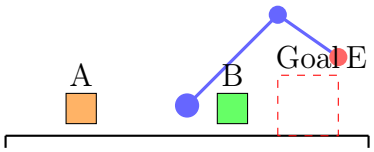
AI Planning

Project 1 - Automated Block Manipulation System

Planning System with Robot Arm Manipulator



Halim Djerroud
LISV - Paris-Saclay University

Contents

1 Introduction

1.1 Project Context

This project constitutes the final assessment for the first part of the Automated Planning course. It allows you to apply all the knowledge acquired on PDDL (Planning Domain Description Language) in a concrete robotic manipulation context.

Learning Objectives Upon completion of this project, you should be able to:

- Model a robotic manipulation problem using PDDL
- Develop an interactive Python application with graphical interface
- Integrate an external planner (Fast Downward)
- Automatically extract system state to generate a PDDL problem
- Interpret and execute a plan generated by a planner
- Manage manipulation constraints (grasping, stacking, etc.)

1.2 General System Description

You will develop a **complete planning system for robot arms** 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 Manipulation Workshop One or more robot arm manipulators operate in a shared workspace. Their mission is to grasp objects located on different tables/zones, manipulate them (stack, move), and position them in target configurations. The arms have limited grasping capacity (one piece at a time) and must manage physical constraints (stacked objects, collisions, accessibility).

1.4 Practical Details

- **Work Mode:** Pairs / Groups of three
- **Duration:** 3 weeks
- **Weight:** 50% of final grade
- **Languages:** Python 3.x + PDDL
- **Planner:** Fast Downward (provided)

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

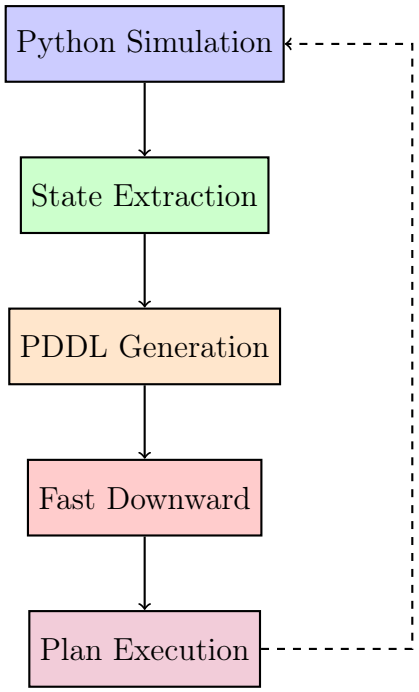


Figure 1: Overall System Architecture

2 Part 1: Simulation Environment (40 points)

2.1 Overview

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

2.2 Functional Specifications

2.2.1 Workspace

- **Type:** 2D space with several zones/tables
- **Zones:** 3 to 5 distinct zones (table1, table2, depot_zone, etc.)
- **Positions:** Each zone can contain multiple objects
- **Accessibility:** Some zones may be accessible by certain arms only

2.2.2 Robot Arms

You must implement **at least 1 robot arm**, ideally 2 arms:

- State: free or holding an object
- Capacity: 1 object at a time (simple gripper)
- Possible actions:
  - `pick(object, position)`: grasp an object

  - `place(object, position)`: place an object
  - `stack(object1, object2)`: stack object1 on object2
  - `unstack(object1, object2)`: unstack object1 from object2
- Internal state: currently held object (or None)

2.2.3 Manipulable Objects

- **Number:** Minimum 5 objects (blocks, pieces)
- **Types:** Cubes, cylinders (optional: different sizes)
- **Properties:**
  - Current position (on table, on another object, in gripper)
  - State: `on_table`, `on(other_object)`, `in_gripper`
  - `clear`: nothing is placed on top
- **Target configuration:** Desired final arrangement

2.3 Required Graphical Interface

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

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

2.3.1 Required Visual Elements

1. **Workspace view** with:
   - Tables/zones represented by rectangles
   - Objects drawn as stackable rectangles/squares
   - Robot arm(s) with state indicator (free/busy)
   - Gripper visualized (open/closed, with/without object)
2. **Side view (optional)**:
   - 2D display of object stacks
   - Clear visualization of stacking relationships
3. **Control panel** with buttons:
   - "Extract Initial State" → generates `problem.pddl`
   - "Define Goal" → specify target configuration
   - "Plan" → calls Fast Downward
   - "Execute Plan" → animates the obtained plan

- "Reset" → returns environment to initial state

4. **Information zone** displaying:
   - State of each arm (free/busy)
   - Object currently in gripper
   - List of objects with their position
   - Last action executed
   - Any error messages

2.4 Python Code Architecture

Your code must be structured in separate modules:

```
project/
|-- main.py              # Entry point
|-- workspace.py         # Workspace class
|-- robot_arm.py         # RobotArm class
|-- block.py             # Block class (manipulable object)
|-- location.py          # Location class (zone/table)
|-- gui.py               # Graphical interface
|-- pddl_generator.py    # PDDL file generation
|-- planner_interface.py # Fast Downward interface
|-- plan_executor.py     # Plan execution and animation
|-- domain.pddl          # PDDL domain (Part 2)
|-- problem.pddl         # Generated automatically
```

2.5 Code Example: RobotArm Class

```python
class RobotArm:
    def __init__(self, arm_id, workspace):
        """
        Initialize a robot arm

        Args:
            arm_id (str): Unique arm identifier
            workspace: Reference to the workspace
        """
        self.id = arm_id
        self.workspace = workspace
        self.holding = None  # Currently held object
        self.is_free = True

    def pick_from_table(self, block, location):
        """
        Grasp an object from a table

```

```
        Args:
            block: The object to grasp
            location: The zone where the object is located

        Returns:
            bool: True if the action succeeded
        """
        if not self.is_free:
            print(f"{self.id} is already holding an object")
            return False

        if not block.is_clear:
            print(f"{block.id} is not clear (object on top)")
            return False

        if block.location != location:
            print(f"{block.id} is not at {location.id}")
            return False

        # Grasp the object
        self.holding = block
        self.is_free = False
        block.location = None
        block.on_table = False

        print(f"{self.id} grasped {block.id} from {location.id}")
        return True

    def place_on_table(self, block, location):
        """
        Place an object on a table

        Args:
            block: The object to place
            location: The target zone

        Returns:
            bool: True if the action succeeded
        """
        if self.holding != block:
            print(f"{self.id} is not holding {block.id}")
            return False

        # Place the object
        self.holding = None
        self.is_free = True
        block.location = location
        block.on_table = True
        block.is_clear = True
        location.add_block(block)

```

```
        print(f"{self.id} placed {block.id} on {location.id}")
        return True

    def unstack(self, top_block, bottom_block):
        """
        Unstack an object from another

        Args:
            top_block: The top object to grasp
            bottom_block: The bottom object

        Returns:
            bool: True if the action succeeded
        """
        if not self.is_free:
            return False

        if top_block.on_block != bottom_block:
            print(f"{top_block.id} is not on {bottom_block.id}")
            return False

        if not top_block.is_clear:
            print(f"{top_block.id} is not clear")
            return False

        # Unstack
        self.holding = top_block
        self.is_free = False
        top_block.on_block = None
        top_block.on_table = False
        bottom_block.is_clear = True

        print(f"{self.id} unstacked {top_block.id} from {bottom_block.id}")
        return True

    def stack(self, top_block, bottom_block):
        """
        Stack an object on another

        Args:
            top_block: The held object to stack
            bottom_block: The target object

        Returns:
            bool: True if the action succeeded
        """
        if self.holding != top_block:
            return False

        if not bottom_block.is_clear:
            print(f"{bottom_block.id} is not clear")
```

```
            return False

        # Stack
        self.holding = None
        self.is_free = True
        top_block.on_block = bottom_block
        top_block.is_clear = True
        bottom_block.is_clear = False

        print(f"{self.id} stacked {top_block.id} on {bottom_block.id}")
        return True
```

2.6 Code Example: Block Class

```
class Block:
    def __init__(self, block_id, location=None):
        """
        Initialize a manipulable block

        Args:
            block_id (str): Unique block identifier
            location: Initial zone (or None if in gripper)
        """
        self.id = block_id
        self.location = location  # Location object or None
        self.on_table = True if location else False
        self.on_block = None  # Block on which this one is placed
        self.is_clear = True  # Nothing is placed on top
        self.color = self._random_color()

    def _random_color(self):
        """Generate a random color for visualization"""
        import random
        colors = ['red', 'blue', 'green', 'yellow',
                  'orange', 'purple', 'pink', 'cyan']
        return random.choice(colors)

    def get_state(self):
        """Return the current state of the block"""
        if self.on_table:
            return f"on_table at {self.location.id}"
        elif self.on_block:
            return f"on {self.on_block.id}"
        else:
            return "in_gripper"

    def is_on_table(self):
        """Check if the block is on a table"""
        return self.on_table

```

```
    def is_stacked(self):
        """Check if the block is stacked on another"""
        return self.on_block is not None
```

Usage Example:

```
# Create workspace
workspace = Workspace()
table1 = Location("table1")
workspace.add_location(table1)

# Create blocks
block_a = Block("block_a", table1)
block_b = Block("block_b", table1)

# Create a robot arm
arm = RobotArm("arm1", workspace)

# Grasp a block
arm.pick_from_table(block_a, table1)
print(f"Arm free: {arm.is_free}")  # False

# Stack on another block
arm.stack(block_a, block_b)
print(f"block_a on: {block_a.on_block.id}")  # block_b
print(f"block_b clear: {block_b.is_clear}")  # False
```

2.7 Initial State Extraction

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

```
def extract_state_to_pddl(workspace, arms, blocks,
                          goal_config, output_file):
    """
    Extract current state and generate problem.pddl

    Args:
        workspace: Workspace object
        arms: List of robot arms
        blocks: List of blocks
        goal_config: Target configuration (dict)
        output_file: Output file path
    """
    with open(output_file, 'w') as f:
        f.write("(define (problem blocks-manipulation)\n")
        f.write("  (:domain blocks-world)\n\n")

```

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

        # Robot arms
        for arm in arms:
            f.write(f"    {arm.id} - arm\n")

        # Blocks
        for block in blocks:
            f.write(f"    {block.id} - block\n")

        # Locations
        for loc in workspace.locations:
            f.write(f"    {loc.id} - location\n")

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

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

        # Arm states
        for arm in arms:
            if arm.is_free:
                f.write(f"    (arm-empty {arm.id})\n")
            else:
                f.write(f"    (holding {arm.id} {arm.holding.id})\n")

        # Block states
        for block in blocks:
            # Position
            if block.on_table:
                f.write(f"    (on-table {block.id})\n")
                f.write(f"    (at {block.id} {block.location.id})\n")
            elif block.on_block:
                f.write(f"    (on {block.id} {block.on_block.id})\n")

            # Clarity
            if block.is_clear:
                f.write(f"    (clear {block.id})\n")

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

        # 3. Goal
        f.write("  (:goal\n    (and\n")

        for block_id, config in goal_config.items():
            if config['type'] == 'on_table':
                f.write(f"      (on-table {block_id})\n")
                if 'location' in config:
                    f.write(f"      (at {block_id} "
                            f"{config['location']})\n")
```

```
            elif config['type'] == 'on':
                f.write(f"      (on {block_id} {config['on']})\n")

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

# Example target configuration
goal_config = {
    'block_a': {'type': 'on', 'on': 'block_b'},
    'block_b': {'type': 'on', 'on': 'block_c'},
    'block_c': {'type': 'on_table', 'location': 'table1'}
}
```

2.8 Evaluation Criteria - Part 1

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

Table 1: Part 1 Grading Rubric

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

3.1 Overview

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

3.2 PDDL Domain Modeling

You must create a complete and correct `domain.pddl` file inspired by the "Blocks World" domain.

3.2.1 Types to Define

```
(:types
    arm block location - object
)
```

3.2.2 Required Predicates

Here are the minimum predicates to implement:

```
(:predicates
    (arm-empty ?a - arm)
    (holding ?a - arm ?b - block)
    (on-table ?b - block)
    (on ?b1 - block ?b2 - block)
    (clear ?b - block)
    (at ?b - block ?l - location)
)
```

Modeling Advice Blocks World is a classic planning domain! Think carefully about relationships:

- A block is either on the table, on another block, or in the gripper
- A block is "clear" if nothing is placed on top
- An arm can hold at most one block
- To stack A on B, B must be "clear" and A must be in the gripper

3.2.3 Actions to Implement

Action 1: PICK-UP (Grasp from table)

```
(:action pick-up
    :parameters (?a - arm ?b - block ?l - location)
    :precondition (and
        (arm-empty ?a)
        (on-table ?b)
        (at ?b ?l)
        (clear ?b)
    )
    :effect (and
        (holding ?a ?b)
        (not (arm-empty ?a))
        (not (on-table ?b))
        (not (at ?b ?l))
    )
)
```

Action 2: PUT-DOWN (Place on table)

```
(:action put-down
    :parameters (?a - arm ?b - block ?l - location)
    :precondition (and
        (holding ?a ?b)
    )
    :effect (and
        (arm-empty ?a)
        (on-table ?b)
        (at ?b ?l)
        (clear ?b)
        (not (holding ?a ?b))
    )
)
```

Action 3: STACK

```
(:action stack
    :parameters (?a - arm ?top - block ?bottom - block)
    :precondition (and
        (holding ?a ?top)
        (clear ?bottom)
    )
    :effect (and
        (arm-empty ?a)
        (on ?top ?bottom)
        (clear ?top)
        (not (holding ?a ?top))
        (not (clear ?bottom))
    )
```

```
)
```

Action 4: UNSTACK

```
(:action unstack
    :parameters (?a - arm ?top - block ?bottom - block)
    :precondition (and
        (arm-empty ?a)
        (on ?top ?bottom)
        (clear ?top)
    )
    :effect (and
        (holding ?a ?top)
        (clear ?bottom)
        (not (arm-empty ?a))
        (not (on ?top ?bottom))
        (not (clear ?top))
    )
)
```

3.3 Complete PDDL Domain Example

```
(define (domain blocks-world)
    (:requirements :strips :typing)

    (:types
        arm block location - object
    )

    (:predicates
        (arm-empty ?a - arm)
        (holding ?a - arm ?b - block)
        (on-table ?b - block)
        (on ?b1 - block ?b2 - block)
        (clear ?b - block)
        (at ?b - block ?l - location)
    )

    (:action pick-up
        :parameters (?a - arm ?b - block ?l - location)
        :precondition (and
            (arm-empty ?a)
            (on-table ?b)
            (at ?b ?l)
            (clear ?b)
        )
        :effect (and
            (holding ?a ?b)
```

```
            (not (arm-empty ?a))
            (not (on-table ?b))
            (not (at ?b ?l))
        )
    )

    (:action put-down
        :parameters (?a - arm ?b - block ?l - location)
        :precondition (and
            (holding ?a ?b)
        )
        :effect (and
            (arm-empty ?a)
            (on-table ?b)
            (at ?b ?l)
            (clear ?b)
            (not (holding ?a ?b))
        )
    )

    (:action stack
        :parameters (?a - arm ?top - block ?bottom - block)
        :precondition (and
            (holding ?a ?top)
            (clear ?bottom)
        )
        :effect (and
            (arm-empty ?a)
            (on ?top ?bottom)
            (clear ?top)
            (not (holding ?a ?top))
            (not (clear ?bottom))
        )
    )

    (:action unstack
        :parameters (?a - arm ?top - block ?bottom - block)
        :precondition (and
            (arm-empty ?a)
            (on ?top ?bottom)
            (clear ?top)
        )
        :effect (and
            (holding ?a ?top)
            (clear ?bottom)
            (not (arm-empty ?a))
            (not (on ?top ?bottom))
            (not (clear ?top))
        )
    )
)
```

3.4 Fast Downward Integration

3.4.1 Installation

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

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

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

3.4.2 Calling the Planner from Python

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

```

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

            # Plan is in 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

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

    Args:
        plan_file: Path to 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: "(pick-up arm1 block_a table1)"
            # or: "pick-up arm1 block_a table1"

            # 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:

```
# 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:
#   pick-up arm1 block_a table1
#   stack arm1 block_a block_b
#   unstack arm1 block_a block_b
#   put-down arm1 block_a table2
```

3.6 Plan Execution in Simulation

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

```
class PlanExecutor:
    def __init__(self, workspace, arms, blocks):
        """
```

```
        Initialize plan executor

        Args:
            workspace: Workspace object
            arms: List of robot arms
            blocks: List of blocks
        """
        self.workspace = workspace
        self.arms = {a.id: a for a in arms}
        self.blocks = {b.id: b for b in blocks}
        self.locations = {l.id: l for l in workspace.locations}

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

        Args:
            action_name: Action name
                         (pick-up, put-down, stack, unstack)
            params: List of parameters

        Returns:
            bool: True if action succeeded
        """
        try:
            if action_name == "pick-up":
                arm_id, block_id, location_id = params
                arm = self.arms[arm_id]
                block = self.blocks[block_id]
                location = self.locations[location_id]

                return arm.pick_from_table(block, location)

            elif action_name == "put-down":
                arm_id, block_id, location_id = params
                arm = self.arms[arm_id]
                block = self.blocks[block_id]
                location = self.locations[location_id]

                return arm.place_on_table(block, location)

            elif action_name == "stack":
                arm_id, top_id, bottom_id = params
                arm = self.arms[arm_id]
                top_block = self.blocks[top_id]
                bottom_block = self.blocks[bottom_id]

                return arm.stack(top_block, bottom_block)

            elif action_name == "unstack":
                arm_id, top_id, bottom_id = params
```

```
                arm = self.arms[arm_id]
                top_block = self.blocks[top_id]
                bottom_block = self.blocks[bottom_id]

                return arm.unstack(top_block, bottom_block)

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

        except KeyError as e:
            print(f"Object not found: {e}")
            return False
        except Exception as e:
            print(f"Error during execution: {e}")
            return False

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

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

        Returns:
            bool: True if entire 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"Failed at step {i+1}")
                return False

            # Pause for animation
            time.sleep(delay)

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

Usage Example:

```
# Create executor
executor = PlanExecutor(workspace, arms, blocks)

```

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

3.7 Graphical Interface Integration

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, workspace, arms, blocks):
        pygame.init()
        self.workspace = workspace
        self.arms = arms
        self.blocks = blocks

        # Graphics parameters
        self.width = 800
        self.height = 600
        self.block_size = 50

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

        # Colors
        self.WHITE = (255, 255, 255)
        self.BLACK = (0, 0, 0)
        self.BLUE = (100, 150, 255)
        self.GRAY = (128, 128, 128)
        self.GREEN = (100, 255, 100)

        # Block colors
        self.color_map = {
            'red': (255, 50, 50),
            'blue': (50, 50, 255),
            'green': (50, 255, 50),
            'yellow': (255, 255, 50),
            'orange': (255, 165, 0),
            'purple': (200, 50, 200),
            'pink': (255, 150, 200),
            'cyan': (50, 255, 255)
        }

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

```

```
        # Draw tables/zones
        for i, location in enumerate(self.workspace.locations):
            x = 100 + i * 200
            y = 450

            # Table
            pygame.draw.rect(self.screen, self.GRAY,
                             (x, y, 150, 20))

            # Label
            font = pygame.font.Font(None, 24)
            text = font.render(location.id, True, self.BLACK)
            self.screen.blit(text, (x, y + 30))

        # Draw blocks
        for block in self.blocks:
            if block.on_table and block.location:
                # Block on table
                loc_index = self.workspace.locations.index(block.location)
                x = 100 + loc_index * 200 + 50
                y = 400

                # Adjust y if stacked
                y = self._get_block_y_position(block)

                color = self.color_map.get(block.color, self.BLUE)
                pygame.draw.rect(self.screen, color,
                                 (x, y, self.block_size, self.block_size))
                pygame.draw.rect(self.screen, self.BLACK,
                                 (x, y, self.block_size, self.block_size), 2)

                # Block label
                font = pygame.font.Font(None, 20)
                text = font.render(block.id.split('_')[-1].upper(),
                                   True, self.WHITE)
                text_rect = text.get_rect(
                    center=(x + self.block_size//2,
                            y + self.block_size//2))
                self.screen.blit(text, text_rect)

        # Draw robot arms
        for i, arm in enumerate(self.arms):
            x = 50 + i * 300
            y = 100

            # Simplified arm
            pygame.draw.circle(self.screen, self.BLUE, (x, y), 20)

            # Arm state
            font = pygame.font.Font(None, 20)
            state = "Free" if arm.is_free else f"Holding: {arm.holding.id}"
```

```
            text = font.render(f"{arm.id}: {state}", True, self.BLACK)
            self.screen.blit(text, (x - 50, y + 30))

            # Draw block in gripper
            if not arm.is_free and arm.holding:
                block = arm.holding
                color = self.color_map.get(block.color, self.BLUE)
                pygame.draw.rect(self.screen, color,
                                 (x - 25, y + 50,
                                  self.block_size, self.block_size))
                pygame.draw.rect(self.screen, self.BLACK,
                                 (x - 25, y + 50,
                                  self.block_size, self.block_size), 2)

        # Display information
        self._draw_info()

        pygame.display.flip()

    def _get_block_y_position(self, block):
        """Calculate Y position of a block (for stacking)"""
        base_y = 400
        height = 0

        # Count blocks below
        current = block
        while current.on_block:
            height += 1
            current = current.on_block

        return base_y - height * self.block_size

    def _draw_info(self):
        """Display information at top of screen"""
        font = pygame.font.Font(None, 24)

        info_text = "Workspace State"
        text = font.render(info_text, True, self.BLACK)
        self.screen.blit(text, (10, 10))

        # Number of blocks by state
        on_table_count = sum(1 for b in self.blocks if b.on_table)
        stacked_count = sum(1 for b in self.blocks if b.on_block)
        in_gripper_count = sum(1 for b in self.blocks
                               if not b.on_table and not b.on_block)

        info = f"On table: {on_table_count} | " \
               f"Stacked: {stacked_count} | " \
               f"In gripper: {in_gripper_count}"
        text = font.render(info, True, self.BLACK)
        self.screen.blit(text, (10, 40))
```

```python

    def animate_plan(self, plan, executor):
        """Animate 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 action
            print(f"\nStep {i+1}: {action} {' '.join(params)}")
            executor.execute_action(action, params)

            # Redraw
            self.draw()

            # Wait
            clock.tick(1)  # 1 FPS to see animation

        return True

    def run(self):
        """Main interface loop"""
        running = True
        clock = pygame.time.Clock()

        while running:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False

            self.draw()
            clock.tick(30)

        pygame.quit()
```

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
- PDDL to Python action translation	8
- Visual animation	7
Testing and validation	5
Part 2 Total	**60**

Table 2: Part 2 Grading Rubric

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 application
   - Feature description
4. **(Optional) Demo video** (2-3 minutes)

4.2 Submission Format

- **Filename:** `Project_PDDL_RobotArm_NAME1_NAME2.zip`
- **Submission:** Via Moodle platform
- **Deadline:** Sunday, December 15th, 11:59 PM

4.3 Overall Grading

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 Tips and Possible Extensions

5.1 Tips for Success

1. **Start simple**: Test first with 3 blocks and 1 arm
2. **Test regularly**: Validate each PDDL action separately
3. **PDDL debugging**: Use online validators like `http://editor.planning.domains`
4. **Visualization**: A good graphical interface facilitates debugging
5. **Documentation**: Comment your code as you go
6. **Version control**: Use Git to save your progress
7. **Unit tests**: Test each manipulation method individually

5.2 Possible Extensions (bonus)

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

Extension 1: Multiple Robot Arms Manage 2 or 3 arms that can work simultaneously on different zones. Watch out for collision constraints!

Extension 2: Different Block Sizes Some larger blocks can only be placed on blocks of the same size or larger.

```
(:predicates
    (larger ?b1 - block ?b2 - block)
)

(:action stack
    :parameters (?a - arm ?top - block ?bottom - block)
    :precondition (and
        (holding ?a ?top)
        (clear ?bottom)
        (or (larger ?bottom ?top) (equal-size ?top ?bottom))
    )
    ...
)
```

Extension 3: Action Costs Minimize number of actions or optimize total time:

```
(:functions
    (total-cost)
    (move-cost ?b - block)
)

```

```
(:action pick-up
    :parameters (?a - arm ?b - block ?l - location)
    :precondition (...)
    :effect (and
        ...
        (increase (total-cost) (move-cost ?b))
    )
)

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

Extension 4: Forbidden Zones Some blocks cannot be placed in certain zones (spatial constraints).

```
(:predicates
    (can-place ?b - block ?l - location)
)

(:action put-down
    :parameters (?a - arm ?b - block ?l - location)
    :precondition (and
        (holding ?a ?b)
        (can-place ?b ?l)
    )
    ...
)
```

Extension 5: Multiple Grippers An arm can have 2 grippers and hold 2 small objects simultaneously.

Extension 6: Fragile Objects Some blocks cannot support weight and must remain at the top of stacks.

```
(:predicates
    (fragile ?b - block)
)

(:action stack
    :parameters (?a - arm ?top - block ?bottom - block)
    :precondition (and
        (holding ?a ?top)
        (clear ?bottom)
        (not (fragile ?bottom))
    )
    ...
)
```

Extension 7: Interactive Mode Allow user to click to define goal graphically (drag & drop of blocks).

Extension 8: Heuristic Comparison Compare performance of different Fast Downward heuristics:

- `astar(blind())`: A* search without heuristic
- `astar(hmax())`: h-max heuristic
- `astar(lmcut())`: LM-cut (recommended)
- `lazy_greedy([ff()])`: Greedy with FF heuristic

5.3 Classic Blocks World Problems

Here are some interesting test configurations:

Test 1: Reverse a Stack

- Initial state: A on B on C (on table)
- Goal: C on B on A (on table)

Test 2: Tower of Hanoi

- Initial state: 3 disks stacked from largest to smallest
- Goal: Move entire tower to another zone
- Constraint: A disk can only be placed on a larger one

Test 3: Block Sorting

- Initial state: Blocks mixed on multiple tables
- Goal: Stack all blocks in alphabetical order

Test 4: Simultaneous Construction

- Initial state: Scattered blocks
- Goal: Build 2 identical stacks in parallel
- Requires 2 robot arms

5.4 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`
- **Classic Blocks World**: `https://en.wikipedia.org/wiki/Blocks_world`
- **IPC Benchmarks**: `https://github.com/potassco/pddl-instances`

5.5 FAQ

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

- All blocks have correct `(clear ...)` predicate in initial state
- Blocks on table have `(on-table ...)`
- Stacking relationships are consistent
- Goal is achievable (no impossible constraints)
- Test with simpler problem (2 blocks, 1 action)

Q: How to debug stacking relationships?
A: Display complete state in your interface:

- For each block, display: what it's on, if it's clear
- Verify that if A is on B, then B is NOT clear
- A block cannot be in two states simultaneously (on table AND on another block)

Q: How to manage multiple robot arms?
A: In your PDDL model, each action must specify which arm performs the action. The planner will automatically choose which arm to use. To avoid conflicts, you can add arm position predicates.

Q: Can I use another planner than Fast Downward?
A: Yes, you can try:

- **LAMA**: Good for problems with costs
- **FF**: Fast but not optimal
- **Madagascar**: SAT-based
- **Pyperplan**: Written in Python, easy to integrate

Q: How to optimize generated plan?
A: Use different heuristics:

```
# For optimal plan (slower)
--search "astar(lmcut())"

# For fast plan (not necessarily optimal)
--search "lazy_greedy([ff()])"

# For good compromise
--search "lazy_wastar([hmax()], w=5)"
```

6 Appendices

6.1 Example requirements.txt File

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

6.2 Example README.md

```
# Robot Arm Manipulation Planner

Automated planning project with PDDL for robot arm manipulator.

## Description

This project implements a complete planning system for a robot arm
manipulator in a "Blocks World" type environment. The system uses
PDDL for modeling and Fast Downward for planning.

## Features

- Visual simulation of workspace with stackable blocks
- Automatic generation of PDDL problems from simulation state
- Automatic planning with Fast Downward
- Plan execution animation
- Support for multiple robot arms (optional)
- Stacking constraint management

## Installation

### Prerequisites

- Python 3.8 or higher
- Fast Downward (see instructions below)

### Python Dependencies Installation

```bash
pip install -r requirements.txt
```

Fast Downward Installation

Linux/Mac

```bash
git clone https://github.com/aibasel/downward.git
cd downward
./build.py
```
```

````
cd ..
```

#### Windows

Download pre-compiled version from official site or
use WSL (Windows Subsystem for Linux).

## Usage

### Launch Application

```bash
python main.py
```

### Interface Usage

1. **Initial configuration**: Blocks are automatically placed on tables
2. **Define goal**: Click "Define Goal" and specify target configuration
3. **Extract state**: Click "Extract Initial State" to generate problem.pddl
4. **Plan**: Click "Plan" to launch Fast Downward
5. **Execute**: Click "Execute Plan" to see animation

### Project Structure

```
project/
|-- main.py                  # Entry point
|-- workspace.py             # Workspace class
|-- robot_arm.py             # RobotArm class
|-- block.py                 # Block class
|-- location.py              # Location class
|-- gui.py                   # Graphical interface
|-- pddl_generator.py        # PDDL generation
|-- planner_interface.py     # Fast Downward interface
|-- plan_executor.py         # Plan execution
|-- domain.pddl              # PDDL domain
|-- problem.pddl             # Problem (generated)
|-- requirements.txt         # Dependencies
```

## Test Scenario Examples

### Scenario 1: Simple Stack

**Initial state**:
- block_a on table1
- block_b on table1
- block_c on table1
````
````

```
**Goal**:
- block_a on block_b
- block_b on block_c
- block_c on table1

### Scenario 2: Stack Reversal

**Initial state**:
- block_a on block_b on block_c on table1

**Goal**:
- block_c on block_b on block_a on table1

### Scenario 3: Multi-table Reorganization

**Initial state**:
- block_a, block_b on table1
- block_c, block_d on table2

**Goal**:
- All blocks stacked on table3

## Configuration

You can modify parameters in `main.py`:

```python
# Number of blocks
NUM_BLOCKS = 5

# Number of robot arms
NUM_ARMS = 1

# Number of tables
NUM_TABLES = 3

# Animation speed
ANIMATION_SPEED = 0.5  # seconds between actions
```

Troubleshooting

Planner doesn't find solution

- Check that goal is achievable
- Simplify problem (fewer blocks)
- Check predicates in problem.pddl

Animation is too fast/slow

Modify `delay` parameter in `execute_plan()`.
```

```
### Error calling Fast Downward

- Check Fast Downward is properly installed
- Check path to fast-downward.py in planner_interface.py

## Authors

- [Your Name] - [Your Email]
- [Partner Name] - [Partner Email]

## Date

[DD/MM/YYYY]

## License

Academic project - LISV, Paris-Saclay University
```

6.3 Report Template

The technical report must follow this structure:

1. **Introduction** (0.5 page)
    - Project presentation
    - Objectives
    - Blocks World context
2. **System Architecture** (1 page)
    - Overall architecture diagram
    - Python module description
    - Data flow and interactions
    - Planning lifecycle
3. **PDDL Modeling** (2 pages)
    - Justification of chosen types
    - Detailed predicate explanation
    - Description of each action with examples
    - Modeled constraints
    - State/transition examples
4. **Python Implementation** (1.5 pages)
    - Design choices (classes, data structures)
    - State and constraint management

- Fast Downward integration
- Plan parsing and execution
- Difficulties encountered and solutions

5. **Graphical Interface** (1 page)
   - Technology choices (pygame/tkinter)
   - Visual object representation
   - Animation management
   - Commented screenshots
6. **Tests and Results** (1.5 pages)
   - Detailed test scenarios
   - Generated plans (number of actions, optimality)
   - Planner performance (time, memory)
   - Heuristic comparison (if done)
   - Results analysis
   - Edge cases and failures
7. **Implemented Extensions** (0.5 page)
   - Bonus feature description
   - Performance impact
8. **Conclusion** (0.5 page)
   - Project summary
   - Skills acquired
   - Possible improvements
   - Evolution perspectives
9. **Appendices**
   - Complete generated problem.pddl example
   - Example of obtained plan
   - Additional screenshots

6.4 Complete problem.pddl Example

```
(define (problem blocks-manipulation-test1)
  (:domain blocks-world)

  (:objects
    arm1 - arm
    block_a block_b block_c block_d block_e - block
```

```
    table1 table2 table3 - location
  )

  (:init
    ; Arm states
    (arm-empty arm1)

    ; Initial block configuration
    ; Stack 1 on table1: block_a on block_b
    (on block_a block_b)
    (on-table block_b)
    (at block_b table1)
    (clear block_a)

    ; block_c alone on table1
    (on-table block_c)
    (at block_c table1)
    (clear block_c)

    ; Stack 2 on table2: block_d on block_e
    (on block_d block_e)
    (on-table block_e)
    (at block_e table2)
    (clear block_d)
  )

  (:goal
    (and
      ; Build a tower on table3
      ; block_a on block_b on block_c on block_d on block_e
      (on block_a block_b)
      (on block_b block_c)
      (on block_c block_d)
      (on block_d block_e)
      (on-table block_e)
      (at block_e table3)
    )
  )
)
```

6.5 Example of Generated Plan

```
; Plan found by Fast Downward
; Number of actions: 11
; Total cost: 11

(unstack arm1 block_a block_b)
(put-down arm1 block_a table1)
(pick-up arm1 block_b table1)
(put-down arm1 block_b table3)
```

```
(unstack arm1 block_d block_e)
(put-down arm1 block_d table2)
(pick-up arm1 block_e table2)
(put-down arm1 block_e table3)
(pick-up arm1 block_c table1)
(stack arm1 block_c block_e)
(pick-up arm1 block_d table2)
(stack arm1 block_d block_c)
(pick-up arm1 block_b table3)
(stack arm1 block_b block_d)
(pick-up arm1 block_a table1)
(stack arm1 block_a block_b)
```

6.6 Final Validation Checklist

Before submitting your project, verify that:

- Python code is well-structured in modules
- All classes have docstrings
- Important functions are commented
- Graphical interface works without errors
- PDDL domain is syntactically correct
- problem.pddl generation works
- Fast Downward finds solutions for tests
- Plan parsing is correct
- Plan execution works and is visible
- At least 3 different test scenarios are provided
- Report contains all required sections
- Screenshots are included in report
- README.md is complete and clear
- requirements.txt lists all dependencies
- All files are in ZIP archive
- ZIP filename follows requested format

6.7 Self-Evaluation Grid

Use this grid to estimate your grade before submission:

Criterion	Max Points	Your Estimate
Part 1: Simulation (40 pts)		
Python code structure	8	/ 8
Functional graphical interface	10	/ 10
Clear visual representation	6	/ 6
Well-implemented classes	8	/ 8
State extraction to PDDL	8	/ 8
Part 2: PDDL (60 pts)		
Correct types and predicates	6	/ 6
Well-defined actions	10	/ 10
Model completeness	4	/ 4
problem.pddl generation	10	/ 10
Planner call	5	/ 5
Plan parsing	5	/ 5
PDDL to Python translation	8	/ 8
Visual animation	7	/ 7
Testing and validation	5	/ 5
Bonus (20 pts)		
Complete technical report	10	/ 10
Overall code quality	5	/ 5
Extensions/Originality	5	/ 5
TOTAL	**120**	**/ 120**

Table 4: Self-Evaluation Grid

Frequent Questions During Presentation

- Why did you choose this PDDL modeling?
- How do you manage conflicts between robot arms?
- What are the limits of your system?
- How could you improve performance?
- Did you test different heuristics?

Good luck with your project!

Don't hesitate to ask questions during lab sessions.

Blocks World is a classic and elegant planning problem.
Take advantage of this project to thoroughly understand PDDL fundamentals!

"The best way to predict the future is to plan it."