The Computer Journal Advance Access published December 20, 2013

doi:10.1093/comjnl/bxt140

A Message-Passing Approach to Decentralized Parallel Machine Scheduling

MERITXELL VINYALS[1,*], KATHRYN S. MACARTHUR[1], ALESSANDRO FARINELLI[2], SARVAPALI D. RAMCHURN[1] AND NICHOLAS R. JENNINGS[1,3]

[1]*School of Electronics and Computer Science, University of Southampton, Southampton, UK*
[2]*University of Verona, Verona, Italy*
[3]*Department of Computing and Information Technology, King Abdulaziz University, Saudi Arabia*
**Corresponding author: mv2y11@ecs.soton.ac.uk*

This paper tackles the problem of parallelizing heterogeneous computational tasks across a number of computational nodes (aka agents) where each agent may not be able to perform all the tasks and may have different computational speeds. An equivalent problem can be found in operations research, and it is known as scheduling tasks on unrelated parallel machines (also known as $R\|C_{\max}$). Given this equivalence observation, we present the spanning tree decentralized task distribution algorithm (ST-DTDA), the first decentralized solution to $R\|C_{\max}$. ST-DTDA achieves decomposition by means of the min–max algorithm, a member of the generalized distributive law family, that performs inference by message-passing along the edges of a graphical model (known as a junction tree). Specifically, ST-DTDA uses min–max to optimally solve an approximation of the original $R\|C_{\max}$ problem that results from eliminating possible agent-task allocations until it is mapped into an acyclic structure. To eliminate those allocations that are least likely to have an impact on the solution quality, ST-DTDA uses a heuristic approach. Moreover, ST-DTDA provides a per-instance approximation ratio that guarantees that the makespan of its solution (optimal in the approximated $R\|C_{\max}$ problem) is not more than a factor ρ times the makespan of the optimal of the original problem. In our empirical evaluation of ST-DTDA, we show that ST-DTDA, with a min-regret heuristic, converges to solutions that are between 78 and 95% optimal whilst providing approximation ratios lower than 3.

Keywords: parallel machine scheduling; task-allocation; min–max; decentralized coordination; bounded approximation; GDL; junction tree

Received 6 December 2012; revised 3 October 2013
Handling editor: Juan Antonio Rodriguez-Aguilar

1. INTRODUCTION

As online and real-time services (e.g. video processing, image classification and route planning) in a range of applications (e.g. social networks, search and rescue and transportation), grow in scale (e.g. hundreds of actors needed to be coordinated during the Haiti earthquake to find victims and hundreds of Mechanical Turk workers needed to execute a project decomposed into micro-tasks), it is crucial that the systems underpinning such services can distribute computation efficiently across the available computational nodes in order to minimize latencies. In the last few years, technologies such as Hadoop,[1] Amazon's Elastic Compute Cloud (EC2)[2] and Ushahidi's CrowdMap platform[3] have been developed to meet this challenge by distributing computation across, typically heterogeneous, computational nodes such as data centres or mobile devices, possibly belonging to different stakeholders (e.g. government agencies, private companies or individuals), and even human operators (e.g. online crowds on Amazon Turk or volunteers on the ground). Given this heterogeneity

[1]http://hadoop.apache.org/.
[2]http://aws.amazon.com/ec2/.
[3]http://www.ushahidi.com.

across computational nodes and the variety of tasks that can be requested, it may not be possible (or even desirable) for all nodes to perform all tasks. For example, only nodes equipped with Graphics Processing Units will be able to handle HD video processing tasks, while only Creole-speaking crowd members may be able to do translation tasks in Haiti. Moreover, significant delays may be introduced if hard computational tasks are allocated to the slowest nodes in the system or if the most efficient nodes are overloaded with too many tasks. Therefore, unless such dependencies (i.e. task-specific nodes leading to node eligibility constraints) and limitations (i.e. computational inefficiencies) are factored into the parallel execution of these computational tasks, these distributed information systems may suffer from significant latencies and, in the worst case, failures to deliver real-time services that may cause major financial losses or, in disaster settings, loss of life.

Now, an analogue to this particular class of distributed task allocation problem has been widely studied in Operations Research and is known as *scheduling on unrelated parallel machines* or $R\|C_{\max}$ [1, 2]. This problem involves a set of unrelated heterogeneous machines (equivalent to computational nodes),[4] and a set of computational tasks that must be performed by those computational nodes, potentially under node eligibility constraints (where only a subset of nodes are capable of processing a task). The objective is to find a mapping from tasks to nodes such that the maximum finish time across nodes, known as the *makespan*, is minimized.

While many algorithms (e.g. [3–5]) have been developed to solve $R\|C_{\max}$, they typically focus on the traditional version of the problem in which a task can be executed by any node. Hence, they do not take account of node eligibility constraints. Moreover, all current approaches require the existence of a central authority that gathers information about all the tasks and all the nodes in the system. This introduces a single point of failure, which is highly undesirable in the context of the online and real-time services exemplified above. In addition, sending all the data to a central authority may be infeasible due to computation or communication constraints. Finally, computational nodes may be managed and owned by different stakeholders that could be unwilling to send such data to a central authority for privacy or security issues.

To overcome these problems, we advocate the design of *decentralized* algorithms, where both information and computation are distributed across individual nodes in the network, thus avoiding the need of such central authority. In particular, we model the system of distributed computational nodes as a multi-agent system (MAS) (and hence recast the problem as the agent-based $R\|C_{\max}$ problem) where each node is managed by a software agent [6]. Each agent is able to communicate with other nodes to exchange information and make decisions about which tasks to run based on its individual computational capacity and load, as well as its capability to run specific types of tasks. Now, since the $R\|C_{\max}$ problem is known to be intractable in the general case [7], we focus on the design of *approximate algorithms*. Hence, we sacrifice optimality for a practical approach that provides bounds on the error with respect to the optimal solution.

In particular, we develop the spanning tree (ST) decentralized task distribution algorithm (ST-DTDA). ST-DTDA specifies a message-passing protocol for agents to compute high quality approximate solutions in a distributed, efficient manner. By doing so, we advance the state of the art in the following ways:

(i) We provide a novel representation of the (Agent-Based) $R\|C_{\max}$ problem in terms of a junction tree (JT) [8]. This representation is able to capture the constraints on the tasks that each agent can perform, as well as define the communication network used by the agent to exchange messages. Thus, we show that the task allocation constraints result in a *sparse* network that can be exploited to reduce computation.

(ii) We present the first decentralized task distribution algorithm (i.e. ST-DTDA) for Agent-Based $R\|C_{\max}$ by building upon the min–max algorithm, a standard message-passing algorithm that solves specific optimization problems using the generalized distributive law (GDL) [9]. In particular, ST-DTDA uses a heuristic approach to weight agent-task allocations and thus, eliminates the ones that are least likely to have an impact on the solution quality.

(iii) We show how ST-DTDA can bound the error of its solutions with respect to the optimal solution of the original problem by computing a per-instance approximate ratio.[5]

(iv) We demonstrate the effectiveness of our approach by empirically evaluating ST-DTDA on a range of network structures and using different distributions of task execution times. Our results show that ST-DTDA finds good solutions (78–95% of the optimal when using a *min-regret* heuristic) and outperforms a standard distributed hill-climbing approach in terms of solution quality.

When taken together, our results establish the first benchmarks for distributed solutions to $R\|C_{\max}$ for settings where not all tasks can be performed by all computational nodes.

The rest of this paper is structured as follows. Section 2 gives background on the current approaches used to solve $R\|C_{\max}$ and on the min–max algorithm. Next, Section 3 introduces the Agent-Based $R\|C_{\max}$ problem and maps it to

[4]Unrelated machines are those in which there is no relation between the execution times of a task on different machines. That is, the time taken by one machine does not affect the time taken by another machine for another task.

[5]A per-instance approximate ratio guarantees that the makespan of its solution is not more than a factor ρ times the makespan of the optimal solution.

a JT representation. Then, in Section 4 we present the ST-DTDA algorithm. Finally, in Section 5 we present our empirical evaluation and in Section 6 we conclude.

2. BACKGROUND

In this section, we provide the foundations of a decentralized solution to the $R\|C_{\max}$ problem. In particular, we first detail the basics of $R\|C_{\max}$ and discuss the relevant literature. Then, we describe the min–max algorithm which we use to develop ST-DTDA.

2.1. The $R\|C_{\max}$ problem

This work focuses on finding solutions to a common scheduling problem known as *scheduling on unrelated parallel machines* ($R\|C_{\max}$) [1]. In more detail, $R\|C_{\max}$ requires scheduling a number of tasks on a number of unrelated machines, and assumes that machines are not identical, so that task execution times can differ among them.

DEFINITION 2.1 ($R\|C_{\max}$). *Given a set T of tasks, a set M of unrelated machines, and for each $j \in T$ and $i \in M$, $p_{ij} \in \mathbb{Z}^+$, the time taken to process task j on machine i, the problem is to schedule the tasks on the machines so as to minimize the makespan.*[6]

Now, $R\|C_{\max}$ has been shown to be NP-hard, and hence, exact solution algorithms generally do not scale well [7]. For example, Horowitz and Sahni [3] give an exact algorithm for $R\|C_{\max}$, with a worst-case time complexity exponential in the number of tasks: $\mathcal{O}(\min\{|T|F, |M|^{|T|}\})$ where F is the end time of the schedule obtained by assigning each task to the processor on which its execution time is minimal and $|M|$ is the number of machines. Scheduling each task on the machine with the minimal execution time is unlikely to produce the optimal solution (i.e. the most efficient machines will be overloaded with too many tasks, increasing the global makespan), so the worst-case run-time for this algorithm is exponential.

Thus, in practical applications, it is more beneficial to focus on approximate algorithms to find solutions to $R\|C_{\max}$. For example, Lenstra *et al.* [5] developed a polynomial-time algorithm with a worst-case performance ratio of 2 (i.e. the solution value is guaranteed to be no more than twice the optimum). Later, Shchepin and Vakhania [4] developed a polynomial-time algorithm for $R\|C_{\max}$ with an improved worst-case bound of $2 - 1/|M|$.

Specifically, in this paper, we focus on $R\|C_{\max}$ problems with machine eligibility restrictions (see [10] for a recent survey in the area). This represents a more general and realistic model in which every machine is capable of executing only a subset of the tasks (as exemplified in Section 1). Such eligibility constraints can be introduced in a traditional $R\|C_{\max}$ problem by setting the execution time of a task on a given machine to ∞ if the machine is cannot perform that task. However, such a direct extension means that existing algorithms cannot exploit the sparsity resulting from eligibility constraints (i.e. each agent can only execute a limited number of tasks). Moreover, despite the existence of good centralized approximation algorithms for $R\|C_{\max}$, there exist no decentralized algorithms as yet.

In contrast, distributed search and constraint handling are a subfield of MASs specifically devoted to this paradigm of decentralized problem solving [11]. Many contributions to this field are message-passing algorithms [12–14], some of which are derived from distributed graphical models representations of the problem (e.g. [13, 14]). To date, however, none of the above-mentioned approaches have been applied to/studied for the $R\|C_{\max}$ problem and, as we show in this paper, can potentially provide high quality solutions under certain conditions. Hence, in the next section we describe just such a message-passing algorithm which forms the basis of our approach.

2.2. The min–max algorithm

In this section, we introduce the min–max algorithm, as well as the JT structure over which it is executed. Specifically, min–max is a member of the GDL framework, a family of message-passing algorithms that exploit the way a global function factors into a combination of local functions to compute the objective function in an efficient manner [9]. GDL is defined over two binary operations: marginalization and combination. These define a commutative semiring (see [15] for a more detailed introduction to semirings). In our case, since we are concerned with the problem of minimizing the makespan, such operations are minimization and maximization (the min–max GDL).[7] In order to guarantee optimality, GDL must operate on a JT [8]. A JT is a popular representation which is used in advanced message-passing algorithms that have been developed in fields such as bioinformatics, image processing and control theory [16]. We define a JT as follows [14].

DEFINITION 2.2. *A JT is a tree of cliques that can be represented as a tuple $\langle X, C, S, \Psi\rangle$ where:*

(i) $X = \{x_1, \ldots, x_n\}$ *is a set of variables defined over domains* $D_1, \ldots, D_n$;[8]
(ii) $C = \{C_1, \ldots, C_m\}$ *is a set of cliques such that each clique* $C_i \subseteq X$;

[6]Makespan: the total time needed to complete the whole set of tasks that has been scheduled on one or more machines.

[7]Concretely, min–max is based on the commutative semiring $\langle \mathbb{R}^+, \min, \max, \infty, 0\rangle$ where $\mathbb{R}^+$ is the set of elements of the semiring, *min* is the marginalization operator, *max* the combination operator, ∞ is the neutral element of marginalization (min) and 0 the neutral element of combination (max).

[8]In general, we will denote D_X as the cartesian product of the domain of each individual variable in X.

(iii) *S is a set of separators, where each separator S_{ij} is an edge between two cliques C_i, C_j containing the intersection of the variables they represent, i.e. $S_{ij} = C_i \cap C_j$;*

(iv) *$\Psi = \{\psi_1, \ldots, \psi_m\}$ is a set of potentials, one per clique, where potential ψ_i is a function $\psi_i : D_{X_i} \to \mathbb{R}^+$ whose scope is a subset of the clique to which it is assigned, namely $X_i \subseteq C_i$.*

Furthermore, the running intersection property must hold as follows:

If a variable x_i is in two cliques C_i and C_j, then it must also be in all cliques on the (unique) path between C_i and C_j.

We use $\mathbf{x_i}$ throughout to refer to a possible value of variable x_i, that is, $\mathbf{x_i} \in D_i$, and $\mathbf{X}$ to refer to a possible value for each variable in X, that is, $\mathbf{X} \in D_X$. Moreover, we will refer to $C_{i\setminus j}$ as the set of variables in clique C_i excluding those in the clique C_j ($C_{i\setminus j} = C_i \setminus C_j = C_i \setminus S_{ij}$).

Given a JT, the GDL approach consists of exchanging messages along the edges of the JT. In particular, here we focus on Action-GDL, a specialization of GDL formulated in [14] to efficiently solve distributed constraint optimization problems (e.g. problems in which the objective is to distributedly find the optimal assignment that maximizes the objective function). Action-GDL is executed over a directed JT[9] where the cliques are distributed across the set of agents. Here, we assume that each agent $a_i \in A$ is assigned a single clique C_i.

Now, to operationalize min–max using Action-GDL (which we term the min–max algorithm), we propose two message-passing phases: (1) *cost propagation*, in which cost messages are sent from leaves up to the root and (2) *value propagation*, in which optimal assignments are decided and communicated down the tree.

Algorithm 1 outlines the pseudocode for min–max (using Action-GDL). Given a JT, each agent a_i starts with a tuple $\langle C_i, S_i, \psi_i(X_i)\rangle$, where C_i stands for its clique, $S_i \subseteq S$ for the separators relating its clique to adjacent cliques and $\psi_i(X_i)$ for its potential function. An agent starts by running the procedure `DFS arrangement`, a decentralized depth-first search (DFS) algorithm that converts the undirected JT into a directed one by running a token passing mechanism as in [17] (line 1). As the outcome of this DFS, each agent knows its parent a_p and its children Ch$_i$ in the directed JT.

After this initialization phase, the agent starts the *cost propagation* phase (lines 2–7). Each agent a_i waits until it receives a cost message from each of its children cliques (line 2). Formally, a cost message $\mu_{j\to i}$ from agent a_j to agent a_i is a function that returns a cost for every configuration of variables in their separator ($\forall \mathbf{S_{ij}}$). In the next step, each agent a_i computes its local knowledge function, κ_i, by combining (using the *max*

[9]Any undirected JT can be converted into a directed one by picking a root clique and directing arcs from there outwards.

Algorithm 1 `min-max()` at agent a_i

```
Agent $a_i$ starts knowing $\langle C_i, S_i, \psi_i(X_i)\rangle$ and runs:
1: $\langle a_p, Ch_i\rangle$ = DFSTransveral($N_i$)
   // PHASE I: cost propagation
2: Wait until receive $\mu_{j\to i}$ from all $a_j \in Ch_i$
3: $\kappa_i(\mathbf{C_i}) = \max\left(\psi_i(\mathbf{X_i}), \max_{a_j \in Ch_i} \mu_{j\to i}(\mathbf{S_{ij}})\right)\ \forall \mathbf{C_i}$
4: if $a_i$ is not root then
5:     $\mu_{i\to p}(\mathbf{S_{ip}}) = \min_{\mathbf{C_{i\setminus p}}} \kappa_i(\mathbf{C}_i)\ \forall \mathbf{S_{ip}}$
6:     Send $\mu_{i\to p}$ to $a_p$
7: end if
   // PHASE II: value propagation
8: if $a_i$ is not root then
9:     Wait until receive $\langle \mathbf{S}^*_{ip}, V^*\rangle$ from $a_p$
10: end if
11: $\mathbf{C}^*_{\mathbf{i\setminus p}} = \arg\min_{\mathbf{C_{i\setminus p}}} \kappa_i(\mathbf{C}_{i\setminus p}; \mathbf{S}^*_{ip})$
12: $\mathbf{C}^*_{\mathbf{i}} = \mathbf{C}^*_{\mathbf{i\setminus p}} \cup \mathbf{S}^*_{\mathbf{ip}}$
13: if $a_i$ is root then
14:     $V^* = \kappa_i(\mathbf{C}^*_{\mathbf{i}})$
15: end if
16: for all $a_j \in Ch_i$ do
17:     Send $\langle \mathbf{S}^*_{\mathbf{ij}}, V^*\rangle$ to $a_j$
18: end for
19: return $\langle \mathbf{C}^*_i, V^*\rangle$
```

operator) the value of its potential function and all the messages received from its children for every possible assignment of variables in C_i (line 3). After computing its local knowledge function, if the agent is not the root, it sends a cost message, $\mu_{i\to p}$, to its parent, a_p (lines 4–7). Note that each agent a_i computes this message by marginalizing out all variables not common with its parent a_p (using the *min* operator) from its local knowledge function (line 5).

During the *value propagation* phase (lines 8–18), agents compute the optimal assignment of variables for their clique variables, along with the value of this optimal assignment, and propagate them down the tree. Each agent waits until it receives optimal assignments for all variables it has in common (in the separator) with its parent, $\mathbf{S}^*_{ip}$, as well as the value of the optimal assignment, V^* (lines 8–10). Then, the agent fixes the values of these pre-inferred variables in its local knowledge function and computes the assignment that minimizes its local knowledge function for the rest, $\mathbf{C}^*_{\mathbf{i\setminus p}}$ (line 11). At this point, agent a_i knows the optimal assignment for all its clique variables (line 12), $\mathbf{C}^*_{\mathbf{i}}$. If a_i is at the root node, after the cost propagation phase, it can assess the value of the optimal configuration as the value returned by its local knowledge function, V^* (lines 13–15). Then, each agent propagates the optimal configuration and its value down the tree to its children (lines 16–18). Note, however, that each agent only propagates the variable assignments required by its children's cliques, namely assignments for variables in their respective separators (line 17). Finally, each agent a_i returns the values specified in its optimal assignment[10] for its clique's variables ($\mathbf{C_i^*}$), being the optimal assignment $\mathbf{X}^*$ defined as

$$\mathbf{X}^* = \arg\min_{\forall \mathbf{X}} \max_{a_i \in A} \psi_i(\mathbf{X_i}), \tag{1}$$

where $\mathbf{X_i}$ contains the values assigned by $\mathbf{X}$ to the variables in X_i.

As stated in [14], we can readily assess the complexity of min–max from cliques' and separators' sizes. In more detail, the local memory/computation required by each agent a_i is exponential in the cardinality of its clique (i.e. scales with the joint domain of variables in its clique). However, the number of messages exchanged is linear in the number of edges in the JT and the communication complexity lies in the size of cost messages which are exponential in the cardinality of the respective separators.

Given the basic definitions of $R\|C_{\max}$ and description of the min–max algorithm, we next turn to formulating $R\|C_{\max}$ in a way that we can apply the min–max algorithm.

3. THE AGENT-BASED $R\|C_{\text{MAX}}$ PROBLEM

We recast $R\|C_{\max}$ as a multi-agent optimization problem where we have a set of computing entities that we refer to as *agents*. Hence, from now on, we term the problem the 'Agent-Based $R\|C_{\max}$' problem. This general modelling approach allows us to take into account the fact that (i) agents take charge of exchanging messages to find an allocation of computation tasks among them (ii) agents execute these tasks (using the hardware on which they run) and (iii) there may be restrictions on the tasks that each agent can execute. Thus, agents here may represent workers or organizations, as well as machines or processors. Let the set of agents be denoted as $A = \{a_1, a_2, \ldots, a_{|A|}\}$, and the set of computational tasks as $T = \{t_1, t_2, \ldots, t_{|T|}\}$. Each agent a_i can perform a set of tasks $T_i \subseteq T$ (recall that a task can be performed by a subset of agents, but not necessarily by all of them). For each agent $a_i \in A$, we denote a cost function as $\chi_i : T_i \to \mathbb{R}^+$, that returns the estimated amount of time it takes for agent a_i to perform a task $t \in T_i$. Thus, $\chi_i(t_k)$ returns the application-specific runtime required for agent a_i to perform task t_k.

A graphical representation of an $R\|C_{\max}$ problem in terms of the task dependency network (TDN) is given in Fig. 1. Here, there are four agents (squares) and five tasks (circles). Each agent is connected to the tasks it can potentially perform by edges in the graph, and edges are labelled with $\chi_i(t_k)$. Thus, for example, agent a_1 will incur a runtime of 30 timesteps to perform task t_2 whereas agent a_2 will only incur a runtime of 20. Finally, throughout N_i will denote the set of agents (neighbours) with which agent a_i shares some tasks ($N_i = \{j \in A, j \neq i \,|\, \exists t_k \in T_i \cap T_j\}$).

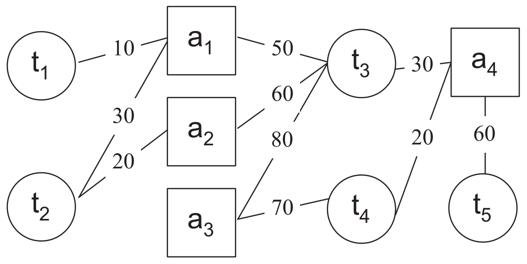


FIGURE 1. Example of a TDN. Agents are represented by squares, tasks by circles and edges between agents and tasks indicate an agent can perform a task (at a cost denoted on the edge).

Given this, the problem is to schedule all of the tasks in T across the agents in A such that all tasks are completed and the makespan is minimized. Note that this problem is equivalent to $R\|C_{\max}$ (as described in Section 2) but formulated from the decentralized perspective of an MAS. We next formally define the objective of the Agent-Based $R\|C_{\max}$ problem.

3.1. Objective function

The objective of Agent-Based $R\|C_{\max}$ is to find a mapping $m : A \to 2^T$ from agents to the set of tasks, such that the makespan is minimized. In particular, we wish to find this mapping subject to a number of constraints. First, each agent may only be assigned tasks that it can execute:

$$m(a_i) \subseteq T_i \quad \forall a_i \in A.$$

Secondly, each task must only be assigned to one agent:

$$m(a_i) \cap m(a_j) = \emptyset \quad \forall a_i, a_j \in A,\ i \neq j.$$

Thirdly, all tasks must be assigned:

$$\bigcup_{a_i \in A} m(a_i) = T$$

in which $m(a_i)$ denotes the set of tasks allocated to agent a_i, under mapping m. Given this, our objective is to find a set of optimal mappings M^* as follows:

$$M^* = \arg\min_{\forall m} \max_{a_i \in A} \sum_{t_k \in m(a_i)} \chi_i(t_k). \tag{2}$$

For instance, Fig. 2 depicts two optimal mappings for the TDN in Fig. 1 where the optimal allocations from agents to tasks are shown with arrows. Thus, the optimal mapping m^* in Fig. 2a is defined as: $m^*(a_1) = \{t_1, t_2\}$, $m^*(a_2) = \{t_3\}$, $m^*(a_3) = \{t_4\}$, $m^*(a_4) = \{t_5\}$ with a makespan value of $\max(10 + 30, 60, 70, 60) = 70$.

[10] In the presence of multiple optimal assignments, the value propagation phase itself ensures that agents converge to the same one.

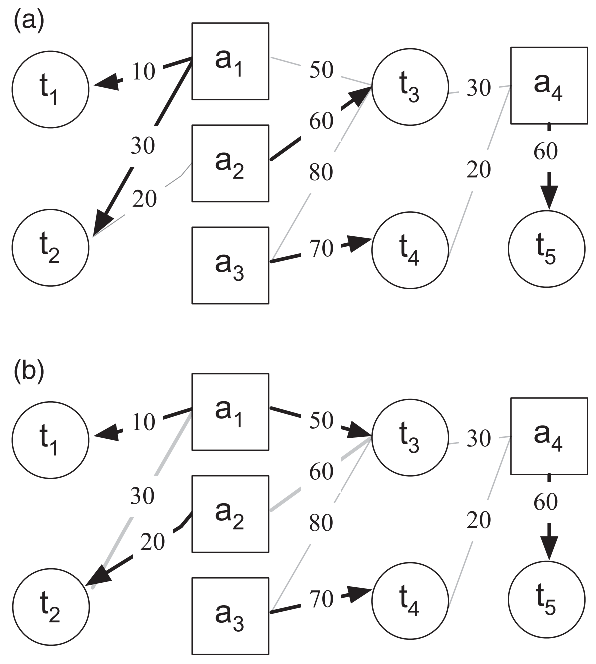


FIGURE 2. Two optimal mappings from agents to tasks with makespan $V^* = 70$, for the problem in Fig. 1. Arrows between agents and tasks depict an agent being assigned to a task.

Now, in order to solve the objective function given in Equation (2) in a decentralized way, in the next section we show how to represent this problem as a JT over which efficient message-passing algorithms such as the min–max algorithm can operate.

3.2. JT representation

To represent an $R\|C_{\max}$ problem as a JT, we need to represent its three components, namely tasks, agents and allocations, in terms of variables, cliques and edges of the JT.

Formally, the JT is built for an $R\|C_{\max}$ problem as follows:

(1) for each task $t_k \in T$, a variable x_k is created whose domain contains all of the indices of agents that can complete task t_k, $D_k = \{i | a_i \in A, t_k \in T_i\}$;[11]
(2) for each agent $a_i \in A$, a clique C_i is created containing the variables related to the set of tasks the agent a_i can execute, $X_i = \{x_k | t_k \in T_i\}$;[12]
(3) each clique $C_i \in C$ is assigned a potential function encoding the cost function of agent a_i. Formally,

$$\psi_i(X_i) = \sum_{\substack{t_k \in T_i,\\ x_k = i}} \chi_i(t_k); \tag{3}$$

(4) any two cliques C_i and C_j that share variables are joined by one edge that contains the intersection of their variables, i.e. $S_{ij} = X_i \cap X_j$.

Now, the graph resulting from applying steps 1–4 above does not always correspond to a JT. This is because the resulting set of edges may not form an acyclic graph (and thus it may form a junction graph, not a JT). Over a junction graph, the min–max algorithm is neither guaranteed to converge nor to find the optimal solution. However, it is always possible to convert this junction graph to a JT, at the expense of increasing the size and number of cliques and separators. Thus, to convert the junction graph to a JT we can apply the following steps [18, 19]:[13]

(5) select a subset of edges that form a connected tree among cliques;
(6) enlarge cliques (and the corresponding separators) until the running intersection property is satisfied (by running a distributed protocol such as the one described in [18]).

In what follows, we provide an example of how the steps above are applied to a TDN.

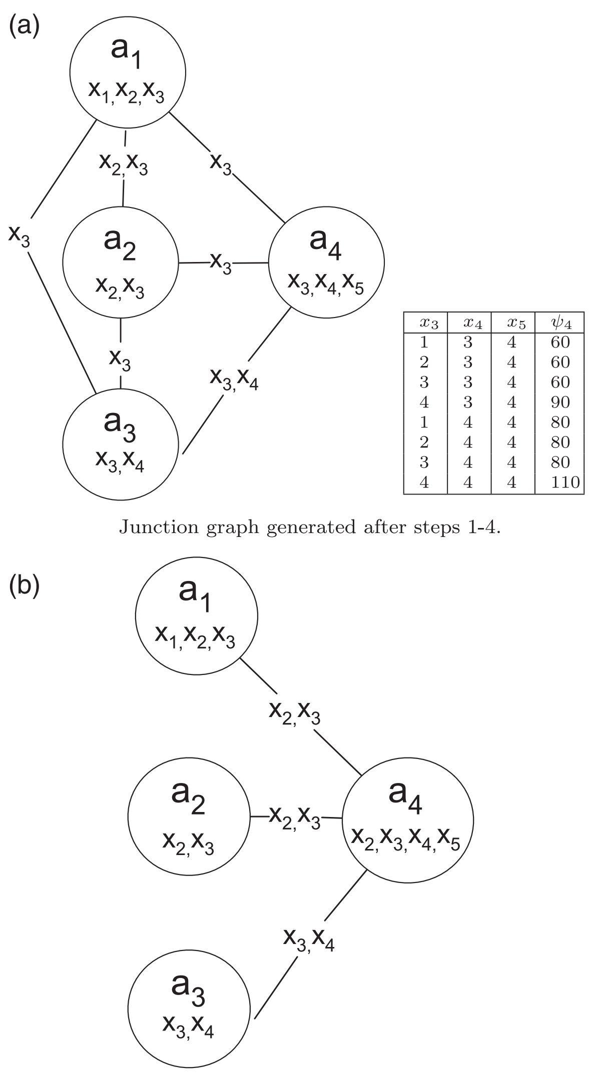
Fig. 3a depicts the graph resulting from applying steps 1–4 to the TDN from Fig. 1. In more detail, the graph contains five variables, $\{x_1, x_2, x_3, x_4, x_5\}$ that correspond to the five tasks in the figure. Each agent a_i's clique, denoted by a large circle, contains all variables in X_i. Thus, the set of variables corresponding to agent a_2's clique, X_2, is composed of x_2 and x_3, which are the two tasks that a_2 can perform in Fig. 1. The domain of x_2 is composed of two values, namely 1 and 2, corresponding to the indices of the agents that can perform task 2, a_1 and a_2. Moreover, Fig. 3 depicts the potential function of agent a_4, ψ_4, in a tabular form (listing one entry per configuration of variables). Thus, ψ_4, defined over variables x_3, x_4 and x_5, returns a runtime of 80 for the configuration $x_3 = 3$, $x_4 = 4$ and $x_5 = 4$, which is the runtime incurred by a_4 to complete tasks 4 and 5 in Fig. 1. Edges are labelled with the intersection of two cliques. Thus, agent a_2 is linked to a_3 by an edge that contains the only common variable in their cliques: x_3. As we can see, this junction graph is not acyclic and thus, steps 5 and 6 are needed.

Fig. 3b shows the JT resulting from applying steps 5 and 6 over the junction graph in Fig. 1. We observe that from the set of edges in the initial junction graph, only those joining the clique of agent a_4 with the rest of cliques have been selected to compose the JT. Moreover, the clique of agent a_4 has been enlarged to include variable x_2 in order to satisfy the running intersection property (e.g. x_2 is already included in the cliques of a_1 and a_2 and so it should be also included in the clique of a_3 because the latter is in the path connecting the first two).

Now, this representation allows the application of the min–max algorithm to find an optimal solution to Agent-Based $R\|C_{\max}$ in a decentralized fashion. However, the complexity of

[11]Note that, by doing so, we enforce the constraint that exactly one agent must perform every task.
[12]$x_k = i$ means that agent a_i is allocated to t_k.

[13]Note that if the TDN is acyclic, then steps 5 and 6 are never needed since the junction graph that results from applying steps 1–4 is guaranteed to be a JT.



x_3	x_4	x_5	ψ_4
1	3	4	60
2	3	4	60
3	3	4	60
4	3	4	90
1	4	4	80
2	4	4	80
3	4	4	80
4	4	4	110

Junction graph generated after steps 1-4.

Junction tree generated after steps 5-6 are applied on (a).

FIGURE 3. A JT representation of the scenario given in Fig. 1. Large circles are cliques, with the elements of the cliques listed. Edges are labelled with common variables between cliques.

running min–max over this JT is exponential in the cardinality of the largest clique (see Section 2.2). This renders this approach computationally intractable in the general case where all agents can perform an unrestricted number of tasks. Crucially, in the particular case when the TDN is acyclic, the computational complexity grows exponentially in the number of tasks which a_i *is connected to in the TDN*. Since, by definition, the cardinality of an agent's clique is bounded below by the number of tasks that an agent is connected to ($|C_i| \geq T_i$), the computational complexity of the algorithm will also be bounded by this. Hence, the main contribution of this paper, which we detail in the next section, is the formulation of solutions that exploit this sparsity in acyclic TDNs that have bounded agent degrees.

Algorithm 2 `ST-DTDA()` at agent a_i

```
Require: The set of tasks ($T_i$), the set of neighbours ($N_i$), the heuristic function ($h_i$), the maximum degree ($\Delta$) and a lower bound ($V^{LB}$)
    // Step 1: Ensure an acyclic TDN
1:  $\tilde{T}_i$ = computeSTApproximation($T_i$, $N_i$, $h_i$, $\Delta$)
    // Step 2: Solving the acyclic TDN
2:  $\langle m(a_i), \tilde{V}^*, \tilde{X}_i, \tilde{S}_i \rangle$ = solveSTApproximation($\tilde{T}_i$, $N_i$)
    // Step 3: Computing per-instance ratio
3:  $\rho$ = computeApproximationRatio($\tilde{X}_i$, $\tilde{S}_i$, $\tilde{V}^*$, $V^{LB}$)
4:  return $\langle m(a_i), \tilde{V}^*, \rho \rangle$
```

4. THE ST-DTDA ALGORITHM

In this section, we describe ST-DTDA, an approximate algorithm to solve the Agent-Based $R\|C_{\max}$ problem. The key feature of ST-DTDA involves a step that prunes the JT representation of the problem in order to allow the min–max algorithm to converge to high quality solutions with low computational cost.[14] Moreover, this allows to provide a bound on the quality of the solution computed. In more detail, ST-DTDA prunes some agent-task allocations so as to form an acyclic TDN where the maximal number of tasks connected to an agent is bounded by a parameter Δ.[15] It then applies the min–max algorithm to compute a high quality solution as well as a worst case bound on it.

Specifically, ST-DTDA consists of three steps (outlined in Algorithm 2):

(1) *Ensure an acyclic TDN*— ST-DTDA agents create an acyclic TDN by finding an ST of the original problem with a bounded number of tasks per agent and omitting all agent-task allocations not contained in the ST.
(2) *Solving the acyclic TDN*—ST-DTDA agents run min–max to optimally solve the JT that encodes the acyclic TDN created in step 1 and find an approximate solution for the $R\|C_{\max}$ problem.
(3) *Computing the per-instance approximation ratio*—ST-DTDA agents execute a third message-passing phase to compute a worst-case bound (ϵ) on the distance between the quality of the optimal solution in the original problem and the quality of the solution of the ST-approximate problem. Then, ST-DTDA agents use this error bound and a lower bound on the optimal

[14]This approach was demonstrated for other algorithms from the GDL family [20], adding a preprocessing step that transforms the original problem to one that is acyclic, can reduce the computational complexity.

[15]Only the tasks that can potentially be executed by at least two agents are considered when computing the agent's degree. Single-connected tasks can be omitted from the TDN by adding their execution times to the initial agent's makespan.

solution value to compute a per-instance approximation ratio.

After running ST-DTDA, each agent knows the set of tasks allocated to it, $m(a_i)$, as well as the makespan, $\tilde{V}^*$, and a per-instance approximation ratio, ρ, of its allocation. We elaborate further on these steps in the remainder of this section.

4.1. Ensuring an acyclic TDN

In this step, the agents build an ST of the TDN with a bounded number of tasks per agent. The key issue is that producing any ST of the TDN involves ignoring some of the potential agent-task allocations (i.e. it produces an acyclic *approximation* of the original network). Therefore, it is crucial that the possible solutions in the ST are of high quality—preferably containing the optimal solution for the original network and, if not, a solution whose makespan is close to that of the optimal. With this aim, agents assign a weight to each agent-task allocation in the TDN that estimates its impact on the solution makespan. Next, agents eliminate the agent-task allocations that are less likely to generate low makespans. However, since computing the exact values for these weights is as difficult as solving the original problem, ST-DTDA uses a heuristic approach to approximate them. Therefore, in this step, agents assign a weight $\omega_{ik} \in \mathbb{R}^+$ to each of the edges that have not already been included in the ST.

First, we detail the particular distributed procedure that ST-DTDA agents follow to compute the ST approximation of the TDN in which each agent is connected to at most Δ tasks.[16] Secondly, we propose three different heuristics functions to weight the edges of the TDN.

4.1.1. Decentralized ST approximation computation

Using Algorithm 3, ST-DTDA agents build the ST iteratively, adding, at each iteration, the edge with minimal weight to the ST by implementing a distributed version of Kruskal's Maximum ST algorithm [21].[17] In more detail, Algorithm 3 outlines the `computeSTApproximation` procedure. Each agent is given the set of tasks it can potentially execute (T_i), the set of neighbours with which a_i shares some tasks (N_i), a heuristic edge weighting function (h_i) and the maximum number of tasks per agent (Δ). An agent starts by running the procedure `DFSTransveral` to compute a directed agent-induced ST that agents will use as a (directed) communication tree (line 1). As the outcome of this DFS, each agent knows its parent a_p and its children Ch_i in the tree. Then, the agent initializes the set of tasks to which it is connected in the ST,

Algorithm 3 `computeSTApproximation()` at agent a_i

Require: The set of tasks (T_i), the set of neighbours (N_i), the heuristic function (h_i) and the maximum number of tasks per agent (Δ)

```
1: ⟨$a_p, Ch_i$⟩ = DFSTransveral($N_i$)
2: $CT = \tilde{T}_i = \{t_k \in T_i | t_k \notin T_j, j \neq i\}$
3: $l = |\tilde{T}_i|$, $w^* = -\infty$
4: while $w^* \neq \infty$ do
5:     if $|\tilde{T}_i| - l < \Delta$ then
6:         $\langle a^*, t^*, w^* \rangle$ = computeMinimalEdge($T_i \setminus \tilde{T}_i$)
7:     else
8:         $\langle a^*, t^*, w^* \rangle$ = computeMinimalEdge($\emptyset$)
9:     end if
10:    $CT = CT \cup \{t^*\}$
11:    if $a^* == a_i$ then
12:        $\tilde{T}_i = \tilde{T}_i \cup t^*$
13:    end if
14: end while
15: for all $t_k \in T_i \setminus CT$ do
16:     $a^* = \arg\min_{\{a_j | t_k \in T_j\}} h_j(t_k)$
17:     if $a_i == a^*$ then
18:         $\tilde{T}_i = \tilde{T}_i \cup t_k$
19:     end if
20: end for
21: return $\tilde{T}_i$
```

denoted as $\tilde{T}_i$, and the set of connected tasks, denoted as CT, as the set of tasks for which a_i is the only agent that can execute them (line 2). Although the edges between a_i and these tasks will be included in the ST-TDN anyway, adding them initially helps a_i to predict its final makespan with the heuristic. Moreover, each agent keeps track of this number of initial single-connected tasks because they are not considered when bounding the agent's degree by Δ (line 3).

Then, each agent iterates over the main loop of this step until no further edge can be added to the ST (lines 4–14), finding at each iteration the not added edge with minimal weight by repetitive calls to `computeMinimalEdge` (line 19). This procedure takes as an input the set of tasks whose edges with the agent can be considered as minimal. If the number of tasks that it is connected to does not exceed the maximum (line 5), then this set of tasks correspond to the tasks not added up to that iteration (line 6). Otherwise, the agent calls `computeMinimalEdge` with the empty set to avoid being linked to more tasks (line 7). Finally, the agent a_i adds to its set of tasks all tasks that have not been connected so far and for which a_i is the agent with the lowest task execution time (lines 15–20). Note that the inclusion of these tasks does not increase Δ since agent a_i will be the only agent connected to them in the ST. At the end of this phase, each agent returns the set of tasks to which it is connected in the ST, $\tilde{T}_i$.

Up to this point, we have described this ST-DTDA step independently of the heuristic edge weighting function used. Hence, for the sake of completeness, in the next section we

[16]This is excluding single-connected tasks: Δ does not count single-connected tasks.

[17]The `computeSTApproximation` procedure can be seen as a distributed version of Kruskal's algorithm with the difference that in Kruskal's algorithm all weights are assumed static during the ST computation whereas here the weights change depending on the set of edges already added to the ST.

propose a number of heuristic edge weighting functions and illustrate their operation using as a running example a trace of ST-DTDA over the TDN depicted in Fig. 1.

4.1.2. *Heuristic edge weighting functions*

We propose three different heuristics to weight edges of the TDN, namely the *best-case*, the *worst-case* and the *min-regret* heuristics. Each heuristic explores an alternative method for encoding the impact of edges on the final makespan. The *best-case* heuristic is a myopic heuristic that simply weights each agent-task allocation with the corresponding task processing time. As a result, we use it as a baseline for our approach. The *worst-case* and the *min-regret* heuristics incorporate information regarding the expected agent workload in their weights. Moreover, *min-regret* aims to improve over *worst-case* by bringing forward the decision regarding the agent-task allocations with higher impact by minimizing the *regret* (i.e. the potential to increase the makespan should the wrong edges be pruned) instead of the processing time. In what follows, all three weighting heuristics are formulated from the point of view of the agents. Thus, heuristic h_i is a function that given a task $t_k \in T_i$ returns the weight for the edge between a_i and t_k, namely w_{ik}.

The best-case heuristic. This simply weights the edge between a_i and t_k with the time it takes agent a_i to compute task t_k. Formally,

$$h_i(t_k) = \chi_i(t_k). \tag{4}$$

Note that the weights given by this heuristic are independent from the edges already added to the ST. The intuition behind this is that by removing high-weight edges we would remove the most costly agent-task allocations, which are the least likely to be in the optimal solution. We refer to this heuristic as *best-case* since it assumes that the only task that will be allocated to a_i is t_k. Thus, the ST that results from this heuristic always encodes the greedy allocation, $\check{m}$, that assigns each task to the agent with the lowest task execution time (i.e. the edge from one task to the agent with minimal execution time is always included). In general, an ST resulting from this heuristic is likely to encode any solution that assigns tasks to one of the first nth agents in the order of its processing time, where n is a small number.

As an example, Fig. 4a shows a trace of the weights computed at each iteration of the `computeSTApproximation` procedure and Fig. 4b the edges included in the ST at the end of this procedure when using the *best-case* heuristic. Since the value returned by this heuristic does not depend on the set of edges already included in the ST, the weight for an edge does not change between iterations. At the initialization step (iteration 0), edges (a_1, t_1) and (a_4, t_5) are added to the ST, as a result of preprocessing tasks with a single neighbour. During the first iteration, the edge between a_2 and t_2 has a starting minimal weight (20) and it is added to the ST. The second edge to be added is a_4 to t_4, then a_4 to t_3, a_1 to t_2, a_1 to t_3, and finally a_3 to t_4. Observe that all edges from one task to the agent with minimal execution time, namely (t_1, a_1), (t_2, a_2), (t_3, a_4) and (t_4, a_4), have been added. In the same way, all the edges form one task to the agent with second minimal execution time have also been added (i.e. (t_2, a_1), (t_3, a_1) and (t_4, a_3)). Thus, for this particular example, the ST encodes any solution that assigns tasks either to the first or the second agent with minimal execution time for that task and hence, it encodes one of the optimal solutions. Thus, for example, the set $\tilde{T}_2$ (related to agent a_2) is composed of t_2 (for which a_2 is the agent with second minimal execution time) but not of t_3 (for which it is the third) since this edge is pruned in the ST.

The worst-case heuristic. The worst-case heuristic aims to improve upon the greedy-like behaviour of the best-case to consider the worst possible allocation an agent could get. Thus, at each iteration, this heuristic considers not only the processing time of a task by an agent but also the time taken for tasks that will be allocated to it in the current iteration.

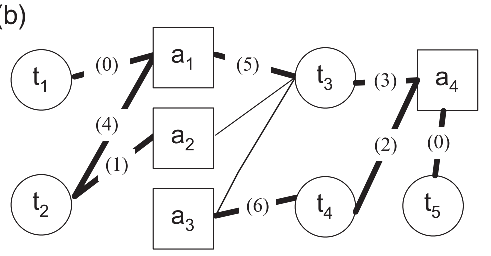
Formally, let m^s be the best allocation for tasks added up to iteration s, namely the optimal allocation of tasks already included up to iteration s. m^s is computed at each iteration by running min–max over the acyclic TDN that form the allocations included up to iteration s. Then, the *worst-case* heuristics

(a)

	0	1	2	3	4	5	6
(a_1, t_1)	**10**	-	-	-	-	-	-
(a_1, t_2)	30	30	30	-	**30**	-	-
(a_1, t_3)	50	50	50	50	50	**50**	-
(a_2, t_2)	20	**20**	-	-	-	-	-
(a_2, t_3)	60	60	60	60	60	60	-
(a_3, t_3)	80	80	80	80	80	80	80
(a_3, t_4)	70	70	70	70	70	70	**70**
(a_4, t_3)	30	30	30	**30**	-	-	-
(a_4, t_4)	20	20	**20**	-	-	-	-
(a_4, t_5)	**60**	-	-	-	-	-	-

Weights computed for not added edges at each iteration. The weight of the edge added in a given iteration appears boldfaced.

(b)



ST-TDN. Boldfaced edges are those included in the ST, each labeled with the number of the iteration at which it has been included.

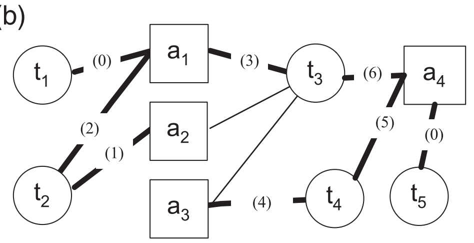
FIGURE 4. Example of (**a**) the weights of not added edges and (**b**) the acyclic TDN computed by ST-DTDA using the *best-case* heuristic over the TDN depicted in Fig. 1.

(a)

	0	1	2	3	4	5	6
(a_1, t_1)	**10**	-	-	-	-	-	-
(a_1, t_2)	40	40	**40**	-	-	-	-
(a_1, t_3)	60	60	60	**60**	-	-	-
(a_2, t_2)	20	**20**	-	-	-	-	-
(a_2, t_3)	60	60	80	80	-	-	-
(a_3, t_3)	80	80	80	80	80	150	150
(a_3, t_4)	70	70	70	70	**70**	-	-
(a_4, t_3)	90	90	90	90	90	90	**90**
(a_4, t_4)	80	80	80	80	80	**80**	-
(a_4, t_5)	**60**	-	-	-	-	-	-

Weights computed for not added edges at each iteration. The weight of the edge added in a given iteration appears boldfaced.

(b)



ST-TDN. Boldfaced edges are those included in the ST, each labeled with the number of the iteration at which it has been included.

FIGURE 5. Example of (**a**) the weights of not added edges and (**b**) the acyclic TDN computed by ST-DTDA using the *worst-case* heuristic over the TDN depicted in Fig. 1.

weights the edge between a_i and t_k at an iteration s with the time required by a_i to execute task t_k plus the total time it would take agent a_i to compute all tasks allocated to it in the current optimal allocation, $m^s(a_i)$. Formally,

$$h_i(t_k) = \chi_i(t_k) + \sum_{\substack{t_l \in m^s(a_i) \\ l \neq k}} \chi_i(t_l). \tag{5}$$

In case of ties, we consider the edge with the highest execution time to be the heaviest. We refer to this heuristic as *worst-case* because the weight w_{ik} it computes corresponds to the runtime incurred at agent a_i when allocated t_k *and* all other tasks assigned in the current ST.

As an example, Fig. 5a shows a trace of the weights computed at each iteration of the `computeSTApproximation` procedure and Fig. 5b the edges included in the ST at the end of this procedure when using the *worst-case* heuristic. At the initialization step, all weights are computed as in the *best-case* heuristic with the exception of those connected to agent a_1 and a_4 that need to add the cost of edges already included as a result of preprocessing tasks with a single neighbour. Then, the first edge to be added is (a_2, t_2) because it is the one with starting minimal weight (20) and the weights not added edges will be updated (the weight of the edge between a_2 to t_3 is updated to 80 for the next iteration). The next edge to be added is a_1 to t_3, then a_3 to t_4, a_4 to t_4 and finally a_4 to t_3. Note that, for this particular example, *worst-case* adds the same edges than *best-case* but in different priority order. Hence, the edge between a_4 and t_4 is added at the fifth iteration instead of the second since this edge is much less attractive when you consider that agent a_4 is already assigned t_5.

The min-regret heuristic. An obvious problem with the *worst-case* heuristic is that it often postpones the decision regarding the most crucial agent-task allocations to the last iterations where there is no much freedom of action. The *min-regret* tries to circumvent this problem by prioritizing allocations based on the cost difference of allocating a task to its best agent or to its second best agent. In other words, *min-regret* aims to minimize the *regret* when omitting edges. Here regret is defined as the difference in terms of cost that thee system would incur if t_k is not allocated to a_i, but rather allocated to another agent other than a_i. The cost of allocating a task t_k to an agent a_i is computed as in the worst-case heuristic: the agent will process task t_k and all other tasks to which it is assigned in the best current allocation in the ST. Formally,

$$h_i(t_k) = \chi_i(t_k) + \sum_{\substack{t_l \in m^s(a_i) \\ l \neq k}} \chi_i(t_l) - \min_{\substack{j \in D_k \\ j \neq i}} \left(\chi_j(t_k) + \sum_{\substack{t_l \in m^s(a_j) \\ l \neq k}} \chi_j(t_l) \right), \tag{6}$$

where m^s is the best allocation for tasks added up to iteration s.

In cases where two edge weights are equal, we break ties in the same way as we do for the *worst-case* heuristic.

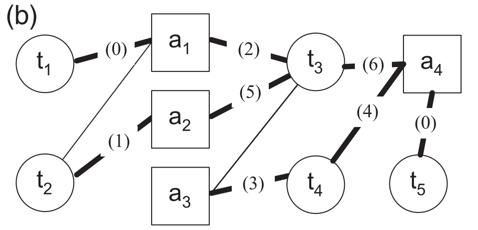
As an example, Fig. 6 shows the resulting ST-TDN and a trace of the weights computed for possible not added edges at each iteration of the `computeSTApproximation` procedure when using the *min-regret* heuristic. At the initialization step (iteration 0), all weights are computed and edges (a_1,t_1) and (a_4, t_5) are included as a result of preprocessing tasks with a single neighbour. Thus, in the first step, the weight between agent a_2 and task t_2 is -20 since 30 is the execution time of task t_2 at a_2 and the alternative best allocation is t_2 to a_1 which has a cost of 30. At this step, this weight is the lowest and hence the edge is included in the ST. In the following steps, the weights are recomputed being the next edge to be added a_1 to t_3, then a_3 to t_4, then a_4 to t_4, then a_2 to t_3 and finally a_4 to t_3. Note that unlike the *best-case* and *worst-case* heuristic, *min-regret* does not prioritizes the inclusion of the edge between a_1 and t_2 which is not finally added to the ST. This is because the edges regarding task t_3 have higher regret and t_4 have higher regret than not the edge between a_1 and t_2 (there is higher difference on the solution impact between their possible allocations).

Now, independently of the heuristic function used, once we have found the ST, we can describe the acyclic TDN as follows: for each agent a_i, we have $\tilde{T}_i$ as the set of tasks to which agent a_i is connected in the ST. Given this, it is now possible to find

(a)

	0	1	2	3	4	5	6
(a_1, t_1)	∞	-	-	-	-	-	-
(a_1, t_2)	20	20	20	70	70	70	-
(a_1, t_3)	0	0	**-20**	-	-	-	-
(a_2, t_2)	-20	**-20**	-	-	-	-	-
(a_2, t_3)	0	0	20	20	20	**20**	-
(a_3, t_3)	20	20	20	20	90	90	90
(a_3, t_4)	-10	-10	-10	**-10**	-	-	-
(a_4, t_3)	30	30	30	30	30	30	**30**
(a_4, t_4)	10	10	10	10	**10**	-	-
(a_4, t_5)	$-\infty$	-	-	-	-	-	-

Weights computed for not added edges at each iteration. The weight of the edge added in a given iteration appears boldfaced.



ST-TDN. Boldfaced edges are those included in the ST, each labeled with the number of the iteration at which it has been included.

FIGURE 6. Example of (**a**) the weights of not added edges and (**b**) the acyclic TDN computed by ST-DTDA using the *min-regret* heuristic over the TDN depicted in Fig. 1.

a mapping m^* that minimizes the makespan incurred by tasks in $\tilde{T}_i$ for any agent a_i.

4.2. Solving the acyclic TDN with min–max

In this step, agents run min–max on the JT representation of the acyclic TDN computed in step 1. As discussed in Section 3.2, any acyclic TDN maps directly to a distributed JT and hence ST-DTDA agents only need to execute steps 1–4 to encode the acyclic TDN into a JT.

In more detail, Algorithm 4 outlines the `solveST-Approximation` procedure that each ST-DTDA agent executes during this step. As input, each agent receives the set of tasks that it is connected to in the ST ($\tilde{T}_i$) and the set of neighbours that a_i shares some tasks (N_i) with. Each agent starts by exchanging this set of tasks with its neighbours (lines 1–4). Once an agent has received the set of tasks from all its neighbours, it proceeds to create the corresponding JT. First, an agent creates one variable x_k for task t_k in $\tilde{T}_i$ whose domain contains the set of indexes of all agents that can execute t_k in the ST (lines 5–7). Secondly, the agent sets its clique and its set of local variables to the set of variables representing the tasks in $\tilde{T}_i$ (line 8). Thirdly, the agent initializes its potential to encode its cost function incurred by tasks $\tilde{T}_i$ (line 9). Finally, the agent creates a separator for each neighbour with which it shares some task in the ST, containing the set of shared tasks (lines 10–12). Given its clique, potential and separators, each agent proceeds to run min–max as described in Section 2.2. As the outcome, each agent knows the optimal configuration of the ST for its local variables ($\tilde{\mathbf{X}}_\mathbf{i}^*$) and its value ($\tilde{V}^*$). The set of allocated tasks to the agent, $\{k|\tilde{\mathbf{x}}_\mathbf{k} \in \tilde{\mathbf{X}}_\mathbf{i} \cap \tilde{\mathbf{x}}_\mathbf{k} = i\}$, is computed from the optimal configuration (line 14).

Algorithm 4 `solveSTApproximation()` at agent a_i

Require: The set of tasks (T_i) and neighbours (N_i)

```
1: for all $a_j \in N_i$ do
2:     Send $\tilde{T}_i$ to $a_j$
3: end for
4: Wait until receiving $\tilde{T}_j$ from all $a_j \in N_i$
5: for all $t_k \in \tilde{T}_i$ do
6:     Create $\tilde{x}_k$ with $\tilde{D}_k = \{j|a_j \in N_i \cap t_k \in \tilde{T}_j\} \cup \{i\}$
7: end for
8: $\tilde{X}_i = \{\tilde{x}_k|t_k \in \tilde{T}_i\}$
9: $\psi_i(\tilde{X}_i) = \sum_{t_k \in \tilde{T}_i, \tilde{x}_k = i} \chi_i(t_k)$
10: for all $a_j \in N_i|\tilde{T}_j \cap \tilde{T}_i \neq \emptyset$ do
11:     $\tilde{S}_{ij} = \{\tilde{x}_k|t_k \in \tilde{T}_i \cap t_k \in \tilde{T}_j\}$
12: end for
13: $\langle \tilde{\mathbf{X}}_\mathbf{i}, \tilde{V}^* \rangle =$ Min-max($\tilde{X}_i$, $\tilde{S}_i$, $\psi_i(\tilde{X}_i)$)
14: return $\langle \{k|\tilde{\mathbf{x}}_\mathbf{k} \in \tilde{\mathbf{X}}_\mathbf{i} \cap \tilde{\mathbf{x}}_\mathbf{k} = i\}, \tilde{V}^*, \tilde{X}_i, \tilde{S}_i \rangle$
```

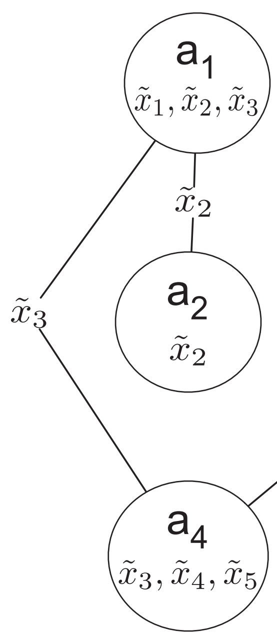

a1
$\tilde{x}_1, \tilde{x}_2, \tilde{x}_3$
$\tilde{x}_2$
$\tilde{x}_3$
a2
$\tilde{x}_2$
a3
$\tilde{x}_4$
$\tilde{x}_4$
a4
$\tilde{x}_3, \tilde{x}_4, \tilde{x}_5$

FIGURE 7. The JT formulation for the acyclic TDN resulting from the best-case heuristic shown in Fig. 4a. Large circles are cliques, with the elements of the cliques listed. Edges are labelled with common variables between cliques.

As an example, Fig. 7 shows the JT computed by the agents during this step for the ST found by the *best-case* heuristic in Fig. 4a. The domain of $\tilde{x}_3$ is restricted to 1, 4 not including 2, 3 since these edges were pruned in the ST. Similarly, clique $\tilde{X}_2$ only contains $\tilde{x}_2$ since the possible allocation of t_3 to a_2 is not considered in the ST. After running min–max on this JT, it will find a solution where $x_1 = 1$, $x_2 = 2$, $x_3 = 1$, $x_4 = 3$ and $x_5 = 4$, and the makespan is $\max(10 + 50, 20, 70, 60) = 70$. Thus, the solution value found in this step, denoted by $\tilde{V}^*$, is 70.

We can assess the complexity of this step from the complexity of running min–max over the JT encoding the acyclic TDN. As discussed at the end of Section 3.2, in acyclic TDNs the computation of the solution grows exponentially in the number of tasks an agent is connected to in the TDN. Since the TDNs that result from Step 1 have the maximum number of tasks that an agent can be connected to bounded by Δ, min–max runs in polynomial time for fixed Δ.

4.3. Computing the per-instance approximation ratio

The final step of ST-DTDA involves computing an approximation ratio, which can always be used to bound the quality of the solution found in step 2. Having an approximation ratio ρ means that $\tilde{V}^*/V^* \leq \rho$ (e.g. the value of the solution found by ST-DTDA will not be more than a factor ρ the value of the optimal solution).

Each agent estimates the approximation ratio ρ of $\tilde{V}^*$ as follows:

$$\rho = \frac{\tilde{V}^*}{\max(\tilde{V}^* - \varepsilon, V^{\mathrm{LB}})}, \tag{7}$$

where ε is an absolute bound on the error of V^* ($\tilde{V}^* - \varepsilon \leq V^*$) and V^{LB} is a lower bound on V^* ($V^{\mathrm{LB}} \leq V^*$). Note that $V^* \geq \max(\tilde{V}^* - \varepsilon, V^{\mathrm{LB}})$ and then the approximation ratio from Equation (7) comes simply from rearranging this expression. We detail the computation of ε and V^{LB} next.

Computing ε. Let $T^M \subseteq T$ be a set that contains tasks whose domains have been *modified* in the ST (i.e. at least one agent-task allocation has been omitted for each $t_k \in T^M$).

As proved in Appendix 2, we can define ε as the value of the minimum makespan in the approximate problem composed only of tasks in T^M. Formally,

$$\varepsilon = \min_{\tilde{X}} \max_{a_i \in A} \sum_{\substack{t_k \in T^M \\ \tilde{x}_k = i}} \chi_i(t_k). \tag{8}$$

To compute ε, ST-DTDA agents run min–max over the JT corresponding to the vertex-induced subgraph of the TDN induced by T^M and the set of agents nodes.

Specifically, Algorithm 5 outlines the procedure that each ST-DTDA agent executes during this step. As input, each agent receives the set of variables and separators of the agent in the JT ($\tilde{X}_i$,$\tilde{S}_i$), the value of the ST-DTDA solution ($\tilde{V}^*$) and a lower bound (V^{LB}). Then, each agent proceeds to compute the JT corresponding to the vertex-induced subgraph of the TDN induced by T^M. First, each agent restricts the set of variables, and its clique, to those corresponding to tasks whose domains have been *modified* in the ST (line 1). Secondly, each agent computes its potential as its cost function but considering only the task execution times of the modified tasks (line 2). Finally, each agent recomputes its separators (lines 3–5). Given its clique, potential and separators, each agent proceeds to run min–max. As the end of this procedure, each agent returns the approximation ratio computed as in Equation (7).

Algorithm 5 `computeApproximationRatio()` at agent a_i

Require: The set of variables ($\tilde{X}_i$), the set of separators ($\tilde{S}_i$), the value of the ST-DTDA solution ($\tilde{V}^*$) and a lower bound on the optimal solution (V^{LB})

```
1: $\tilde{C}_i^M = \tilde{X}_i^M = \{\tilde{x}_k \in \tilde{X}_i | D_k \neq \tilde{D}_k\}$
2: $\psi_i(\tilde{X}_i^M) = \sum_{\tilde{x}_k \in \tilde{X}_i^M, \tilde{x}_k = i} \chi_i(t_k)$
3: for all $s_{ij} \in \tilde{S}_i$ do
4:     $\tilde{S}_{ij}^M = \tilde{S}_{ij} \cap \tilde{X}_i^M$
5: end for
6: $\langle \tilde{\mathbf{C}}_\mathbf{i}^\mathbf{M}, \varepsilon \rangle =$ Min-max($\tilde{C}_i^M$, $\tilde{S}_i$, $\psi_i(\tilde{X}_i^M)$)
7: return $\tilde{V}^*/(\max(\tilde{V}^* - \epsilon, V^{LB}))$
```

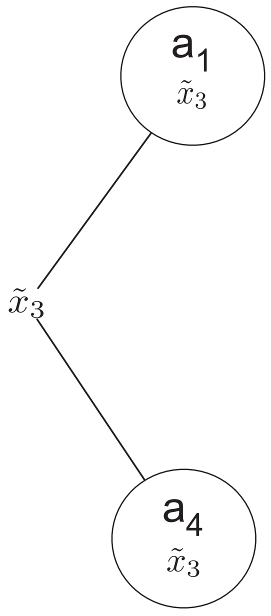


FIGURE 8. The JT whose solution gives a worst-case bound on the quality of the solution returned by ST-DTDA in the JT in Fig. 7. Edges are labelled with common variables between cliques.

Figure 8 shows the JT the agents need to solve to obtain the worst-case bound for the solution produced by ST-DTDA over the JT specified in Fig. 7. The only edges removed in the ST found by the *best-case* in Fig. 4a are (a_2, t_3) and (a_3, t_3). Hence, in this case, $T^M = \{t_3\}$. To compute the bound, min–max is used to find the optimal value of $\tilde{x}_3$, which, in this case, is a_4 (e.g. $\arg_{a_1,a_4} \min(50, 30) = a_4$). Note that, to compute the absolute error bound, ST-DTDA agents are only interested in the value of the best configuration, not in the best configuration itself. Thus, $\varepsilon = 30$ in our example, which means that any solution found in step 3 will be at most 30 away from the optimal solution value.

Computing V^{LB}. Now, there can be cases where ε is equal to $\tilde{V}^*$. In particular, in cases where at least one edge per task needs to be pruned to create the ST. In such cases, there is no guarantee on the quality of the solution returned by agents running ST-DTDA because the value of the absolute error bound is the same than the value of the solution. To provide a guarantee in these cases, we combine this bound with a simple lower bound on the optimal solution, V^{LB}. In particular, we define V^{LB} as the time taken for the fastest agent to perform the longest task. Formally,

$$V^{\mathrm{LB}} = \max_{t_k \in T} \min_{a_i \in A} \chi_i(t_k). \tag{9}$$

Thus, for the TDN in Fig. 1, V^{LB} is computed as

$$\begin{aligned} V^{\mathrm{LB}} &= \max\Big(\min_{i\in\{1\}}(\chi_i(t_1)), \min_{i\in\{1,2\}}(\chi_i(t_2)), \min_{i\in\{1,2,3,4\}}(\chi_i(t_3)), \\ &\qquad \min_{i\in\{3,4\}}(\chi_i(t_4)), \min_{i\in\{4\}}(\chi_i(t_5))\Big) \\ &= \max(\min(10), \min(20, 30), \min(50, 60, 80, 30), \\ &\qquad \min(70, 20), \min(60)) = 60. \end{aligned}$$

Finally, given $\varepsilon = 30$, $\tilde{V}^* = 70$ and $V^{\mathrm{LB}} = 60$, we compute the approximation ratio, according to Equation (7), as follows:

$$\rho = \frac{70}{\max(70 - 30, 60)} = \frac{70}{60} = 1.16.$$

Thus, in this case, the *best-case* heuristic achieves an approximation ratio of 1.16, despite the fact that the approximate solution is equal to the optimal.

We can assess the complexity of this step from the complexity of running min–max over the JT encoding the acyclic TDN restricted to tasks whose domains have been *modified* in the ST.

Thus, clearly the complexity of solving this JT is bounded above by the complexity of solving the JT for the whole acyclic TDN in Step 2.

Having completed our definition of ST-DTDA, we next proceed to evaluate it on a number of scenarios and compare it against a baseline algorithm in order to identify its strengths and weakness on different types of TDNs since its performance is clearly dependent on the graph structure (node degrees in particular) of the TDNs.

5. EMPIRICAL EVALUATION

In this section, we present an evaluation of the ST-DTDA algorithm (using our heuristics, namely *best-case*, *worst-case* and *min-regret* as given in Section 4.1). Moreover, we benchmark ST-DTDA against distributed stochastic algorithm (DSA),[18] a decentralized hill-climbing algorithm we propose as a baseline approach to solve Agent-Based $R\|C_{\max}$, and against IBM ILOG CPLEX,[19] a centralized mixed integer programming solver.

In what follows, first, we explain the details of our experimental setup in Section 5.1. Then, we describe and analyse our empirical results in Section 5.2.

5.1. Experimental setup

To determine the average-case performance of ST-DTDA and DSA, we evaluate them on random TDNs. The process of generating a TDN is divided in two steps. First, we generate the structure of the TDN, namely the set of agent/task nodes and the set of edges that stand for possible agent-task allocations. Secondly, we generate the task execution time for each edge between a task and an agent in the TDN.

As discussed in [7], the task-to-agent ratio and the density are the two key *structural* parameters that are most likely to impact performance. Hence, to control these parameters, the TDN is generated by specifying: the number of agents ($|A|$), the number of tasks ($|T|$) and the average number of tasks connected to an agent ($|T_a|$). Thus, the first two parameters fix the task-to-agent ratio ($|T|/|A|$) of the TDN whereas the latter fixes its density (or, equivalently, its sparsity).[20] Then, a random tree can be generated with vertices composed of agents and tasks (this guarantees that the corresponding TDN is connected) where for each edge to be added, we randomly select one task and one agent not directly connected (i.e. there is no edge between that task and the agent) and add an edge between them. We then check that all tasks can potentially be executed by at least two agents. Otherwise, the instance is discarded. This check avoids the generation of trivial decision variables; a task with degree 1 generates a variable that can only take one value, that is, the only agent that can execute this task.

Once the structure of a TDN is generated, we proceed to generate the task execution time for each edge between a task and an agent. Now, as stated in [22], the solution quality of algorithms may depend on the method used to generate task execution times. To take this into account, we consider, for each instance, three different methods to generate the execution time of task t_k at agent a_i ($\psi_i(t_k)$):

(i) *Uncorrelated:* $\psi_i(t_k)$ is drawn from a truncated normal distribution $N[100, 10]$ bounded below[21] by 10;
(ii) *Agent correlated:* times are correlated with the agents' computational delay. Thus, $\psi_i(t_k)$ is drawn from a

[18]A complete description of DSA algorithm is given in Appendix 1.

[19]https://www.ibm.com/software/commerce/optimization/cplex-optimizer/.

[20]From the number of agents and the average number of tasks connected to an agent, we can assess the number of edges included in the network ($|E| = |A| \cdot |T_a|$).

[21]We use a bounded distribution to ensure that task execution times are never negative nor close to zero.

truncated normal distribution $N[\alpha_i, 10]$ bounded below by 10 where α_i, the agent's computational delay, is drawn from the uniform distribution $U[50, 100]$;

(iii) *Agent/task correlated:* times are correlated with the agents' speed and the workload of the corresponding task. Thus, $\psi_i(t_k)$ is drawn from a truncated normal distribution $N[\beta_k + \alpha_i, 10]$ bounded below by 10, where β_k, α_i stand, respectively, for the task's workload and the agent's delay. Both β_k,α_i are drawn from the uniform distribution $U[50, 100]$.

Below we summarize the parameters used in our experiments:

(i) number of tasks: 100;
(ii) number of agents: 40, 60, 80;
(iii) number of tasks per agent: 5, 10, 15;
(iv) task execution times: uncorrelated, agent correlated and agent/task correlated;
(v) number of problems tested per combination: 100.

For our benchmarks, we set the activation probability of DSA to 0.7 (a value that is reported to work well in [23])[22] and the number of iterations to 10 000 ($I_{\max} = 10000$),[23] while for CPLEX we use the version 12.4 and stop the computation after 900 s. Finally, for ST-DTDA we set the maximum number of tasks per agent to 10 ($\Delta = 10$). All the experiments were performed on a 3.2 GHz Intel Core i5 with 16 GB ram.

5.2. Results

Tables 1–3 list our results for each task execution time distribution. Each table contains an entry for each scenario, namely each combination of tasks ($|T|$), agents ($|A|$) and average number of tasks per agent ($|T_a|$). The fourth column, headed by $m^* \setminus \check{m}/|T|$, reports the fraction of tasks in the optimal allocation, m^*, that differ from the greedy allocation, $\check{m}$, that assigns each task to the agent with minimal processing time. For each metric, we report the mean over 100 instances plus/minus the 95% confidence interval.

For CPLEX, we report the CPU time (in seconds) to capture the hardness of the problem when solved by a centralized algorithm. The number inside the parentheses is the number of problems that did not terminate within the time limit (CPU time averages are computed over the solved problems only). We observe that time taken by CPLEX to solve a problem can vary significantly across execution time distributions. Thus, CPLEX can solve most of the instances with uncorrelated times or times correlated with agents within a few seconds. In contrast, the same TDN structures with times correlated to agents and tasks tend to be computationally intensive. For instance, when ($|T| = 100, |A| = 60, |T_a| = 15$) more than one half of the instances (58) were not solved within the time limit (see Table 3).

For the rest of algorithms (ST-DTDA and DSA), they are extremely fast as expected and therefore we do not report computational times for them. In fact, all problems were solved within a few seconds and hence with much lower CPU requirements than CPLEX in many agent/task correlated instances. Instead, we focus our attention on the quality of their solutions. For each algorithm, we assess the quality of the solution it generates by dividing the optimal makespan found by CPLEX (V^*) by the makespan of the solution found by the algorithm ($\tilde{V}^*$).[24] For ST-DTDA, Tables 1–3 also show the per-instance approximation ratio obtained in step 3 of ST-DTDA execution.

In general, we observe that the quality of the solutions found by ST-DTDA and DSA algorithms follow the same trend as the CPLEX solving times: instances with execution times agent-task correlated times lead to lower quality solutions than instances with uncorrelated or agent correlated times. For example, for uncorrelated times and the configuration $(|T|, |A|, |T_a|) = (100, 40, 15)$ (see Table 1), the average solution quality of ST-DTDA with *best-case* is 0.57. However, for the same configuration but with agent-task correlated times, the average solution quality decreases to 0.35 (see Table 3). A similar behaviour is observed for *worst-case*, *min regret* heuristics and DSA. To explain this result, we use the measure of the fraction of tasks that were not allocated in a greedy fashion in the optimal solution (i.e. the average fraction of tasks that were not assigned to the agent with minimal execution time), that is, $(m^* \setminus \check{m})/|T|$. We observe that the fraction of tasks not allocated in a greedy fashion is significantly higher in agent-task correlated times than in agent-correlated and uncorrelated times (e.g. compare the 0.55 of uncorrelated with the 0.79 of agent-task correlated for configuration ($|T| = 100$, $|A| = 80$, $|T_a| = 15$)). Hence, the first problems are more difficult than the latter (i.e. the higher the fraction of tasks allocations that differ from the greedy allocation, the lower the quality of the solutions of the algorithms).

On the other hand, although the number of tasks per agent affects the density of the TDN, this does not seem to have a significant impact on the solution quality of the algorithms for the parameters explored. Nevertheless, the quality of the solutions does vary significantly from one algorithm to the other as we analyse next.

First, we observe that ST-DTDA with *best-case* exhibits poor performance, especially for agent-task correlated times for which the average quality of the solutions ranges from 0.22 to 0.44 (see Table 3). This can be explained by the greedy behaviour of the *best-case* heuristic that assigns each task to the agents with the lowest processing times without taking into

[22]We run DSA with different activation probabilities (i.e. 0.3 and 0.5) but results obtained were similar than the ones presented here for 0.7.

[23]DSA always run this number of iterations, independently if its solution stabilize before.

[24]For problems in which CPLEX reaches the maximum time without finding the optimal solution, the best solution found up to this time limit is used.

TABLE 1. Performance of ST-DTDA and DSA algorithms for *uncorrelated* execution times.

Problem characteristics				CPLEX	ST-DTDA best-case	ST-DTDA worst-case	ST-DTDA min-regret	DSA
$\|T\|$	$\|A\|$	$\|T_a\|$	$\frac{m^* \setminus \check{m}}{\|T\|}$	Time (s)	Sol. Quality / Approx. Ratio	Sol. Quality / Approx. Ratio	Sol. Quality / Approx. Ratio	Sol. quality
100	40	5	0.40 ± 0.01	<1 (0)	0.58 ± 0.02	0.83 ± 0.02	0.95 ± 0.03	0.72 ± 0.04
					4.45 ± 0.13	3.05 ± 0.06	2.66 ± 0.02	
		10	0.44 ± 0.01	<1 (0)	0.53 ± 0.09	0.85 ± 0.08	0.94 ± 0.01	0.76 ± 0.04
					4.85 ± 0.16	3.00 ± 0.06	2.63 ± 0.02	
		15	0.46 ± 0.01	12 ± 10 (2)	0.57 ± 0.02	0.89 ± 0.01	0.93 ± 0.00	0.88 ± 0.02
					4.49 ± 0.14	2.85 ± 0.05	2.70 ± 0.02	
	60	5	0.44 ± 0.01	<1 (0)	0.54 ± 0.02	0.79 ± 0.02	0.93 ± 0.03	0.89 ± 0.01
					3.29 ± 0.12	2.24 ± 0.06	1.85 ± 0.01	
		10	0.46 ± 0.04	2 ± 1 (0)	0.50 ± 0.09	0.81 ± 0.11	0.94 ± 0.01	0.89 ± 0.01
					3.56 ± 0.13	2.15 ± 0.06	1.85 ± 0.01	
		15	0.46 ± 0.01	3 ± 0.46 (0)	0.52 ± 0.02	0.88 ± 0.01	0.94 ± 0.00	0.90 ± 0.01
					3.47 ± 0.12	2.00 ± 0.04	1.86 ± 0.01	
	80	5	0.53 ± 0.01	<1 (0)	0.59 ± 0.02	0.89 ± 0.01	0.93 ± 0.03	0.90 ± 0.00
					2.86 ± 0.11	1.87 ± 0.03	1.76 ± 0.01	
		10	0.57 ± 0.05	14 ± 35 (0)	0.59 ± 0.09	0.90 ± 0.04	0.94 ± 0.01	0.90 ± 0.00
					2.92 ± 0.10	1.87 ± 0.03	1.77 ± 0.01	
		15	0.55 ± 0.01	2 ± 0.15 (0)	0.60 ± 0.02	0.93 ± 0.01	0.94 ± 0.00	0.89 ± 0.00
					2.96 ± 0.12	1.83 ± 0.01	1.81 ± 0.01	

The number between parentheses in the CPLEX column is the number of instances not completed within 900 s.

TABLE 2. Performance of ST-DTDA and DSA algorithms for *agent correlated* execution times.

Problem characteristics				CPLEX	ST-DTDA best-case	ST-DTDA worst-case	ST-DTDA min-regret	DSA
$\|T\|$	$\|A\|$	$\|T_a\|$	$\frac{m^* \setminus \check{m}}{\|T\|}$	Time (s)	Sol. Quality / Approx. Ratio	Sol. Quality / Approx. Ratio	Sol. Quality / Approx. Ratio	Sol. quality
100	40	5	0.42 ± 0.01	<1 (0)	0.58 ± 0.02	0.80 ± 0.01	0.93 ± 0.01	0.84 ± 0.01
					3.30 ± 0.12	2.36 ± 0.06	2.02 ± 0.03	
		10	0.49 ± 0.01	<1 (0)	0.54 ± 0.01	0.81 ± 0.01	0.92 ± 0.00	0.85 ± 0.01
					3.53 ± 0.13	2.32 ± 0.05	2.03 ± 0.03	
		15	0.58 ± 0.01	<1 (0)	0.52 ± 0.01	0.85 ± 0.01	0.89 ± 0.00	0.85 ± 0.02
					3.90 ± 0.12	2.36 ± 0.06	2.23 ± 0.04	
	60	5	0.49 ± 0.01	<1 (0)	0.56 ± 0.02	0.76 ± 0.01	0.89 ± 0.01	0.80 ± 0.01
					2.58 ± 0.01	1.85 ± 0.04	1.57 ± 0.03	
		10	0.58 ± 0.01	<1 (0)	0.50 ± 0.01	0.78 ± 0.02	0.90 ± 0.00	0.82 ± 0.01
					2.88 ± 0.11	1.85 ± 0.05	1.58 ± 0.03	
		15	0.64 ± 0.01	<1 (0)	0.48 ± 0.01	0.83 ± 0.01	0.87 ± 0.01	0.85 ± 0.01
					3.50 ± 0.12	1.99 ± 0.04	1.88 ± 0.04	
	80	5	0.58 ± 0.01	<1 (0)	0.52 ± 0.02	0.76 ± 0.02	0.90 ± 0.01	0.79 ± 0.01
					2.18 ± 0.10	1.50 ± 0.05	1.23 ± 0.02	
		10	0.65 ± 0.01	<1 (0)	0.46 ± 0.01	0.80 ± 0.02	0.90 ± 0.00	0.82 ± 0.01
					2.75 ± 0.10	1.56 ± 0.05	1.39 ± 0.03	
		15	0.69 ± 0.01	<1 (0)	0.46 ± 0.01	0.87 ± 0.01	0.87 ± 0.01	0.83 ± 0.01
					3.21 ± 0.14	1.66 ± 0.04	1.64 ± 0.03	

The number between parentheses in the CPLEX column is the number of instances not completed within 900 s.

TABLE 3. Performance of ST-DTDA and DSA algorithms for *agent/task correlated* execution times.

Problem characteristics				CPLEX	ST-DTDA best-case	ST-DTDA worst-case	ST-DTDA min-regret	DSA
$\vert T\vert$	$\vert A\vert$	$\vert T_a\vert$	$\frac{m^* \setminus \check{m}}{\vert T\vert}$	Time (s)	Sol. Quality / Approx. Ratio	Sol. Quality / Approx. Ratio	Sol. Quality / Approx. Ratio	Sol. quality
100	40	5	0.55 ± 0.01	11 ± 12 (1)	0.44 ± 0.01	0.74 ± 0.01	0.88 ± 0.01	0.67 ± 0.03
					4.61 ± 0.12	2.74 ± 0.05	2.29 ± 0.02	
		10	0.60 ± 0.01	152 ± 39 (41)	0.39 ± 0.01	0.74 ± 0.01	0.87 ± 0.00	0.65 ± 0.03
					5.04 ± 0.12	2.65 ± 0.04	2.29 ± 0.03	
		15	0.68 ± 0.01	205 ± 31 (20)	0.35 ± 0.01	0.75 ± 0.01	0.85 ± 0.00	0.73 ± 0.03
					5.77 ± 0.12	2.69 ± 0.05	2.37 ± 0.03	
	60	5	0.62 ± 0.01	2 ± 0.60 (0)	0.38 ± 0.10	0.73 ± 0.01	0.86 ± 0.01	0.81 ± 0.01
					4.04 ± 0.14	2.05 ± 0.04	1.74 ± 0.02	
		10	0.70 ± 0.01	72 ± 29 (8)	0.31 ± 0.01	0.73 ± 0.01	0.87 ± 0.01	0.80 ± 0.01
					4.91 ± 0.14	2.05 ± 0.04	1.80 ± 0.02	
		15	0.78 ± 0.01	232 ± 54 (58)	0.27 ± 0.01	0.74 ± 0.01	0.87 ± 0.01	0.79 ± 0.01
					5.68 ± 0.14	2.07 ± 0.03	1.77 ± 0.02	
	80	5	0.66 ± 0.01	9 ± 14 (1)	0.32 ± 0.01	0.69 ± 0.01	0.81 ± 0.01	0.76 ± 0.01
					3.80 ± 0.14	1.76 ± 0.04	1.48 ± 0.02	
		10	0.74 ± 0.01	7 ± 4 (2)	0.25 ± 0.01	0.69 ± 0.01	0.81 ± 0.01	0.76 ± 0.01
					4.87 ± 0.15	1.77 ± 0.03	1.53 ± 0.02	
		15	0.79 ± 0.01	12 ± 6 (2)	0.22 ± 0.01	0.71 ± 0.01	0.78 ± 0.01	0.76 ± 0.01
					5.65 ± 0.15	1.75 ± 0.03	1.59 ± 0.02	

The number between parentheses in the CPLEX column is the number of instances not completed within 900 s.

account the fact that these agents may already be overloaded. In contrast, as expected, ST-DTDA with *worst-case* and *min-regret* achieves significantly higher quality (around 0.69–0.75 and 0.78–0.88, respectively, with agent-task correlated times, see Table 3). This is explained by the fact that the *worst-case* and *min-regret* heuristics consider the current agents' workload as defined by the best allocation of the existing tasks connected to the agents. Thus, these heuristics can better minimize the possibility of overloading some agents and therefore creating bottlenecks. Secondly, we find that the *min-regret* and *worst-case* heuristics have very similar performances but the quality of the solutions of *min-regret* is always higher (around 5–15% higher). This is because the *min-regret* heuristic prioritized the tasks with higher regret. Finally, DSA returns competitive solutions, sometimes better than ST-DTDA with the *worst-case* heuristic on average. Nevertheless, DSA always achieves worse average solutions than ST-DTDA with *min-regret*.

Finally, regarding the per-instance approximation ratio we observe that in general the approximation ratio returned by ST-DTDA is not very accurate (i.e. there is a significant gap between the real approximation ratio of the solution found and the one reported by ST-DTDA). For instance, observe in Table 2, that the average approximation ratio returned by ST-DTDA with *min-regret* for configuration ($|T| = 100$, $|A| = 40$, $|T_a| = 5$) is 2.02 whilst its real average approximation ratio is $\frac{1}{0.93} = 1.07$. We also observe that the accuracy of the bound gets worse as we reduce the number of agents in the problem. For instance, the average approximation ratio returned by ST-DTDA with *min-regret* for the same configuration ($|T| = 100$, $|A| = 80$, $|T_a| = 5$) but with 80 agents is 1.23 (see Table 2), much closer to it real average approximation ratio, $\frac{1}{0.90} = 1.11$.

Overall, the best results in each scenario are achieved by ST-DTDA with *min-regret*. On all configurations, ST-DTDA is thus able to achieve high quality solutions (>0.78 of the optimal) and outperform the baseline algorithm.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented ST-DTDA, the first decentralized algorithm for the problem of scheduling a set of tasks on unrelated parallel machines (i.e. $R\|C_{\max}$). In more detail, ST-DTDA uses the min–max message-passing algorithm to compute an approximate solution for the Agent-Based $R\|C_{\max}$ problem in a completely decentralized manner. In so doing, ST-DTDA agents execute three steps. First, they use a heuristic approach that deletes possible agent-task allocations in order to form an acyclic version of the original problem. In particular, we define, and evaluate, three different heuristics for weighting the importance of links between agents and tasks they can do. Secondly, agents run the min–max message-passing algorithm to compute the optimal solution to this relaxed $R\|C_{\max}$ problem. Finally, they compute a per-instance approximation ratio for its solution.

Our empirical evaluation shows that ST-DTDA with the *min-regret* heuristic is able to produce high-quality solutions ($>78\%$ of the optimal) on a range of network structures and distributions of task processing times. In particular, we show that the *min-regret* heuristics that recompute weights during the formation of the relaxed $R\|C_{\max}$ problem and prioritizes the allocations with respect to the regret leads to typically high quality solutions. Finally, we benchmark ST-DTDA against a distributed hill-climbing approach and show that it outperforms it in all tested configurations.

We believe ST-DTDA is a significant first step in the space of decentralized solutions to the unrelated parallel machines problem with machine eligibility restrictions. Yet, the bounding of complexity of ST-DTDA by means of Δ only makes ST-DTDA truly scalable in terms of solution quality in domains where the interactions between the tasks and machines are sparse (i.e. each agent can execute a small number of tasks and each task can be executed by a small number of agents). Otherwise, in very dense problems, the restriction imposed by Δ is likely to lead to many tasks single-connected in the ST and hence, with its allocation fixed already in the first step. Hence, future work will look into extensions that make it scalable and provide more accurate bounds in domains where agent-task interactions are not as sparse as those evaluated in this paper. Finally, in this paper we focused on scheduling problems where the set of tasks are independent. Hence, in future, we intend to extend our approach to deal with problems with task precedence constraints (i.e. where one or more tasks may have to be completed before another task is allowed to start its processing) as in [24].

FUNDING

This work was supported by the iDEaS (www.ideasproject.info) and ORCHID (www.orchid.ac.uk) projects at the University of Southampton.

REFERENCES

[1] Graham, R.L., Lawler, E.L., Lenstra, J.K. and Rinnoy Kan, A.H.G. (1979) Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Discrete Math.*, **5**, 287–326.

[2] Pinedo, M.L. (2008) *Scheduling: Theory, Algorithms, and Systems* (3rd edn). Springer.

[3] Horowitz, E. and Sahni, S. (1976) Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, **23**, 317–327.

[4] Shchepin, E.V. and Vakhania, N. (2005) An optimal rounding gives a better approximation for scheduling unrelated machines. *Oper. Res. Lett.*, **33**, 127–133.

[5] Lenstra, J.K., Shmoys, D.B. and Tardos, E. (1990) Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, **46**, 259–271.

[6] Wooldridge, M. and Jennings, N.R. (1995) Intelligent agents: theory and practice. *Knowl. Eng. Rev.*, **10**, 115–152.

[7] Wotzlaw, A. (2006) Scheduling unrelated parallel machines: algorithms, complexity, and performance. PhD Thesis, Universität Paderborn.

[8] Jensen, F.V. and Nielsen, T.D. (2007) *Bayesian Networks and Decision Graphs* (2nd edn). Springer.

[9] Aji, S.M. and McEliece, R.J. (2000) The generalized distributive law. *IEEE Trans. Inf. Theory*, **46**, 325–343.

[10] Leung, J.Y.-T. and Li, C.-L. (2008) Scheduling with processing set restrictions: a survey. *Int. J. Prod. Econ.*, **116**, 251–262.

[11] Farinelli, A., Vinyals, M., Rogers, A. and Jennings, N.R. (2013) Distributed Search and Constraint Handling. In Weiss, G. (ed.), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press.

[12] Fitzpatrick, S. and Meertens, L. (2002) Experiments on Dense Graphs with a Stochastic, Peer-to-Peer Colorer. *Probabilistic Approaches in Search, Workshop at 18th National Conf. on Artificial Intelligence (AAAI)*, Alberta, Canada, pp. 24–28. AAAI Press.

[13] Farinelli, A., Rogers, A. and Jennings, N.R. (2009) Bounded Approximate Decentralised Coordination using the Max-sum Algorithm. *At the 21st Int. Joint Conf. on Artificial Intelligence (IJCAI)*, Pasadena, California, USA, July, pp. 46–59.

[14] Vinyals, M., Rodríguez-Aguilar, J.A. and Cerquides, J. (2011) Constructing a unifying theory of dynamic programming DCOP algorithms via the generalized distributive law. *J. Auton. Agents Multi-Agent Syst.*, **22**, 439–464.

[15] Pouly, M. and Kohlas, J. (2011) *Generic Inference: A Unifying Theory for Automated Reasoning* (1st edn). Wiley.

[16] Wainwright, M.J. and Jordan, M.I. (2008) Graphical models, exponential families, and variational inference. *Found. Trends Mach. Learn.*, **1**, 1–305.

[17] Collin, Z. and Dolev, S. (1994) Self-stabilizing depth-first search. *Inf. Process. Lett.*, **49**, 297–301.

[18] Paskin, M.A., Guestrin, C. and McFadden, J. (2005) A Robust Architecture for Distributed Inference in Sensor Networks. *Proc. 4th Int. Conf. Information Processing in Sensor Networks (IPSN)*, UCLA, Los Angeles, California, USA, April, pp. 55–62. IEEE.

[19] Vinyals, M., Rodriguez-Aguilar, J.A. and Cerquides, J. (2009) Generalizing DPOP: Action-GDL, A New Complete Algorithm for DCOPs. *Proc. 8th Int. Joint Conf. Autonomous Agents and Multiagent Systems (AAMAS)*, Budapest, Hungary, May, pp. 1239–1240. IFAAMAS.

[20] Rogers, A., Farinelli, A., Stranders, R. and Jennings, N.R. (2011) Bounded approximate decentralised coordination via the max-sum algorithm. *Artif. Intell.*, **175**, 730–759.

[21] Kruskal, J.B. (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, **7**, 48–50.

[22] Vredeveld, T. and Hurkens, C.A.J. (2002) Experimental comparison of approximation algorithms for scheduling unrelated parallel machines. *INFORMS J. Comput.*, **14**, 175–189.

[23] Zhang, W., Wang, G., Xing, Z. and Wittenburg, L. (2005) Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artif. Intell.*, **161**, 55–87.

[24] Satish, N., Ravindran, K. and Keutzer, K. (2007) A Decomposition-Based Constraint Optimization Approach for Statically Scheduling Task Graphs with Communication Delays to Multiprocessors. *Design, Automation and Test in Europe Conf. Exposition (DATE)*, Nice, France, April 16–20, pp. 57–62. ACM.

APPENDIX 1. A BASELINE APPROACH TO DECENTRALIZED $R\|C_{\mathrm{MAX}}$

As stated in Section 2, there are no decentralized algorithms for $R\|C_{\max}$ in the literature. Given this, we describe a baseline benchmarking approach to decentralized $R\|C_{\max}$ that results from applying a standard distributed hill-climbing algorithm to this problem. More concretely, this algorithm is inspired by the DSA, formulated in [12] for distributed constraint optimization problems. The key idea behind DSA is that each variable iteratively selects (with some probability), the value that minimizes its cost given the values chosen by other variables in the last iteration.

In more detail, Algorithm A.1 describes our implementation of DSA for $R\|C_{\max}$. Since decisions in $R\|C_{\max}$ correspond to which agent we assign each task to, we initially assign a group of tasks to each agent $a_i \in A$ for which they are responsible for computing an assignment. We denote this as P_i. The decision as to partition the tasks does not influence the final task allocation. Hence, we use a simple rule where each shared task goes to the agent with the lowest ID. Agents also agree on the maximum number of iterations, $I_{\max}$, that they will run the algorithm for. Finally, DSA uses an activation threshold, p_A, to reduce the likelihood that multiple changes occur simultaneously with outdated information, a phenomenon that results from the decentralization. Thus, Algorithm A.1 receives P_i, $I_{\max}$ and p_A as inputs.

Following Algorithm A.1, each agent starts by running the procedure `initialize` (Algorithm A.1, lines 1–7). First, the agent's assigned task set, $m(a_i)$, is initialized to the set of tasks for which the agent should find an allocation (line 2). Then, for each task, the agent initializes its iteration counter, denoted as $I(t_k)$, to 0 (line 4) and sends a request message to each of the agents which can be assigned that task asking for their marginal costs (line 5).

When an agent a_i receives a `REQUEST`(t_k) message, it runs the procedure given in lines 8–11 to compute its marginal cost for task t_k (given the assignment of all other tasks). The marginal cost of an agent a_i for a task t_k, namely Λ_{ik}, is computed as the sum of the time it takes a_i to compute t_k plus the total time it would take a_i to compute all tasks assigned to it in the current assignment (line 9). This value, along with the time it takes a_i to compute t_k, is then sent back to the requesting agent (line 10).

Algorithm A.1 `DSA()` at agent a_i.

```
Require: The set of tasks for which $a_i$ is responsible for
    computing the allocation ($P_i$), the maximum number of
    iterations ($I_{\max}$) and the probability threshold ($p_T$)
 1: procedure initialise
 2: $m(a_i) = P_i$ //initialise assigned tasks
 3: for all $t_k \in P_i$ do
 4:     $I(t_k) = 0$ //initialise iterations counters
 5:     Send REQUEST($t_k$) to all $\{a_k \in A | t_k \in T_j\}$//ask for marginal costs
 6: end for
 7: end procedure

 8: procedure received REQUEST($t_k$) from agent $a_j$
 9: $\Lambda_{ik} = \chi_i(t_k) + \sum_{t_l \in m(a_i), l \neq k} \chi_i(t_l)$ //recompute marginal cost
10: Send $\langle \Lambda_{ik}, \chi_i(t_k), t_k \rangle$ to $a_j$
11: end procedure

12: procedure received $\langle \Lambda_{jk}, \chi_i(t_k), t_k \rangle$ from agent $a_j$
13: if Received $\Lambda_{lk}$ from all $A_k = \{a_l \in A | t_k \in T_l\}$ then
14:     if $rand() \leq p_T$ then
15:         $a^* = \arg\min_{a_l \in A_k} \max\left(\Lambda_{lk}, \max_{a_m \in A_k, m \neq s} \Lambda_{mk} - \chi_m(t_k)\right)$
16:         Send ASSIGN($t_k$) to $a^*$// Assign to agent with minimum marginal cost
17:         Send UNASSIGN($t_k$) to agent $t_k$ was previously assigned to
18:         $I(t_k) = I(t_k) + 1$ //Increment the number of iterations for this task
19:         if $I(t_k) < I_{\max}$ //Start next iteration then
20:             Send REQUEST($t_k$) to all $a_j \in \mathcal{A} | t_k \in T_j$
21:         end if
22:     end if
23: else
24:     Defer until next received message
25: end if
26: end procedure

27: procedure received ASSIGN($t_k$)
28: $m(a_i) = m(a_i) \cup t_k$ //Add to assigned tasks
29: end procedure

30: procedure received UNASSIGN($t_k$)
31: $m(a_i) = m(a_i) \setminus t_k$ //Remove from assigned tasks
32: end procedure
```

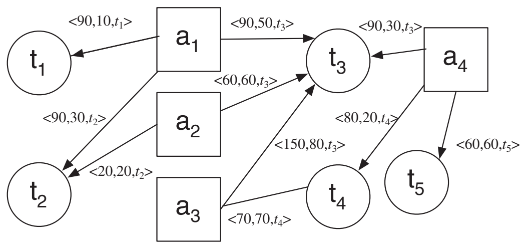


FIGURE A1. Agents respond to REQUEST messages during the first iteration of DSA algorithm over the TDN depicted in Fig. 1.

Once an agent has received the marginal costs from all agents to which a task t_k can be assigned (line 13), it proceeds to consider changing the current assignment of this task with a probability p_A (line 14). Thus, if the generated random number (given by the rand() function) is lower than p_A, the agent proceeds to compute the best assignment by finding the agent that minimizes the cost of the assignment. The cost of assigning an agent a_l to task t_k is computed as the maximum between this agent's marginal cost, Λ_{lk}, and the marginal cost of the rest of the agents $\{a_m, t_k \in T_m, m \neq l\}$, when excluding this assignment, $\Lambda_{mk} - \chi_m(t_k)$ (line 15). Agent a_i sends an ASSIGN(t_k) message to the agent with the minimum cost of assignment (line 16) and an UNASSIGN(t_k) message to the agent to which t_k is currently assigned (line 17). Then, the iteration counter is incremented (line 18), and, if the algorithm has not reached the maximum number of iterations, the process starts over again (lines 19–21). Once all iterations have been completed for all tasks $t_k \in T$, each agent has a set of tasks assigned by the algorithm.

Figure A1 shows a single iteration of the first step of DSA over the TDN in Fig. 1. In this example, agent a_1 is initially assigned tasks t_1, t_2 and t_3 for which it is in charge of computing the allocation. Similarly, t_4 is assigned to a_3 and t_5 to a_4. Agents begin by sending a REQUEST message, effectively from each task to each agent that can compute it (following the links agent-task in Fig. 1) and on receipt of a REQUEST(t_k) from another agent, each agent computes its Λ_{ik} value as detailed in Algorithm A.1, and sends that back, along with the task processing cost, to the sender for t_k (this is denoted by the text on the arrows in Fig. A1). Observe that, for example, the marginal cost of assigning t_2 to a_1 is 90, even when the execution time of t_2 is 30, because in the initial assignment agent a_1 executes t_1 and t_3.

APPENDIX 2. PROOFS OF BOUNDED APPROXIMATION

In this appendix, we give the proofs for the absolute bound ε. Namely, we prove that

$$\tilde{V}^* - V^* \leq \left[\varepsilon = \min_{\tilde{X}} \max_{a_i \in A} \sum_{\substack{t_k \in T^M \\ \tilde{x}_k = i}} \chi_i(t_k) \right]. \tag{A.1}$$

To prove Equation (A.1), consider the subset of assignments of variables in $\tilde{X}$ in which the set of tasks in T^M (e.g. tasks whose domain have been modified with respect to the original problem) are assigned to the same agent as they are assigned to in the optimal solution. That is, any assignment $\tilde{X}^\alpha$ such that $x_i^\alpha = x_i^*$ if $t_i \notin T^M$. Note that since $\tilde{X}^\alpha$ is an assignment of variables $\tilde{X}$, the value of any $\tilde{X}^\alpha$, $\tilde{V}^\alpha$, will always be higher (worse) than the optimal value of the ST approximate problem, $\tilde{V}^*$. Then, it follows that $\forall \tilde{X}^\alpha$:

$$\begin{aligned} \tilde{V}^* - V^* &\leq \tilde{V}^\alpha - V^* \\ &= \max_{a_i \in A} \left(\sum_{\substack{t_k \in T^M \\ x_k^* = i}} \chi_i(t_k) + \sum_{\substack{t_k \notin T^M \\ \tilde{x}_k^\alpha = i}} \chi_i(t_k) \right) \\ &\quad - \max_{a_i \in A} \left(\sum_{\substack{t_k \in T \\ x_k^* = i}} \chi_i(t_k) \right) \\ &\leq \max_{a_i \in A} \left(\sum_{\substack{t_k \notin T^M \\ \tilde{x}_k^\alpha = i}} \chi_i(t_k) - \sum_{\substack{t_k \notin T^M \\ x_k^* = i}} \chi_i(t_k) \right) \\ &\leq \max_{a_i \in A} \left(\sum_{\substack{t_k \notin T^M \\ \tilde{x}_k^\alpha = i}} \chi_i(t_k) \right). \end{aligned}$$

Thus, the value of the makespan of any assignment of tasks in T^M to agents in the ST problem bounds the distance of $\tilde{V}^*$ to the optimal V^*. This also holds for the particular assignment of tasks T^M that minimizes the makespan. Thus, Equation (A.1) holds.