POLITECNICO DI TORINO

Master's Degree
in Mathematical Engineering

Master's Thesis

Dynamic Optimization of Intralogistic Robot Scheduling

Supervisors
Prof. Paolo Brandimarte
Dr. Nicolò Mazzi - *Spindox, αhead*

Candidate
Margherita Battistotti

Academic Year 2023-2024

To my future.

Che io sia accompagnata dalle stesse persone che mi hanno supportata nei miei studi, nelle mie passioni, nella mia vita, fino a questo traguardo.

To whom believes in me,
more than I do.

Abstract

The employment of intralogistic robots in warehouses has become increasingly prevalent due to technological advancements; in fact, they enable more efficient logistic operations, increasing productivity and cost-effectiveness.
The research community is showing a keen interest in the subject and directing its attention towards the implementation of risk-aware robots with planning capabilities. Nevertheless, to our knowledge, no one has yet investigated the application of the Dynamic Programming paradigm for the scheduling of tasks for intralogistic robots. Therefore, this dissertation focuses on addressing this topic, proposing various resolution methods for a stochastic dynamic scheduling problem, based on Dynamic Programming and its approximated versions. We find the paradigm particularly intriguing and deserving of further exploration because it resides between a static approach and a purely dynamic one: it promises a greater capacity to account for stochasticity than the former and surely appears less myopic than the latter.
Although it is well-known that Dynamic Programming is enormously susceptible to the dimensions of the problem to which it is applied, we are convinced of its efficacy on modest-scale instances. Therefore, we consider its results on small scale problems as benchmarks to be compared with the solutions provided by approximate approaches, with the aim of demonstrating how the latter yield equally valuable outputs. Despite our foremost objective is to accurately solve the stochastic dynamic scheduling problem, it is also crucial to rapidly obtain solutions, especially when it comes to larger scale problems. In fact, an additional goal of ours is to illustrate that such efficiency is exclusively attainable through the application of Approximate Dynamic Programming, as opposed to the exact paradigm. However, we do not entirely dismiss exact Dynamic Programming: we partially exploit it to solve a more intricate version of the scheduling problem that incorporates prioritizing rules. The rationale behind the introduction of priorities is to closely align to real-world applications. To this aim, not only we employ resolution methods that use Dynamic Programming but also present some heuristics.
The results of our thorough experiments allow us to conclude that the Approximate Dynamic Programming methods analysed in the dissertation perform admirably, outputting sub-optimal scheduling sequences that do not excessively deviate from the optimal ones obtained through the application of the exact Dynamic Programming paradigm. Some of the presented approaches even exhibit exceptionally fast execution times. Nonetheless, in our case, speed is a trait resulting from approximation, inevitably leading to a trade-off with lower performance. In fact, the determination of the most suitable method for solving an intralogistic robot scheduling problem is subjective. Among the several approaches we present, the decision on which to use is left to whom must address the real-life problem and is contingent on specific requirements related to speed and accuracy.

In conclusion, our implemented methods represent a valid choice for solving single-agent dynamic scheduling problems subjected to risk. We contribute to the state of the art by providing a novel perspective for the planning of intralogistic robot scheduling, demonstrating that the Dynamic Programming paradigm and its approximate counterpart reveal suitable alternatives to standard resolution methods.

Contents

Introduction

In the era of Industry 4.0, fueled by unprecedented technological advancements in robotics, we witness a transformative shift where intelligent machines revolutionize industrial processes, laying the foundation for a future defined by enhanced efficiency, productivity, and innovation.
Our dissertation delves into this framework by contributing to a European-funded international research project on Dynamic Agile production Robots that learn and optimize Knowledge and Operations (DARKO Project), whose overarching aim is to realize a new generation of energy-efficient and easy to deploy intralogistic robots, capable of highly dynamic motions and of operating safely within unknown and changing environments. Indeed, this thesis work has been conducted under the supervision of αHead Research, the research division of Spindox, a prominent company in the ICT sector actively engaged in the project.
In contrast to the current deployment of existing robots within production facilities, distribution centers, etc., DARKO proposes an innovative perspective. Consider the example of Amazon, a pioneer in deploying robots in its fulfillment centers. For over a decade, the colossal online retail store has been researching into a variety of robots to optimize its efficiency, in order to meet the relentlessly growing demands of the e-commerce market. Even with its most recent strategy, the company has leaned towards robots that are specifically designed to collaborate with humans and primarily serve as drivers of racks, leaving the more agile action of directly picking the objects on the shelves to human workers (Allgor et al. [2023]). Due to its proven success, the same strategy has been embraced by numerous fulfillment centers and ongoing research in the field of robots as collaborative rack drivers remains highly active (see Rimelé et al. [2022], Wang et al. [2022], Löfflet et al. [2023]). However, for analogous applications beyond the realm of e-commerce, it may become necessary for robots to independently perform dynamic motions as well, while humans take on supervisory roles. For this reason, the DARKO project directs its attention to agile robots capable of planning not only their routing, but the entire sequence of tasks involved in the transportation of objects in a warehouse. This topic aligns closely with other current research trends exploring more agile and flexible robots, which are showcasing a promising trajectory (Tipary and Erdős [2021], Babin and Gosselin [2021]).
Confident of the simultaneous advancements in the mechatronic and robotic fields, in this dissertation we specifically address the mathematical segment of the DARKO project dedicated to the task scheduling of the robots. For a realistic, yet simplified representation of the problem, we envision a scenario where a single robot is responsible for collecting

and transporting objects from specific locations to designated destinations in a warehouse, while being subjected to risk factors directly associated with the actions that it can perform. The main objective of this thesis work is to propose innovative methods based on the Dynamic Programming paradigm for a low-risk and time-efficient scheduling of tasks in the described scenario.
Among the various approaches commonly employed for the resolution of robot task scheduling, like genetic algorithms (Zacharia and Aspragathos [2005]), Dynamic Programming is notably one of the least represented in the literature. Nevertheless, we are convinced of its suitability for addressing the problem at hand. Indeed, it represents a more precise method compared to a purely dynamic one where decisions are made greedily, solely based on the best immediate action, without considering their impact on the future. Moreover, it is also preferrable to a static scheduling approach that, despite the thorough look-ahead, typically overlooks stochasticity, necessitating a replanning whenever an unexpected event occurs. Instead, during the application of a Dynamic Programming approach for the resolution of a stochastic scheduling problem, unforeseen realizations of risk factors do not jeopardize the previously computed optimal solution. These advantages motivate our attempt of employing the Dynamic Programming paradigm for our problem, despite its significantly heavy computational workload when it comes large scale instances.
The opening sections of this work accurately describe the problem setting and define the essential notation. Then, after a brief introduction of the general Dynamic Programming paradigm, the second chapter focuses on its exact application to both deterministic and stochastic versions of our scheduling problem. However, due to its excessive resource requirement the exact paradigm is only applied to small and medium-sized instances. In fact, it is only in the third chapter that we delve into the resolution of larger problems by applying faster Approximate Dynamic Programming approaches: Approximate Policy Iteration, Myopic Rollout, and Monte Carlo Tree Search. These approximate methods are validated on smaller instances through comparisons with the solutions output by the exact paradigm; subsequently, their performances are compared to each other on larger instances to determine the most accurate and efficient approach. In the fourth chapter we introduce the concept of priorities, mirroring some delicate real-world situations and propose and analyze three alternative resolution methods that we specifically implemented for the redefined problem. Finally, we conclude by summarizing considerations and findings, and by providing an overview on potential further developments on the topic.

Chapter 1

DARKO: the project and the problem definition

DARKO, which stands for Dynamic Agile production Robots that learn and optimise Knowledge and Operations, is an international research project funded by the European Union. The overarching aim of the project is to research and innovate for efficient and safe intralogistic robots in agile production. This main goal can be decomposed in five separate objectives: robots as components of intralogistic processes must possess energy-efficient elastic actuators to execute highly dynamic motions; be able to operate safely within unknown, changing environments; be easy (cost-efficient) to deploy; have predictive planning capabilities to decide for most efficient actions while limiting associated risks; and be aware of humans and their intentions to smoothly and intuitively interact with them (DARKO Project). This dissertation specifically focuses on the efficient and risk-aware scheduling objective, proposing an innovative solution method based on the paradigm of Dynamic Programming.

1.1 Scheduling problem definition

Agile logistic robots naturally find their employment in manufacturing plants or warehouses, that we can assume to be organized in storage areas from where components or objects must be picked and transported to other specific locations within the same warehouse, e.g., a conveyor belt. Trusting this assumption, let us consider a warehouse where each object type has its univocal storing *box* and there are designated destinations, referred to as *trays*, where said objects must be placed, as illustrated in the toy example in Figure 1.1.
The list of object types to be moved to the trays and the respective quantities are defined by a set of order lines, i.e., a *mission*, assigned to a robot, for example: "*collect 5 units of object A and put them in tray 1, 8 units of object D, and put them in tray 2, etc.*". Once received the order, the robot needs to schedule a time-efficient and minimum-risk sequence of tasks to complete it, while being constrained to a fixed maximum carrying capacity of c objects, regardless of their type.

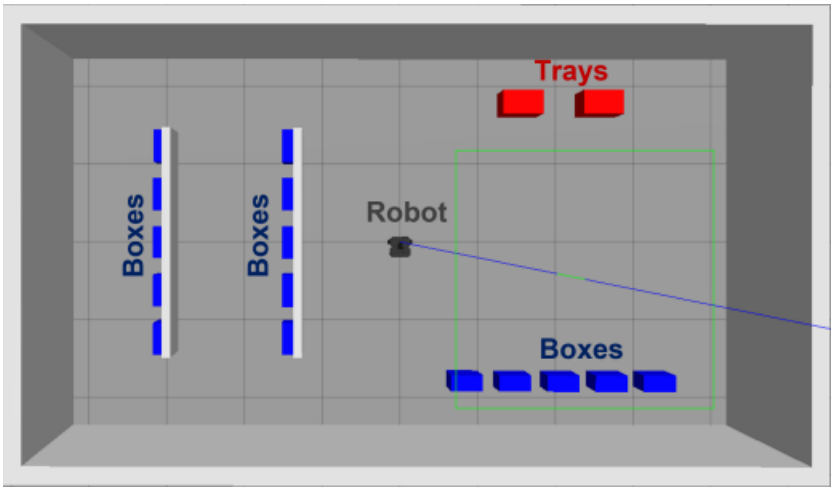


Figure 1.1: Toy example of DARKO use case.

The environment is indeed a three-dimensional space where a single agent, identified as the robot, can perform four main action types: it can move from one location to another, pick objects from the boxes, and place or throw them into the trays. Not all actions guarantee deterministic outcomes because collisions with humans or with shelving units may occur during navigation, while throws may fail. Collisions and failures represent indeed the exogenous risk factors affecting the system, and the consequences of their occurrence are, respectively, a time delay and the loss of the object whose throw was attempted.

1.1.1 The setting

Respecting the assumption of a well-organized plant, we solve the problem on a completely connected undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where each vertex $n \in \mathcal{N}$ represents either a picking location, uniquely associated to a box containing a specific object type, or a placing / throwing location. Placing and throwing vertices refer to the same vertices on the graph, but the different nomenclature is needed to understand that, while a throwing action can be performed from any throwing vertex, a placing action is only performed when the placing vertex is the closest to the tray where to place the object at hand. Another reason for the differentiation of placing and throwing actions is that the former are associated to no risk, while the latter may cause a failure with a probability dependent on the distance between the throwing vertex and the tray involved in the action. In the following, we will first face a deterministic problem in which throwing actions are discarded and there is no risk in navigation either; later, we will consider a stochastic problem where placing actions are substituted by throwing actions and navigation risks arise as well.
The set $\mathcal{E}$ of edges connecting the vertices represents the optimal paths to follow in terms of time-efficiency and risk-avoidance: in fact, the completely connected graph $\mathcal{G}$ is the result of a previously solved routing optimization problem. As depicted in Figure 1.2 each edge $e = (n_0, n_1) \in \mathcal{E}$ is indeed associated to two parameters: Δt_e, which is the time needed to go from the start vertex n_0 to the destination vertex n_1, and r_e, which is the

risk the robot will face travelling through e. While solving the deterministic problem we will discard the parameter r_e since we assume no navigation risk.
Since the routing optimization problem defining the edges of the graph is solved once and before the scheduling begins, both parameters r_e and Δt_e are assumed fixed during the scheduling problem resolution. This choice leads to a static view of the risk factors associated to moving actions, which are actually dynamic in the real world. Nevertheless, as we will later see, the time horizons for the mission's completion are generally set to be short in our experiments, and a unique snapshot of the initial situation is a close approximation of the risk throughout the entire scheduling. If deemed necessary, an option would be to repeatedly solve the routing problem and dynamically adjust the data associated with the edges of graph $\mathcal{G}$.

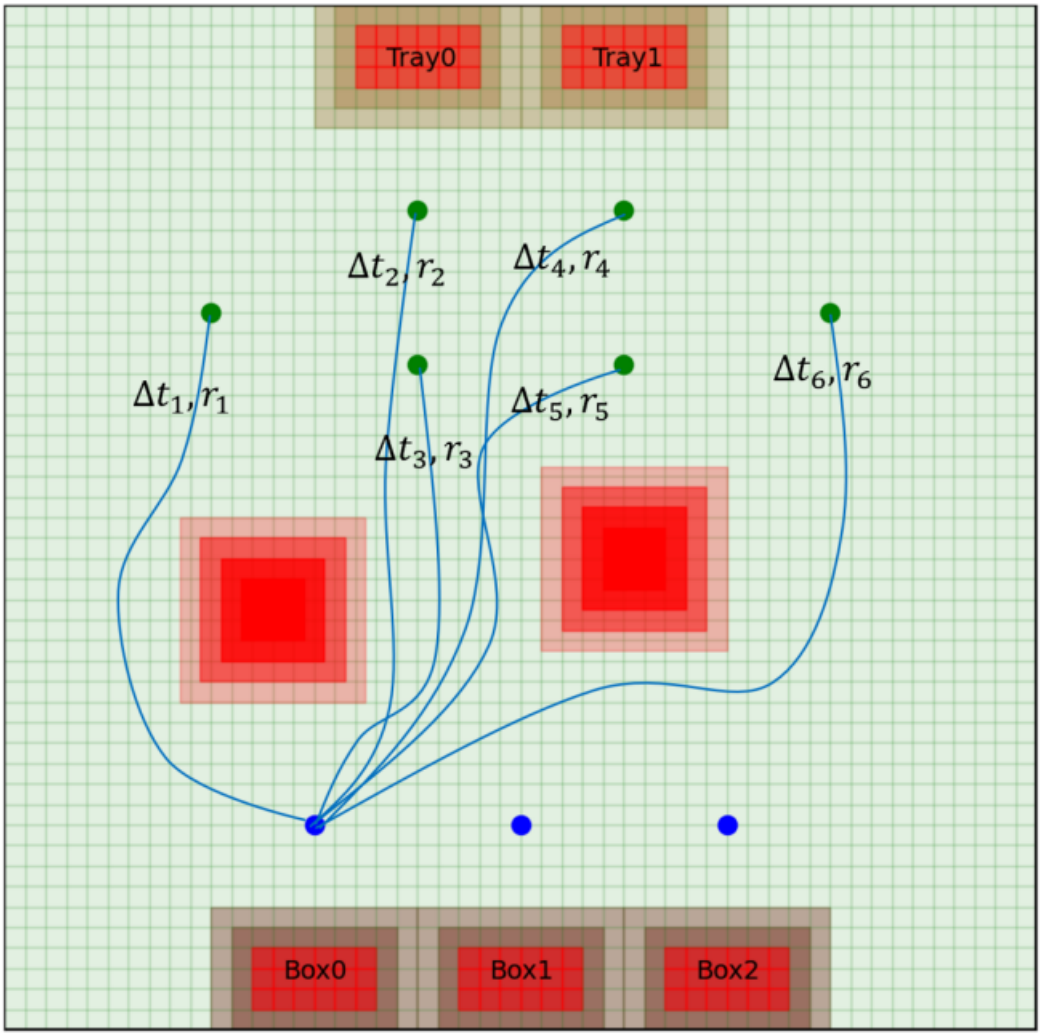


Figure 1.2: Partial representation of graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$: each edge $e \in \mathcal{E}$ is associated to the parameters pair $(\Delta t_e, r_e)$. Central red zones represent risky areas.

Notation

Let $\mathcal{O}$ be the set of O object types and $\mathcal{K}$ the set of K trays where objects might need to be placed into. Each object type is associated to a box where it is stored and from where it can be picked. Box locations are defined, for the sake of simplicity, by the picking vertices $n_o \in \mathcal{N}_{pick}$, while trays locations are associated to coordinates (x_k, y_k), $k \in \mathcal{K}$, that differ from the placing/throwing vertices $n_k \in \mathcal{N}_{place}$. Nevertheless each vertex $n \in \mathcal{N}$ is also associated to a pair of coordinates (x_n, y_n). Let then $\mathcal{T} = \{0,1,2,...,T\}$ be a set of discrete time instants, i.e., seconds, with T being the time horizon fixed for the mission's completion.
The mission assigned to the robot is defined by a set of order lines $\mathcal{M} = \{(k_m, o_m, q_m), m = 1, ..., M\}$, where q_m is the quantity of objects of type o_m to place in tray k_m.
We are ready to define the state space $\mathcal{S}$. Let $s \in \mathcal{S}$ be defined as an array of dimension $\Omega = 2 + O \times (K + 1)$, resulting from a concatenation of the following:

- $s_1 \in \mathcal{T}$, which represents the time elapsed from the beginning of the mission;
- $s_2 \in \mathcal{N}$, which represents the position of the robot on the graph at time s_1;
- $\underline{s}^o \in \mathbb{N}_0^{(K+1)}$, defined $\forall\, o \in \mathcal{O} = \{1, ..., O\}$, such that:
  - $\underline{s}^o_1 \in \{0, ..., \bar{O}\}$ is the number of objects of type o picked before time s_1, where $\bar{O} = \sum_{m=1}^{M} q_m \mid o_m = o$ represents the total number of objects of type o to pick during the mission;
  - $\underline{s}^o_{k+1} \in \{0, ..., \bar{O}_k\}$ is the number of objects of type o placed in tray $k \in \mathcal{K} = \{1, ..., K\}$ before time s_1, where $\bar{O}_k = \sum_{m=1}^{M} q_m \mid o_m = o,\ k_m = k$, represents the total number of objects of type o to place in tray k during the mission.

For example, let $O = 2$ and $K = 2$, so that $\Omega = 2 + 2 \times (2 + 1) = 8$. A generic state $s \in \mathcal{S}$ would be denoted as $s = (t \in \mathcal{T}, n \in \mathcal{N}, \underline{s}^1, \underline{s}^2)$ and, given a suitable mission $\mathcal{M}$, one could have $\underline{s}^1 = (4, 2, 1)$ and $\underline{s}^2 = (1, 1, 0)$. Note that each state records information about which objects have been picked and/or placed up to a specific time, and enables a deduction of the remaining tasks required to complete the mission.
Finally, let us define the action space. As previously stated there are four main action types: move, pick, place, throw, which can further branch off into more specific actions by associating to each type an additional set. For example, to a moving action type we shall associate a vertex where to move, and for a picking action type we shall specify which object has been chosen to be picked. We will work with a set $\mathcal{A} = \mathcal{A}_{move} \cup \mathcal{A}_{pick} \cup \mathcal{A}_{place} \cup \mathcal{A}_{throw}$, where $\mathcal{A}_{move} \subseteq \mathcal{N}$, $\mathcal{A}_{pick} \subseteq \mathcal{O}$ and $\mathcal{A}_{place}, \mathcal{A}_{throw} = \{(o_a, k_a), o_a \in \mathcal{O}, k_a \in \mathcal{K}\}$, and various subsets $\mathcal{A}_s \subset \mathcal{A}$ only containing the admissible actions given a state $s \in \mathcal{S}$. Furthermore, each action is associated to the time duration of its execution, namely $\Delta t_{pt} = 5$ time units for all placing and throwing actions, $\Delta t_p = 7$ for all picking actions, and Δt_e, as previously defined, for moving actions on edge $e \in \mathcal{E}$.

Chapter 2

Deterministic and stochastic toy models resolution through Dynamic Programming

The paradigm of Dynamic Programming (DP) has been chosen to innovatively solve the problem of intralogistic robot scheduling described in the previous chapter because of its notorious flexibility. Indeed, its applications span across various fields, from operations research to economics, from control theory to machine learning.
DP is not a fixed and defined algorithm, but rather an optimization principle, and as such its implementation for a specific problem may require a considerable customization effort (Brandimarte [2021]) that counterbalances its appealing flexibility. Furthermore, it is as flexible as computationally expensive: curses of dimensionality are its Achilles' heel, and it might prove impractical for larger scale problems. For this reason, in this thesis, exact DP will only be used to solve toy models that will then be employed as benchmarks to validate approximate implementations for larger and real scale instances.

2.1 The Dynamic Programming principle

In order to properly present and understand the DP approach, let us first consider a general discrete-time model describing a stochastic dynamic decision problem with finite horizon T, such that time flow is completely described by time instants $t = 0,1,...,T$ at which we observe the system and time intervals $t = (t-1,t]$, during which the system evolves. The fundamental components of the model are the states s of the system collected in a state space $\mathcal{S}$, and the sets $\mathcal{A}_s$ of admissible actions given a state, that, combined, define the whole action space $\mathcal{A}$. The transition from a state s_t of the system to the next one is defined by a state transition equation

$$s_{t+1} = g_{t+1}(s_t, a_t, w_{t+1}), \tag{2.1}$$

and depends on the state s_t itself, on the chosen action $a_t \in \mathcal{A}_{s_t}$, and on the realization of external risk factors w_{t+1} that may occur after the decision is made, during the subsequent time interval from instant t to instant $t+1$. The dependence on just the previous state of the system rather than on the whole path is a common assumption better known as Markovian property.
The described stochastic problem could be stated as

$$\text{opt } \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t C_t(s_t, a_t, w_{t+1}) + \gamma^T F_T(s_T)\right], \tag{2.2}$$

where $C_t(s_t, a_t, w_{t+1})$ is an immediate cost/reward function depending on the action chosen from state s_t and on the realization of risk factors, $F_T(\cdot)$ is a terminal cost/reward function that assigns values to terminal states at time horizon T, and $\gamma \in (0,1]$ is a discount factor.
The main idea of DP is to recursively solve a multi-stage dynamic decision problem as the one presented by decomposing it into smaller sub-problems. The key procedure is to evaluate states based on their appeal through the use of value functions $V_t : \mathcal{S} \to \mathbb{R}$, that measure the quality of being in a certain state at time instant t. As for actions, their quality is somehow evaluated through cost/reward functions $C_t(\cdot)$, as in the general formulation. More generally, cost/reward functions are referred to as immediate contributions and may be stochastic.
Let us now finally describe how the recursive resolution of the sub-problems is performed through a backward pass. In finite-horizon problems, a value is assigned to all possible terminal states based on their quality; then, for each previous state that would bring the system to a terminal one, the assigned value is the result of an optimization problem over the admissible actions. The objective of the sub-problem is the expectation of the sum of the immediate contribution and the discounted future state value, conditional to the current state and the chosen action. The process is then repeated until the initial state is reached.
In brief, what just described is the recursive application of the Bellman's Equation:

$$\begin{cases} V_T(s_T) = F_T(s_T) \\ V_t(s_t) = \underset{a_t \in \mathcal{A}_{s_t}}{\text{opt}} \ \mathbb{E}\big[C(s_t, a_t, w_{t+1}) + \gamma V_{t+1}(g_{t+1}(s_t, a_t, w_{t+1}))|s_t, a_t\big], \quad t = T-1, ...,0. \end{cases} \tag{2.3}$$

In conclusion, the final optimal solution given by the application of the Dynamic Programming principle is defined by one last pass, forward in time. Starting from a given initial state, for each time instant t at which a decision must be made, the final forward pass selects the optimal action as the argument satisfying (2.3).

2.2 Dynamic Programming application to toy models

Recalling the problem setting and notation introduced in the previous chapter, Section 1.1.1, what we are left to define to apply DP to the robot scheduling problem are terminal

value function, immediate contribution and state transition equation. The first has been chosen as

$$F(s) = (T - s_1) - \sum_o \Big(\sum_m (q_m | o_m = o) - \sum_k \underline{s}^o_{k+1} \Big) + \sum_o \underline{s}^o_1, \tag{2.4}$$

where the term $T - s_1$ linearly rewards the early completion of the mission with respect to the fixed time horizon T, the first summation over o is a penalization of one unit for every object that was supposed to be placed but has not (there is no penalty if the mission is completed before time horizon is reached), while the last summation is a prize of one unit for every picked object.
As for immediate contribution, the reward engineering behind its definition differs between deterministic and stochastic instance. Hence, while details are left to the next sections, let us just introduce the deterministic rewards for each type of action: $r_{\text{pick}} = 10$, $r_{\text{place}} = r_{\text{throw}} = 12$, $r_{\text{move}} = 0$.
Before defining the state transition equations ruling the dynamics of the system for the deterministic and stochastic instances, it is necessary to briefly dwell on time notation. In the first chapter, Section 1.1.1, time has been defined as the prime entry of the state of the system itself and denoted by $t \in \mathcal{T} = \{0,1,...,T\}$ where T is the fixed time horizon. On the other hand, the time duration of each action type has been defined as a time interval, whose span varies depending on the action. For this reason, the notation presented in Section 2.1 for a general dynamic decision problem must be slightly modified in order to use DP on our specific scheduling problem. Let the time flow be described by time instants $t \in \mathcal{T} = \{0,1,...,T\}$ and by uneven intervals $\Delta t \in \Delta = \{\Delta t_p, \Delta t_{pt}, \Delta t_e, e \in \mathcal{E}\}$, so that given a state of the system s_t at time t one can denote a succeeding state as $s_{t+\Delta t}, \Delta t \in \Delta$.
A consequence of this adjustment can be noted in (2.4): time indexing is missing from state notation s, as well as from terminal value function $F(\cdot)$, because a terminal state is such not only when the system has reached the time horizon, but also when the mission has been completed in advance. Since for the latter case an indexing by T would not be accurate, we opted for its dismissal.
We are now ready to define the state transition equation for the deterministic case, as we will later see in Section 2.2.2 that the stochastic case requires a few adjustments. Bearing in mind the time notation recently introduced, the state transition equation for a deterministic instance of our intralogistic robot scheduling problem can be generally written as

$$s_{t+\Delta t} = g_{t+\Delta t}(s_t, a_t), \quad \Delta t \in \Delta, \tag{2.5}$$

since there are no exogenous risk factors.
More specifically, the new state of the system $s_{t+\Delta t}$ at time $t + \Delta t$ will depend on the specific action a_t chosen at time t:

- if $a_t = n \in \mathcal{A}_{s_t,move}$, i.e., the action chosen is to move from s^2_t to n along $e = (s^2_t, n) \in \mathcal{E}$, then

$$s^i_{t+\Delta t} = s^i_{t+\Delta t_e} = \begin{cases} s^1_t + \Delta t_e, & \text{for } i = 1 \\ n, & \text{for } i = 2; \end{cases} \tag{2.6}$$

- if $a_t = o \in \mathcal{A}_{s_t,pick}$, i.e., the chosen action is to pick object $o \in \mathcal{O}$, then

$$s^1_{t+\Delta t} = s^1_{t+\Delta t_p} = s^1_t + \Delta t_p, \tag{2.7}$$

$$\underline{s}^{o,i}_{t+\Delta t} = \underline{s}^{o,i}_{t+\Delta t_p} = \begin{cases} \underline{s}^{o,i}_t, & \forall i \neq 1 \\ \underline{s}^{o,i}_t + 1, & \text{for } i = 1; \end{cases} \tag{2.8}$$

- if $a_t = (o,k) \in \mathcal{A}_{s_t,place}$, i.e., the chosen action is to place object $o \in \mathcal{O}$ in tray $k \in \mathcal{K}$, then

$$s^1_{t+\Delta t} = s^1_{t+\Delta t_{pt}} = s^1_t + \Delta t_{pt}, \tag{2.9}$$

$$\underline{s}^{o,i}_{t+\Delta t} = \underline{s}^{o,i}_{t+\Delta t_{pt}} = \begin{cases} \underline{s}^{o,i}_t, & \forall i \neq k+1 \\ \underline{s}^{o,i}_t + 1, & \text{for } i = k+1. \end{cases} \tag{2.10}$$

Note that all state entries whose transition has not been made explicit in Equations (2.6)-(2.10) do not vary when the respective action is performed.
Needless to say, if an action is not admissible, a state transition cannot take place. In general, an action is not admissible whenever the fixed time horizon would be exceeded if the action was executed, or whenever the robot is not in a suitable location to perform it; for example, object $o \in \mathcal{O}$ cannot be placed in tray k if the robot is not in vertex $n_k \in \mathcal{N}_{place}$. Moreover, there exist one more constraint for picking actions: in fact, they are not possible whenever the robot has reached its maximum capacity c.

Toy models definition

Finally, let us present the small scale problems chosen for the application of exact DP:

1. Mini instance:
   - time horizon $T = 120$,
   - $O = 3$ object types,
   - $K = 1$ tray,
   - $|\mathcal{N}| = 4$ graph vertices,
   - mission $\mathcal{M} = \{(1,1,3),(1,2,2),(1,3,2)\}$,
   - $Q = \sum_{m \in \mathcal{M}} q_m = 7$ total number of objects;
2. Medium-small instance:
   - time horizon $T = 200$,
   - $O = 5$ object types,
   - $K = 2$ trays,
   - $|\mathcal{N}| = 7$ graph vertices,
   - mission $\mathcal{M} = \{(1,2,1),(1,2,2),(2,1,2),(2,3,1),(2,4,3)\}$,
   - $Q = \sum_{m \in \mathcal{M}} q_m = 9$ total number of objects.

3. Medium instance:
   - time horizon $T = 230$,
   - $O = 3$ object types,
   - $K = 2$ trays,
   - $|\mathcal{N}| = 5$ graph vertices,
   - mission $\mathcal{M} = \{(1,1,3),(1,2,1),(1,3,2),(2,1,2),(2,2,2),(2,3,1)\}$,
   - $Q = \sum_{m \in \mathcal{M}} q_m = 11$ total number of objects.

For all instances, immediate deterministic rewards are set to the values introduced previously in this section, while the maximum capacity c of the robot is fixed to 4 objects.

2.2.1 Deterministic instance

The DP principle is mostly used to solve stochastic dynamic decision problems, since their deterministic counterparts are easily solvable with many other methods due to their static nature. In fact, deterministic decision problems are at most multiperiod, meaning that the decision making can entirely occur at time $t = 0$, even though actions may be performed in the future. Since there is no external interference, once a rigid strategy is defined it proves unnecessary to revise it over time. On the other hand, stochastic decision problems are multistage problems, meaning that decisions must be made dynamically and adapted after having observed the realization of external factors. In multistage problems it is not possible to define at time $t = 0$ an optimal strategy that will work for every sample path without being revised over time (Brandimarte [2021]).
For these reasons, it comes natural to seek the help of DP to solve multistage dynamic decision problems, rather than multiperiod ones. However, given our problem structure it is possible to treat its deterministic version as a multistage problem and employ DP for its resolution. In this manner we will establish a foundation, allowing us to gradually progress towards the resolution of the stochastic instance.
First of all, notice that the absence of external risk factors is explicitly captured in the state transition equation for a deterministic instance defined in (2.5), where the realization w is missing with respect to the more general definition of state transition presented in (2.1). Similarly, also immediate contributions are not affected by any exogenous factors, thus they are simply identified by the result of the multiplication of the previously defined deterministic rewards by a factor that decreases with time:

$$C_t(s_t, a_t) = \begin{cases} r_{\text{pick}} \cdot \frac{(2T - s_t^1)}{T}, & \text{if } a_t \text{ is a pick action} \\ r_{\text{place}} \cdot \frac{(2T - s_t^1)}{T}, & \text{if } a_t \text{ is a place action} \\ r_{\text{move}} = 0, & \text{if } a_t \text{ is a move action.} \end{cases} \tag{2.11}$$

The dependence on the time elapsed from the beginning of the mission, $s_t^1 = t$, is used to emphasize the importance of performing picking and placing actions at the earliest convenient opportunity.

The objective of the deterministic problem at hand can be generally stated as:

$$\max_{\pi\in\Gamma} \sum_{t\in\mathcal{I}} \gamma^t C_t(s_t, \pi_t(s_t)) + \gamma^I F(s_I), \tag{2.12}$$

where Γ is the set of admissible policies, and $\mathcal{I} \subseteq \mathcal{T}$ is an ordered set of increasing time instants[1], I being the last. A policy is a sequence of functions $\pi = (\pi_t)_{t\in\mathcal{I}\subseteq\mathcal{T}}$, such that each $\pi_t : \mathcal{S} \to \mathcal{A}$ maps a state of the system $s_t \in \mathcal{S}$ at time t to an admissible action $a_t \in \mathcal{A}_{s_t}$.
However, the paradigm of DP allows to solve (2.12) recursively, by applying the following adapted Bellman's equation:

$$\begin{cases} V_I(s_I) = F(s_I) \\ V_t(s_t) = \max\limits_{a_t\in\mathcal{A}_{s_t}} \left[C(s_t, a_t) + \gamma V_{t+\Delta t}(g_{t+\Delta t}(s_t, a_t))\right], \quad t \in \mathcal{I} \setminus \{I\}. \end{cases} \tag{2.13}$$

Results

Having defined the objective and all the components needed for the implementation of a DP method for the resolution of deterministic instances, we can finally present the results of the three chosen toy models. Refer to Appendix A.1.1 for further details on the outputs.

Problem Size	**State Space Cardinality**	**Mean Execution Time**[2]
MINI(1)	159 360	2.577 s
MEDIUM-SMALL(2)	3 136 000	71.11 s
MEDIUM(3)	7 937 300	106.91s

Table 2.1: Mean execution time over 10 runs of DP principle applied to deterministic instances of different sizes.

As expected, the results shown in Table 2.1 confirm that execution time increases with the increase of state space cardinality, but to what extent?
Let us consider the following larger scale model:

4. Large instance:
   - time horizon $T = 300$,
   - $O = 5$ object types,
   - $K = 3$ trays,
   - $|\mathcal{N}| = 8$ graph vertices,

[1] Of course $0 \in \mathcal{I}$, then the following time instant would be $t_1 = 0 + \Delta t, \Delta t \in \Delta$, and so on.

[2] Python Implementation on PyCharm Professional, Version 2022.3.2, on a machine equipped with RAM 16GB, intelCore i7, intel iRISx^e.

- mission $\mathcal{M} = \{(1,1,1), (1,2,2), (1,5,1), (2,2,2), (2,3,1), (2,4,3), (3,1,1), (3,2,1), (3,5,1)\}$,
- $Q = \sum_{m \in \mathcal{M}} q_m = 13$ total number of objects;
- maximum robot capacity $c = 4$;

Apparently it is only slightly larger than Instances 2. and 3., but its state space counts 176 150 400 elements and its resolution time exceeds 4 hours.
The above observations lead to the conclusion that, although DP can be used to solve deterministic multiperiod problems, it shall not be chosen over other faster, yet equivalent, optimization methods, like Linear Programming (Hillier and Lieberman [2021]), when trying to directly solve larger scale instances. On the other hand, a widely spread technique is to use it for breaking down a large problem into a sequence of smaller ones (Powell [2011]).

2.2.2 Stochastic instance

While for deterministic instances there are many equivalent methods that would output an optimal solution faster, for stochastic dynamic decision problems the span of choices is much tighter. Since the DP paradigm revealed successful for our problem in its deterministic version, it represents a promising resolution method for its stochastic counterpart as well. Moreover, it is essential to investigate the behaviour of exact DP before seeking for more time-efficient approximate approaches, in order to be able to make comparisons using the solutions given by the exact method as benchmarks.
With respect to the deterministic instance there are obviously more challenges, mainly regarding the dependence of state transition equation and immediate contributions on external risk factors. Indeed, for a better understanding of the role of such risk factors in the problem at hand, we analyse stochasticity and how it affects the definition of state transition and immediate contributions separately for throwing and moving actions. For what concerns picking actions, they are assumed to always succeed, therefore the state transition equation and the immediate contribution to them associated are fixed as in the deterministic instance (see (2.7), (2.8), (2.11)).

Stochasticity in throwing actions

As previously mentioned, for the stochastic version of our problem all placing actions are substituted by throwing actions, which, unlike the others, may fail. The probability of success of a throwing action from throwing vertex $n \in \mathcal{N}_{place}$ to tray k is given by:

$$p_{\text{throw}} = \begin{cases} 0, & \text{if } ||(x_n, y_n), (x_k, y_k)||_2 \geq 80 \\ \frac{1}{72}(80 - ||(x_n, y_n), (x_k, y_k)||_2), & \text{otherwise;} \end{cases} \tag{2.14}$$

hence, it is inversely proportional to the distance between the throwing location and the destination, and completely independent of the object thrown.

The possible outcomes arising from a throwing action are thus binary: success, denoted by $w_{t+\Delta t_{pt}} = 1$, is expected with probability p_{throw} as defined in (2.14), while failure, $w_{t+\Delta t_{pt}} = 0$, may occur with probability $1 - p_{\text{throw}}$.
The introduction of the exogenous risk of throwing failures leads to an adjustment in the state transition equation when the chosen action a_t is a throw:

$$s_{t+\Delta t_{pt}} = g_{t+\Delta t_{pt}}(s_t, a_t, w_{t+\Delta t_{pt}}). \tag{2.15}$$

Specifically the success of throwing action $a_t = (o, k) \in \mathcal{A}_{s_t,throw}$[3] leads to a next state as defined in (2.9) and (2.10), while the state transition in case of failure is ruled by the following:

$$s^1_{t+\Delta t_{pt}} = s^1_t + \Delta t_{pt}, \tag{2.16}$$

$$\underline{s}^{o,i}_{t+\Delta t_{pt}} = \begin{cases} \underline{s}^{o,i}_t, & \forall i \neq 1 \\ \underline{s}^{o,i}_t - 1, & \text{for } i = 1, \end{cases} \tag{2.17}$$

which means that the object of type o that the robot tried to throw is lost, e.g., it fell, hence it is subtracted from the count of objects of that type picked so far. As for the time elapsed for the action performance, it is consistent regardless of whether the outcome is successful.
Similarly, the immediate contribution for a throwing action is also affected by its stochastic outcome:

$$C_t(s_t, a_t, w_{t+\Delta t_{pt}}) = \begin{cases} 12\,\frac{(2T - s^1_t)}{T}, & \text{if } w_{t+\Delta t_{pt}} = 1 \\ 0, & \text{otherwise.} \end{cases} \tag{2.18}$$

Stochasticity in moving actions

The risk introduced in moving actions is intrinsic in the action itself. We recall from Chapter 1, Section 1.1.1 that the graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is completely connected and each edge $e \in \mathcal{E}$ is associated to two parameters: Δt_e and r_e, the travelling time and risk. Both are set as the results of a previously solved routing optimization and are assumed fixed throughout the resolution of the scheduling problem.
The parameter r_e actually indicates a risk percentage, hence the probability of having a failure $w_{t+\Delta t_e} = 0$, i.e., a collision, when travelling through edge $e \in \mathcal{E}$ is

$$p_{\text{move}} = \frac{r_e}{100}. \tag{2.19}$$

Instead, a smooth crossing of edge e happens with probability $1 - p_{\text{move}}$, and is denoted by $w_{t+\Delta t_e} = 1$.

[3]Recall that throwing actions can be performed from any throwing vertex $n \in \mathcal{N}_{place}$ to any tray $k \in \mathcal{K}$, while placing actions (that are deterministic, hence must always succeed by definition) can only be performed from a specific vertex $n_k \in \mathcal{N}_{place}$, that is supposedly the closest to tray k. For this reason it holds $\mathcal{A}_{s_t,place} \subseteq \mathcal{A}_{s_t,throw}$.

Exactly as highlighted in the previous subsection about the stochasticity of throwing actions, the introduction of an additional form of exogenous risk factor demands an additional definition of the state transition equation. Indeed, for a moving action $a_t = n \in \mathcal{A}_{s_t,move}$, that intends for the robot to transition through edge $e = (s_t^2, n) \in \mathcal{E}$, we have

$$s_{t+\Delta t_e} = g_{t+\Delta t_e}(s_t, a_t, w_{t+\Delta t_e}), \tag{2.20}$$

that, for $w_{t+\Delta t_e} = 0$, leads to:

$$s^i_{t+\Delta t_e} = s^i_{t+\Delta t_e+5} = \begin{cases} s_t^1 + \Delta t_e + 5, & \text{for } i = 1 \\ n, & \text{for } i = 2, \end{cases} \tag{2.21}$$

where the 5 time units added to the first entry of the state represent a penalty that can be thought as a delay due to the collision caused by the failure.
On the contrary, for $w_{t+\Delta t_e} = 1$, the state transition follows the same rule applied to the deterministic case (2.6).
Moreover, the distinct realizations of external factors also affect the definition of the immediate contribution for moving actions, according to:

$$C_t(s_t, a_t, w_{t+\Delta t_e}) = \begin{cases} 0, & \text{if } w_{t+\Delta t_e} = 1 \\ -2, & \text{otherwise.} \end{cases} \tag{2.22}$$

In conclusion, the overall general objective of the stochastic problem is the following:

$$\max_{\pi \in \Gamma} \mathbb{E}\left[\sum_{t \in \mathcal{I}} \gamma^t C_t(s_t, a_t, w_{t+\Delta t}) + \gamma^I F(s_I) \right], \tag{2.23}$$

where Γ and $\mathcal{I}$ are respectively the policy and time indexing sets as defined for the deterministic objective (2.12). In this case, the Bellman's equation needed for the problem resolution through DP includes, as in its general definition, an expected value:

$$\begin{cases} V_I(s_I) = F(s_I) \\ V_t(s_t) = \max\limits_{a_t \in \mathcal{A}_{s_t}} \mathbb{E}\big[C(s_t, a_t, w_{t+\Delta t}) + \gamma V_{t+\Delta t}(g_{t+\Delta t}(s_t, a_t, w_{t+\Delta t}))|s_t, a_t\big], \quad t \in \mathcal{I} \setminus \{I\}. \end{cases} \tag{2.24}$$

Results

As for the deterministic instance, we now present the results output by the application of the DP paradigm to the three toy versions of the above-described problem, stochastic in both moving and throwing actions. Refer to Appendix A.1.2 for further details on the outputs.
The results in Table 2.2 further validate the thesis of the curse of dimensionality of state space from which the DP paradigm suffers. However, if for a deterministic problem the

use of DP is not necessary, since there exist many other approaches to solve it exploiting its actual static nature, for a stochastic problem it is instead worth investigating deeper. The strong belief that DP is a valid and promising technique for the resolution of our specific problem, leads us indeed to the next chapter, where we present various approximate methods and analyse their outputs with validations and comparisons.

Problem Size	**State Space Cardinality**	**Mean Execution Time**[4]
MINI(1)	159 360	$3.18s$
MEDIUM-SMALL(2)	3 136 000	$204.3s$
MEDIUM(3)	7 937 300	$255.2s$
LARGE(4)	176 150 400	$4h22m$

Table 2.2: Mean execution time over 10 runs[5] of DP principle applied to stochastic instances of different sizes.

[4]Python Implementation on PyCharm Professional, Version 2022.3.2, on a machine equipped with RAM 16GB, intelCore i7, intel iRISx^e.

[5]Except for large instance, whose execution time is given by just one run, due to resources constraints.

Chapter 3

Approximate Dynamic Programming for large scale stochastic instances

As we earlier mentioned in Section 2.1 and further proved with the results presented in Sections 2.2.1 and 2.2.2, DP suffers from a demanding computational expense when it comes to larger scale problems. For this reason, it reveals necessary to allow sub-optimal solutions in exchange for faster outputs, and to this aim we resort to Approximate Dynamic Programming (ADP). ADP is a collection of strategies for solving problems that suffer from the three curses of dimensionality: exponential growth of 1) state, 2) outcome and 3) action spaces, given an increase in the variables dimension (Powell [2011]).
Luckily, our specific problem does not suffer from the curse on the outcome space since outcomes are simply binary. As for the action space, its cardinality is $|\mathcal{A}| = |\mathcal{N}| + O + O \times K$, where the number of vertices $|\mathcal{N}|$ in graph $\mathcal{G}$ defines the number of all possible moving actions, the amount of object types O represents the picking actions, and the last multiplicative term reflects the number of throwing actions. As a result, the action space cardinality does grow in all variables that define it, yet just linearly in all of them, and this cannot be referred to as a curse. In fact, the real curse is cast on the state space. A mission defines the total number of objects to collect before time horizon T, ergo, a larger mission extends the range of potential values for the state entries: suppose that for each object type there were q items to pick, then the state space cardinality would increase to more than $T \cdot (q^O)$. Other than the exponential growth produced by the state entries accounting for the number of objects picked (and placed), that causes the curse to happen in the first place, an extended time horizon, being time an entry of the state, significantly affects the state space cardinality as well.
It is now clear what motivates us to discard exact DP and seek for approximate techniques. The term ADP refers to a vast variety of approaches, but in this chapter we focus on two specific ones: Value Function Approximation (VFA), and Lookahed Policies (LAPs). The former aims at finding approximations of value functions, as the name suggests, while the latter are used to optimize a problem over a restricted horizon in order to

capture the impact of current decisions on the future. What unites these approaches and generally characterizes ADP as a whole is an algorithmic strategy that steps forward in time. While the fundamental backward pass of exact DP permits a thorough exploration that simply assigns to each state or state-action pair a precise value, a learning procedure through a forward pass requires the definition of sampling and decision-making rules for its exploration phase. Furthermore, a single forward pass alone would not lead to any learning, in fact, the process must be repeated iteratively. Despite the impossibility of visiting the entire state space, a sufficient number of iterations and an adequate rule for updating the approximations enable an effective learning process.
Before digging into specific methods of VFA and LAPs and analyzing their performance on various instances of our problem, let us introduce some notation and a few key aspects of a general procedure with the help of an example (borrowed from Powell [2011]) using the one-step transition matrix $Q(s)$, whose rows collect all the possible subsequent states given state s, based on the action chosen. Let $\bar{V}_t(s_t)$ be the approximation of the value function at time t that must be estimated, and assume to have clever initial approximations (otherwise simply set $\bar{V}_t^0 = 0 \ \ \forall\, t$). At iteration j and time t there are three fundamental steps to execute:

1. Decision making:
$$a_t^j = \arg\max_{a \in \mathcal{A}_{s_t}^j} C_t(s_t^j, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s_t^j, a) \bar{V}_{t+\Delta t}^{j-1}(s'), \tag{3.1}$$
which is carried out exploiting the information collected until previous iteration $j-1$ and provides just a partial overview of the whole system.
Be also aware that it is usually bold to assume to be able to compute all one-step succeeding states, and even bolder to assume to know the conditional probability of their occurrence. However, for our problem, it is not prohibitive to exploit the one-step transition matrix since, as noticed, the action and outcome spaces do not explode and all the exogenous factors distributions are known.

2. Value function approximations update:
$$\bar{V}_t^j(s_t) = U\big(s_t, a_t^j, \bar{V}_t^{j-1}(s_t)\big) = \begin{cases} \max\limits_{a \in \mathcal{A}_{s_t}} C_t(s_t, a) + \gamma \sum\limits_{s' \in \mathcal{S}} \mathbb{P}(s'|s_t, a) \bar{V}_{t+\Delta t}^{j-1}(s') \text{ if } s_t = s_t^j \\ \bar{V}_t^{j-1}(s_t) \ \text{ otherwise,} \end{cases} \tag{3.2}$$
which only occurs for current state s_t^j for this naive example, although it would be more useful to have a clever way to also estimate the value of being in states that have not been visited.

3. Simulation:
$$s_{t+\Delta t}^j = g_{t+\Delta t}(s_t^j, a_t^j, w_{t+\Delta t}^j), \tag{3.3}$$
which outputs the next state given the just chosen optimal action and the realization of external factors. Note that for our problem, sampling is straightforward since the distributions of both throwing and moving failures are known. In other cases, one could resort to computer or Monte Carlo simulations, or to real world processes.

The efficacy of all three steps of this example is affected by a few drawbacks, some of which can be mitigated with a smart understanding of the problem structure, of its definition and assumptions, as it somehow happens for the use of the transition matrix in Step 1 and for the sampling of Step 3. On the other hand, Step 2, for example, can only be improved by tailoring a more effective updating rule and eventually redefining the approximating functions' structure. This would certainly lead to a more general learning of the system, that would reveal even more effective if paired with a decision-making rule based on exploration too, rather than solely on exploitation.
Now that the fundamental requirements for the implementation of an efficient approximate algorithm have been clarified, it is time to introduce three specific methods. The plan is to test their performances and compare them to the exact solution, when feasible, and to each other, especially when it comes to larger scale problems. We will first present the methods separately based on the general approach they follow, and then present the results altogether at the end of the chapter.
It was implicit so far, but from now on we will solely focus on the stochastic instance of the problem, it being the most realistic.

3.1 Approximate Policy Iteration

Approximate Policy Iteration (API) falls under the category of VFA techniques, despite its name may suggest otherwise. In fact, its ultimate goal is to find an optimal or sub-optimal policy π, which is actually directly determined by value function approximations, such that:

$$\pi_t(s_t) = \arg\max_a \sum_{w \in \mathcal{W}_a} \mathbb{P}(w)\Big[C_t(s_t, a_t, w) + \gamma \bar{V}_{t+\Delta t}(g_{t+\Delta t}(s_t, a_t, w))\Big], \tag{3.4}$$

where $\mathcal{W}_a$ is the outcome space given action a, that, for our problem is either $\mathcal{W}_a = \{0,1\}$ for throwing and moving actions, or $\mathcal{W}_a = \{1\}$ for picking actions.
To accomplish the goal, API iteratively undergoes two phases: policy evaluation and policy improvement. The former is performed with a (many) forward in time simulation(s) at the end of which a backward pass on the visited states allows for the learning of the value functions defining the current policy. The latter consists in updating the policy based on the information gained during the evaluation phase.
Prior to delving into the details of the API algorithm, it is necessary to define the approximation technique for the value functions, and consequently identify a suitable updating rule. For the resolution of our specific problem, we have chosen to approximate the value functions with the following parametric model using Basis Functions (BFs):

$$\bar{V}_t(s_t) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(s_t), \tag{3.5}$$

where $\mathcal{F}$ is the set of features identifying the BFs $\phi_f(\cdot)$, and $\theta_f, f \in \mathcal{F}$ are the associated parameters. Note that in (3.5) there is no explicit dependence on time for either the parameters or the BFs, even though the latter depend on the state of the system, which does depend on time. In a more general framework time-dependence should be explicitly

considered for both the parameters and the BFs, but we will later discover that for our problem this is not necessary.
The approximations are thus linear and we can use Recursive Least Square (RLS) to update the parameters. Specifically for the implementation of our API algorithm, we refer to the RLS algorithm proposed in Powell [2019]. Let $\bar{V}_t^j(s_t) = \bar{V}_t(s_t|\theta^j) = \phi(s_t)^{\mathrm{T}}\theta^j$ be the approximation in vector form of the value function at time t and iteration j, and let $\hat{v}^j$ denote a real observation of the value of state s_t at iteration j. The update of the parameters is defined by:

$$\theta^j = \theta^{j-1} - H^j\phi(s_t)\hat{e}^j, \tag{3.6}$$

where

$$\hat{e}^j = \bar{V}_t(s_t|\theta^{j-1}) - \hat{v}^j \tag{3.7}$$

and

$$H^j = \frac{1}{\lambda^j}B^{j-1}, \tag{3.8}$$

with

$$B^j = B^{j-1} - \frac{1}{\lambda^j}\Big(B^{j-1} - \phi(s_t)(\phi(s_t))^{\mathrm{T}}B^{j-1}\Big) \tag{3.9}$$

and

$$\lambda^j = 1 + (\phi(s_t))^{\mathrm{T}}B^{j-1}\phi(s_t). \tag{3.10}$$

Having defined the updating rule for the parameters associated to the BFs, what is left to present is the choice of the BFs themselves.

3.1.1 Features and reward engineering

The selection of the BFs is a very critical step when dealing with approximations through parametric models as the one defined above, since they must reflect intrinsic features (from which their alternative name) of the system, and poor choices may lead to poor approximations. For this reason we dedicated a generous amount of time to the analysis of the exact values output by the DP algorithm. We observed their dependence on the state entries and tried to capture significant behaviours, in order to replicate such behaviours within the BFs. To this aim we created a new instance of the problem, with the unique scope of using it to study the above-mentioned aspects, that is:

5. API Test instance:
   - time horizon $T = 300$,
   - $O = 2$ object types,
   - $K = 2$ trays,
   - $|\mathcal{N}| = 4$ graph vertices,
   - mission $\mathcal{M} = \{(1,1,4), (1,2,4), (2,1,4), (2,2,4)\}$,
   - $Q = \sum_{m\in\mathcal{M}} q_m = 16$ total number of objects,
   - maximum robot capacity $c = 4$.

Its peculiarity is being characterized by a mission with fewer object types but more items to collect with respect to the instances presented in Chapter 2, Section 2.2.1, and this permits, in some cases, to capture more information.
We also defined and used the following new terminal value function:

$$F(s) = 5(T - s_1) - r_{\text{throw}} \sum_o \Big(\sum_m (q_m | o_m = o) - \sum_k \underline{s}^o_{k+1} \Big) + r_{\text{pick}} \sum_o \underline{s}^o_1, \tag{3.11}$$

for reasons that will later become clear.
The analyses consist of various plots displaying the behaviour of the exact values associated to each state, obtained through the DP paradigm application to the stochastic version of API Test instance(5), when a state entry is let free while all others are fixed, in order to create 2-dimensional visualizations for a better understanding.
The heaviest dependence of exact values is the one on the first entry of the state of the system, i.e., the dependence on time, clearly visible from Figure 3.1, that shows the monotone decrease of the exact values when the time elapsed from the beginning of the mission increases.

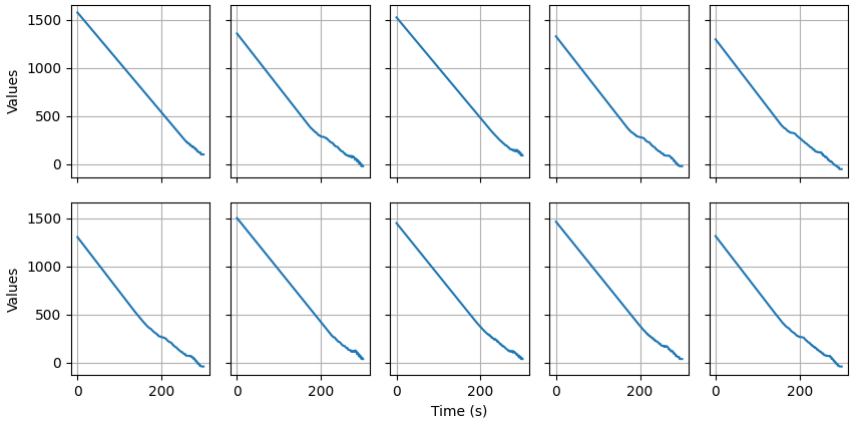


Figure 3.1: Value functions dependence on time. Each one of the ten plots represents the variation of the value associated to a randomly chosen state when varying its time entry.

A less obvious behaviour is captured in Figure 3.2, and is influenced by the position of the robot on the graph. Out of the ten plots, three display slightly higher values when the robot is in a placing/throwing vertex, while six show higher values associated to picking locations. Despite such variations could be considered almost negligible, they provided us with a valuable clue. The thesis is that higher values are assigned to a state associated to a placing location only if the robot is carrying a considerate number of items that have yet to be placed; on the contrary, picking vertices are more promising when the robot is far from reaching its carrying capacity. Convinced of this intuition, we plotted six graphs showing the values dependence on the positional entry of states for which the robot has reached maximum capacity, and six graphs for when it is empty-handed: Figures 3.3a and 3.3b confirm the thesis.

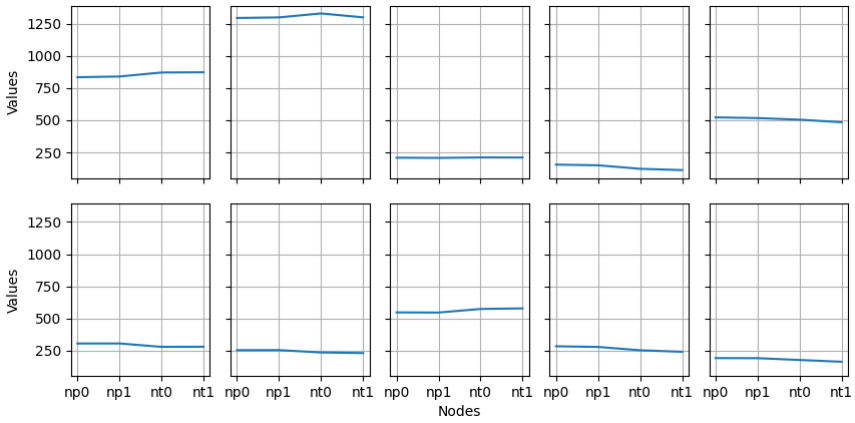
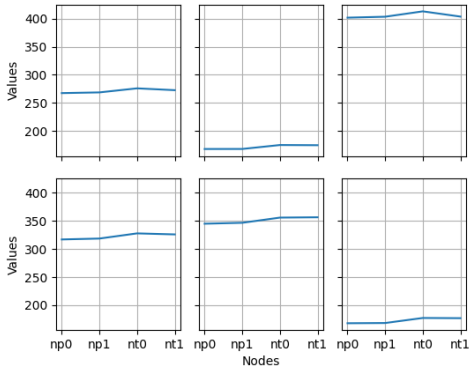
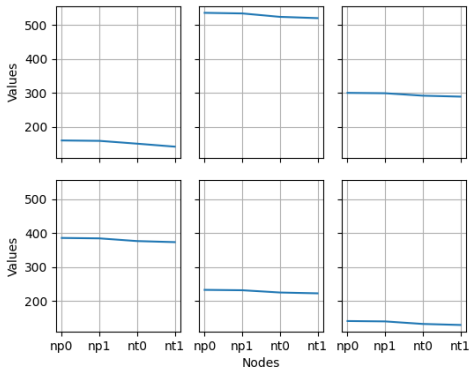


Figure 3.2: Value functions dependence on position. Each one of the ten plots represents the variation of the value associated to a randomly chosen state when varying its positional entry.



(a) Value functions dependence on position, when the robot has reached maximum capacity.

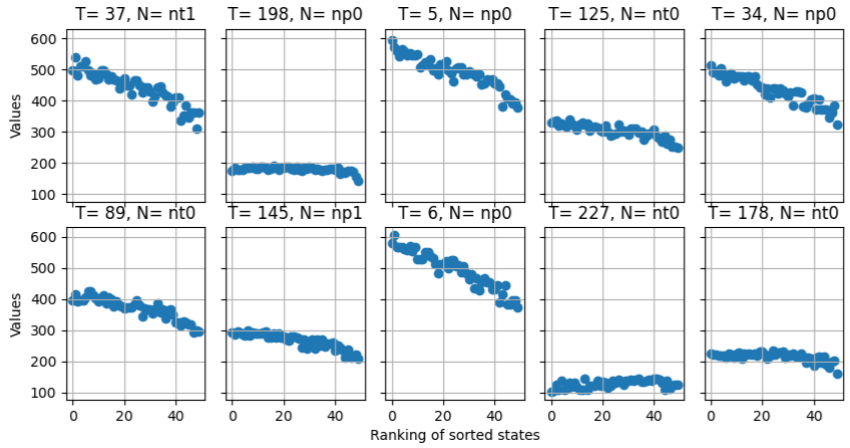


(b) Value functions dependence on position, when the robot is carrying no items.

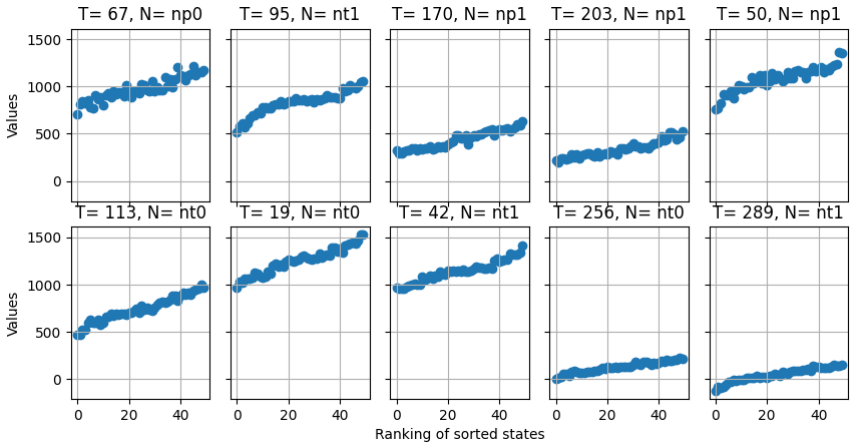
Figure 3.3: Each one of the twelve plots represents the variation of the value associated to a randomly chosen state between the ones identifying situations where the robot has reached maximum capacity (a) or where the robot is empty-handed (b), when varying its positional entry.

As for the other entries of the state of the system there is no intuitive influence on the exact values associated to the states. However, one last interesting fact regards the dependence of the values on the total number of objects picked and on the total number of objects placed when changing the terminal value function. To analyse this kind of dependence a new plot is created: 50 random states with fixed time and position are generated and sorted in increasing order based on the total number of objects picked and/or placed; then, the behaviour of the values with respect to the state sorting rule is observed. Results are

shown in Figures 3.4a and 3.4b, for the total number of picked objects only, since for the total number of placed objects the values present a similar behaviour.



(a) Value functions dependence on total number of objects picked, when using the old definition of terminal value function.



(b) Value functions dependence on total number of objects picked, when using the new definition of terminal value function.

Figure 3.4: Each one of the twenty plots represents the variation of the values associated to randomly chosen states with fixed time and position when sorted in increasing order of total number of objects picked. Figure (a) shows the behaviour caused by the use of terminal value function (2.4), while Figure (b) the one caused by (3.11).

If the terminal value function defined in (2.4) is used we notice a slight monotonic decrease of the exact values when a moderate amount of time is elapsed, which is cancelled, if not inverted, when approaching the time horizon. On the other hand, terminal value function (3.11) guarantees a monotone increasing behaviour in all situations. Since this general analysis reflects the more specific dependence of the values on the state entries that refer

to the number of items picked and placed, separately for object type and tray, the second terminal function is preferred for an API algorithm through basis function because it leads to more linear and monotone behaviours overall.
Based on the results of the conducted analyses we identified the following BFs:

- $\phi_1(s) = \begin{cases} 1 \text{ if } 0 \leq s^1 \leq \frac{T}{3} \\ 0 \text{ otherwise;} \end{cases}$
- $\phi_2(s) = \begin{cases} 1 \text{ if } \frac{T}{3} \leq s^1 \leq \frac{2T}{3} \\ 0 \text{ otherwise;} \end{cases}$
- $\phi_3(s) = \begin{cases} 1 \text{ if } \frac{2T}{3} \leq s^1 \leq T \\ 0 \text{ otherwise;} \end{cases}$
- $\phi_4(s) = s^1$;
- $\phi_{4+i}(s) = s^{2+i}, \quad \text{for } i = 1, .., \Omega - 2;$
- $\phi_{2+\Omega+n}(s) = \begin{cases} 1 \text{ if } s^2 = n \in \mathcal{N} \\ 0 \text{ otherwise,} \end{cases} \qquad \text{for } n = 1, ..., |\mathcal{N}|;$
- $\phi_{2+\Omega+|\mathcal{N}|+n_t}(s) = 1$ if $s^2 = n_t \in \mathcal{N}_{place} = \{1, ..., |\mathcal{N}_{place}|\}$ and the robot is at maximum carrying capacity or it has already collected all items requested by the mission, while some still need to be placed; it is null otherwise;
- $\phi_{2+\Omega+|\mathcal{N}|+|\mathcal{N}_{place}|+o}(s) = 1$ if $s^2 = n_o \in \mathcal{N}_{pick}$, the robot has not reached maximum capacity, and items of object type $o \in \mathcal{O} = \{1, ..., O\}$ still need to be picked; it is null otherwise.

The first three BFs, ϕ_1, ϕ_2, ϕ_3, define the time-varying intercept of the model, in fact they are indicator functions associated to three discretized intervals of the time horizon; ϕ_4 and ϕ_{4+i} simply return the entries of the state except for the positional entry that is one-hot encoded by the functions $\phi_{2+\Omega+n}$, because the positional entry is actually defined with a string. However, we can also sort vertices and identify them by their index, mirroring the notation used in the BFs definition. The last two types of BFs are the direct consequence of our analysis on the effects caused by the position of the robot on the value functions behaviour. They both are a collection of indicator functions whose value depends on the robot being in a picking or placing vertex and on the collected number of items related to its maximum capacity.
Moreover, as a consequence of a further analysis on the performances of the API algorithm with the VFA through the above-introduced BFs, the deterministic immediate rewards have been changed to $r_{\text{pick}} = 20, r_{\text{throw}} = 25, r_{\text{move}} = 0$, affecting the definition (3.11) of the new terminal value function and of the immediate contributions (2.11), (2.18) chosen for implementation.

Algorithm 1 API with BFs

Initialization. Set initial state S, initial parameters $\theta_f^0 = 0,\ \forall f$
Iteration counter. $j \leftarrow 1$
while $j \leq 2000$ **do**
  Simulation counter. $m \leftarrow 1$
  while $m \leq 10$ **do**
    Time index. $t \leftarrow 0$
    Initial state reset. $s_t^j \leftarrow S$
    while s_t^j is **not** terminal **do** ▷ Forward Simulation
      $a_t^j \leftarrow \begin{cases} \text{random.choice}(\mathcal{A}_{s_t}^j), & \text{if random}() \leq \mathcal{E} \\ (3.1), & \text{otherwise} \end{cases}$
      Exogenous factor realization. Simulate outcome $w_{t+\Delta t}^j$
      $s_{t+\Delta t}^j \leftarrow g_{t+\Delta t}(s_t^j, a_t^j, w_{t+\Delta t}^j)$
      $t \leftarrow t + \Delta t$
    end while
    $\hat{v}_t^j \leftarrow F(s_t^j)$
    $t \leftarrow t - \Delta t$
    while $t \geq 0$ **do** ▷ Backward pass
      $\hat{v}_t^j \leftarrow C_t(s_t^j, a_t^j) + \gamma \hat{v}_{t+\Delta t}^j$
      Parameters update. Update θ_f^j with RLS ((3.6) - (3.10))
      $t \leftarrow t - \Delta t$
    end while
    $m \leftarrow m + 1$
  end while
  $\mathcal{E} \leftarrow 0.995 \times \mathcal{E}$
  $j \leftarrow j + 1$
end while

3.1.2 Exploration in decision making

The use of the BFs approach for approximating the value functions in our API algorithm satisfies the need of having function approximations that could output updated and reasonable values for all states, even for the ones that are not visited during the simulations. This requirement was introduced in the beginning of the chapter when listing the fundamental steps of a general ADP paradigm (Step 2) but it was not the only drawback of the proposed example. In fact, a closer look on Step 1 steers the attention to the problem of making decisions solely exploiting the information collected in the previous iterations. This technique for choosing the best action to perform may guide the algorithm to an impasse, where the most promising states are simply the ones that are visited more frequently: there might be better states, whose potential is not known because they have never been visited, that will keep being discarded.
It becomes essential to define an alternative decision-making rule that accounts for the exploration of states beyond the ones that just seem more promising. To this aim we define an exploration parameter $\mathcal{E} \in (0,1)$ that defines the probability under which the decision is randomly made between all the available actions. On the other hand, with probability $1-\mathcal{E}$, the best action is chosen based on past experience, hence following the naive exploitation-based decision making defined by Equation (3.1).
In conclusion, we opt for an $\mathcal{E}$-annealing exploration rule, that causes the gradual decrease of the parameter as the algorithm gains more information about the problem at hand; specifically, $\mathcal{E}$ is scaled by a multiplicative factor less than 1, that we set to 0.995.

3.1.3 The algorithm

Finally, we can present the algorithmic structure of the Approximate Policy Iteration method with Basis Functions, whose results are presented at the end of the chapter, in Sections 3.3 and 3.4.
Pseudo-Algorithm 1 collects the fundamental passages of our API approach. Note that the forward simulation process is repeated 10 times (*line* 5), and each time is accompanied by a backward pass (*line* 16-20) during which, at each iteration, a parameters update is performed (*line* 18). However, the decision-making rule used in *line* 8 keeps depending on the old parameters θ_f^{j-1}, that indeed define $\bar{V}_{t+\Delta t}^{j-1}(s') = \phi(s')^{\mathrm{T}}\theta^{j-1}$ in Equation (3.1). The updated BFs parameters θ_f^j are then used in the next iteration $j+1$, after the exploration parameter $\mathcal{E}$ has been revised as well (*line* 23).
Of course, once the learning procedure has terminated, the obtained approximations are used to finally make the actual decisions for the problem.

3.2 Lookahead Policies

As mentioned when introduced, Lookahead Policies manage to capture the impact that making a certain decision at a certain moment has on the future. There are various approaches to collect such information, but it is common to simulate a path with a predetermined and limited number of time steps in the future. Then, exploiting the information captured on the future effects of many possible decisions, LAPs help to choose which

decision is actually the best at that certain moment.
They are simple and are usually used when other more complex approximation methods seem to not be working. With LAPs, it is not necessary to define any approximating structure for value or policy functions, but it is sufficient to produce a decision rule by roughly peeking in the future.
We decided to also investigate this kind of approach because, as we will later see, despite the run-times of the API algorithm are noticeably faster than the ones of the DP paradigm, still their speed is not exceptional. In general, LAPs should be computationally hard since they are technically brute-force methods (Powell [2019]), but the two we opted to implement are as naive as fast, nevertheless quite accurate. Furthermore, we were guided by the intuition that they would prove suitable for our problem, given its limited action space, and even more restricted outcome space, where there are at most two elements per action type, one of which presents a much higher probability of happening: it seems like a prefect context for simulations to provide valid estimate of what could really happen.
Based on this intuition we decided to apply to our dynamic scheduling problem two Lookahead Policies: a Myopic Rollout (MR), an improved version of the most naive existing approach, and a Monte Carlo Tree search (MCTS), a slightly more complex and clever method, which employs MR itself in its implementation.

3.2.1 Myopic Rollout

The adjective *Myopic* describes the activity of making a decision just by looking roughly into the future, without a crystal ball. Note that we introduced MR as an improved version of a more naive algorithm: we were referring to a Myopic Policy that only relies on the values of immediate contributions to define its decision-making rule. The improvement is reflected in the noun *Rollout*, which stands for the recursive procedure of rolling to the next state after making the myopic decision and then repeating the process for a fixed number of steps. During this phase an estimate of the value of being in the state from where the rollout has started is produced, based on a probable future path. If, given a state s, the MR procedure is performed for all the states deterministically reachable from s, a sub-optimal action can be chosen as the one that would lead the system to the succeeding state with the highest value produced.
The approach is thus almost entirely portrayed by its own name, and this fact suggests the simplicity of the algorithm, whose implementation only requires the definition of a few parameters. It is essential to define how far in the future to "roll" and how to myopically choose between a set of available actions. For the decision rule we simply opted for a Myopic Policy as the one previously mentioned, that outputs an action solely based on its deterministic immediate contribution, i.e.,:

$$a_t^\star = \arg\max_{a \in \mathcal{A}_{s_t}} C_t(s_t, a) = \arg\max_{a \in \mathcal{A}_{s_t}} \cdot \frac{r_a(2T - s_t^1)}{T}, \tag{3.12}$$

where $r_a \in \{r_{\text{pick}} = 10, r_{\text{throw}} = 12, r_{\text{move}} = 0\}$ depends on action a.

Algorithm 2 Decision making through MR

procedure BESTDECISION(s)
  BestValue $\leftarrow 0$
  for $a \in \mathcal{A}_s$ **do**
    $s' = g_{t+\Delta t}(s, a)$
    $V' =$MYOPICROLLOUT$(s', 0)$
    if $V' >$ BestValue **then**
      BestValue $\leftarrow V'$
      BestAction $\leftarrow a$
    end if
  end for
  return BestAction
end procedure

procedure MYOPICROLLOUT(s, r)
  $a =$ (3.12)
  Contribution $= C_t(s, a)$
  if $r \leq R$ **then**
    Exogenous factor realization. Simulate outcome w
    $s' = g_{t+\Delta t}(s, a, w)$
    RealContribution $= C_t(s, a, w)$
    $r \leftarrow r + 1$
    if s is **not** terminal **then**
      $V' =$MYOPICROLLOUT(s, j)
      $V =$ RealContribution $+\gamma V'$
    else $V = F(s)$
    end if
  else $V =$ Contribution
  end if
  return V
end procedure

The choice of the myopic decision rule directly affects the estimates of the values of the states produced during the rollout. In fact, after every myopic decision, the estimates are recursively defined as:

$$\bar{V}(s_t) = C_t(s_t, a_t^\star, w_{t+\Delta t}) + \gamma \bar{V}(g_{t+\Delta t}(s_t, a_t^\star, w_{t+\Delta t})). \tag{3.13}$$

As for the number of recursion steps to perform, denoted by R, its choice may vary depending on the problem: in general R should increase with the problem size. However, its choice is slightly more complicated. In fact, we noticed that an increase in the value of parameter R does not negatively affect the performances of the algorithm on small scale problems, but, if significant, it may worsen them for larger scale problems. Supposedly, such behaviour is caused by the distorted effects of noisy estimates, because the further in the future we myopically look, the less accurate are the values produced by the MR.
Accounting for the attentive considerations, we set $R = 15$ for all problem's instances, but what if the rollout reaches a terminal state while the number of recursive steps performed so far is below R? Of course, once the rollout approaches the time horizon or reaches a terminal state it cannot proceed, and a precise value must be assigned to the terminal state reached. To this aim, we introduce another terminal value function:

$$F(s) = 0.5(T - s_1) - \sum_o \Big(\sum_m (q_m | o_m = o) - \sum_k \underline{s}^o_{k+1} \Big) + \sum_o \underline{s}^o_1, \tag{3.14}$$

whose alternative definition with respect to the terminal functions so far presented revealed necessary. Indeed, for LAPs, it is common to simply assign a null value to all terminal states of the system, at the expenses of evaluating their appeal. Partially respecting this practice, Equation (4.2) defines a "soft" value function, that assigns moderate values to the states while still capturing useful information for distinguishing them.
Finally, we conclude the subsection by summarizing the overall decision-making procedure through MR in Pseudo-Algorithm 2.

3.2.2 Monte Carlo Tree search

MCTS is a search method based on a randomized exploration of the state space. Its algorithm uses the results of previous explorations to gradually build up a tree in memory, hence it progressively becomes better at accurately estimating the values of the most promising actions (Winands [2015]).
It performs admirably for problems affected by risk factors with a multitude of random outcomes, as long as they present a conservative number of actions per state. As noticed at the beginning of the chapter, our dynamic scheduling problem has indeed a limited action space's cardinality, therefore we supposed that a MCTS would reveal a good fit for its resolution.
As the name implies, the search is performed by means of a gradually constructed decision tree, but before introducing the overall procedure of building it, let us define its nodes. There are two types of nodes in a decision tree: the decision nodes, at which decisions are made, and the outcome nodes, at which new random information becomes available. In a DP context like the one we are dealing with, the decision nodes identify the standard states of the system, while the outcome nodes represent the post-decision states. A

post-decision state is the state that the system intends to reach when a specific action is performed, as if there were no exogenous risk factors. For example, in our problem, the choice of moving action $a \in \mathcal{A}_{move}$ when in state $s_t|s_t^1 = t, s_t^2 = n$, is made with the intention of reaching state $s_{t+\Delta t_e}|s_{t+\Delta t_e}^1 = t + \Delta t_e, s_{t+\Delta t_e}^2 = a, e = (n, a) \in \mathcal{E}$, which is indeed a post-decision state, more precisely denoted by s_t^a to emphasize the dependence on chosen action a. After reaching a post-decision state, information on the realization of external factors becomes available and determines the actual transition to another standard state, that we refer to as pre-decision state. In the previous example, the next pre-decision state may be equal to the post-decision one, if no collision takes place during navigation, or it can result in $s_{t+\Delta t_e+5} \neq s_t^a$ when a failure occurs.
From now on, in our MTCS, the transition from a pre-decision state s_t (decision node) to the next one $s_{t+\Delta t}$ is thus divided into two steps: first, an action a_t is chosen and the algorithm transitions to a post-decision state s_t^a (outcome node) following the deterministic transition equations defined in (2.5)-(2.10) (with s_t^a replacing $s_{t+\Delta t}$); then, after the random realizations of the risk factors, it proceeds according to the transitions defined for (2.15) and (2.20), for which we use the novel notation:

$$s_{t+\Delta t} = g_{t+\Delta t}^a(s_t, a_t, w_{t+\Delta t}). \tag{3.15}$$

It is worth noting that the concept of post-decision state fits like a glove for the modelling of board games. This, coupled with their typically moderate number of actions, explains the frequent employment of MCTS in software developed to play (and win) them. In fact, in board games, a post-decision state identifies the effects of the action of a player before knowing the opponent's move, which represents the only exogenous risk factor.
Unluckily, our real-life intralogistic robot scheduling problem is not a game, however, MCTS still seems a reasonable choice for solving it. Regardless the application for which it is employed, the algorithm always follows four main steps iteratively. In fact, after having identified as the tree root the state at which the robot needs to choose the best action to perform, the MCTS begins and repeatedly undergoes the phases of selection, expansion, simulation and backpropagation.
In the following, a precise and problem-driven description for arbitrary iteration j.

1. **Selection)** It is the first phase of the MCTS and aims at selecting the most suitable action to perform at a pre-decision state s_t^j, in order to keep exploring the tree. If the number of children for the decision node identified by s_t^j is less than a fixed allowed offspring limit, the action is chosen among the available ones. The decision is based on a one-step simulation followed by a MR, as in:

$$a_t^{j,\star} = \arg\max_{a \in \mathcal{A}_{s_t}^j} C_t(s_t^j, a) + \textsc{MyopicRollout}(g_{t+\Delta t}(s_t^j, a, w_{t+\Delta t}^j)). \tag{3.16}$$

   On the other hand, if the offspring limit has already been reached in earlier iterations, the action is chosen among the previously visited ones, collected in $\bar{\mathcal{A}}_{s^j}$, according to the Upper Confidence bounding for Trees (UCT) (Kocsis and Szepesvári [2006]):

$$a_t^{j,*} = \arg\max_{a \in \bar{\mathcal{A}}_s^j} \hat{Q}(s_t^j, a) + \epsilon \sqrt{\frac{2 \ln N(s_t^j)}{N(s_t^{a,j})}}. \tag{3.17}$$

The exploration coefficient ϵ and the number of visits $N(s_t^j)$ and $N(s_t^{a,j})$ of the decision node identified by s_t^j and of the outcome node identified by $s_t^{a,j}$ respectively, define an exploration term voluntarily biased towards post-decision states that have been visited less frequently. On the contrary, the term $\hat{Q}(s_t^j, a) = C_t(s_t^j, a) + \hat{V}^a(s_t^{a,j})$, where $\hat{V}^a(s_t^{a,j})$ indicates the approximate value assigned to post-decision state s_t^j up until iteration j, steers the choice towards actions so far considered more promising.

2. **Expansion)** Right after the action selection, it comes the expansion phase, whose procedure differs depending on earlier explorations. In fact:

   - if the selected action has never been tried before, the outcome node corresponding to the post-decision state is created. Then, an outcome is uniformly sampled among the available ones and the corresponding pre-decision state is created. At this point, the search enters its next phase;
   - if the selected action has already been tried, there are two further distinct situations:
     - all outcomes have been visited. In this case, an outcome simulation is performed, that will bring the search to a next pre-decision state, from which a new selection phase will begin;
     - not all outcomes have been visited. Therefore, an outcome is uniformly sampled among the ones not yet explored, and the corresponding next pre-decision state is created, leading the search to its next phase.

   Note that, while a limit is set for the number of actions to try at each state, i.e., for the offspring of the corresponding decision node, all outcomes are potentially explored. The choice is due to the fact that, in our specific problem, the admissible outcome space given a state is at most binary. For this same reason, there is no negative effect in uniformly sampling the outcomes during the expansion phase, actually we think it may fasten the initial exploration.
   Nevertheless, after having sampled all the possible realizations given an action, external risk factors are simulated according to their real probability distributions.

3. **Simulation)** Whenever a new pre-decision state is created during expansion, this last phase stops and a simulation begins: a value estimated through a MR is associated to the state representing the new leaf node.

4. **Backpropagation)** During this last phase of the MCTS, the newly simulated value associated to the newly created leaf node is back-propagated towards the parent-node, iteratively until the root, following the path sampled during the previous phases of the current iteration, j. In the meanwhile, also the counters of the number of visits for each node in the path are updated. The overall procedure is illustrated in Pseudo-Algorithm 3, where $\bar{\mathcal{W}}_a^j$ denotes the set of outcomes visited after the play of action a up until iteration j, and is similar to the approach proposed in Powell [2019], but specifically adapted for single temporal step updates.

Algorithm 3 Backpropagation phase of MCTS

procedure BACKPROPAGATION(s_t^j)
  $N(s_t^j) \leftarrow N(s_t^j) + 1$
  while $s_{t-\Delta t}^{a,j}$ is **not** null **do**
    $N(s_{t-\Delta t}^{a,j}) \leftarrow N(s_{t-\Delta t}^{a,j}) + 1$
    $\hat{V}^a(s_{t-\Delta t}^{a,j}) = \frac{1}{\sum_{w \in \bar{\mathcal{W}}_a^j} \mathbb{P}(w)} \sum_{w \in \bar{\mathcal{W}}_a^j} \mathbb{P}(w) \hat{V}(g_t^a(s_{t-\Delta t}^j, a_{t-\Delta t}^j, w))$
    reward $= C_{t-\Delta t}(s_{t-\Delta t}^j, a_{t-\Delta t}^j, w_t^j)$
    delta $= \hat{V}^{a,j}(s_{t-\Delta t}^{a,j})$
    $\hat{V}(s_{t-\Delta t}^j) \leftarrow \hat{V}(s_{t-\Delta t}^j) + \frac{(\text{delta} - \hat{V}(s_{t-\Delta t}^j))}{N(s_{t-\Delta t}^j)+1}$
    BACKPROPAGATION($s_{t-\Delta t}^j$)
  end while
end procedure

Once the four phases of the search are repeated for a fixed time of iterations, ($J = 100$ in our experiments), the policy, guiding the choice of the best action to perform when in state s_{t_0}, corresponding to the root node of the just created Monte Carlo Tree, is:

$$\pi^\star(s_{t_0}) = \arg\max_{a \in \mathcal{A}_{s_{t_0}}} \hat{Q}(s_{t_0}, a) = \arg\max_{a \in \mathcal{A}_{s_{t_0}}} C_{t_0}(s_{t_0}, a) + \hat{V}^a(s_{t_0}^a). \tag{3.18}$$

Let us conclude by focusing in detail on a couple of essential hyperparameters that have been mentioned when explaining the MCTS phases: the exploration parameter ϵ and the maximum number of children per decision node.
Both parameters define the exploration degree of the search but to different extents. Exploration factor ϵ affects the selection phase of the algorithm by means of the choice of the action to perform, attributing weight to actions that have been previously, but less frequently, selected. The offspring limit per decision node represents the maximum number of actions to explore given a state, hence it influences the selection phase, too, and also affects the tree width. The construction of a wider tree may significantly compromise its depth, therefore it is necessary to elevate the number of iterations J when a less conservative offspring limit is chosen, if one wants to maintain a fixed lookahead in the future.
However, the number of iterations is not the only parameter to adjust when modifying the offspring limit per decision node. In fact, the choices of the two presented parameters defining the exploration degree of the search strictly depend on each other. There are, indeed, two situations that are preferably to be avoided during a MCTS, and that can be partially dodged with a reasonable tuning of the exploration factor ϵ and the offspring limit (from now on denoted as L). The first inconvenient situation consists in iteratively lowering the approximate value of a state, up until its exclusion from further exploration,

although it may lead to a very promising future state when the correct action is selected. In fact, during backpropagation, the value of a state can be repeatedly compromised by the values of its other, less promising offspring, if numerous. On the contrary, the algorithm might become interested in visiting states that only appear as favorable, when it neglects less frequently visited, yet better, actions.
The first scenario is likely to happen when ϵ and L are set to elevate values, whilst the second is mainly caused by the lowering of the former. Thus, it is naturally inferred that the two parameters shall be antithetically fixed: an excessive exploration factor should be accompanied by a conservative offspring limit, and viceversa. We opted for the first alternative and set $\epsilon = 3.5$ and $L = 5$ in all our experiments, conscious of the necessity of increasing both of them, and consequently the number of iterations J, when dealing with significantly larger scale problems. Note that our choice for the value of the two parameters might appear excessive for small instances, nevertheless a thorough exploration degree is nothing but an advantage when paired with an adequate number of iterations, that must not be unreasonable for small problems.
The above analysis concludes the detailed description of the MCTS algorithm and of its application to our scheduling problem. Its performances are presented in the following sections, alongside the results of the other approximate approaches covered so far.

3.3 ADP vs DP: comparisons on toy models

When it comes to the validation of ADP methods, comparing their performances with the results output by the exact DP counterpart applied to the same problems might seem the most natural choice. Nevertheless, it represents a complicated matter because the curses of dimensionality we desperately attempt to avoid by applying ADP methods deeply affect the computational expense of the exact DP paradigm, as we have already highlighted a couple of times earlier in the dissertation. Therefore, the direct comparison between the performances of approximate and exact DP methods is only computationally possible on small scale instances. Indeed, we will commence the experimental analysis for the validation of the approximate methods presented earlier in this chapter, by comparing their results to the exact ones, exclusively for Instances 1., 2., 3., 4., presented in Chapter 2, Section 2.2, with the only adjustment of setting $r_{\text{pick}} = 20, r_{\text{throw}} = 25$ when the problems are solved with the API. We leave further considerations on their performances on larger scale instances to the next section.
We gathered information about the mean execution times of our approximate algorithms and about the accuracy of their output with respect to the benchmark results given by the DP approach (Chapter 2, Section 2.2.2) in Tables 3.1-3.4. For all methods, the performance assessment is based on the evaluation of the terminal state of the system output when using the specific method for scheduling tasks throughout a simulation. Such evaluation is computed by means of an evaluating function that we set as the terminal value function of Equation (3.11), with $r_{\text{pick}} = 20, r_{\text{throw}} = 25$, whose choice considerably affects the percentage accuracies with respect to the benchmark results. In fact, different evaluating functions reflect different characteristics about the terminal states and assign different weights to specific details. With our choice, for example, we decided to let time

play a massive role, and a 5 seconds delay in completing the mission results in a penalty of 25 points for the corresponding terminal state evaluation. Moreover, also coefficients r_{pick} and r_{throw}, that respectively assign a reward for having picked an object and a penalty for not having it placed, are set to significant values, but they counterbalance each other's weight in most situations. Therefore, one should choose an evaluating function based on their needs and on the focuses of their analysis, ergo, the choice is left to the implementer. Since we only require the evaluation to validate the overall functioning of the approaches, the evaluating method we defined by recycling the terminal value function used in API is enough for the purpose.

Resolution Method	**Mean Execution Time**[6]	**Terminal State Evaluation**[7]	**Percentage Accuracy**
DP	$3.18s$	226	-
API	$68s$	207.5	91.8%
MR	$1.03s$	208.3	92.2%
MCTS	$5s$	221.3	97.9%

Table 3.1: Comparison of the performances of our approximate methods with the results given by the application of exact DP, for a Mini instance(1) with state space cardinality equal to 159 360.

Resolution Method	**Mean Execution Time**[6]	**Terminal State Evaluation**[7]	**Percentage Accuracy**
DP	$204.3s$	453.5	-
API	$143s$	348.9	76.9%
MR	$1.06s$	305.7	67.4%
MCTS	$15s$	323.3	71.3%

Table 3.2: Comparison of the performances of our approximate methods with the results given by the application of exact DP, for a Medium-small instance(2) with state space cardinality equal to 3 136 000.

For what concerns execution times, the results are presented in the second column of each table, for all methods. It is immediate to notice that, except for the Mini instance of Table 3.1, for whose resolution one might definitely want to use the exact method if available, the execution times for the approximate approaches are faster than the ones observed for DP. Moreover, the divergence becomes tremendously substantial with the increase in the problem's size.
Note that the worst performing algorithm in terms of execution speed is the API, while the fastest is the MR; however, the ranking is inverted when observing their percentage accuracy. In fact, the overall results, compared to the DP benchmark, are very promising for all methods on all instances, but the performances of our API stand out.

Resolution Method	**Mean Execution Time**[6]	**Terminal State Evaluation**[7]	**Percentage Accuracy**
DP	$255.2s$	503.2	-
API	$110s$	421.4	83.7%
MR	$1.03s$	346.9	68.9%
MCTS	$11s$	410.8	81.6%

Table 3.3: Comparison of the performances of our approximate methods with the results given by the application of exact DP, for a Medium instance(3) with state space cardinality equal to 7 937 300.

Resolution Method	**Mean Execution Time**[6]	**Terminal State Evaluation**[8]	**Percentage Accuracy**
DP	$4h22m$	625	-
API	$261s$	574.5	91.9%
MR	$1.11s$	392.1	62.7%
MCTS	$45s$	389.6	62.3%

Table 3.4: Comparison of the performances of our approximate methods with the results given by the application of exact DP, for a Large instance(4) with state space cardinality equal to 176 150 400.

3.3.1 Accuracy of value function approximations through API

The protagonism of the performances of the API algorithm is worth investigating, although its execution times are far from desirable. Indeed, staring at the tables presented in the previous section, it comes natural to wonder to what are due its superior results compared to the other methods'.
With the aim of providing a satisfactory explanation to the matter, we performed an analysis on the value function approximations output by our API. Figure 3.5 depicts a graph showing the values assigned by the VFA produced with the API to 100 randomly sampled states of the API Test instance(5), compared with the corresponding exact values output by the DP paradigm. The accuracy of the approximations is astonishing and provides a strong motivation for the algorithm's outstanding results.

[6]Mean execution over at least 10 runs per method. Python Implementation on PyCharm Professional, Version 2022.3.2, on a machine equipped with RAM 16GB, intelCore i7, intel iRISx[e].

[7]Mean evaluation over 50 runs for DP, API and MR, and over 30 for MCTS

[8]Mean evaluation over 50 runs for API and MR, and over 30 for MCTS. For DP the value is the output of a single run.

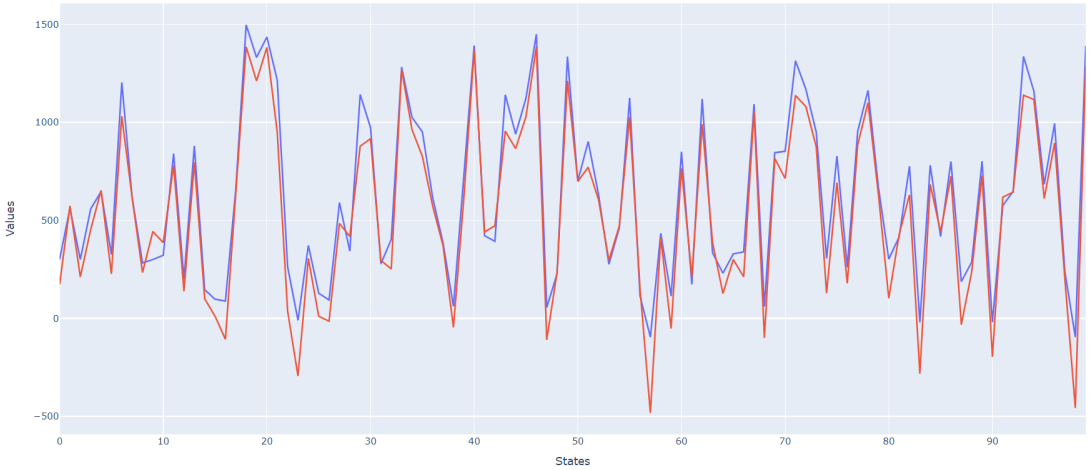


Figure 3.5: In red, the values produced by the API VFA and, in blue, the exact DP values for 100 randomly sampled states of the API Test instance(5).

3.4 ADP performances on larger scale problems

We premised that it becomes intractable to carry out experiments using the exact DP paradigm when it comes to larger scale problems. Nevertheless, in the previous section we have already established the validity of our approximate methods through comparisons with the exact results on various restrained instances. Thus, we are free to continue to analyse their performances without the juxtaposition to DP. In fact, we will now discuss additional results output by our approximate approaches when applied to larger scale problems, compare them solely to each other and raise further considerations.
First of all, let us introduce the two large scale instances on which the further experiments are conducted:

6. Large instance I:
    - time horizon $T = 360$,
    - $O = 3$ object types,
    - $K = 3$ trays,
    - $|\mathcal{N}| = 6$ graph vertices,
    - mission $\mathcal{M} = \{(1,1,3), (1, 2, 2), (1,3,2), (2, 2, 2), (2,3,4), (3,1,1), (3,2,4)\}$,
    - $Q = \sum_{m \in \mathcal{M}} q_m = 18$ total number of objects,
    - maximum robot capacity $c = 4$,
    - rewards $r_{\text{pick}} = 10$, $r_{\text{throw}} = 12$ for MR and MCTS, $r_{\text{pick}} = 20$, $r_{\text{throw}} = 25$ for API, and $r_{\text{move}} = 0$ for all,
    - state space cardinality $|\mathcal{S}| = 279\,899\,040$;

7. Large instance II:
    - time horizon $T = 360$,
    - $O = 5$ object types,
    - $K = 3$ trays,
    - $|\mathcal{N}| = 8$ graph vertices,
    - mission $\mathcal{M} = \{(1,1,2), (1,2,2), (1,4,1), (1,5,2), (2,2,2), (2,3,3), (2,4,2), (3,1,4), (3,2,1), (3,3,2), (3,5,1)\}$,
    - $Q = \sum_{m \in \mathcal{M}} q_m = 16$ total number of objects,
    - maximum robot capacity $c = 4$,
    - rewards $r_{\text{pick}} = 10$, $r_{\text{throw}} = 12$ for MR and MCTS, $r_{\text{pick}} = 20$, $r_{\text{throw}} = 25$ for API, and $r_{\text{move}} = 0$ for all,
    - state space cardinality $|\mathcal{S}| = 1\,516\,308\,000$.

The two chosen problems differ in the number of object types, hence of the graph vertices, and in the total number of items to collect during the mission. Surprisingly, we notice that this last input does not significantly influence the increase in the state space cardinality, which, instead, seems to be caused by a combination of the total number of object types and the nature of the mission (of its being *compact* or *sparse*). Indeed, the number of object types does affect the dimension of a state and the mission defines the values that each one of its entries can assume; therefore, they both contribute to variations of the state space cardinality.
The comparison of the second column of Table 3.5 with the same column of Table 3.6 proves that all methods present slower execution times when applied to Large instance II. We deduce that the state space dimension affects the computational expense of approximate methods, as it does with exact DP. Nevertheless, in this case, the effects are not intractable and represent a natural consequence of the enlargement in the problem size.
Concluded the brief consideration on the connection between state space cardinality and computational expense, let us return to our main focus: identifying the best method in terms of quality of results and resolution speed.
In the previous section, we learned how our API algorithm outputs exceptional results on medium scale instances, although its execution times are not as appreciable as the ones of other methods, e.g., MR. In this section, we partially confirm the same thesis, especially when observing the results projected in Table 3.6, that prove the superior performance of API on Large instance II$^{(7)}$ with respect to the other approximate methods. However, its mean terminal state evaluation for Large instance I$^{(6)}$ is less optimal than the one of the MCTS algorithm, suggesting a vulnerability to problems with more compact missions. Hence, MCTS eventually outshines API while utilizing considerably fewer temporal resources, and thus represents an equivalently promising resolution approach.
For what concerns MR, its employment as dynamic decision-making method is worth to be considered for its impressive speed rather than for its results, fine but generally outperformed by both API and MCTS.

Bear in mind that different users may express different personal preferences on the method to employ for the resolution of their intralogistic robot scheduling problem based on their necessities. For example, if they absolutely require an immediate resolution and are willing to settle for lower-quality results, they would certainly opt for the MR approach. On the contrary, if time resources do not represent an issue, one might embrace the API algorithm. An evergreen valid compromise would be to apply a MCTS, whose execution time is moderately fast with respect to API, and whose quality of results exceeds the one of MR.

Resolution Method	**Execution Time**	**Terminal State Evaluation**[9]
API	$277.1s$	447.7
MR	$1.11s$	419.9
MCTS	$31.7s$	505.0

Table 3.5: Overall comparisons between the performances of our approximate approaches on Large instance I(6).

Resolution Method	**Execution Time**	**Terminal State Evaluation**[9]
API	$433.6s$	548.5
MR	$1.27s$	493.3
MCTS	$58.25s$	499.6

Table 3.6: Overall comparisons between the performances of our approximate approaches on Large instance II(7).

Finally, it appears necessary to specify that the expression "API resolution time", as used thus far, refers to the execution speed of its learning process, defined in Pseudo-Algorithm 1. The learning procedure must indeed be repeated whenever a mission is changed. Nevertheless, if a warehouse is used to always satisfy the same orders, its repetition would result unnecessary, and a sequence of live schedules could be performed by using the VFA output by a single learning process. On the other hand, it is not possible to immediately identify the optimal task to perform in a state when using a MCTS. In fact, this approach involves the time-consuming steps of construction and search of a novel tree for every single decision.
These last observations strengthen the argument that the suitability of an approximate method substantially depends on the applications and the objective of the problem.

[9]Mean evaluation over 30 runs for API and MR, and over 15 runs for MCTS. Python Implementation on PyCharm Professional, Version 2022.3.2, on a machine equipped with RAM 16GB, intelCore i7, intel iRISx[e].

Chapter 4

Introducing priorities

The problem definition on which we have focused so far quite reflects a real-world situation despite the various assumptions, e.g., the static nature of risk factors and the knowledge of their probability distributions. Whilst maintaining such simplifying assumptions, we will now modify the problem definition by adding one real-world complication: priorities. By definition, the term *priority* refers to the right of being considered as more important than others, and different levels of priority define indeed a ranking of importance. We are interested in investigating how this concept aligns with our problem and how it leads to its redefinition by introducing rules of precedence for the objects to pick and place.

4.1 Problem reintrepretation

Consider the warehouse introduced in Chapter 1 as a distribution center responsible for assembling orders from clients, defined as demands of the kind $d = \{(o, q_o)|o = 1, ...O\}$, where q_o is the required number of items of object type $o \in \mathcal{O}$. Thus, the robot's task remains the transportation of objects to designated trays, which now symbolize assembly areas. An assembly area is the location where the various items of an order must be gathered in specified amounts.
Such reinterpretation of the problem prompts a redefinition of the mission as a primary step: instead of thinking it as a series of order lines let it be organized based on the orders assigned to the trays, as $\mathcal{M} = \cup_{k \in \mathcal{K}} \mathcal{M}_k$, where $\mathcal{M}_k = \{(o_k, q_{o_k})|o = 1, ...O\} \equiv \{(k_m, o_m, q_m)|k_m = k\}$ in the old definition.
Note that at most a number of orders equivalent to the number of available trays can define a mission and, hence, be served simultaneously. However, in real-world applications there are usually more orders than available assembly spots and, in fact, a queue of requests is created. In our setting we define the queue as a previously defined static list of $D > K$ orders sorted by their delivery times $T_d, d = 1, ..., D$. Delivery times may be regarded as deadlines for the completion of the orders' assembly and considered as an additional request from demanding clients. Bear in mind that the definition of a static queue, as for the risk factors, depends on the the fact that time horizons are generally set to be short in our experiments. An easily scalable alternative for a more realistic, dynamic queue would

be to define a stochastic process for new orders' arrival; then, an order should be inserted in the correct position of the queue based on its delivery time, as soon as it arrives.
Respecting our assumption on the static nature of the queue, we create an initial mission by placing the first K orders in the K available trays. Whenever an order is assembled and a tray is thus emptied, the next order in the queue is served and the mission is updated.
In the just described setting, we let the priority of an order in the service queue being uniquely determined by its delivery time: the closer it is, the earlier the order is served. We will later discuss how delivery times should also affect the priority definition for the completion of an order while it is being served concurrently with others.

4.2 Resolution approaches

We now present three different approaches to solve the redefined intralogistic robot scheduling problem with priorities. We start by introducing a naive sequential heuristic and later improve it with the addition of the DP paradigm. Finally, we will move beyond the basic sequential rule and propose a more general DP approach that addresses tray-filling priorities other than solely serving precedence.
The implementation of a heuristic is mainly needed to characterize an extremely fast, yet quite accurate, scheduling procedure to be employed for the resolution of larger scale problems and, in general, as a benchmark for analyzing the execution times of the other more complex methods. In fact, as we pointed out in the previous chapter while evaluating the performances of the ADP methods, in real-world applications it is sometimes necessary to establish a trade-off between execution time and optimality of the output, especially when it comes to larger scale problems.
Further considerations and overall results will be discussed in Section 4.3.

4.2.1 Sequential Heuristic

A sequential serving and tray-filling procedure requires just one assembly spot. In fact, once the order with highest priority is served, the robot's gathering of objects focuses solely one its items until completion. Only after having completed the request associated to the first order, a tray substitution takes place, and the procedure is repeated. As previously mentioned, the serving rule is based on priorities uniquely determined by the orders' delivery times.
So far, we presented an intuitive and simple way of modelling the precedence of orders in a distribution center when a sequential rule is applied. In fact, a similar reasoning would (and will, in the next subsection) be employed for analogous situations. Undoubtedly, a more curious aspect of this approach is the heuristic procedure applied to the filling of the one tray available, which is tailored to our specific problem setting.
The procedure is based on the exploitation of the proximity of the boxes. Each time a robot is assigned an order, it is initially guided towards a vertex corresponding to a box in a marginal position. From there, it starts picking items of the object type associated to the box if any are demanded, otherwise it moves to the next, adjacent box. Let us suppose some items are actually requested: the robot continues to pick the needed items

until it reaches the required number or its capacity. In the former case, it quit picking and moves towards the next, adjacent box to repeat the procedure for a different object type. In the latter case, it must move to the assembly area to unload all the items collected; the same action is also performed when the robot has visited all the vertices associated to the boxes. After the entire load has been placed in or thrown to the tray, if there are missing items to meet the order's demand, the robot returns to the box from where the picking was interrupted. On the other hand, when the mission is completed, the robot recedes to the initial marginal box's location in order to repeat the procedure for the immediately succeeding order. The process is repeated for all orders in the queue or until a fixed time horizon is reached, and is summarized in the Flowchart of Figure 4.1.
Naturally, a scheduling output by the just described heuristic may hardly ever result optimal, mostly due to the constraints of visiting all boxes and of always restarting from a marginally located one, despite there might be no objects to collect from there. Nevertheless, our heuristic represents a clever procedure for not missing requests and it does lead to a time-efficient scheduling whenever the picking spots are approximately lined up, which is the case for our setting. In opposite contexts, where for instance boxes are sparse around the warehouse, the robot would be far from following a sub-optimal path, causing a poor scheduling of tasks. For this reason, we formerly referred to our heuristic procedure as specifically tailored for our problem.

4.2.2 Sequential DP

Our second approach is based on a sequential serving rule identical to the one presented for the heuristic. Once more, we consider a unique assembly spot and serve the orders consecutively, removing them one at a time from a queue where they are sorted by their delivery times. The fundamental difference from the previous approach regards the tray-filling rule: the robot schedules its tasks relying on the application of the exact DP paradigm. Indeed, we consider this approach far more intriguing than the heuristic, and expect superior performances.
Let us summarize the procedure and point out some considerations. The order with the highest priority is served and a one-tray mission is defined. The robot's scheduling for the order's completion is performed as we have learned, applying the DP paradigm defined in Chapter 2. Each time a demand is fulfilled and a new order is assigned to the again available tray, an additional backward pass must be performed. Although it represents the most time-consuming step of the paradigm, we are less concerned about the curses of dimensionality than we were for the standard problem definition, because the sequential serving significantly reduces the mission's size, the state's dimension and the state space cardinality consequently. Moreover, as more substitutions are carried out, the backward pass accelerates since it only requires backtracking in time until the moment of the last substitution; for this reason, it reveals necessary to retain a record of the orders' entering times. The procedure concludes when all orders have been fulfilled or the predetermined time horizon is reached.

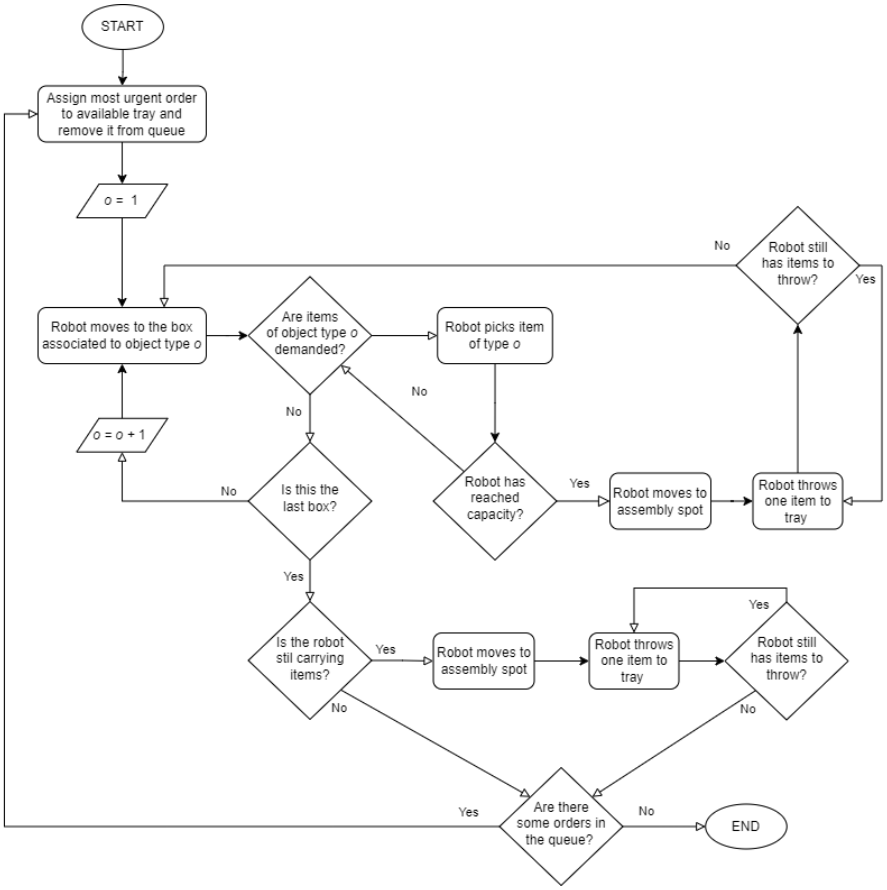


Figure 4.1: Sequential Heuristic's algorithmic flowchart. The condition on not having reached the time horizon is implicit: it is not shown in the flowchart but it is intrinsically verified at every step.

4.2.3 Prioritizing DP

For the presentation of this last method we return to a setting where we have more than one tray and some orders are thus assembled concurrently. Therefore, the definition of a priority rule for the assembling of the orders becomes necessary.
The robot's scheduling output by this last approach is derived from an adjusted version of the resolution method based on the exact DP paradigm defined in Chapter 2. In fact, we characterize a tray-filling priority rule by modifying the immediate contributions of throwing actions, while leaving the ones of picking and moving actions as defined in (2.11) and (2.22) respectively. For a throwing action of the type $a_t = (o, k)$ we have:

$$C_t(s_t, a_t, w_{t+\Delta t_{pt}}) = \begin{cases} r_{\text{throw}}\left(1 - \dfrac{s_t^1}{T_d}\right) - \dfrac{(s_t^1 - T_k^E)}{1000} - \dfrac{100}{T_d} \cdot \mathbb{1}_{(s_t^1 > T_d)}, & \text{if } w_{t+\Delta t_{pt}} = 1 \\ 0, \text{ if } w_{t+\Delta t_{pt}} = 0, \end{cases} \tag{4.1}$$

where T_d and T_k^E are the delivery time and the entering time of the order d associated to tray k. The multiplicative term affecting the deterministic reward r_{throw} causes a monotone decrease, and even switches to a negative value as soon as the order's delivery time is surpassed. When this happens, the immediate contribution is further penalized by a constant term, which is greater for orders with earlier delivery times. Instead, the impact of the penalty associated to the order's waiting time $s_t^1 - T_k^E$ is relatively minor with respect to the overall context.
As for the case without priorities, the forcing of a decrease with time in the value of immediate contributions for throwing actions has once again proved necessary to emphasize the importance of placing objects at the earliest convenient opportunity. Moreover, in this novel scenario with priorities, the extent to which immediate contributions depend on time is determined by, and varies with, delivery times. In fact, the introduction of this dependence contributes to giving precedence to the fulfillment of orders with a closest deadline.
It is worth mentioning that the decision of solely modifying the immediate contribution of throwing actions was dictated by the fact that the picking of an object is always executed before the respective throw. Since in a DP paradigm the values associated to the states are assigned through a backward pass, the priority of a throwing action $a_{\text{throw}} = (o, k)$ is inevitably backtracked to the respective, previously executed, picking action $a_{\text{pick}} = o$.
For what concerns the serving rule, it is simply based on the delivery times of the orders in the queue, as it is for the other two presented approaches. Accordingly, an initial mission is defined by assigning to all K available trays the K most urgent orders in the queue. Then, after a first backward pass, the definitive scheduling begins. When the demand of an order d is fulfilled, and a tray becomes available again, the placement of a new order leads to a redefinition of the mission and another backward pass is required. In this case, although the new pass only necessitate backtracking until the time at which the latest substitution takes place, we expect its execution time to be longer than the one of Sequential DP. In fact, a greater quantity of available trays entails more extensive missions, higher-dimensional states and, consequently larger state spaces.

4.3 Results

The moment has come to showcase the performances of the three novel approaches for the redefined intralogistic robot scheduling problem with priorities. At first, we will focus on the analysis of the results of experiments on medium scale instances, with the aim of comparing the multiple strategies. Later on, we will discuss which techniques could be applied to solve larger scale problems and how.
Let us commence by introducing the problem's instances on which experiments are conducted. There are three scenarios, all having in common the following input data:

- time horizon $T = 230$,
- $O = 5$ object types,
- $K = 1$ tray for the sequential approaches, $K = 2$ tray for Prioritizing DP,

- $|\mathcal{N}| = 6$ graph vertices for the sequential approaches, $|\mathcal{N}| = 7$ graph vertices for Prioritizing DP,
- $D = 3$ orders,
- maximum robot capacity $c = 4$,

Moreover, for all experiments regarding the approaches employing the DP paradigm, the terminal value function has been set to:

$$F(s) = 0.5(T - s_1) - 15 \sum_{o} \big(\sum_{m} (q_m | o_m = o) - \sum_{k} \underline{s}^{o}_{k+1} \big), \tag{4.2}$$

while all other specifics about rewards, state transitions, etc., are left as presented in Chapter 2, except for the immediate contributions of the Prioritizing DP.
Each instance is mainly characterized by the definition of the orders in its queue and the corresponding delivery times. In fact, queues serve as primary distinguishing factors among the scenarios listed below:

8a. Priority instance I:

Queue	Obj 1	Obj 2	Obj 3	Obj 4	Obj 5	Delivery time
Order 1	1	1	0	2	0	$90s$
Order 2	0	1	2	0	0	$120s$
Order 3	1	1	1	0	2	$210s$

with total number of items to collect $Q = \sum_{d \in \mathcal{D}} \sum_{o \in \mathcal{O}} q^d_o = 12$;

8b. Priority instance II:

Queue	Obj 1	Obj 2	Obj 3	Obj 4	Obj 5	Delivery time
Order 1	1	1	0	0	1	$70s$
Order 2	1	2	0	0	1	$100s$
Order 3	0	2	1	3	0	$150s$

with total number of items to collect $Q = \sum_{d \in \mathcal{D}} \sum_{o \in \mathcal{O}} q^d_o = 13$;

8c. Priority instance III:

Queue	Obj 1	Obj 2	Obj 3	Obj 4	Obj 5	Delivery time
Order 1	0	1	2	0	1	$120s$
Order 2	1	1	1	0	2	$130s$
Order 3	1	1	0	3	0	$190s$

with total number of items to collect $Q = \sum_{d \in \mathcal{D}} \sum_{o \in \mathcal{O}} q^d_o = 14$.

Before showing and discussing the results of the application of our approaches to the aforementioned problem's instances, it is important to address the challenge of making

comparisons. In fact, for each scenario, we would like to compare the orders' actual completion times, output when scheduling with a specific approach, with their delivery times and with the completion times output by the other approaches. However, in a stochastic context it is almost impossible to get the same number of incidents (collisions and failed throws), whose occurrences inevitably cause delays, when repeating experiments. For this reason, to get more homogeneous and comparable results, at the beginning of each experiment the seeds for two Pseudo Random Number Generator (PRNG) streams, used for the simulation of collisions and throwing successes, are fixed.
Despite the decision of fixing the same two seeds when different approaches are applied to the same instance, we still cannot observe the exact same outcomes. We find the explanation lying in the different scheduling of tasks that each approach outputs for the robot. In fact, a variation in the timing of performing a specific risky action can result in a shift within the associated stream's sequence and cause a different outcome; on the other hand, also the execution of distinct actions at the same step of a scheduling, and indeed associated to the same output of a stream, can affect a simulation. For example, let us consider a scheduling rule that, as first action, requires the robot to move to a position adjacent to the initial one. The risk associated to such moving action is indeed very low; in fact, let us suppose it being $r_e = 10$, i.e., with probability $p_{\text{move}} = 0.1$ the robot collides with a human or a shelving unit. Then, let the first output of the PRNG stream for moving actions be equal to 0.15: no collision happens during the simulation because $0.15 > 0.1$. Now, if we used a different scheduling rule that would guide the robot to move to a further location, with an associated navigation risk of 0.25, the robot would collide with a human or a shelve and cause a delay. Therefore, during the discussion of the results we will take into account the occurrences of incidents.

Resolution Method	**Instance**	**Execution Time**[10]
Sequential Heuristic	Priority instance $I^{8(a)}$	$1.007s$
	Priority instance $II^{8(b)}$	$1.008s$
	Priority instance $III^{8(c)}$	$1.005s$
Sequential DP	Priority instance $I^{8(a)}$	$4.28s$
	Priority instance $II^{8(b)}$	$4.51s$
	Priority instance $III^{8(c)}$	$5.41s$
Prioritizing DP	Priority instance $I^{8(a)}$	$106s$
	Priority instance $II^{8(b)}$	$244s$
	Priority instance $III^{8(c)}$	$394s$

Table 4.1: Execution times of all three approaches for every instance of the redefined scheduling problem with priorities.

[10]Python Implementation on PyCharm Professional, Version 2022.3.2, on a machine equipped with RAM 16GB, intelCore i7, intel iRISx[e].

Having highlighted the difficulty of making comparisons and suggested a solution to the challenge, we can now finally focus on the analysis of the results. First of all, recall that while introducing the resolution approaches we anticipated that we would expect longer execution times for the Prioritizing DP approach, due to its recurrent application of a backward pass on a larger state space: Table 4.1 confirms the intuition.
On the other hand, as shown in Tables 4.2, 4.3 and 4.4, where the entries in bold represent the finest results for the columns to which they belong, Prioritizing DP is undoubtedly the best performing approach in terms of time taken to fulfill all the orders, despite a few difficulties in respecting the deadlines set by delivery times. However, the main purpose of delivery times within our experiments is to define priorities, and they might be myopically set for guaranteeing the fulfillment of demands in time, especially when they are very close to the beginning of the mission and/or to each other, as in the second instance for example. In real-world applications, similar situations may happen and what becomes of primary importance is the completion ranking of the orders: in all our cases it is in line with the delivery times.

Resolution Method	**Completion of Order 1 (90s)**	**Completion of Order 2 (120s)**	**Completion of Order 3 (210s)**
Sequential Heuristic	**57*s***	110*s*	211*s*
Sequential DP	57*s*	**104*s***	190*s*
Prioritizing DP	57*s*	123*s*	**186*s***

Table 4.2: Comparison of the performances of our approaches on Priority instance $I^{8(a)}$. Seeds are set to 1609 and 793 for the PRNG streams for moving and throwing actions respectively. Two collisions happened during the simulation with the Sequential Heuristic; no incidents took place when applying the Sequential DP; just one collision happened while simulating with the Prioritizing DP. For further details on the outputs of our approaches on this instance refer to Appendix A.2.

Resolution Method	**Completion of Order 1 (70s)**	**Completion of Order 2 (100s)**	**Completion of Order 3 (150s)**
Sequential Heuristic	52*s*	122*s*	228*s*
Sequential DP	**47*s***	112*s*	211*s*
Prioritizing DP	93*s*	**103*s***	**197*s***

Table 4.3: Comparison of the performances of our approaches on Priority instance $II^{8(b)}$. Seeds are set to 2404 and 610 for the PRNG streams for moving and throwing actions respectively. Many collisions happened during the simulation with the Sequential Heuristic; just one collision happened while simulating with the Sequential DP; no incidents took place when applying the Prioritizing DP.

Resolution Method	**Completion of Order 1 (120s)**	**Completion of Order 2 (130s)**	**Completion of Order 3 (190s)**
Sequential Heuristic	$\mathbf{59s}$	$150s$	$244s$
Sequential DP	$59s$	$\mathbf{146s}$	$230s$
Prioritizing DP	$121s$	$166s$	$\mathbf{222s}$

Table 4.4: Comparison of the performances of our approaches on Priority instance III$^{8(c)}$. Seeds are set to 1505 and 191 for the PRNG streams for moving and throwing actions respectively. Juts one collision happened during the simulation with the Sequential Heuristic; no incidents took place when applying the Sequential DP; two collisions happened while simulating with the Prioritizing DP.

Let us continue the discussion on how the Prioritizing DP approach enables achieving an overall demand fulfillment sooner, compared to employing the alternative sequential methods. In particular, we notice that for Priority instance I$^{8(a)}$ and III$^{8(c)}$, whose results are presented in Tables 4.2 and 4.4 respectively, the earlier completion is evident, despite collisions happened during the simulation. Moreover, we observe from Table 4.3 that the Prioritizing DP outperforms the sequential approaches also for Priority instance II$^{8(b)}$. However, we know that no incidents occurred during the simulation of its scheduling, while a collision took place when employing the Sequential DP, for example. Nevertheless, even in the absence of such collision, we would still observe superior performances for the Prioritizing DP, because the overall completion time for the sequential approach would decrease only to $206s$.
Applying the same rationale of balancing the results considering the collisions and failures happened during the simulations, we can also infer that the Sequential DP outperforms the Sequential Heuristic. However, the latter proves very fast and indeed represents a promising choice for the resolution of larger scale problems.

4.3.1 Larger Scale Solutions

As earlier mentioned, we are convinced that the Sequential Heuristic would represent a valid choice for quite accurately solving larger scale problems. This section serves to prove our thesis.
Furthermore, we also believe it is worth analysing the performances of the Sequential DP approach on larger instances. In fact, being its scheduling finer than the heuristic's and its resolution time faster than the Prioritizing DP's, its application to our problem could offer a trade-off of speed and accuracy. On the other hand, we will refrain from further examining the Prioritizing DP's performances, since in the last section the method already proved excessively computationally expensive.
The two large scale instances employed for the analysis are both characterized by

- $O = 5$ object types,
- $K = 1$ tray,

- $|\mathcal{N}| = 6$ graph vertices,
- maximum robot capacity $c = 4$.

Specifically, they are:

9a. Large Priority instance I:

Queue	Obj 1	Obj 2	Obj 3	Obj 4	Obj 5	Delivery time
Order 1	1	2	3	2	3	$230s$
Order 2	3	2	1	2	1	$370s$
Order 3	2	2	4	3	0	$550s$

- time horizon $T = 600$,
- $D = 3$ orders,
- total number of items to collect $Q = \sum_{d \in \mathcal{D}} \sum_{o \in \mathcal{O}} q_o^d = 31$;

9b. Large Priority instance II:

Queue	Obj 1	Obj 2	Obj 3	Obj 4	Obj 5	Delivery time
Order 1	0	0	1	2	2	$100s$
Order 2	0	2	1	1	1	$150s$
Order 3	3	0	0	0	0	$230s$
Order 4	1	2	0	0	1	$300s$
Order 5	2	0	2	0	1	$390s$

- time horizon $T = 430$,
- $D = 5$ orders,
- total number of items to collect $Q = \sum_{d \in \mathcal{D}} \sum_{o \in \mathcal{O}} q_o^d = 22$.

The problems defined above are *large in different ways*. Large Priority instance I[9(a)] presents a copious amount of items to collect during the entire mission, consequently requiring a longer-term time horizon. Instead, Large Priority instance II[9(b)]'s main characteristic is the elevate number of more modest orders. Such difference greatly affects the execution time of the Sequential DP. In fact, Table 4.5 shows that the approach is extremely slow for Large Priority instance I[9(a)], while its speed is acceptable for the other instance. Nevertheless, Sequential DP surely presents better performances for both cases, as expected (bold entries of Tables 4.6 and 4.7 refer to finer results).
We conclude by noticing that the Sequential Heuristic is highly efficient in all situations. Its results may not match the superior performance achieved when applying the DP paradigm, but are sufficiently close. Therefore, it stands out as the preferred choice when dealing with larger scale instances, especially if characterized by a significant number of objects to collect. However, in the case of low-quantity orders, one could still exploit the sequential approach with DP if seeking for superior results.

Resolution Method	**Instance**	**Execution Time**[11]
Sequential Heuristic	Large Priority instance I[9(a)]	$1.081s$
	Large priority instance II [9(b)]	$1.002s$
Sequential DP	Large Priority instance I[9(a)]	$15m49s$
	Large Priority instance II[9(b)]	$16.87s$

Table 4.5: Execution times of the sequential approaches for larger scale instances of the redefined scheduling problem with priorities.

Resolution Method	**Completion of Order 1 (230s)**	**Completion of Order 2 (370s)**	**Completion of Order 3 (550s)**
Heuristic	**120*s***	$322s$	$495s$
DP	$164s$	**313*s***	**484*s***

Table 4.6: Comparison of the performances of our sequential approaches on Large Priority instance I[9(a)]. Seeds are set to 2404 and 610 for the PRNG streams for moving and throwing actions respectively. Three collisions happened during the simulation with the Sequential Heuristic; two took place when applying the Sequential DP.

Resolution Method	**Order 1 (100s)**	**Order 2 (150s)**	**Order 3 (230s)**	**Order 4 (300s)**	**Order 5 (390s)**
Heuristic	$95s$	$184s$	$237s$	$302s$	$389s$
DP	**81*s***	**165*s***	**213*s***	**278*s***	**362*s***

Table 4.7: Comparison of the performances of our sequential approaches on Large Priority instance II[9(b)]. Columns refer to completion times of respective orders. Seeds are set to 2404 and 610 for the PRNG streams for moving and throwing actions respectively. Two collisions happened during the simulation with the Sequential Heuristic; none took place when applying the Sequential DP.

[11]Python Implementation on PyCharm Professional, Version 2022.3.2, on a machine equipped with RAM 16GB, intelCore i7, intel iRISx[e].

Chapter 5

Conclusions

Throughout the dissertation we have discussed the application of the Dynamic Programming paradigm and of its Approximate counterpart on a specific intralogistic robot scheduling problem. We envisioned a scenario where a single agent, referred to as robot, was tasked with the transportation of objects from boxes to trays situated in a warehouse. We modelled the warehouse as a completely connected undirected graph, on which edges the robot could move along, from vertices corresponding to box locations, where the robot could pick the various object types, to vertices corresponding to trays, where the robot could place or throw the items collected. We let the system be subjected to static risk factors associated to moving and throwing actions and distributed as Bernoulli random variables. The problem's main objective was to define a time-efficient and risk-aware scheduling of tasks for the robot.
We first observed that the chosen paradigm, in its exact version, provides optimal scheduling of tasks when employed for solving both deterministic and stochastic versions of our problem. However, due to the curses of dimensionality from which it suffers, its application revealed possible solely on small scale instances. Therefore, for the resolution of larger scale stochastic instances we relied on Approximate Dynamic Programming: we presented an Approximate Policy Iteration algorithm, and two look-ahead techniques, the Myopic Rollout and the Monte Carlo Tree Search. Through juxtaposition with the performances of the exact Dynamic Programming paradigm on small and medium scale instances we validated the approximate methods and demonstrated their accuracy. Later, we compared their performances to each other when applied to larger scale instances, analysing their execution times and evaluating the terminal states output by their scheduling simulations. The experiments conducted on the approximated approaches showed a significant reduction in execution times with respect to the application of the exact paradigm. Moreover, their solutions on small and medium instances presented a noteworthy accuracy compared to the solutions given by the application of exact Dynamic Programming, used as benchmarks. Furthermore, we observed an evident trade-off between the optimality of the scheduling and the time required for its planning, even in the case of larger scale instances. For example, the Approximate Policy Iteration algorithm generally exhibits performances closely aligned with the benchmark, but its runtime seems excessively expensive with respect to the more rapid Myopic Rollout, which in exchange presents slightly less accurate

results. Since such trade-off is inevitably encountered whenever it is required to solve a real-life instance of our problem, we concluded that the choice of the most suitable approach depends on specific necessities; if immediate outputs are needed one might opt for a less accurate resolution method, and, vice versa, if performances are of essential importance longer execution times are tolerated. However, the Monte Carlo Tree Search proved to be an adequate compromise in all analysed scenarios.
In the last chapter, our focus shifted to the real-life complication of priorities, whose introduction forced a conceptual reformulation of the problem. Henceforth we regarded the problem's mission as a list of orders to be fulfilled within specified deadlines, i.e., delivery times, and the trays as assembly spots for the orders. Moreover, we defined a static queue where demands were sorted by their delivery times, so as to define a service precedence. Then, for the resolution of such redefined scheduling problem with priorities, whose objective became the completion of the orders before (or close to) the associated deadlines, we proposed three different approaches. We implemented two sequential methods that serve the orders in the queue one at a time, starting from the most urgent, and a more complex approach serving more orders simultaneously. Among the two sequential techniques, one follows a heuristic procedure for the filling of the unique tray, while the other one relies on the application of the exact Dynamic Programming paradigm, as also does the third method. In alignment with the findings of the other chapters, we observed how the application of the Dynamic Programming paradigm yields superior results, unfortunately counterbalanced by slower execution times. Nevertheless, we succeeded in defining two sequential methods that, other than respecting our priorities assumptions, can generally be applied for the resolution of larger scale instances due to their acceptable runtimes.
Be aware that many of the considerations we discussed throughout the thesis still hold for alternative task scheduling scenarios with similar problem definitions. In fact, all our methods can be extended to situations where certain restricting assumptions are discarded, as we hinted on how to overcome eventual issues.

5.1 Further Developments

We extensively analysed the application of Dynamic and Approximate Dynamic Programming to our scheduling problem, despite the reliance on various constraining assumptions. For instance, we treated risk factors as static in nature and thus maintained them fixed throughout our simulations. However, over longer time horizons, it is plausible that they may dynamically change, particularly those linked to moving actions involving collisions with humans, who are mobile as well. A future topic of research could indeed involve more extensive experimentation with dynamic risks and incorporate a suitable rolling procedure to our scheduling problem. The values of risk factors could be left unchanged for a specified duration, then redefined as a consequence of the resolution of an updated routing optimization problem on the graph modelling the warehouse.
Similarly, for the redefined scheduling problem with priorities, the queue could be regarded as dynamic, and further experiments could be conducted after having defined a suitable stochastic process for the arrival of the orders.

Regarding the implementation of the algorithms we opted for native Python language. However, speed-enhancing techniques, as leveraging Python packages like Numba, could be employed to optimize execution times. In fact, immediate solutions may reveal essential in a real-life application, in which case one might even consider opting for more efficient languages and software to enhance performance. Nevertheless, our choice does not compromise the validity of our conclusions: our experiments enable valuable comparisons under a time perspective despite the overall slower runtimes.
A last further development could concern a more extensive search of the optimal parameters for the approximated approaches since our focus on reward and feature engineering was constrained by our limited resources. Even though we succeeded in presenting valuable results, we are confident that allocating the deserved additional resource to the selection of certain parameters could enhance the performances of our presented methods.

Appendix A

Outputs

For the overall clarity of the work, it is of essential importance to present a selection of the outputs of some of the methods discussed in the dissertation. The only aim is to make the reader aware of the kind of results we refer to in the text, hence it will not be necessary to present all of them.

A.1 Exact DP

A.1.1 Deterministic instance

This section presents the results output by the application of the exact DP paradigm to the deterministic version of the four toy instances introduced in Chapter 2, Section 2.2.

```
# MINI INSTANCE

# Scheduled sequence of tasks
['pick 0', 'pick 0', 'pick 0', 'move np1', 'pick 1', 'move nt0',
'place (0, 0)', 'place (0, 0)', 'place (0, 0)', 'place (1, 0)',
'move np2', 'pick 2', 'pick 2', 'move np1', 'pick 1', 'move nt0',
'place (1, 0)', 'place (2, 0)', 'place (2, 0)']

# Terminal state
[101, 'nt0', 3, 3, 2, 2, 2, 2]
```

```
# MEDIUM-SMALL INSTANCE

# Scheduled sequence of tasks
['pick 0', 'pick 0', 'move np1', 'move np2', 'pick 2', 'move np3',
'pick 3', 'move np4', 'move nt1', 'place (0, 1)', 'place (0, 1)',
'place (2, 1)', 'place (3, 1)', 'move np3', 'pick 3', 'move np2',
'pick 2', 'pick 2', 'move np1', 'pick 1', 'move nt0', 'place (1, 0)',
'place (2, 0)', 'place (2, 0)', 'move np2', 'move np3', 'pick 3',
'move np4', 'move nt1', 'place (3, 1)', 'place (3, 1)']

# Terminal state
[142, 'nt1', 2, 0, 2, 1, 1, 0, 3, 2, 1, 3, 0, 3, 0, 0, 0]
```

```
# MEDIUM INSTANCE

# Scheduled sequence of tasks
['pick 0', 'pick 0', 'pick 0', 'move np1', 'pick 1', 'move nt0',
'place (0, 0)', 'place (0, 0)', 'place (0, 0)', 'place (1, 0)',
'move np1', 'pick 1', 'pick 1', 'move np0', 'pick 0', 'pick 0',
'move nt1', 'place (0, 1)', 'place (0, 1)', 'place (1, 1)',
'place (1, 1)', 'move np2', 'pick 2', 'pick 2', 'pick 2', 'move nt0',
'place (2, 0)', 'place (2, 0)', 'move nt1', 'place (2, 1)']

# Terminal state
[168, 'nt1', 5, 3, 2, 3, 1, 2, 3, 2, 1]
```

```
# LARGE INSTANCE

# Scheduled sequence of tasks
['pick 0', 'move np1', 'pick 1', 'pick 1', 'move nt0', 'place (0, 0)',
'place (1, 0)', 'place (1, 0)', 'move np2', 'pick 2', 'move np3',
'pick 3', 'pick 3', 'pick 3', 'move np4', 'move nt1', 'place (2, 1)',
'place (3, 1)', 'place (3, 1)', 'place (3, 1)', 'move np4', 'pick 4',
'pick 4', 'move np1', 'pick 1', 'move np0', 'pick 0', 'move nt2',
'place (0, 2)', 'place (1, 2)', 'place (4, 2)', 'move np1', 'pick 1',
'pick 1', 'move nt0', 'place (4, 0)', 'move nt1', 'place (1, 1)',
'place (1, 1)']

# Terminal state
[202, 'nt1', 2, 1, 0, 1, 5, 2, 2, 1, 1, 0, 1, 0, 3, 0, 3, 0, 2, 1, 0, 1]
```

Bear in mind that, in a deterministic context, the scheduling never depends on the evolution of the system. An optimal sequence of tasks is determined through a backward pass before starting the scheduling and it does not vary dynamically. Therefore, the outputs presented above do not change if experiments are repeated.

A.1.2 Stochastic instance

This section presents the results output by the application of the exact DP paradigm to the stochastic version of the four toy instances introduced in Chapter 2, Section 2.2.
In this case, each output sequence of tasks is dynamic and depends on the realization of external risk factors during the specific simulation. Note that failures are explicitly indicated in the solution through the use of brackets.

```
# MINI INSTANCE

# Dynamic sequence of tasks
['pick 0', 'pick 0', 'pick 0', 'move nt0 (COLLISION)', 'throw (0, 0)',
'throw (0, 0)', 'throw (0, 0)', 'move np1', 'pick 1', 'pick 1',
'move np2', 'pick 2', 'pick 2', 'move nt0', 'throw (1, 0)',
'throw (1, 0)', 'throw (2, 0)', 'throw (2, 0)']

# Terminal state
[106, 'nt0', 3, 3, 2, 2, 2, 2]
```

```
# MEDIUM SMALL INSTANCE

# Dynamic sequence of tasks
['move np1', 'pick 1', 'move np2', 'pick 2', 'pick 2', 'pick 2',
'move nt0', 'throw (1, 0)', 'throw (2, 0)', 'throw (2, 0)', 'move np0',
'pick 0', 'pick 0', 'move nt1', 'throw (0, 1)', 'throw (0, 1)',
'move np3 (COLLISION)', 'pick 3', 'pick 3', 'pick 3', 'move nt1',
'throw (2, 1)', 'throw (3, 1)', 'throw (3, 1)', 'throw (3, 1)']

# Terminal state
[146, 'nt1', 2, 0, 2, 1, 1, 0, 3, 2, 1, 3, 0, 3, 0, 0, 0]
```

```
# MEDIUM INSTANCE

# Dynamic sequence of tasks
['pick 0', 'pick 0', 'pick 0', 'move np1', 'pick 1', 'move nt0',
'throw (1, 1) (FAILED)', 'throw (0, 1)', 'throw (0, 0)', 'throw (0, 0)',
'move np2', 'pick 2', 'pick 2', 'move np1', 'pick 1', 'move np0',
'pick 0', 'move nt0', 'throw (0, 0)', 'throw (1, 0)', 'throw (2, 0)',
'throw (2, 0)', 'move np1', 'pick 1', 'pick 1', 'move np0', 'pick 0',
'move np1', 'move np2', 'pick 2', 'move nt1', 'throw (0, 1)',
'throw (1, 1)', 'throw (1, 1)', 'throw (2, 1)']

# Terminal state
[178, 'nt1', 5, 3, 2, 3, 1, 2, 3, 2, 1]
```

```
# LARGE INSTANCE

# Dynamic sequence of tasks
['pick 0', 'pick 0', 'move np1', 'pick 1', 'pick 1', 'move nt0',
'throw (1, 1)', 'throw (1, 1)', 'throw (0, 2) (FAILED)', 'throw (0, 0)',
'move np2', 'pick 2', 'move np3', 'pick 3', 'pick 3', 'pick 3',
'move nt1', 'throw (2, 1)', 'throw (3, 1)', 'throw (3, 1)',
'throw (3, 1)', 'move np4', 'pick 4', 'pick 4', 'move np1', 'pick 1',
'move np0', 'pick 0', 'move nt2', 'throw (4, 0) (FAILED)',
'throw (0, 2)', 'throw (1, 2)', 'throw (4, 2)', 'move np1', 'pick 1',
'pick 1', 'move np2', 'move np3', 'move np4', 'pick 4', 'move nt0',
'throw (1, 0)', 'throw (1, 0)', 'throw (4, 0)']

# Terminal state
[227, 'nt0', 2, 1, 0, 1, 5, 2, 2, 1, 1, 0, 1, 0, 3, 0, 3, 0, 2, 1, 0, 1]
```

A.2 Problem with priorities

In the following, the scheduling produced by the three different approaches chosen to solve the redefined version of the intralogistic robot scheduling problem with priorities (Chapter 4). All three outputs regard the application of the methods on Priority instance $I^{8(a)}$.

A.2.1 Sequential heuristic

The sequences below represent the output given by the naive Sequential Heuristic approach. There are three distinct sequences because orders are served sequentially, being there a single available tray. Note, in fact, that for throwing actions the associated tray is not specified.

```
# First order's dynamic sequence of tasks
['pick 0', 'move np1', 'pick 1', 'move np2', 'move np3', 'pick 3',
'pick 3', 'move nt0', 'throw 0', 'throw 1', 'throw 3', 'throw 3']
# Intermediate state reached
[57, 'nt0', 1, 1, 1, 1, 0, 0, 2, 2, 0, 0]]

# Second order's dynamic sequence of tasks
['move np0', 'move np1', 'pick 1', 'move np2', 'pick 2', 'pick 2',
'move np3', 'move np4', 'move nt0', 'throw 1', 'throw 2', 'throw 2']
# Intermediate state reached
[110, 'nt0', 0, 0, 1, 1, 2, 2, 0, 0, 0, 0]]

# Third order's dynamic sequence of tasks
['move np0 (COLLISION)', 'pick 0', 'move np1', 'pick 1', 'move np2',
'pick 2', 'move np3', 'move np4', 'pick 4', 'move nt0', 'throw 0',
'throw 1', 'throw 2', 'throw 4', 'move np4 (COLLISION)', 'pick 4',
'move nt0', 'throw 4']
# Terminal state
[211, 'nt0', 1, 1, 1, 1, 1, 1, 0, 0, 2, 2]]
```

A.2.2 Sequential DP

The sequences below represent the output given by the Sequential DP approach. Being it sequential, as for the approach whose output are presented above, there are three distinct sequences and a single tray, which is thus not specified when referring to throwing actions.

```
# First order's dynamic sequence of tasks
['pick 0', 'move np1', 'pick 1', 'move np2', 'move np3', 'pick 3',
'pick 3', 'move nt0', 'throw 0', 'throw 1', 'throw 3', 'throw 3']
# Intermediate state reached
[57, 'nt0', 1, 1, 1, 1, 0, 0, 2, 2, 0, 0]]

# Second order's dynamic sequence of tasks
['move np2', 'pick 2', 'pick 2', 'move np1', 'pick 1', 'move nt0',
'throw 1', 'throw 2', 'throw 2']
# Intermediate state reached
[104, 'nt0', 0, 0, 1, 1, 2, 2, 0, 0, 0, 0]]

# Third order's dynamic sequence of tasks
['move np2', 'pick 2', 'move np3', 'move np4', 'pick 4', 'pick 4',
'move nt0', 'throw 2', 'throw 4', 'throw 4', 'move np1', 'pick 1',
'move np0', 'pick 0', 'move nt0', 'throw 0', 'throw 1']
# Terminal state
[190, 'nt0', 1, 1, 1, 1, 1, 1, 0, 0, 2, 2]]
```

A.2.3 Prioritizing DP

This section presents the result output by the application of the exact DP paradigm to the redefined problem with priorities. Here, only one sequence of tasks is presented, since orders are served simultaneously and substituted as soon as a tray becomes available. Note that the output shows when a tray is full, i.e., an order's demand is fulfilled.

```
# Dynamic sequence of tasks
['pick 0', 'move np1', 'pick 1', 'move np2', 'move np3', 'pick 3',
'pick 3', 'move nt0', 'throw (0, 0)', 'throw (1, 0)', 'throw (3, 0)',
'throw (3, 0)', 'tray0 is full! Must change it.', "Now tray0 contains a
new order {'objectA': 1, 'objectB': 1, 'objectC': 1, 'objectD': 0,
'objectE': 2}", 'move np1 (COLLISION)', 'pick 1', 'pick 1', 'move np2',
'pick 2', 'pick 2', 'move nt1', 'throw (1, 0)', 'throw (1, 1)',
'throw (2, 1)', 'throw (2, 1)', 'tray1 is full!', 'move np4', 'pick 4',
'pick 4', 'move np2', 'pick 2', 'move np0', 'pick 0', 'move nt0',
'throw (0, 0)', 'throw (2, 0)', 'throw (4, 0)', 'throw (4, 0)']

# Terminal state
[186, 'nt0', 1, 1, 0, 2, 1, 1, 3, 1, 2, 0, 0, 0, 2, 2, 0]
```

A.3 Implementation details

Note that in the printed outputs object types are referred to as numbers from 0 to 4 instead of 1 to 5, and trays as numbers from 0 to 2 instead of 1 to 3: Python was chosen as implementation language. All code implementation and further details can be found at https://github.com/margheritabattistotti/opt_robot_scheduling-withADP.git.

Bibliography

Russel Allgor, Tolga Cezik, and Daniel Chen. Algorithm for robotic picking in amazon fulfillment centers enables humans and robots to work together effectively. *INFORMS Journal on Applied Analytics*, 53(4):266–282, 2023. doi: https://doi.org/10.1287/inte.2022.1143.

αHead Research. About αHead. https://ahead-research.com/en/discover-spindox/. Accessed: 2023-09-29.

Vincent Babin and Clément Gosselin. Mechanisms for robotic grasping and manipulation. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(1):573–593, 2021. doi: https://doi.org/10.1146/annurev-control-061520-010405.

Paolo Brandimarte. *From Shortest Paths to Reinforcement Learning*. Springer, 2021.

EU DARKO Project. About DARKO. https://darko-project.eu/about/. Accessed: 2024-02-16.

Frederick Hillier and Gerald Lieberman. *Introduction to Operations Research*. McGraw-Hill Education, 2021.

Levente Kocsis and Csaba Szepesvári. *Bandit Based Monte-Carlo Planning*, pages 282–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi: https://doi.org/10.1007/11871842_29.

Maximillian Löfflet, Nils Boysen, and Micheal Schneider. Human-robot cooperation: Coordinating autonomous mobile robots and human orders pickers. *Transportation Science*, 57(4):979–998, 2023. doi: https://doi.org/10.1287/trsc.2023.1207.

Warren B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley, 2011.

Warren B. Powell. *Reinforcement Learning and Stochastic Optimization: A unified framework for sequential decisions*. Wiley, 2019.

Adrien Rimelé, Michel Gamache, Michel Gendreau, Grangier Philippe, and Rousseaus Louis-Martin. Robotic mobile fulfillment systems: a mathematical modelling framework for e-commerce applications. *Taylor Francis Journals*, 60(11):3589–3605, 2022. doi: https://doi.org/10.1080/00207543.2021.1926570.

Spindox. When the challenge for innovation becomes serious, Spindox begins to stand out. https://www.spindox.it/en/. Accessed: 2024-02-16.

Bence Tipary and Gábor Erdős. Generic development methodology for flexible robotic pick-and-place workcells based on digital twin. *Robotics and Computer-Integrated Manufacturing*, 71, 2021. doi: https://doi.org/10.1016/j.rcim.2021.102140.

Zheng Wang, Jiuh-Biing Sheu, Chung-Piaw Teo, and Guiqin Xue. Robot scheduling for mobile-rack warehouses: Human–robot coordinated order picking systems. *Prod Oper Manag*, 31:98–116, 2022. doi: https://doi.org/10.1111/poms.13406.

Mark H. M. Winands. *Monte-Carlo Tree Search*, pages 1–6. Springer International Publishing, Cham, 2015. doi: https://doi.org/10.1007/978-3-319-08234-9_12-1.

P.Th. Zacharia and N.A. Aspragathos. Optimal robot task scheduling based on genetic algorithms. *Robotics and Computer-Integrated Manufacturing*, 21(1):67–79, 2005. doi: https://doi.org/10.1016/j.rcim.2004.04.003.