the graph representation already includes the following changeover times: $T^{chot}(t1, t7) = 1$, $T^{chot}(t2, t5) = 1$, and $T^{chot}(t4, t8) = 2$.

For later sections, a brief introduction to the formal definitions of the S-graph is needed. A recipe-graph is denoted by $G(N, A_1, \emptyset)$ where N is the set of nodes and A_1 is the set of recipe-arcs. N is partitioned into two distinct subsets, N^T referring to task-nodes, which need to be scheduled, and product-nodes, N^P, which do not require any resource assignment. A schedule-graph of recipe-graph $G(N, A_1, \emptyset)$ is denoted by $G(N, A_1, A_2)$ where A_2 is the set of schedule-arcs. Both arc sets contain quadruplets in the form of (n_1, n_2, t^l, t^u), where $n_1, n_2 \in N$ and $t^l, t^u \in [0, \infty[$. R denotes the set of resources and N_r $(\subseteq N)$ denotes the nodes, which can be performed

by resource r ($\in R$). Through the solution process set N^U ($\subseteq N^T$) denotes the set of unscheduled task-nodes and M ($\subseteq N^T \times R$) denotes the set of assignments made in the form of (n, r) pairs, where $n \in N_r \setminus N^U$. Obviously, if $(n, r) \in M$, there exists no $r' \neq r$ such that $(n, r') \in M$ too, and $\{n \mid \exists r \in R, (n, r) \in M\} = N^T \setminus N^U$, i.e., M can be considered as a function.

The input of the S-graph optimization algorithm applies a Branch-and-Bound based search to find the optimal schedule-graph to the given scheduling problem. The specific version of the optimization algorithm applied in this paper is given formally in Appendix 3.

4 Modeling the problem with S-graph

The proposed approach relies on an extension of the S-graph framework to address all parameters of the problem at hand without the need to make any further assumptions atop those stated in Sect. 2. The core of the extension is a model transformation, i.e., providing a mapping between the input data and the S-graph model described in Sects. 2 and 3, respectively. This mapping is described in detail in Sect. 4.1 and summarized formally in Appendix 2. To address the cost-based objective and the travel distance limits, new bounding and feasibility functions had to be defined, which are described in Sects. 4.2 and 4.3. To solve the problem, there is no need to modify the standard S-graph algorithm for UIS-LW storage policy with changeover times. However, for easy understanding, the pseudo code for this algorithm is included in Appendix 3, as this specific version in its entirety has not been presented in the literature before.

4.1 Model transformation

The problem at hand is modeled as a precedential batch scheduling problem. Throughout the whole problem UIS storage policy is considered as the order locations can be left without any specialists. To model the various timing constraints of the problem, UIS-LW-arcs are needed. However, the terms ZW-arc and UW-arc are used when referring to (t, t) or (t, ∞) weighted UIS-LW-arcs.

The set of resources R will be the set of cars, i.e., $R = C$. As in various applications of the method, 3 special time reference nodes are introduced to the graph: $\{Z, S, E\} \in N$. These nodes will represent midnight and the start and the end of the shift. To fix the timing between these nodes, a ZW-arc is put between them: $(Z, S, t^{shiftstart}, t^{shiftstart}), (Z, E, t^{shiftend}, t^{shiftend}) \in A_1$ These three nodes (shown in Fig. 5) provide the frame for the other nodes and do not need to be scheduled or assigned to any resource similarly to the product-nodes of the framework.

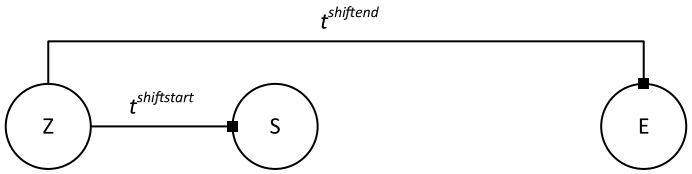

Fig. 5 Introducing nodes Z, S and E

4.1.1 Orders

Modeling the orders is rather straightforward. Each task of an order is represented by a node in the graph. Between those that are dependent on each other a recipe-arc is inserted with the weight of the processing time of the preceding task. Formally:

$$(O, I) \in N \qquad \forall O \in \mathcal{O}, I \in \mathcal{I}_{T_O}$$
$$((O, I_1), (O, I_2), t^{perform}(I_1), \infty) \in A_1 \qquad \forall O \in \mathcal{O}, (I_1, I_2) \in D_{T_O}$$

For the sake of transparency and keeping S-graph modeling conventions, an additional product-node is generated for each order, i.e., $N^P = \mathcal{O}$. As usual, a processing time weighted recipe-arc is inserted between all final tasks and the product node:

$$((O, I), O, t^{perform}(I), \infty) \in A_1 \qquad \forall O \in \mathcal{O}, I \in \mathcal{I}^{last}_{T_O}$$

where $\mathcal{I}^{last}_T$ is the set of tasks in template T that do not have a successor:

$$\mathcal{I}^{last}_T = \{I \in \mathcal{I}_T \mid \nexists I' \in \mathcal{I}_T, (I, I') \in D_T\}$$

The earliest starting time of an order can easily be expressed by an UW-arc between the special node Z and the first task or tasks of an order:

$$(Z, (O, I), t^{startafter}(O), \infty) \in A_1 \qquad \forall O \in \mathcal{O}, I \in \mathcal{I}^{first}_{T_O}$$

where $\mathcal{I}^{first}_T$ is the set of tasks in template T that do not have a predecessor,[1] i.e.:

$$\mathcal{I}^{first}_T = \{I \in \mathcal{I}_T \mid \nexists I' \in \mathcal{I}_T, (I', I) \in D_T\}$$

Moreover, no tasks of any order can be performed before the start or after the end of a shift. Thus, for each order, two additional 0 weighted UW-arcs are inserted between S and the starting tasks of an order; and the product-node of an order and E:

$$(S, (O, I), 0, \infty) \in A_1 \qquad \forall O \in \mathcal{O}, I \in \mathcal{I}^{first}_{T_O}$$
$$(O, E, 0, \infty) \in A_1 \qquad \forall O \in \mathcal{O}$$

[1] Note that instead of using the set $\mathcal{I}^{first}_T$, these arcs could be freely inserted to all of the task nodes of an order. The additional arcs would be redundant, and wouldn't express additional constraints, nor would they improve bounds or have any other effect on the behavior of the approach.

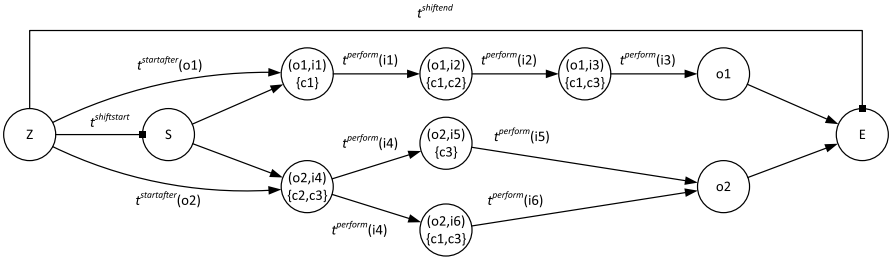

Fig. 6 Representing orders

The part of the recipe-graph belonging to the orders is illustrated in Fig. 6.

4.1.2 Travel times

The time needed for a car to travel between tasks must be included in the model, as well. Since this time depends on the locations of the tasks, it can be mapped to changeover times of batch scheduling problems that depend on the equipments and the two tasks. As a result, if a car will be assigned to a node (O_2, I_2) after (O_1, I_1), the UIS schedule-arc that the algorithm will insert into the graph will be this:

$$((O_1, I_1), (O_2, I_2), t^{perform}(I_1) + t^{travel}(L_{O_1}, L_{O_2}), \infty)$$

4.1.3 Depot

Each car has to start from the depot and return there by the end of the shift. To model this, the recipe-graph is extended by two task-nodes for each car, representing the loading and unloading of the cars at the depot. The operating times for these tasks are 0, the location for both of them is the depot, and the only suitable equipment is the related car. Formally:

$$L_C, U_C \in N^T \qquad \forall C \in \mathcal{C}$$

While locations belong to orders, the loading and unloading tasks must have the depot as their location. For the sake of generality, the notation $L(n)$ is introduced:

$$L(n) = \begin{cases} L^D & \text{if } n = L_C \text{ or } n = U_C \\ L_O & \text{if } n = (O, I) \end{cases} \qquad \forall n \in N^T$$

Similarly, $t^{perform}(L_C) = t^{perform}(U_C) = 0$ and $L_C, U_C \in N_C$ but $L_C, U_C \notin N_{C'}$ if $C' \neq C$.

To ensure that L_C is the first task assigned to C and U_C is the last one, some additional changes are needed. This rule could easily be enforced by slightly changing the core S-graph algorithms to adhere to it. However, that is not necessary as this constraint can be expressed by some additional recipe-arcs, as well.

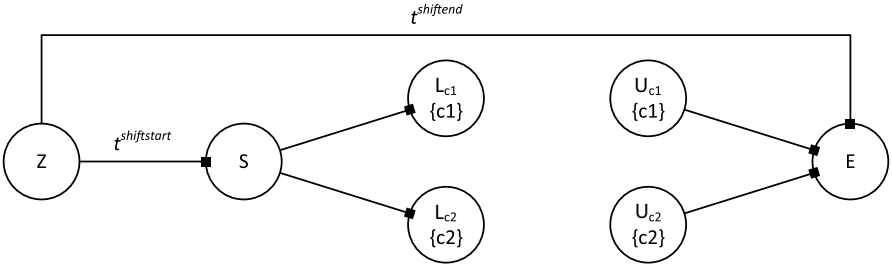

Fig. 7 Representing depot

First, a 0 weighted ZW-arc needs to be inserted between S and all of the L_C nodes:

$$(S, L_C, 0, 0) \in A_1 \qquad \forall C \in \mathcal{C}$$

These arcs ensure that L_C must be the first to be assigned to C, otherwise, if another node (O, I) is assigned to C first, the scheduling algorithm immediately inserts a weighted arc from (O, I) to L_C. Together with the arc inserted between S and (O, I), these arcs would form an infeasible schedule, and the subproblem would be pruned immediately.

Similarly, a 0 weighted ZW-arc is inserted between U_C and E:

$$(U_C, E, 0, 0) \in A_1 \qquad \forall C \in \mathcal{C}$$

These additional nodes and arcs are illustrated in Fig. 7.

4.2 Bounding function

The general B&B algorithms of the S-graph framework were originally developed for makespan minimization. However, for any other objective, if a proper bounding function is provided, the scheduling algorithms will provide an optimal solution.

For the investigated problem class, the objective is to minimize the total cost, which consists of the following elements:

- Processing costs of tasks
- Fix costs for cars if they are used
- Travel costs of cars
- Penalties for lateness for scheduled orders
- Penalties for deadline violations

In Sect. 4.2.1, a basic lower bound function is presented which incorporates the costs associated with the decisions already made in a partial schedule. This simple bound can be tightened if a lower bound is given for the costs corresponding to decisions that are not yet made. Section 4.2.2 provides several

examples for such procedures. In Sect. 4.2.3 the different options for the bounding function are summarized.

4.2.1 Basic bounding function

The basic bounding procedure summarizes all costs based on decisions already made in previous branching steps. The function starts with initializing the *bound* variable to zero. Each following block increases its value by the lower bound of a cost-component so that it will contain the lower bound on the overall cost in the end.

First, the sum of the costs of processing the tasks is added, which will be constant for all schedules. This value does not depend on any decision because all orders have to be performed.

```
1: procedure BasicBound
2:     bound := 0
3:     for all (O, I) ∈ N do
4:         bound := bound + c^perform(I)
5:     end for
```

Then the fix costs of those cars are added that already have at least one task assigned to them. These cars will be used in each solution reachable from the current subproblem so their fix costs will increase the total cost.

```
6:     for all r ∈ R do
7:         if ∃n ∈ N^T : (n, r) ∈ M then
8:             bound := bound + c^fix(r)
9:         end if
10:    end for
```

A lower bound for the travel costs of cars can be easily established by summing the costs for travels that have already been assigned to a car.

```
11:    for all (n1, n2, t^l, t^u) ∈ A2, (n1, r), (n2, r) ∈ M do
12:        bound := bound + c^travel(r) · d^travel(L(n1), L(n2))
13:    end for
```

The longest path from Z provides a lower bound on the starting time of each node. Thus, a lower bound on the penalties for starting orders too late can be given by summing the costs caused by the lateness that the partial schedule already ensures.

```
    for all O ∈ O^scheduled do
        start := ∞
        for all I ∈ I^first_{T_O} do
            if LongestPath(Z, (O, I)) < start then
                start := LongestPath(Z, (O, I))
            end if
        end for
        bound := bound + c^penalty(O) · max(0, start − (t^startafter(O) + 15))
    end for
```

Similarly, a lower bound on the lateness for deadlines of the orders can be established and used to further increase the bound with the corresponding costs.

```
    for all O ∈ O do
        finish := LongestPath(Z, O)
        if finish > t^deadline(O) then
            bound := bound + (finish − t^deadline(O)) · c^penalty(O)
        end if
    end for
    return bound
end procedure
```

After these steps, the returned *bound* variable contains the sum of the lower bounds for all of the cost-components, thus, a lower bound on the total cost itself.

4.2.2 Tightening the bound

The bound described in Sect. 4.2.1 only considers direct implications of decisions that have already been made in the partial schedule. This bound can be further tightened by including some additional costs that arise from decisions not yet made, but are certain to be included later in the B&B tree.

Each of the following procedures provide a lower bound on such costs and can be called to further increase the value of the *bound* variable from Sect. 4.2.1.

Fix costs of cars One way to achieve a tighter bound is to include the fix costs of cars that have not yet been assigned to any task but their usage is inevitable. This is the case, for example, if there is at least one unscheduled task that can only be performed by a single, yet unused car.

```
1: procedure AdditionalFixCosts
2:     plusfixcost := 0
3:     for all r ∈ R do
4:         if ∄n ∈ N^T : (n, r) ∈ M then
5:             if ∃n ∈ N^U : R_n \ R' = {r} then
6:                 plusfixcost := plusfixcost + c^fix(r)
7:             end if
8:         end if
9:     end for
10:    return plusfixcost
11: end procedure
```

Cost of remaining trips The bound can be further tightened by computing a lower bound for the cost of the remaining trips that the cars have to make. Two such different lower bounds are given here.

The first method relies on the minimal costs of entering and leaving a location. These values ($c^{enter}(l)$, $c^{leave}(l)$) calculated at the start of the solution will remain constant:

```
1: for all l ∈ L do
2:     c^enter(l) := ∞
3:     c^leave(l) := ∞
4: end for
5: for all r ∈ R do
6:     for all l ∈ L do
7:         for all l' ∈ {l'' ∈ L \ {l} | ∃(O, I) ∈ N^T : L_O = l'' ∧ r ∈ C_I} do
8:             c^enter(l) := min(c^enter(l), c^travel(r) · d^travel(l', l))
9:             c^leave(l) := min(c^leave(l), c^travel(r) · d^travel(l, l'))
10:        end for
11:    end for
12: end for
```

For every location with unscheduled tasks, the cost of leaving this location can be added. If a car has a task with this location assigned to it most recently, it may be able to perform the remaining tasks there. Otherwise, a car will have to travel there so the cost of entering can be added. The sums of entering and leaving costs are kept separately, as every trip both leaves a location and enters another. Only one of the sums can be included in the bound, preferably the higher.

$M^L \subseteq M$ is the set of assignments that were last assigned to the cars. It contains up to one task-resource pair for each car.

```
1: procedure AdditionalVisitCosts(L' ⊆ L)
2:     costs^in := 0
3:     costs^out := 0
4:     for all l ∈ L' \ {L^D} do
5:         if ∃n ∈ N^U : L(n) = l then
6:             costs^out := costs^out + c^leave(l)
7:             if ∄(n', r) ∈ M^L : L(n') = l then
8:                 costs^in := costs^in + c^enter(l)
9:             end if
10:        end if
11:    end for
```

We can also add the costs of returning to the depot or leaving the last location for each car. Note, that triangle inequality is assumed to hold for d^{travel}.

```
12:    for all r ∈ R do
13:        if ∃(n, r) ∈ M^L : L(n) ≠ L^D then
14:            costs^in := costs^in + c^travel(r) · d^travel(L(n), L^D)
15:            costs^out := costs^out + c^leave(L(n))
16:        end if
17:    end for
18:    return max(costs^in, costs^out)
19: end procedure
```

The second method considers the remaining trips for each car in more detail. The locations that the car must visit are determined by the tasks that can only be performed by the car in question. The lowest possible cost of visiting these locations is the solution of a traveling salesman problem. However, solving an NP-hard problem at each bounding step would be inefficient. The minimum spanning tree of the location-distance graph gives a computationally easy lower bound on the future trips. If distances are asymmetric, the smaller distance can be used for each location pair. Finally, for the locations whose every task can be performed by multiple cars ($\mathcal{L}^x$), the first method can be used to get a lower bound for further costs.

```
1: procedure CarBasedCosts
2:     L^x := {l ∈ L | ∀n ∈ N^U : L(n) = l}
3:     for all r ∈ R do
4:         L_r := {l ∈ L | ∀n ∈ N^U : C_n = {r} ∧ L(n) = l}
5:         if ∃(n, r) ∈ M^L then
6:             L_r := L_r ∪ {L(n)}
7:         end if
8:         bound^t := bound^t + c^travel(r)·MST(L_r, d^travel)
9:         L^x := L^x \ L_r
10:    end for
11:    bound^t := bound^t + AdditionalVisitCosts(L^x)
12: end procedure
```

4.2.3 Variants for the bounding function

Theoretically, there are 12 different options how the above procedures can be combined to have a proper bounding function:

- BasicBound is either used or not
- AdditionalFixCost is either used or not
- For future travel costs, either AdditionalVisitCosts or CarBasedCosts is used, or neither of them.

Not using the BasicBound is an unreasonable decision as it is easy to evaluate and contributes a lot to the bound. From the 6 remaining options the following 3 are selected:

Bound-0 The simplest option is to use only the BasicBound function. This is the fastest approach but it also provides the poorest bound. This option is mostly used as a baseline reference.

Bound-1 Another option is to use the sum of the return values of BasicBound, AdditionalFixCosts and AdditionalVisitCosts. This option requires slightly more computation but provides a tighter bound.

Bound-2 The third option is to use the sum of the return values of BasicBound, AdditionalFixCosts and CarBasedCosts. The last procedure requires to solve a minimal spanning tree problem, thus it is expected to be the slowest, but it can also provide even tighter bounds.

4.3 Feasibility function

The B&B algorithms of the S-graph framework perform a feasibility check at each partial schedule. The basic feasibility function of the framework searches for a directed cycle within the graph, which indicates infeasible schedules.

This function is adequate for most of the problems solved by the S-graph framework so far. However, for the problem at hand, there is a limit on the maximal distance a car can travel a day, $d^{max}(C)$.

The distance that a car must cover in the current partial problem can be calculated very similarly to its corresponding cost contribution and if it exceeds the given limit for any car, the function must return false:

1: **for all** $r \in R$ **do**
2: $dist_r := 0$
3: **for all** $(n_1, n_2, t^l, t^u) \in A_2, (n_1, r), (n_2, r) \in M$ **do**
4: $dist_r := dist_r + d^{travel}(L(n_1), L(n_2))$
5: **end for**
6: **end for**
7: **if** $\exists r \in R : dist_r > d^{max}(r)$ **then**
8: **return** false;
9: **end if**
10: **return hasDirectedCycle**$(G(N, A_1, A_2))$

This feasibility function considers only the trips already assigned to a car. The second bound tightening method calculates a lower bound on the future trips of each car. It is also used to detect a future infeasibility when the lower bound exceeds $d^{max}(r)$.

4.4 Branching strategies

Two main S-graph branching strategies have been published to make the assignment and sequencing decisions expressed by schedule-arcs. The original algorithm (Sanmartí et al. 2002) selects an equipment unit in each branching step, and different tasks are added to the end of its production sequence in each children node. More recently, a different algorithm has been proposed (Adonyi et al. 2007), where an unassigned, unscheduled task is selected, and in each child node, the task is added to a different position in the production sequence of any suitable units. To highlight this difference in the branching step, the two algorithms had been labeled as equipment-based and task-based, respectively. Both branching procedures can have many variants based on the selection method of the pivoting unit or task and the order in which the children are generated.

Table 1 Distance of locations in du

	Depot	l1	l2	l3
Depot	0.0	0.5	1.0	0.5
l1	0.5	0.0	0.5	1.0
l2	1.0	0.5	0.0	0.5
l3	0.5	1.0	0.5	0.0

Table 2 Data of cars

Car	d^{max} [du]	c^{travel} [cu/du]	c^{fix} [cu]
c1	10	150	100
c2	15	120	120

Table 3 Process templates

Template	Task	Time [h]	Cost [cu]	Next tasks	Suitable cars
t1	i1	0.25	3000	–	c1
t2	i2	0.35	5000	i3	c1
	i3	0.5	10,000	–	c1
t3	i4	0.25	1000	i5, i6	c1, c2
	i5	0.5	2000	i7	c1, c2
	i6	0.5	3000	i7	c1, c2
	i7	0.5	4000	–	c2

Table 4 Data of orders

Order O	L_O	T_O	Constraint	Delay cost [cu/h]
o1	l1	t1	10:00–12:00	0
o2	l2	t2	Till 14:00	1
o3	l2	t2	–	0
o4	l2	t2	At 14:00	100
o5	l3	t3	9:00–15:00	100
o6	l3	t3	Till 15:00	200

The proposed algorithm of Appendix 3 applies the equipment-based strategy for several reasons:

- Unless there is a very defining bottleneck, it outperforms the task-based strategy in most of the general cases (Adonyi 2008).
- More acceleration techniques have been published to further enhance its performance (Holczinger et al. 2002; Adonyi et al. 2008).
- Some components of the bounding functions detailed in Sect. 4.2 rely on this strategy.

For the computational tests, unit selection was based on alphabetical order and the B&B tree was explored in a depth-first fashion.

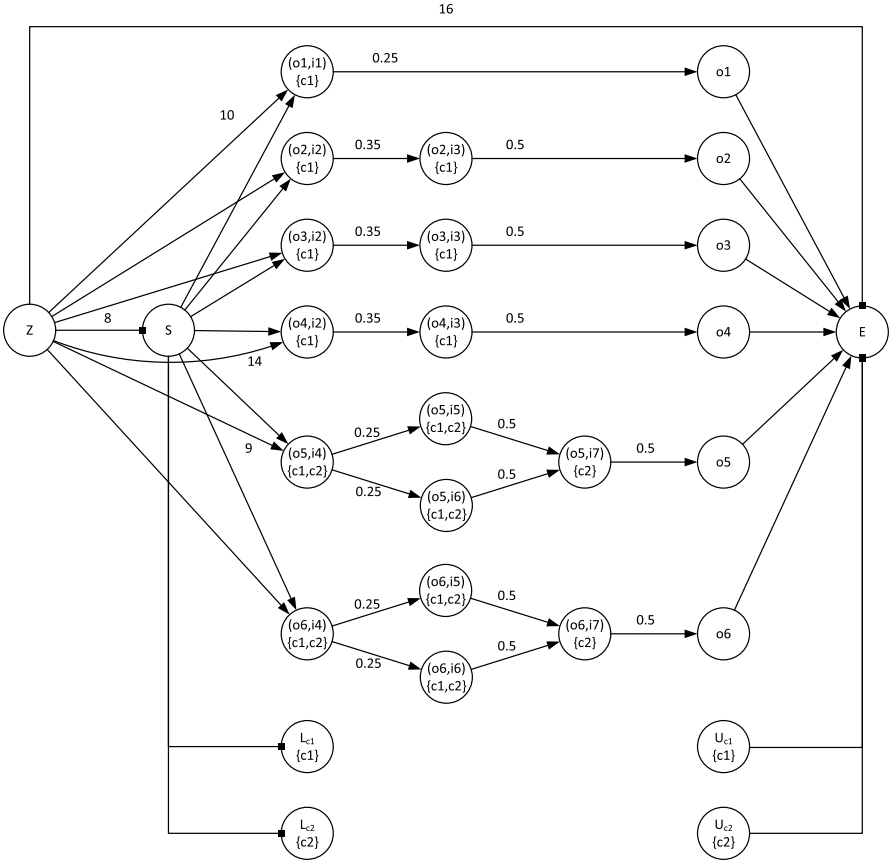

Fig. 8 Recipe-graph of the example

5 Computational tests

The performance of the proposed approach was tested on several problem instances. The example in Sect. 5.1 illustrates the solution process of a problem, showing the input parameters, the resulting recipe-graph and the schedule of the obtained optimal solution. Then Sect. 5.2 presents more results for problem instances that were random-generated with different parameter settings.

The tests were run on an Intel Core i5-660 3.33 GHz CPU with 4 GB RAM with 3600 s time limit.

5.1 Illustrative example

The example presented below shows how a problem can be represented by an S-graph. It contains orders with all types of time constraints.

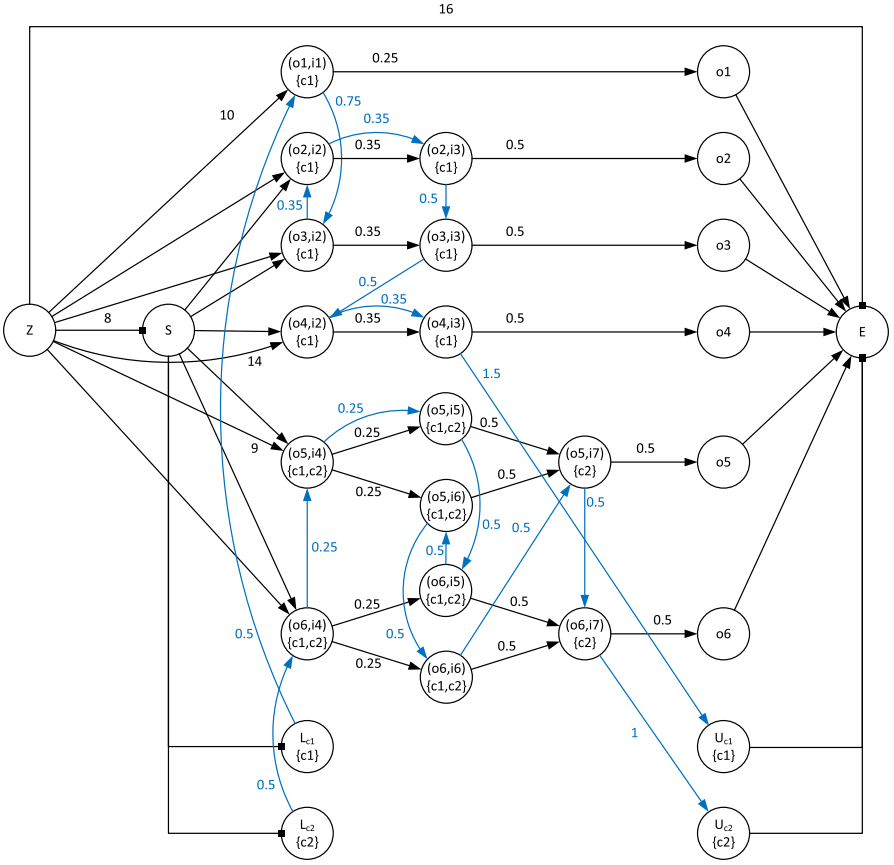

Fig. 9 Schedule-graph of the example

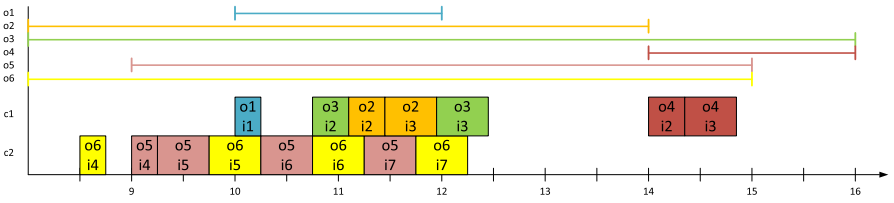

Fig. 10 The schedule of the example

Suppose that a company has 6 orders in 3 different locations. The distances of the locations are shown in Table 1. The travel times between locations are proportional to the distance where the rate is 1, i.e., if distance of two locations is 2 units, then the travel time is 2 h. There are two cars which can perform the orders, their parameters are given in Table 2.

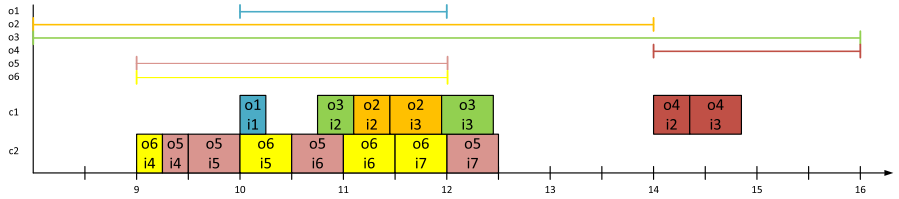

Fig. 11 The schedule of the modified example

Table 5 Parameters for problem generation

Parameter	A-n	B-m	C-w-x-y	D-z
`params.taskNumber` $\equiv \lvert N^T \rvert$	n	15	15	15
`params.orderNumber` $\equiv \lvert \mathcal{T} \rvert = \lvert \mathcal{O} \rvert = \lvert \mathcal{L} \rvert$	5	m	5	5
`params.carNumber` $\equiv \lvert \mathcal{C} \rvert$	2	2	2	z
`params.ordersWithTimeWindow`	3	3	w	3
`params.windowBuffer`	1.4	1.4	0.5	1.4
`params.ordersWithExactStart` $\equiv \lvert \mathcal{O}^{scheduled} \rvert$	3	3	x	3
`params.orderDeadlineMin`	10	10	40–30y	40

There are 3 process templates which define the tasks of the orders, the processing times, the costs and the plausible cars. The process templates are defined in Table 3. Template t1 consists of only one task (i1) and template t2 has two consecutive tasks (i2 followed by i3). Template t3 contains four tasks where i4 precedes i5 and i6 moreover i7 follows i5 and i6, i.e., i5 and i6 can be parallel tasks.

Order o1 is based on template t1 and it has a time window constraint. Order o2, o3 and o4 are based on process template t2 with the following time constraints: deadline, no constraint and exact time, respectively. There are two orders (o5 and o6) that are based on template t3, where o5 has time window and o6 has deadline constraint. The data of the orders is shown in Table 4.

In the example, the working hours start at 8:00 and finish at 16:00, i.e., there are 8 h to perform all orders.

The recipe-graph of the example is given in Fig. 8. The S and the Z nodes are connected to the first task-nodes of the orders and all product-nodes are connected to node E. The product-nodes contain only the name of the order, moreover, the task-nodes contain the name of the task-nodes and the set of plausible resources. The task-nodes and product-nodes are connected as the process templates define their precedence and the weights of the recipe-arcs are equal to the processing times. Z node is connected with ZW-arc to S and E nodes. Moreover, zero weighted ZW-arcs are established from S to L_c nodes and from U_c nodes to E.

The example problem was solved to optimality with the proposed method. The resulting schedule-graph is given in Fig. 9 and the Gantt chart of the solution is given in Fig. 10. The Gantt chart has been extended by the time constraints of the orders at the top. For example, the blue bar shows the allowed time interval of order o1. As the Gantt

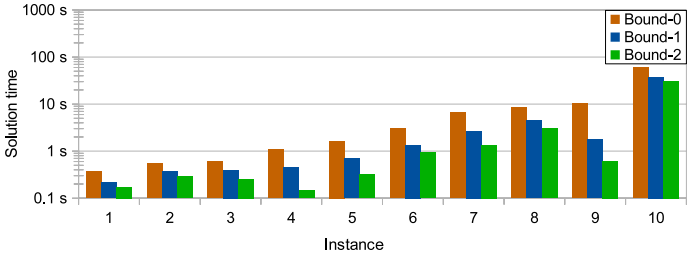

Fig. 12 Results for dataset A-12

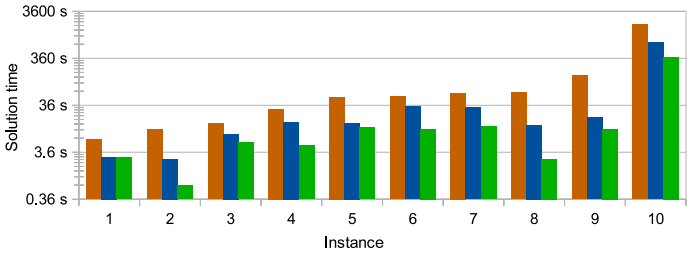

Fig. 13 Results for dataset A-14

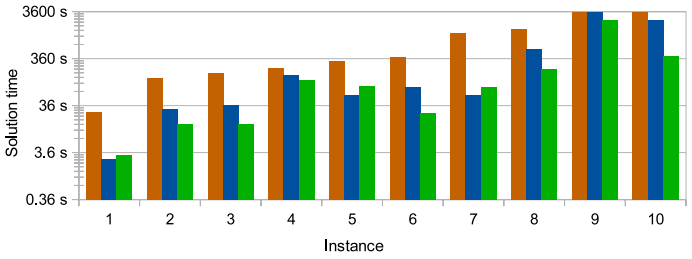

Fig. 14 Results for dataset A-16

chart shows, all orders can be performed in the given time intervals without delays so the cost only depends on the task performing and travel costs. The optimal cost is 68,640.

The example was solved with the different bounding methods defined in Sect. 4.2.3. The solution times using **Bound-0**, **Bound-1** and **Bound-2** are 181.52 s, 1.10 s and 0.02 s, respectively.

The example was changed to get a solution where time constraints cannot be held. Orders o5 and o6 got time window constraints from 9:00 to 12:00. The Gantt chart of the solution is shown in Fig. 11. It can be seen that in the optimal solution, order o5 cannot be finished before 12:00 so it imposes some penalty costs increasing the value of the objective function. The optimal cost of the modified example is 68,690. The CPU times for the 3 bounding methods increased to 66.04, 7.84 and 6.21 s, respectively.

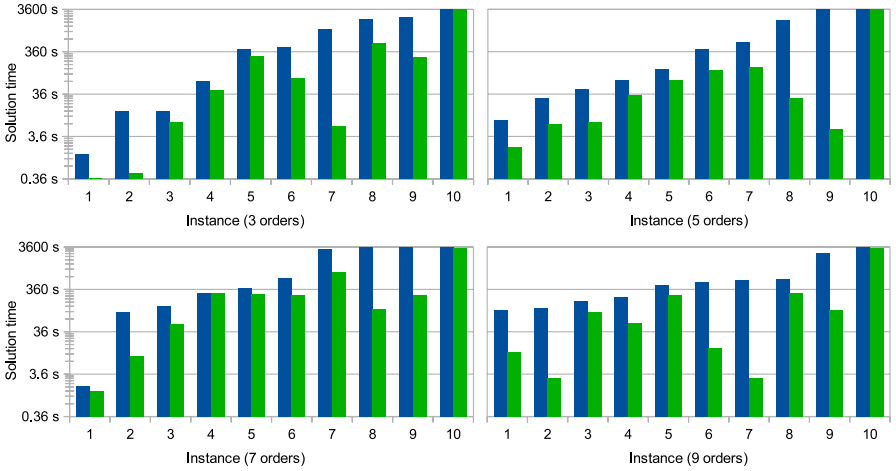

Fig. 15 Results for datasets B-3, B-5, B-7, B-9

5.2 Parameter analysis

To earn more insight about how problem characteristics affect the solution performance, some more tests were completed with different input parameters. The problem instances were random-generated with a Python script. A shortened version of the script is supplied in Appendix 4. The input parameters for problem generation are shown in Table 5. The table only displays the parameters which are changed through parameter sets A, B, C, D, the unchanged parameters are shown as constants in the script, for example $t^{shiftend} = 40$.

10 instances were generated for each parameter setting. In the charts they are sorted by the solution time of the slowest method for more clarity. Because of the high variance among solution times, the values are displayed with a logarithmic scale.

As in most scheduling problems, the number of tasks to be scheduled, is a good measure of the problem size and it relates to the difficulty of solving a problem. Hence, the first set of test instances are scaled on the task number.

In A-12, 12 tasks are in each instance. As Fig. 12 shows, this size can be solved in less than 100 s even without using more sophisticated bounding methods. With 14 tasks solution time increased, and in one case the basic method needed 1878 s for finding the optimal solution (Fig. 13). The instances of A-16 are complex enough that even **Bound-1** hit the 1-h time limit for 1 instance (Fig. 14). The solution times well illustrate the performance improvements of tightening the bound. Even **Bound-1** is significantly faster than the basic method. The more complex **Bound-2** performs even better on most instances suggesting that the benefits of a tighter bound justify the usage of a more computational-heavy subroutine.

Based on the results of A-*n*, the task number was set to 15 for other datasets. **Bound-0** performed the worst in all tests and could not find the optimal solution

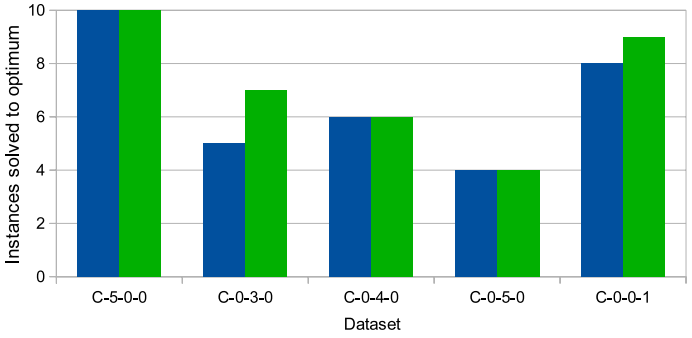

Fig. 16 Results for datasets C-*w*-*x*-*y*

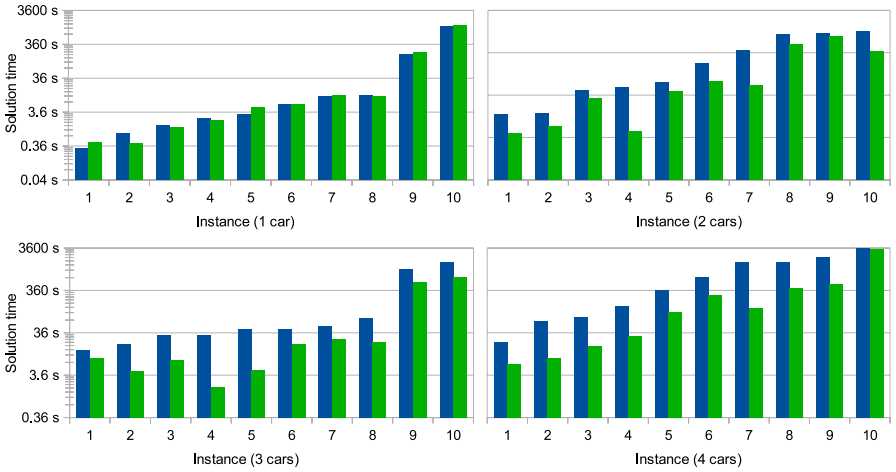

Fig. 17 Results for datasets D-*z*

within the 1-h time limit. Because of this, its results are omitted from the following charts.

In this scheduling problem the tasks are grouped into orders and the timing constraints and location are given for orders. The datasets B-*m* were used to examine how the number of orders contribute to the complexity. Figure 15 shows the solution times for 3, 5, 7, and 9 orders. The average solution times are the highest for B-7. Solution times show a tendency to increase as order number increases, especially for **Bound-1** method (shown in blue) but with 15 tasks, most orders in B-9 are single-stage, which can simplify some scheduling decisions.

Based on these results, the number of orders has a low contribution to problem complexity on its own. For further tests, 5 orders are generated, the same as in A-*n*.

The hypothesis behind datasets C-*w*-*x*-*y* was that the types of timing constraints contribute more to the complexity rather than the number of orders. As they were

introduced in Sect. 2, there are 3 types of constraints: time window, exact start and deadline. Parameters w and x represent the number of orders with time window and exact start constraints. y is a binary parameter, denoting whether the orders have deadlines or not. In previous tests, their values were 3–3–1 to get orders with mixed constraints.

The datasets of C-w-x-y were used to examine solution performance with only 1 type of order constraints. Instead of solution times, which showed a similar difference between the bounding methods as the previous tests, Fig. 16 shows how many instances were solved to optimality under the time limit.

It can be seen that the exact start constraints had the highest contribution to complexity. Note that these constraints do not fix the starting times of orders, they can be started later but with penalty. As the proposed method performed worse on these test cases than others, it poses a future research objective to improve the approach for problems with exact start constraints. For example, a heuristic search could provide good approximate solutions to prune the search tree.

Problems with only time window constraints were the easiest to solve, regardless of the length of the window. Orders with time windows can only start after $t^{startafter}$ but there is no penalty in starting later unlike with exact start constraints. This reduces the search space and hence, complexity, as well.

The number of cars is an important parameter of the problem too, as they are the scarce resources to be scheduled to execute the tasks and they must be routed among the order locations. Datasets D-z were used to test the methods with 1, 2, 3 and 4 cars. Figure 17 presents the solution times. The case with only 1 car was the easiest to solve on average and the difference of the methods is small on these instances. Having more than 2 cars did not result in much worse solution performance but here the advantage of the more complex bounding method is significant.

Based on the input parameter analysis, **Bound-2** has the best performance on most problems. Also, the exact start constraint causes the most difficulties for the approach, therefore, future research is needed to improve performance on this particular type of timing constraint.

6 Comments

The proposed approach was specifically developed for the problem class given in Sect. 2, however, requests for solving the instances of a more general problem class can be expected in the future. Thus, in this section, some perspectives are given on what can be included in the proposed model with no difficulties, which requires major modifications, and how these may influence the efficiency of the approach.

Car specific shifts The specialists, and thus the cars may work in different work shifts. This can easily be addressed by having a direct ZW-arc between Z and the L_C nodes. If there is no general earliest starting time for the orders themselves, the S node is removed and direct arcs can be inserted from Z to the starting nodes of the jobs. Some additional attention is needed to prevent the algorithm from not scheduling the L_C nodes as first in the sequence of the car, however, after these minor

tweaks, the approach can address car specific shifts without any computational disadvantage.

Car specific travel times/distances The cars in the problem may not be uniform, some can travel faster, some must avoid certain roads for various limitations, etc. Modeling such circumstances does not require any further extensions, as the S-graph framework is already equipped with unit-specific changeover times. Just like in the previous case, this generalization only needs additional input data and has negligible effect on the CPU requirements of the approach.

Strict deadlines At the moment, the only strict deadline for a job is the end of a shift. However, a separate deadline could similarly be added to the graph without any changes in the algorithm.

Parallel tasks at the same location The proposed approach cannot address situations when a job has tasks that could be performed in parallel by distinct subsets of the same team. Since parallel execution of tasks by the same resource is prohibited in the very core of the S-graph framework, addressing this feature would require a more complex extension of the framework.

7 Summary

In this paper a scheduling problem has been presented where mobile groups of workers have to perform orders in different locations. The orders have various time constraints, some of which can be violated and thus incurring penalties. The S-graph framework has been extended to handle the special aspects of the problem by introducing a sound construction rule for the S-graph model and problem class-specific bounding functions. The applicability of the approach was illustrated in detail, and additional tests have been carried out on a large number of random-generated examples to examine solution performance of variants of the proposed algorithm on problems with different characteristics.

In the future, the approach can be extended to handle cases where there are not enough resources to perform all orders. Performance can be improved on problems with exact start constraints. The representation and the corresponding algorithm can be modified to take into account multiple days or shifts, i.e., with modifications it can also define the daily work assignments. Other potential extension is to use flexible groups, i.e., the group of workers can be modified before the working hours, therefore, the available cars for tasks can be changed.

The motivating problem of the research, which was solved by the proposed method, came from a service provider company. However, the attributes of the problem, for example the sequence-dependent changeover times or the time window constraints also exist in other scheduling problems. With some minor modifications of the method a large class of scheduling problems could be solved, which is a future aim of this research.

Acknowledgements Open access funding provided by University of Pannonia (PE). We acknowledge the financial support of Széchenyi 2020 under the EFOP-3.6.1-16-2016-00015.

Appendix 1: Nomenclature

Input problem

Sets and relations

$\mathcal{O}$	Set of orders
$\mathcal{O}^{scheduled}$	Set of orders, that have their start scheduled
$T_O \in \mathcal{T}$	The template of order $O \in \mathcal{O}$
$L_O \in \mathcal{L}$	The location of order $O \in \mathcal{O}$
$\mathcal{T}$	Set of templates
$\mathcal{I}_T$	Set of tasks for a given template $T \in \mathcal{T}$
$\mathcal{I}_T^{first}$	Set of tasks of template $T \in \mathcal{T}$, that does not have a predecessor
$\mathcal{I}_T^{last}$	Set of tasks of template $T \in \mathcal{T}$, that does not have a successor
$D_T \in \mathcal{I}_T \times \mathcal{I}_T$	Mandatory precedences among the tasks of template $T \in \mathcal{T}$
$\mathcal{C}_I \subseteq \mathcal{C}$	Set of cars able to perform task I (where $I \in \mathcal{I}_T$ for some $T \in \mathcal{T}$)
$\mathcal{L}$	Set of locations
$L^D \in \mathcal{L}$	The special location of the depot
$\mathcal{C}$	Set of cars

Parameters[2]

$c^{penalty}(O)$	`[cu/tu]` is the penalty for violating the deadline or the exact starting time of scheduled order O.
$c^{perform}(I)$	`[cu]` is the cost needed to perform task I.
$c^{travel}(C)$	`[cu/du]` is the cost of traveling with car C proportional to the distance.
$c^{fix}(C)$	`[cu]` is the fix cost of using car C in a single shift.
$d^{max}(C)$	`[du]` is the maximal distance of car C allowed to travel in a single shift.
$d^{travel}(L_1, L_2)$	`[du]` is the distance of traveling from location L_1 to location L_2.
$t^{deadline}(O)$	`[tu]` is the deadline of order O, meaning that all corresponding tasks must end by this time.

[2] ***du***, ***tu***, ***cu*** refer to distance unit, time unit, and cost unit, respectively.

$t^{perform}(I)$	[tu] is the time needed to perform task I. Note, that this value is not dependent on the car, which is assigned to the task.
$t^{shiftstart}$	[tu] is the time at which the shift starts.
$t^{shiftend}$	[tu] is the time at which the shift ends.
$t^{startafter}(O)$	[tu] is the earliest time when working on a task of order O can start.
$t^{travel}(L_1, L_2)$	[tu] is the time needed to travel from location L_1 to location L_2.

S-graph related notations

R	Set of resources
$R_n \subseteq R$	Set of resources that can perform the (task) node n
$R' \subseteq R$	Set of resources whose job list is not closed in a partial schedule
N	Set of nodes
N^T	$\subseteq N$ Set of (task) nodes to be scheduled
$N^P \subseteq N$	Set of (product) nodes that do not require scheduling
$N_r \subseteq N^T$	Set of (task) nodes that can be performed by resource $r \in R$
$N^U \subseteq N^T$	Set of unscheduled task-nodes for a partial schedule
$A_1 \subseteq N \times N \times \mathbb{R}_0^+ \times \mathbb{R}_0^+$	Set of recipe-arcs
$A_2 \subseteq N \times N \times \mathbb{R}_0^+ \times \mathbb{R}_0^+$	Set of schedule-arcs
$M \subseteq N^T \times R$	Set of assignments made in a partial schedule
$M^L \subseteq M$	Set of last assignments in a partial schedule containing only the latest task assigned to each resource
$T^{proc}(n) \subseteq N^T \rightarrow \mathbb{R}_0^+$	The processing times
$T^{chot}(n_1, n_2) \subseteq N^T \times N^T \rightarrow \mathbb{R}_0^+$	The changeover times
$L(n) \subseteq N^T \rightarrow \mathcal{L}$	The location of a task-node

Appendix 2: Model transformation

$$
\begin{aligned}
N^P &= \{Z, S, E\} \\
&\cup \{O \mid O \in \mathcal{O}\} \\
N^T &= \{L_C, U_C \mid C \in \mathcal{C}\} \\
&\cup \{(O, I) \mid O \in \mathcal{O}, I \in \mathcal{I}_{T_O}\} \\
A_1 &= \{(Z, S, t^{shiftstart}, t^{shiftstart}), (Z, E, t^{shiftend}, t^{shiftend})\} \\
&\cup \{(S, L_C, 0, 0), (U_C, E, 0, 0) \mid C \in \mathcal{C}\} \\
&\cup \{((O, I_1), (O, I_2), t^{perform}(I_1), \infty) \mid O \in \mathcal{O}, (I_1, I_2) \in D_{T_O}\} \\
&\cup \{((O, I), O, t^{perform}(I), \infty) \mid O \in \mathcal{O}, I \in \mathcal{I}^{last}_{T_O}\} \\
&\cup \{(Z, (O, I), t^{startafter}(O), \infty) \mid O \in \mathcal{O}, I \in \mathcal{I}^{first}_{T_O}\} \\
&\cup \{(S, (O, I), 0, \infty) \mid O \in \mathcal{O}, I \in \mathcal{I}^{first}_{T_O}\} \\
&\cup \{(O, E, 0, \infty) \mid O \in \mathcal{O}\} \\
R &= \mathcal{C} \\
N_r &= \{(O, I) \mid C \in \mathcal{C}_I, C = r\} \\
&\cup \{L_C, U_C \mid r = C\} \qquad \forall r \in R \\
T^{chot}(n_1, n_2) &= \begin{cases}
t^{travel}(L(n_1), L(n_2)) & \text{if } n_1 = (O_1, I_1), n_2 = (O_2, I_2) \\
t^{travel}(L^D, L_O) & \text{if } n_1 = L_C, n_2 = (O, I) \\
t^{travel}(L_O, L^D) & \text{if } n_1 = (O, I), n_2 = U_C \\
0 & \text{if } n_1 = L_C, n_2 = U_C \\
\infty & \text{if } n_2 = L_C \text{ or } n_1 = U_C
\end{cases} \qquad \forall n_1, n_2 \in N^T, n_1 \neq n_2 \\
T^{proc}(n) &= \begin{cases}
t^{perform}(I) & \text{if } n = (O, I)) \\
0 & \text{if } n = L_C, n = U_C
\end{cases} \qquad \forall n \in N^T
\end{aligned}
$$

Appendix 3: Optimization algorithm

For the sake of simplicity, a specialized version of the S-graph minimizer algorithm is provided here, where the processing times and changeover times are resource-independent and the storage policy is UIS with LW-arcs. The objective is to minimize the value provided by the **bound** function, which can be anything (as long as it is monotonically increasing lower bound as required by the B&B

algorithm), for example the longest path of the graph for makespan minimization or the bounding function described in Sect. 4.2.

```
1: procedure SOLVE(N, N^T, N^P, A_1, R, N_r, T^proc, T^chot)
2:     opt := ∞
3:     S := {(G(N, A_1, ∅), N^T, R, ∅, ∅)}
4:     repeat
5:         (G(N, A_1, A_2), N^U, R', M, M^L) := take(S)
6:         z := bound((G(N, A_1, A_2), N^U, R', M, M^L))
7:         if z < opt ∧ isfeasible(G(N, A_1, A_2)) then
8:             if N^U = ∅ then
9:                 opt := z
10:                optimal_solution := (G(N, A_1, A_2), M)
11:            else
12:                r := select(R')
13:                for all n ∈ N_r ∩ N^U do
14:                    A'_2 := A_2 ∪ {(n', n, T^proc(n') + T^chot(n', n)) | (n', r) ∈ M^L}
15:                    S := S ∪ {(G(N, A_1, A'_2), N^U \ {n}, R',
       M ∪ {(n, r)}, M^L \ {(n', r) ∈ M^L} ∪ {(n, r)})}
16:                end for
17:                if N^U ⊆ ⋃_{r'∈R'\{r}} N_{r'} then
18:                    S := S ∪ {(G(N, A_1, A_2), N^U, R' \ {r}, M, M^L)}
19:                end if
20:            end if
21:        end if
22:    until S = ∅
23:    if opt ≠ ∞ then
24:        return optimal_solution
25:    end if
26: end procedure
```

$\mathcal{S}$ is the set of open nodes, which initially contains only the recipe-graph. The function **take** selects and removes a node from $\mathcal{S}$. The selection logic can be implemented in many ways, the tests mentioned in this paper used a depth-first traversal.

If there are still decisions to be made at the current node, a resource *r* is selected by the **select** method, in which there is also a lot of freedom in the selection logic. The method used in the tests follows a lexicographic resource order. Then a new node is added to $\mathcal{S}$ for each task *n*, with *n* appended to the execution list of *r*. Also, if every possible task can be executed with different resources, as well, another node is added to $\mathcal{S}$ where *r* is removed from the set of resources to be scheduled.

Appendix 4: Problem generator script

The following Python function was used to generate a random problem instance with a given parameter set. A different random seed was set for each parameter set. The generator parameters are shown in Table 5. Travel times between locations are calculated from the Euclidean distance with 1 du/tu as travel speed.

Listing 1 Python function to generate a problem instance

```
import random

def generateInstance(params):
  prob = ProblemData()
  prob.shiftStart = 0
  prob.shiftEnd = 40
  # generate cars
  for c in range(0, params.carNumber):
    car = Car(name=c)
    car.fixCost = random.uniform(400, 600)
    car.travelCost = random.uniform(100, 200)
    car.maxDist = 1000
    prob.cars += car
  # generate tasks
  emptytasks = []
  for t in range(0, params.taskNumber):
    task = Task(name=t)
    task.cost = random.uniform(400, 600)
    carcount = random.randint(1, 2)
    usable = cars
    random.shuffle(usable)
    for i in range(0, carcount):
      task.carTimes[usable[i].name] = random.uniform(1, 2)
    emptytasks += task
  # decompose set of tasks into products and generate their
    recipes
  prodsizes = []
  for pr in range(0, params.orderNumber):
    product = Product(name=pr)
    prob.products += [product]
    prodsizes += [1]
  for i in range(0, params.taskNumber - params.orderNumber):
    prodsizes[random.randrange(0, len(prodsizes))] += 1
  taskidx = 0
  prodidx = 0
  procTimeSums = []
  for ps in prodsizes:
    tasksOfProd = emptytasks[taskidx : taskidx+ps]
    i = 1
    for t in tasksOfProd:
      succ = random.randint(i, len(tasksOfProd))
      if succ == len(tasksOfProd):
        t.successors.append(prob.products[prodidx].name)
      else:
        t.successors.append(tasksOfProd[succ].name)
      i += 1
    prob.tasks += tasksOfProd
    procTimeSums.append(0)
    for i in range(taskidx, taskidx+ps):
      procTimeSums[prodidx] +=
    min(emptytasks[i].carTimes.values())
    taskidx += ps
    prodidx += 1
  # add depot location at origin
```

```
53    depot = Location()
54    depot.name = 'depot'
55    depot.x, depot.y = 0,0
56    # generate orders, each with a different location
57    orders = []
58    i = 0
59    for product in prob.products:
60      order = Order(name=product.name, exactStart=False)
61      order.deadline = random.uniform(params.orderDeadlineMin, 40)
62      order.delayCost = random.uniform(200, 300)
63      order.procTimeBound = procTimeSums[i]
64      i += 1
65      location = Location(name=product.name)
66      location.x = random.uniform(0, 2)
67      location.y = random.uniform(0, 2)
68      order.location = location.name
69      prob.locations.append(location)
70      orders.append(order)
71    ordersWithTiming = list(orders)
72    random.shuffle(ordersWithTiming)
73    for i in range(0, params.ordersWithExactStart):
74      ordersWithTiming[i].exactStart = True
75    random.shuffle(ordersWithTiming)
76    for i in range(0, params.ordersWithTimeWindow):
77      self.genTimeWindowFor(ordersWithTiming[i])
78      windowLen = min(40, order.procTimeBound *
      (1+params.windowBuffer))
79      windowStart = random.uniform(0, 40 - windowLen)
80      order.startAfter = windowStart
81      order.deadline = windowStart + windowLength
82    prob.orders = orders
83    return prob
```

References

Adonyi R (2008) Batch process scheduling with the extensions of the S-graph framework. Ph.D. Thesis, University of Pannonia

Adonyi R, Holczinger T, Majozi T, Friedler F (2007) Novel branching procedure for S-graphs based scheduling of batch processes. In: Proceedings of 19th Polish conference of chemical and process engineering, p 9

Adonyi R, Biros G, Holczinger T, Friedler F (2008) Effective scheduling of a large-scale paint production system. J Clean Prod 16(2):225–232

Adonyi R, Heckl I, Szalamin A, Olti F (2010) Routing of railway systems with the S-graph framework for effective scheduling. Chem Eng Trans 21:913–918

Braekers K, Ramaekers K, Nieuwenhuyse IV (2016) The vehicle routing problem: State of the art classification and review. Comput Ind Eng 99:300–313

Bräysy O, Gendreau M (2005a) Vehicle routing problem with time windows, part I: route construction and local search algorithms. Transp Sci 39(1):104–118

Bräysy O, Gendreau M (2005b) Vehicle routing problem with time windows, part II: metaheuristics. Transp Sci 39(1):119–139

Dantzig GB, Ramser JH (1959) The truck dispatching problem. Manag Sci 6(1):80–91

D'Ariano A, Pacciarelli D, Pranzo M (2007) A branch and bound algorithm for scheduling trains in a railway network. Eur J Oper Res 183(2):643–657

Ferrer-Nadal S, Capón-García E, Méndez CA, Puigjaner L (2008) Material transfer operations in batch scheduling. A critical modeling issue. Ind Eng Chem Res 47:7721–7732

Gendreau M, Tarantilis CD (2010) Solving large-scale vehicle routing problems with time windows: the state-of-the-art. Cirrelt, Montreal
Ghaeli M, Bahri PA, Lee P, Gu T (2005) Petri-net based formulation and algorithm for short-term scheduling of batch plants. Comput Chem Eng 29(2):249–259
Guinet A (1993) Scheduling sequence-dependent jobs on identical parallel machines to minimize completion time criteria. Int J Prod Res 31(7):1579–1594
Harzi M, Krichen S (2017) Variable neighborhood descent for solving the vehicle routing problem with time windows. Electronic notes in discrete mathematics. In: 4th international conference on variable neighborhood search, vol. 58, pp. 175–182
Hegyháti M (2015) Batch process scheduling: extensions of the S-graph framework. Ph.D. Thesis, University of Pannonia
Hegyháti M, Friedler F (2010) Overview of industrial batch process scheduling. Chem Eng Trans 21:895–900
Hegyháti M, Majozi T, Holczinger T, Friedler F (2009) Practical infeasibility of cross-transfer in batch plants with complex recipes: S-graph vs MILP methods. Chem Eng Sci 64(3):605–610
Hegyháti M, Holczinger T, Szoldatics AA, Friedler F (2011) Combinatorial approach to address batch scheduling problems with limited storage time. Chem Eng Trans 25:495–500
Hegyháti M, Ösz O, Kovács B, Friedler F (2014) Scheduling of automated wet-etch stations. Chem Eng Trans 39:433–438
Holczinger T, Romero J, Puigjaner L, Friedler F (2002) Scheduling of multipurpose batch processes with multiple batches of the products. Hung J Ind Chem 30:263–270
Juan AA, Adelantado F, Grasman SE, Faulin J, Montoya-Torres JR (2009) Solving the capacitated vehicle routing problem with maximum traveling distance and service time requirements: an approach based on Monte Carlo simulation. In: Winter simulation conference, WSC '09, pp 2467–2475
Larsen R, Pranzo M, D'Ariano A, Corman F, Pacciarelli D (2014) Susceptibility of optimal train schedules to stochastic disturbances of process times. Flex Serv Manuf J 26(4):466–489
Masoud M, Kozan E, Kent G (2011) A job-shop scheduling approach for optimising sugarcane rail operations. Flex Serv Manuf J 23(2):181–206
Méndez CA, Cerdá J, Grossmann IE, Harjunkoski I, Fahl M (2006) State-of-the-art review of optimization methods for short-term scheduling of batch processes. Comput Chem Eng 30(6–7):913–946
Reil S, Bortfeldt A, Mönch L (2018) Heuristics for vehicle routing problems with backhauls, time windows, and 3d loading constraints. Eur J Oper Res 266(3):877–894
Ruiz R, Andrés-Romano C (2011) Scheduling unrelated parallel machines with resource-assignable sequence-dependent setup times. Int J Adv Manuf Technol 57(5):777–794
Samà M, D'Ariano A, D'Ariano P, Pacciarelli D (2017) Scheduling models for optimal aircraft traffic control at busy airports: tardiness, priorities, equity and violations considerations. Omega 67:81–98
Sanmartí E, Puigjaner L, Holczinger T, Friedler F (2002) Combinatorial framework for effective scheduling of multipurpose batch plants. AIChE J 48(11):2557–2570
Schoppmeyer C, Subbiah S, Engell S (2012) Modeling and solving batch scheduling problems with various storage policies and operational policies using timed automata. In: Karimi IA, Srinivasan R (eds) 11th international symposium on process systems engineering, vol 31, Elsevier, pp 635–639
Shaik MA, Floudas CA (2009) Novel unified modeling approach for short-term scheduling. Ind Eng Chem Res 48(6):2947–2964
Urgo M, Váncza J (2018) A branch-and-bound approach for the single machine maximum lateness stochastic scheduling problem to minimize the value-at-risk. Flex Serv Manuf J. https://doi.org/10.1007/s10696-018-9316-z

Tibor Holczinger is an associate professor of the Department of Applied Informatics at University of Pannonia. He received his MSc in Computer Engineering (1997) and PhD in Information Technology (2004). His research includes the scheduling of production systems focusing on S-graph framework and the modeling of manufacturing systems in Industry 4.0. He lectures courses about C/C++ programming, operation research, production optimization and supply chain modeling. He is a member of the Hungarian Operations Research Society. He published 18 scientific papers that received 74+ independent citations.

Olivér Ősz is an assistant lecturer and PhD student at Széchenyi István University, Győr, Hungary. He obtained MSc in Computer Engineering (2016) at University of Pannonia. His PhD research is about application of scheduling methods for various industrial systems. He lectures courses about C/C++ programming.

Máté Hegyháti is a senior research fellow at the Department of Information Technology, Széchenyi István University, Győr, Hungary. He received his MSc in Computer Engineering (2010) and PhD in Information Technology (2015). His research focuses on the scheduling of batch production systems with special emphasis on the application and development of the S-graph framework. He lectures courses about basics of optimization, formal language and automata theory, modeling of discrete event systems.