THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

COMP7702 - Artificial Intelligence

ASSIGNMENT 1 - REPORT

Team: **BotMate**

- 45510760 - Wenhui Han
- 45270916 - Canggih Pramono Gultom
- 44907635 - Vu Anh Le

Table of contents

1 . Agent Design Problem

As the assignment specification described, an agent that can move one or multiple moving-boxes to their respective goal positions is required to be designed and implemented. The environment type can be defined as below:

- **Continuous:** The workspace is continuous space for agent/box to travel, the robot can move to any position in the 2D environment of size [0,1] X [0,1].
- **Deterministic:** Once a move is made, the agent understands exactly what the next state would be.
- **Fully observable:** The agent fully possesses information about the environment.
- **Static:** No external force is interrupting the environment and agent is the only cause of any state change.

The agent design problem can be defined in aspects as per below:

1.1. State Space

State space of the problem includes the position and the orientation of the robot, the positions of moving-boxes and movable obstacles.

The configuration of the robot can be described as (x_r, y_r, Θ_r)

The position of a moving box and movable obstacle can be described as (x_{mb}, y_{mb}) and (x_{mo}, y_{mo})

For example, the state space of the environment with 2 moving-boxes and 2 movable obstacles can be represented as:

$$\mathbf{S} = (x_r, y_r, \Theta_r, x_{mb1}, y_{mb}, x_{mb2}, y_{mb2}, x_{ob1}, y_{ob1}, x_{ob2}, y_{ob2})$$

1.2. Action Space

The robot can move to any point in the environment and rotate to any angle through a sequence of primitive actions. Each primitive action allows a travelling distance of at most 0.001 unit. which can be represented as:

$$\mathbf{A}: (\Delta x_r, \Delta y_r, \Delta \Theta)$$

Then the robot config transition is defined by: $(x_1, y_1, \Theta_1) \rightarrow (x_2, y_2, \Theta_2)$

Note: If the robot is coupled with a moving-box or movable obstacle with ¾ of its length coincided with one of the object's sides, it is able to push that objects only in horizontal or vertical direction. The robot is only allowed to move if next state is collision-free.

1.3. Percept Space

The environment is fully observable, so the percept space is same in state space.

1.4. World Dynamics

Once robot is coupled with a moving-box or movable obstacle with ¾ of its length coincided with one of the object's sides, it is able to push that moving-box/obstacle horizontally or vertically. For instance, if the robot coincides with the left side of a moving-box, it will be able to push the box to the right horizontally.

Move robot from one position (rotation Θ_1) to another position (Rotation Θ_2)

State: $\mathbf{S_1} = (x_{r1}, y_{r1}, \Theta_1, x_{mb1}, y_{mb1}, x_{mb2}, y_{mb2} \dots x_{mbi}, y_{mbi}, x_{mo1}, y_{mo1}, x_{mo2}, y_{mo2} \dots x_{moj}, y_{moj})$

Action: $\mathbf{A_1} = (\Delta x_r, \Delta y_r, \Delta \Theta)$

Transition can be described as: $\mathbf{T_1} : \mathbf{S_1} \text{ X } \mathbf{A_1} \rightarrow \mathbf{S_2}$

While $\mathbf{S_2} = (x_{r2}, y_{r2}, \Theta_2, x_{mb1}, y_{mb1}, x_{mb2}, y_{mb2} \dots x_{mbi}, y_{mbi}, x_{mo1}, y_{mo1}, x_{mo2}, y_{mo2} \dots x_{moj}, y_{moj})$

When an robot pushes moving-box from one position to another position:

State: $\mathbf{S_1} = (x_{r1}, y_{r1}, \Theta, x_{mb1}, y_{mb1}, x_{mb2}, y_{mb2} \dots x_{mbi}, y_{mbi}, x_{mo1}, y_{mo1}, x_{mo2}, y_{mo2} \dots x_{moj}, y_{moj})$

Action: $\mathbf{A_2} = (\Delta x_r, \Delta y_r, 0)$

Assume the robot is pushing the first moving-box, and the position of this moving-box changed to (x'_{mb1}, y'_{mb1}) then the transition can be described as: $\mathbf{T_2} : \mathbf{S_1} \text{ X } \mathbf{A_2} \rightarrow \mathbf{S_2}$

While $\mathbf{S_2} = (x_{r2}, y_{r2}, \Theta, x'_{mb1}, y'_{mb1}, x_{mb2}, y_{mb2}, \dots x_{mbi}, y_{mbi}, x_{mo1}, y_{mo1}, x_{mo2}, y_{mo}, \dots x_{moj}, y_{moj})$

1.5. Percept Function

In this task , perception is always fully observable, a percept function is not required.

1.6. Free Space

Free space here is defined as the set of all configurations of robot, moving-boxes and movable obstacles that is collision free.

1.7. Utility Function

Since the robot is supposed to push the moving-box to to the goal with shortest path. The utility function can be represented in the form of utility by assigning the robot disutility, proportional to the distance that the moving-boxes need to travel through to the goal.

In other words, a utility could be measured by decrease of cost from one node(n) to the goal position for one moving-box. Cost is calculated as:

$$\mathbf{f(n) = g(n) + h(n)}$$

- **g(n)**: Cost from an initial position of the moving-box to node n.
- **h(n)**: Estimated cost (by Manhattan distance) from node n to the goal position.

To maximize an utility function, the agent is required to select a successor node with the lowest **f(n)**

2. Search Method at Conceptual Level

2.1. Sub-problems

The problem is divided into three sub-problems with respective search methods.

2.1.1. Move moving-box from initial position to its respective goal

The robot pushes moving-boxes to their respective goal positions one by one. The box path planning includes frontier expansion and choosing next node based on its f(n). A typical expansion step is represented in the graph below:

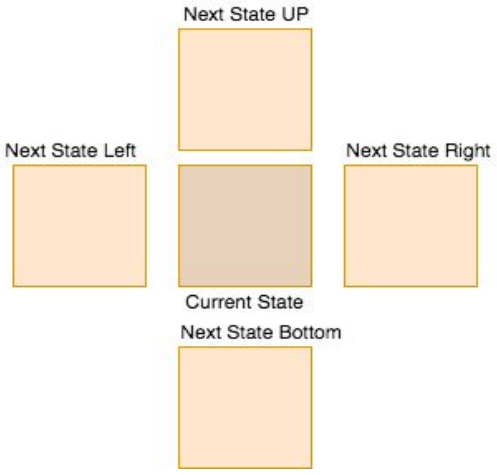


Figure 2.1.1.1: The new state will be constructed on the left, right, up and bottom of a current state

Based on figure 2.1.1.1, the agent will construct 4 possible states in four directions, left, right, up and down of current rectangle position until a possible state reaching the goal state. Every state will be tested by collision check, and feasible positions will be ranked and prioritised for expansion. The moving-box path planning search method can be represented as below:

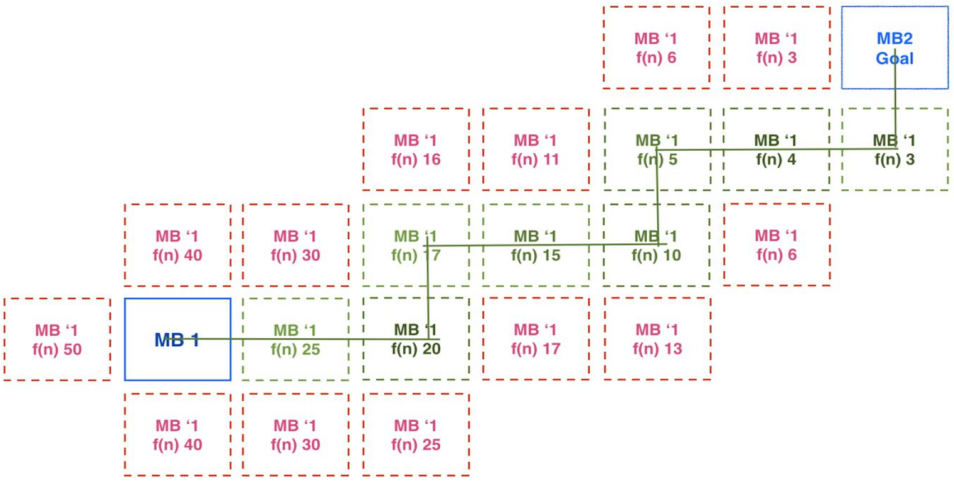


Figure 2.1.1.2: Search using A* by expanding frontiers

Using A*, the agent will find the shortest path to the goal based on **f(n),** which is the sum of (**MovingCost** + **RobotCost** + **ObstacleCost**) and estimated cost by Manhattan distance from current node to the goal position. The details of f(n) will be explained in section 2.2.

The agent grows all objects in the workspace by MAX_ERROR(minimum distance between objects that is collision-free) and checks whether the grown rectangles intersects with the next state of moving-box. The agent also checks if there is collision between objects and agent itself, and whether a successor node is colliding with workspace boundary.

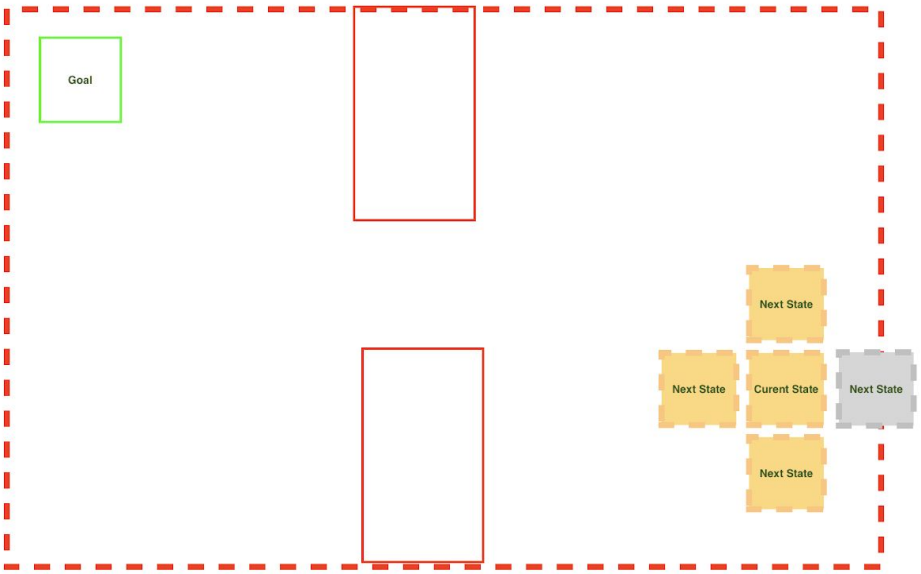


Figure 2.1.1.3: Checking collision with boundary

2.1.2: Move movable obstacle away from the planned path

The agent always tries to find the path that has no obstacles first (Figure 2.1.2.1) because the cost of movable obstacles is included in the total moving cost. However, if there is only a path that has movable obstacles on it, the agent will try to move the movable obstacles away from the path. For example, in the figure 2.1.2.2, the agent will try to move the MO1 and MO2 out of the path to the goal (to MO1' and MO2' position)

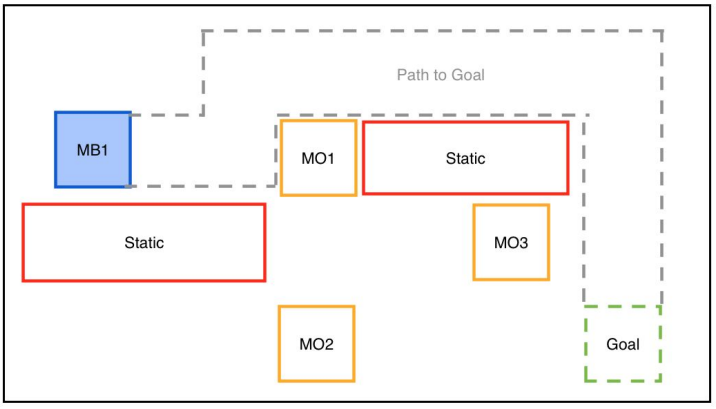


Figure 2.1.2.1: Agent find the path that has no obstacles first

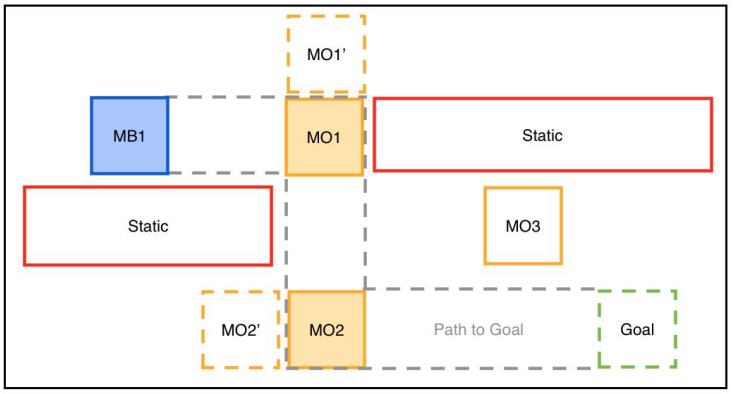


Figure 2.1.2.2: Move movable obstacle away from the planned path

2.1.3. Move the robot to the objects

Before starting pushing any moving-boxes or movable obstacles, the robot needs to place itself in a "coupled" position next to the target object. To discover a path for this, the agent will project a virtual straight line to its desired position, and mark every obstacle that intersects with this virtual line. Afterwards, the agent takes a few samples around the marked obstacles as vertices and creates a state graph by connecting collision-free vertices. The search method is A*, which stores vertices and their respective f(n) (f(n)=g(n)+h(n)) in the priority queue.

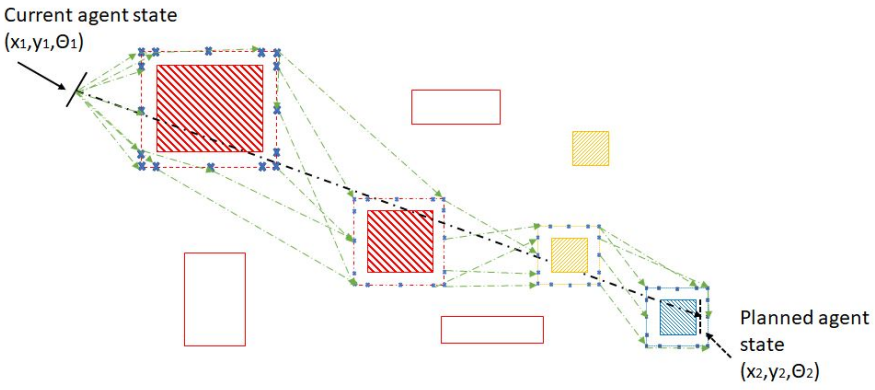


Figure 2.1.3.1: Search Method for agent approaching a distant box

2.2. Container Data Structure

Every state movement in workspace is stored in a "search node", which is a generic class containing properties such as priority, total cost, heuristic, cost, state, and parent node. The state consists of a robot config, list of moving-box, list of movable obstacles.

```
public class SearchNode implements Comparable<SearchNode> {

    public double priority;
    public double totalCost;
    public double heuristic;
    public double cost;
    public State state;
    public SearchNode parent;
```

4 types of cost are evaluated when comparing two nodes:

- **MovingCost** is calculated by Manhattan distance, it represents the cost of moving from current node to next node.
- **RobotCost** refers to the cost of robot moving from one position to the box to another position.

  Fox example :

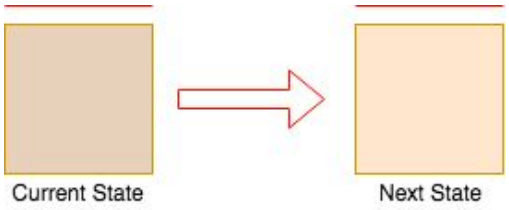


Figure 2.2.2: **RobotCost=0,** when no position change

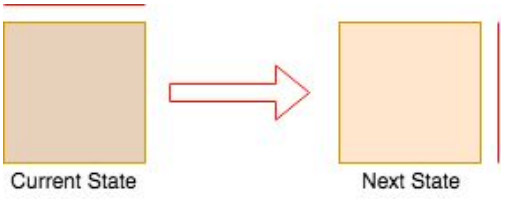


Figure 2.2.3: **RobotCost=box width** when robot in need to rotate once

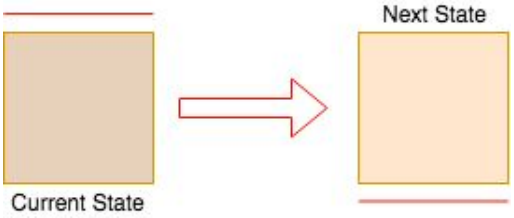


Figure 2.2.4: **RobotCost=box width*2** when robot need to rotate twice

- **ObstacleCost** is the cost that is calculated by the count of movable obstacles that block the moving path of the moving box
- **HeuristicCost** is heuristic moving cost from the next state to the goal state based on Manhattan distance.

Based on these costs, the program will calculate the sum of **MovingCost**, **RobotCost,** and **ObstacleCost** as the cost value of each node.

Nodes and their respective cost value will be stored in PriorityQueue<SearchNode>, where nodes are compared and ranked based on their **f(n)**, which is the sum of (**MovingCost** + **RobotCost** + **ObstacleCost**) and estimated cost (Manhattan distance) from a current state to the goal state.

2.3. Pseudocode

2.3.1. Algorithm SolveProblem(problem)

```
problemSolution <- null
unsolveBoxes <- problem.movingBoxes
stepSize = <- initialStepSize
while (unsolvedBoxes.size() > 0) && !isTimeOut():
      pendingBoxes <- unsolvedBoxes
      while pendingBoxes.size() > 0:
            movingBox <- get nearest pending box
            boxSolution <- solveMovingBox(box, stepSize)
            if (movingBoxSolution != null):
                  problemSolution += boxSolution
                  unsolvedBoxes.remove(box)
            else if stepSize == MIN_STEP_SIZE:
                  randomMoveObstacle()
      stepSize <- stepSize - MIN_STEP_SIZE
      if stepSize < MIN_STEP_SIZE:
            stepSize <- MIN_STEP_SIZE
return problemSolution
```

2.3.2. Algorithm SolveMovingBox(movingBox)

```
movingBoxSolution <- null
movingSteps <- find the path to move the moving-box to goal
if movingSteps == null:
      return null
else:
      obstacles <- find obstacles in the moving path
      while obstacles.size() > 0:
            obstacle <- get the nearest obstacle
            obstacleSolution <- solveObstacle(obstacle)
            if (obstacleSolution != null):
                  movingBoxSolution += obstacleSolution
            else: return null
      movingSteps <- Solve the moving-box again
      if movingSteps != null:
            robotSteps = generateRobotSteps(obstacleSteps)
            if robotSteps != null:
                  movingBoxSolution += robotSteps
            else: return null
return movingBoxSolution
```

2.3.3. Algorithm SolveObstacle(obstacle)

```
obstacleSolution <- null
obstacleSteps <- find solution to move the obstacle away from the path
if (obstacleSteps == null):
      return null
else:
      robotSteps = generateRobotSteps(obstacleSteps)
      if (robotSteps != null):
            obstacleSolution += robotSteps
      else: return null
return obstacleSolution
```

2.4. Connection Strategy

2.4.1. Robot configuration connection strategy

To test whether the robot is able to reach a new position without colliding into objects on the path, the agent can test "connection" between the current state and the desired next state by checking collision for its "movement boundary". Two robot states are considered "connected" if the movement boundary does not collide with any other objects. The movement boundary consists of:

- Collision-free rectangle for rotation which contains 4 points p1, p2, p1' and p2'
- Four connection lines between two endpoints of the robot p1', p1", p2' and p2"
- Two robot lines (p1', p2') and (p1", p2")

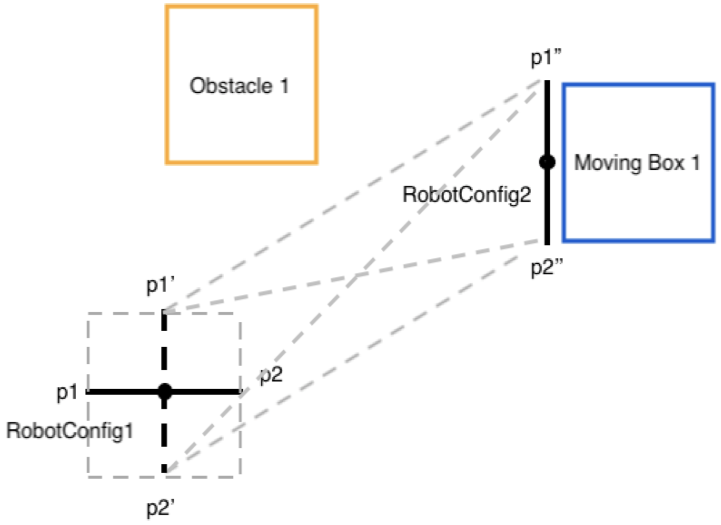


Figure 2.4.1.1: Robot connection strategy

2.4.2. Box position connection strategy

In a particular state, the agent will perform the connection tests on possible states in 4 directions. If there is a space for robot to couple with the box in one side, the box can be pushed in an opposite direction, The next state after the box is pushed also be checked, if it is a collision-free state then this state will be considered "connected" with current state, which indicates a safe state transition.

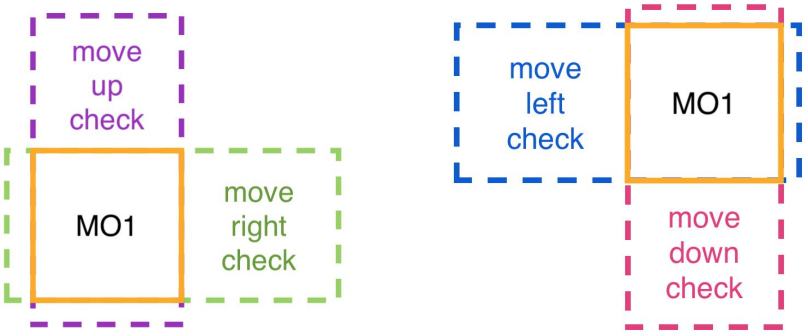


Figure: 2.4.2.1 Connection test on successor states

2.5. Discretization Method

Discretization is implemented in a number of aspects of the program to simplify path planning and reduce complexity.

2.5.1. Explore neighbouring spaces during planning using A* search method

Discretization is used at the moving-box path planning phase, dividing the whole continuous path into a number of rectangles, and performing a collision test on each of them. If every virtual rectangle on the path has passed collision test, the whole path is considered feasible:

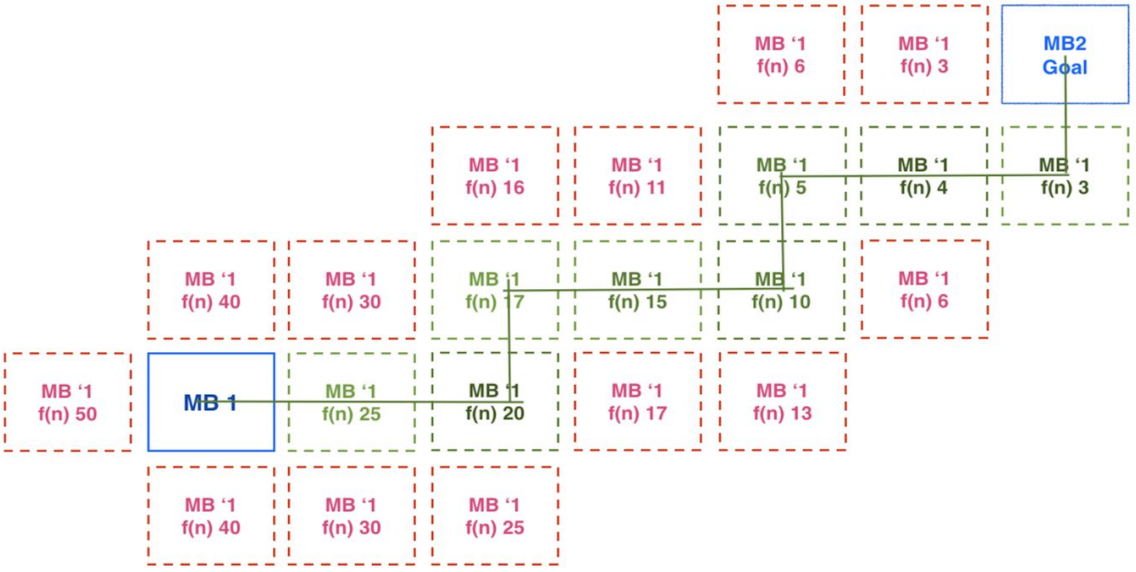


Figure 2.5.1.1: Discretise box movement during planning phase

2.5.2. Discretise robot movement

For the robot movement, the agent only selects the positions around objects, the agent takes 16 positions for the normal objects and extra 8 neighbouring positions for the object that the robot is pushing.

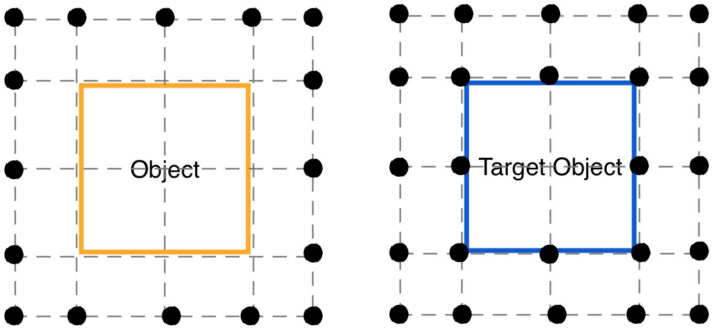


Figure: 2.5.2.1

For instance, the green vertices in the graph are picked for the purpose of general robot movement passing around the obstacle, while the yellow ones are selected for cases where space between obstacles are extremely limited, so the agent has to rotate into position parallel to the side and slide

in to pass the tiny channel. Note that distance d=w/2+Max_Error, which is the sum of half of robot length and the minimum collision free distance.

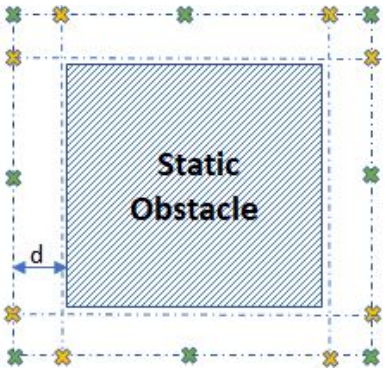


Figure 2.5.2.2

To simplify the path and reduce the search time, the agent only focuses on the objects that are on the moving path (which is a line connects two centre points) or close to the target position of the robot. For example, in the Figure 2.5.2.3, the agent should only take account of objects MO1, MO2 and the target object.

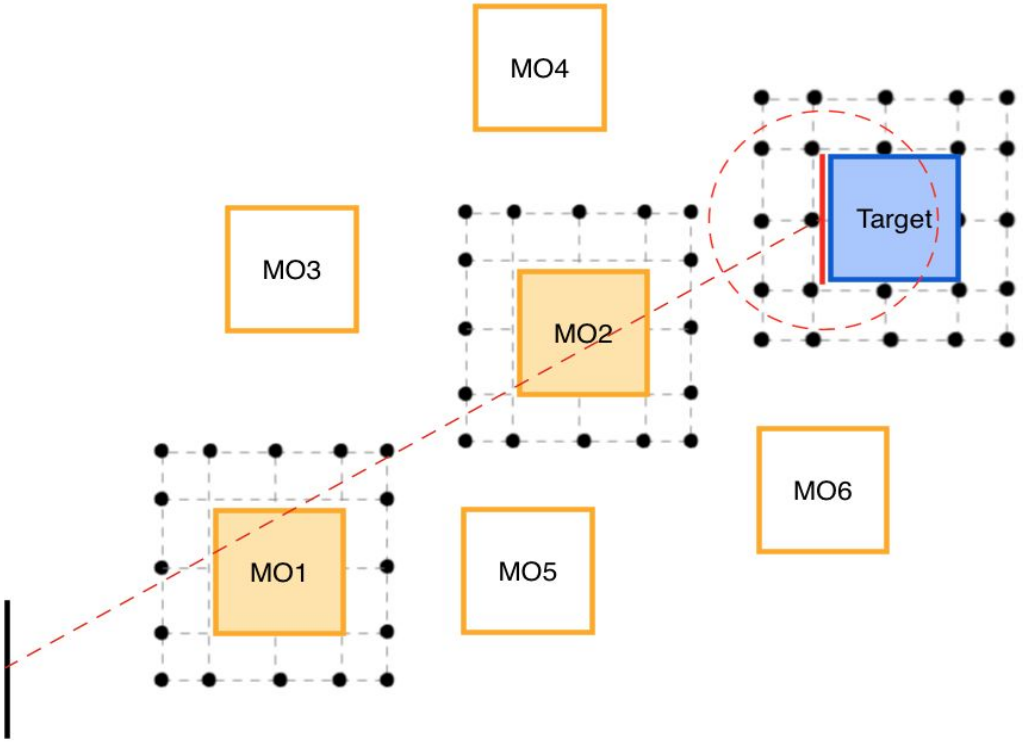


Figure 2.5.2.3: Discover feasible vertices by discretization

3. Scenarios that the program can solve well

In general, the agent can solve the problems with 1-10 moving-boxes, 1-15 movable obstacles and 1-10 static obstacle in less than 2 minutes, as long as the obstacles and the moving-boxes are scattered. This is enabled by features below:

- Keep solving the problem until all the obstacles are solved or timed out.
- Be able to find the nearest moving box or obstacle to solve first.
- If the agent cannot find a solution for a box, it will skip that one and move on to the next.
- If the problem cannot be solved with a particular step size, the agent will try again with a smaller step size until the step size reaches the MIN_STEP_SIZE, which is set to be 0.01 unit.
- When the agent still cannot solve a problem after attempts above, it will keep moving random obstacles to solve the problem.

Sample cases the agent can solve:

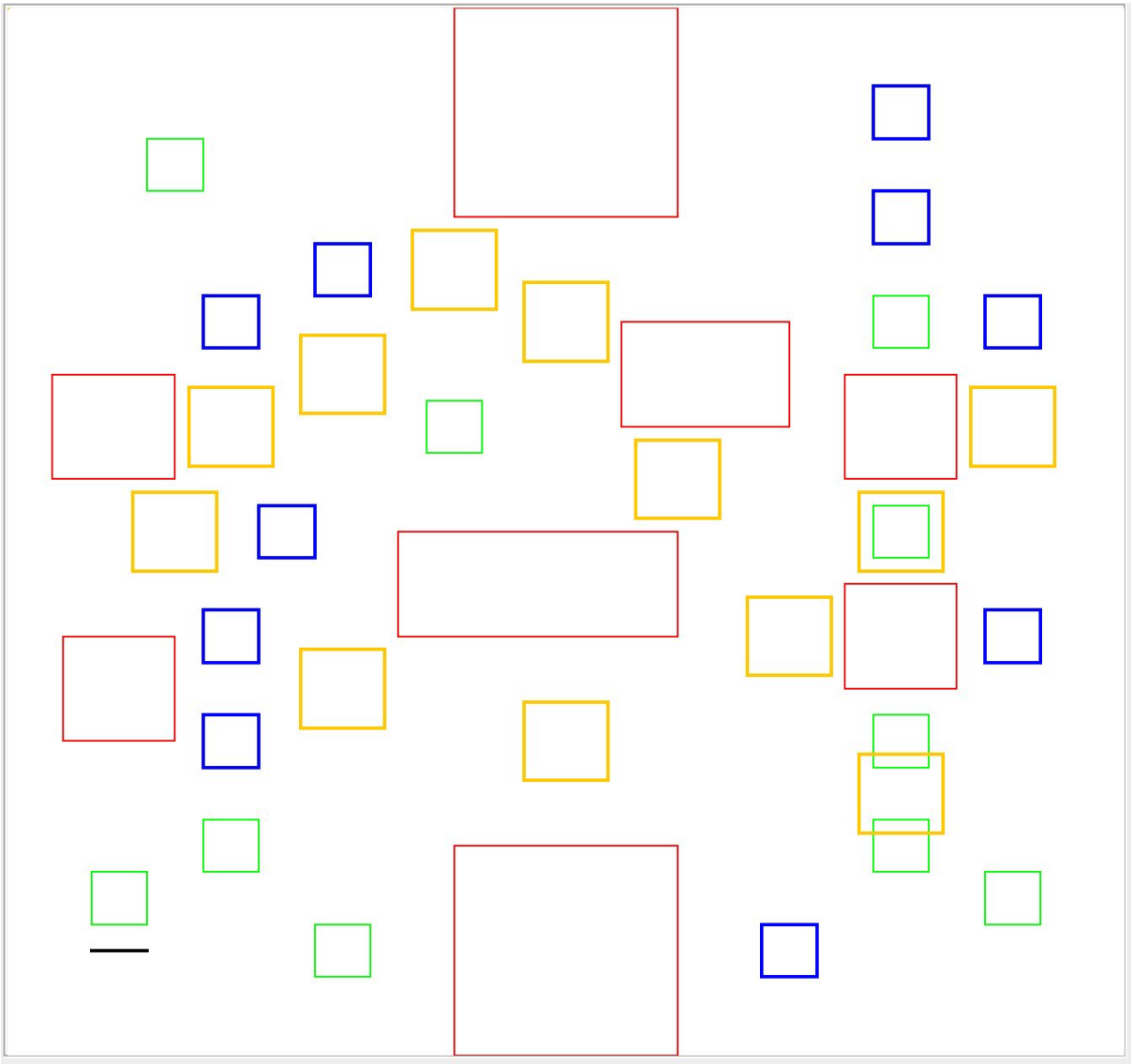
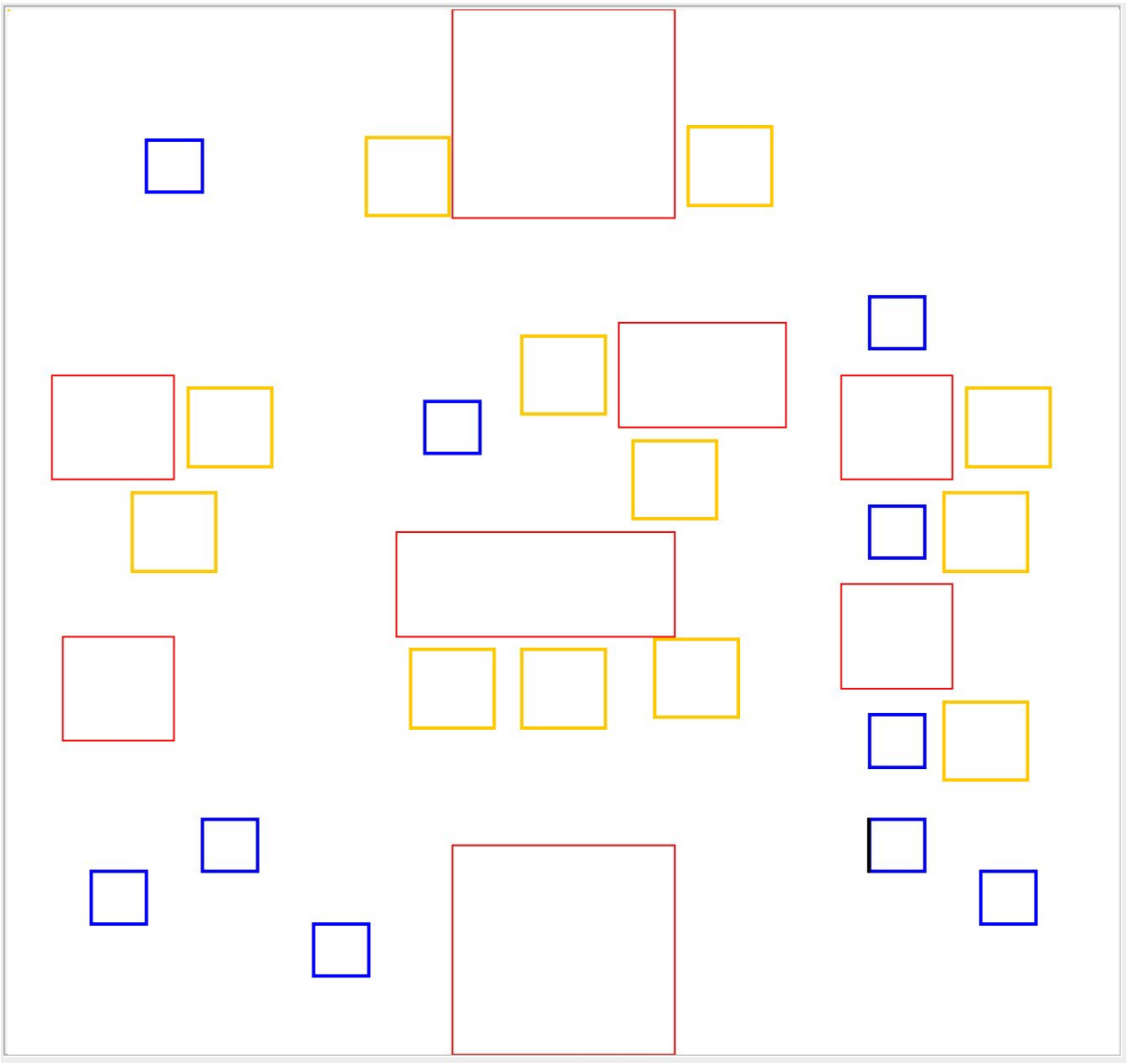


Figure 3.0.1: General cases with 30 objects



```
Number of MovingBox: 10
Number of Obstacle: 12

Start solving with step size: 0.05
    Solving MovingBox: 2
        Solving Obstacle: 3
            Find path for Robot: 2 steps
        Solving Obstacle: 0
            Find path for Robot: 3 steps
        Find path for Robot: 22 steps
        No solution for Robot!
    Solving MovingBox: 9
        Solving Obstacle: 10
            Unable to move Obstacle!
        Skip MovingBox: 9
    Solving MovingBox: 5
        Solving Obstacle: 5
            Find path for Robot: 1 steps
        Solving Obstacle: 9
            Find path for Robot: 1 steps
        Find path for Robot: 17 steps
        No solution for Robot!
    Solving MovingBox: 8
        Solving Obstacle: 0
            Find path for Robot: 2 steps
        Find path for Robot: 20 steps
        No solution for Robot!
    Solving MovingBox: 7
        Solving Obstacle: 10
            Find path for Robot: 2 steps
        Find path for Robot: 25 steps
        No solution for Robot!
    Solving MovingBox: 4
```

```
        Solving Obstacle: 11
            Unable to move Obstacle!
        Skip MovingBox: 4
    Solving MovingBox: 6
        Solving Obstacle: 6
            Find path for Robot: 7 steps
        Find path for Robot: 16 steps
    Solving MovingBox: 3
        Solving Obstacle: 10
            Find path for Robot: 1 steps
        Find path for Robot: 26 steps
        No solution for Robot!
    Solving MovingBox: 1
        Find path for Robot: 26 steps
        No solution for Robot!
    Solving MovingBox: 0
        Find path for Robot: 23 steps

Start solving with step size: 0.04
    Solving MovingBox: 2
        Find path for Robot: 27 steps
        No solution for Robot!
    Solving MovingBox: 4
        Solving Obstacle: 2
            Find path for Robot: 3 steps
        Solving Obstacle: 7
            Find path for Robot: 2 steps
        Find path for Robot: 24 steps
    Solving MovingBox: 1
        Find path for Robot: 33 steps
        No solution for Robot!
    Solving MovingBox: 8
        Solving Obstacle: 10
```

```
            Find path for Robot: 2 steps
        Find path for Robot: 26 steps
    Solving MovingBox: 5
        Find path for Robot: 22 steps
        No solution for Robot!
    Solving MovingBox: 9
        Solving Obstacle: 9
            Find path for Robot: 2 steps
        Find path for Robot: 26 steps
    Solving MovingBox: 3
        Solving Obstacle: 10
            Find path for Robot: 5 steps
        Find path for Robot: 33 steps
    Solving MovingBox: 7
        Find path for Robot: 32 steps

Start solving with step size: 0.03
    Solving MovingBox: 1
        Find path for Robot: 43 steps
    Solving MovingBox: 2
        Find path for Robot: 37 steps
    Solving MovingBox: 5
        Find path for Robot: 29 steps
        No solution for Robot!

Start solving with step size: 0.02
    Solving MovingBox: 5
        Find path for Robot: 43 steps

Total time: 74.83 seconds
Number of Steps: 33892

Process finished with exit code 0
```

Figure 3.0.2: General case solved with output

3.1. Scenario A: A narrow passage not aligned with current box position

In this case, the agent would not be able to solve the box path to goal if it remains the same expansion step size. Hence, it gradually reduces the step size each time when there is not feasible path identified, as shown in the screenshot below:

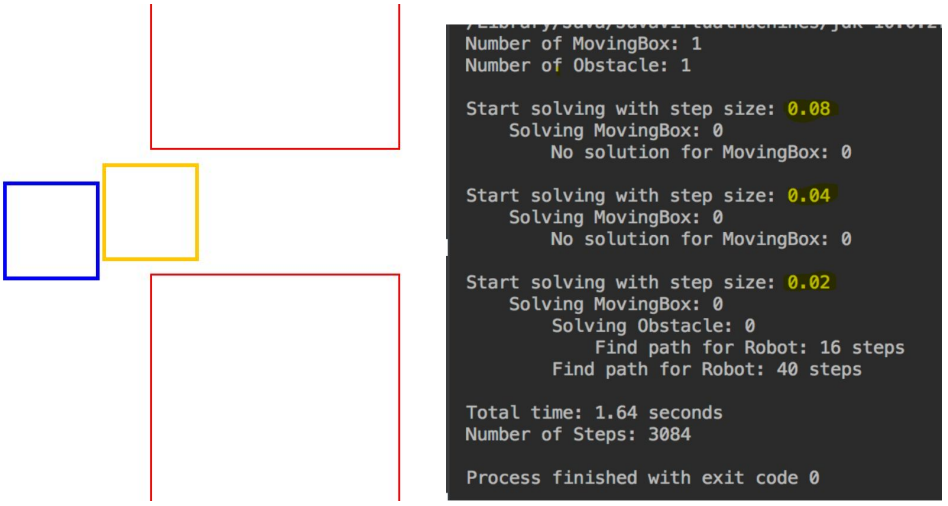


Figure 3.1: A cases when the narrow corridor is not aligned with box expansion step

3.2. Scenario B: Movable obstacles block the moving path

In case there are movable obstacles on the path to goal, the agent can plan out a path first, pretending all movable objects do not exist, and after the path is determined, the robot will be able to push all movable obstacles away and then push moving box to its goal.

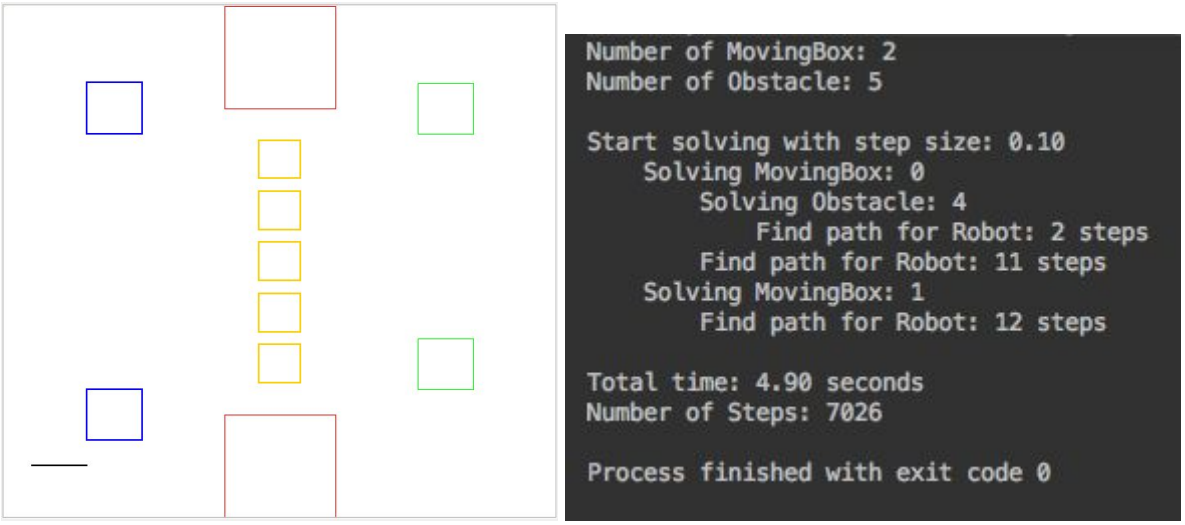


Figure 3.2.1: A cases when the are many obstacles block the path to the goal

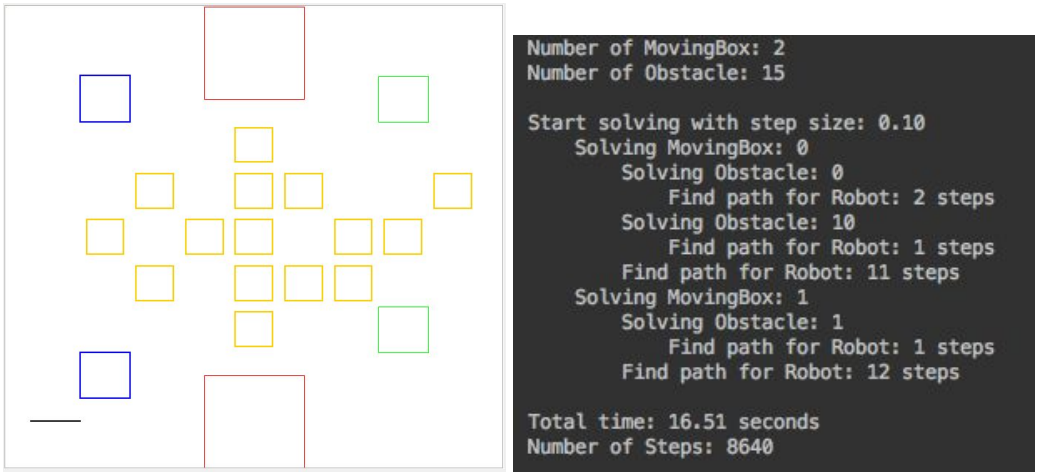


Figure 3.2.2: A cases when the are many obstacles block the path to the goal

3.3. Scenario C: When the agent needs to take a detour to enter the correct coupling position

In earlier versions of the program, the agent was not able to find a solution shown as figure 3.3.1 because the static rectangle is not on the direct path between agent and its desired position, which is the left side of moving-box. Hence the static obstacle did not have sufficient sample positions around it.

After the revision, the agent now understands how to go around the static obstacle and reaches the desired coupling position.

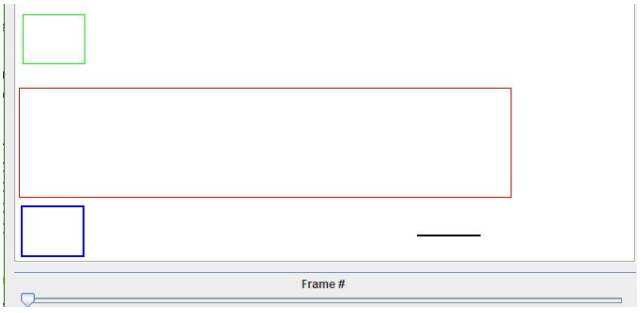


Figure 3.3.1: Initial State

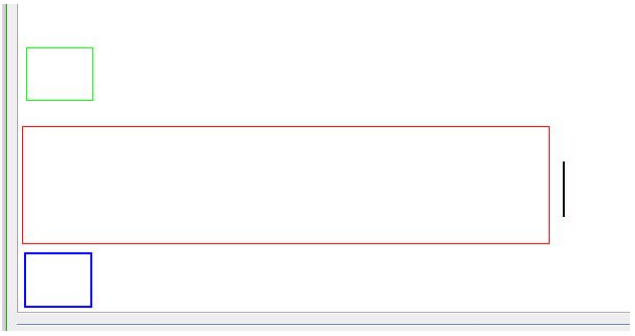


Figure 3.3.2: Robt takes a detour around static obstacle

Figure 3.3.3: Robot slide into the corridor

Figure 3.3.4: Robot reaches its desired position

Figure 3.3.5: Robot pushes moving-box towards goal

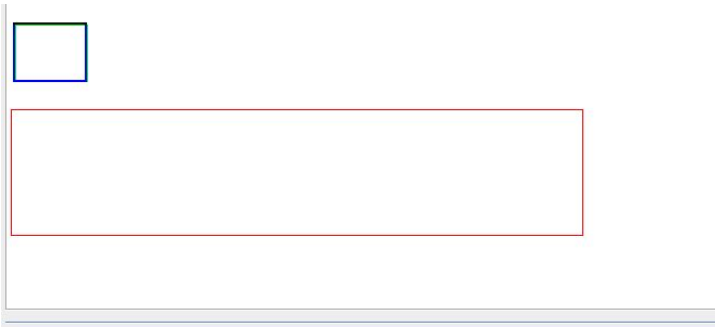


Figure 3.3.6: Robot pushes moving-box reaching the goal

3.4. Scenario D: When pushing position is difficult to reach

In this case, the agent is able to travel directly to the coupling position, knowing exactly how to slide into the tiny gap. This is enabled by the discretization strategy explained in section 2.5.2.

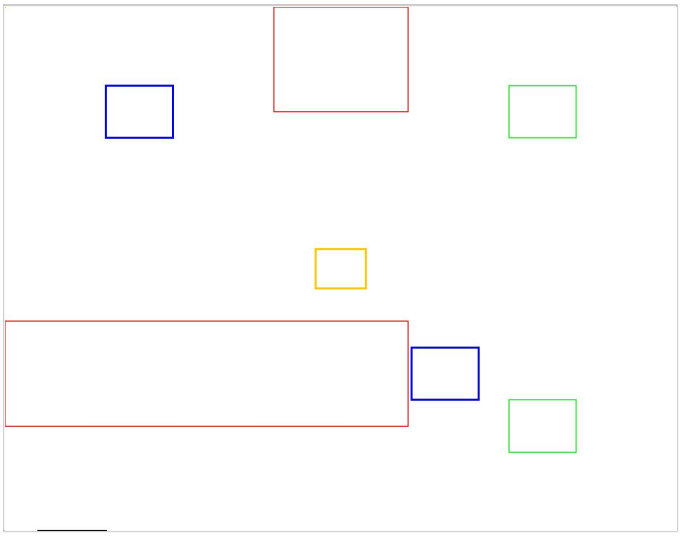


Figure 3.4.1: A case with very limited space around coupling position

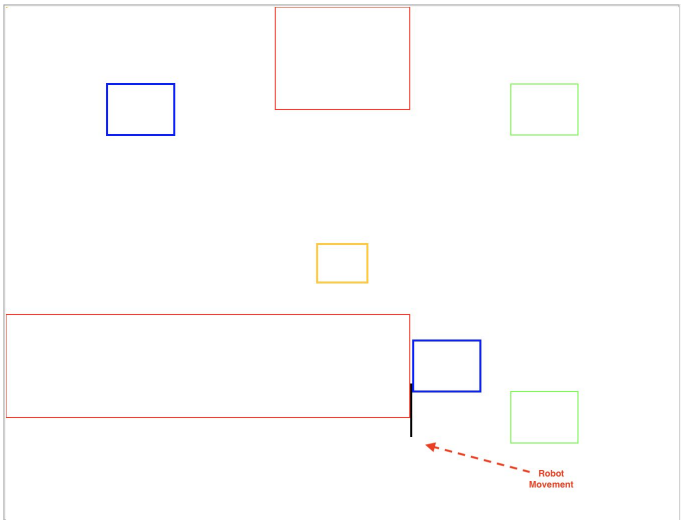


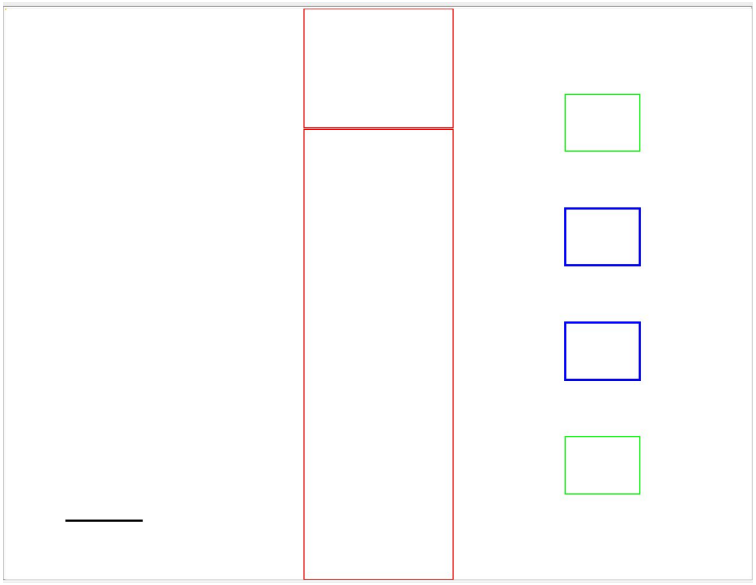
Figure 3.4.2: The robot directly goes to the tiny gap aiming at pushing position

```
Number of MovingBox: 2
Number of Obstacle: 1

Start solving with step size: 0.10
    Solving MovingBox: 0
        Find path for Robot: 7 steps
    Solving MovingBox: 1
        Find path for Robot: 12 steps

Total time: 2.74 seconds
Number of Steps: 4609

Process finished with exit code 0
```

Figure 3.4.3: Agent solves the problem

3.5. Scenario E: When distance between static obstacles equals to 2 * MAX_ERROR

This is an extreme case where the corridor between two static objects is exactly 0.0002 unit wide, which is the minimum gap for robot passing through without triggering collision. The agent is able to solve it well as coordinates around the tiny corridor have been sufficiently sampled for robot movement.



```
Number of MovingBox: 2
Number of Obstacle: 0

Start solving with step size: 0.10
    Solving MovingBox: 0
        Find path for Robot: 2 steps
    Solving MovingBox: 1
        Find path for Robot: 2 steps

Total time: 0.76 seconds
Number of Steps: 2184

Process finished with exit code 0
```

Figure 3.5.1: A cases when the free space between two static obstacles equal to 2 * MAX_ERROR

4. Situations that the program will fail

4.1. Unable to move through successive, tight, misaligned passages

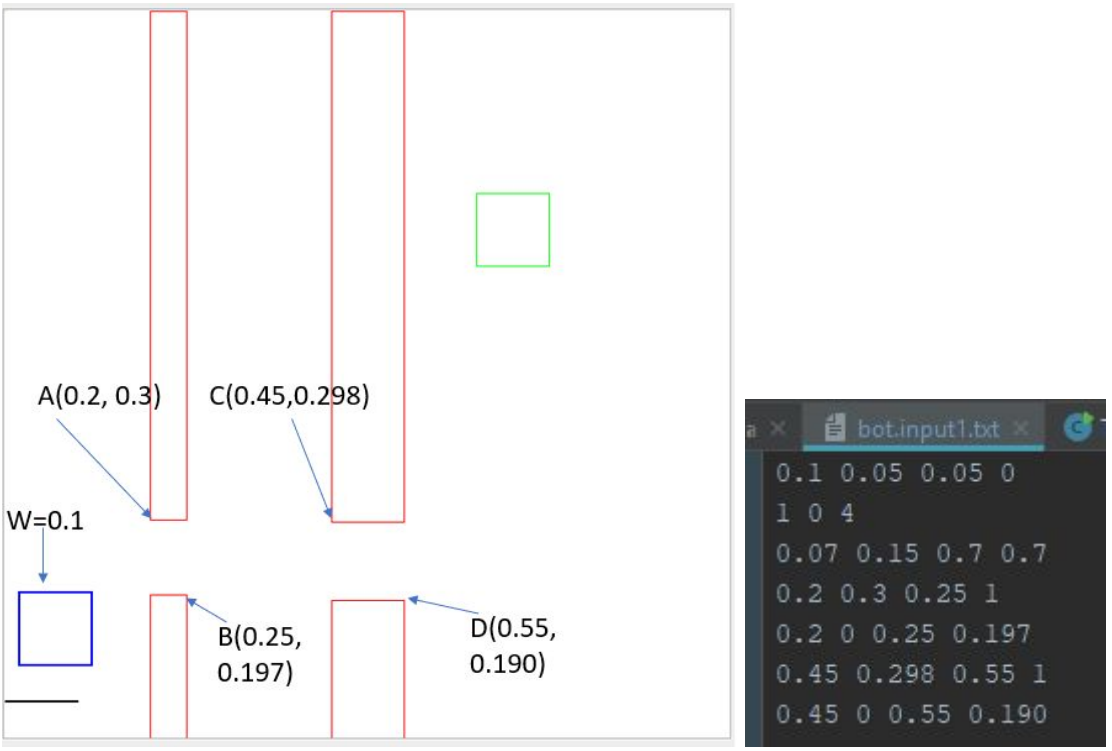


Figure 4.1.1: A case with tight misaligned corridors and Input configuration

In this case, both corridors are just wide enough for the box to pass, but they are a little bit misaligned with each other. The agent can solve either one pair of static obstacles, but not both at the same time.

Since the solution implemented implies a minimum expansion step size of 0.01, it is impossible for the same set of configuration (initial position and expansion step size) to fit for each slightly misaligned corridor.

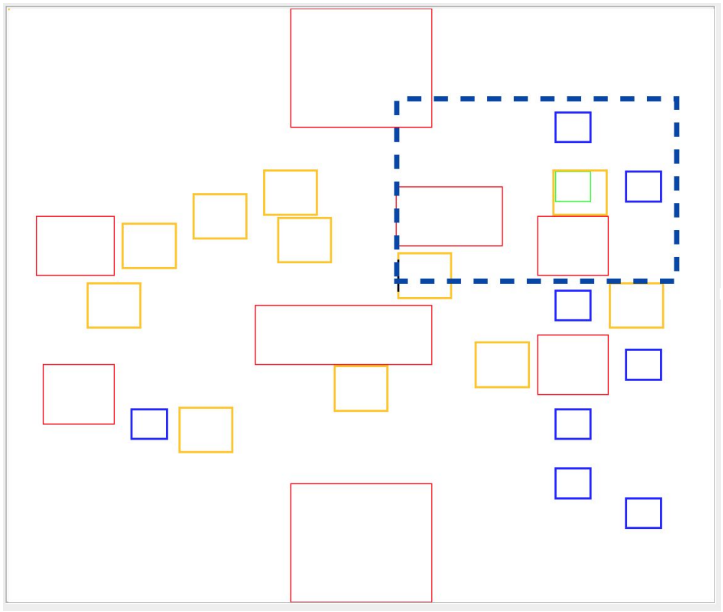
Further decreasing expansion step size is a solution to this particular issue, which could on the other hand dramatically increase the computational cost, stopping the agent to handle cases where large number of objects co-exist in workspace.

The agent in this scenario will just keep calculating on box path planning until time-out, as the following screenshot:

```
    Solving MovingBox: 0
        No solution for MovingBox: 0

Start solving with step size: 0.01
    Solving MovingBox: 0
        No solution for MovingBox: 0

Start solving with step size: 0.01
    Solving MovingBox: 0
        No solution for MovingBox: 0

Total time: 110.16 seconds
Number of Steps: 1

Process finished with exit code 0
```

Figure 4.1.2: Agent fails in path planning

4.2. Unable to move obstacles surrounded by other objects



```
        Skip MovingBox: 2

Start solving with step size: 0.01
    Solving MovingBox: 2
        Solving Obstacle: 8
            Unable to move Obstacle!
        Skip MovingBox: 2

Total time: 130.87 seconds
Number of Steps: 26624

Process finished with exit code 0
```

Figure 4.2.1: Unable to move obstacle out of the goal area

This problem occurred as the moveable obstacle that blocks the target cannot be moved away because the obstacle is surrounded by static obstacles and moving-boxes, and the agent does not know to push other moving boxes away. This issue could be potentially resolved by using another approach as treating the moving-box as movable obstacle. or randomly selecting moving-boxes to be solved rather than solving them in a particular order.

4.3. Unable to clear a path for robot rotation

Figure 4.3.1: initial state and agent failed attempts

For this scenario, an ideal agent would be able to push all boxes aside to create space for rotation in order to move some obstacles out before reaching the moving-box (Figure 4.4.2).

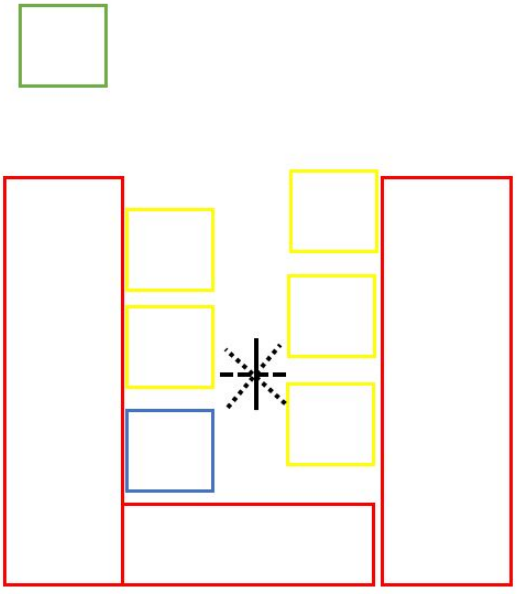


Figure 4.3.2: Ideal agent solution

However, our current agent does not have such planning capability and could not provide a solution, as shown in figure 4.3.1.

4.4. Unable to clear a path prior to pushing the obstacle away

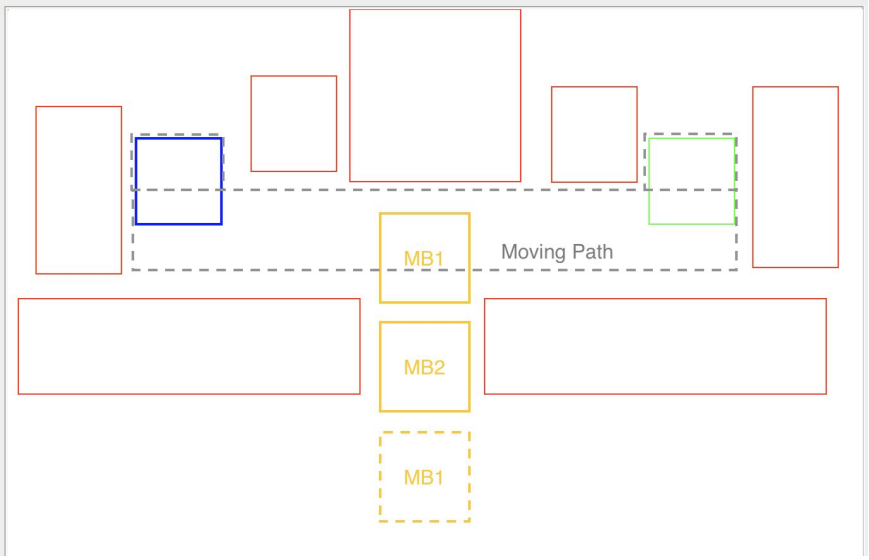


Figure 4.4.1: A case when the tunnel has 2 moving obstacles

In this problem, the robot found a possible moving path during planning stage, but the MB1 object blocks the path. The robot can only move MB1 away from the path if MB2 is not blocking it. Agent tries to move to other paths of tunnel but it was blocked by other static objects. In this condition, it may be necessary to implement a function to push of movable obstacle that block other movable obstacle, which may increase complexity or cause unseen issues.

4.5. Unable to find path for robot

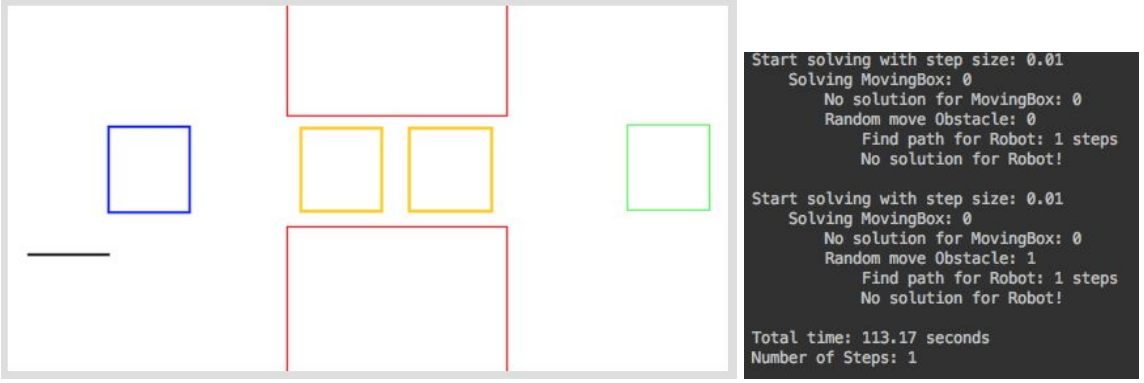


Figure 4.5.1: unsolved problem

In this problem the robot cannot find a path to goal because the path blocked by two moveable obstacles and the robot cannot rotate between these two movable obstacles. This can be considered as unsolvable problem and the agent will keep looking for solution until time-out.