Thymio the builder

Henrique Nascimento
Student 66617
fc66617@alunos.fc.ul.pt

Daniel Malheiro
Student 66461
fc66461@alunos.fc.ul.pt

***Abstract*—This project implements an autonomous system for the Thymio robot to locate and transport blocks. Using an A-Star based path-finding algorithm and a three layer architecture, Jupyter, Python, and ASEBA, the robot plans its path and executes movements reliably from start to target.**

I. Introduction

This project consists of an autonomous control system for Thymio to perform sequential block manipulation tasks. Taking inspiration from the Sokoban game, the core goal is to move blocks from a pre-known starting location to a predefined target location. The project is structured into three progressive stages, focusing first on moving precision and error mitigation in straight-line movement, advancing to path finding and block management, and finally, the assembly of multiple blocks into a structure (for example, the letter ”L”).

II. Approach

A. Phase one

This phase focuses on achieving accurate straight-line motion, in both the x and y directions, while pushing a single block.

To accomplish this, an odometry system for the Thymio was developed using functions that measure the distance traveled, the angle, and the displacements. The goal was to compute the distance covered in both the x and y coordinates, in centimeters. However, this approach produced several errors. The most significant issue was the inaccuracy in the measured distance: the difference between the real distance traveled and the distance estimated by the odometry system was too large. This level of error was unacceptable, as the Thymio needed to know its exact position to perform path-finding and reach the block reliably. To address this, a second odometry system was created, based not on centimeters but on a grid in which each square has the exact size of the Thymio robot. Instead of providing coordinates in centimeters, the robot is given a grid location, for example, the coordinate (2, 5) corresponds to two squares in the x direction and five squares in the y direction. With this approach, the errors were significantly reduced because the robot always moves a fixed amount of distance per square (12 cm), ensuring consistent and reliable motion.

The second part of this phase involved making the Thymio push a block in a straight line. Initially, a PID controller was created using the front sensors, prox.horizontal[1] and prox.horizontal[3]. This was necessary because the front of the robot is curved, so a correction mechanism was required to prevent the block from sliding sideways. The PID controller automatically adjusted the robot's pushing angle by comparing the distances measured by the two sensors. If one sensor detected the block at a shorter distance than the other, the robot would slightly turn to correct its alignment. However, this approach presented several problems. The PID often over-corrected, and each correction to one side was followed by a correction to the opposite side, causing oscillatory behavior. As a result, the push became very inaccurate. To solve this, the Thymio was used in reverse: instead of pushing with the curved front, it pushed with the back side, which is flat. This made the pushing motion more stable and ensured a uniform, straight-line movement.

B. Phase two

The second phase focused on the “brain” of the Thymio: the path-finding system and the instructions required for it to reach and push the block.

The approach is straightforward. The system is given the coordinates of the blocks and their destinations. Using the A* algorithm, the optimal path is then computed. The path-finding algorithm was designed to penalize paths with many turns in order to reduce potential errors.

Once the path was determined, the system translated it into a sequence of instructions, which included the following:

- F - Move one cell forwards
- TR - Turns right in the same cell
- TL - Turns left in the same cell
- B - Move one cell backwards

Based on the known coordinates, the system first determines whether the block needs to move north, south, east, or west so the robot can position itself on the opposite side. For example, if the block must move north, the robot positions itself to the south of the block.

Next, the path to the block is calculated using the algorithm described earlier. Once the robot reaches the block, a second path-finding step is executed this time to determine the path from the block's current position to its target location.

During this transportation phase, if a turn is required, the robot must perform a “dance,” moving around the block until it reaches the correct side from which to push it toward the desired direction.

The final result would be, for example, if thymio starts in position (0,0) facing east and a block is placed in position

(2,2) and the target position is (3,3), the array sent to the robot would be:

- Array to reach the block - [F, F, TR, F]
- Array to push the block - [F, TR, F, TL, F, TL, F]

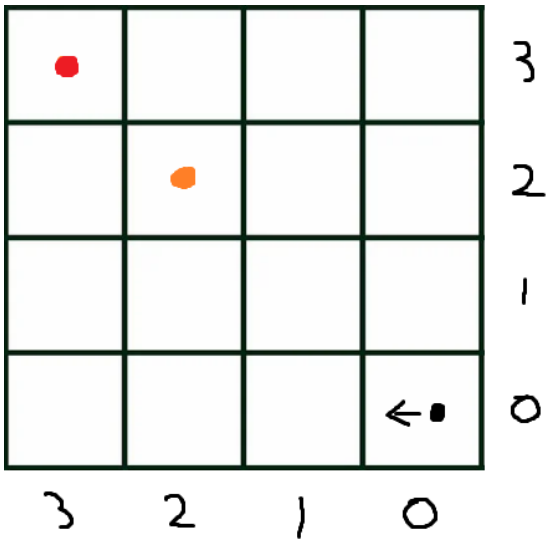

Fig. 1. Path example

C. Phase three

The third and final phase was to connect both systems, the "brain" and the odometry. The idea was to have the brain system (running in Python) handle the instructions that the Thymio must follow, while the odometry system (running in ASEBA) handled the execution of those instructions.

III. Implementation

A. System Architecture

The architecture of the system is organized into three layers, the Jupyter Notebook, the Python ("Brain"), and the ASEBA execution layer running on the robot. The Jupyter Notebook acts as the main controller of the entire workflow, acting as the interface through which the robot is accessed via tdmclient. When execution begins, the notebook triggers the Python module responsible for all high-level computation, including pathfinding, phase control, and generation of the final movement instructions. Python processes the known coordinates of the block and target location, computes the necessary paths for each phase of the task, and encodes the resulting actions into a numerical instruction array. This array is then sent back to the notebook, which transmits it to ASEBA on the robot. ASEBA interprets these integer instructions and executes them directly through low-level motor functions. This three-layer architecture cleanly separates planning from execution, ensuring that Python handles all computationally heavy decision-making while ASEBA focuses on performing the movements reliably on hardware.

This makes the communication flow the following:
Jupyter → Python → Jupyter → ASEBA → Robot

B. Python as the Brain

Python works as the intelligence layer of the system, responsible for all high-level decision-making and path computation. Given pre-known coordinates for the block's location and its target destination, Python executes a custom A-Star pathfinding algorithm designed to penalize each curve, resulting in smoother and more efficient navigation paths for the robot. The computation is organized into three phases, **Approach**, where Python calculates a path to bring the robot close to the block, **Docking**, where a PID is executed to align the robot with the block, and **Transport**, where a new path is generated to move the block to its final position. After completing these calculations, Python translates the resulting motions into a sequence of integer-coded instructions, with each number representing a specific robot action such as moving forward or turning. This array of instructions is then returned to the Jupyter Notebook for further transmission to ASEBA.

On a higher level of planning, it has a Block Manager that keeps track of the current block's position and goal position, with the intent of forming the final structure.

```
grid = GridMap(width_cells=20,
    height_cells=15, cell_size=50)

   # Define final positions structure
   structure_plan = {
      "cube1": (4, 3),
      "cube2": (2, 1),
      "cube3": (5, 5),
   }


   global block_manager, planner, action_queue


   block_manager = BlockManager()
   block_manager.add_block("cube1", Block(2,
       2, 1, 1))
   block_manager.add_block("cube2", Block(2,
       0, 1, 1))

   planner = PathPlanner(grid)
   action_queue = ActionQueue()
...
      mission1_queue =
          planner.generate_mission(robot_start,
          robot_angle, block1_start,
          block1_goal)
```

C. ASEBA as the Execution

ASEBA serves as the robot's execution layer, responsible for interpreting and performing the low-level actions generated by Python. Once the Jupyter Notebook transmits the instruction array, ASEBA processes it sequentially, mapping each integer value to a specific motor behavior. For example, one instruction may correspond to moving forward, while another triggers a right turn or a left turn. ASEBA's role is just operational, it does not compute paths or make decisions, but instead

focuses on executing the actions exactly as defined by Python. This separation ensures that the robot's movements remain predictable and consistent, with ASEBA providing the precise motor control and sensor interaction needed to bring the high-level plan to action. Through this mechanism, ASEBA acts as the final step in the workflow.

```
onevent action_queue

   ACTION_QUEUE[0] = event.args[0]
   ACTION_QUEUE[1] = event.args[1]
   ACTION_QUEUE[2] = event.args[2]
   ACTION_QUEUE[3] = event.args[3]
   ACTION_QUEUE[4] = event.args[4]
   ACTION_QUEUE[5] = event.args[5]
   ACTION_QUEUE[6] = event.args[6]
   ACTION_QUEUE[7] = event.args[7]
   ACTION_QUEUE[8] = event.args[8]
   ACTION_QUEUE[9] = event.args[9]

      on_performed = 0
      queue_index = 0 # reset consumption
          pointer

   callsub execute_next_action

sub execute_next_action
   act = ACTION_QUEUE[queue_index]

   if act == 0 then
      # 0 = no action, skip
      queue_index += 1
      return
   end

   if act == 1 then callsub move_one_cell end
   if act == 2 then callsub
       move_half_cell_forward end
   if act == 3 then callsub
       move_half_cell_back end
   if act == 4 then callsub rotate_left end
   if act == 5 then callsub rotate_right end
   if act == 6 then callsub move_cell_back
       end
   if act == 7 then callsub
       align_with_block_pid end
   if act == 8 then callsub move_to_block end

   queue_index += 1
```

D. Workflow

The overall workflow follows a clear and structured pipeline that connects high-level planning with low-level execution:

1) User executes notebook cell
2) Notebook connects to robot using tdmclient
3) Notebook calls Python "brains"
4) Python computes paths (Approach → Docking → Transport)
5) Python returns an integer instruction array
6) Notebook sends array to ASEBA
7) ASEBA iterates through instructions and drives the robot
8) Block moves from A → B

E. Communication Callbacks

Lastly, our previous tests and implementations made us come to the conclusion that time-based events weren't reliable due to Thymio's inconsistent performance times and communication delays. This would often lead to sending an array of actions before the current one executed fully. To mitigate this, internally in ASEBA there's an `on_performed` flag that indicates when an array is fully executed and the Thymio is ready to receive another one; it can also let the Python brain know if an unexpected occurrence happened. Additionally, with this implementation we also now split the action list/arrays into several 10 position chunks, and the Thymio then consumes them and requests them dynamically, thus keeping a good balance between speed and accuracy in instructions.

```
PRIMITIVE_MAP = {
   "F": [6],      # Forward
   "B": [1],      # Backwards full
   "HF": [2],     # Half forward
   "HB": [3],     # Half backwards
   "TL": [2, 5, 3], # Turn left = half
       forward + rotate left + half back
   "TR": [2, 4, 3], # Turn right = half
       forward + rotate right + half back
   "AB": [7, 8],  # Align Block + Approach
       Block
}


def expand_actions(action_list):
   expanded = []
   for act in action_list:
      if act not in PRIMITIVE_MAP:
         print("Unknown action:", act)
         continue
      expanded.extend(PRIMITIVE_MAP[act])
   return expanded

EVENT_SIZE = 10

def make_payloads(action_list):
   expanded = expand_actions(action_list)
   payloads = []
   for i in range(0, len(expanded),
       EVENT_SIZE):
      chunk = expanded[i:i+EVENT_SIZE]
      chunk += [0] * (EVENT_SIZE -
          len(chunk)) # pad if shorter
      payloads.append(chunk)
   return payloads

async def
    send_action_queue_event(action_list):
   payloads = make_payloads(action_list)
   for idx, payload in enumerate(payloads):
      print(f"Sending payload
          {idx+1}/{len(payloads)}:", payload)
      await node.send_events({"action_queue":
          payload})
      # wait for robot confirmation before
          next payload
      wait_until_performed(var_name="on_performed",
          timeout=20)
```

In this snippet, it is notable how turning right or left actually requires three ASEBA actions in order to keep the Thymio centered in it's cell. This quick growth in array length was another reason why we needed to be able to split it into several payloads.

IV. Results

Presentation and critical analysis of the results obtained

A key observation worth noting is that initial conditions are among the most decisive factors in the success of our tests. Any minimal offset in the Thymio or the cube's starting positions could completely throw off the whole procedure. This occurred despite our attempts to make the robot more adaptable to change, for example by aligning itself with a block before starting to push.

An additional complication arose from the lack of precision in our cube prototype, which was made from less firm cardboard. The waviness on the surface caused the PID alignment to fail, as the sensor data regarding the distance to the cube became inaccurate.

Each movement of the robot introduces some error, so turning motions were particularly error-prone since they require three distinct actions to keep the robot centered.

A more ambitious test involved moving a cube from the $(2, 2)$ position to the $(4, 3)$ position, which requires approaching it from different sides. Our results were as follows:

The Thymio starts aligned with the grid and begins its journey.

When it reaches the cube and begins the docking process, a notable offset occurs, causing one of the sensors to miss the cube. This results in incorrect alignment.

This offset unfortunately causes the rest of the route to deviate from the planned trajectory.

The Jupyter console output allows a deeper look at what was supposed to happen and how the communication was actually handled:

```
Generating mission from (2, 2) to (4, 3)...
--- PLANNING MISSION ---
Robot: (0, 0) facing 90
Block: (2, 2) -> (4, 3)
```

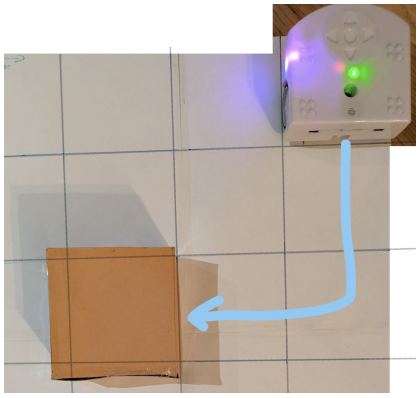

Fig. 2. Initial conditions

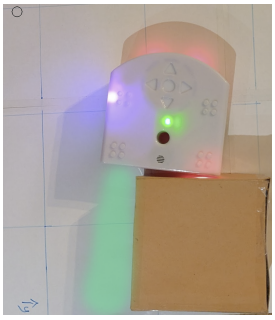

Fig. 3. Thymio aligned with cube

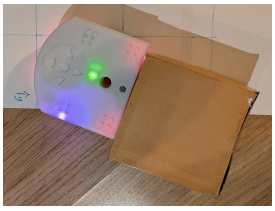

Fig. 4. Final position after misalignment

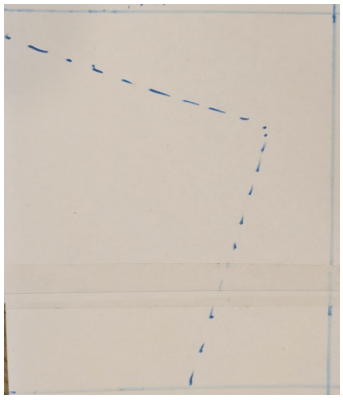

Fig. 5. Traced position vs. expected position. The dotted line indicates the measured position, and the solid line indicates the expected path.

```
Block Path: [(2, 2), (3, 2), (4, 2), (4, 3)]
Phase 1 (Approach): 4 moves
Phase 2 (Transport): 8 moves
Final Command Queue:
F -> F -> TL -> F -> AB -> F -> F -> TL -> F -> TR -> F -> TR -> F
Sending payload 1/3: [6, 6, 2, 5, 3, 6, 7, 8, 6, 6]
 Timeout waiting for action_queue to finish
Sending payload 2/3: [2, 5, 3, 6, 2, 4, 3, 6, 2, 4]
 Action queue finished!
Sending payload 3/3: [3, 6, 0, 0, 0, 0, 0, 0, 0, 0]
 Action queue finished!
```

Unfortunately, at this time we were unable to experiment with complex structures as initially intended. However, some of our proof-of-concept implementations were successful, for example the pathfinding, planning, and movement-to-feedback cycle.

V. Final comments

This system showcases the potential of autonomous robots to perform structured tasks accurately, which can be applied in education, prototyping, and automation research.

This project still has some limitations. One notable issue is the connectivity to the robot. Connections often failed, or the wireless communication was too slow, causing delays in the robot's actions. In a project like this, precision is crucial.

Possible improvements include enabling the Thymio to assemble any structure specified by a user through a web interface, as well as the ability to build structures vertically by using ramps to place blocks at different heights.