Robot Path Planning in a Warehouse

Idriss Mortadi & Abdellah Oumida

February 10, 2025

1 Introduction

In modern warehouses, autonomous robots play a crucial role in streamlining logistics by transporting goods efficiently. However, as the number of robots increases, ensuring smooth and collision-free navigation becomes a significant challenge. Poor path planning can lead to congestion, delays, and inefficient warehouse operations, negatively impacting productivity and throughput.

The problem of multi-robot path planning (MRPP) involves computing efficient routes for multiple robots navigating a shared environment while avoiding obstacles and preventing collisions. This task is particularly challenging in warehouse settings, where robots must operate in tight spaces with dynamically changing conditions. Efficient solutions must balance computational feasibility with optimality, ensuring all robots reach their destinations in minimal time while adhering to movement constraints.

To address this, we formulate the multi-robot path planning problem as a Satisfiability (SAT) problem, leveraging the efficiency of modern SAT solvers to compute collision-free paths. By encoding robot movements, obstacle constraints, and avoidance rules into Boolean variables and logical formulas, we can transform path planning into a constraint satisfaction problem solvable using SAT-solving techniques.

2 Problem Formulation

2.1 Parameters

We define the warehouse as a *two-dimensional grid* of size $n \times m$, denoted as:

$$G = \{(x, y) \mid 0 \leq x < n,\ 0 \leq y < m\}$$

where each cell in G represents a valid position in the warehouse.

2.2 Robots

A set of robots

$$R = \{r_1, r_2, \ldots, r_k\}$$

operates in the grid. Each robot r has:

- A **start position** $(x_{\text{start}_r}, y_{\text{start}_r})$
- A **goal position** $(x_{\text{goal}_r}, y_{\text{goal}_r})$

2.3 Time Horizon

We introduce a fixed maximum time T, representing the upper bound on the number of time steps a robot can take to reach its goal. The problem is then solved over a discrete sequence of time steps $t \in \{0, 1, \ldots, T\}$.

2.4 Variables

To model the problem as a SAT instance, we define the following Boolean variables:

- $P(r, x, y, t)$: **True** if robot r is at position (x, y) at time t.
- $O(x, y)$: **True** if there is an obstacle at position (x, y).

2.5 Constraints

2.5.1 Robot Can Only Be at One Cell At a Time

This constraint ensures that each robot occupies only a single cell at any given time step.

$$\forall r \in R,\ \forall t \in \{0, \ldots, T\},\ \forall x, y \in G,\ \forall x', y' \in G; (x, y) \neq (x', y') : \quad P(r, x, y, t) \implies \neg P(r, x', y', t)$$

2.5.2 Movement Restricted to Free Cells

Robots can only move to cells that are not obstacles.

$$\forall x, y \in G,\ \forall r \in R,\ \forall t \in [0, T] : \quad O(x, y) \implies \neg P(r, x, y, t)$$

2.5.3 Movement to Adjacent Cells Only

Robots can only move to adjacent cells or remain stationary at each time step.

$$\forall r \in R,\ \forall x, y, t : \quad P(r, x, y, t) \implies \bigvee_{(\Delta x, \Delta y) \in M} P\Big(r, \text{clamp}(x{+}\Delta x, 0, n{-}1), \text{clamp}(y{+}\Delta y, 0, m{-}1), \text{clamp}(t{+}1, 0, T)\Big)$$

- The function `clamp(x, 0, n)` restricts the value of `x` within the inclusive range from 0 to n. If `x` is less than 0, the function returns 0. If `x` is greater than n, the function returns n. Otherwise, it returns `x`.
- Here,
$$M = \{(0, 1), (1, 0), (0, 0), (-1, 0), (0, -1)\}$$
represents valid movements (up, right, stay, left, down).

2.5.4 Collision Avoidance

Two robots cannot occupy the same cell at the same time.

$$\forall r, r' \in R,\ r \neq r',\ \forall x, y, t : \quad P(r, x, y, t) \implies \neg P(r', x, y, t)$$

2.5.5 Position Switching Prohibition

Two robots cannot swap positions between consecutive cell (x, y) at time t where robot r is located. time steps.

$$\forall r, r' \in R,\ r \neq r',\ \forall x, y, t,\ \forall (\Delta x, \Delta y) \in M \setminus \{(0, 0)\} :$$
$$P(r, x, y, t) \wedge P(r', x + \Delta x, y + \Delta y, t) \implies \neg\Big(P(r, x + \Delta x, y + \Delta y, t + 1) \wedge P(r', x, y, t + 1)\Big)$$

2.6 Objective

The goal is to compute a valid set of paths for all robots such that:

- All robots reach their designated goals within the time horizon T.
- All constraints are satisfied.
- (Optional) The total path length or time taken is minimized.

3 Results

3.1 Basic Scenario

Consider a simple warehouse scenario with three robots, as shown in Figure 1. The blue robot starts at the top left and needs to reach the bottom right, the green robot starts at the bottom right and needs to reach the top left, and the red robot starts at the bottom left and needs to reach the top right. The time horizon is 10.

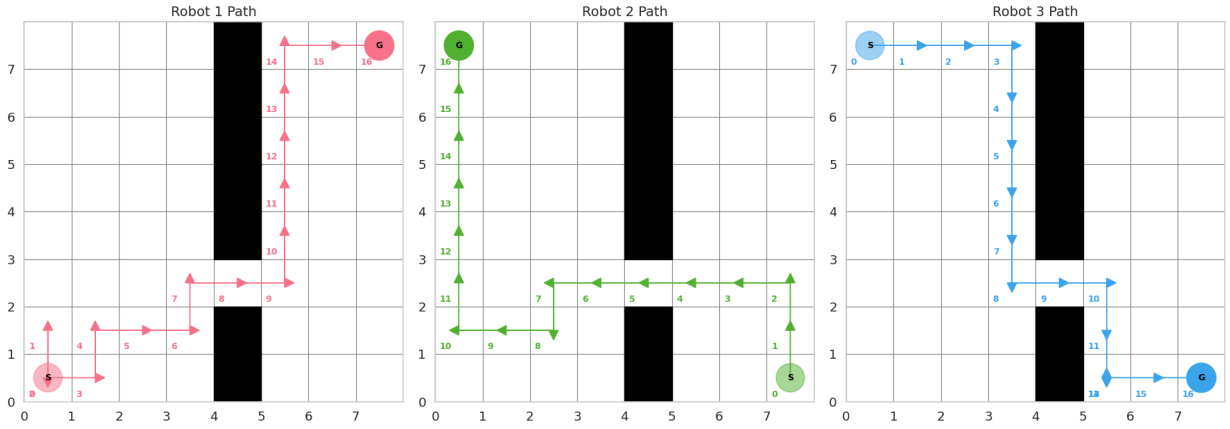

Figure 1: A solution for a simple warehouse scenario with three robots. The blue robot starts at the top left and needs to reach the bottom right, the green robot starts at the bottom right and needs to reach the top left, and the red robot starts at the bottom left and needs to reach the top right. The time horizon is 10.

4 Implementation Details

4.1 Classes and Data Structures

Position

- **Attributes:**
  - `x`: Integer representing the x-coordinate.
  - `y`: Integer representing the y-coordinate.
- **Methods:**
  - `_hash_`: Returns a hash value based on the coordinates.
  - `_eq_`: Checks equality between two `Position` instances.

Robot

- **Attributes:**
  - `id`: Unique identifier for the robot.
  - `start`: `Position` object representing the starting position.
  - `goal`: `Position` object representing the goal position.

4.2 WarehousePathPlanner Class

Attributes:

- `width`: Width of the warehouse grid.
- `height`: Height of the warehouse grid.
- `time_horizon`: Maximum time steps allowed.
- `cnf`: CNF formula to store constraints.
- `var_map`: Mapping of variable names to integers.
- `next_var`: Next available variable number.
- `moves`: List of valid movements (stay, up, right, down, left).

Methods:

- `create_variable(robot_id, x, y, t)`: Creates a variable representing a robot's position at a given time.
- `add_initial_positions(robots)`: Adds constraints for the initial positions of robots.
- `add_goal_positions(robots)`: Adds constraints for the goal positions of robots.
- `add_obstacle_constraints(obstacles, robots)`: Adds constraints to prevent robots from occupying obstacle positions.
- `add_movement_constraints(robots)`: Adds constraints for valid movements between time steps.
- `add_collision_avoidance(robots)`: Adds constraints to prevent robots from occupying the same position.
- `add_position_switching_prohibition(robots)`: Adds constraints to prevent robots from switching positions.
- `add_implication(antecedent, consequents)`: Adds implication constraints to the CNF formula.
- `solve()`: Solves the SAT problem and returns the solution if one exists.
- `decode_solution(solution)`: Converts the SAT solution to robot paths.

4.3 Code Commentary

Imports and Data Structures

```
import itertools
from dataclasses import dataclass
from typing import Dict, List, Optional, Set

from pysat.formula import CNF
from pysat.solvers import Glucose3
```

Interpretation:

- `itertools`: Provides functions for efficient looping, useful for generating combinations.
- `dataclasses`: Simplifies the creation of classes that primarily store data.
- `typing`: Provides support for type hints, enhancing code readability and maintainability.
- `pysat.formula` and `pysat.solvers`: Used for creating and solving SAT problems.

Position Class

```python
@dataclass(frozen=True)
class Position:
    x: int
    y: int

    def _hash_(self):
        return hash((self.x, self.y))

    def _eq_(self, other):
        if not isinstance(other, Position):
            return NotImplemented
        return self.x == other.x and self.y == other.y
```

Interpretation:

- Represents a position in the grid with `x` and `y` coordinates.
- `frozen=True` makes instances immutable and hashable, allowing them to be used as keys in dictionaries.
- `_hash_` and `_eq_` enable comparison and hashing of `Position` objects.

Robot Class

```python
@dataclass
class Robot:
    id: int
    start: Position
    goal: Position
```

Interpretation:

- Represents a robot with a unique `id`, a starting `Position`, and a goal `Position`.

4.4 WarehousePathPlanner Class Overview

The `WarehousePathPlanner` class encapsulates the logic for modeling the multi-robot path planning problem as a SAT instance. Key aspects include:

Initialization

```python
def __init__(self, width: int, height: int, time_horizon: int):
    self.width = width
    self.height = height
    self.time_horizon = time_horizon
    self.cnf = CNF()
    self.var_map = {}
    self.next_var = 1
    # Valid movements: stay in place, move up, right, down, or left.
    self.moves = [(0, 0), (0, 1), (1, 0), (0, -1), (-1, 0)]
```

Interpretation:

- The planner is aware of the grid dimensions and the available time steps.
- The CNF object and variable mappings are initialized for building the SAT instance.
- The allowed movements include staying in place along with directional changes.

Creating Variables

```
def create_variable(self, robot_id: int, x: int, y: int, t: int) -> int:
    key = f"R{robot_id}X{x}Y{y}T{t}"
    if key not in self.var_map:
        self.var_map[key] = self.next_var
        self.next_var += 1
    return self.var_map[key]
```

Interpretation:

- Generates a unique variable for each robot, position, and time configuration.
- The descriptive key (e.g., `"R1X2Y3T4"`) supports traceability in the SAT variables.

Adding Initial Position Constraints

```
def add_initial_positions(self, robots: List[Robot]):
    for robot in robots:
        start_var = self.create_variable(robot.id, robot.start.x, robot.start.y, 0)
        self.cnf.append([start_var])
        for x in range(self.width):
            for y in range(self.height):
                var = self.create_variable(robot.id, x, y, 0)
                if x != robot.start.x or y != robot.start.y:
                    self.cnf.append([-var])
```

Interpretation:

- The robot is fixed at its starting position at time 0 via a positive clause.
- All other cells at time 0 are negated, enforcing that the robot can only start at its designated location.

Adding Goal Position Constraints

```
def add_goal_positions(self, robots: List[Robot]):
    for robot in robots:
        goal_var = self.create_variable(robot.id, robot.goal.x, robot.goal.y, self.
            time_horizon)
        self.cnf.append([goal_var])
        for x in range(self.width):
            for y in range(self.height):
                if x != robot.goal.x or y != robot.goal.y:
                    var = self.create_variable(robot.id, x, y, self.time_horizon)
                    self.cnf.append([-var])
```

Interpretation:

- Ensures each robot is at its designated goal at the final time step.
- Prohibits the robot from being in any other cell at time T.

Adding Obstacle Constraints

```
def add_obstacle_constraints(self, obstacles: Set[Position], robots: List[Robot]):
    for obstacle in obstacles:
        if not (0 <= obstacle.x < self.width and 0 <= obstacle.y < self.height):
            raise ValueError("Obstacle is outside the warehouse bounds")
        for robot in robots:
            for t in range(self.time_horizon + 1):
                var = self.create_variable(robot.id, obstacle.x, obstacle.y, t)
                self.cnf.append([-var])
```

Interpretation:

- Prevents any robot from occupying cells designated as obstacles at any time step.

Adding Movement Constraints

```python
def add_movement_constraints(self, robots: List[Robot]):
    for robot in robots:
        for t in range(self.time_horizon + 1):
            positions_at_t = []
            for x in range(self.width):
                for y in range(self.height):
                    var = self.create_variable(robot.id, x, y, t)
                    positions_at_t.append(var)
            self.cnf.append(positions_at_t)
            for pos1, pos2 in itertools.combinations(positions_at_t, 2):
                self.cnf.append([-pos1, -pos2])
            if t < self.time_horizon:
                for x in range(self.width):
                    for y in range(self.height):
                        current_pos = self.create_variable(robot.id, x, y, t)
                        next_positions = []
                        for dx, dy in self.moves:
                            next_x, next_y = x + dx, y + dy
                            if 0 <= next_x < self.width and 0 <= next_y < self.height:
                                next_pos = self.create_variable(robot.id, next_x, next_y, t
                                    + 1)
                                next_positions.append(next_pos)
                        if next_positions:
                            self.add_implication(current_pos, next_positions)
```

Interpretation:

- Ensures that at every time step a robot occupies exactly one cell.
- Enforces that if a robot is at a given position at time t, then it moves to one of the valid positions at time $t + 1$.

Adding Collision Avoidance Constraints

```
def add_collision_avoidance(self, robots: List[Robot]):
    for t in range(self.time_horizon + 1):
        for x in range(self.width):
            for y in range(self.height):
                robot_vars = []
                for robot in robots:
                    var = self.create_variable(robot.id, x, y, t)
                    robot_vars.append(var)
                for var1, var2 in itertools.combinations(robot_vars, 2):
                    self.cnf.append([-var1, -var2])
```

Interpretation:

- Prevents more than one robot from occupying the same cell at the same time.

Adding Position Switching Prohibition

```
def add_position_switching_prohibition(self, robots: List[Robot]):
    for t in range(self.time_horizon):
        for x1 in range(self.width):
            for y1 in range(self.height):
                for dx, dy in self.moves[1:]:
                    x2, y2 = x1 + dx, y1 + dy
                    if 0 <= x2 < self.width and 0 <= y2 < self.height:
                        for r1, r2 in itertools.combinations(robots, 2):
                            r1_pos1_t = self.create_variable(r1.id, x1, y1, t)
                            r2_pos2_t = self.create_variable(r2.id, x2, y2, t)
                            r1_pos2_t1 = self.create_variable(r1.id, x2, y2, t + 1)
                            r2_pos1_t1 = self.create_variable(r2.id, x1, y1, t + 1)
                            self.cnf.append([-r1_pos1_t, -r2_pos2_t, -r1_pos2_t1, -
                                r2_pos1_t1])
```

Interpretation:

- Prevents two robots from swapping positions between consecutive time steps.

Adding Implication Constraints

```
def add_implication(self, antecedent: int, consequents: List[int]):
    self.cnf.append([-antecedent] + consequents)
```

Interpretation:

- This helper function expresses the implication that if the antecedent is true, then at least one of the consequents must be true.

Solving and Decoding the SAT Problem

```
def solve(self) -> Optional[Dict[str, bool]]:
    with Glucose3() as solver:
        solver.append_formula(self.cnf.clauses)
        if solver.solve():
            model = solver.get_model()
            if not isinstance(model, list):
                raise ValueError("No model found")
            return {
                var_name: lit > 0
                for var_name, var_num in self.var_map.items()
                for lit in model
                if abs(lit) == var_num
            }
        return None
```

```python
def decode_solution(self, solution: Dict[str, bool]) -> Dict[int, List[Position]]:
    paths = {}
    for var_name, is_true in solution.items():
        if is_true:
            numbers = "".join(c if c.isdigit() else " " for c in var_name).split()
            robot_id, x, y, t = map(int, numbers)
            if robot_id not in paths:
                paths[robot_id] = [None] * (self.time_horizon + 1)
            paths[robot_id][t] = Position(x, y)
    for path in paths.values():
        if None in path:
            raise ValueError("Invalid solution: incomplete path detected")
    return paths
```

Interpretation:

- The `solve` function uses the Glucose3 solver to obtain a satisfying assignment.
- The `decode_solution` function reconstructs robot paths from the model, ensuring each time step is accounted for.

Decoding the SAT Solution

```python
def decode_solution(self, solution: Dict[str, bool]) -> Dict[int, List[Position]]:
    paths = {}
    for var_name, is_true in solution.items():
        if is_true:
            numbers = "".join(c if c.isdigit() else " " for c in var_name).split()
            robot_id, x, y, t = map(int, numbers)
            if robot_id not in paths:
                paths[robot_id] = [None] * (self.time_horizon + 1)
            paths[robot_id][t] = Position(x, y)
    for path in paths.values():
        if None in path:
            raise ValueError("Invalid solution: incomplete path detected")
    return paths
```

Interpretation:

- *Solution Decoding:* This method parses each true SAT variable (e.g., `"R1X2Y3T4"` corresponds to robot 1 being at position $(2, 3)$ at time 4) from the solution.
- *Reconstructing Paths:* It reconstructs the complete path for each robot over the entire time horizon.
- *Completeness Check:* The method ensures that every time step contains a valid position. If any time step is missing, it raises an error indicating an incomplete path.