technologies

Article

Multi-Arm Trajectory Planning for Optimal Collision-Free Pick-and-Place Operations

Daniel Mateu-Gomez [1], Francisco José Martínez-Peral [2,*] and Carlos Perez-Vidal [2]

[1] Analog Devices Spain, Parc Cientific Universitat de Valencia, C/Catedratico Agustin Escardino, 9, 46980 Paterna, Spain; daniel.mateu@analog.com
[2] Instituto de Investigación en Ingeniería I3E, Miguel Hernández University, Av. de la Universidad s/n, 03202 Elche, Spain; carlos.perez@umh.es
* Correspondence: francisco.martinezp@umh.es

Abstract: This article addresses the problem of automating a multi-arm pick-and-place robotic system. The objective is to optimize the execution time of a task simultaneously performed by multiple robots, sharing the same workspace, and determining the order of operations to be performed. Due to its ability to address decision-making problems of all kinds, the system is modeled under the mathematical framework of the Markov Decision Process (MDP). In this particular work, the model is adjusted to a deterministic, single-agent, and fully observable system, which allows for its comparison with other resolution methods such as graph search algorithms and Planning Domain Definition Language (PDDL). The proposed approach provides three advantages: it plans the trajectory to perform the task in minimum time; it considers how to avoid collisions between robots; and it automatically generates the robot code for any robot manufacturer and any initial objects' positions in the workspace. The result meets the objectives and is a fast and robust system that can be safely employed in a production line.

Keywords: pick-and-place operations; robotic sequence order optimization; dual-arm collision avoidance; Markov Decision Process; PDDL in robotic manipulation

Citation: Mateu-Gomez, D.; Martínez-Peral, F.J.; Perez-Vidal, C. Multi-Arm Trajectory Planning for Optimal Collision-Free Pick-and-Place Operations. *Technologies* **2024**, *12*, 12. https://doi.org/10.3390/technologies12010012

Academic Editor: Ahmad Lotfi

Received: 9 November 2023
Revised: 22 December 2023
Accepted: 16 January 2024
Published: 22 January 2024

1. Introduction

Production lines increasingly require short and customized manufacturing batches tailored to customer demand, which affects the configuration of the factories and their level of automation [1]. Factory systems must become more flexible and adaptable [2,3]. On the other hand, robots are transitioning from being rigid, pre-programmed, and almost "blind" to being collaborative, easy to reprogram and equipped with a large number of sensors. Consequently, this new generation of robots is increasingly being used in assembly, disassembly, or packaging lines of small factories or reduced batches. By operating in a shared workspace with human operators, production systems can further increase efficiency and minimize the workspace area. However, this creates a problem in terms of collision control. Figure 1 shows a group of robots sharing the same workspace and performing a common pick-and-place task. In this case, the group of robots needs to place pieces of the same color in their corresponding trays without colliding with each other and completing the task as efficiently as possible.

Typically, robot trajectories are programmed to ensure there is no collision between the robot and a static environment. Since robots are often used for repetitive tasks, it is sufficient to plan collision-free trajectories only once. If certain parts of the production process are modified, it becomes necessary to re-plan collision-free trajectories and reprogram all manipulators. In cases where the environment is unpredictable, such as when multiple robots are present, operators modify the workspace, or there are changes in the production process conditions (e.g., the repositioning of production elements), modular approaches are

required where each robot can react to changes in real time. The ability to optimally adapt to flexible manufacturing needs, would be an advantage in new production processes.

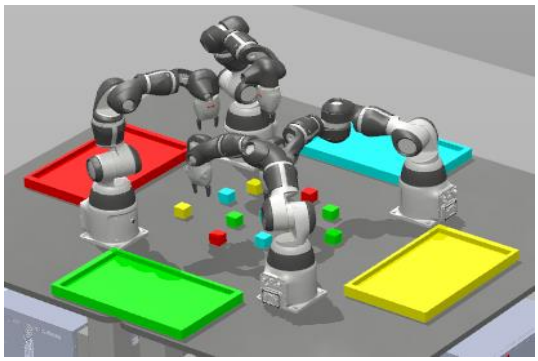
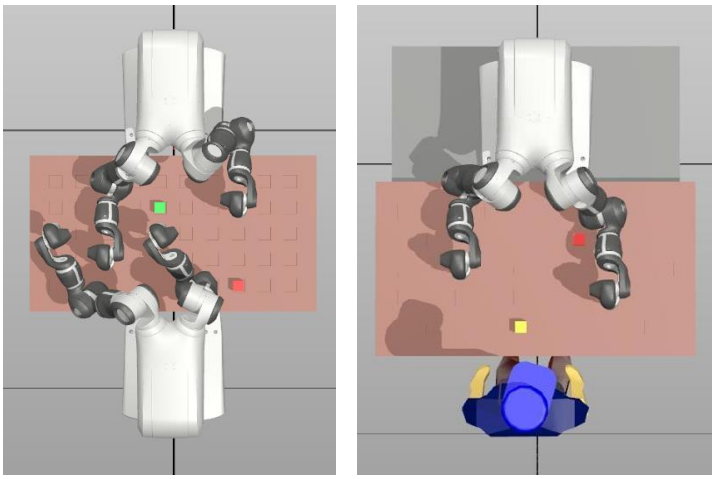


Figure 1. Pick-and-place scenario with four collaborative ABB IRB 14,050 robots.

Object manipulation involves not only performing tasks while avoiding collisions between robots and the environment but also doing so in the shortest possible time. Figure 2 depicts a representation of a production process where a series of pick-and-place operations need to be performed in a non-predefined order. This means that a certain number of pieces (e.g., kitting operation [4]) must be manipulated in any order, while considering the establishment of a pick-and-place operation sequence that minimizes the operation time. In this sense, a set of robotic arms, as shown in Figure 2a, or a dual-arm robot and a human operator, as shown in Figure 2b, could be considered. This last case is particularly relevant because the system needs to be adapted to the "disappearance" of pieces grabbed by the operator, which entails an instantaneous change from one state to another, requiring the recalculation of the optimal solution.



(**a**) (**b**)

Figure 2. Pick-and-place scenario with: (**a**) two collaborative ABB YuMi robots (IRB 14,000); (**b**) one collaborative robot and a human operator sharing the workspace.

A particular case, as shown in Figure 3, is when a specific robot (e.g., left robot) requires the collaboration of another (e.g., right robot) to reach a piece and place it in the desired location. This type of task involves more than just trajectory planning; it requires high-level task planning, which becomes even more complex when a task must be completed in the shortest possible time.

This research introduces a new approach for effectively and efficiently solving pick-and-place tasks by using machine learning algorithms and smart architectural design. It becomes more significant in collaborative scenarios with multiple pieces and multiple

robots sharing a common workspace. It plays an automation role, eliminating the requirement for a human to manually create the trajectories for the robots, and placing emphasis on task time minimization. This study shows this approach applied to a scenario comprising multiple pieces, ranging from 2 to 10, and a pair of robots, specifically a collaborative robot with two arms.

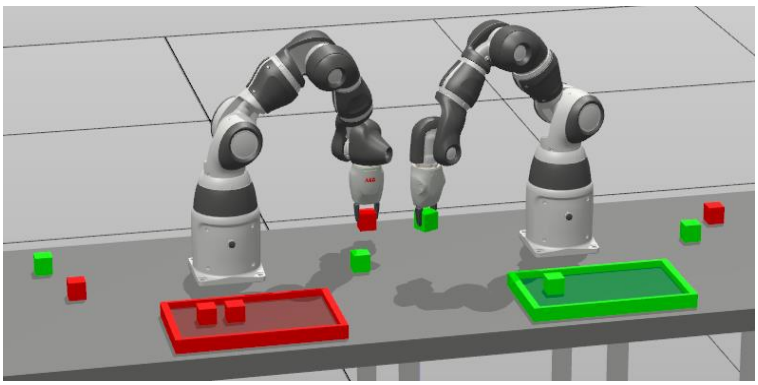


Figure 3. Collaboration required between both arms to complete the task.

This article is organized as follows. Section 2 provides a brief analysis of the current state-of-the-art in pick-and-place algorithms. Section 3 describes the proposed approach, formulated as a discretization of the robot's workspace. The actions that the robot can perform to solve the problem and collision avoidance system are presented. Section 4 focuses on solving the problem, addressed as a Markov Decision Process (MDP) and as a planning problem using Planning Domain Definition Language (PDDL). The study's results are presented in Section 5, where, in addition to demonstrating the algorithm's performance, the results obtained in a visual environment are shown. Simulations provide an intuitive evaluation of the outcome. Finally, Section 6 presents the conclusions and proposes future work in this field.

2. Related Work

In recent years, there has been an increasing emphasis on developing algorithms to optimize the pick-and-place task, with a particular focus on minimizing operation time. This challenge has been addressed in different applications, such as shoe production [5], PCB assembly [6], and the pick-and-place of individual nanowires in Scanning Electron Microscopy (SEM) [7]. Previous research has addressed different subproblems related to pick-and-place optimization, including placement sequencing and feeder allocation [8], among others. Minimizing operation time has been extensively explored in the field using metaheuristic algorithms. Given the complexities associated with these problems, heuristic and metaheuristic algorithms are widely employed to solve them. Examples of such algorithms include Particle Swarm Optimization (PSO) [9], the Genetic Algorithm (GA) [10], and Ant Colony Optimization (ACO) [11]. These methods employ stochastic search techniques that emulate biological or natural evolution and natural selection. One advantage of these methods is that they often provide solutions that are close to optimal in most cases [12].

Several studies have employed mathematical algorithms in industrial pick-and-place processes. In [13], an ACO algorithm was proposed, which outperformed the Exhaustive Enumeration Method (EEM), although the study did not specifically address multi-objective pick-and-place processes. GA and ACO algorithms presented better solutions in terms of execution time in pick-and-place problem-solving for PCB assembly, compared to PSO and SFLA algorithms due to their optimized structures [8]. The Hybrid Iterated Local Search (IHLS) algorithm combines local search and integer programming to drastically reduce computing time for large-scale processes compared to other heuristics. In [12], a novel metaheuristic algorithm called Best Uniformity Algorithm (BUA) was developed.

It generates random solutions within a search space to find the optimal pick-and-place sequence. The BUA algorithm achieved remarkable results in less time when compared to the GA algorithm, although for smaller problem sizes. Its implementation focused on a sequential pick-and-place process involving a single head moving one object at a time. A similar approach was presented in [5], where a Decision Tree Algorithm optimized the pick-and-place sequence using a two-armed robot to place shoe components into a mold. The Decision Tree Algorithm was employed to recognize patterns and calculate the best real-time optimized sequence. The results indicated that implementing the algorithm improves performance compared to random planning.

While [14] shares a similar objective, it diverges in terms of approach. This work focuses on collision-free path planning using priority arbitration based on the distance to the goal. However, unlike the referenced work, it does not incorporate prior planning of the operation order to minimize the task execution time. This aspect has been successfully addressed in the algorithm proposed in this article.

3. Approach to This Research

3.1. Materials

The setup used in this research comprises multiple pieces, ranging from 2 to 10, and it is operated by one of the most widespread collaborative bimanual robots, the ABB YuMi IRB 14000. The YuMi robot is designed for small-piece manipulation tasks and can cooperate with humans in the same environment without the need for a protection cell, as shown in Figure 2b. It is a redundant robot, featuring 7 degrees of freedom (DOF) per arm and it is equipped with a gripper, used to manipulate the pieces.

The overall goal of the task is to carry a certain number of pieces to their designated location. Specifically, for that study, the destination of the pieces has been predetermined and the source location is provided in real time when the task is about to start. That information might be provided by a device such as the Intel D415 Stereo Depth Camera, but for this study that is emulated by arbitrarily generating those positions on each run. All the experiments in this study have been done using RobotStudio®.

3.2. Design of the System Architecture

The task under study requires considering several factors: which piece to assign to each robot, what order should the pieces be taken in, how to avoid collision between robots, which trajectory should follow a robot, and, in the end, how to minimize the overall task time. This study approaches this project from the perspective of a decision-making problem and the purpose is to devise a solution that can be implemented in real-world processes in the industry. When adopting a decision-making approach, it is essential to define the set of actions that can be taken within the system and understand the concept of the current state of the system. YuMi is a high-precision bi-manual robot featuring 14-DOF. A strategy based on direct control of the arm's joint configuration would be impractical from the perspective of a solution designed for real-world processes, due to the explosion on the combinatory. An over-discretized, low-resolution joints scenario would lack enough precision to undertake the task and would still be impractical; a hypothetical scenario with low 10 degrees resolution, would lead to an average joint bin size of 20 values and a combinatory exceeding a quadrillion combinations.

The approach presented in this study is based on the Cartesian coordinates of the robot. The robots' workspace, defined by their area of influence, has been discretized as a 3D constellation of points, henceforth grid, used as waypoints to guide the trajectory of the robots. The resolution of this grid is configurable, although, for this work, a grid of ten columns, five rows, and three heights has been defined, resulting in a total of one hundred and fifty possible waypoints, each separated by 100 mm. Following the strategy of Cartesian coordinates and based on the real-world applicability requirement, the proposed architecture consists of a dual-layer structure. At a high level, the decision-making controller manages all the high-level concepts previously mentioned, such as trajectory

generation, pieces assignment, etc. At a low level, the manufacturer's software and hardware layers are responsible for the robots' transitions, ensuring precision and robustness. This layering division enables the solution to be scaled to other robot models. Another key aspect of the design concerns the robot's movements. The robot is allowed to move towards any of its neighboring waypoints in the 3D grid. This decision facilitates the robot arms' synchronization and reduces the complexity of the problem. This synchronization happens at each time-step, where all robots point to their next waypoint target. Regarding the complexity, the number of waypoints to choose from is reduced to 27, which includes the option of staying at the current location if that is the best option. This is a significant improvement over the possibility of moving to any of the 150-grid waypoints. Based on that, robot trajectories have been represented as an ordered set of waypoints in the grid. At a low level, the manufacturer layer will execute a continuous and smooth trajectory guided by these waypoints (see Figure 4 for more information).

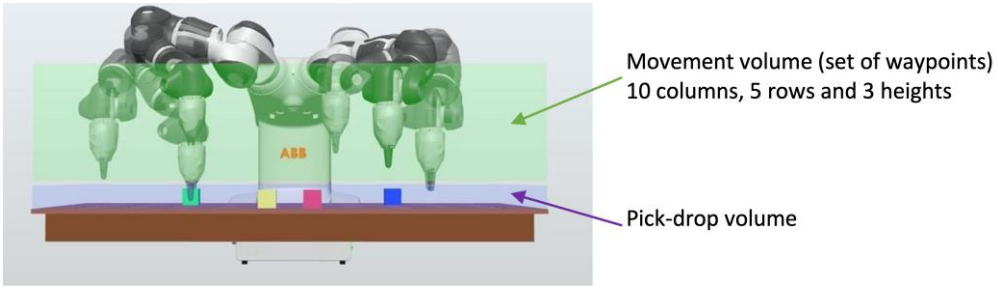


Figure 4. Visual description of volumes defined in the system architecture.

This architecture models the system as a set of robots and pieces, all defined as 3D points within the workspace. This, by itself, is not enough to account for some factors, such as the fact that robot bodies are not single points in the workspace, causing the problem of collisions, robots have a limited area of influence, gripper grasp operation details, and robot joints have angular range limitations, causing discontinuities in the movement if a joint limit is violated. To address these gaps, some additional design measures have been adopted. A prior analysis was carried out to collect information about the area of influence of each robot within the grid, the potential collision between robots for all grid location combinations, and the preferred robot joint configuration (see Section 3.3.5) for each arm and every grid location. All this information is stored in a database that the controller has access to. So, in essence, this architecture represents the robots as 3D points in the workspace, but accounting for their body presence and range of actuation by means of the database information. The last key factor of the design is the strategy followed to execute any robot trajectory. This architecture is, by nature, Cartesian-coordinates-based (transition from point A to point B), dealing with gripper 3D locations, but the actions taken by the controller are joint-based (transition from joint configuration Q1 to joint configuration Q2). The purpose is to gain direct control over the robots' joint configuration, instead of leaving that decision to the robot criteria (there are multiple valid joint configurations that satisfy the criteria for being in point B). This helps to ensure a natural posture avoiding discontinuities or undesired movements during transitions between neighboring waypoints.

3.3. Design of the System Model

The system's model represents the real-world problem by ignoring the irrelevant details and only keeping the essential ones for the solution. Based on the system's architecture, which has established the fundamentals of how the system works, it is essential to keep track of the location of robots, the assignment and status of the pieces, and the source and destination location of the pieces. Other relevant inputs, such as collision information or robot location viability, are derived from the fundamental features or internal variables that determine the system model. Specifically, this information is stored in a database that the model has access to.

3.3.1. State-Space

The state of the model is the representation of the current situation. For this study, it has been designed to be fully observable, therefore it captures all the relevant information that the controller needs in order to make a decision, as shown next:

$$
\begin{aligned}
state = f\left(robotPos_i, robotStatus_i, piecesStatus_j, initPiecesPos_j, endPiecesPos_j\right)\\
robotPos_i &= 3D\ location\ of\ the\ robot_i\ gripper,\ \forall i = 0 \ldots N\\
robotStatus_i &= \ Indication\ of\ the\ piece,\ if\ any,\ carried\ out\ by\ the\ robot_i, \forall i = 0 \ldots N\\
piecesStatus_j &= \ Indication\ whether\ the\ piece_j\ has\ been\ processed,\ \forall j = 0 \ldots K\\
initPiecesPos_j &= \ Source\ 3Dlocationofthepiece_j, \forall j = 0 \ldots K\\
endPiecesPos_j &= \ Target\ 3Dlocationofthepiece_j, \forall j = 0 \ldots K\\
N &= \ Number\ of\ robots\\
K &= \ Number\ of\ pieces
\end{aligned}
$$

Similarly, it has been designed to be discrete and single agent. The state is defined at the system level, not at the individual robot level, and captures, at the same time, the information of all the elements in the system. Following the defined dual-layer system architecture, the model sits on the high-level layer, where the robots transition from the current waypoint to the next one. Even though the final solution achieves smooth and continuous movements, thanks to the dual-layer design, the model is made up of a set of discrete variables.

The state-space is a concept that captures all the feasible configurations of the system. In this case, the state-space is finite, since all internal model variables are discrete, and its size evolves according to the equation shown in Equation (1).

$$Num\ states = (G + K{\cdot}P)^N{\cdot}(K+1)^N{\cdot}N^K \tag{1}$$

where G is the 3D grid size, K is the number of pieces, N is the number of robots, and P is the number of steps for pick or drop.

For this work, the number of robots is 2, the number of pieces ranges from 2 to 10, and the grid size is 150 ($10 \times 5 \times 3$). The $K{\cdot}P$ term comes as a result of applying a special treatment technique for the pick-and-place operations. The technique decouples the area of movement of the robots, used to reach and transport the pieces, from the pick-and-place operations themselves. The source and destination locations of the pieces are on an inferior plane of the volume filled by the grid. This technique prevents the need to enlarge the grid to accommodate the pieces on it. P refers to the number of time-steps required for the vertical moves during pick and place operations.

3.3.2. Action-Space

The system architecture defines the robot navigation as a sequence of transitions between neighboring waypoints in the grid. The number of neighboring waypoints is always 26 in a 3D grid, except for the boundary, as shown in Figure 5. Additionally, in a multi-robot setup, there is the option to stay in the same position. This might be the most effective way to evade a collision, or only because there is nothing else to process for that robot. Pick-up and drop-down operations are handled in different ways. Since the source and destination location of the pieces sit on a working table, on an inferior plane to the grid, this action cannot be executed by the regular intra-grid navigation moves. By design, those operations must be carried out by the robot in the lower plane of the grid and in the same XY coordinates as the source or destination location of the piece. These operations involve pure vertical moves and gripper open or close actions. In order to ensure a successful operation, the configuration of the robot places the gripper pointing towards the piece and the pieces have been defined as cubic blocks, just as it was done in [15].

The total number of available actions per robot is:

$$Num\ robot\ actions = \left(N_{navigate} + N_{stay} + N_{pick} + N_{drop}\right) = 26 + 1 + 1 + 1 = 29$$

At a system level, an action is defined as the aggregate of all actions performed by the robots. Since the action taken by a robot is independent from the action taken by the others, the total number of available system actions exponentially increases with the number of robots. In this case, for two robots it is:

$$Num\ system\ actions = Num\ robot\ actions^{N} = 29^{2} = 841$$

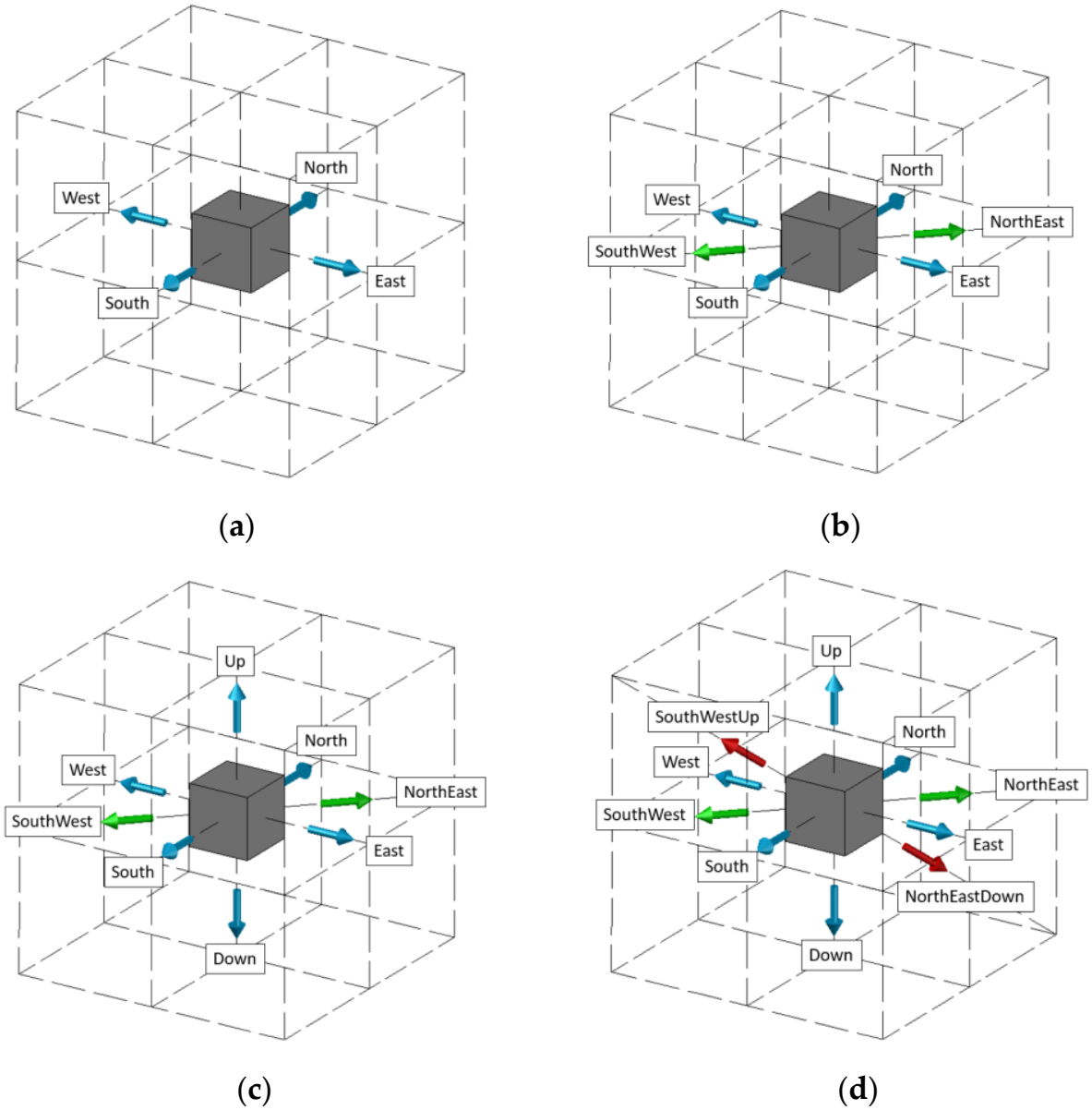


Figure 5. Action-space Modes defined in the project: (**a**) Orthogonal movements (Mode 1); (**b**) Orthogonal and diagonal movements (Mode 2); (**c**) Addition Up/Down movement (Mode 3); (**d**) Movement to all neighboring waypoints (Mode 4).

Navigation Nodes:

The computational complexity of the problem is related to the size of both state and action spaces. For the state-space, experiments with a different number of pieces were conducted. For the action-space, experiments with different sets of supported system actions have been conducted as well. To capture this idea, three additional incremental action modes have been specified, each containing a subset of the possible navigation actions:

1. Mode 1: Only orthogonal actions are allowed (North, South, East, West), as shown in Figure 5a, on the inferior plane of the grid.
2. Mode 2: This Mode adds diagonal actions (e.g., SouthWest, NorthEast, etc.), as depicted in Figure 5b, on the inferior plane of the grid.
3. Mode 3: This Mode adds Up and Down actions, as shown in Figure 5c. They enable robots to navigate the whole 3D grid.
4. Mode 4: This Mode considers all neighboring waypoint directions. It is the most flexible one, considering combinations of three orthogonal movements, as shown in Figure 5d (e.g., SouthWestUp, NorthEastDown, etc.).

As the robot increases its mobility with more degrees of freedom, combining movements along different axes, the number of time-steps required to complete the task might be reduced, at the expense of an increase in computation. That trade-off is evaluated in performed tests.

3.3.3. Reward-Space

The cost is a feedback indication, received by the controller when executing an action, and it is exclusively dependent on the current state of the system and the action taken on it. The controller makes use of it to plan a solution in the long term. This term is also known as reward in the Reinforcement Learning domain. By design, there are only two kinds of feedback, a general one to penalize any action and another one to indicate the success of the task. Specific values depend on the type of algorithm applied to solve the problem.

3.3.4. Integration with the Controller

Conceptually, the task involves multiple robots collaborating to transport a certain number of pieces in the least amount of time. For each piece, a robot must navigate to its location, pick it up, transport it to the designated location and drop it down. Robot navigation or piece transportation entails the gripper following a trajectory from point A to point B, which, according to defined system architecture, is formulated as a sequence of transitions between neighboring waypoints. The proposed system model operates at this abstraction level and takes advantage of robot capabilities to execute these transitions (from point A to point B).

The task starts at t_0 with the system model at state s_{t0}. The source and destination location of the pieces is not shown in the state representation because their values stay invariant during the whole task:

$$\begin{aligned} s_{t0} = (\ & robotPos_0 \dots robotPos_{N-1},\ robotStatus_0 \dots\ robotStatus_{N-1},\ pieceStatus_0\ \dots\ pieceStatus_{N-1}) \\ & robotPos_i = \ arbitraryposition,\ \forall i = 0 \dots N \\ & robotStatus_i = \ CARRYING\ NO\ PIECE,\ \forall i = 0 \dots N \\ & piecesStatus_j = \ PIECE\ NOT\ PROCESSED,\ \forall j = 0 \dots K \end{aligned}$$

An action a_{t0} is taken and the system state evolves to s_{t1} according to the system model behaviour. The pseudo-code simulating model behavior is shown in Algorithm 1. That process is repeated until the state s_{tM} becomes a goal state. There might be multiple goal states in the state-space. In this case, a goal state is defined by:

$$pieceStatus_i = PIECE\ PROCESSED, \forall i = 0 \dots K$$

Regarding the collision check on the pseudo-code, it is worth noting that it is carried out both before and after taking the action. Since the grid is small enough, it is assumed there is no collision during this transition if there is not either before or after, except for the case when robots exchange location. That assumption was tested in RobotStudio® as well.

Algorithm 1: Pseudo-code to emulate system model behavior.

```
// State is a tuple containing all internal Model variables
// Action is a tuple containing the actions taken by each robot
ModelBehaviour(state, action)
    Check valid action in this state
    if action is not valid then
        return state, HIGH_PENALTY
    end if
    Compute expected next_state based on current state and action
    Check grid boundaries for each robot
    Check collision between robots
    for I from 1 to N do
        Check accessibility of robot i to its new location
        if any check fails then
            # state is unchanged
            reward = HIGH_PENALTY
        else
            state = next_state
            reward = SMALL_PENALTY
        end if
    end for
return state, reward
```

3.3.5. Database

The database contains key information used to obtain the robot joint configuration, detect collision between robots, and locations not reachable by any robot. All this information is not encoded in the state-space because it is dependent on other internal variables of the state-space, specifically the robot location. The collision detection and accessibility of the robots to a specific location are critical inputs for the controller in order to make a good decision.

A preliminary study of collisions and accessibility was performed using RobotStudio®. Collision validation is based on the combination of all robots' locations in the grid. Its size exponentially grows with the number of robots, $(X \cdot Y \cdot Z)^N$, where X, Y, Z corresponds to the size of all three axes of the grid, and N to the number of robots. The output is a Boolean indication of collision for each test. For this study, $(X \cdot Y \cdot Z)^N = (10 \cdot 5 \cdot 3)^2 = 22{,}500$ tests. Accessibility validation is individually carried out per robot, and it is based on the robot location and its area of influence. Its size is equal to the size of the grid, so $(X \cdot Y \cdot Z) = 150$ tests. The output is a Boolean indication of a feasible location. All this information is stored in a database (DB) that the controller has access to.

Figure 6 graphically depicts some examples of collision indication for some arbitrary configurations stored in the DB.

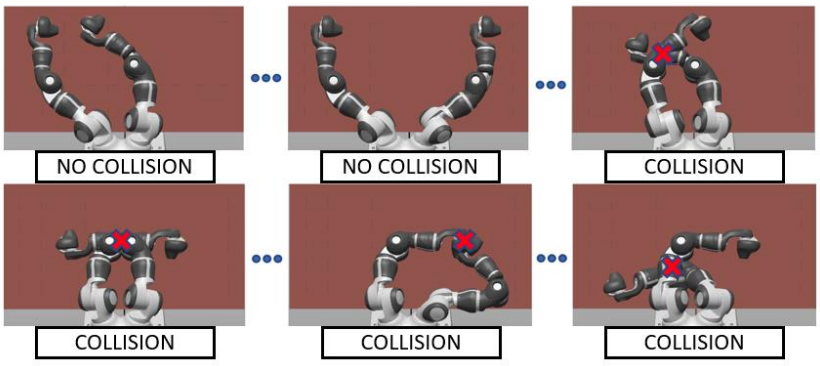


Figure 6. Visual representation of situations of collision detection.

The pseudo-code shown in Algorithm 2 has been used to collect the DB information, including the collision indication minimum distance between robots, and the accessibility and joint configuration ($\vec{q}$) per robot. This distance can be optionally used as a safety measure, forcing collision indication when it is lower than an established threshold. Its calculation involves obtaining the minimum distance between all links of one arm (α) and all links of the other arm (β), and keeping within the minimum among them, as indicated in Equations (2) and (3).

$$\delta_{ij} = \left\| \lambda i_{\alpha} - \lambda j_{\beta} \right\| \tag{2}$$

$$\delta_{min} = \left(\delta_{ij} \right) \tag{3}$$

where δ_{ij} is the distance between links i and j, i_{α} is the i link of the α arm, i_{β} is the i link of the β arm, and δ_{min} is the minimum value of all δ_{ij} calculated for a specific configuration.

Another key piece of information stored in the database is the joint configuration to be used by any robot based on the grid location. This association between grid location and joint configuration is unique. The selection of appropriate joint configurations is based on certain criteria. All the configurations meeting these criteria are called feasible configurations and all are equally eligible. Finally, one configuration among all feasible ones is stored in the database. The criteria used in this study consider the following factors: the robot gripper is always pointing towards the worktable, the arm's elbow is correctly positioned to avoid collisions with the objects in the working area, singularities (reaching an angular limit) during operation must not occur, ensuring a smooth transition between adjacent cells, and in turn, through the whole trajectory. At a high level, the database solves the kinematics of the robots.

The database is accessed as a LookUp Table (LUT). The LUT is indexed by the robot's location to obtain either the robot accessibility information or the associated joint configuration, and it is also indexed by the location of all the robots to obtain collision information.

Algorithm 2: Pseudo-code for data base generation (including collision detection, reachability, and joint configuration).

```
// For each arm of the robot get robot configuration
Move robot to initial position
for i from 1 to (X × Y × Z) do
    Read coordinates from file and RobTarget calculation
    if RobTarget = reachable then
        JointTarget calculation from RobTarget
        if JointPosError = not reachable or OutReach then
            Pos not reachable and RobTarget values set to 0
        end if
    Write Coordinates, Reachability and JointTarget data in file
    end if
end for
// For each height of the grid get collision data
Move α & β to initial position
for i from 1 to (X × Y × Z) do
    for j from 1 to (X × Y × Z) do
        Read α and β joints values from file
        Move α and β
        Check positions reachability
        if α & β = reachable then
            Measure distances between links
            Distance comparison and get δmin
            Write Coordinates & δmin info in file
        else
            Write Coordinates & δmin = 0 info in file
        end if
    end for
    end for
```

4. Solution of the Problem

The problem has been addressed as a decision-making challenge. Reinforcement Learning (RL), a growing field in Artificial Intelligence, is dedicated to this domain, encompassing a wide range of situations, including uncertainty, vast or continuous state and action spaces, the absence of a model, multi-agent environments, and so on. RL necessitates the representation of the model as a Markov Decision Process (MDP).

In this study, the architecture of the system and the model has been defined by design, hence the model is perfectly known, the state and action spaces are discrete, finite, deterministic, and relatively huge, but do not exceed the limits, requiring some kind of function approximator. The architecture is designed such that the controller acts as a single brain for all robots, making it a single agent. Given these properties, methods from other areas, such as graph search or planning languages, may be applied as well.

Figure 7 represents the whole experiment process for all three approaches (MDP, Graph search, and PPDL). The diagram illustrates a sequence of tasks that each approach needs to complete in a specific order. The flow is divided into three stages: Project Level, Task Level, and Execution Level.

The Project Level stage is common for all approaches and involves any task depending on predefined and invariable design decisions for the project, such as the robot selection (YuMi) or the grid resolution ($10 \times 5 \times 3$). This stage needs to be performed only once, unless any of those design decisions change, such as using a new robot. This stage is responsible for generating the system database (described in Section 3.3.5), which in turn, is an input for the Task Level stage. The database includes information related to robot location feasibility, the robots' collision, and the robots' joint configuration.

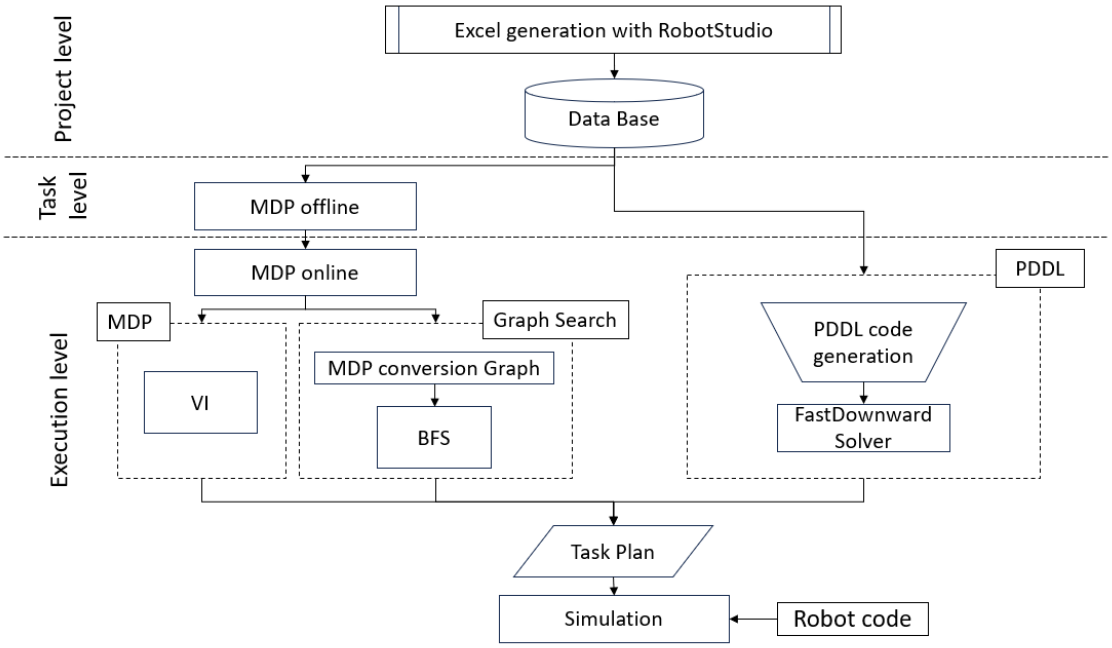


Figure 7. Process description for all three approaches (MDP, Graph search, and PDDL).

The Task Level stage is related to any task that involves parameter changes in successive experiments. This stage only needs to be re-executed if any of those parameters change. These parameters are the number of available actions, the grid size, and the number of pieces. The number of available actions and the grid size depend on the mode of action (described in Section 3.3.2) selected for the experiment. While still keeping the same grid resolution (100 mm between cells), some experiments restrict the robot movement to a plane (reducing the grid to $10 \times 5 \times 1$). The number of pieces ranges from 2 to 10. This stage only applies to approaches based on the MDP (MDP and Graph search) and it is

responsible for building the MDP model and saving it to disk. This stage is performed offline, and the resulting MDP model is valid for any parameter changes in the next stage.

The Execution Level is also known as the online stage. At this stage, the process receives information about the location of the pieces and robots. The entity providing the location of the pieces is out of the scope of this study. At this stage, the flow for each approach diverges.

The first step in the MDP approach is to update the MDP model. Even though the MDP model is built in the previous stage, the location of the pieces is unknown at that point, hence the state-action pairs related to pick or drop operations need to be updated in this stage once this information is available. MDP generation is split to reduce the online execution time, as most of the computation is performed offline. Once the MDP is built, an MDP solver can be applied to generate a solution. Value Iteration was selected (described in Section 4.1) for this study.

The first step in the Graph search flow is common with the MDP flow, the MDP model update. Once the MDP is fully defined, there is a conversion task devoted to transforming the MDP model into a graph (described in Section 4.2). This graph could be solved by any search algorithm. BFS was selected for this study (described in Section 4.2).

The last flow, PDDL, involves two tasks, the generation of the PDDL source code and generating a solution for the PDDL description. The first task is automated. This automation takes some inputs: the number of pieces, their locations, the robot locations, and the target, and generates the PDDL description for this particular problem. The solution generation is carried out by a PDDL planner solver. FastDownward was selected for this study.

All three approaches converge again in the Task Plan, where a common file format solution is generated. Specifically, for this study, this file contains the sequence of joint configurations and gripper actions required for each robot to complete the overall task.

The final step of the process involves implementing the solution to either a real or simulated problem. In this study, RobotStudio® was used to simulate the process. RobotStudio® requires the solution file, the process project (including the robot, the pieces, and the worktable), and the robot source code as inputs. The robot source code is written in RAPID, which is a programming language used for ABB robots. The robot program was used in all experiments to parse the solution file and execute the actions on the robots (see simulation video links in Section 5.2).

4.1. MDP Model

A Markov Decision Process is a mathematical framework for modeling decision-making problems under uncertainty. The four core components of an MDP model are: a set of states S, a set of actions A, the state transition function $P(s'|s,a)$, and the reward function $R(s,a,s')$. The transition function indicates the probability of reaching a state s' if an action a is taken in state s. The reward function indicates the reward obtained for that same transition. The key property of an MDP is the Markov property, which states that the effects of an action taken in a state only depend on that state and not on the prior history. Figure 8 represents this concept, where the next state s_{t+1} and reward r_{t+1} only depend on the current state s_t and the action taken in it.

In the context of this study, the agent is the brain making decisions, and the environment is the system under control, including the pieces and robots. Depending on the nature of the problem, the MDP may be fully known, partially known (i.e., the reward function is unknown), or totally unknown. In this work, the MDP is totally known (all four core components are defined by design). The state-space and action-space are described in detail in Sections 3.3.2 and 3.3.3, respectively.

The system is deterministic, so the state transition function in this study is not expressed in terms of probabilities $P(s,a,s')$, but as a function $T(s,a)$, which specifies the next state of the system given its current state and the action taken. This component has been modelled in the software implementation as a LookUp Table (LUT), which is indexed

by the current state and action and returns the next state. The logic used to populate the table has been implemented in Python and takes all the information into consideration (i.e., the grid boundary exceeded by a robot, collision between robots, location not reachable, invalid action on the current state, etc.).

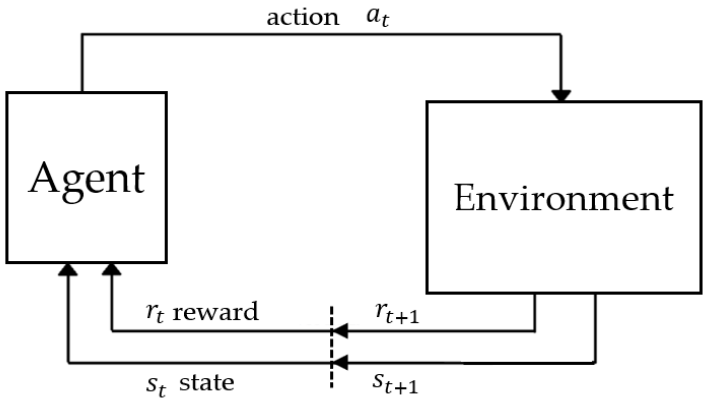


Figure 8. MDP model and building elements (for deterministic scenario).

For the same reason (deterministic system), the reward function is expressed in this study as $R(s,a)$ instead of $R(s,a,s')$. This component is also modelled as a LUT, and is indexed the same way as the transition function, but it returns the reward value. The logic to populate this LUT is implemented in Python as well. By design, there are three kinds of defined rewards:

$$R(s,a) = \begin{cases} 100 & if\ s' = s_{goal} \\ -1 & else\ if\ (s,a)\ is\ valid \\ -20 & else \end{cases} \quad (4)$$

A big reward (+100) indicates that the goal is met (all pieces have been transported). A big penalty (-20) indicates that the action taken on the state s is not valid (i.e., exceeding grid boundaries, collision between robots, etc.). Finally, a small penalty is applied in all other cases as a strategy for RL algorithms to find the optimal solution (smallest number of steps), since their purpose is to maximize the long-term reward.

The core idea behind this framework is simple. At any point in time, the agent takes an action based on the current system state. As a result, the system state changes, according to the Transition function, and the agent receives a feedback or reward, according to the Reward function.

The purpose of any RL algorithm over an MDP model is to find the optimal action to take on any state. This concept is called Policy, $\pi(s,a)$, and based on that policy the agent can make a sequence of optimal decisions from the initial state up to the goal state. Typically, RL algorithms are classified into two groups, model-based and model-free. Unlike model-free algorithms, model-based ones explicitly learn the model (reward and transition functions) of the system, and then apply planning over it. In this work, the model is known, and consequently, only the planning stage is required.

MDP Solver

There are many reinforcement learning algorithms that aim to learn the MDP model either implicitly or explicitly with the goal of finding a good policy [16]. In this study, the MDP is defined by design, hence there is no need to learn it. In this scenario, finding a good policy is a matter of planning. There are several dynamic programming algorithms that are well suited to this task, such as Value Iteration and Policy Iteration. Value Iteration (VI) was selected for this study.

The Value Iteration core component is called the Value function (V) and represents, for any given state s, the expected cumulative reward from that state s to the goal state, following a certain policy π. Initially, the policy $\pi(s)$ is usually arbitrary, so the Value

function $V^{\pi}(s)$ presented in Equation (5) is not a good estimation of the cumulative reward. The aim of Value Iteration is to find a policy that maximizes V, which is known as V^{*} presented in Equation (6).

$$V^{\pi}(s) = E_{\pi}\left[\sum_{k=0}^{H} \gamma^{k} R_{k+1} \middle| s_{t=s}\right] \tag{5}$$

$$V^{*}(s) = \max_{\pi}\left[\sum_{k=0}^{H} \gamma^{k} R_{k+1} \middle| \pi, s_{t=s}\right] \tag{6}$$

Indeed, it is an iterative algorithm, where $V^{\pi}(s)$ is gradually improved on each iteration by applying the Bellman equation presented as (7).

$$\mathrm{V}(s) = \max_{a} \sum_{s'} P_{a}(s'|s)\left[R(s,a,s') + \gamma \times V(s')\right] = \max_{a}\left[R(s,a) + V(s')\right] \tag{7}$$

The General Bellman equation may be simplified for deterministic environments when using additive discounting (a small negative reward for moving to non-terminal states). Under this scenario, the multiplicative discount (discount factor) may be set to 1, and the expression is reduced to its simplest form, as seen in (7).

The algorithm stops when V^{π} converges to V^{*}, since the Value function is not going to improve anymore, that is, the *V matrix* will remain the same on additional iterations. In practice, the algorithm usually stops when the variation in V is less than a certain predefined threshold. At this point, the optimal policy π^{*} can be directly derived from V^{*} and it is presented in Equation (8).

$$\pi^{*}(s) = \operatorname*{argmax}_{a}\left[R(s,a) + \gamma \times \sum_{S'} T(s,a,s') \times V^{*}(s')\right] = \operatorname*{argmax}_{a}\left[R(s,a) + V^{*}(s')\right] \tag{8}$$

Applying the same reasoning as in (7), the discount factor is set to 1 in (8). T is used to model the uncertainty in the environment. The system in this study is deterministic, hence there is only one state s' out of the whole state-space with a probability bigger than 0 (actually, probability 1). Under this scenario, this term can be removed, and the expression is reduced to its simplest form, as seen in (8).

It is worth noting the optimal V^{*} is unique, but there might be multiple π^{*} leading to this optimal Value function. This means that different sequences of actions might lead to the same result. This has a direct impact on this problem, specifically in the situation where a robot has nothing else to do (i.e., there are two robots and only one piece left).

The implementation of this algorithm was written in Python. The simplified Bellman Equation (7) was expressed in matrix form, and the matrix computations were performed using a numerical library called NumPy. Once the Value function has converged, the optimal policy is obtained using the simplified policy Equation (8).

From the point of view of the algorithm, it is irrelevant if one of the robots is idle or randomly moving. In practice, it is usually preferred that this robot keeps idle. To tackle this point, a smart ordering of the system actions was performed (system actions that affect only one robot's movement have priority over those in which both robots move), so in case of multiple actions leading to the same result, the preferred one will be returned by the argmax operation in Equation (8).

4.2. Graph Model

In this approach, the system model was represented as a graph. Specifically, the graph was directly derived from the matrix version of the MDP model (transition and reward functions) described in Section 4.1. A graph basically consists of three types of elements: nodes, transitions, and costs. There is a direct mapping between MDP states and graph nodes, and between MDP state transitions and graph node transitions. Graph costs are derived from the MDP rewards, although the mapping is not direct and specific

transformations have been performed. The resulting graph is a reduced version of the MDP model that incorporates the following reward-cost transformations, as shown in Table 1.

Table 1. MDP reward conversion to Graph cost.

MDP	Graph	Meaning
−20	-	Invalid MDP state-action transition
+100	+1	Direct-to-goal transition (very last transition of the path)
−1	+1	Any other transition

All invalid state-action pairs (i.e., an action in a specific state results in exceeding the grid boundaries) have not been included in the graph. The small penalty (−1) reward in the MDP is transformed as a small positive reward (+1), since the target of a graph search algorithm is to minimize the path cost, unlike MDP-based algorithms where their target is to maximize the long-term reward.

Finally, all direct-to-goal transitions in MDP (transitions from state s to s' where s' is a goal state) have the same reward value (+100). The Graph search algorithm's target is to minimize the path cost, and the key point is that every solution (optimal or not) will include one and only one of those direct-to-goal transitions (the very last transition in the path). Based on that property, any arbitrary cost value would be valid for those transitions as long as all of them keep the same value. The cost value chosen for those transitions is +1, resulting in a graph where all costs are the same value (unweighted graph).

Graph Search

There is a wide range of graph search algorithms. Commonly, they are classified as informed search algorithms (i.e., Dijkstra [17] or A* [18]) or uninformed search algorithms (i.e., BFS [19] or DFS [20]). Uninformed search algorithms explore the state space in a blind manner, while informed search algorithms take advantage of additional information (such as the transition cost values) to guide and reduce the state-space exploration.

Given the characteristics of the current graph model, where all the costs have the same value, which is equivalent to say that there is no information at all to guide the search, an informed search algorithm does not provide any advantage over an uninformed search one. Moreover, under this specific scenario, an uninformed search implementation is easier and lighter, resulting in slightly better execution times. The BFS (uninformed search) algorithm was selected for this approach.

BFS is an algorithm used to explore a graph or a tree-like data structure. BFS guarantees to find an optimal solution if the graph is finite and deterministic [21]. The graph representing the system in this work meets those properties. Additionally, it is an unweighted graph, hence the computational cost might be further reduced since the algorithm can be stopped at the depth level where the first solution is found. Any solution found on a deeper depth level would imply a path with more actions involved.

It is worth noting that the common mental representation of a graph solution is the shortest path between two points, seen as a physical distance magnitude. Actually, the path between two nodes may involve multiple concepts other than physical distance. In this study, the shortest path between the node representing the initial state of the system and the node representing the goal state includes all kinds of trajectories carried out by every robot, the optimal assignment of the pieces, the order in which they are processed, etc.

The main language used for the implementation of this study was Python. Generally, a Python application is not as efficient as one written in a compiled language, such as C. On the other hand, Python supports using modules written in other languages, such as NumPy [22], a very optimized numerical library written in C. The Value Iteration algorithm described in Section 4.1 was implemented in Matrix form, taking advantage of the NumPy matrix operations. In order to enable a fair comparison between algorithms, BFS was implemented in C as well, and imported in Python as a module. Its pseudo-code can be seen in Algorithm 3.

Algorithm 3: Pseudo-code to implement Breadth First Search algorithm.

```
1:  BFS (G, s) // Where G is the graph, and s is the source node
2:      let Q be the FIFO queue.
3:      // Inserting s in queue
4:      Q.enqueue(s)
5:      mark s as visited
6:      // Loop until the queue is empty
7:      while (Q is not empty)
8:          // Remove the first node from the queue
9:          v = Q.dequeue
10:         // Process all its neighbors
11:         for all neighbors w of v in Graph G
12:             if w is not visited
13:                 // Insert w in Q
14:                     Q.enqueue(w)
15:             end if
16:         end for
17:     end while
```

4.3. Planning Domain Definition Language

Planning Domain Definition Language (PDDL) is an attempt to standardize Artificial Intelligence (AI) planning languages. PDDL has a variety of versions, such as Probabilistic PDDL (PPDDL) [23] to account for uncertainty, Multi-Agent PDDL (MA-PDDL) to allow planning for multiple agents, etc. In this study, the problem was designed as single-agent and deterministic, so the official PDDL language was used. PDDL models the problem in two separate files: the Domain file and the Problem file. The Domain file contains the predicates and actions; predicates are facts of interest, such as "is *i* robot in XYZ location?", "is there collision in the current state?", and "is *i* robot carrying the piece *j*?", and those predicates are evaluated as true or false; actions are the counterpart of the transition function in the MDP model; that is, the way the system state changes. The Problem file contains the initial state, the goal specification, and the objects; the initial state and goal specification are straightforwardly associated with the initial and goal state in the MDP model; the objects are the counterpart of the internal variables of the MDP model (robot location, robot status, and piece status).

Unlike MDP or graph search methods, this approach ends with the model problem definition. Then a well-proved software planner is used to obtain a solution. Some of the most used planners are FastDownward [24], LPG [25], and PDDL.jl [26]. In this project, FastDownward was used. FastDownward is a planner that relies on a heuristic search to address several planning problems, including PDDL. It is recognized as one of the most efficient planners currently available. Unlike other planners, FastDownward employs an alternative translation known as "multivalued planning tasks" instead of using the propositional representation of PDDL. This approach significantly enhances search efficiency.

5. Results

5.1. Validation by Comparison

RobotStudio® was used to validate the results obtained in this work. This simulator is based on Algoryx [27] and provides high precision in both kinematic and dynamic simulations. In this way, it is possible to simulate a robotic station with a high level of accuracy regarding movements and/or collisions between robots. Therefore, the conclusions reached using this software are considered to faithfully reflect the behavior of the real system. In this project, values and data generated by the simulator were accepted as valid.

5.2. Comparison of Results

To comparatively assess the results obtained from the proposed algorithms, the following data have been considered: the number of steps performed by the solution provided by each algorithm required off-line resources (RAM and disk storage) and on-line computational cost (associated with scalability and implementation feasibility in a production line).

The initial comparison among algorithms focuses on the number of steps or movements that algorithms need to perform to solve the task. This is the factor to be minimized as it determines how fast the task is performed. The number of pieces used in each industrial operation can significantly vary. Some assemblies only require a few pieces to obtain the intermediate or final product, while in other cases, hundreds of pieces are needed. This work is focused on SimplicityWorks Europe manufacturing process [5], where the company carries out pick-and-place operations without any specific order constraints. The goal is to minimize the task time by selecting the best sequence for placing between 4 and 9 pieces. Table 2 shows a comparison between solutions provided by several algorithms: Value Iteration (VI), Breadth First Search (BFS) and Planning Domain Definition Language (PDDL). Multiple tests for different scenarios (number of pieces and action mode) have been conducted. Initial conditions for each test are the same for every algorithm, including the source and destination location of the pieces and the initial location of the robots. The primary observation indicates that every algorithm reaches a solution with the same number of steps. This makes sense as all three algorithms are able to find an optimal solution to the problem. Those solutions may be different but with the same level of quality. The second point to note is that the PDDL algorithm scales better than the others, being the only one able to solve all tests. Finally, both VI and BFS solve the same test cases. This is because both of them rely on the MDP model, which does not scale well when the number of pieces and actions increases.

Table 2. Number of steps required to complete the pick-and-place task using two robotic arms.

NUMBER OF STEPS				
NAVIGATION MODE	**# OF PIECES**	**VI**	**BFS**	**PDDL**
1	2	28	28	28
	4	46	46	46
	6	55	55	55
	8	68	68	68
	9	77	77	77
	10	-	-	88
2	2	25	25	25
	4	37	37	37
	6	46	46	46
	8	56	56	56
	9	-	-	67
	10	-	-	75
3	2	25	25	25
	4	37	37	37
	6	46	46	46
	8	-	-	56
	9	-	-	67
	10	-	-	75
4	2	24	24	24
	4	35	35	35
	6	-	-	44
	8	-	-	56
	9	-	-	66
	10	-	-	73

The number of steps required for a set of robots to manipulate N pieces depends on their initial and final locations. This information is accessible just before the task starts. At that moment, the MDP must be updated (for all state-action pairs where the action is pick or drop), and after that, the solution for this particular MDP must be found. This whole process is carried out during the process, hence it is called "online computation" in this work. There is only a few seconds or minutes available in the best case scenario. Table 3 shows the time required for each algorithm to find a solution. The BFS algorithm is clearly the fastest whenever it can provide a solution. The comparison between BFS and PDDL is more intriguing. There is not a clear winner in all test scenarios. As a reference, VI is slower than PDDL in the test case "Mode 1 and 9 pieces" (5 min 57.5 s vs. 1 min 15.2 s), but it is faster in the test case "Mode 4 and 4 pieces" (6 min 27.4 s vs. 15 min 37.3 s).

Another important issue is the offline resources required to calculate the VI and BFS algorithms described in Tables 2 and 3. Both algorithms use the MDP architecture described in Section 4.1, which describes the number of states and their behavior according to the actions taken. The number of states of the MDP can be calculated according to Equation (9), where ξ_σ is the number of states and N is the number of pieces.

$$\xi_\sigma = ((M{\cdot}N{\cdot}Z) + (K{\cdot}P))^2{\cdot}(2+T)^K{\cdot}(K+1)^2 \tag{9}$$

where M, N, and Z are the number of rows, columns, and heights of the grid, K is the number of pieces to be picked, T is the number of possible intermediate positions, and P is the number of states required for the pick and drop operations. Intermediate positions (T) are useful in case a piece cannot be transported by any robot, due to accessibility problems, either on the source or destination location of the piece. In this case, the piece can be partially transported by a robot up to a certain intermediate position, and then another robot can complete the transportation. These kinds of problems are represented in Figure 3.

Table 3. Online computation time of each algorithm for a different number of pieces and different Navigation Modes.

ONLINE TIME				
NAVIGATION MODE	**# OF PIECES**	**VI**	**BFS**	**PDDL**
1	2	0.4 s	0.001 s	15 s
	4	4.6 s	0.016 s	16 s
	6	39 s	0.125 s	19 s
	8	4 min 37 s	0.973 s	36 s
	9	5 min 57 s	2.918 s	1 min 15 s
	10	-	-	2 min 56 s
2	2	0.8 s	0.016 s	39 s
	4	10 s	0.047 s	41 s
	6	1 min 26 s	0.267 s	47 s
	8	9 min 45 s	1.851 s	1 min 20 s
	9	-	-	2 min 38 s
	10	-	-	5 min 49 s
3	2	8.2 s	0.032 s	3 min 39 s
	4	1 min 26 s	0.376 s	4 min 02 s
	6	11 min 18 s	2.907 s	5 min 53 s
	8	-	-	16 min 08 s
	9	-	-	36 min 54 s
	10	-	-	1 h 29 min 24 s
4	2	37 s	0.110 s	14 min 30 s
	4	6 min 27 s	1.146 s	15 min 37 s
	6	-	-	22 min 19 s
	8	-	-	1 h 27 min 18 s
	9	-	-	2 h 14 min 46 s
	10	-	-	5 h 24 min 10 s

Table 4 shows the computational cost (time) and disk resources and memory required to build the MDP. This computation is performed only once for a specific problem topology (e.g., a certain grid size and number of pieces). This is particularly relevant because the MDP may take several hours to be calculated on a medium-power computer such as the one used in this project (e.g., 4 h 39 min and 42 s to calculate an MDP in Navigation Mode 4 with four pieces to be manipulated). In addition, the MDP size on disk exceeded 5GB in some test cases. Actually, this size grew as the numbers of pieces increased.

There are empty cells on some test cases for all the columns except for the BRUTE STATES because either the computer ran out of memory when solving the MDP or it could not be done in a reasonable amount of time. BRUTE STATES refer to the number of states in the state-space according to Equation (9). However, there is a considerable subset of those states that are not valid data (i.e., states where the robots collide, a robot is not able to reach a specific grid location). The MDP states are the number of actual states used in the MDP after filtering invalid states in the original BRUTE state-space. This information is provided by the software when building the MDP. The number of MDP states exponentially increases with the number of pieces, and the other variables (disk, RAM, and time) depends on this size, hence they exponentially increase as well.

Table 4. Offline computational cost (time), DISK memory used to store the state-transition table (GB), RAM memory necessary (MB), number of states of the MDP, and number of brute states of the MDP.

STATE-TRANSITION TABLE (MDP)						
NAVIGATION MODE (# ACTIONS)	**# PIECES**	**TIME**	**DISK (GB)**	**RAM (MB)**	**# MDP STATES**	**# BRUTE STATES**
1 (49)	2	12 s	0.004	8	13,568	104,976
	4	2 min 06 s	0.043	81	138,240	1,345,600
	6	15 min 57 s	0.328	619	1,052,672	12,054,784
	8	1 h 43 min 58 s	2.142	4.037	6,864,896	90,326,016
	9	4 h 28 min 04 s	5.265	9.923	16,875,520	236,748,800
	10	-	-	-	-	607,129,600
2 (121)	2	36 s	0.010	20	13,568	104,976
	4	6 min 04 s	0.103	201	138,240	1,345,600
	6	45 min 46 s	0.783	1.523	1,052,672	12,054,784
	8	4 h 49 min 30 s	5.107	9.968	6,864,896	90,326,016
	9	-	-	-	-	236,748,800
	10	-	-	-	-	607,129,600
3 (169)	2	05 min 42 s	0.090	28	13,568	853,776
	4	54 min 58 s	0.863	280	138,240	9,985,600
	6	6 h 47 min 04 s	6.383	2.135	1,052,672	82,301,184
	8	-	-	-	-	571,401,216
	9	-	-	-	-	1,445,068,800
	10	-	-	-	-	3,580,825,600
4 (841)	2	28 min 40 s	0.444	137	13,568	853,776
	4	4 h 39 min 42 s	4.235	1.395	138,240	9,985,600
	6	-	-	-	-	82,301,184
	8	-	-	-	-	571,401,216
	9	-	-	-	-	1,445,068,800
	10	-	-	-	-	3,580,825,600

Disk resource information required to store the MDP is directly obtained from the filesystem. The time and MDP states columns are provided by the software when building the MDP. The RAM required is computed based on the number of states, the number of actions, and the number of bytes needed to encode the MDP information. In this implementation, the state value is represented by a 32-bit value and the reward by a 16-bit value, making a total of 6 bytes per state-action pair. Additionally, during

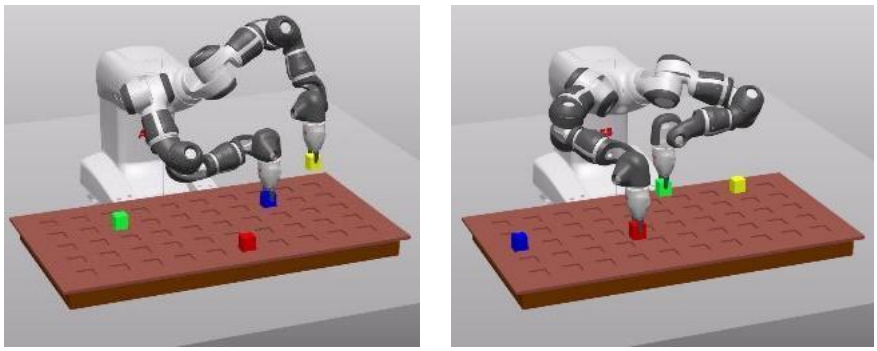
the Value Iteration algorithm execution, the implementation keeps a copy of the Value function in the previous time-step in order to be able to check if it converges.

The PC used in this research is an Intel Core i7-12700KF 12th Gen 3.60 Ghz, with 64 GB of RAM, 1TB SSD, a Nvidia Quadro P100 graphics card, and a Windows 11 Pro 22H2 operating system.

5.3. Data Validation

The results presented in this section show the behavior of the algorithms being compared (VI, BFS, and PDDL). For this purpose, RobotStudio® software was used. It provides a graphical representation of the setup, including a dual arm collaborative robot ABB IRB 14,000 (also called YuMi), multiple pieces, and a worktable. It also monitors the whole simulation, detecting any fault, such as collisions.

The first simulation involves a YuMi robot only supporting orthogonal movements in a horizontal plane (except for the pick and place operations). This corresponds to Navigation Mode 1 (see Section 3.3.2). Figure 9 shows two frames of the simulation, and a demonstrative video is available on https://www.youtube.com/watch?v=3ysBDbBr5mQ (accessed on 6 June 2023). This approach offers a collision-free optimized solution to the manipulation problem, as the robot is capable of completing the task in 46 steps. It is worth noting that the robot's speed for each movement is configured to take approximately 0.5 s per transition (time to move from one state to the next one). This means that the entire task is completed in 23 s.

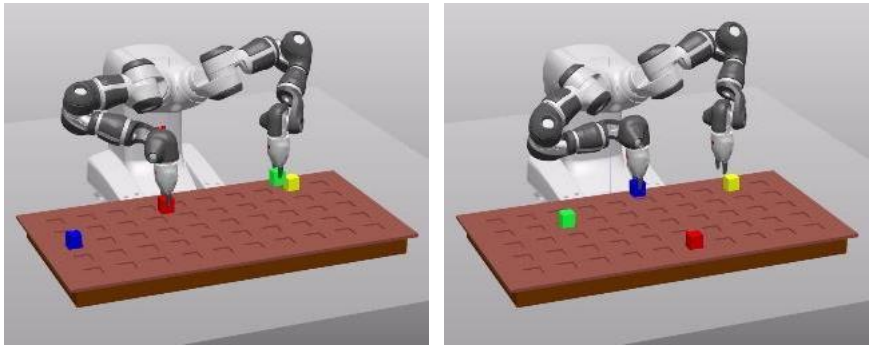


(**a**) (**b**)

Figure 9. Simulation of a pick-and-place operation with Mode 1 movements. (**a**) t = 9 s; (**b**) t = 22 s.

Figure 10 shows another simulation of the same pick-and-place process. In this case, the robot supports both orthogonal and diagonal movements in the horizontal plane (Navigation Mode 2 described in Section 3.3.2). By giving the system the ability to diagonally move, the robot can combine two movements into one and reduce the number of steps. The simulation can be viewed on https://www.youtube.com/watch?v=3ysBDbBr5mQ (accessed on 6 June 2023). The number of steps is lower than in the previous case, even though the initial and final positions of pieces are the same. In this case, the task is completed in 37 steps, taking 18.5 s. It confirms that with a more flexible combination of movements, the algorithm finds a better solution to complete the task.

A third simulation is presented, where the robot supports moving through a 3D virtual grid, enabling the arm's intersection without collision by operating on different planes. This configuration corresponds to Navigation Mode 4 described in Section 3.3.2. The objective of this simulation is to check if the algorithm finds a solution with a lower number of steps. The full video is available at https://www.youtube.com/watch?v=kqKprKYazKk (accessed on 6 June 2023). In this case, the task is completed in 35 steps, taking 17.5 s for the robot to place all pieces in their final positions. Once again, with a larger set of movements, the algorithm finds a better solution for the task than in previous cases. On this occasion, arms have the ability to cross each other to minimize the operation time, avoiding collisions.

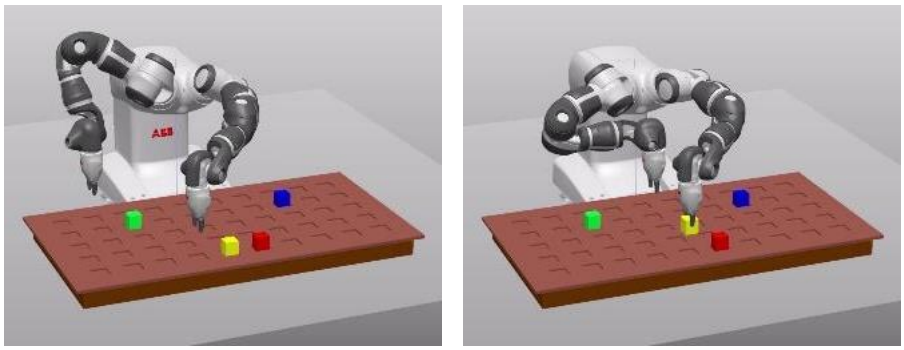


(**a**) (**b**)

Figure 10. Simulation of a pick-and-place operation with Navigation Mode 2 movement. (**a**) t = 10 s; (**b**) t = 19 s.

Figure 11 shows a set of frames extracted from this simulation, representing a couple of situations where the algorithm takes advantage of both arms moving on different planes. The first situation, shown in frames (a), (b) and (c) of Figure 11 involves the left arm picking up and placing the first piece (yellow), and the right arm picking up the second piece (blue). In this sequence, the left arm transitions to a higher plane, allowing the right arm to approximate the blue piece without interrupting the left arm trajectory. In the second situation, shown in frames (c), (d) and (e) of Figure 11, the algorithm finds optimal to move the right arm to a higher plane in order to maintain its course without collision. At t = 14 s (Figure 11d), the right arm is about to pick up the blue piece and the left arm is approximating the green piece. After that, the right arm starts moving towards the red piece. At t = 16 s (Figure 11e), the right arm is moving at a higher plane than the left arm to avoid a collision. After that, both arms progress in parallel in the same direction. At t = 20 s (Figure 11f), the left arm is still transporting the green piece while the right arm has already picked up the red piece and is moving toward its destination.

Finally, a new simulation is presented. The robot still supports moving through the virtual 3D grid, but the number of pieces is increased to 10. In this case, the task is successfully performed in 73 steps and a total time of 36.5 s. Note that in frames (c), (f), and (h) of Figure 12, both arms are crossing, which is possible because the algorithm places them at different heights. As seen in Table 2, only the PDDL algorithm is capable of finding a solution for such a high number of pieces, scaling the problem much better than the VI and BFS options to solve the proposed MDP.



(**a**) (**b**)

Figure 11. *Cont.*

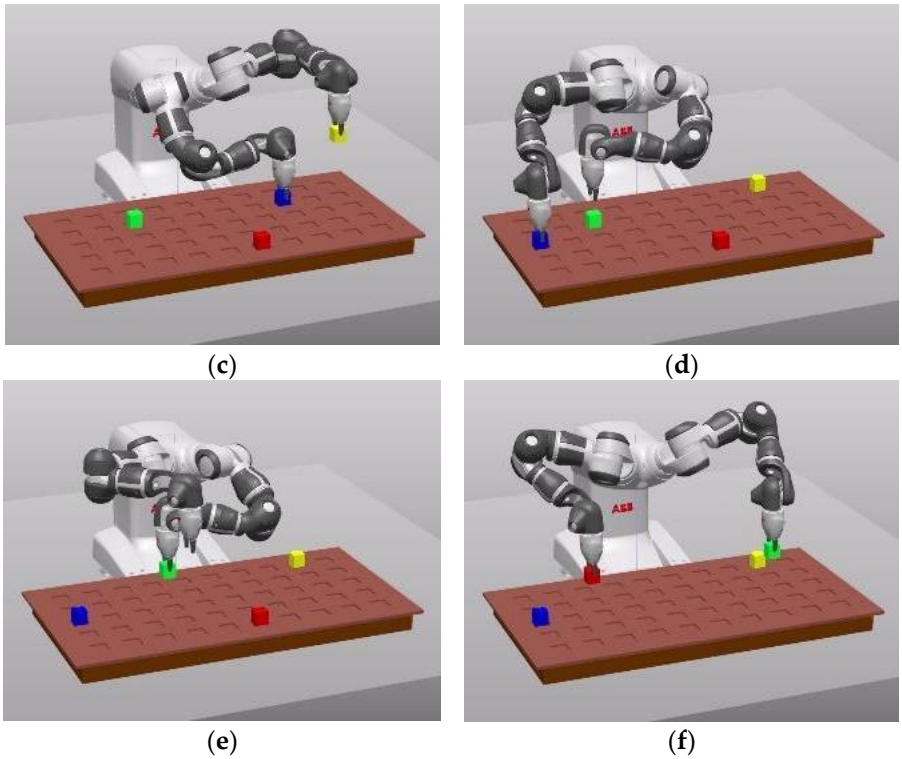


Figure 11. Simulation of a pick-and-place operation with Navigation Mode 4 movements (movements in 3 different heights). (**a**) t = 3 s; (**b**) t = 6 s; (**c**) t = 8 s; (**d**) t = 14 s; (**e**) t = 16 s; (**f**) t = 20 s.

A comprehensive collection of simulations, classified per algorithm, can be found at the following links: Value Iteration (https://www.youtube.com/watch?v=cXEfuw8WPqA, accessed on 8 November 2023), BFS (https://www.youtube.com/watch?v=2T0z_3_9az8, accessed on 8 November 2023), and PDDL (https://www.youtube.com/watch?v=vV5W80SKIOo, accessed on 8 November 2023). Additionally, at https://www.youtube.com/watch?v=jl47uA9LN_w (accessed on 8 November 2023) is presented a test case where both arms must cooperate to transport one of the pieces. In this case, one of the arms can only access the initial location of the piece, and the other one can only access its destination location. The first arm leaves the piece in an intermediate position so that the second one can finish the subtask. In this case, the approach proposed can solve the problem presented in Figure 3, where two or more robots should coordinately collaborate between them and in the minimum time.

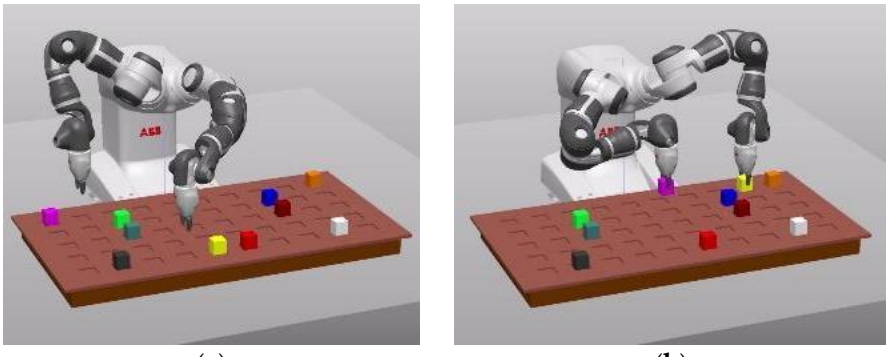


Figure 12. *Cont.*

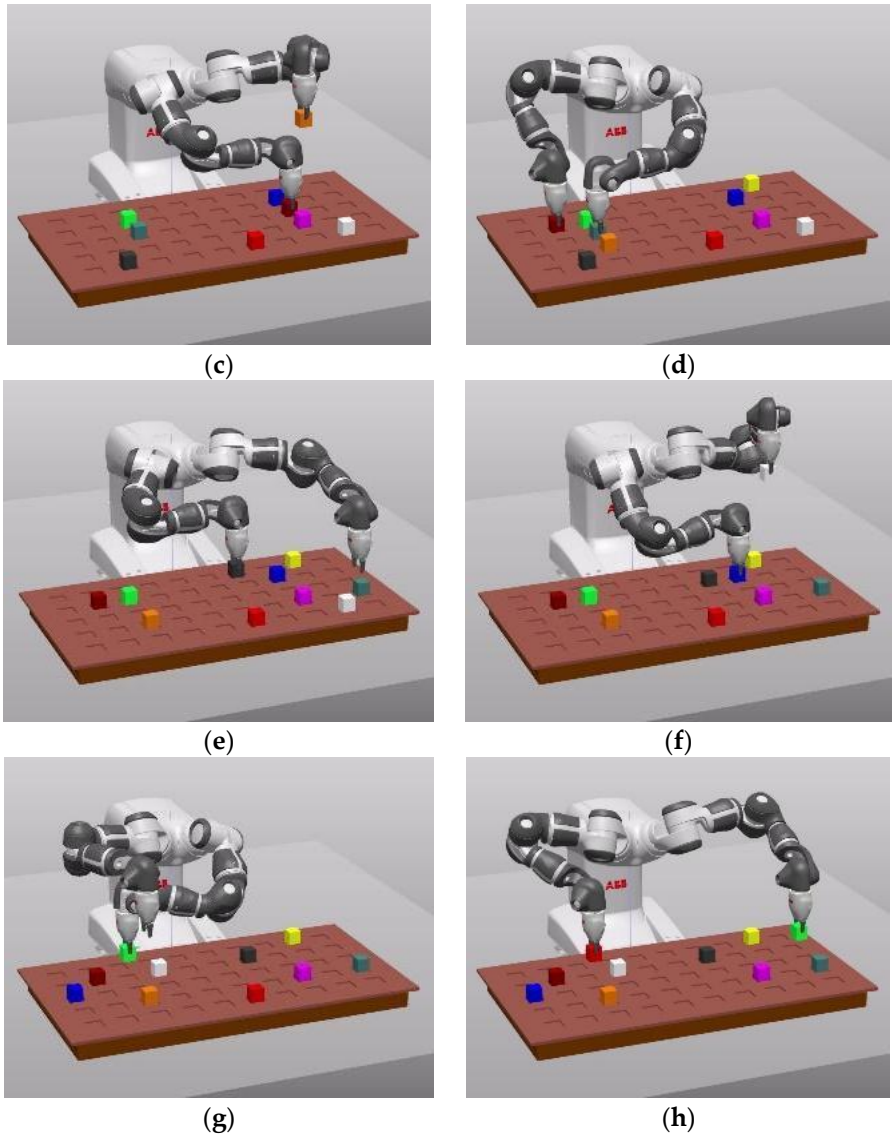


Figure 12. Simulation of a pick-and-place operation solved with the PDDL algorithm for 10 pieces. (**a**) t = 3 s; (**b**) t = 8 s; (**c**) t = 12 s; (**d**) t = 18 s; (**e**) t = 24 s; (**f**) t = 28 s; (**g**) t = 35 s; (**h**) t = 40 s.

6. Conclusions and Future Work

This article presents a method to coordinate robotic manipulators, avoiding collisions between arms during pick-and-place operations, and minimizing the task time by selecting the best manipulation sequence and the best trajectory planning. Due to the assumed simplifications in the system (deterministic, single-agent, and fully observable), the Markov Decision Process (MDP) proposed can be transformed into a graph that can be solved using a search algorithm, such as Breadth First Search (BFS). Additionally, the Planning Domain Definition Language (PDDL) has been used to generate an alternative solution that can be compared with VI and BFS.

For every test case where a solution can be obtained, all three algorithms consistently produce an optimal solution, which is not unique and may differ between all algorithms. In this context, optimal refers to the least number of steps required to complete the task. Regarding the computational cost to find a solution, the graph search method (BFS) turns out to be the fastest one, although limited in terms of scalability. In fact, the generated graph is derived from the MDP model, so actually both MDP and BFS approaches suffer this same scalability issue, directly related to the number of pieces and robots. On the other hand, the PDDL scales better and is able to solve all tested cases. The MDP approach

is neither the fastest nor the least eager on resources, but provides a general framework capable of scaling to more complex problems, including uncertainty, the continuous state and action spaces, the partially observable state, etc. The last factor, irrespective of the algorithm used, is the flexibility of the robot's navigation. A significant enhancement of the solution is observed when diagonal movements are incorporated into the 2D plane (as outlined in Mode 2 in Section 3.3.2). While the inclusion of extra movements for 3D space navigation results in a marginal improvement in the solution, it comes at a substantial expense due to the computational cost being exponential to the number of robots (in this case, quadratic, as there are two robots).

However, the proposed method has several disadvantages. Firstly, the system does not scale well. The provided solution is feasible for certain production processes where the number of pieces to manipulate is small (less than 10 elements). Secondly, the movements that the robots must perform are constrained within a grid, which is less natural compared to other solutions such as [15], although it provides a robust and faster industrial solution to the problem. Lastly, a preliminary study using RobotStudio® is required to create a database, including information about collision situations between arms.

In future work, the intention is to explore new techniques, such as hierarchical reinforcement learning or multi-agent learning, that can overcome the scalability limitation of the current single-agent approach. The goal is to introduce a larger number of pieces and robots that could operate within the same workspace. Another objective is to investigate alternative implementation strategies, such as using specialized HW (GPU), since the algorithm (VI) core is heavily based on matrix operations. Additionally, the preliminary study of robot collisions demands considerable time and complexity. Therefore, an alternative method is being considered. A possible solution is to frame the robot links within a set of ellipsoids and check for collisions between these ellipsoids in space. This approach would eliminate the need to rely on a graphical environment such as RobotStudio®, reducing offline computational costs but potentially increasing online computational requirements. Hence, an analysis of the performance of this new approach would be required.

Author Contributions: Conceptualization, D.M.-G., F.J.M.-P. and C.P.-V.; methodology, D.M.-G. and C.P.-V.; software, D.M.-G. and F.J.M.-P.; validation, D.M.-G. and F.J.M.-P.; formal analysis, C.P.-V.; investigation, D.M.-G. and F.J.M.-P.; resources, D.M.-G. and F.J.M.-P.; writing—original draft preparation, D.M.-G., F.J.M.-P. and C.P.-V.; writing—review and editing, D.M.-G., F.J.M.-P. and C.P.-V.; visualization, F.J.M.-P.; supervision, C.P.-V.; project administration, C.P.-V.; funding acquisition, C.P.-V.; All authors have read and agreed to the published version of the manuscript.

Funding: This research has been partly funded by project CPP2021-008593, grant MCIN/AEI/10.13039/501100011033 and by the European Union-NextGenerationEU/PRTR.

Data Availability Statement: All files created can be found in the following Git repository: https://github.com/da-mago/Two-arms-pick-place-planning accessed on 8 November 2023.

Acknowledgments: This project would not have been possible without the help of ABB Spain, which has provided software licenses and hardware.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Hermann, J.; David, A.; Wagner, A.; Ruskowski, M. Considering interdependencies for a dynamic generation of process chains for production as a service. *Procedia Manuf.* **2020**, *51*, 1454–1461. [CrossRef]
2. Ruskowski, M.; Herget, A.; Hermann, J.; Motsch, W.; Pahlevannejad, P.; Sidorenko, A.; Bergweiler, S.; David, A.; Plociennik, C.; Popper, J.; et al. Production Bots fur Production Level Skill-basierte Systeme fur die Produktion der Zukunft. *Atp Mag.* **2020**, *62*, 62–71. [CrossRef]
3. Wrede, S.; Beyer, O.; Dreyer, C.; Wojtynek, M.; Steil, J. Vertical integration and service orchestration for modular production systems using business process models. *Procedia Technol.* **2016**, *26*, 259–266. [CrossRef]
4. Sellers, C.J.; Nof, S.Y. Performance analysis of robotic kitting systems. *Robot. Comput.-Integr. Manuf.* **1989**, *6*, 15–24. [CrossRef]
5. Méndez, J.B.; Perez-Vidal, C.; Heras, J.V.S.; Pérez-Hernández, J.J. Robotic Pick-and-Place Time Optimization: Application to Footwear Production. *IEEE Access* **2020**, *8*, 209428–209440. [CrossRef]
6. He, T.; Wang, H.; Yoon, S.W. Comparison of Four Population-Based Meta-Heuristic Algorithms on Pick-and-Place Optimization. *Procedia Manuf.* **2018**, *17*, 944–951. [CrossRef]
7. Ye, X.; Zhang, Y.; Ru, C.; Luo, J.; Xie, S.; Sun, Y. Automated Pick-Place of Silicon Nanowires. *IEEE Trans. Autom. Sci. Eng.* **2013**, *10*, 554–561. [CrossRef]
8. Gao, J.; Zhu, X.; Liu, A.; Meng, Q.; Zhang, R. An Iterated Hybrid Local Search Algorithm for Pick-and-Place Sequence Optimization. *Symmetry* **2018**, *10*, 633. [CrossRef]
9. Kennedy, J.; Eberhart, R. Particle swarm optimization. In Proceedings of the ICNN'95—International Conference on Neural Networks, Perth, WA, Australia, 27 November–1 December 1995; Volume 4, pp. 1942–1948. [CrossRef]
10. Kumar, M.; Husian, M.; Upreti, N.; Gupta, D. Genetic Algorithm: Review and Application. *Int. J. Inf. Technol. Knowl. Manag.* **2010**, *2*, 451–454. [CrossRef]
11. Dorigo, M.; Birattari, M.; Stutzle, T. Ant colony optimization. *IEEE Comput. Intell. Mag.* **2006**, *1*, 28–39. [CrossRef]
12. Alazzam, A.R. Using BUA algorithm to solve a sequential pick and place problem. In Proceedings of the 2018 International Conference on Information and Computer Technologies (ICICT), DeKalb, IL, USA, 23–25 March 2018; pp. 144–149. [CrossRef]
13. Daoud, S.; Chehade, H.; Yalaoui, F.; Amodeo, L. Efficient metaheuristics for pick and place robotic systems optimization. *J. Intell. Manuf.* **2014**, *25*, 27–41. [CrossRef]
14. Gafur, N.; Kanagalingam, G.; Ruskowski, M. Dynamic collision avoidance for multiple robotic manipulators based on a non-cooperative multi-agent game. *arXiv* **2021**. [CrossRef]
15. Gafur, N.; Kanagalingam, G.; Wagner, A.; Ruskowski, M. Dynamic Collision and Deadlock Avoidance for Multiple Robotic Manipulators. *IEEE Access* **2022**, *10*, 55766–55781. [CrossRef]
16. AlMahamid, F.; Grolinger, K. Reinforcement Learning Algorithms: An Overview and Classification. In Proceedings of the 2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), Virtually, 12–17 September 2021; pp. 1–7. [CrossRef]
17. Javaid, A. Understanding Dijkstra's algorithm. *SSRN Electron. J.* **2013**, *10*, 1–27. [CrossRef]
18. Foead, D.; Ghifari, A.; Kusuma, M.B.; Fiah, N.H.; Gunuwan, E. A systematic literature Review of A* PathFinding. In Proceedings of the 5th International Conference on Computer Science and Computational Intelligence 2020, Online, 19–20 November 2020. [CrossRef]
19. Gao, P.; Liu, Z.; Wu, Z.; Wang, D. A Global Path Planning Algorithm for Robots Using Reinforcement Learning. In Proceedings of the 2019 IEEE International Conference on Robotics and Biomimetics (ROBIO), Dali, China, 6–8 December 2019. [CrossRef]
20. Paulino, L.; Hannum, C.; Varde, A.S. Search Methods in Motion Planning for Mobile Robots. In Proceedings of the Intelligent Systems and Applications: Proceedings of the 2021 Intelligent Systems Conference (IntelliSys), Amsterdam, The Netherlands, 2–3 September 2021. [CrossRef]
21. Sadik, A.M.J.; Dhali, M.A.; Farid, H.M.A.B.; Rashid, T.U.; Syeed, A. A Comprehensive and Comparative Study of Maze-Solving Techniques by Implementing Graph Theory. In Proceedings of the 2010 International Conference on Artificial Intelligence and Computational Intelligence, Sanya, China, 23–24 October 2010; pp. 52–56. [CrossRef]
22. Shen, G.; Liu, Q. *Performance Analysis of Linear Regression Based on Python*; En Cognitive Cities, IC3 2019, CCIS 1227; Springer Nature: Singapore, 2020; pp. 695–702. [CrossRef]
23. Younes, H.L.S.; Littman, M.L. PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162* **2004**, *2*, 99.
24. Helmert, M. The fast downward planning system. *J. Artif. Intell. Res.* **2006**, *26*, 191–246. [CrossRef]
25. Gerevini, A.; Saetti, A.; Serin, I. Planning through stochastic local search and temporal action graphs in LPG. *J. Artif. Intell. Res.* **2003**, *20*, 239–290. [CrossRef]

26. Tan, Z.-X. PDDL.jl: An Extensible Interpreter and Compiler Interface for Fast and Flexible AI Planning. Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2022.
27. Backman, A. Algoryx—interactive physics. In Proceedings of the SIGRAD 2008, the Annual SIGRAD Conference Special Theme: Interaction, Stockholm, Sweden, 27–28 November 2008; Linköping University Electronic Press: Linköping, Sweden, 2008; p. 87.