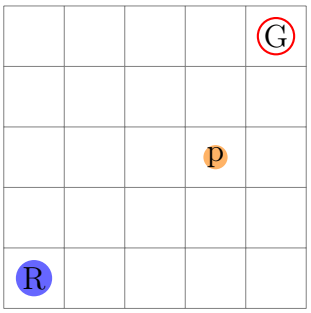
AI Planning

Project 1 - Automated Warehouse Management

Robotic Planning System with Mobile Robot



Halim Djerroud
LISV - Université Paris-Saclay

Contents

1 Introduction **3**
1.1 Project Context . . . 3
1.2 System Overview . . . 3
1.3 Project Scenario . . . 3
1.4 Practical Details . . . 3

2 Part 1: Simulation Environment (40 points) **4**
2.1 Overview . . . 4
2.2 Functional Specifications . . . 4
2.2.1 Environment . . . 4
2.2.2 Robots . . . 4
2.2.3 Packages . . . 5
2.3 Required Graphical Interface . . . 5
2.3.1 Mandatory Visual Elements . . . 5
2.4 Python Code Architecture . . . 6
2.5 Example Code: Robot Class . . . 6
2.6 Initial State Extraction . . . 7
2.7 Evaluation Criteria - Part 1 . . . 9

3 Part 2: PDDL Modeling and Planning (60 points) **10**
3.1 Overview . . . 10
3.2 PDDL Domain Modeling . . . 10
3.2.1 Types to Define . . . 10
3.2.2 Required Predicates . . . 10
3.2.3 Actions to Implement . . . 10
3.3 Complete PDDL Domain Example . . . 12
3.4 Fast Downward Integration . . . 13
3.4.1 Installation . . . 13
3.4.2 Calling the Planner from Python . . . 13
3.5 Plan Parsing . . . 14
3.6 Plan Execution in Simulation . . . 16
3.7 Integration into Graphical Interface . . . 19
3.8 Evaluation Criteria - Part 2 . . . 21

4 Deliverables and Evaluation **22**
4.1 Expected Deliverables . . . 22
4.2 Submission Format . . . 22
4.3 Overall Grading . . . 22
4.4 Presentation (optional) . . . 22

5 Advice and Possible Extensions **23**
5.1 Tips for Success . . . 23
5.2 Possible Extensions (bonus) . . . 23
5.3 Useful Resources . . . 24
5.4 FAQ . . . 24

6 Appendices **26**
6.1 Example requirements.txt File 26
6.2 Example README.md 26
6.3 Report Template 27

1 Introduction

1.1 Project Context

This project constitutes the final evaluation of the first part of the Automated Planning course. It allows you to put into practice all the knowledge acquired about PDDL (Planning Domain Description Language) in a concrete robotics application context.

Learning Objectives By the end of this project, you should be able to:

- Model a robotic problem using PDDL
- Develop an interactive Python application with graphical interface
- Integrate an external planner (Fast Downward)
- Automatically extract a system's state to generate a PDDL problem
- Interpret and execute a plan generated by a planner
- Analyze a planner's performance

1.2 System Overview

You will develop a **complete robotic planning system** composed of two complementary parts:

Part 1: Simulation Environment (Python + Graphical Interface)

Part 2: Planning Module (PDDL + Fast Downward)

1.3 Project Scenario

Scenario: Automated Warehouse One or more mobile robots operate in a warehouse organized as a grid. Their mission is to collect packages located in different zones and deliver them to their respective destinations. The robots have limited carrying capacity and must plan their movements optimally.

1.4 Practical Details

- **Work:** In pairs / groups of three
- **Duration:** 3 weeks (sessions 11 to 15)
- **Weight:** 50% of final grade
- **Languages:** Python 3.x + PDDL
- **Planner:** Fast Downward (provided)

Submission Deadline **Sunday, December 15 at 11:59 PM** Failure to meet this deadline will result in a penalty of 2 points per day late.

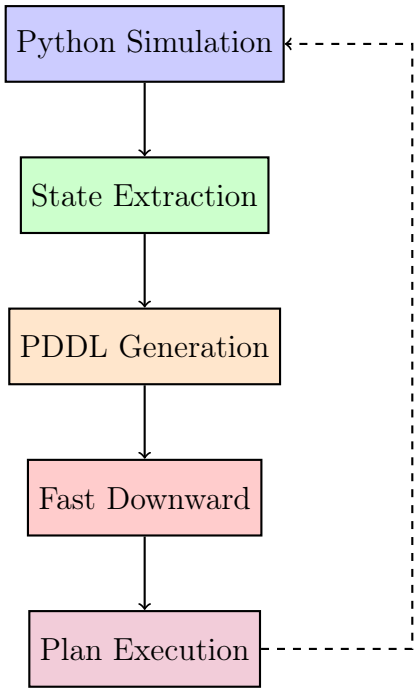


Figure 1: Overall system architecture

2 Part 1: Simulation Environment (40 points)

2.1 Overview

This first part involves developing a Python application with a graphical interface to simulate a robotic environment. This simulation will serve as the basis for testing plans generated by the PDDL planner.

2.2 Functional Specifications

2.2.1 Environment

- **Type:** 2D grid of size N×N (minimum 5×5, recommended 7×7)
- **Cells:** Each cell can be free or contain an obstacle
- **Representation:** Coordinates (x, y) with x, y ∈ [0, N-1]

2.2.2 Robots

You must implement **at least 1 robot**, ideally 2 or 3 robots:

- Initial position in the grid
- Transport capacity: 1 to 2 packages maximum
- Possible actions: move (up, down, left, right), load a package, unload a package
- Internal state: current position, list of carried packages

2.2.3 Packages

- **Number:** Minimum 4 packages
- **Initial position:** Location in the grid
- **Destination:** Target delivery zone
- **State:** Waiting / Transported / Delivered

2.3 Required Graphical Interface

The interface must be developed with one of the following libraries:

- `pygame` (recommended for smooth animation)
- `tkinter` (simpler but less visual)
- `matplotlib` with animation

2.3.1 Mandatory Visual Elements

1. **Environment Grid** with:
   - Free cells (white or light gray)
   - Obstacles (black or dark gray)
   - Robots (icon or distinctive color)
   - Packages (symbol or color)
   - Destinations (visual marker)
2. **Control Panel** with buttons:
   - "Extract Initial State" → generates `problem.pddl` file
   - "Define Goal" → specify target configuration
   - "Plan" → calls Fast Downward
   - "Execute Plan" → animates the obtained plan
   - "Reset" → returns the environment to initial state
3. **Information Area** displaying:
   - Current system state
   - Number of packages delivered / total
   - Last action executed
   - Any error messages

2.4 Python Code Architecture

Your code must be structured in separate modules:

```
project/
|-- main.py               # Entry point
|-- environment.py        # Environment class (grid)
|-- robot.py             # Robot class
|-- package.py           # Package class
|-- gui.py               # Graphical interface
|-- pddl_generator.py    # PDDL file generation
|-- planner_interface.py # Fast Downward call
|-- plan_executor.py     # Plan execution and animation
|-- domain.pddl          # PDDL domain (Part 2)
|-- problem.pddl         # Generated automatically
```

2.5 Example Code: Robot Class

```python
class Robot:
    def __init__(self, robot_id, position, capacity=1):
        """
        Initialize a robot

        Args:
            robot_id (str): Unique robot identifier
            position (tuple): Initial position (x, y)
            capacity (int): Maximum package capacity
        """
        self.id = robot_id
        self.position = position
        self.capacity = capacity
        self.carrying = []  # List of carried packages

    def move(self, direction):
        """
        Move the robot in a direction

        Args:
            direction (str): 'up', 'down', 'left', 'right'

        Returns:
            bool: True if the move is valid
        """
        x, y = self.position
        moves = {
            'up': (x, y+1),
            'down': (x, y-1),
            'left': (x-1, y),
            'right': (x+1, y)
```

```
        }
        new_pos = moves.get(direction)

        # Check validity (to implement)
        # ...

        self.position = new_pos
        return True

    def pickup(self, package):
        """Load a package"""
        if len(self.carrying) < self.capacity:
            self.carrying.append(package)
            return True
        return False

    def drop(self, package):
        """Unload a package"""
        if package in self.carrying:
            self.carrying.remove(package)
            return True
        return False

    def is_at(self, position):
        """Check if robot is at a given position"""
        return self.position == position

    def can_carry_more(self):
        """Check if robot can carry more packages"""
        return len(self.carrying) < self.capacity
```

Usage Example:

```
# Create a robot
robot1 = Robot("robot1", position=(0, 0), capacity=2)

# Move the robot
robot1.move('right')
print(f"Position: {robot1.position}")  # (1, 0)

# Load a package
package1 = Package("pkg1", position=(1, 0), destination=(5, 5))
robot1.pickup(package1)
print(f"Robot carries: {len(robot1.carrying)} packages")
```

2.6 Initial State Extraction

You must implement a function that analyzes the current state of your simulation and automatically generates the `problem.pddl` file.

```python
def extract_state_to_pddl(environment, robots, packages, output_file):
    """
    Extract current state and generate problem.pddl

    Args:
        environment: Environment object (grid)
        robots: List of robots
        packages: List of packages
        output_file: Output file path
    """
    with open(output_file, 'w') as f:
        f.write("(define (problem warehouse-delivery)\n")
        f.write("  (:domain warehouse)\n\n")

        # 1. Declare objects
        f.write("  (:objects\n")

        # Robots
        for robot in robots:
            f.write(f"    {robot.id} - robot\n")

        # Packages
        for pkg in packages:
            f.write(f"    {pkg.id} - package\n")

        # Zones (grid positions)
        for x in range(environment.width):
            for y in range(environment.height):
                if not environment.is_obstacle(x, y):
                    f.write(f"    zone_{x}_{y} - location\n")

        f.write("  )\n\n")

        # 2. Initial state
        f.write("  (:init\n")

        # Robot positions
        for robot in robots:
            x, y = robot.position
            f.write(f"    (at-robot {robot.id} zone_{x}_{y})\n")
            f.write(f"    (robot-free {robot.id})\n")

        # Package positions
        for pkg in packages:
            if pkg.is_carried:
                f.write(f"    (in-robot {pkg.id} {pkg.carrier_id})\n")
            else:
                x, y = pkg.position
                f.write(f"    (at-package {pkg.id} zone_{x}_{y})\n")

```

```
        # Connections between adjacent zones
        for x in range(environment.width):
            for y in range(environment.height):
                if not environment.is_obstacle(x, y):
                    # Neighbors
                    for dx, dy in [(0,1), (0,-1), (1,0), (-1,0)]:
                        nx, ny = x + dx, y + dy
                        if environment.is_valid_position(nx, ny):
                            f.write(f"    (connected zone_{x}_{y} ")
                            f.write(f"zone_{nx}_{ny})\n")

        f.write("  )\n\n")

        # 3. Goal (objective)
        f.write("  (:goal\n    (and\n")
        for pkg in packages:
            dest_x, dest_y = pkg.destination
            f.write(f"      (at-package {pkg.id} ")
            f.write(f"zone_{dest_x}_{dest_y})\n")
        f.write("    )\n  )\n")

        f.write(")\n")
```

Usage Example:

```
# Create the environment
env = Environment(width=5, height=5)
robots = [Robot("robot1", (0, 0))]
packages = [Package("pkg1", (2, 2), destination=(4, 4))]

# Extract state to PDDL
extract_state_to_pddl(env, robots, packages, "problem.pddl")
print("File problem.pddl generated successfully!")
```

2.7 Evaluation Criteria - Part 1

Criterion	Points
Python code quality (structure, comments)	8
Functional and clear graphical interface	10
Visual representation of environment	6
Class implementation (Robot, Package, etc.)	8
State extraction to PDDL function	8
Total Part 1	**40**

Table 1: Part 1 Grading

3 Part 2: PDDL Modeling and Planning (60 points)

3.1 Overview

This second part involves modeling the robotic problem in PDDL, integrating the Fast Downward planner, and interpreting the generated plans to execute them in the simulation.

3.2 PDDL Domain Modeling

You must create a complete and correct `domain.pddl` file.

3.2.1 Types to Define

```
(:types
    robot package location - object
)
```

3.2.2 Required Predicates

Here are the minimum predicates to implement:

```
(:predicates
    (at-robot ?r - robot ?l - location)
    (at-package ?p - package ?l - location)
    (in-robot ?p - package ?r - robot)
    (robot-free ?r - robot)
    (connected ?l1 - location ?l2 - location)
)
```

Modeling Advice Think carefully about your predicates! They must capture all the information necessary to describe the system state. Ask yourself:

- Where are the robots?
- Where are the packages?
- Which packages are being transported and by whom?
- Which zones are connected?
- Does the robot have available capacity?

3.2.3 Actions to Implement

You must define the following actions:

Action 1: MOVE (Move a robot)

```
(:action move
    :parameters (?r - robot ?from - location ?to - location)
    :precondition (and
        ; Robot is at the starting position
        ; Both positions are connected
        ; Add other preconditions if necessary
    )
    :effect (and
        ; Robot is no longer at from
        ; Robot is now at to
    )
)
```

Action 2: PICKUP (Load a package)

```
(:action pickup
    :parameters (?r - robot ?p - package ?l - location)
    :precondition (and
        ; Robot is at position l
        ; Package is at position l
        ; Robot is free (capacity available)
    )
    :effect (and
        ; Package is no longer at position l
        ; Package is in the robot
        ; Robot is no longer free
    )
)
```

Action 3: DROP (Unload a package)

```
(:action drop
    :parameters (?r - robot ?p - package ?l - location)
    :precondition (and
        ; Robot is at position l
        ; Package is in the robot
    )
    :effect (and
        ; Package is no longer in the robot
        ; Package is at position l
        ; Robot is free
    )
)
```

Watch for Consistency! Your PDDL actions must correspond EXACTLY to your Python class methods. For example, if your PDDL action is `(move robot1 zone_0_0 zone_0_1)`, your Python code must be able to interpret this action and call `robot1.move('right')`.

3.3 Complete PDDL Domain Example

```
(define (domain warehouse)
    (:requirements :strips :typing)

    (:types
        robot package location - object
    )

    (:predicates
        (at-robot ?r - robot ?l - location)
        (at-package ?p - package ?l - location)
        (in-robot ?p - package ?r - robot)
        (robot-free ?r - robot)
        (connected ?l1 - location ?l2 - location)
    )

    (:action move
        :parameters (?r - robot ?from - location ?to - location)
        :precondition (and
            (at-robot ?r ?from)
            (connected ?from ?to)
        )
        :effect (and
            (not (at-robot ?r ?from))
            (at-robot ?r ?to)
        )
    )

    (:action pickup
        :parameters (?r - robot ?p - package ?l - location)
        :precondition (and
            (at-robot ?r ?l)
            (at-package ?p ?l)
            (robot-free ?r)
        )
        :effect (and
            (not (at-package ?p ?l))
            (in-robot ?p ?r)
            (not (robot-free ?r))
        )
    )

    (:action drop
        :parameters (?r - robot ?p - package ?l - location)
        :precondition (and
```

```
            (at-robot ?r ?l)
            (in-robot ?p ?r)
        )
        :effect (and
            (not (in-robot ?p ?r))
            (at-package ?p ?l)
            (robot-free ?r)
        )
    )
)
```

3.4 Fast Downward Integration

3.4.1 Installation

Fast Downward is already installed on lab machines. To install on your personal machine:

```
# Download from official website
git clone https://github.com/aibasel/downward.git
cd downward
./build.py

# Or use the provided pre-compiled version if using Windows
```

3.4.2 Calling the Planner from Python

```
import subprocess
import os

def call_planner(domain_file, problem_file, output_file="plan.txt"):
    """
    Call Fast Downward to solve the problem

    Args:
        domain_file: Path to domain.pddl
        problem_file: Path to problem.pddl
        output_file: Plan output file

    Returns:
        bool: True if a plan was found
    """
    try:
        # Fast Downward command with A* search
        cmd = [
            "python3",
            "fast-downward.py",
            domain_file,
            problem_file,
            "--search",
```

```
            "astar(lmcut())"  # LM-Cut heuristic
        ]

        # Execute the planner
        result = subprocess.run(
            cmd,
            capture_output=True,
            text=True,
            timeout=60  # 60 second timeout
        )

        # Check if a plan was found
        if "Solution found" in result.stdout:
            print("Plan found successfully!")

            # The plan is in the sas_plan file
            if os.path.exists("sas_plan"):
                os.rename("sas_plan", output_file)
                return True
        else:
            print("No plan found.")
            print(result.stdout)
            return False

    except subprocess.TimeoutExpired:
        print("Timeout: planner took too long.")
        return False
    except Exception as e:
        print(f"Error calling planner: {e}")
        return False
```

Usage Example:

```
# Call the planner
success = call_planner("domain.pddl", "problem.pddl", "solution.txt")

if success:
    print("Plan generated in solution.txt")
else:
    print("Planning failed")
```

3.5 Plan Parsing

The plan returned by Fast Downward is a text file with one action per line. You must parse this file to extract the actions.

```
import re

```

```python
def parse_plan(plan_file):
    """
    Parse the plan file generated by Fast Downward

    Args:
        plan_file: Path to the plan file

    Returns:
        list: List of actions [(action, params), ...]
    """
    actions = []

    with open(plan_file, 'r') as f:
        for line in f:
            line = line.strip()

            # Ignore comment lines and empty lines
            if not line or line.startswith(';'):
                continue

            # Typical format: "(move robot1 zone_0_0 zone_0_1)"
            # or: "move robot1 zone_0_0 zone_0_1"

            # Remove parentheses if present
            line = line.strip('()')

            # Separate action and parameters
            parts = line.split()

            if len(parts) > 0:
                action_name = parts[0]
                params = parts[1:]
                actions.append((action_name, params))

    return actions
```

Usage Example:

```python
# Parse the plan
plan = parse_plan("plan.txt")
print(f"Plan contains {len(plan)} actions:")
for action, params in plan:
    print(f"  {action} {' '.join(params)}")

# Possible output:
# Plan contains 5 actions:
#   move robot1 zone_0_0 zone_1_0
#   move robot1 zone_1_0 zone_2_0
#   pickup robot1 pkg1 zone_2_0
#   move robot1 zone_2_0 zone_2_1
```

```
#   drop robot1 pkg1 zone_2_1
```

3.6 Plan Execution in Simulation

Once the plan is parsed, you must execute it in your simulation by translating each PDDL action into a corresponding Python action.

```python
class PlanExecutor:
    def __init__(self, environment, robots, packages):
        """
        Initialize the plan executor

        Args:
            environment: Environment object (grid)
            robots: List of robots
            packages: List of packages
        """
        self.environment = environment
        self.robots = {r.id: r for r in robots}
        self.packages = {p.id: p for p in packages}

    def execute_action(self, action_name, params):
        """
        Execute a plan action in the simulation

        Args:
            action_name: Action name (move, pickup, drop)
            params: List of parameters

        Returns:
            bool: True if the action succeeded
        """
        try:
            if action_name == "move":
                robot_id, from_zone, to_zone = params
                robot = self.robots[robot_id]

                # Extract target zone coordinates
                # Format: zone_x_y
                to_coords = self._parse_zone(to_zone)
                from_coords = self._parse_zone(from_zone)

                # Calculate movement direction
                direction = self._get_direction(from_coords, to_coords)

                # Move the robot
                success = robot.move(direction)
                return success

            elif action_name == "pickup":
```

```
                robot_id, package_id, location = params
                robot = self.robots[robot_id]
                package = self.packages[package_id]

                # Load the package
                success = robot.pickup(package)
                if success:
                    package.is_carried = True
                    package.carrier_id = robot_id
                return success

            elif action_name == "drop":
                robot_id, package_id, location = params
                robot = self.robots[robot_id]
                package = self.packages[package_id]

                # Unload the package
                success = robot.drop(package)
                if success:
                    package.is_carried = False
                    package.position = robot.position
                return success

            else:
                print(f"Unknown action: {action_name}")
                return False

        except Exception as e:
            print(f"Error during execution: {e}")
            return False

    def _parse_zone(self, zone_name):
        """
        Extract coordinates from a zone name

        Args:
            zone_name: Zone name (ex: "zone_2_3")

        Returns:
            tuple: Coordinates (x, y)
        """
        parts = zone_name.split('_')
        x = int(parts[1])
        y = int(parts[2])
        return (x, y)

    def _get_direction(self, from_pos, to_pos):
        """
        Calculate movement direction

        Args:
```

```
            from_pos: Starting position (x, y)
            to_pos: Ending position (x, y)

        Returns:
            str: Direction ('up', 'down', 'left', 'right')
        """
        dx = to_pos[0] - from_pos[0]
        dy = to_pos[1] - from_pos[1]

        if dx > 0:
            return 'right'
        elif dx < 0:
            return 'left'
        elif dy > 0:
            return 'up'
        elif dy < 0:
            return 'down'
        else:
            return None

    def execute_plan(self, plan, delay=0.5):
        """
        Execute the entire plan with animation

        Args:
            plan: List of actions [(action, params), ...]
            delay: Delay between each action (seconds)

        Returns:
            bool: True if the entire % Continuation of the PDDL Project English
            ↪  Translation

plan was executed
        """
        import time

        for i, (action, params) in enumerate(plan):
            print(f"Step {i+1}/{len(plan)}: {action} {' '.join(params)}")

            success = self.execute_action(action, params)

            if not success:
                print(f"Failure at step {i+1}")
                return False

            # Pause for animation
            time.sleep(delay)

        print("Plan executed successfully!")
        return True
```

Usage Example:

```
# Create the executor
executor = PlanExecutor(environment, robots, packages)

# Parse and execute the plan
plan = parse_plan("solution.txt")
executor.execute_plan(plan, delay=0.5)
```

3.7 Integration into Graphical Interface

Your graphical interface must allow visualization of plan execution in real-time. Here's an example with pygame:

```
import pygame
import time

class GUI:
    def __init__(self, environment, robots, packages):
        pygame.init()
        self.environment = environment
        self.robots = robots
        self.packages = packages

        # Graphics parameters
        self.cell_size = 60
        self.width = environment.width * self.cell_size
        self.height = environment.height * self.cell_size + 100

        self.screen = pygame.display.set_mode((self.width, self.height))
        pygame.display.set_caption("Robot Warehouse Planner")

        # Colors
        self.WHITE = (255, 255, 255)
        self.BLACK = (0, 0, 0)
        self.BLUE = (100, 150, 255)
        self.ORANGE = (255, 165, 0)
        self.RED = (255, 50, 50)
        self.GRAY = (128, 128, 128)

    def draw(self):
        """Draw the current state"""
        self.screen.fill(self.WHITE)

        # Draw the grid
        for x in range(self.environment.width):
            for y in range(self.environment.height):
                rect = pygame.Rect(
                    x * self.cell_size,
```

```
                    y * self.cell_size,
                    self.cell_size,
                    self.cell_size
                )

                if self.environment.is_obstacle(x, y):
                    pygame.draw.rect(self.screen, self.GRAY, rect)

                pygame.draw.rect(self.screen, self.BLACK, rect, 1)

        # Draw destinations
        for pkg in self.packages:
            if not pkg.is_carried:
                dest_x, dest_y = pkg.destination
                center = (
                    dest_x * self.cell_size + self.cell_size // 2,
                    dest_y * self.cell_size + self.cell_size // 2
                )
                pygame.draw.circle(self.screen, self.RED, center, 10, 2)

        # Draw packages
        for pkg in self.packages:
            if not pkg.is_carried:
                x, y = pkg.position
                center = (
                    x * self.cell_size + self.cell_size // 2,
                    y * self.cell_size + self.cell_size // 2
                )
                pygame.draw.circle(self.screen, self.ORANGE, center, 15)

        # Draw robots
        for robot in self.robots:
            x, y = robot.position
            center = (
                x * self.cell_size + self.cell_size // 2,
                y * self.cell_size + self.cell_size // 2
            )
            pygame.draw.circle(self.screen, self.BLUE, center, 20)

            # Display number of packages carried
            font = pygame.font.Font(None, 24)
            text = font.render(str(len(robot.carrying)), True, self.WHITE)
            text_rect = text.get_rect(center=center)
            self.screen.blit(text, text_rect)

        # Display information
        self._draw_info()

        pygame.display.flip()

    def _draw_info(self):
```

```
        """Display information at the bottom of the screen"""
        info_y = self.environment.height * self.cell_size + 10
        font = pygame.font.Font(None, 20)

        delivered = sum(1 for p in self.packages
                        if p.position == p.destination)
        info_text = f"Packages delivered: {delivered}/{len(self.packages)}"
        text = font.render(info_text, True, self.BLACK)
        self.screen.blit(text, (10, info_y))

    def animate_plan(self, plan, executor):
        """Animate the plan execution"""
        clock = pygame.time.Clock()

        for i, (action, params) in enumerate(plan):
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    return False

            # Execute the action
            executor.execute_action(action, params)

            # Redraw
            self.draw()

            # Wait
            clock.tick(2)  # 2 FPS

        return True
```

3.8 Evaluation Criteria - Part 2

Criterion	Points
PDDL Modeling (domain.pddl)	20
- Correct types and predicates	6
- Well-defined actions (preconditions/effects)	10
- Model completeness	4
Automatic generation of problem.pddl	10
Fast Downward integration	10
- Correct planner call	5
- Plan parsing	5
Plan execution in simulation	15
- Translation PDDL actions → Python	8
- Visual animation	7
Testing and validation	5
Total Part 2	**60**

Table 2: Part 2 Grading

4 Deliverables and Evaluation

4.1 Expected Deliverables

You must submit a ZIP file containing:

1. **Complete source code**:
   - All Python files (.py)
   - The domain.pddl file
   - A requirements.txt file listing dependencies
2. **Technical report** (PDF, 5-8 pages) containing:
   - System architecture description
   - Explanation of PDDL modeling choices
   - Application user guide
   - Test results with screenshots
   - Planner performance analysis
   - Difficulties encountered and solutions provided
3. **README.md file** with:
   - Installation instructions
   - Command to launch the application
   - Feature description
4. **(Optional) Demo video** (2-3 minutes)

4.2 Submission Format

- **File name:** `Project_PDDL_LASTNAME1_LASTNAME2.zip`
- **Submission:** Via Moodle platform
- **Deadline:** Sunday, December 15, 11:59 PM

4.3 Overall Grading

4.4 Presentation (optional)

An oral presentation of 15 minutes may be organized:

- 10 minutes of presentation
- 5 minutes of Q&A
- Live demonstration of the application

Component	Points
Part 1: Python Simulation	40
Part 2: PDDL Modeling and Planning	60
Technical Report	10 (bonus)
Overall code quality	5 (bonus)
Originality / Advanced features	5 (bonus)
Total	**100 (+ 20 bonus)**

Table 3: Overall project grading

5 Advice and Possible Extensions

5.1 Tips for Success

1. **Start simple**: Test first with 1 robot and 2 packages on a small grid
2. **Test regularly**: Validate each component before moving to the next
3. **PDDL Debugging**: Use online validators like `http://editor.planning.domains`
4. **Error handling**: Provide clear error messages
5. **Documentation**: Comment your code as you go
6. **Version control**: Use Git to save your progress

5.2 Possible Extensions (bonus)

If you want to go further, here are interesting extensions:

Extension 1: Collision Management Prevent two robots from occupying the same cell simultaneously by adding an `(occupied ?l - location)` predicate in your PDDL domain.

Extension 2: Variable Capacity Allow different robots to have different capacities (1, 2, or 3 packages).

Extension 3: Action Cost Use PDDL with numeric functions to minimize the total number of movements:

```
(:functions
    (total-cost)
)

(:action move
    :parameters (?r - robot ?from - location ?to - location)
    :precondition (...)
    :effect (and
        ...
        (increase (total-cost) 1)
```

```
    )
)

(:metric minimize (total-cost))
```

Extension 4: Advanced Interface

- Buttons to add/remove robots and packages dynamically
- Save/load scenarios
- Compare different heuristics
- Display statistics (planning time, explored states)

Extension 5: Dynamic Obstacles Allow the user to place obstacles by clicking on the grid.

Extension 6: Multi-Objective Add priorities to packages (urgent vs. normal delivery).

Extension 7: Manual Mode Allow manual control of robots (with arrow keys) to compare with the automatic plan.

5.3 Useful Resources

- **PDDL Documentation**: `https://planning.wiki`
- **Fast Downward**: `https://www.fast-downward.org`
- **Online PDDL Editor**: `http://editor.planning.domains`
- **Pygame tutorials**: `https://www.pygame.org/docs`
- **PDDL Examples**: `https://github.com/AI-Planning/pddl-generators`

5.4 FAQ

Q: My planner doesn't find a solution, what should I do?
A: Check that:

- The initial state contains all necessary facts
- Actions have correct effects
- There exists a valid path to the goal
- Test with a simpler problem first

Q: How do I debug my PDDL file?

A: Use the online editor `http://editor.planning.domains` which detects syntax errors and allows plan visualization.

Q: Can I use a different planner than Fast Downward?

A: Yes, but Fast Downward is recommended. You can also try LAMA, FF, or other planners.

Q: How do I handle multiple robots that don't block each other?

A: This is a complex problem! For the basic version, you can assume robots don't meet. To go further, add collision management predicates.

6 Appendices

6.1 Example requirements.txt File

```
pygame==2.5.2
numpy==1.24.3
```

6.2 Example README.md

````
# Robot Warehouse Planner

Automated planning project with PDDL and Python.

## Installation

1. Install dependencies:
```bash
pip install -r requirements.txt
```

2. Install Fast Downward:
```bash
git clone https://github.com/aibasel/downward.git
cd downward
./build.py
```

## Usage

Launch the application:
```bash
python main.py
```

## Features

- Visual simulation of a warehouse with robots and packages
- Automatic PDDL problem generation
- Planning with Fast Downward
- Plan execution animation

## Authors

- LASTNAME1 Firstname1
- LASTNAME2 Firstname2

## Date xx/xx/xxxx
````

6.3 Report Template

The technical report should follow this structure:

1. **Introduction** (0.5 page)
    - Project presentation
    - Objectives
2. **System Architecture** (1 page)
    - Architecture diagram
    - Python module description
    - Data flow
3. **PDDL Modeling** (2 pages)
    - Justification of types and predicates
    - Detailed action description
    - Example of generated problem
4. **Implementation** (1.5 pages)
    - Technical choices
    - Difficulties encountered
    - Solutions provided
5. **Testing and Results** (1.5 pages)
    - Test scenarios
    - Screenshots
    - Planner performance
    - Results analysis
6. **Conclusion** (0.5 page)
    - Summary
    - Possible improvements

Good luck with your project!

Don't hesitate to ask questions during the lab sessions.