Dynamic Project Expediting: A Stochastic Shortest-Path Approach

Luca Bertazzi,[a] Riccardo Mogre,[b,*] Nikolaos Trichakis[c]

[a] Department of Economics and Management, University of Brescia, 25122 Brescia, Italy; [b] Durham University Business School, Durham University, Durham DH1 3LB, United Kingdom; [c] Operations Research Center and Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142
*Corresponding author
Contact: luca.bertazzi@unibs.it, https://orcid.org/0000-0002-0227-9135 (LB); riccardo.mogre@durham.ac.uk,
https://orcid.org/0000-0001-5280-0518 (RM); ntrichakis@mit.edu, https://orcid.org/0000-0002-8324-9148 (NT)

Abstract. We deal with the problem of managing a project or a complex operational process by controlling the execution pace of the activities it comprises. We consider a setting in which these activities are clearly defined, are subject to precedence constraints, and progress randomly. We formulate a discrete-time, infinite-horizon Markov decision process in which the manager reviews progress in each period and decides which activities to expedite to balance expediting costs with delay costs. We derive structural properties for this dynamic project expediting problem. These enable us then to devise exact solution methods that we show to reduce computational burden significantly. We illustrate how our method generalizes and can be used to tackle a wide range of so-called stochastic shortest-path problems that are characterized by an intuitive property and can capture other applications, including medical decision-making and disease-modeling problems. Moreover, we also deal with the state identification issue for our problem, which is a challenging task in and of itself, owing to precedence constraints. We complement our analytical results with numerical experiments, demonstrating that both our solution and state identification methods significantly outperform extant methods for a supply chain example and for various randomly generated instances.

1. Introduction

1.1. Motivation

Managing projects, or complex operational processes, in general, is a notoriously difficult problem. In their "Pulse of the Profession" studies, the Project Management Institute reported that, in the last decade, only about 50% of projects were on time, only 55% were within budget, and that firms wasted, on average, more than 11% of total investment due to poor project performance (PMI 2018, 2020). Among the main underlying challenges, and therefore also presenting opportunities, are ineffective monitoring, forecasting, and control (Pinto and Mantel 1990). Monitoring and forecasting entail efficient ways to measure and predict progress. Control entails timely and appropriate corrective actions based on the monitored progress, such as, for example, resource deployment to expedite progress—a procedure known as *dynamic expediting* (Bregman 2009). Recent innovations and advances in automation, digitization, and

artificial intelligence (AI) have drastically catalyzed effective monitoring and forecasting, as we illustrate below via a couple of examples, laying fertile ground for control processes, like dynamic expediting, to have an impact in practice. Motivated by this opportunity and envisioning that future project management will rely ever more heavily on rigorous quantitative methods, we seek to devise optimal dynamic expediting policies that are computationally efficient so that they are implementable and can help improve the management of complex operational processes in practice.

First, consider supply chain operations, like the ones undertaken by specialist firm Li & Fung, for example, to produce garments (Magretta 1998). In such a chain, yarn, bought from Korea, is woven and dyed in Taiwan, while at the same time, zippers are manufactured by Japanese contractors in their Chinese plants. As supply chain tasks are orchestrated, various sensors dispersed from manufacturing machines to transit vehicles could

inform Li & Fung managers in real time about the progress of each task, alerting them to potential delays. Such delays are of course exceedingly costly due to lost demand. Based on this monitoring information, managers have the ability to deploy resources in a smart way to expedite progress, for example, to contract additional suppliers or capacity to ramp up production or to switch to different transport modes to speed up transshipments.

Second, consider a complex pharma project, like BioNTech's repurposing of the Marburg biologics facility, for example, that it bought from Novartis in fall of 2020 so that it can produce BNT162, the company's mRNA COVID-19 vaccine (BioNTech 2021). Although the facility was previously used by Novartis' vaccine division, several conversion tasks needed to be undertaken for it to handle the manufacturing needs of BNT162, which relied on the novel mRNA platform. These included machine conversion tasks, such as calibration of bioreactors for mRNA production and adjustments of purification processes, as well as workforce training tasks. As these tasks are undertaken, AI models could help provide managers in real time with forecasts about potential delays based on current progress, notwithstanding that some of these tasks are unprecedented. Delays could again result in costly lost demand but also loss of life as the pandemic continues to spread. Based on delay forecasts, managers have again the ability to deploy resources in a smart way to expedite progress, for example, to buy extra equipment or to employ additional staff specialists.

The infrastructure for effective forecasting and monitoring of processes is hardly unique to the previous two examples. As in the case of Li & Fung, Industry 4.0 practices are becoming widespread with equipment increasingly using sensors and Internet-of-Things technology to provide real-time monitoring. As in the case of BioNTech, equipped with historical data, deep learning and AI can identify patterns of similarity among project tasks, hierarchies, and precedent relations, enabling progress forecasting for complex processes. For instance, nPlan, a London-based startup, is using AI and data from tens of thousands of projects involving millions of tasks to generate accurate forecasts for project completion, including information about delay risks (Grushka-Cockayne 2020). Indeed, be it a manufacturing task, a supply chain process, or a complex project, monitoring and forecasting capabilities for operations management are becoming increasingly powerful.

To leverage these novel capabilities, implementable control policies are called for to "close the loop" and go from data to decisions, paving the way for even more efficient and dynamic management of operating processes. In this paper, we study one such important class of control policies for project management, namely dynamic project expediting, that can be applied to a broad range of problems. In particular, we assume that there is a project with clearly defined activities, which the manager can effectively monitor, as was described in the aforementioned practical applications. We strive to derive control policies that, based on the monitoring information, prescribe corrective actions in terms of when and which processes to expedite, balancing costs associated with expediting activities and project delays.

The types of control policies we research are not simply interesting and relevant because of the opportunities presented by effective monitoring discussed so far, they are also necessary to deal with the increasing complexity of projects and operating processes (De Meyer et al. 2006). Indeed, with complexity rising, it becomes exceedingly harder to predict how changes in the effort of some activities will affect other activities and ultimately the outcome of the project (Sterman 1992). Consequently, in practice, dynamic expediting decisions are often dealt with in an unstructured manner (ProjectManagementdotcom 2017). In response, rigorous control policies are called for.

At the same time, project complexity also increases the scale of the underlying control problems, and in turn the required computational burden. Therefore, control policies have to be computationally efficient to be practical, and this is an important consideration in our work.

1.2. Our Study

We aim to provide support to decision makers who are managing a set of processes and need to dynamically decide when and which processes to expedite, balancing expediting costs and delay costs. In particular, in our problem, a decision maker is in charge of a project with clearly defined activities, some sequential and others parallel, subject to certain precedence constraints. Motivated by the technological advancements discussed above, we consider a setting in which the decision maker is able to periodically monitor the progress of activities. The nominal duration of each activity is known, but progress is uncertain as is usually the case in practice. Motivated by data availability as discussed previously, we consider a setting in which the decision maker has a credible probabilistic description of progress uncertainty for each activity.

As illustrated in the aforementioned supply chain and pharma project examples, the decision maker has some control over the progress of the project activities, and, in particular, has the option of expediting them. To model this, we consider that there is a level of effort exerted for each activity, which the decision maker chooses in each period. For example, for each activity she could decide whether to exert regular effort or to exert high effort, that is, to expedite the activity. Higher levels of effort "speed up" an activity's random progress evolution, but also incur higher effort costs.

As also illustrated in the supply chain and pharma project examples, the longer the project takes to complete, the higher the associated delay costs are, for example, due to lost demand. To model this, we consider a delay cost that grows linearly with the project completion time.

In this setting, the problem for the decision maker is to dynamically decide in each period the level of effort to be exerted in each activity of the project, based on observed progress. The decision maker's objective is to minimize the sum of expected future effort and delay costs. We refer to this problem as the *dynamic project expediting (DPE)* problem. We formulate the DPE problem as a discrete-time Markov decision process (MDP), and we seek a nonanticipatory policy that prescribes effort levels for each activity, based on observed progress.

1.3. Our Contributions

The DPE problem we study belongs to a class of MDP problems known as stochastic shortest-path (SSP) problems. The tools currently available in the extant literature to solve such problems are primarily based on generic dynamic programming techniques: value iteration, policy iteration, or via linear programming (Bertsekas 2017). These techniques are exact, but it is well known that they quickly become computationally prohibitive to use in practice as the state space grows. Indeed, for the case of the supply chain managed by Li & Fung we discussed previously, for example, we found in our computational experiments that these techniques fail.

Against this backdrop, among our main contributions is to develop and analyze an exact solution method, which we call the *topological backward recursion (TBR)* algorithm. The *TBR* algorithm yields optimal policies for the DPE problem in a significantly less computationally burdensome way. We also show that our algorithm can be applied not only to our DPE problem but also to a large class of stochastic shortest-path problems that are characterized by an intuitive property.

In particular, we first derive insightful structural properties that the DPE problem possesses. These structural properties enable a topological ordering of the state space, which in turn enables us then to develop *TBR*. Furthermore, the analysis brings forth a broad class of SSP problems that share similar ordering properties. We call problems that belong to this class as *forward-only SSP problems*, and as we shall see, besides DPE, they arise often in practice, particularly within medical decision-making applications. We show that the *TBR* algorithm works not only for the DPE problem but for this larger class of forward-only SSP problems.

Then, we show that our *TBR* algorithm significantly dominates the existing value iteration, policy iteration, and linear programming algorithms in terms of the computational burden. We do this by providing both

analytical and numerical evidence. For the former, we first demonstrate how the *TBR* leverages the DPE's structural properties we established so that it can calculate the optimal cost-to-go in a single iteration for each recursion step. Second, we argue that it also requires an amount of data storage that is proportional to $s/2$ on average, where s is the cardinality of the state space, because in each iteration it accesses exclusively the states, the transition probabilities, and the immediate costs needed for that iteration. In contrast, extant algorithms typically employ expensive iterative procedures to calculate costs-to-go and require an amount of data storage that is proportional to s^2—because they store all the states, all the transition probabilities, and all the immediate costs. Put together, these two features help explain the computational benefits of *TBR*, while the drastic reduction in data storage proves key to solving problem instances that are of significantly larger scale in practice, and for which extant algorithms fail.

A further contribution we make is also in the direction of alleviating overall computational burden, by introducing a faster method to identifying feasible states for the DPE problem. Feasible state identification, which is a prerequisite step to applying our *TBR* algorithm, or any other extant dynamic programming algorithm for that matter, is a rather challenging task for the DPE problem in and of itself, owing to the precedence constraints that activities have to satisfy. To tackle this task, we follow a similar recipe as before: we first identify a key structural property, namely, an equivalence relation on the states that partitions the feasible space into classes. This property motivates a class identification (*CId*) algorithm we introduce. We then leverage the output partition of the *CId* algorithm to identify feasible states through a state enumeration (*SEn*) algorithm. Taken together, these algorithms provide a faster way to identify feasible states compared with extant methods—a claim we back up with extensive numerical evidence.

Finally, we conduct a series of numerical experiments to demonstrate the practical value of our algorithms. We begin by considering the Li&Fung supply chain example. In our tests, we found that our algorithms succeed in solving the problem, whereas extant methods fail as they run out of memory. We further conduct a range of experiments using synthetically generated data. We again find that using our algorithm enables us to solve much larger problem instances. For smaller instances that are also solvable using extant methods, we document a significant reduction in computational time: in particular, compared with a value iteration implementation tailored to stochastic shortest-path problems on directed acyclic graphs (Bertsekas 2012, pp. 209–210), which was the best performing among the dynamic programming algorithms in our experiments, the *TBR* algorithm reduces computation times on average by 42% and as much as 100.0%. In

terms of feasible state identification, compared with the fast class identification algorithm proposed by Creemers et al. (2010), the average percent reduction of *CId* was 43% and the maximum percent reduction was 99%.

To be sure, the algorithms we devise still scale with the size of the state space, thus the curse of dimensionality would still kick in when applied in practice. After all, the DPE remains a quite challenging stochastic optimization problem. However, the analytical and numerical evidence we have provided demonstrates that our algorithms can greatly expand the sizes of problems we can solve in practice. More importantly, the structural properties we uncover provide useful insights that could help pave the way to devising further lines of attack to dynamic project expediting problems and to the broader class of forward-only stochastic shortest-path problems.

Next, in Section 2, we position our contributions vis-a-vis the existing literature and review relevant previous contributions. In Section 3, we provide a formulation of the DPE problem as a stochastic shortest-path problem. In Section 4, we introduce the *TBR* algorithm, whereas, in Section 5, we tackle the feasible state identification problem, introducing our *CId* and *SEn* algorithms. Section 6 includes our numerical studies that compare the *TBR* and *CId + SEn* algorithms to existing algorithms. In Section 7, we conclude and discuss directions for future research.

2. Related Work

In a nutshell, we contribute to the sparse literature on dynamic project expediting by describing a new expediting problem, proposing an MDP formulation for the problem, deriving insightful structural properties, and devising algorithms that can solve the problem optimally in a significantly less computationally burdensome way compared with extant dynamic programming algorithms. More broadly, our work contributes to the literature on stochastic shortest-path problems (Bertsekas and Tsitsiklis 1991) by bringing forth, for a class of problems that we call forward-only SSP problems, an important ordering property, which we then use to devise more efficient solution techniques. Besides the DPE problem, forward-only SSP problems subsume other important practical problems, specifically within medical decision-making and disease-modeling applications. Moreover, we contribute to the literature on dynamic project management by proposing algorithms that (1) can identify activities that can be performed in parallel and (2) can enumerate the feasible states of a dynamic project management problem in a way that appears to be significantly faster compared with extant methods.

We next present a more detailed review, by first discussing previous literature that is thematically related to our problem, namely on dynamic project expediting, dynamic project management, and project crashing. Second, we focus on the methods used in previous literature and compare them with the ones used in this article.

2.1. Dynamic Project Expediting

Literature on dynamic expediting includes contributions in a variety of settings, primarily in inventory management, but also in manufacturing management. Very few studies analyze dynamic expediting in projects.

Bregman (2009) is among the first who study dynamic project expediting. In their context, expediting consists of the allocation of extra resources to a project activity to accelerate its progress. For a project with uncertain activity durations, they investigate the dynamic selection of expediting options and aim to reduce the additional resource cost under the constraint of completing the project before the due date. To solve this problem, they propose a heuristic solution procedure that measures the probability of completing the project before the due date by using a matrix-simulation approach.

For a project with uncertain activity times, Godinho and Branco (2012) investigate dynamic scheduling and expediting policies. They restrict their attention to threshold policies based on activity starting times that they use to determine expediting policies. To identify scheduling and expediting policies, they propose an electromagnetic heuristic procedure.

Like Bregman (2009) and Godinho and Branco (2012), we aim to identify dynamic expediting policies for a project with uncertain activity progress. Different from them, we seek to solve the problem using exact algorithms, in the context of an MDP formulation.

2.2. Dynamic Project Management

Few more studies investigate the more general problem of dynamically managing a project in presence of uncertainty. One of the earliest contributions in this area is Jørgensen and Wallace (2000), which call for more stochastic dynamic models for the allocation of resources, at least for budgeting purposes.

For a project with exponentially distributed activity times, Sobel et al. (2009) investigate how to dynamically choose the optimal start date of each activity. In their MDP formulation, they optimize the expected present value of the project cash flow. They show that their formulation corresponds to an optimization problem on a continuous-time Markov decision chain (CTDMC) and propose an algorithm to compute an optimal dynamic policy.

Klastorin and Mitchell (2013) analyze a project which is under the threat of a disruptive event that can stop all activities at the same time. They assume that the decision maker reviews the status of the project periodically and learns for how long the activities stop if a disruptive event

happens. In this setting, they study the time-cost tradeoff problem, in which the decision maker aims to minimize the sum of a tardiness penalty and a resource deployment cost by choosing, for each activity, between a regular time—cost combination and a faster, more expensive, crash time—cost combination. They propose an MDP formulation for the problem and devise an algorithm to solve it. Moreover, they show that, in comparison with a base case without a disruptive event, the decision maker should consider additional deployment of resources only at the beginning of the project or just after the occurrence of the disruption.

For a project with uncertain activity durations, Li and Womer (2015) investigate the dynamic allocation of resources to the activities of the project. In their MDP formulation, they aim to minimize the total project duration. To solve this problem, they develop approximate dynamic programming (ADP) algorithms based on a roll-out policy. Instead, our focus is on deriving exact solution methods for a somewhat similar problem. That is, our methodological contribution includes the solution of our MDP formulation with an exact, computationally efficient algorithm.

Like Sobel et al. (2009), Klastorin and Mitchell (2013), and Li and Womer (2015), we propose an MDP formulation for our dynamic project management problem. However, the focus of these studies is on scheduling and resource allocation. Instead, we aim to derive dynamic expediting policies.

The enumeration of the states for our problem requires the identification of the activities that can be processed in parallel at any given time. Sobel et al. (2009) tackle the same problem. They propose a simple algorithm that identifies more states than feasible ones. Creemers et al. (2010), who build on Sobel et al. (2009), define the set of activities that can be processed in parallel as uniformly directed cutsets (UDC), a term which was introduced by Sigal et al. (1980). They propose an efficient tree-based algorithm to identify the project UDCs. We also propose a very efficient tree-based algorithm to identify all the classes of states for a project, which are equivalent to UDCs. Our numerical studies demonstrate that our algorithm is significantly faster to compute.

Finally, Sobel et al. (2009) and Creemers et al. (2010) aim to identify the optimal start date for each activity. They show that it is optimal to restrict the choice of start dates to time epochs when activities are completed. With the additional assumption that activity times are exponentially distributed, their formulation leads to an optimization problem on a CTDMC. Different from them, we investigate how much effort should be allocated to activities. Moreover, we consider partial completion of activities, because it could be optimal to change the effort allocated to an activity while it is in progress.

2.3. Project Crashing

The DPE problem is also closely related to the project crashing problem, where crashing is an action that spends a certain amount of resources and shortens the duration of the activity accordingly. This problem is rooted in the project management literature including the seminal works by Kelley (1961) and Fulkerson (1961), among others.

The stochastic project crashing problem aims to identify a schedule for a project with uncertain progress that minimizes the sum of a cost associated with the project duration and a cost that increases in the deployment of the resources. As these costs are in tradeoff with each other, these problems are also called stochastic time-cost tradeoff problems.

Stochastic project crashing problems are especially complicated because the critical path of the network, which is used to estimate the total duration of the project, is not unique and varies with the realizations of the random durations of the activities in the project. For this reason, studies in this stream of literature approach this problem using heuristics rather than exact methods. Gutjahr et al. (2000) use a stochastic branch-and-bound approach to solve the stochastic discrete crashing problem. Laslo (2003) extends the project crashing problem by considering a continuum of time–cost combinations. Azaron et al. (2007) solve a goal-programming formulation of the discrete-time approximation of the stochastic continuous project crashing problem. Mitchell and Klastorin (2007) devise a tailor-made compression heuristic to solve the stochastic continuous project crashing problem. Shen et al. (2010) consider a variant of the project crashing problem in which the project manager can insure activities instead of allocating extra resources to the project; they provide decomposition strategies for solving this problem. Goh and Hall (2013) use robust optimization to solve the stochastic continuous project crashing problem; they derive rules for starting and crashing activities that can be used not only to define baseline schedules but also to revise schedules during execution by applying static optimization iteratively in a rolling-horizon setting.

Our project expediting problem is similar to the project crashing problem, in that it aims to minimize the sum of two types of costs in trade-off with each other: project duration costs and expediting costs. However, our problem differs from the project crashing problem, in that its output is not a baseline schedule but rather a list of expedited activities at each stage at which the project is reviewed.

2.4. Shortest-Path Problems

We also show that our problem is a specific type of MDP, called the stochastic shortest-path (SSP) problem. One of the first studies that investigate SSP problems is Bertsekas and Tsitsiklis (1991). We observe that our SSP

problem has some special ordering properties, which are shared by other problems. We define the problems with such properties as "forward-only SSP problems." Our contribution to the literature on SSP problems is to elicit these important ordering properties and use them to construct a more efficient, exact algorithm for forward-only SSP problems.

Moreover, our study contributes to the literature on stochastic shortest problems on directed acyclic graphs (DAGs). For deterministic shortest-path problems on DAGs, effective algorithms have been devised (Cormen et al. 2009, pp. 655–658). However, we are not aware of any results or any ways these results could be modified to obtain similarly effective algorithms for stochastic shortest-path problems. One of the most efficient implementations has been suggested by Bertsekas (2012, pp. 209–210), who propose an algorithm tailored to stochastic shortest-path problems on DAGs combining the traditional value iteration algorithm and the Dijkstra algorithm, which was developed for deterministic shortest-path problems. In our computational experiments, we show that our *TBR* algorithm is substantially faster than this algorithm, which we use as a benchmark.

3. Problem Formulation

In this section, we introduce the model that we study for the dynamic project expediting problem. To this end, we first provide a high-level description of the problem and introduce a network structure to characterize the project's activities. We then formalize the problem as a Markov decision process, describing its states, controls, probabilities, and costs.

We consider a discrete-time horizon over which a decision maker is in charge of a project with clearly defined activities of finite time length, some sequential and others parallel, all of which are required for the completion of the project. Once an activity is started, its progress is random. In each time period $t \in \{0, 1, \dots\}$, the decision maker tracks the progress on each activity of the project. In practice, the amount of time between two consecutive periods (i.e., between the start of one period and the start of the next one) could be in the order of seconds, hours, or days, depending on the tracking frequency and the specific application. Just after tracking the progress of all activities, the decision maker chooses the level of effort to be invested in each activity in the following time period. Higher levels of effort yield higher expected progress, but are also associated with higher effort costs. Besides effort costs, there is also a cost that depends on the project completion time. The problem for the decision maker is to derive a nonanticipatory dynamic expediting policy that minimizes her expected costs, the sum of effort costs and project completion costs.

Figure 1. AoN Representation of a Sample Project
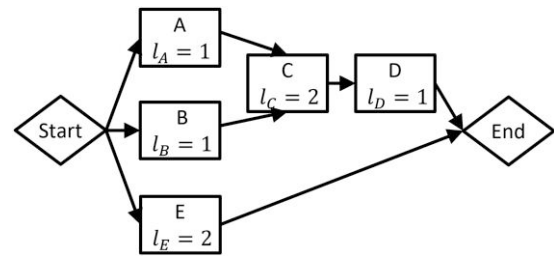
3.1. Network of Activities

To describe the activities of the project, we model it as a network using the so-called activity-on-node (AoN) diagramming technique. Figure 1 provides an example of a sample project. In particular, the AoN network of a project is a directed acyclic graph with one source, the "Start" node, and one sink, the "End" node. A directed acyclic graph (DAG) is a directed graph with no cycle. We denote a network or a graph by $G(\mathcal{A}, \mathcal{E})$. $\mathcal{A}$ is the set of nodes $1, 2, \dots, n$, which represent the activities. $\mathcal{E}$ is the set of arcs (oriented edges), $1, 2, \dots, m$, representing the precedence relations. We denote the nodes or activities by a and the arcs by e. For an activity a, we define the set of all its predecessors, a subset of $\mathcal{A}$, by Π_a. Activity a' belongs to Π_a if there exists at least one path from a' to a. Similarly, we define the set of all the successors of a, also a subset of $\mathcal{A}$, by $\mathcal{S}_a$. Activity a'' belongs to $\mathcal{S}_a$ if there exists at least one path from a to a''. The length l_a of activity a is the number of time periods, such as days or weeks, within which the manager expects the activity to be completed if a regular effort is used.

3.2. States

The progress on activity a, expressed in terms of the number of time periods' worth of work actually done, is x_a. If activity a could be started but there has been no progress on it, we set x_a to zero. If activity a cannot be started because of precedence constraints, we set x_a to -1. Therefore, x_a can take the values $-1, 0, 1, \dots, l_a$. We say that an activity a is *engaged* if $x_a \neq -1$, and that it is $un-engageable$ otherwise. We say that an engaged activity is *in progress* if $x_a \neq l_a$ and that it is *completed* otherwise. The state of the system is the vector $\mathbf{x} = (x_1, x_2, \dots, x_n)$ that lists the progress on each activity. Because of precedence constraints, not every vector $\mathbf{x}$ corresponds to a feasible state. We define a feasible state as follows.

Definition 1. A *feasible state* is an n-dimensional vector $\mathbf{x}$ such that, for each of its components, which corresponds to a specific activity a, the following hold:

1. If a is in progress, then all $a' \in \Pi_a$ are completed and all $a'' \in \mathcal{S}_a$ are unengageable.

2. If a is unengageable, then not all $a' \in \Pi_a$ are completed and all $a'' \in \mathcal{S}_a$ are unengageable.

Table 1. States for the Sample Project

	States	
$1:[0,0,-1,-1,0]$	$8:[1,0,-1,-1,1]$	$15:[1,1,1,-1,2]$
$2:[0,0,-1,-1,1]$	$9:[1,0,-1,-1,2]$	$16:[1,1,2,0,0]$
$3:[0,0,-1,-1,2]$	$10:[1,1,0,-1,0]$	$17:[1,1,2,0,1]$
$4:[0,1,-1,-1,0]$	$11:[1,1,0,-1,1]$	$18:[1,1,2,0,2]$
$5:[0,1,-1,-1,1]$	$12:[1,1,0,-1,2]$	$19:[1,1,2,1,0]$
$6:[0,1,-1,-1,2]$	$13:[1,1,1,-1,0]$	$20:[1,1,2,1,1]$
$7:[1,0,-1,-1,0]$	$14:[1,1,1,-1,1]$	$21:[1,1,2,1,2]$

3. If a is completed, then all $a' \in \Pi_a$ are completed and for all $a'' \in S_a$:

   If all elements of $\Pi_{a''}$ are completed, a'' is engaged;

   otherwise, a'' is unengageable.

Let $\mathcal{X}$ be the feasible state space. Identifying $\mathcal{X}$ is a challenging problem—For now, let us take $\mathcal{X}$ as given, and we shall deal with the state identification problem in Section 5. We denote the realization of the state in period t by $\mathbf{X}_t$. Initially, the state of the project lists 0 for the activities with the source as starting node and -1 for all other activities. We refer to this state as the initial state and denote it with $\mathbf{i}$. That is, we have that $\mathbf{X}_0 = \mathbf{i}$. The terminal or final state is $\mathbf{f} = (l_1, l_2, \ldots, l_n)$.

To exemplify, consider the sample project from Figure 1. In Table 1, we show the 21 feasible states for our sample project. For example, for this sample project, if, in period t, activities C and E are engaged, there is no cumulative progress on activity C, and there is one unit of cumulative progress on activity E, then $\mathbf{X}_t = (1, 1, 0, -1, 1)$.

3.3. Controls

A control vector, which we shall denote with $\mathbf{k}$, lists the level of effort k_a to use for activity a; $k_a = 0$ corresponds to no effort and $k_a = 1, \ldots, K$ denote increasing effort levels. For example, for $K = 2$, the effort levels are zero, one, and two, which could correspond to no effort, regular effort, and expediting, respectively. For the sample project, a control that expedites activity C and uses regular effort for activity E is $\mathbf{k} = [0, 0, 2, 0, 1]$.

Associated with each activity a and effort level k_a, is an effort cost c_{a,k_a}. These effort costs are increasing in k_a, and $c_{a,k_a} = 0$ for $k_a = 0$. For example, for costs linear in the effort levels, we have $c_{a,k_a} = q_a k_a$, with $q_a > 0$ the cost per unit of effort for activity a. The effort cost $c_{\mathbf{k}}$ that corresponds to control $\mathbf{k}$ is $c_{\mathbf{k}} = \sum_a c_{a,k_a}$.

Associated with each state $\mathbf{x}$ is a set of feasible control vectors, denoted with $\mathcal{K}(\mathbf{x})$, which could capture certain constraints that the controls need to satisfy. For example, the manager may have to satisfy a budget constraint b on the cost $c_{\mathbf{k}}$ in each period, effectively reducing the control space $\mathcal{K}(\mathbf{x})$, that is, $c_{\mathbf{k}} > b$ implies $\mathbf{k} \notin \mathcal{K}(\mathbf{x})$ for each state $\mathbf{x}$.

Consider now a function μ_t that maps each state to a single control at time t. That is, at period t, the control $\mu_t(\mathbf{X}_t)$ is applied. A *policy* is a collection $\{\mu_0, \mu_1, \mu_2, \ldots\}$. We denote a stationary policy by the collection $\mu = \{\mu, \mu, \mu, \ldots\}$.

3.4. Transition Probabilities

We model the progress on activity a at a time period in which the activity is in state x_a and the manager uses effort k_a with the discrete random variable $W^a_{x_a, k_a}$ that takes values in $[0, l_a - x_a]$. The greater the effort level used, the greater, in expectation, the progress toward activity completion. More formally, if $W^a_{x_a, k'_a}$ and $W^a_{x_a, k''_a}$ are two discrete random variables with $k'_a < k''_a$, we have the following relation for their expected values: $\mathbb{E}(W^a_{x_a, k''_a}) \geq \mathbb{E}(W^a_{x_a, k'_a})$.

When the state of activity a is x_a and the manager uses effort k_a, the progress of activity a transitions to the state $y_a = x_a + W^a_{x_a, k_a}$. The probability of transitioning from x_a to y_a using k_a is denoted with $P^a_{x_a, y_a}(k_a)$. In general, precedence constraints make the dependence of $P^a_{x_a, y_a}(k_a)$ on the probabilistic description of $W^a_{x_a, k_a}$ somewhat complex, so in the appendix, we explain how to calculate $P^a_{x_a, y_a}(k_a)$ in all possible cases.

For analytical tractability, we assume that the progress on each activity is independent of time and of the progress on the other activities. In practice, the latter is realistic for projects in which teams do not multitask. Let $\mathbf{x} \in \mathcal{X}$ and $\mathbf{y} \in \mathcal{X}$ denote two generic states. Assuming independence among activities, the transition probability from $\mathbf{x}$ to $\mathbf{y}$ using $\mathbf{k}$ is

$$P_{\mathbf{x}, \mathbf{y}}(\mathbf{k}) = P^1_{x_1, y_1}(k_1) \cdot P^2_{x_2, y_2}(k_2) \cdot \ldots \cdot P^n_{x_n, y_n}(k_n).$$

If $P^a_{x_a, y_a}(k_a) = 0$ for at least one activity a, no transition from $\mathbf{x}$ to $\mathbf{y}$ is feasible.

3.5. Costs

In period t, the manager observes the state $\mathbf{X}_t$ and decides the control $\mathbf{k} \in \mathcal{K}(\mathbf{X}_t)$ to use from t to $t+1$ bearing the effort cost $c_{\mathbf{k}}$. Additionally, for each time period t at the end of which the project is still not completed, that is, $\mathbf{X}_{t+1} \neq \mathbf{f}$, the manager bears a cost, which could represent a penalty for waiting, that is, due to benefits that could have been achieved if the project was completed. Let u denote this per-period waiting cost. For tractability, we implicitly assumed that costs are stationary; that is, they do not depend on time. The total cost in a period starting at state $\mathbf{x}$ and arriving at state $\mathbf{y}$ while using control $\mathbf{k}$ is then

$$g(\mathbf{x}, \mathbf{k}, \mathbf{y}) = \begin{cases} c_{\mathbf{k}} + u, & \text{if } \mathbf{x} \neq \mathbf{f} \text{ and } \mathbf{y} \neq \mathbf{f}; \\ c_{\mathbf{k}}, & \text{if } \mathbf{x} \neq \mathbf{f} \text{ and } \mathbf{y} = \mathbf{f}; \\ 0, & \text{if } \mathbf{x} = \mathbf{f} \text{ and } \mathbf{y} = \mathbf{f}. \end{cases}$$

The expected total cost in a period starting at state $\mathbf{x}$ and using control $\mathbf{k}$ is then

$$\overline{g}(\mathbf{x}, \mathbf{k}) = \sum_{\mathbf{y} \in \mathcal{X}} P_{\mathbf{x}, \mathbf{y}}(\mathbf{k}) g(\mathbf{x}, \mathbf{k}, \mathbf{y}).$$

When applying policy $\{\mu_t\}$, the cost-to-go at some time t is then given by

$$\lim_{T \to \infty} \mathbb{E} \left\{ \sum_{\tau=t}^{T-1} \overline{g}(\mathbf{X}_\tau, \mu_\tau(\mathbf{X}_\tau)) \right\}. \qquad (1)$$

Let $J(\mathbf{i})$ be the cost-to-go at time 0. The DPE problem that we study involves finding a policy that minimizes this cost.

3.6. Discussion
Now that we formally introduced the DPE, it is worth discussing the assumptions and generalizability of our model. In short, despite the specific application of project expediting and certain modeling assumptions that we made, the problem we are tackling can still capture a large subset of an important class of problems, as we discuss in the analysis section next.

To be more concrete, we assumed stationary costs and no discounting. Although these might appear limiting at first glance, note that the important and broad class of SSP problems also adopt these modeling choices. Within project management, there could be a plethora of different cost structures in practice. Indeed, some instances might not be compatible with the model, we study, but there remains a large swath of problems that fit very well, among others, projects that are not years long, for example, and feature final completion time costs that scale approximately linear in the time taken to complete the project.

In the analysis that follows, we first present our solution technique tailored to the DPE formulation we described. We then discuss how it can be applied to a large subset of the important class of SSP problems, which we call forward-only, characterized by an intuitive property.

4. Analysis
We start this section by positioning the DPE problem as an SSP problem and we discuss why extant solution approaches could be impractical to use for real-life-sized instances. Then, we introduce a key structural property of our problem, namely a strict partial order relation on the state space. We show that it is possible to obtain a linear order of the state space from this partial order by using a topological sorting algorithm. Next, we devise an algorithm, using the linearly ordered state space, that identifies the optimal solution for our problem. It does so in a more efficient way than existing algorithms. Finally, we show that our algorithm can be applied not only to our DPE problem but also to a large class of stochastic shortest-path problems that are characterized by an intuitive property.

4.1. SSP Problem
The lack of a discounting factor and the presence of the cost-free terminal state $\mathbf{f}$ categorize the DPE problem as an SSP problem (Definition 2).

Definition 2 (Bertsekas 2017, p. 233). A cost-minimization problem of an infinite-horizon MDP formulation is a stochastic shortest-path problem if (1) no discount factor is present and (2) there is a special cost-free terminal state.

We use the fact that our problem is a stochastic shortest-path problem to show that there exists a cost-minimizing stationary policy μ^* for Equation (1).

Lemma 1. *For the stochastic shortest-path problem that aims to minimize the cost in Equation* (1), *there exists an optimal stationary policy* μ^*.

We can reformulate our problem using Lemma 1 as follows: The project manager aims to find a stationary policy μ that minimizes the cost-to-go:

$$J_\mu(\mathbf{i}) = \lim_{T \to \infty} \mathbb{E} \left\{ \sum_{\tau=0}^{T-1} \overline{g}(\mathbf{X}_\tau, \mu(\mathbf{X}_\tau)) \,\middle|\, \mathbf{X}_0 = \mathbf{i} \right\}. \qquad (2)$$

We denote such an optimal policy with μ^*.

4.2. Existing Algorithms
Extant methods to solve SSP problems, including the DPE problem, are applications of value iteration, policy iteration, and linear programming algorithms. They are based on Bellman's equation, an optimality principle for MDP problems, which for our problem is as follows:

$$J^*(\mathbf{x})$$
$$= \begin{cases} \min_{\mathbf{k} \in \mathcal{K}(\mathbf{x})} \left[\overline{g}(\mathbf{x}, \mathbf{k}) + \sum_{\mathbf{y} \in \mathcal{X}} P_{\mathbf{x}, \mathbf{y}}(\mathbf{k}) J^*(\mathbf{y}) \right] & \text{if } \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\} \\ 0 & \text{if } \mathbf{x} = \mathbf{f}, \end{cases}$$
$$\qquad (3)$$

where J^* maps each feasible state to the anticipated minimum cost-to-go starting from that state. We present the pseudo-codes of the value iteration, policy iteration, and linear programming algorithms for our problem in the Appendix; a more detailed description can be found in Bertsekas (2017, pp. 245–249). The results of computational experiments presented in Section 6 show that the use of these algorithms to identify the optimal policy for our problem is impractical. Their computation times rapidly increase with the size of the network and the number of activities in parallel. This motivated our work to identify more computationally efficient solution methods.

4.3. Partial and Linear Orders on the State Space
We now introduce a key structural property of the DPE problem. In particular, we shall show how to obtain a linear order on the state space in two steps: the first and important step establishes how we can represent the states using a DAG, whereas the second step simply

applies the known result that every DAG has a topological order of all its vertices.

We provide the definition of a linear order on a generic set H as follows.

Definition 3. Let a, b, and c be elements of a set H. A binary relation $\leq$ on H is a linear order if it has the following properties:

1. Antisymmetry: If $a \leq b$ and $b \leq a$, then $a = b$.
2. Transitivity: If $a \leq b$ and $b \leq c$, then $a \leq c$.
3. Connex property: $a \leq b$ or $b \leq a$.

Consider state $17 : [1, 1, 2, 0, 1]$ and state $19 : [1, 1, 2, 1, 0]$ from Table 1, for example. These two states are not comparable. Lacking the connex property, a "natural" linear order relation on the state space $\mathcal{X}$ does not exist. Therefore, we look for a partial order relation on the state space, and for a strict partial order relation in particular. Despite its name, a strict partial order relation is an order relation that is less restrictive than a linear order relation. We provide the definition of a strict partial order on a generic set H as follows, where the symbol $\neg$ denotes logical negation.

Definition 4 (Lehman et al. 2017, p. 400). Let a, b, and c be elements of a set H. A binary relation $<$ on H is a strict partial order if it has the following properties, with asymmetry implied by irreflexivity and transitivity.

1. Irreflexivity: $\neg(a < a)$;
2. Transitivity: if $a < b$, and $b < c$, then $a < c$;
3. Asymmetry: if $a < b$, then $\neg(b < a)$.

Intuitively, state $\mathbf{x}$ precedes state $\mathbf{y}$ if and only if the project progress in state $\mathbf{x}$ is less than that in state $\mathbf{y}$. We formalize this binary relation in Definition 5.

Definition 5. For $\mathbf{x}, \mathbf{y} \in \mathcal{X}$, we say that $\mathbf{x} \prec \mathbf{y}$ if the following hold:

1. $x_a \leq y_a$ for all activities a that are engaged in $\mathbf{x}$, and $x_{a'} < y_{a'}$ for at least one of these activities.
2. $x_a \leq y_a$ for all activities a that are unengageable in $\mathbf{x}$, and $x_a < y_a$ if and only if all predecessors of the starting node of a are completed in $\mathbf{y}$.

We highlight an important fact that links the relation $\prec$ to the probability $P_{\mathbf{x}, \mathbf{y}}(\mathbf{k})$ in Remark 1.

Remark 1. If $\mathbf{x}, \mathbf{y} \in \mathcal{X}$ and $\mathbf{x} \prec \mathbf{y}$, then $P_{\mathbf{y}, \mathbf{x}}(\mathbf{k}) = 0$ for every $\mathbf{k} \in \mathcal{K}(\mathbf{x})$.

Remark 1 follows from the definition of $P_{\mathbf{x}, \mathbf{y}}(\mathbf{k})$ in Section 3. Intuitively, it says that it is not possible to move from a state $\mathbf{y}$ to a state $\mathbf{x}$ if the progress in $\mathbf{x}$ is less than the progress in $\mathbf{y}$.

The relation $\prec$ is the "natural" order relation on the state space and provides a strict partial order of the states (Lemma 2).

Lemma 2. *The binary relation $\prec$ is a strict partial order on $\mathcal{X}$.*

We are ready to supply a graphical characterization of $\prec$. Consider the directed acyclic graph $\mathcal{G}^{\prec}(\mathcal{V}^{\prec}, \mathcal{E}^{\prec})$, constructed as follows:

1. We include a node v in $\mathcal{V}^{\prec}$ for every state $\mathbf{x} \in \mathcal{X}$.
2. We include an arc between node v_1 corresponding to state $\mathbf{x}$ and node v_2 corresponding to state $\mathbf{y}$ if and only if $\mathbf{x} \prec \mathbf{y}$.

The number of arcs in the graph $\mathcal{G}^{\prec}(\mathcal{V}^{\prec}, \mathcal{E}^{\prec})$ is in general large, even for small problems. For this reason, it is convenient to introduce a graph with the same number of nodes as $\mathcal{G}^{\prec}(\mathcal{V}^{\prec}, \mathcal{E}^{\prec})$ but with a smaller set of arcs. The relation "a is covered by b" on a generic set H, which is defined as follows, is instrumental to the introduction of this graph.

Definition 6. Let a, b, and c be elements of a set H with a strict partial order relation $<$. We say that "a is covered by b" if and only if $a < b$ and $\neg(a < c < b)$ for all $c \in H$.

We can also provide a definition of what it means to say that a state $\mathbf{x}$ is covered by $\mathbf{y}$ based on the strict partial order $\prec$. Intuitively, state $\mathbf{x}$ is covered by state $\mathbf{y}$ if and only if the progress in $\mathbf{y}$ is the progress in $\mathbf{x}$ increased by one unit. This guarantees that there are no other states between $\mathbf{x}$ and $\mathbf{y}$.

Definition 7. For $\mathbf{x}, \mathbf{y} \in \mathcal{X}$, we say that $\mathbf{x}$ is covered by $\mathbf{y}$ if and only if the following hold:

1. There is one activity a' in progress in $\mathbf{x}$ such that $x_{a'} = y_{a'} - 1$ and $x_a = y_a$ for all activities $a \neq a'$ that are in progress in $\mathbf{x}$.
2. $x_a = y_a$ for all activities a that are completed in $\mathbf{x}$.
3. $x_a \leq y_a$ for all activities a that are unengageable in $\mathbf{x}$, where $x_a < y_a$ holds if and only if all predecessors of the starting node of a are completed in $\mathbf{y}$.

We are ready to supply a graphical characterization of the relation $\mathbf{x}$ is covered by $\mathbf{y}$. Consider the directed acyclic graph $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$, constructed as follows:

1. We include a node v in $\mathcal{V}'$ for every state $\mathbf{x} \in \mathcal{X}$.
2. We include an arc between node v_1 corresponding to state $\mathbf{x}$ and node v_2 corresponding to state $\mathbf{y}$ if and only if $\mathbf{x}$ is covered by $\mathbf{y}$.

We call $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ the Hasse diagram of $\prec$. Formally, $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ is the transitive reduction of $\mathcal{G}^{\prec}(\mathcal{V}^{\prec}, \mathcal{E}^{\prec})$, as detailed in the proof of Lemma 3. We present in Figure 2 the Hasse diagram for the states in Table 1. State 1 is the source of the graph and the initial state $\mathbf{i}$. State 21 is the sink of the graph and the final state $\mathbf{f}$. Both states are depicted in gray in Figure 2.

The binary relation R^+ allows us to formalize the relationship between $\prec$ and its Hasse diagram. We provide the definition of R^+ for a generic directed acyclic graph $\mathcal{G}$ as follows:

Definition 8. Let v_1 and v_2 be nodes of a directed acyclic graph $\mathcal{G}$. We say that $v_1 R^+ v_2$ if and only if v_1 and v_2 are connected by a walk of positive length.

We note that R^+ is transitive and irreflexive, and therefore asymmetric. Moreover, R^+ is equivalent to $\prec$, as made clear in Lemma 3.

Figure 2. Hasse Diagram for the Sample Project
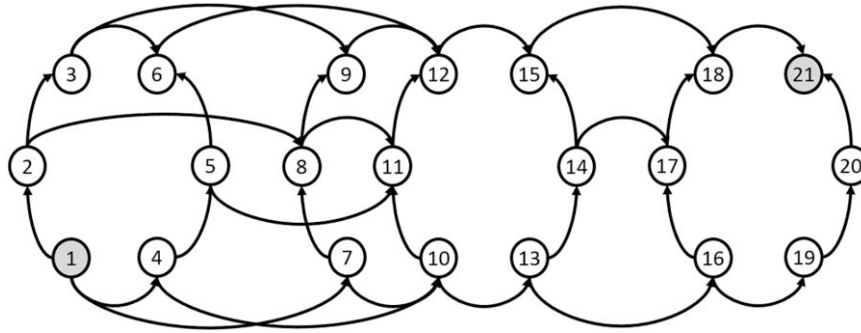
Lemma 3. *The binary relation $\prec$ is equivalent to the positive-walk relation R^+ on its Hasse diagram $\mathcal{G}'(\mathcal{V}', \mathcal{A}')$.*

From the diagram in Figure 2, it is clear that $12 : [1, 1, 0, -1, 2] > 7 : [1, 0, -1, -1, 0]$, because there is at least one path in the graph from state 7 to state 12 (for example, through states 8 and 11). Moreover, it is clear that $\neg(6 : [0, 1, -1, -1, 2] > 7 : [1, 0, -1, -1, 0])$, because there is no path in the graph from state 7 to state 6. A topological sort is an ordered list of nodes obtained from a finite directed acyclic graph (Definition 9).

Definition 9 (Lehman et al. 2017, 393). Let $\mathcal{G}$ be a finite directed acyclic graph. A topological sort of $\mathcal{G}$ is a list of all nodes such that each node v appears earlier in the list than all the nodes connected from v.

We can now leverage previous results on finite directed acyclic graphs to argue that we can obtain a linear order of the state space $\mathcal{X}$. Theorem 10.5.4 in Lehman et al. (2017, p. 395) or proposition 1.4.3 in Jensen and Gutin (2007) show that every finite directed acyclic graph has a topological sort. The Hasse diagram $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ is finite. Therefore, its topological sort is a list of nodes $v \in \mathcal{V}'$. Every node v corresponds to a state $\mathbf{x}$; hence, we obtain a linear order on the state space $\mathcal{X}$.

Remark 2. The Hasse diagram $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ has a topological sort which defines a linear order of the state space $\mathcal{X}$.

We denote the cardinality of the state space $\mathcal{X}$ by s. We also denoted the linearly ordered state space by $\widetilde{\mathcal{X}} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_s\}$. The initial state is always the first element of $\widetilde{\mathcal{X}}$, that is, $\mathbf{x}_1 = \mathbf{i}$. The final state is always the last element of $\widetilde{\mathcal{X}}$, that is, $\mathbf{x}_s = \mathbf{f}$. A topological sort of a directed acyclic graph needs not be unique. We can use the depth-first search (DFS) algorithm, as described in Cormen et al. (2009, pp. 612–614), to obtain a topological sort of a DAG.

4.4. TBR Algorithm
We now formally introduce our solution approach, which we refer to as the *TBR* algorithm. It uses the linearly

ordered state space $\widetilde{\mathcal{X}}$ to identify the optimal solution for our problem. We present the pseudo-code for the *TBR* algorithm as Algorithm 1.

Algorithm 1 (TBR Algorithm)
> **begin**
> $\quad \widetilde{\mathcal{X}} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_s\};$
> $\quad J^*(\mathbf{x}_s) \triangleq J^*(\mathbf{f}) = 0;$
> **end**
> **for** $i = s - 1, s - 2, \ldots, 1$ **do**
> $\quad$ Compute $J^*(\mathbf{x}_i) = \min_{\mathbf{k} \in \mathcal{K}(\mathbf{x}_i)} \frac{\overline{g}(\mathbf{x}_i, \mathbf{k}) + \sum_{j=i+1}^{s-1} P_{\mathbf{x}_i, \mathbf{x}_j}(\mathbf{k}) J^*(\mathbf{x}_j)}{1 - P_{\mathbf{x}_i, \mathbf{x}_i}(\mathbf{k})};$
> **end**
> **return** $J^*(\mathbf{x}_i)$ *and* $k^*(\mathbf{x}_i)$ *for* $i = s - 1, s - 2, \ldots, 1$

The *TBR* algorithm calculates the optimal cost-to-go function using a backward recursion that iterates on the topologically sorted state space. The algorithm only requires a single computation for each state as opposed to existing dynamic programming algorithms such as value iteration, which requires multiple iterations of the algorithm for each state—The interested reader could also contrast *TBR*'s pseudo-code with that of value iteration, presented in the appendix as Algorithm A.1. This important feature already provides an explanation for the computational benefits that can be obtained using the *TBR* algorithm.

To shed more light on how *TBR* achieves the calculation of the optimal cost-to-go function at each state using a single iteration, consider the following technical lemma.

Lemma 4. *Consider $k \in \mathcal{K}$, where $\mathcal{K}$ is a finite set. Let $a_k > 0$ and $0 < \delta_k < 1$. Then $\hat{y} = \frac{a_{k^*}}{1 - \delta_{k^*}}$, with $k^* = \arg\min_k \frac{a_k}{1 - \delta_k}$, solves the equation*

$$y = \min_k [a_k + \delta_k y]. \tag{4}$$

Now, let us apply the previous lemma with the following choice of parameters y, a_k, and δ_k:
1. $y = J^*(\mathbf{x}_i)$;
2. $a_k = \overline{g}(\mathbf{x}_i, \mathbf{k}) + \sum_{j=i+1}^{s-1} P_{\mathbf{x}_i, \mathbf{x}_j}(\mathbf{k}) J^*(\mathbf{x}_j)$;
3. $\delta_k = P_{\mathbf{x}_i, \mathbf{x}_i}(\mathbf{k})$.

With this choice, and by leveraging the topological sort of the state space that we established, Equation (4)

corresponds to the Bellman equation for our problem, applied for the *i*th state. Therefore, Lemma 4 then suggests that to evaluate the optimal cost-to-go function at that state, namely $J^*(\mathbf{x}_i)$, we require the optimal cost-to-go values only at states $i + 1, \ldots, s$, which, in a backward recursion process, would be already calculated. The *TBR* algorithm leverages this structure to obtain $J^*(\mathbf{x}_i)$ with a single calculation. In contrast, to calculate $J^*(\mathbf{x}_i)$, classical applications of the Bellman equation require knowledge of $J^*(\mathbf{x}_i)$ as well, and therefore require iterative procedures to solve.

The above application of Lemma 4 can also be used to prove the optimality of the *TBR* algorithm.

Theorem 1. *The TBR algorithm finds an optimal policy for our problem.*

With the *TBR* algorithm formally introduced, we are now in a position to make two important points. First, we argue that the *TBR* algorithm can be applied not merely to the DPE problem, but to a large class of SSP problems. Second, we discuss further and shed more light on why the *TBR* algorithm outperforms all extant solution methods by significantly reducing the required computational burden.

4.5. Forward-Only SSP Problems
The *TBR* algorithm identifies an optimal policy for a more general class of problems, which we term forward-only SSP problems. We first provide a formal definition, followed by an intuitive interpretation.

Definition 10. The forward-only stochastic shortest-path problem is a stochastic shortest-path problem (Definition 2) in which the following hold:

1. There exists a strict partial order on the state space.

2. For any two states *a* and *b* of the problem, if *a* precedes *b* in the strict partial order, then the probability of moving from *b* to *a* is zero.

Intuitively, the forward-only property asks that most states are transient and that state transitioning is governed by some progression process. The latter means that there is some dominant progression pattern that states follow, as is, for example, the progression of a project toward the completion within the DPE. Other examples could include disease management when dealing with diseases that worsen over time, such as end-stage renal disease, cardiac allograft vasculopathy, Alzheimer's, or osteoarthritis to name a few. Thus, our analysis and the *TBR* algorithm could be useful in a wide range of medical decision-making and disease-modeling applications.

In particular, for a forward-only stochastic shortest-path problem, we can obtain a linearly ordered state space by applying the same procedure described in this section. Moreover, the proof of Theorem 1 requires only

the assumptions stated in Definition 10. Therefore, we can state a corollary of Theorem 1 without proof.

Corollary 1. *The TBR algorithm finds an optimal policy for any forward-only stochastic shortest-path problem.*

4.6. Comparison with Existing Algorithms
The computational benefits of the *TBR* algorithm stem from two important points. The first point relates to the elimination of the need for an iterative procedure in the calculation of the optimal cost-to-go. That is, as discussed previously, the topological sort of the state space we established enables one to calculate the optimal cost-to-go function for each state using a single iteration in each step of the backward recursion.

Second, the *TBR* deals with data that are linear in the size of the state space, as opposed to quadratic. In particular, extant implementations of the value iteration algorithm, the policy iteration algorithm, and the linear programming algorithm need memory access to all the transition probabilities and the costs at all times. Thus, these algorithms store an amount of data proportional to s^2 for each control. In contrast, the *TBR* algorithm stores an amount of data proportional to one in the first iteration and to the number of states *s* in the last iteration. To see this, note that the *TBR* can be applied without the need to generate all the transition probabilities and the costs beforehand; at each iteration, it computes only the transition probabilities and the costs needed for that iteration. Importantly, this is only possible because of the particular structure of the *TBR* algorithm and it is unclear how to leverage this approach in implementations of extant algorithms. Consequently, the amount of data stored by the *TBR* is proportional to $s/2$ on average for each control. This drastic reduction in memory usage and access not only yields a significant speed-up in the computational time but also critically reduces memory requirements.

In Section 6, we conduct several numerical experiments that verify the anticipated gains in computational performance provided by the *TBR* algorithm.

5. State Identification
Recall that up to this point we have assumed that the set of feasible states is given. As we discuss later, it turns out that identifying this set for the DPE problem is a rather challenging task in and of itself. Given that this set is an input to the *TBR* algorithm, or to any of the classical extant dynamic programming approaches for that matter, it is essential to have access to a "smart" way of obtaining it.

To this end, we shall devise a process to identify all feasible states. We follow the same recipe as before: we first identify an important structural property, namely, we introduce an equivalence relation on states that partitions the feasible state space into classes. Next, we

introduce two algorithms, termed *CId* and *SEn*, that we devised, leveraging this partitioning property, to enumerate classes and states, respectively. In the next section, we pit our algorithms against existing state identification approaches to illustrate the important computational gains that they offer.

5.1. State-Space Identification Problem

Identifying the feasible state space $\mathcal{X}$ is challenging, primarily because of precedence constraints, but also because we consider partial completion of activities. A possible approach to tackling this problem could be the complete enumeration of the n-dimensional vectors over the n activities, followed by the removal of vectors that are not feasible states. This approach is impractical because of the large number of vectors it would generate that are unfeasible states. For example, complete enumeration for the sample project in Figure 1 leads to $3 \cdot 3 \cdot 4 \cdot 3 \cdot 4 = 432$ vectors of which only 21 are feasible states. For this reason, we propose two algorithms that identify the feasible state space $\mathcal{X}$ directly. This approach is in general more efficient than the complete enumeration of vectors over the activities, followed by the removal of vectors that are not feasible states. Our algorithms enumerate the feasible states from classes of states.

5.2. Classes of States

A class of states is a subset of activities that can be processed in parallel, also known in the literature as UDCs. We induce the class of states by an equivalence relation. We provide the relation on the states in Definition 11. Lemma 5 shows that this relation is an equivalence relation that partitions the feasible state space into classes.

Definition 11. We say that $\mathbf{x} \sim \mathbf{y}$ if the following hold:
  1. If x_a is engaged, y_a is also engaged.
  2. If x_a is unengageable, y_a is also unengageable.

Lemma 5. *The relation $\sim$ is an equivalence relation that partitions the feasible state space $\mathcal{X}$ into classes of states.*

We use $\hat{\mathcal{X}}_1, \hat{\mathcal{X}}_2, \ldots$ to denote the classes of states induced on the feasible state space $\mathcal{X}$ by the equivalence relation of Definition 11. An extended notation for class $\widehat{\mathcal{X}}_i$ is an n-dimensional vector that, for each a, lists the components corresponding to a as follows:
  1. It is -1 if a is unengageable.
  2. It is $\hat{x}_a$ if there exists a state in $\widehat{\mathcal{X}}_i$ with $x_a < l_a$.
  3. It is l_a if $x_a = l_a$ for all states in $\widehat{\mathcal{X}}_i$.
Every class of states has a subgraph representation. From the class $\widehat{\mathcal{X}}_i$, we obtain its subgraph representation $\Gamma(\hat{\mathcal{A}}_i, \hat{\mathcal{E}}_i)$ as follows:
  1. $\hat{\mathcal{A}}_i \subset \mathcal{A}$ includes the activities a whose component in $\widehat{\mathcal{X}}_i$ is $\hat{x}_a$;
  2. $\hat{\mathcal{E}}_i \subset \mathcal{E}$ includes all the arcs that connect activities $a \in \hat{\mathcal{A}}_i$.

As a consequence of Lemma 5, we can identify the state space in two distinct phases. In the first phase, the class identification (*CId*) algorithm identifies all classes of states for a project network. In the second phase, the state enumeration (*SEn*) algorithm enumerates all states from each class.

5.3. *CId* Algorithm

The *CId* algorithm is a recursive algorithm that generates the subgraph representations of the classes of states from the network $G(\mathcal{A}, \mathcal{E})$. The algorithm uses a collection H of visited graphs and a collection L that will contain all the classes generated. Recall that Π_a and $\mathcal{S}_a$ are the sets of all predecessors and all successors, respectively, of activity a. We define $\bar{\mathcal{A}}_a$ as follows: $\bar{\mathcal{A}}_a = \hat{\mathcal{A}} \setminus \{\Pi_a \cup \mathcal{S}_a\}$. We denote by $\Gamma(\bar{\mathcal{A}}_a, \bar{\mathcal{E}}_a)$ the graph where $\bar{\mathcal{E}}_a$ is the set of arcs that connect activities $a' \in \bar{\mathcal{A}}_a$. We present the pseudo-code for the *CId* algorithm as Algorithm 2.

Algorithm 2 (CId Algorithm)

> **Algorithm** CId()
>> $L \leftarrow \emptyset$;
>> $H \leftarrow \emptyset$;
>> build $G(\mathcal{A}, \mathcal{E})$;
>> call CIdvisit($G(\mathcal{A}, \mathcal{E})$, L, H);
>> **return** L;
>
> **Procedure** CIdvisit($\Gamma(\hat{\mathcal{A}}, \hat{\mathcal{E}})$, L, H)
>> $H = H \cup \{\Gamma(\hat{\mathcal{A}}, \hat{\mathcal{E}})\}$;
>> for all $a \in \hat{\mathcal{A}}$, compute Π_a and $\mathcal{S}_a$;
>> **if** $\Pi_a = \emptyset$ *for all $a \in \hat{\mathcal{A}}$* **then**
>>> compute the class $\widehat{\mathcal{X}}$ corresponding to $\Gamma(\hat{\mathcal{A}}, \hat{\mathcal{E}})$;
>>> $L = L \cup \{\widehat{\mathcal{X}}\}$;
>>
>> **else**
>>> for all $a \in \hat{\mathcal{A}}$, compute $\Pi_a \cup \mathcal{S}_a$ and $\bar{\mathcal{A}}_a = \hat{\mathcal{A}} \setminus \{\Pi_a \cup \mathcal{S}_a\}$;
>>> **for** *each a such that $\Pi_a \cup \mathcal{S}_a \neq \emptyset$* **do**
>>>> build $\Gamma(\bar{\mathcal{A}}_a, \bar{\mathcal{E}}_a)$, where $\bar{\mathcal{E}}_a$ corresponds to $\bar{\mathcal{A}}_a$;
>>>> **if** $\Gamma(\bar{\mathcal{A}}_a, \bar{\mathcal{E}}_a) \notin H$ **then**
>>>>> call CIdvisit($\Gamma(\bar{\mathcal{A}}_a, \bar{\mathcal{E}}_a)$, L, H);
>>>>
>>>> **end**
>>>
>>> **end**
>>
>> **end**
> **end**

The *CId* algorithm is based on the observation that, for any activity, its predecessors cannot be processed in parallel with its successors. We illustrate the application of the *CId* algorithm to the sample project of Figure 1.

In the *CId* algorithm, we initialize the collections L and H as empty sets, and we build the graph $G(\mathcal{A}, \mathcal{E})$ with $\mathcal{A} = \{A, B, C, D, E\}$ depicted in Figure 1. Then we call the recursive procedure *CIdvisit* on $G(\mathcal{A}, \mathcal{E})$. When we call the *CIdvisit* procedure on $G(\mathcal{A}, \mathcal{E})$, we add the graph to the list H of visited graphs to avoid examining

Table 2. *CId* Algorithm Calculations for the Sample Project

a	Π_a	$\mathcal{S}_a$	$\bar{\mathcal{A}}_a = \hat{\mathcal{A}} \setminus \{\Pi_a \cup \mathcal{S}_a\}$
A	$\emptyset$	C,D	A,B,E
B	$\emptyset$	C,D	A,B,E
C	A,B	D	C,E
D	A,B,C	$\emptyset$	D,E
E	$\emptyset$	$\emptyset$	A,B,C,D,E

the graph again in the future. In Table 2, we present the details for Π_a, $\mathcal{S}_a$, and $\bar{\mathcal{A}}_a$ for each activity a in $G(\mathcal{A}, \mathcal{E})$.

We proceed to compute the subgraphs $\Gamma(\bar{\mathcal{A}}_a, \bar{\mathcal{E}}_a)$ because $\Pi_a \neq \emptyset$ for activities C and D. We obtain the three subgraphs depicted in Figure 3 for the activities a such that $\Pi_a \cup \mathcal{S}_a \neq \emptyset$.

Then we call the *CIdvisit* procedure on these graphs. In all three graphs, $\Pi_a = \emptyset$ for all the activities. Therefore, we add the corresponding classes of states to L. Using the lengths of the activities from Figure 1, we obtain the classes as follows:

$$\widehat{\mathcal{X}}_1 = (\hat{x}_A, \hat{x}_B, -1, -1, \hat{x}_E);$$
$$\widehat{\mathcal{X}}_2 = (1, 1, \hat{x}_C, -1, \hat{x}_E);$$
$$\widehat{\mathcal{X}}_3 = (1, 1, 2, \hat{x}_D, \hat{x}_E).$$

Finally, the *CId* algorithm returns the list $L = \{\widehat{\mathcal{X}}_1, \widehat{\mathcal{X}}_2, \widehat{\mathcal{X}}_3\}$; the *SEn* algorithm will enumerate the feasible states from these three classes. Toward the end of this section, we provide proof of correctness for the *CId* algorithm, that is, that it succeeds for any problem instance.

Creemers et al. (2010) proposed a fast algorithm to identify all subsets of activities that can be processed in parallel (UDCs). This algorithm generates a tree in which each node is an activity. Also, the *CId* can be viewed as a tree-generation algorithm in which each node is a subgraph. In Section 6, we compare the computation times of our *CId* algorithm against those of our implementation of the one suggested by Creemers et al. (2010) and find that our algorithm provides significant reductions.

5.4. *SEn* Algorithm

The *SEn* algorithm generates the feasible states for each class. The collection of states $\widehat{\mathcal{X}}_i$ will include the states generated. For each class, the algorithm calls the recursive procedure *SEnvisit*. At each iteration, the procedure *SEnvisit* takes an activity a and a vector u as input.

The vector u represents a partial state, that is, a state whose elements are included up to activity $a - 1$. The procedure stores the partial states in a temporary collection T. For each value taken by the corresponding state x_a, the procedure creates a temporary vector v, sets $v = u$, and adds the value of x_a to the vector. In other words, the vectors are built up coordinate by coordinate. If a vector does not meet the precedence constraints, it is immediately discarded. We present the pseudo-code for the *SEn* algorithm as Algorithm 3.

Algorithm 3 (SEn Algorithm)

> **Algorithm** SEn()
>> $\mathcal{X} \leftarrow \emptyset$;
>> **for** *all classes* $\widehat{\mathcal{X}}_i \in L$ **do**
>>> create empty vector u;
>>> call SEnvisit(u, 0, $\widehat{\mathcal{X}}_i$, $\mathcal{X}$);
>>
>> **end**
>> **return** $\mathcal{X}$;
>
> **Procedure** SEnvisit(u, a, $\widehat{\mathcal{X}}_i$, $\mathcal{X}$)
>> $T \leftarrow \emptyset$;
>> **if** $\widehat{\mathcal{X}}_i[a] \neq -1$ *or* $\widehat{\mathcal{X}}_i[a] \neq l_a$ **then**
>>> **for** *each i such that* $0 \leq i \leq l_a$ **do**
>>>> create vector v and set $v = u$;
>>>> add i as the next element of v;
>>>
>>> **end**
>>> **if** $a \neq n$ **then**
>>>> $T = T \cup \{v\}$;
>>>
>>> **else**
>>>> $\mathcal{X} = \mathcal{X} \cup \{v\}$;
>>>
>>> **end**
>>
>> **else**
>>> create vector v and set $v = u$;
>>> add $\widehat{\mathcal{X}}_i[a]$ as the next element of v;
>>> **if** *v satisfies the precedence constraints* **then**
>>>> **if** $a \neq n$ **then**
>>>>> $T = T \cup \{v\}$;
>>>>
>>>> **else**
>>>>> $\mathcal{X} = \mathcal{X} \cup \{v\}$;
>>>>
>>>> **end**
>>>
>>> **end**
>>
>> **end**
>> **for** *all* $v \in T$ **do**
>>> call SEnvisit(u, $a + 1$, $\widehat{\mathcal{X}}_i$, $\mathcal{X}$);
>>
>> **end**

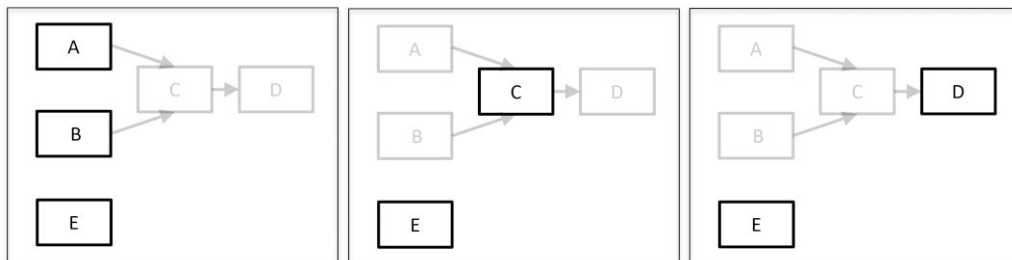**Figure 3.** Subgraphs for the Sample Project

Table 3. *SEn* Algorithm Results for the Sample Project

$\widehat{\mathcal{X}}_1 =$ $(\hat{x}_A, \hat{x}_B, -1, -1, \hat{x}_E)$	$\widehat{\mathcal{X}}_2 =$ $(1, 1, \hat{x}_C, -1, \hat{x}_E)$	$\widehat{\mathcal{X}}_3 =$ $(1, 1, 2, \hat{x}_D, \hat{x}_E)$
$1 : [0, 0, -1, -1, 0]$	$10 : [1, 1, 0, -1, 0]$	$16 : [1, 1, 2, 0, 0]$
$2 : [0, 0, -1, -1, 1]$	$11 : [1, 1, 0, -1, 1]$	$17 : [1, 1, 2, 0, 1]$
$3 : [0, 0, -1, -1, 2]$	$12 : [1, 1, 0, -1, 2]$	$18 : [1, 1, 2, 0, 2]$
$4 : [0, 1, -1, -1, 0]$	$13 : [1, 1, 1, -1, 0]$	$19 : [1, 1, 2, 1, 0]$
$5 : [0, 1, -1, -1, 1]$	$14 : [1, 1, 1, -1, 1]$	$20 : [1, 1, 2, 1, 1]$
$6 : [0, 1, -1, -1, 2]$	$15 : [1, 1, 1, -1, 2]$	$21 : [1, 1, 2, 1, 2]$
$7 : [1, 0, -1, -1, 0]$		
$8 : [1, 0, -1, -1, 1]$		
$9 : [1, 0, -1, -1, 2]$		

In Table 3 we present, for each class, the feasible states generated by applying the *CId* algorithm to our example. The state space $\mathcal{X}$ for our example has 21 states.

5.5. Proof of Correctness

Theorem 2 summarizes the main theoretical results for algorithms *CId* and *SEn*.

Theorem 2. *For any project network $G(\mathcal{V}, \mathcal{A})$ with $a \in \mathcal{A}$ of length l_a:*

1. The CId algorithm enumerates all the equivalence classes $\widehat{\mathcal{X}}_i$ of feasible states.

2. The SEn algorithm enumerates all the feasible states $\mathbf{x}$ in each equivalence class.

Lemma 5 shows that $\mathcal{X} = \cup_i \widehat{\mathcal{X}}_i$. Therefore, the application of *CId* and *SEn* in succession will find all the feasible states $\mathbf{x} \in \mathcal{X}$ for the problem.

6. Computational Studies

In this section, we present numerical studies aimed at demonstrating the advantages of using our algorithms compared with existing approaches. As a short summary, our experiments showcase the significant computational benefits of our algorithms in terms of computational time, with reductions averaging 40% and being as much as 99% across numerous problem instances. Of particular note is also that the memory requirement reduction of our approach makes the solution of practical problem sizes often feasible, while extant algorithms fail to compute. We also consider a heuristic to solve the problem, and, by measuring its optimality gap, we shed light on when using the exact method is worthwhile.

After introducing some implementation details next, in Section 6.1, we consider the *TBR* algorithm within the context of the sample Li & Fung supply chain discussed in the Introduction and randomly generated instances. In Section 6.2, we consider a heuristic approach and compare it with our exact method. In Section 6.3, we assess the efficiency of our class identification algorithms for various randomly generated networks.

In all our experiments, we used the following parameters. For activity a, the project manager could choose the effort levels $k_a = 0, 1, 2$, with 0 corresponding to no effort, 1 to the regular effort, and 2 to expediting. The costs for regular effort and expediting are $c_{a, k_a = 1} = \$10$ and $c_{a, k_a = 2} = \$40$, respectively. The waiting cost per unit of time, which is borne by the clients of the project, is $u = \$60$. The budget constraint on the cost $c_{\mathbf{k}}$ in each period t is $b = \$120$. Unless stated otherwise, the progress of activity a when the manager uses regular effort or expediting follows a discrete uniform distribution: $W^a_{k_a = 1} \sim \mathcal{U}\{0, 2\}$ and $W^a_{k_a = 2} \sim \mathcal{U}\{1, 3\}$.

We compare the computation times of the TBR algorithm against existing dynamic programming algorithms. The implementations of the value iteration (*VI*), policy iteration (*PI*), and linear programming (*LP*) algorithms were based on pseudo-codes presented in the appendix as Algorithms A.1–A.3, respectively. The value iteration implementation is based on an algorithm tailored to stochastic shortest-path problems on DAGs, which combines the traditional value iteration algorithm and the Dijkstra algorithm, originally developed for deterministic shortest-path problems (Bertsekas 2012, pp. 209–210).

6.1. TBR Algorithm Computational Performance

First, we consider a dynamic project expediting problem for the Li & Fung supply chain discussed in the Introduction. The AoN network representation of the problem is as described in Magretta (1998), with representative durations, and is depicted in Figure 4.

We succeeded in solving the problem using the *TBR* algorithm. The classes associated with this example are 23, the states are 122 and 431, and the average number of controls for each state is 145.57. In our tests, the existing dynamic programming algorithms failed to solve this problem because they ran out of memory when trying to generate the transition matrix.

Second, to study in more depth the performance of our algorithms we devised a series of computational experiments using randomly generated networks. In particular, we compared the computation times of *TBR* against existing dynamic programming algorithms using random AoN networks generated with the *Rangen2* software, described in Vanhoucke et al. (2008).

To generate a random network, *Rangen2* requires two factors in input: n, the number of activities in a network, and I_2, with $0 \le I_2 \le 1$, the relative length of a network. I_2 measures how "sequential" the shape of a network is (Tavares et al. 1999). $I_2 = 1$ if all activities in the network are sequential, and $I_2 = 0$ if all activities in the network are parallel. *Rangen2* also generates a random duration value for each activity of a network. *Rangen2* is primarily used to generate project networks for static scheduling problems and its duration generation is not very suitable for dynamic problems like ours, in which the durations of the activities affect the size of the state space substantially. As the software does not

Figure 4. Li & Fung Supply Chain Example
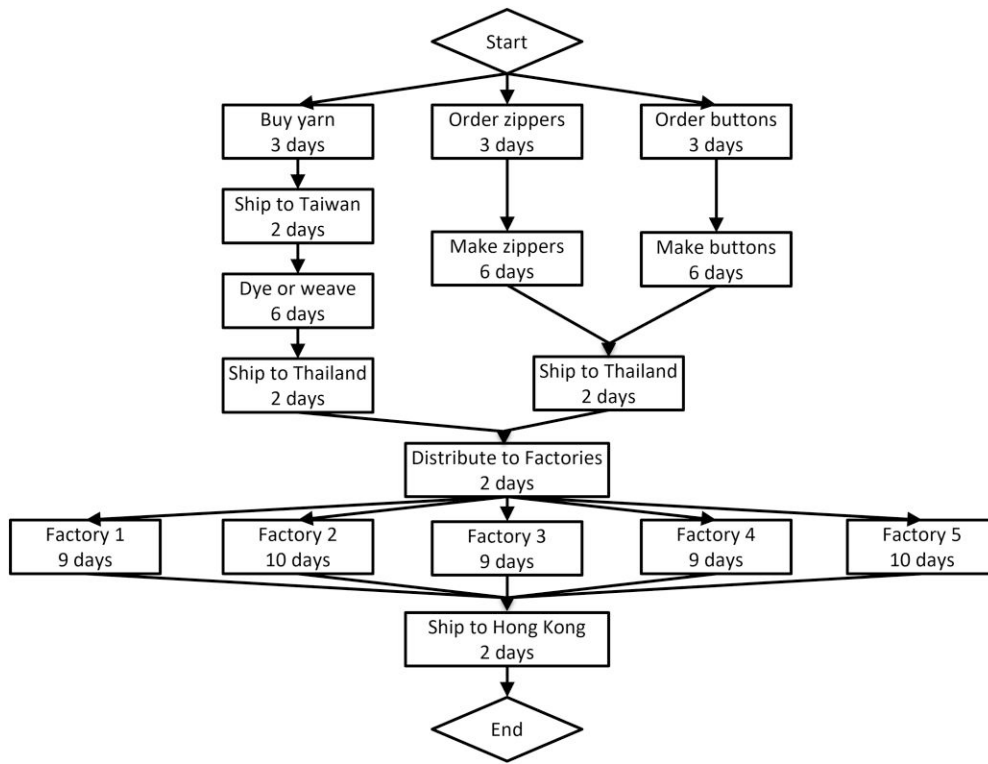
allow us to explicitly control the duration of the activities, to obtain problems of a size that is more appropriate to dynamic problems, we divided each random activity generated by three and rounded it up to obtain the length of each activity.

In our study, an experiment is defined by the combination of n and I_2. We generated 10 random networks for each experiment. In our experiments, I_2 takes values 0.9, 0.6, and 0.3 and n takes values 10, 15, 20, 25, and 30.

We implemented all the algorithms in $C++$ and ran them on an HP Z400 Workstation with an Intel(R) Xeon(R) 3.33-GHz CPU and 12 GB of RAM. We used CPLEX 12.8.0 to solve the linear programming and the mixed-integer programming models. We present the results of our computational study in Tables 4 and 5.

In Table 4, we present the percentage of instances successfully solved by the algorithms. The first three columns of the table give the coefficient I_2, the average

number of states over 50 instances, and the average number of controls per state over 50 instances. The fourth to seventh columns of Table 1 give the percentage of successfully solved instances for value iteration (*VI*), policy iteration (*PI*), linear programming (*LP*), and the *TBR* algorithms, respectively.

The *TBR* algorithm successfully identified the solution for all the instances. *VI* and *PI* did not find an optimal

Table 5. Improvement in Computational Times Compared with the Slowest Algorithm on Average (*LP*) for Instances in Which All Algorithms Solve the Problem

I_2	n	No. of states	No. of controls	Δt_{VI}	Δt_{PI}	Δt_{TBR}
0.9	10	190.9	11.6	−4.8%	+5.9%	−63.4%
0.9	15	622.0	20.8	−3.7%	+10.8%	−66.5%
0.9	20	1,239.7	22.9	−56.0%	−32.1%	−70.5%
0.9	25	1,828.9	25.9	−67.2%	−42.8%	−75.5%
0.9	30	2,633.2	27.6	−65.2%	−33.5%	−78.0%
0.6	10	291.5	26.6	−3.6%	−0.4%	−65.8%
0.6	15	1,108.7	36.6	−42.4%	−30.4%	−78.9%
0.6	20	1,946.1	71.3	−42.9%	−34.0%	−56.9%
0.6	25	2,994.2	50.0	−64.8%	−46.2%	−76.0%
0.6	30	2,135.9	37.8	−37.5%	−15.0%	−58.1%
0.3	10	1,675.2	154.6	−63.5%	−61.9%	−73.5%
0.3	15	3,003.5	111.3	−49.1%	−39.0%	−80.1%
0.3	20	3,283.2	62.3	−42.7%	−19.5%	−75.1%
0.3	25	3,541.0	68.9	−56.9%	−46.3%	−78.2%

Table 4. Percentage of Solved Instances for Each Algorithm

I_2	No. of states	No. of controls	% *VI*	% *PI*	% *LP*	% *TBR*
0.9	2,144.9	28.0	94%	94%	86%	100%
0.6	2,801.8	59.4	88%	88%	84%	100%
0.3	6,859.5	152.8	42%	42%	26%	100%

policy only when the memory of our workstation was not sufficient to store all the transition probabilities and the costs for our problem. *LP* did not find an optimal policy also for other instances.

In Table 5, we compare the computation times of the *TBR* algorithm against those of existing dynamic programming algorithms only for instances in which all algorithms solve the problem. The first four columns of the table give the coefficient I_2, the number of activities n, the average number of states over the instances solved by all algorithms, and the average number of controls per state over the instances solved by all algorithms. The fifth through seventh columns of Table 5 give the improvement in the average computation times over the instances solved by all algorithms compared with the slowest algorithm on average (*LP*), for the value iteration (*VI*), policy iteration (*PI*), and the *TBR* algorithms, respectively.

The average computation times for *TBR* were always shorter compared with the other algorithms. *TBR* reduced the computation times of *VI*, the second-best algorithm, on average by 41.9% and as much as 100.0%. Notice that it is not possible to obtain statistics when $I_2 = 0.3$ and $n = 30$ because the *TBR* algorithm is the only algorithm that can solve the instances associated with this experiment.

The computation times of all algorithms increased when the number of activities increased. Moreover, the computation times of all algorithms increased when the relative length I_2 decreased, that is, when the number of activities in parallel increased. The effect of a reduction in I_2 had a larger effect on the increment in the computation times of all algorithms than the increment in n. The computation times were strongly affected by increases in the number of states and controls, which in turn were affected by the number of activities, their lengths, and the number of parallel activities.

6.2. Heuristic Approach

In this section, we consider a heuristic approach to obtain approximate solutions to the DPE problem. By measuring the optimality gaps across different experimental setups, we shed some light on conditions under which obtaining exact solutions could yield significant benefits. In particular, we consider a heuristic approach based on popular "certainty equivalent model predictive control" methods. Across our experiments, we find optimality gaps to range between 3.93% and 35.64%. Higher optimality gap values corresponded to problem instances in which the underlying "randomness" was higher, as measured by the standard deviation of random activities' progress.

In certainty equivalent model predictive control (CEMPC) methods, at any point in time, unknown values of future parameters are replaced with point estimates over a planning horizon extending from the current time to some time in the future. For the DPE problem, at the end of time t, we consider a planning horizon $\{t+1, \ldots, T\}$, where T is selected large enough so that the project is guaranteed to have been completed by then. We also substitute the random variables measuring activities' progress with their conditional expectations based on the current state. This process gives rise to a deterministic optimal control problem, the solution of which yields a plan of action, that is, control vectors, over the remaining horizon. The CEMPC heuristic then simply implements at time $t+1$ the first control vector in the plan. After uncertainty realizes and we transition to the next state at $t+2$, this process is repeated using updated point estimates. Although CEMPC heuristics are usually suboptimal in general, they often perform remarkably well in practice and are therefore widely used to deal with stochastic control problems in several application areas, including, for example, revenue management, inventory control, scheduling, and others.

For the DPE problem, using CEMPC is appealing as the resulting deterministic optimal control problem that one obtains at each time step can be readily solved using mixed-integer optimization techniques. Although various formulations can be used to this end, we present the one that we used in our computational studies in the appendix.

We compared the average total costs obtained with the CEMPC heuristic over 50 simulation runs against the total expected optimal costs calculated using the *TBR* algorithm.

Table 6 shows the comparison when the progress of activity a follows a discrete uniform distribution: $W_{k_a=1}^a \sim \mathcal{U}\{0,2\}$ and $W_{k_a=2}^a \sim \mathcal{U}\{1,3\}$, as in our previous experiments. The average optimality gap $\Delta J^*(0)$ is 5.76%.

Table 7 shows the cost comparison between CEMPC and *TBR* in presence of different discrete distributions

Table 6. Comparison Between CEMPC and *TBR* with $W_{k_a=1}^a \sim \mathcal{U}\{0,2\}$ and $W_{k_a=2}^a \sim \mathcal{U}\{1,3\}$

I_2	n	No. of states	No. of controls	$J^*(0)$	$\Delta J^*(0)$
0.9	10	190.9	11.6	919.6	8.13%
0.9	15	622.0	20.8	1,358.0	5.65%
0.9	20	1,239.7	22.9	1,887.4	4.46%
0.9	25	2,987.6	36.7	2,381.3	5.23%
0.9	30	5,684.3	48.0	2,801.6	4.07%
0.6	10	291.5	26.6	846.2	9.28%
0.6	15	1,108.7	36.6	1,283.7	5.30%
0.6	20	2,699.2	76.8	1,692.0	5.31%
0.6	25	7,773.9	119.2	2,016.4	5.48%
0.6	30	2,135.9	37.8	2,636.8	4.57%
0.3	10	4,265.2	251.8	638.5	10.96%
0.3	15	5,300.6	135.2	1,098.3	4.15%
0.3	20	4,257.6	69.1	1,513.4	5.46%
0.3	25	9,610.9	187.9	1,832.6	4.40%
0.3	30	10,863.2	120.1	2,080.3	3.93%

Table 7. Comparison Between CEMPC and *TBR* with $P(W_{k_a=1}^a = 0) = P(W_{k_a=1}^a = 2) = 0.5$ and $P(W_{k_a=2}^a = 1) = P(W_{k_a=2}^a = 3) = 0.5$

I_2	n	No. of states	No. of controls	$J^*(0)$	$\Delta J^*(0)$
0.9	5	48.9	9.3	390.3	35.01%
0.9	10	190.9	11.6	431.8	35.64%
0.9	15	622.0	20.8	457.8	32.99%
0.9	20	1,239.7	22.9	667.0	30.03%
0.9	25	2,987.6	36.7	854.9	30.82%
0.9	30	5,684.3	48.0	918.9	32.23%
0.6	5	62.8	11.4	1,125.8	21.38%
0.6	10	291.5	26.6	1,302.1	23.31%
0.6	15	1,108.7	36.6	1,365.0	22.45%
0.6	20	2,699.2	76.8	1,562.1	18.50%
0.6	25	7,773.9	119.2	1,714.8	21.31%
0.6	30	2,135.9	37.8	1,889.6	20.59%
0.3	5	174.1	27.1	1,879.0	18.35%
0.3	10	4,265.2	251.8	2,057.3	19.89%
0.3	15	5,300.6	135.2	2,396.8	19.54%
0.3	20	4,257.6	69.1	2,161.9	17.32%
0.3	25	9,610.9	187.9	2,674.2	21.61%
0.3	30	10,863.2	120.1	2,795.7	19.24%

used to model the progress of activity *a*. These distributions have the same mean of the discrete uniform distributions used elsewhere in the paper, but a higher standard deviation: when $k = 1$, the progress is 0 with probability 0.5 and 2 with probability 0.5, and when $k = 2$ the progress is 1 with probability 0.5 and 3 with probability 0.5. The average optimality gap $\Delta J^*(0)$ is 24.37%.

In summary, the CEMPC heuristic shows limited optimality gaps when the standard deviations of the random activity progress is small. In this case, project managers could consider using the heuristic for very large projects, for which even our *TBR* algorithm cannot identify the solution. However, the optimality gaps increase when the standard deviations of the random activity progress increase. For projects with high underlying "randomness," it is crucial to use modeling frameworks that explicitly consider the variability of activities.

6.3. Class Identification Algorithms

We compared the computational performance of *CId* against our implementation of the fast class identification

Table 8. Comparison Between Creemers et al. (2010) and *CId* for $I_2 = 0.9$

I_2	n	No. of classes	$t_{Creemers}$ (s)	t_{CId} (s)
0.9	30	165.7	<0.01	<0.01
0.9	60	453.1	0.02	0.04
0.9	90	577.7	0.05	0.03
0.9	120	824.3	0.05	0.05
0.9	180	1,269.9	0.36	0.14
0.9	240	1,863.6	0.40	0.31
0.9	300	3,044.1	11.42	0.97

Table 9. Comparison Between Creemers et al. (2010) and *CId* for $I_2 = 0.6$

I_2	n	No. of classes	$t_{Creemers}$ (s)	t_{CId} (s)
0.6	30	55.9	<0.01	<0.01
0.6	60	133.4	0.01	0.01
0.6	90	232.9	0.03	0.02
0.6	120	211.1	0.25	0.03
0.6	180	347.6	0.11	0.08
0.6	240	697.9	1.19	0.23
0.6	300	787.2	Out of Memory	0.39

algorithm proposed by Creemers et al. (2010). As before, we compared the computation times of the two algorithms using networks randomly generated by the *Rangen2* software.

In our experiments, the number of activities *n* takes values ranging from 30 to 300, and I_2, which measures how sequential a network is, takes values 0.9, 0.6, and 0.3. For each experiment, we calculate the average computation times by averaging the times required to identify the classes of 10 random networks.

The third through fifth columns of Tables 8–10 give the average number of classes for each experiment, and the average computation times over 10 instances (in seconds) for the *CId* algorithm and for our implementation of the algorithm developed by Creemers et al. (2010) (Creemers).

The average computation times of *CId* are significantly smaller than those of our implementation of the algorithm developed by Creemers et al. (2010). The average percent reduction of *CId* with respect to Creemers' algorithm was 43% and the maximum percent reduction was 99%.

The *CId* algorithm has a clear performance advantage over Creemers' algorithm for networks with a large number of activities. The computation times of the two algorithms increased when the number of activities increased.

For $I_2 = 0.9$, the *CId* has a performance advantage against the Creemers' algorithm when the number of activities is very large.

For $I_2 = 0.6$ and $I_2 = 0.3$ and a large number of activities, Creemers' algorithm runs out of memory in some instances. This is because the algorithm needs to store

Table 10. Comparison Between Creemers et al. (2010) and *CId* for $I_2 = 0.3$

I_2	n	No. of classes	$t_{Creemers}$ (s)	t_{CId} (s)
0.3	30	66.7	<0.01	0.01
0.3	60	101.5	0.03	0.02
0.3	90	207.5	0.22	0.04
0.3	120	169.1	3.80	0.04
0.3	180	963.7	2.05	0.88
0.3	240	210.8	NA	0.17
0.3	300	853.8	NA	0.66

upfront all the data that it needs to identify the classes of a network. Instead, the use of recursion in the *CId* algorithm limits its data storage requirements.

7. Conclusions

In this paper, we formulated and solved a dynamic model to identify optimal expediting policies for a project with random progress. Our results contribute to the effectiveness of monitoring and control, which is critical to project success (Pinto and Mantel 1990). It provides practical control policies that build on recent technological advances, which have enabled effective monitoring of complex operational processes.

In particular, we dealt with an MDP formulation of the dynamic project expediting problem and derived useful structural properties. In turn, these enabled us to devise *TBR*, an exact solution method that is significantly less computationally burdensome compared with extant available solution methods. We showed how our method can be used to attack a larger class of so-called stochastic shortest-path problems. The latter, termed forward-only stochastic shortest-path problems, are characterized by intuitive ordering properties and could capture other important applications, including medical decision-making and disease modeling. We also dealt with the state identification problem and devised the *SEn* and *CId* algorithms. Numerical experiments demonstrated that both our solution and state identification methods significantly outperform extant methods for a supply chain example and for various randomly generated instances.

We briefly discuss some possibilities for future research. The formulation of the problem and its state-space identification are quite general and could be used in future contributions on dynamic project management under uncertainty, such as those that study dynamic stochastic models for leveling and resource-constrained scheduling. The scope of application of the *TBR* algorithm is also quite general. We showed that the algorithm finds the optimal solution to forward-only stochastic shortest-path problems, a broad class of MDP problems we characterized. Future studies could apply the *TBR* algorithm to solve other problems belonging to this class.

Appendix
How to calculate $P^a_{x_a,y_a}(k_a)$ in all possible cases.
1. If a is in progress in $\mathbf{x}$:
   If $y_a \geq x_a$:
   If a is in progress in $\mathbf{y}$, $P^a_{x_a,y_a}(k_a) = P(W_{a,k_a} = y_a - x_a)$.

If a is completed in $\mathbf{y}$, $P^a_{x_a,y_a}(k_a) = 1 - \sum_{i=0}^{y_a-x_a-1} P(W_{a,k_a} = i)$.
   If $y_a < x_a$, $P^a_{x_a,y_a}(k_a) = 0$.
2. If a is un-engageable in $\mathbf{x}$:
   If a is un-engageable in $\mathbf{y}$, $P^a_{x_a,y_a}(k_a) = 1$.
   If $y_a = 0$:
   If all $a' \in \Pi_a$ are completed in $\mathbf{y}$, $P^a_{x_a,y_a}(k_a) = 1$; otherwise, $P^a_{x_a,y_a}(k_a) = 0$.
   If $y_a > 0$, $P^a_{x_a,y_a}(k_a) = 0$.
3. If a is completed in $\mathbf{x}$:
   If $y_a < l_a$, $P^a_{x_a,y_a}(k_a) = 0$.
   If a is completed in $\mathbf{y}$ ($y_a = l_a$), $P^a_{x_a,y_a}(k_a) = 1$.

Proof of Lemma 1. Bertsekas (2017, pp. 237–240) shows that for a stochastic shortest-path problem, there exists an optimal stationary policy μ if there exists an integer M such that, regardless of the policy used and the initial state (which in our case is always equal to $\mathbf{i}$), there is a positive probability that the terminal state will be reached after no more than M periods. Equivalently, there exists an optimal stationary policy for a stochastic shortest-path problem if, for each control, every state is connected to the terminal state by a path of positive-probability transitions. To prove this for our problem, it is sufficient to show that for each state $\mathbf{x} \in \mathcal{S}$ and each control $\mathbf{k} \in \mathcal{K}(\mathbf{x})$ there is a positive probability to move to at least one state $\mathbf{y}$ that is "closer" than $\mathbf{x}$ to the terminal state $\mathbf{f}$ in a sense that will now be made clear.

If $\mathbf{x} = \mathbf{f}$, there is nothing to prove. Therefore, let $\mathbf{x} \neq \mathbf{f}$. This implies that $\mathcal{P}$, the set of all activities in progress in $\mathbf{x}$, is nonempty. The assumption $\mathbf{0} \notin \mathcal{K}(x)$ implies that there exists at least one $a' \in \mathcal{P}$ such that $k_{a'} \neq 0$. Take a state $\mathbf{y} \in \mathcal{X}$ such that the following hold:

1. $y_{a'} > x_{a'}$, and $y_a \geq x_a$ for all other activities a that are in progress, where $y_a > x_a$ holds only if $k_a \neq 0$.

2. $y_a = x_a$ for all completed activities a.

3. $y_a \geq x_a$ for all un-engageable activities a, where $y_a > x_a$ holds only if all predecessors of a are completed in a possible transition from $\mathbf{x}$ to $\mathbf{y}$.

The probability of moving from $\mathbf{x}$ to $\mathbf{y}$ is $P_{\mathbf{x},\mathbf{y}}(\mathbf{k}) = \prod_{a \in \mathcal{A}} P^a_{x_a,y_a}(k_a)$. For $a = a'$, and for all other activities a in progress such that $y_a > x_a$, $0 < P^a_{x_a,y_a}(k_a) < 1$. For all other activities a, $P^a_{x_a,y_a}(k_a) = 1$. Therefore, $P_{\mathbf{x},\mathbf{y}}(\mathbf{k}) > 0$. $\square$

Proof of Lemma 2. To prove this lemma, we have to show that the relation $\prec$ is irreflexive and transitive. It is irreflexive, that is, $\neg(\mathbf{x} < \mathbf{x})$, because $x_a = x_a$ for all activities a. It is obviously transitive, that is, $\mathbf{x} < \mathbf{y}$ and $\mathbf{y} < \mathbf{z}$ imply that $\mathbf{x} < \mathbf{z}$. Therefore, it is a strict partial order on the nonempty state space $\mathcal{X}$. $\square$

Proof of Lemma 3. Theorem 10.6.8 in Lehman et al. (2017, p. 400) shows that a relation is a strict partial order if and only if it is the positive-walk relation on a directed acyclic graph. The theorem establishes the equivalence between $\prec$ and the positive-walk relation on $\mathcal{G}^\prec(\mathcal{V}^\prec, \mathcal{A}^\prec)$. A transitive reduction of a directed graph $\mathcal{G}$ is a directed graph $\mathcal{G}'$ with the same set of nodes and as few arcs as possible, such that if there is a path from node v_1 to node v_2 in $\mathcal{G}$, there exists such a path in $\mathcal{G}'$ as well. $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ is the transitive reduction of $\mathcal{G}^\prec(\mathcal{V}^\prec, \mathcal{E}^\prec)$. This is because every arc e in $\mathcal{E}'$ corresponds to just one unit of progress, which is the smallest progress possible. It is obvious from the definition of transitive reduction that the positive-walk relation on a graph is equivalent to the positive-walk

relation on its transitive reduction. This applies to $\mathcal{G}^{\prec}(\mathcal{V}^{\prec}, \mathcal{E}^{\prec})$ and its transitive reduction $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$. □

Proof of Lemma 4. Substituting $\hat{y}$ for y in Equation (4), we obtain

$$\frac{a_{k^*}}{1 - \delta_{k^*}} = \min_k \left[a_k + \delta_k \frac{a_{k^*}}{1 - \delta_{k^*}} \right];$$

$$\frac{a_{k^*}}{1 - \delta_{k^*}} = \min \left[a_{k^*} + \delta_{k^*} \frac{a_{k^*}}{1 - \delta_{k^*}}, \min_{k \neq k^*} \left[a_k + \delta_k \frac{a_{k^*}}{1 - \delta_{k^*}} \right] \right];$$

$$\frac{a_{k^*}}{1 - \delta_{k^*}} = \min \left[\frac{a_{k^*}}{1 - \delta_{k^*}}, \min_{k \neq k^*} \frac{(1 - \delta_{k^*})a_k + \delta_k a_{k^*}}{1 - \delta_{k^*}} \right].$$

From $\frac{a_{k^*}}{1 - \delta_{k^*}} = \min_k \frac{a_k}{1 - \delta_k}$, it follows that

$$a_k \geq \frac{a_{k^*}}{1 - \delta_{k^*}} (1 - \delta_k), \quad k \neq k^*;$$

$$(1 - \delta_k)a_k + \delta_k a_{k^*} \geq (1 - \delta_k) \frac{a_{k^*}}{1 - \delta_{k^*}} (1 - \delta_k) + \delta_k a_{k^*} = a_{k^*}, \quad k \neq k^*;$$

$$\min \left[\frac{a_{k^*}}{1 - \delta_{k^*}}, \min_{k \neq k^*} \frac{(1 - \delta_{k^*})a_k + \delta_k a_{k^*}}{1 - \delta_{k^*}} \right] = \frac{a_{k^*}}{1 - \delta_{k^*}}.$$

Therefore, $\hat{y} = \frac{a_{k^*}}{1 - \delta_{k^*}}$ solves Equation (4). □

Proof of Theorem 1. The linearly ordered state space $\widetilde{\mathcal{X}} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_s\}$ is a list of states. Because we obtained $\widetilde{\mathcal{X}}$ through topological sort (Definition 9), we have that for any state $\mathbf{x}_i \in \widetilde{\mathcal{X}}$, all states $\mathbf{x}_j$ with $\mathbf{x}_i \prec \mathbf{x}_j$ will appear later in $\widetilde{\mathcal{X}}$. Thus if $\mathbf{x}_l$ precedes $\mathbf{x}_i$ in the list $\widetilde{\mathcal{X}}$, there are two possibilities:

1. Case 1 $\mathbf{x}_l$ and $\mathbf{x}_i$ are incomparable. Then $P_{\mathbf{x}_i, \mathbf{x}_l}(\mathbf{k}) = 0$ and $P_{\mathbf{x}_l, \mathbf{x}_i}(\mathbf{k}) = 0$ for every control $\mathbf{k}$.

2. Case 2 $\mathbf{x}_l \prec \mathbf{x}_i$. Then Remark 1 implies that $P_{\mathbf{x}_i, \mathbf{x}_l}(\mathbf{k}) = 0$ for every $\mathbf{k} \in \mathcal{K}(\mathbf{x}_i)$.

Thus for any state $\mathbf{x}_i$, it suffices to consider in its Bellman's equation only $\mathbf{x}_i$ and those states that appear later than $\mathbf{x}_i$ in $\widetilde{\mathcal{X}}$, that is, $\mathbf{x}_j$ such that $j \geq i$. Therefore, we can write Bellman's equation for state $\mathbf{x}_i$ as follows:

$$J^*(\mathbf{x}_i) = \min_{\mathbf{k} \in \mathcal{K}(\mathbf{x}_i)} \left[\overline{g}(\mathbf{x}_i, \mathbf{k}) + \sum_{j=i}^{s-1} P_{\mathbf{x}_i, \mathbf{x}_j}(\mathbf{k}) J^*(\mathbf{x}_j) \right] \quad i = 1, 2, \ldots, s - 1.$$

For a generic state $\mathbf{x}_i$, we already obtained the costs-to-go $J^*(\mathbf{x}_{i+1}), \ldots, J^*(\mathbf{x}_{s-1})$ by applying the *TBR* algorithm. Next, we apply Lemma 4 with the following:

$y = J^*(\mathbf{x}_i)$;
$a_k = \overline{g}(\mathbf{x}_i, \mathbf{k}) + \sum_{j=i+1}^{s-1} P_{\mathbf{x}_i, \mathbf{x}_j}(\mathbf{k}) J^*(\mathbf{x}_j)$;
$\delta_k = P_{\mathbf{x}_i, \mathbf{x}_i}(\mathbf{k})$.
This is possible because
$\overline{g}(\mathbf{x}_i, \mathbf{k}) + \sum_{j=i}^{s-1} P_{\mathbf{x}_i, \mathbf{x}_j}(\mathbf{k}) J^*(\mathbf{x}_j) > 0$; and
$0 < P_{\mathbf{x}_i, \mathbf{x}_i}(\mathbf{k}) < 1$.
Then the following cost satisfies Bellman's equation:

$$J^*(\mathbf{x}_i) = \frac{\overline{g}(\mathbf{x}_i, \mathbf{k}^*) + \sum_{j=i+1}^{s-1} P_{\mathbf{x}_i, \mathbf{x}_j}(\mathbf{k}^*) J^*(\mathbf{x}_j)}{1 - P_{\mathbf{x}_i, \mathbf{x}_i}(\mathbf{k}^*)}$$

$$= \min_{\mathbf{k} \in \mathcal{K}(\mathbf{x}_i)} \frac{\overline{g}(\mathbf{x}_i, \mathbf{k}) + \sum_{j=i+1}^{s-1} P_{\mathbf{x}_i, \mathbf{x}_j}(\mathbf{k}) J^*(\mathbf{x}_j)}{1 - P_{\mathbf{x}_i, \mathbf{x}_i}(\mathbf{k})}.$$

Therefore, it is optimal. □

Proof of Lemma 5. The relation in Definition 11 is reflexive, symmetric, and transitive because each component of a state is either engaged or unengageable. Therefore, it is an equivalence relation on the state space $\mathcal{X}$. As $\mathcal{X}$ is nonempty, the equivalence classes $\hat{S}_1, \hat{S}_2, \ldots$ partition the state space $\mathcal{X}$. □

Proof of Theorem 2. First, we prove that the *CId* algorithm enumerates all classes of feasible states. Consider any subgraph $\Gamma(\hat{\mathcal{A}}, \hat{\mathcal{E}}) \in H$.

If the set of predecessors Π_a is empty for all activities $a \in \hat{\mathcal{A}}$, then all $a \in \hat{\mathcal{A}}$ are parallel, because, for each a, their predecessors are completed. Therefore, $\Gamma(\hat{\mathcal{A}}, \hat{\mathcal{E}})$ provides a class of states.

If the set of predecessors Π_a is nonempty for at least one $a \in \hat{\mathcal{A}}$, then some activities are sequential in $\Gamma(\hat{\mathcal{A}}, \hat{\mathcal{E}})$. Consider one such a. For this activity, build the subgraph $\Gamma(\bar{\mathcal{A}}_a, \bar{\mathcal{E}}_a)$, where $\bar{\mathcal{A}}_a = \hat{\mathcal{A}} \setminus \Pi_a \cup S_a$. Notice that $a \in \bar{\mathcal{A}}_a$ because it cannot be a predecessor or a successor of itself. Then, in $\Gamma(\bar{\mathcal{A}}_a, \bar{\mathcal{E}}_a)$, a is no longer in a sequence because its predecessors and successors are not in $\bar{\mathcal{A}}_a$ by definition. Consider the activities $a' \in \bar{\mathcal{A}}_a$. If $\Pi_{a'}$ is empty for all activities $a' \in \bar{\mathcal{A}}_a$, then $\Gamma(\bar{\mathcal{A}}_a, \hat{\mathcal{E}}_a)$ provides a class of states. Otherwise, there exists an activity $a' \neq a$ that is in a sequence in $\bar{\mathcal{A}}_a$. If that is the case, build a subgraph $\Gamma(\bar{\mathcal{A}}_{a'}, \bar{\mathcal{E}}_{a'})$ where a' is no longer in a sequence. In this way, it is possible to recursively build the subgraphs until we obtain a collection of sub-graphs in which all their activities are in parallel. These are the classes of states.

The *CId* algorithm starts with $\Gamma(\hat{\mathcal{A}}, \hat{\mathcal{E}}) = G(\mathcal{A}, \mathcal{E})$ and terminates only when $H = \emptyset$, that is, when all the subgraphs have been visited. Therefore, it generates all the classes of states.

Second, we prove that the *SEn* algorithm enumerates all feasible states.

In a project without precedence constraints, for each class, the recursive procedure *SEnvisit* generates a complete tree in which each level corresponds to an activity a and each leaf corresponds to a state. In a project with precedence constraints, for each class, the only leaves that are not generated by the procedure are those that correspond to infeasible states because of precedence constraints. Therefore, the *SEn* algorithm enumerates all the feasible states for all classes. □

Algorithm A.1 (Dijkstra-Based Value Iteration Algorithm)
begin
  $e = 0$;
  $\widetilde{g}(\mathbf{x}, \mathbf{k}) = \frac{\overline{g}(\mathbf{x}, \mathbf{k})}{1 - P_{\mathbf{x}, \mathbf{x}}(\mathbf{k})} \quad \forall \mathbf{x} \in \mathcal{X}$;
  $\widetilde{P}_{\mathbf{x}, \mathbf{x}}(\mathbf{k}) = 0$;
  $\widetilde{P}_{\mathbf{x}, \mathbf{y}}(\mathbf{k}) = \frac{P_{\mathbf{x}, \mathbf{y}}(\mathbf{k})}{1 - P_{\mathbf{x}, \mathbf{x}}(\mathbf{k})} \quad \forall \mathbf{x}, \mathbf{y} \in \mathcal{X} | \mathbf{x} \neq \mathbf{y}$;
  $L \leftarrow \mathbf{f}$;
  $B \leftarrow \emptyset$;
  Set $J(\mathbf{x})$ to $+\infty \quad \forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$ and $J(\mathbf{f}) = 0$;
end
repeat
  $\mathbf{y}^* = \arg \min_{\mathbf{y} \in L} J(\mathbf{y})$;
  $L = L \setminus \{\mathbf{y}^*\}$;
  $B = B \cup \{\mathbf{y}^*\}$;
  for *each* $\mathbf{x} \notin B$ **do**
    $\hat{\mathcal{K}}(\mathbf{y}) = \{\mathbf{k} \in \mathcal{K}(\mathbf{x}) | \widetilde{P}_{\mathbf{x}, \mathbf{y}^*}(\mathbf{k}) > 0 \text{ and } \widetilde{P}_{\mathbf{x}, \mathbf{y}}(\mathbf{k}) = 0 \ \forall \mathbf{y} \notin B\}$;
    $J(\mathbf{x}) = \min \left[J(\mathbf{x}), \min_{\mathbf{k} \in \hat{\mathcal{K}}(\mathbf{x})} [\widetilde{g}(\mathbf{x}, \mathbf{k}) + \sum_{\mathbf{y} \in B} \widetilde{P}_{\mathbf{x}, \mathbf{y}}(\mathbf{k}) J(\mathbf{y})] \right]$;
    $L = L \cup \{\mathbf{x}\}$;
  end
  $e = e + 1$;
until
$L = \mathcal{X}$;
return $\mathbf{k}^*(\mathbf{x})$, *the control corresponding to* $J(\mathbf{x})$, $\forall \mathbf{x} \in \mathcal{X}$.

Algorithm A.2 (Policy Iteration Algorithm)
begin
  $e = 0$;
  Set $\mathbf{k}_e(\mathbf{x})$ to a random value $\forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$ and $\mathbf{k}_e(\mathbf{f}) = \mathbf{0}$;
end
repeat
  Solve the linear system of equations:
  $J_{\mathbf{k}_e}(\mathbf{x}) = \overline{g}(\mathbf{x}, \mathbf{k}_e(\mathbf{x})) + \sum_{\mathbf{y} \in \mathcal{X}} P_{\mathbf{x},\mathbf{y}}(\mathbf{k}_e(\mathbf{x})) J_{\mathbf{k}_e}(\mathbf{y}) \quad \forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$;
  Compute:
  $\mathbf{k}_{e+1}(\mathbf{x}) = \arg\min_{\mathbf{k} \in \mathcal{K}(\mathbf{x})} [\overline{g}(\mathbf{x}, \mathbf{k}) + \sum_{\mathbf{y} \in \mathcal{X}} P_{\mathbf{x},\mathbf{y}}(\mathbf{k}) J_{\mathbf{k}_e}(\mathbf{y})] \quad \forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$;
  $e = e+1$;
until $J_{\mathbf{k}_{e+1}}(\mathbf{x}) = J_{\mathbf{k}_e}(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$;
return $\mathbf{k}^*(\mathbf{x}) = \mathbf{k}_{e+1}(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{X}$.

Algorithm A.3 (Linear Programming Algorithm)
begin
  Let $\omega(\mathbf{x})$ be non–negative continuous variables $\forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$;
end
Obtain optimal $\omega^*(\mathbf{x})$ by solving the linear programming model:
$\max \sum_{\mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}} \omega(\mathbf{x})$
$\omega(\mathbf{x}) \leq \overline{g}(\mathbf{x}, \mathbf{k}) + \sum_{\mathcal{X}} P_{\mathbf{x},\mathbf{y}}(\mathbf{k}) \omega(\mathbf{y}) \; \forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$ and $\forall \mathbf{k} \in \mathcal{K}(\mathbf{x})$
$\omega(\mathbf{x}) \geq 0 \quad \forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$;
Compute:
$\mathbf{k}^*(\mathbf{x}) = \arg\min_{\mathbf{k} \in \mathcal{K}(\mathbf{x})} \left[\overline{g}(\mathbf{x}, \mathbf{k}) + \sum_{\mathbf{y} \in \mathcal{X}} P_{\mathbf{x},\mathbf{y}}(\mathbf{k}) \omega^*(\mathbf{y}) \right] \quad \forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$;
return $\mathbf{k}^*(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{X} \setminus \{\mathbf{f}\}$

Algorithm A.4 (Tree Constructor (Based on Creemers et al. 2010))
Algorithm TreeConstructor()
  build empty tree $T(\mathcal{V}, \mathcal{E})$;
  build root node r;
  associate to r fictitious activity $a = -1$;
  call AddNodes($T(\mathcal{V}, \mathcal{E}), r$);
  return $T(\mathcal{V}, \mathcal{E})$;
Procedure AddNodes($T(\mathcal{V}, \mathcal{E}), v$)
  if $a \neq -1$ **then**
    add $\mathcal{S}_a$ to forbidden branches $T.frb[v]$;
  end
  for $a' = a + 1, a + 2, \ldots, n$ **do**
    if $a = -1$ **then**
      add u (node associated to a') to $\mathcal{V}$;
      add u to adjacency list $T.adj[v]$;
    else
      if $a' \notin T.frb[v]$ **then**
        add u (node associated to a') to $\mathcal{V}$;
        add u to $T.adj[v]$;
        $T.frb[u] = T.frb[u] \cup T.frb[v]$;
      end
    end
    if $a = n$ **then**
      clear $T.frb[v]$;
    end
  end
  for each $u \in T.Adj[v]$ **do**
    call AddNodes($T(\mathcal{V}, \mathcal{E}), u$);
  end

Algorithm A.5 (Tree Exploration (Based on Depth First Search and Creemers et al. 2010))
Algorithm TreeExplorationDFS($T(\mathcal{V}, \mathcal{E})$)
  for *each node* $v \in \mathcal{V}$ **do**
    $v.visited = FALSE$;
  end
  $L \leftarrow \emptyset$;
  call DFSVisit($root\ r, T(\mathcal{V}, \mathcal{E}), L$);
  return L;
Procedure DFSVisit($v, T(\mathcal{V}, \mathcal{E}), L$)
  $v.visited = TRUE$;
  if $a(v) \neq -1$ **then**
    add $a(v)$ to $T.Sbs[v]$ (subset of activities associated to v);
  end
  if v *is a leaf* **then**
    if $T.Sbs[v]$ *is not included in any of the elements of* L **then**
      $L = L \cup T.Sbs[v]$;
    end
  else
    for *each* $u \in T.Adj[v]$ **do**
      if $u.visited = FALSE$ **then**
        $T.Sbs[u] = T.Sbs[v]$;
        if $a(v) \neq -1$ **then**
          $T.Sbs[u] = T.Sbs[u] \cup \{a(v)\}$;
        end
        call DFSVisit($u, T(\mathcal{V}, \mathcal{E}), L$);
      end
    end
  end
end

A.1. CEMPC Heuristic

We present the optimization problem formulation that we use to solve the certainty equivalent deterministic optimal control problem at the end of each time t, as discussed in Section 6.2.

For each activity a, state x_a, and control k_a, we consider the expected progress to be made and denote it by $w^a_{x_a, k_a} = \lfloor \mathbb{E}[W^a_{x_a, k_a}] \rfloor$. The main decision variables are then the effort levels to be exerted in the planning horizon. Let $y_{a, \tau, \kappa}$ indicate whether the planned effort level for activity a at some time $\tau \in \{t + 1, \ldots, T\}$ is at least $\kappa \in \{1, \ldots, K\}$. A formulation for the deterministic optimal control problem at time t that yields an optimal solution for the main decision variables is as follows:

$$\text{minimize} \quad \sum_{\tau=t+1}^{T} u \cdot z_\tau + \sum_{\tau=t+1}^{T} \sum_a \sum_{\kappa=1}^{K} (c_{a, \kappa} - c_{a, \kappa-1}) y_{a, \tau, \kappa} \quad \text{(A.1)}$$

$$\text{subject to} \quad x_{a, \tau} = x_{a, \tau-1} + \sum_{\kappa=1}^{K} (w^a_{x_{a, \tau-1}, \kappa} - w^a_{x_{a, \tau-1}, \kappa-1}) y_{a, \tau, \kappa},$$
$$\tau = t+1, \ldots, T, \quad \forall a, \quad \text{(A.2)}$$

$$y_{a, \tau, \kappa+1} \leq y_{a, \tau, \kappa}, \quad \tau = t+1, \ldots, T, \; \kappa = 1, \ldots, K-1,$$
$$\forall a, \quad \text{(A.3)}$$

$$y_{a, \tau, 1} \leq \frac{x_{a', \tau-1}}{l_{a'}}, \quad \tau = t+1, \ldots, T,$$
$$\forall a, \; \forall a' \text{ immediate predecessor to } a, \quad \text{(A.4)}$$

$$z_\tau \geq 1 - \frac{x_{a,\tau-1}}{l_a}, \quad \tau = t+1, \ldots, T, \quad \forall a, \quad \text{(A.5)}$$

$$\sum_a \sum_{\kappa=1}^K (c_{a,\kappa} - c_{a,\kappa-1})y_{a,\tau,\kappa} \leq b, \quad \tau = t+1, \ldots, T, \quad \text{(A.6)}$$

$$x_{a,\tau} \in \{0, 1, \ldots, l_a\}, \ \tau = t+1, \ldots, T, \ \forall a, \quad \text{(A.7)}$$

$$y_{a,\tau,\kappa} \in \{0,1\}, \quad \tau = t+1, \ldots, T, \quad \kappa = 1, \ldots, K, \ \forall a, \quad \text{(A.8)}$$

$$z_\tau \in \{0,1\}, \quad \tau = t+1, \ldots, T, \quad \text{(A.9)}$$

with variables $x_{a,\tau}$, $y_{a,\tau,\kappa}$, and z_τ, for $\tau = t+1, \ldots, T$, $\kappa = 1, \ldots, K$, and $\forall a$. The variables $x_{a,\tau}$ correspond to the state that activity a is planned to be at time τ—$x_{a,t}$ is not a variable, but rather input data, as the state at time t is known. The variables z_τ indicate whether the project is still in progress at time τ.

The first constraint reflects state transitioning. The second constraint enforces that if the effort level is at least $\kappa + 1$, it is also at least κ and that if it is not at least κ, then it is not at least $\kappa + 1$ either. The third constraint ensures that effort can be exerted for some activity at some time only if all immediate predecessor activities are complete. The fourth constraint ensures that the project remains in progress if any of the activities is still in progress. The fifth constraint reflects the budget constraint.

References

Azaron A, Katagiri H, Sakawa M (2007) Time-cost trade-off via optimal control theory in Markov PERT networks. *Ann. Oper. Res.* 150(1):47–64.

Bertsekas D (2012) *Dynamic Programming and Optimal Control*, vol. 2, 4th ed. (Athena Scientific, Nashua, NH).

Bertsekas D (2017) *Dynamic Programming and Optimal Control*, vol. 1, 4th ed. (Athena Scientific, Nashua, NH).

Bertsekas D, Tsitsiklis J (1991) An analysis of stochastic shortest path problems. *Math. Oper. Res.* 16(3):580–595.

BioNTech (2021) Update on vaccine production at biontech's manufacturing site in marburg. Retrieved March 23, 2021, https://investors.biontech.de/node/9301/pdf.

Bregman R (2009) A heuristic procedure for solving the dynamic probabilistic project expediting problem. *Eur. J. Oper. Res.* 192(1):125–137.

Cormen T, Leiserson C, Rivest R, Stein C (2009) *Introduction to Algorithms*, 3rd ed. (MIT Press, Cambridge, MA).

Creemers S, Leus R, Lambrecht M (2010) Scheduling Markovian PERT networks to maximize the net present value. *Oper. Res. Lett.* 38(1):51–56.

De Meyer A, Loch C, Pich M (2006) Management of novel projects under conditions of high uncertainty. Working paper, Cambridge Judge Business School, Cambridge, UK.

Fulkerson D (1961) A network flow computation for project cost curves. *Management Sci.* 7(2):167–178.

Godinho P, Branco F (2012) Adaptive policies for multi-mode project scheduling under uncertainty. *Eur. J. Oper. Res.* 216(3):553–562.

Goh J, Hall N (2013) Total cost control in project management via satisficing. *Management Sci.* 59(6):1354–1372.

Grushka-Cockayne Y (2020) Use data to revolutionize project planning. *Harvard Bus. Rev.* https://hbr.org/2020/02/use-data-to-revolutionize-project-planning.

Gutjahr W, Strauss C, Wagner E (2000) A stochastic branch-and-bound approach to activity crashing in project management. *INFORMS J. Comput.* 12(2):125–135.

Jensen J, Gutin G (2007) *Digraphs: Theory, Algorithms and Applications* (Springer-Verlag, Berlin).

Jørgensen T, Wallace S (2000) Improving project cost estimation by taking into account managerial flexibility. *Eur. J. Oper. Res.* 127(2):239–251.

Kelley J Jr (1961) Critical path planning and scheduling: Mathematical basis. *Oper. Res.* 9(3):296–320.

Klastorin T, Mitchell G (2013) Optimal project planning under the threat of a disruptive event. *IIE Trans.* 45(1):68–80.

Laslo Z (2003) Activity time-cost tradeoffs under time and cost chance constraints. *Comput. Industry Engrg.* 44(3):365–384.

Lehman E, Leighton F, Meyer A (2017) *Mathematics for Computer Science*, 3rd ed. https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_and_Computation_Fundamentals/Mathematics_for_Computer_Science_(Lehman_Leighton_and_Meyer) .

Li H, Womer N (2015) Solving stochastic resource-constrained project scheduling problems by closed-loop approximate dynamic programming. *Eur. J. Oper. Res.* 246(1):20–33.

Magretta J (1998) Fast, global, and entrepreneurial: Supply chain management, Hong Kong style. *Harvard Bus. Rev.* https://hbr.org/1998/09/fast-global-and-entrepreneurial-supply-chain-management-hong-kong-style.

Mitchell G, Klastorin T (2007) An effective methodology for the stochastic project compression problem. *IIE Trans.* 39(10):957–969.

Pinto J, Mantel S (1990) The causes of project failure. *IEEE Trans. Engrg. Management* 37(4):269–276.

PMI (2018) Project management institute: Pulse of the profession. Retrieved March 23, 2021, https://www.pmi.org/-/media/pmi/documents/public/pdf/learning/thought-leadership/pulse/pulse-of-the-profession-2018.pdf.

PMI (2020) Project management institute: Pulse of the profession. Retrieved March 23, 2021, https://www.pmi.org/learning/library/forging-future-focused-culture-11908.

ProjectManagementdotcom (2017) Discussion board: When should you update baselines and when should you not? Retrieved October 22, 2018, https://www.projectmanagement.com/discussion-topic/65058/When-should-you-update-baselines-and-when-should-you-not-.

Shen S, Smith J, Ahmed S (2010) Expectation and chance-constrained models and algorithms for insuring critical paths. *Management Sci.* 56(10):1794–1814.

Sigal C, Pritsker A, Solberg J (1980) The stochastic shortest route problem. *Oper. Res.* 28(5):1122–1129.

Sobel MJ, Szmerekovsky JG, Tilson V (2009) Scheduling projects with stochastic activity duration to maximize expected net present value. *Eur. J. Oper. Res.* 198(3):697–705.

Sterman J (1992) System dynamics modeling for project management. Working paper, Massachusetts Institute of Technology, Cambridge, MA.

Tavares L, Ferreira J, Coelho J (1999) The risk of delay of a project in terms of the morphology of its network. *Eur. J. Oper. Res.* 119(2):510–537.

Vanhoucke M, Coelho J, Debels D, Maenhout B, Tavares L (2008) An evaluation of the adequacy of project network generators with systematically sampled networks. *Eur. J. Oper. Res.* 187(2):511–524.