Reachability Games for Optimal Multi-agent Scheduling of Tasks with Variable Durations

Dhananjay Raju(✉), Niklas Lauffer, and Ufuk Topcu

The University of Texas at Austin, Austin, USA
{draju,nlauffer,utopcu}@utexas.edu

Abstract. Scheduling tasks with variable durations across multiple agents is an NP-hard problem for even two agents. Typically, the runtime of any exact algorithm is dominated by the number of tasks because of an exponential dependence. We shift this exponential dependency from the number of tasks to a new parameter, which we call *window length*. This novel parameterization enables to reduce the problem of finding an optimal schedule to one of searching for winning strategies in a two-player reachability game on graphs of *size polynomial in the number of tasks*. As such, the complexity of finding an optimal schedule is polynomial in the number of tasks but exponential in the window length. We demonstrate that, in practice our algorithm runs faster than the worst-case complexity. The approach we present is applicable for most common optimization criteria, such as minimization of *makespan* and *total load*. We demonstrate the practical value of this technique by finding optimal schedules for astronauts aboard the International Space Station. Finally, experiments on randomly generated instances show that, on average, this technique is at least two orders of magnitude faster than an integer program formulation.

Keywords: Multi-agent scheduling · Graph games · Variable durations · Linear optimization criteria · Schedulability

1 Introduction

We aim to understand the role of task structure in the complexity of finding an optimal schedule for the *agent resource-constrained project scheduling problem* (ARCPSP) with variable task durations [14]. The ARCPSP is an extension of the *resource-constrained project scheduling problem* (RCPSP) with a notion of agents that execute tasks in parallel. The problem includes lower and upper limits for each task's duration. Therefore, scheduling also involves assigning execution durations from the interval for each task.

This work has been supported in part by the grants NASA NNX17AD04G, NSF 1652113 and NSF 1646522.

W. Wu and Z. Zhang (Eds.): COCOA 2020, LNCS 12577, pp. 151–167, 2020.
https://doi.org/10.1007/978-3-030-64843-5_11

The ARCPSP is a combinatorial optimization problem and is NP-hard for even two agents [17]. It is possible to solve the problem efficiently if one can divide the planning horizon into smaller time windows so that tasks start and end within individual windows. However, this condition is often impractical and tasks do spillover. We relax this restrictive condition. Specifically, we allow such spillovers but limit them to the next window, but not the windows beyond. This relaxed condition is natural in many scheduling scenarios. We describe two of them here. Consider software development teams (a group of agents) that aim to create software through *sprints*. A sprint is a short time period when a team works to complete a number of tasks. Ideally, tasks that start in a sprint (window) should end in the same sprint. However, it is not always possible to satisfy this condition. In practice, task spillovers are pushed to the next sprint and special efforts are made to avoid additional spillovers [13]. Consider a second scenario where tasks correspond to goods being delivered to customers by a fixed number of agents. After a customer chooses her delivery slot, the company has to ensure that the deliveries are performed with minimal latencies. In other words, the task spillovers are restricted.

In Sect. 2, we introduce a new parameter which we call *window length* (Δ). The window length is chosen such that task spillovers are restricted to adjacent windows. More specifically, the window length is the smallest integer such that tasks that start in a window only spillover to the next window but not the windows after. This parameterization enables to encode all feasible schedules as paths in a graph of *size polynomial in the number of tasks*. An alternate notion of *windows* has been used to restrict the difference between the start times of various tasks, when the task durations are fixed [19]. However, we use windows to restrict the length of individual tasks.

Uncertainty is prevalent in scheduling due to a lack of accurate process models and variability on the process and environmental data [11]. As such, it is impossible to estimate the durations for tasks without uncertainty. Following [15], we model the uncertainty in the task durations by allowing the tasks to have variable durations. In this paper, we compensate for such uncertainty by allocating each task the maximum permissible duration while simultaneously ensuring that the resulting schedule does not violate the resource constraints.

We study optimization criteria defined as functions on *non-idling* durations of individual agents. Due to the high complexity of the problem, such objectives are seldom studied even if they have a wide range of practical applications. For example, such optimization criteria enable to assign weights to agents to give preference to schedules that maximize the use of agents with higher weights. In Sect. 5, we find optimal schedules for astronauts aboard the International Space Station (ISS). We model a scenario where some of the astronauts can be ill or injured and try to minimize the assignment of tasks to such astronauts while simultaneously finding a schedule that maximizes the execution time for the tasks.

We cast the problem of finding an optimal schedule as a two-player reachability game on graphs of *size polynomial in the number of tasks*. In the context

of scheduling, reachability games have previously been used for finding feasible schedules for sporadic tasks [7]. On the other hand, we use these games to find optimal schedules. In Table 1, we list the complexities of finding optimal schedules for different optimization criteria when the window length Δ and the number of agents k are fixed constants. The definition of Δ implies that the *planning horizon* $\mathcal{H}$ is $\mathcal{O}(2n\Delta)$, where n is the number of tasks (we assume that there are no empty windows). Even though the worst-case complexities are exponential in k and Δ, the experiments in Sect. 5 show that we can find optimal schedules for five agents that have to complete 100 tasks within a day, in under a minute, when a) each time step is ten minutes long b) the maximum duration of every task is at most five hours and c) the difference between the earliest start time and latest start time for every task is at most five hours.

Table 1. The complexity of finding optimal schedules. $\mathcal{H}$ is the length of the planning horizon, k is the number of agents and Δ is the window length.

Type of optimization	Complexity
None (feasibility)	$\mathcal{O}_{k,\Delta}(\mathcal{H})$
Linear function	$\mathcal{O}_{k,\Delta}(\mathcal{H}^3)$
Min total load	$\mathcal{O}_{\Delta}(\mathcal{H}^{k+2})$
Min makespan	$\mathcal{O}_{k,\Delta}(\mathcal{H}^3)$

In Sect. 5, we validate the technique for different objectives on randomly generated instances. The experiments show that the technique works well, even for a large number of tasks and long planning horizons. Lastly, we compare our technique against an integer programming encoding of the problem that we run in Gurobi [10]. Experiments show that the technique is at least two orders of magnitude faster.

2 The Agent Resource-Constrained Project Scheduling Problem and Windows

An instance I of the *agent resource-constrained project scheduling problem* is a tuple $(\mathcal{A}, \mathcal{T}, \mathcal{D}, \mathcal{S}, \mathcal{C}, B, R)$ defined as follows.

- $\mathcal{A} = \{1, 2, \ldots, k\}$ is the set of agents.
- $\mathcal{T} = \{1, 2, \ldots, n\}$ is the set of tasks.
- $\mathcal{D} = \left\{(d_t^{min}, d_t^{max}) \mid t \in \mathcal{T} \text{ and } d_t^{min}, d_t^{max} \in \mathbb{N}\right\}$ is a set of pairs of *minimum and maximum duration* for every task. Each task has to be scheduled for at least the minimum duration and at most the maximum duration.
- $\mathcal{S} = \left\{(s_t^e, s_t^\ell) \mid t \in \mathcal{T} \text{ and } s_t^e, s_t^\ell \in \mathbb{N}\right\}$ is a set of pairs of *earliest start time* and *latest start time* for every task. Every task has to be scheduled at or after the earliest start time and before or at the latest start time.

- For any t in $\mathcal{T}$, $\mathcal{C}_t \subseteq \mathcal{A}$ is the set of agents that can perform task t. Let $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$.
- There are m types of renewable resources. The maximum quantities of the resources are encoded in a vector B in $\mathbb{N}^m$. B_i gives the maximum quantity of resource i.
- R in $\mathbb{N}^{n \times m}$ is a matrix that encodes the resource requirements of the tasks. R_{tj} gives the quantity of resource j required by task t.

The *planning horizon* $\mathcal{H}$ of an instance I is defined as $\max\left\{s_t^\ell + d_t^{max} \mid t \in \mathcal{T}\right\}$. In the rest of the paper, we use x to denote an arbitrary non-negative integer.

Associated with every ARCPSP instance is a parameter, *window length* denoted by Δ, that is intrinsic to the instance. Formally, Δ is the smallest positive integer such that, for all tasks t, if $x\Delta < s_t^e \leqslant (x+1)\Delta$, then $s_t^\ell + d_t^{max} \leqslant (x+2)\Delta$. The definition of Δ implies that, if the entire planning horizon is partitioned into intervals of size Δ, then the tasks that start in an interval can only spill over to the next interval but not the interval after. We refer to the interval $(x\Delta, (x+1)\Delta]$ as *window* x. The definition of window length implies that $0 < \Delta \leqslant d_{max}$, where $d_{max} = \max\{(s_t^\ell + d_t^{max}) - s_t^e + 1 \mid t \in \mathcal{T}\}$. Therefore, given a scheduling instance I, the window length can be determined in time polynomial in the number n of tasks and d_{max}. Henceforth, we assume that Δ is given.

Remark 1. The window length restricts the maximum duration for the tasks depending on the starting time inside a window. For example, consider a task t that has a maximum duration of 2Δ and can start in the interval $(x\Delta, (x+1)\Delta]$. If it starts at $x\Delta+1$, then its maximum duration is 2Δ. If it starts at $x\Delta+r+1$ (where, $r < 2\Delta$ and $r \in \mathbb{N}$), then the maximum duration is at most $2\Delta - r$. The scheduling of tasks with starting-time-dependent execution times has been extensively studied [5,6]. More specifically, it is known to be NP-hard [12].

A set $\mathcal{T}' \subseteq \mathcal{T}$ of tasks is *permissible* if the sum of the quantities of each type of resource required by the tasks in $\mathcal{T}'$ is less than or equal to its maximum quantity. Formally, $\mathcal{T}'$ is permissible, if for every resource $j \in [m]$, $\sum_{t \in \mathcal{T}'} R_{tj} \leqslant B_j$. The set of all permissible sets is the complement of the set of all *forbidden sets of tasks* (a set of tasks that cannot be scheduled together) [20]. In general, the enumeration of forbidden sets (consequently of permissible sets) for any instance is computationally expensive [21]. Intuitively, if the earliest starting time of a task is after another task ends, then the two tasks can never interfere with each other. By the definition of window length, only tasks with the earliest start time in the intervals $((x-1)\Delta, x\Delta]$, $(x\Delta, (x+1)\Delta]$ or $((x+1)\Delta, (x+2)\Delta]$ can interfere with a task with the earliest start time in the interval $(x\Delta, (x+1)\Delta]$. Permissible sets over such tasks are said to be *relevant*. We denote the set of all relevant permissible sets by $\mathcal{P}$. Algorithm 1 computes all relevant permissible sets in time $\mathcal{O}\big((k\Delta)^k\mathcal{H}\big)$. In the rest of the paper, we assume that $\mathcal{P}$ is given.

A *schedule* is a set of tuples of the form (s_t, a_t, d_t) for every task t, where s_t is the actual start time, a_t is the agent assigned to the task and d_t is the exact

Algorithm 1. Finding all the relevant permissible sets.

Input an instance I of the ARCPSP.
Output the relevant permissible sets in I.

```
P ← ∅
for i := 0 to floor(H/Δ) do
    X ← {t : s_t^e ∈ (iΔ, (i+1)Δ]}
    for all U ⊂ X s.t. |U| ⩽ k do
        if Σ_{k∈U} R_kj < B_j for all j ∈ [1, m] then
            P ← P ∪ {U}
        end if
    end for
end for
return P
```

duration allocated to the task. A schedule is *valid* if the following conditions are satisfied.

(a) At any time, every agent is assigned at most one task.
(b) At any time, the set of tasks scheduled together is permissible.
(c) Tasks that have been scheduled are not preempted.
(d) For every task t in $\mathcal{T}$, $s_t^e \leqslant s_t \leqslant s_t^\ell$, a_t in $\mathcal{C}_t$ and $d_t^{min} \leqslant d_t \leqslant d_t^{max}$.

One reason for the nonexistence of a valid schedule may be the lack of sufficiently many agents. In any window, k agents can complete at most $k\Delta$ tasks. Therefore, if there is a window $(x\Delta, (x+1)\Delta]$ such that the number of tasks that have to be completed in the window is more than $k\Delta$, then no valid schedule exists. In this case, we say that the number of agents is *insufficient*. In the rest of the paper, we assume that the number of agents is sufficient. For clarity, we restate the assumptions.

(A1) The number k of agents is sufficient.
(A2) The window length Δ is given.
(A3) The set $\mathcal{P}$ of all relevant permissible sets is given.

3 Encoding Valid Schedules as Paths in a Graph

With every instance $I = (\mathcal{A}, \mathcal{T}, \mathcal{D}, \mathcal{S}, \mathcal{C}, B, R)$, we associate a graph $G_I = (V_I, E_I)$. Intuitively, each vertex in V_I corresponds to a configuration of the agents at some time in the planning horizon of I. There is an edge (u, v) in E_I if and only if it is possible for the configuration of the agents corresponding to vertex u to progress to the configuration corresponding to vertex v in the following time step without violating the constraints of I. If a vertex has more than one out-edge, it means that the corresponding configuration of the agents can progress in different ways depending on the scheduling decision.

Every path in the graph G_I corresponds to a valid sequence of configurations of the agents. By designating an initial vertex v_{init} and a final vertex v_f corresponding, respectively, to the initial configuration of agents before execution

and the final configuration of agents after completing all the tasks, a path from v_{init} to v_f corresponds to a sequence of scheduling decisions constituting a valid schedule for I.

Every vertex in V_I has four components. The first component is a vector that holds the task assignment for every agent along with the duration left to complete the task. An *idling* agent, i.e., an agent which is not assigned any task from $\mathcal{T}$, is assigned a dummy task 0. The second component is the current time. The third component has a set of tasks that have to be completed by the end of the window corresponding to the current time. The fourth component records the set of tasks completed so far in the window. Formally,

$$v = \big(\big((1, t_1, \ell_1), \ldots, (k, t_k, \ell_k)\big), \tau, F, C\big) \in ([k] \times \mathcal{T} \times [2\Delta])^k \times [\mathcal{H}+1] \times 2^{\mathcal{T}} \times 2^{\mathcal{T}}$$

belongs to V_I if the following conditions are satisfied.

1. There exists $P \in \mathcal{P}$ such that $\{t_1, \ldots, t_k\} \subseteq P \cup \{0\}$, i.e., the set of tasks scheduled at any time is permissible.
2. For every currently assigned task $t_a \in \{t_1, \ldots, t_k\}$, $a \in \mathcal{C}_{t_a}$, i.e., the agent assigned to the task can perform it.
3. For all pairs t, t' of tasks in $\{t_1, \ldots, t_k\} \setminus \{0\}$, $a_t \neq a_{t'}$, i.e., the same agent cannot be assigned multiple tasks (not idling).
4. For every agent a in $\mathcal{A}$, $0 \leqslant \ell_a \leqslant d_{t_a}^{max}$, i.e., the duration left for the task allocated to agent a is shorter than or equal to the maximum duration.
5. $0 \leqslant \tau \leqslant \mathcal{H}+1$.
6. $F \subseteq \mathcal{T}$ and if $x\Delta < \tau \leqslant (x+1)\Delta$, then, for every task t in F, $x\Delta < s_t^{\ell} + d_t^{max} \leqslant (x+1)\Delta$.
7. $C \subseteq \mathcal{T}$ and if $x\Delta < \tau \leqslant (x+1)\Delta$, then, for every task t in C, $(x-1)\Delta < s_t^{e} + d_t^{min} \leqslant s_t^{\ell} + d_t^{max} \leqslant (x+1)\Delta$.
8. For every task t in $\{t_1, \ldots, t_k\}$, if $t \neq 0$, then $t \in F$ and $t \notin C$.

Let $v = \big(\big((1, t_1, \ell_1), \ldots, (k, t_k, \ell_k)\big), \tau, F, C\big)$ and $v' = \big(\big((1, t'_1, \ell'_1), \ldots, (k, t'_k, \ell'_k)\big), \tau', F', C'\big)$ be two vertices in G_I. The vertex v is said to be *in the window* x *if* $\tau \in (x\Delta, (x+1)\Delta]$. The initial vertex v_{init} is $\big(\big((1,0,1),(2,0,1),\ldots,(k,0,1)\big), 0, \varnothing, \varnothing\big)$ and the final vertex v_f is $\big(\big((1,0,1),(2,0,1),\ldots,(k,0,1)\big), H+1, \varnothing, \varnothing\big)$.

There are three types of edges in G_I.

(E1) *The edges between two vertices in the same window.*
(E2) *The edges from vertices in a window to vertices in the next window.*
(E3) *The edges to the final vertex.*

We formally define the three types of edges in the graph G_I in Table 2. The edges of type **(E1)** and **(E1)** correspond to the assignment of new tasks; some of the agents may be assigned new tasks, while others continue their previously assigned task. The edges of type **(E1)** are from vertices corresponding to the completion of all the tasks to the final vertex v_f.

The following lemma provides a necessary and sufficient condition for the existence of a valid schedule.

Table 2. The three types of edges in G_I.

Edge type	Conditions for edge between v and v'
(E1)	$\forall x \in \mathbb{N}_{\geqslant 0} : \tau \neq x\Delta$ and $\tau \neq \mathcal{H}$. $\forall a \in [k]$: if $t_a = t'_a \neq 0$, then $\ell'_a = \ell_a - 1$. $\forall a \in [k]$: if $t_a = t'_a = 0$, then $\ell'_a = 1$. $\forall a \in [k]$: if $t_a \neq t'_a$, then $\ell_a \leqslant d^{max}_{t_a} - d^{min}_{t_a}$, $\ell'_a = d^{max}_{t'_a}$, $s^e_{t'_a} \leqslant \tau + 1 \leqslant s^{\ell}_{t'_a}$ and if $t'_a = 0$, then $\ell'_a = 1$. $(\tau', F', C') = (\tau + 1, F \backslash \{t_a \mid t_a \neq t'_a\}, C \cup \{t_a \mid t_a \neq t'_a\})$.
(E2)	$\exists x \in \mathbb{N}_{\geqslant 0} : x < \lfloor \mathcal{H}/\Delta \rfloor$ and $\tau = x\Delta$. $\forall a \in [k]$: if $t_a = t'_a \neq 0$, then $\ell'_a = \ell_a - 1$. $\forall a \in [k]$: if $t_a = t'_a = 0$, then $\ell'_a = 1$. $\forall a \in [k]$: if $t_a \neq t'_a$, then $\ell_a \leqslant d^{max}_{t_a} - d^{min}_{t_a}$, $\ell'_a = d^{max}_{t'_a}$, $s^e_{t'_a} \leqslant \tau + 1 \leqslant s^{\ell}_{t'_a}$ and if $t'_a = 0$, then $\ell'_a = 1$. $\tau' = \tau + 1$. $F' = (F \cup \{t \in \mathcal{T} : x\Delta < s^{\ell}_t + d^{max}_t \leqslant (x+1)\Delta\}) \backslash (\{t_a \mid t_a \neq t'_a\} \cup C)$. $C' = \varnothing$.
(E3)	$\tau = \mathcal{H}$ and $\tau' = \mathcal{H} + 1$. $F = \{t \in \mathcal{T} : \exists\ a \in [k],\ t = t_a$ and $\ell_a \leqslant d^{max}_{t_a} - d^{min}_{t_a}\}$. $\forall a \in [k]$: if $t_a \neq 0$, then $\ell_a \leqslant d^{max}_{t_a} - d^{min}_{t_a}$, $\ell'_a = 1$ and $t'_a = 0$. $(F', C') = (\varnothing, \varnothing)$.

Lemma 1. *There is a valid schedule for the instance I if and only if there is a path from v_{init} to v_f in G_I.*

Proof. ($\Rightarrow$) In any path ρ from v_{init} to v_f, task t has been assigned to an agent a if there is a vertex of the form $\big((\dots, (a, t, d^{max}_t), \dots), \tau, F, C\big)$. If $x\Delta \leqslant s^{\ell}_t + d^{max}_t \leqslant (x+1)\Delta$ and the task is not completed by either time $x\Delta$ or $x + 1\Delta$, then there will be no out-edge from the vertex with time $\tau + 1$ in the path. Additionally, by the definition of edge types **(E1)** and **(E2)**, when task t is completed, it is removed from F, hence it cannot be reassigned. Furthermore, no task can be assigned simultaneously to multiple agents. The start time for task t is $s_t = \min\big\{\tau \mid \big((\dots, (a, t, d^{max}_t), \dots), \tau, F, C\big) \in \rho\big\}$ and the end time is $e_t = \max\big\{\tau \mid \big((\dots, (a, t, d^{max}_t, \dots), \tau, F, C\big) \in \rho\big\}$. The duration is the difference between the end time and the start time, i.e., $d_t = e_t - s_t$.

($\Leftarrow$) Every valid schedule induces a path from v_{init} to v_f. □

Given an ARCPSP instance I, we compute a valid schedule by constructing the corresponding graph G_I and searching for a path from v_{init} to v_f in G_I. Given a path from v_{init} to v_f, Algorithm 2 presents the procedure to extract a valid schedule corresponding to a path from v_{init} to v_f.

Lemma 2. *A valid path can be computed in time* $\mathcal{O}\big(\mathcal{H}\big((2k\Delta^2)^k \cdot 2^{4k\Delta}\big)^2\big)$.

Proof. The number of vertices in the graph G_I is $\mathcal{O}\big((2k\Delta^2)^k \cdot \mathcal{H} \cdot 2^{4k\Delta}\big)$. The first component of any vertex in V_I has a task assignment for each agent and the duration left for the task. Since the number of agents is sufficient (Assumption **(A1)**), any agent may start at most $2k\Delta$ tasks over the course of the schedule. Moreover, the duration left for the task is at most 2Δ. The size of the first component is $\mathcal{O}(2k\Delta^2)^k)$. The size of the second component is $\mathcal{H}$. The third component F encodes the set of tasks to be completed by the end of the current window. If the number of agents are sufficient, then the maximum number of tasks that can finish in the window is $2k\Delta$. In the worst case, the size of F (third component) is $2^{2k\Delta}$. The size of the third component is the same as the size of the fourth component. The number of edges in G_I is $\mathcal{O}\big(\mathcal{H}\big((2k\Delta^2)^k \cdot 2^{4k\Delta}\big)^2\big)$. In any graph, a path between any two vertices can be computed in $\mathcal{O}(|V| + |E|)$. □

Algorithm 2. Schedule corresponding to a path from v_{init} to v_f in G_I.

Input path $\rho = (v_{init}, v_1, ..., v_k, v_f)$ in G_I.
Output valid schedule S.

```
S ← {0, ..., 0} {empty schedule of length n}
for ((1, i, ℓ_i), ..., (k, j, ℓ_j)), τ, F, C) ∈ ρ do
    for (a, t, ℓ) ∈ ((1, i, ℓ_i), ..., (k, j, ℓ_j)) do
        if t ≠ 0 then
            d ← d_t^max − ℓ {duration that t has run for so far}
            s ← τ − d {time that t was started}
            S_t ← (s, a, d)
        end if
    end for
end for
return S
```

Remark 2. The complexity of finding a valid schedule depends on the number of tasks that have to be completed in each window. If we fix $\mathcal{H}$ and Δ and increase n, then the number of tasks per window increases. The worst-case complexity for particular values of $\mathcal{H}$, k and Δ occurs when we have to schedule close to $k\Delta$ tasks in every window.

4 Reachability Games for Optimal Scheduling

In this section, we define optimal schedules and provide a technique to compute optimal schedules by building upon the graph construction presented in Sect. 3.

Let S be a schedule for an instance $I = (\mathcal{A}, \mathcal{T}, \mathcal{D}, \mathcal{S}, \mathcal{C}, B, R)$. For every agent a, let $w_a \in \mathbb{Z}$ be the *weight* of the agent. The *value of the agent* in the schedule S denoted by $value_S(a)$ is defined as

$$value_S(a) = \sum_{(s_t, a, d_t) \in S} d_t,$$

i.e., the value of the agent a in the schedule S is defined as the sum of the duration of the tasks assigned to the agent a in schedule S. The value of the schedule S denoted by $val(S)$ is defined as

$$val(S) = \sum_{a \in \mathcal{A}} w_a \cdot value_S(a).$$

A schedule S for instance I is *optimal* if for every other schedule S', $val(S') \leqslant val(S)$. Let $\tau \in [0, \mathcal{H} + 1]$, where $\mathcal{H}$ is the planning horizon of I, then the *value of a schedule up to a time* τ denoted by $val(\tau; S)$ is defined as

$$val(\tau; S) = \sum_{a \in \mathcal{A}} w_a \sum_{(s_t, a, d_t) : s_t \leqslant \tau} \min\{d_t, \tau - s_t + 1\}.$$

Since the value of a schedule is a linear function on the value of each agent in the schedule, we observe that

$$val(\tau + 1; S) = val(\tau; S) + \sum_{\substack{a \in \mathcal{A}: \\ \text{a is not idle at } \tau+1}} w_a. \tag{1}$$

We incentivize the assignment of tasks to a particular agent by giving it a greater weight compared to the weights of the other agents.

4.1 Two-Player Reachability Games

For each problem instance, we construct a *two-player reachability game* such that we can obtain an optimal schedule from any *memoryless winning strategy* for one of the players, which we call the *reachability player*.

A two-player *reachability game* [16] is played on a graph $G = (V, E)$ between a reachability player and a safety player. Initially, a token is placed on a designated *initial vertex* v_η in V. The two players take turns moving the token along the edges of the graph. The objective of the reachability player is to move the token to a vertex in $\mathcal{F} \subseteq V$. The safety player tries to prevent the token from reaching a vertex in $\mathcal{F}$. A play ρ is a (possibly infinite) sequence $v_\eta v_1 ...$ of vertices such that $(v_i, v_{i+1}) \in E$ for all $0 \leqslant i$. The play ρ is *winning* for the reachability player if, for some v_i in ρ, $v_i \in \mathcal{F}$. A *memoryless strategy* $\sigma : V \to V$ maps every

vertex to one of its successors. The play ρ is said to *agree* with the strategy σ for player P if $v_{i+1} = \sigma(v_i)$ whenever player P has to play from v_i. A memoryless strategy is said to be *winning* for player P if all plays that agree with it are winning for player P. Such games are *determined*, i.e., one of the players has a memoryless winning strategy [16]. Additionally, a memoryless winning strategy for the winning player can be found in $\mathcal{O}(|V| + |E|)$ [4,18]. An extensive survey of reachability games on graphs can be found in [2,9].

Two-player reachability games on graphs have been used for *online scheduling* of *sporadic tasks* [7,8]. In online scheduling, the reachability player tries to create tasks that will miss the deadline and the safety player is the scheduler who tries to ensure that none of the tasks miss their deadlines.

4.2 Optimal Scheduling

In the setting of this paper, the reachability player tries to find an optimal schedule, whereas the safety player tries to produce an alternate schedule with a value greater than the one produced by the reachability player.

With every instance $I = (\mathcal{A}, \mathcal{T}, \mathcal{D}, \mathcal{S}, \mathcal{C}, B, R)$, we associate a graph $G_I^{linear} = (V, E)$. The vertices of the graph have four components. We use the first and second components to record the schedules for the reachability player and the safety player, respectively. The third component, which we call the *counter*, holds the difference between the values of the two schedules. The difference between the values of any two valid schedules is at most $c\mathcal{H}$ and at least $-c\mathcal{H}$, where $c = k \cdot \max\{|w_a| \mid a \in \mathcal{A}\}$. The last component records the *turn*, i.e., the player that has to move the token next. Let $G_I = (V_I, E_I)$ be the graph corresponding to instance I as defined in Sect. 3.

The set of vertices of the graph G_I^{linear} is $V = V_I \times V_I \times [-c\mathcal{H}, c\mathcal{H}] \times \{r, s\}$. The initial vertex v_η is the tuple $(v_{init}, v_{init}, 0, r)$ and the set of winning vertices for the reachability player is $\mathcal{F} = \{(v_f, v_f, \delta, \alpha) \mid \delta \geqslant 0,\ \alpha = s \vee \alpha = r\}$. There are three types of edges in G_I^{linear}.

(E4) *The edges starting from vertices with turn* r.
(E5) *The edges starting from vertices with turn* s.
(E6) *The direct edges to* $(v_f, v_f, 0, r)$.

Table 3. The three types of edge in G_I^{linear}.

Edge type	Conditions for edge between χ and χ'
(E4)	$\alpha = r$, $\alpha' = s$, $v = v'$, $(u, u') \in E_I$ and $\delta' = \delta + \sum_{t'_a \neq 0 \ \in u'} w_a$.
(E5)	$\alpha = s$, $\alpha' = r$, $u = u'$, $(v, v') \in E_I$ and $\delta' = \delta - \sum_{t'_a \neq 0 \ \in v'} w_a$.
(E6)	There is no out going edge from v in G_I, $\alpha = s$, $\alpha' = r$, $u' = v' = v_f$ and $\delta' = 0$.

The edges of type **(E4)** and **(E5)** record the scheduling choices of the reachability player and the safety player, respectively. The edges of type **(E6)** ensure that the reachability player wins the game when the safety player has no action to extend its schedule. Let $\chi = (u, v, \delta, \alpha)$ and $\chi' = (u', v', \delta', \alpha')$ be two vertices in V. We formally define the three types of edges in Table 3.

By Lemma 1, every valid schedule for I induces a valid path from v_{init} to v_f in the graph G_I. Since the first component of the vertex set V is V_I, every valid schedule induces a valid path on this component. Thus, reachabilty player can use any valid schedule as a memoryless strategy. Moreover, any play ρ that agrees with such a strategy, induces a path from v_ι to a vertex of the form (v_f, v, δ, s), where $v \in V_I$ and $\delta \in [-c\mathcal{H}, c\mathcal{H}]$.

In the reachability game on the graph G_I^{linear}, the two players take turns to assign tasks to agents for each time step in the scheduling horizon such that the constructed schedules are valid. If the reachability and safety players follow schedules S_1 and S_2 respectively, then Eq. (1) implies that

$$val(\tau+1; S_1) - val(\tau+1; S_2) = (val(\tau; S_1) - val(\tau; S_2)) + \sum_{\substack{a \text{ is not idle at } \tau \\ \text{in } S_1}} w_a - \sum_{\substack{a \text{ is not idle at } \tau \\ \text{in } S_2}} w_a. \quad (2)$$

For finding the optimal schedule, we require that the value of the schedule chosen by the reachability player is greater than or equal to the value of the schedule chosen by the safety player, i.e., the difference between these values is non-negative. We maintain and update this difference according to Eq. (2), by recording the difference in the counter. If the difference in the value of the two chosen schedules is non-negative at the end of the scheduling horizon, then the reachability player reaches a vertex in $\mathcal{F}$.

Lemma 3. *If the reachability player follows an optimal schedule, then it wins the game.*

Proof. In the reachability game on G_I^{linear}, the two players take turns in constructing their respective schedules. We maintain the difference between the values of the two schedules constructed so far in the counter. This difference is always contained in the closed interval $[-c\mathcal{H}, c\mathcal{H}]$. If the safety player follows a valid schedule and the reachability player follows an optimal schedule, by construction of G_I^{linear}, the difference is non-negative and a vertex in $\mathcal{F}$ is reached. However, if the safety player does not follow a valid schedule, then the reachability player wins by using an edge of type **(E6)**. □

Corollary 1. *If the reachability player does not follow an optimal schedule and the safety player follows one, then the safety player wins the game.*

Theorem 1. *A memoryless winning strategy for the reachability player can be computed in time* $\mathcal{O}\big(\mathcal{H}^3\big((2k\Delta^2)^k \cdot 2^{4k\Delta}\big)^2\big)$.

Proof. The number of vertices in G_I^{linear} is $\mathcal{O}\big(|V_I|^2 \cdot 2k\mathcal{H}\big)$. Suppose it is the turn of the reachability player, upon fixing the first component of the vertex, the other three components are directly determined. Therefore, the number of edges in G_I^{max} is $|E_I|^2$. □

4.3 Extracting the Optimal Schedule

We present a technique to extract the optimal schedule corresponding to any memoryless winning strategy σ for the reachability player. Consider the scenario where the safety player uses the same memoryless winning strategy σ. Let ρ denote the play corresponding to this scenario. Algorithm 3 presents the procedure to construct this play ρ and extract the optimal schedule.

Algorithm 3. Extracting the optimal schedule.

Input memoryless winning strategy σ.
Output optimal schedule S.

$v \leftarrow (v_{init}, v_{init}, 0, r)$
$\rho \leftarrow \varnothing$
while $v \neq (v_f, *, 0, s)$ **do**
  if fourth component of $v = r$ **then**
    $\rho \leftarrow \rho \cup \{v\}$
  end if
  $v \leftarrow \sigma(v)$
end while
apply Algorithm 2 to ρ to retrieve S
return S

4.4 Minimizing Total Load and Makespan

The *completion time* of an agent is the time when the agent finishes all of its assigned tasks. *Total load* is defined as the sum of the completion times of all the agents [5]. To minimize the total load, we modify the counter in construction from Sect. 4; it now has k components, one for each agent. In the counter corresponding to agent a, we maintain the difference between the latest time when agent a is not idle across the two schedules (corresponding to the moves of the reachability player and the safety player). Since the smallest value of completion time is zero and the greatest value of completion time is $\mathcal{H}$, this difference is contained in the closed interval $[-\mathcal{H}, \mathcal{H}]$. Thus, the counter takes values from this interval.

Makespan is the total length of the schedule, i.e., the maximum value among the completion times of the agents. Makespan minimization is another common optimization criterion in the literature [1]. For minimizing the makespan, we modify the counter in the construction from Sect. 4. The counter takes values from the interval $[-1, \mathcal{H}]$. In the counter, we maintain the difference between the

latest time when all the agents are idle across the two schedules (corresponding to the moves of the reachability player and the safety player). In Table 4, we present the counters for each type of optimization criteria.

Table 4. The number of components in the counter and the range of each component corresponding to the optimization criterion.

Optimization type	#components	Range of each component
Minimize makespan	1	$[-1, \mathcal{H}]$
Linear function	1	$[-c\mathcal{H}, c\mathcal{H}]$
Minimize total load	k	$[-\mathcal{H}, \mathcal{H}]$

5 Experimental Evaluation

In this section, we validate the reachability game approach for finding optimal schedules using 1) a case study for scheduling tasks for astronauts aboard the International Space Station and 2) randomized experiments for different optimization criteria.

5.1 Qualitative Evaluation

We solve a scheduling problem for six astronauts aboard the International Space Station (ISS). The astronauts have to perform a set $\mathcal{T}$ of lab tasks with variable durations. Due to power requirements, the astronauts can perform only three lab tasks at any time. Additionally, to stay healthy, the astronauts have to a) eat, b) use a treadmill and c) lift weights. An astronaut cannot exercise after she eats. Additionally, we also model a scenario where some of the astronauts are unhealthy (they may be injured or ill). We penalize schedules that use unhealthy astronauts. Thus, we assign an unhealthy astronaut a weight of -1 and assign all others a weight of 1.

Table 5. Time(s) for computing a valid schedule and an optimal schedule.

Number (n) of tasks	Valid	Optimal
10	0.07	0.179
20	0.08	0.191
30	0.18	0.592
40	0.20	0.608
50	3.56	4.450
60	3.62	4.515

Let $\mathcal{A}_h$ denote the set of healthy astronauts. We compute schedules that are optimal with respect to the linear function $\sum_{a \in \mathcal{A}_h} value_S(a) - \sum_{a \notin \mathcal{A}_h} value_S(a)$. For the experiments, we fix the window length Δ as 20 and the planning horizon $\mathcal{H}$ as 300. The time in seconds for computing both a valid schedule and an optimal schedule versus the number n of lab tasks is presented in Table 5.

Since only three lab tasks can be performed at any time, the worst-case complexity for finding an optimal schedule is reached when around 3Δ tasks ($n = 50$) have to be scheduled.

5.2 Randomized Evaluation and Comparison with an Integer Programming Formulation

For each optimization criterion, we generate random problem instances and record the time for synthesizing an optimal schedule for these instances. We fix the number of agents as five and assume that all the agents can perform all

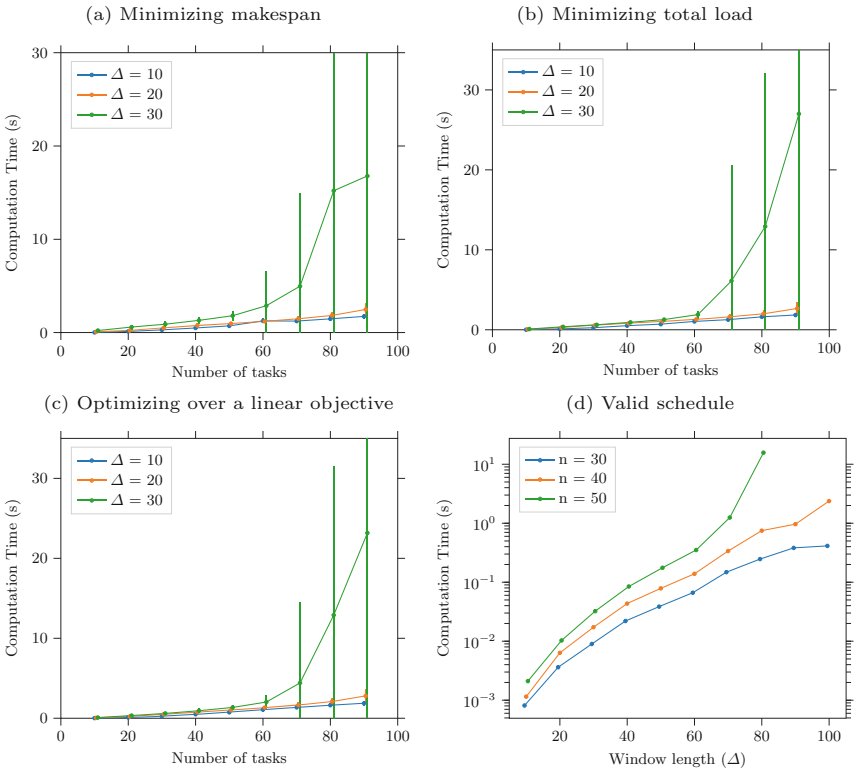

Fig. 1. (a)–(c) provide average running time for computing an optimal schedule for the corresponding optimization criterion. (d) provides average running time for computing a valid schedule as a function of window length.

the tasks. We fix the planning horizon $\mathcal{H} = 200$. We vary the window length Δ and the number n of tasks and observe its impact on the running time.

For each value of Δ and n, we generate 100 random instances and record the running time as the average over the running times of these 100 instances. In total, we run 10800 experiments to generate the graphs presented in Fig. 1. We performed the experiments on an Ubuntu 18.04 system with an Intel i7-8550U (1.80 GHz) processor and 16 GB memory.

The experiments show that we can compute optimal schedules in less than a minute for up to 100 tasks when $\Delta = 30$. After we fix the values of Δ and $\mathcal{H}$, if the number n of tasks is small compared to $\mathcal{H}$, the tasks are distributed sparsely across the windows. In this case, the $\mathcal{H}$ term dominates the complexity of finding an optimal schedule. However, as we increase n (until its upper bound $\mathcal{O}(k\mathcal{H})$), the density of tasks in each window increases. As a result, the $2^{4k\Delta}$ term dominates the complexity of finding an optimal schedule. This observation is consistent with Remark 2.

Finally, we compare the reachability game technique against an integer programming encoding for the ARCPSP problem. We use Gurobi [10], a state-of-the-art mixed-integer linear programming (MILP) solver for solving the integer programming formulation. Figure 2 contains the results of this comparison. The experiments show that the reachability game technique is at least two orders of magnitude faster than the integer program that we run on Gurobi.

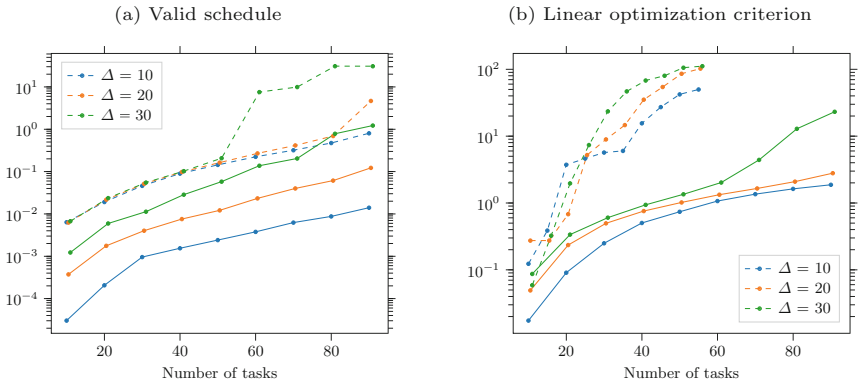

Fig. 2. Average running times for computing (a) a valid schedule and (b) an optimal schedule with respect to a linear optimization criterion using the reachability game formulation versus an IP encoding run in Gurobi. The solid lines correspond to run-times obtained by using the reachability game technique and the dashed lines correspond to the Gurobi implementation.

6 Conclusion

We identified a new parameter called *window length* for the agent resource-constrained project scheduling problem (ARCPSP). Using this parameter, we provide a novel algorithm for finding optimal schedules that scales polynomially in the number of tasks as long as the window length is a fixed constant. We illustrate the applicability of this method by solving a scheduling problem for astronauts aboard the International Space Station (ISS). Furthermore, a direct comparison with an integer program formulation that we run in Gurobi shows that this technique is at least two orders of magnitude faster.

References

1. Artigues, C., Demassey, S., Neron, E.: Resource-Constrained Project Scheduling: Models, Algorithms. Extensions and Applications, ISTE (2007)
2. Chatterjee, K.: Graph games with reachability objectives. In: Delzanno, G., Potapov, I. (eds.) RP 2011. LNCS, vol. 6945, p. 1. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24288-5_1
3. Cheng, T., Ding, Q., Lin, B.: A concise survey of scheduling with time-dependent processing times. Eur. J. Oper. Res. **152**(1), 1–13 (2004)
4. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. IEEE, October 1991
5. Gawiejnowicz, S.: Time-Dependent Scheduling. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69446-5
6. Gawiejnowicz, S., Lee, W.-C., Lin, C.-L., Wu, C.-C.: Single-machine scheduling of proportionally deteriorating jobs by two agents. J. Oper. Res. Soc. **62**(11), 1983–1991 (2011)
7. Geeraerts, G., Goossens, J., Nguyen, T.-V.-A.: A backward algorithm for the multiprocessor online feasibility of sporadic tasks. In: 2017 17th International Conference on Application of Concurrency to System Design (ACSD), pp. 116–125, June 2017
8. Geeraerts, G., Goossens, J., Nguyen, T.-V.-A., Stainer, A.: Synthesising succinct strategies in safety games with an application to real-time scheduling. Theor. Comput. Sci. **735**, 24–49 (2018)
9. Grädel, E., Thomas, W., Wilke, T.: Automata, Logics, and Infinite Games - A Guide to Current Research. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36387-4
10. Gurobi Optimization: L. Gurobi optimizer reference manual (2020)
11. Hughes, M.: Why projects fail: the effect of ignoring the obvious. Ind. Eng. **18**, 14–18 (1986)
12. Kononov, A.: Scheduling problems with linear increasing processing times. Operations Research Proceedings 1996, pp. 208–212. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-60744-8_38
13. Larman, C.: Agile and Iterative Development. Addison-Wesley, Boston (2004)
14. Lauffer, N.T., Topcu, U.: Human-understandable explanations of infeasibility for resource-constrained scheduling problems. In: Workshop on Explainable Planning (XAIP 2019) (2019)

15. Lombardi, M., Milano, M.: A precedence constraint posting approach for the RCPSP with time lags and variable durations. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 569–583. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_45
16. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Logic **65**(2), 149–184 (1993)
17. Mosheiov, G.: Multi-machine scheduling with linear deterioration. INFOR: Inf. Syst. Oper. Res. **36**(4), 205–214 (1998)
18. Mostowski, A.W.: Games with Forbidden Positions. UG (1991)
19. Neumann, K., Schwindt, C., Zimmermann, J.: Resource-constrained project scheduling with time windows. In: Józefowska, J., Weglarz, J. (eds.) Perspectives in Modern Project Scheduling. ISOR, pp. 375–407. Springer, Heidelberg (2006). https://doi.org/10.1007/978-0-387-33768-5_15
20. Radermacher, F.J.: Scheduling of project networks. Ann. Oper. Res. **4**(1), 227–252 (1985)
21. Stork, F., Uetz, M.: Enumeration of circuits and minimal forbidden sets. Electron. Notes Discret. Math. **13**, 108–111 (2003)