2018

# Search for Movers

COMP3702 – Assignment 1

Team WLS

# 1. Agent Design Problem

The agent is to solve a motion planning problem. The environment is continuous (can be discretized), deterministic, fully observable and static. To reduce the dimensionality, the agent consists of two subagents (see Section 2 for details): one solves paths of moving boxes and obstacles, while another solves robot paths to required moving objects from the former agent output. Therefore, every design component is broken into two stages.

## 1.1 State Space:

The overall state space is defined as:

$S = \{Robot\ configurations\ (x_{bot}, y_{bot}, \alpha_{bot});\ Moving\ box\ locations\ (X_{mb}, Y_{mb});\ Moving\ obstacle\ locations (X_{mo}, Y_{mo});\ Static\ obstacle\ locations (X_{so}, Y_{so})\}$

The state space of the robot and moving obstacles are rearranged as follows:

$$S_{bot} = \{(x_{bot}, y_{bot}, \alpha_{bot}); S_{mov}\}$$

$$S_{mov} = \{(X_{mb}, Y_{mb}); (X_{mo}, Y_{mo}); (X_{so}, Y_{so})\}$$

## 1.2 Action Space:

The action space of the robot is defined as:

$$A_{bot} = \{(\Delta x_{bot}, \Delta y_{bot}, \Delta\alpha_{bot})\}$$

Where positive directions are right, up and anticlockwise.

With the addition of box subagent, another action space is defined as:

$$A_{mov} = \{(\Delta X_{mb}, \Delta Y_{mb}); (\Delta X_{mo}, \Delta Y_{mo})\}$$

Where positive directions are right and up.

## 1.3 World Dynamics:

$$T: S \times A \rightarrow S'$$

Where $T$ means the transition from one state to another.

In terms of the robot subagent, it is one of the following (where $\Delta$ is the required step size such that displacement is no more than 0.001):

$$x'_{bot} = x_{bot} + \Delta x_{bot}$$
$$y'_{bot} = y_{bot} + \Delta y_{bot}$$
$$\alpha_{bot}' = \alpha_{bot} + \Delta \alpha_{bot}$$

In terms of the box subagent, it is one of the following (where: $\Delta$ is the defined step (discretization) size; and $i$ stands for the index since only one of the movable boxes is allowed to be moved at a time):

$$X'_{mb}[i] = X_{mb}[i] + \Delta x_{mb}$$
$$Y'_{mb}[i] = Y_{mb}[i] + \Delta y_{mb}$$
$$X'_{mo}[i] = X_{mo}[i] + \Delta x_{mo}$$
$$Y'_{mo}[i] = Y_{mo}[i] + \Delta y_{mo}$$

## 1.4 Percept Space:

The map is fully observable, thus $O$ equals to the state space $S$.

## 1.5 Percept Function:

The map is fully observable, thus $Z: S \rightarrow O$ is the identity function and can be omitted.

## 1.6 Utility Function:

$$U: S \rightarrow r \in \mathbb{R}$$

Conventionally, $U$ needs to be maximized for the solution. In this search problem, it can be set as 0 for non-goal states and 1 for the goal state. It is further interpreted as minimizing the cost $f$ to achieve the goal.

For box path subagent:

$$f_{mov} = k_m \sum_{i=1}^{mb.size} d_{mbg}(i)$$

, where: $d_{mbg}(i)$ is the Euclidean distance, for the $i^{th}$ moving box, from its current position to its goal; $k_m$ is a positive constant weight assigned.

For robot path subagent:

$$f_{bot} = k_{b1} d_{xy} + k_{b2} d_{\alpha}$$

, where: $d_{xy}$ is the Euclidean distance from the robot's current location to the temporary goal (the required moving box or moving obstacle) location; $d_{\alpha}$ is the difference between the current robot orientation and the required angle at the temporary goal; $k_{b1}$ and $k_{b2}$ are two positive constant weights assigned.

Note that none of the obstacles is taken into account, as it is desirable to have an admissible heuristic which is never overestimated.

# 2. Search Methodology

Firstly, the box subagent finds the path for each box from the initial position to the goal position. Then the robot subagent plans the motion of the robot based on the output from the box subagent. Both of the subagents are based on the Best-First algorithm and utilize Rapidly-exploring Random Trees (RRT) if Best-First is stuck at a local optimum (the state result does not have the global minimal cost). The reason why sampling is not the first stage is because the amount of required sample can vary drastically on different problems and for some simple cases Best-First can be much faster. More justifications are in Section 3 and 4.

## 2.1 Primary Search

Best-First expands the initial state with respect to the action space and the cost. For most situations (explained in Section 3), this should be sufficient and faster to get a solution. The state is respectively implemented as class "TreeNode" for box and class "RobotNode" for robot. Other data structures involved are listed below:

a set $V$ is used for Best-First visited states

a priority queue $PQ$ is used for storing temporarily and then obtaining the lowest-cost state in Best-First (a cost comparator is implemented for polling)

a list $C$ is used for storing all valid children of the state node in Best-First

a map $M$ is used as the state graph which contains pairs of one state node and its child pair (child node $s$, the corresponding action).

**Algorithm 1: Best-First Search (Denoted as $aStar()$)**

**Input:** Initial box or robot state $s_{start}$, Discretization size $\Delta$
$V \leftarrow ()$ // Collection of visited box states
$PQ \leftarrow (s_{start})$ // Queue of states to be visited
$M \leftarrow ((s_{start}, null))$ // State graph (accessed externally to reconstruct the path)
**while** $|PQ| > 0$ **and** $|V| <$ EXPANSION_LIMIT **do**
    $s \leftarrow PQ.poll()$ // Get and remove the node with lowest cost
    $V \leftarrow (V, s)$ // Append the node to the visited set
    **if** $s = s_{goal}$ **then** // Can be final goal or intermediate step goal
        **return** $s$
    **end if**
    $C \leftarrow s.getChildren(\Delta)$ // Expand current node with collision-free next states
    **for all the** $c \in C$ **do** // The format of $c$ is $(s, action)$
        **if** $c(0) \notin (PQ \cup V)$ **then**
            $PQ \leftarrow (PQ, c(0))$
            $M \leftarrow (M, (s, c))$
        **end if**

**end for**
**end while**
**return** $s$ // No solution to goal, need to refine step or switch to RRT
**Output:** Final state of the algorithm

---

As mentioned before, $\Delta$ has to be 0.001 for the robot. On the other hand, $\Delta$ is chosen to be the robot width for the box subagent since all moving boxes have this side length. $\Delta_{\text{box}} < w$ has found to be less effective because it could lead to objects too close to each other in some intermediate steps which makes sampling more complicated. The algorithm for finding possible node children is shown below. In all possible degrees of freedom, the box subagent will try to move each movable object, while the robot subagent will try to move the robot. Invalid children (e.g. collision) will be eliminated.

---

**Algorithm 2: State Child Node Retrieval (Denoted as $getChildren()$)**

---

**Input:** Discretization size $\Delta$, The state node (parent) itself $s$
$C \leftarrow ()$ // Empty list to put child node pairs
**for all the** $o \in S$ **do** // Objects in state space (any movable objects or the robot)
  **for all the** $a \in A$ **do** // Elements in action space (2 directions*degrees of freedom)
    $s' \leftarrow s.deepCopy()$ // Without changing the parent
    $action \leftarrow (o, a, \Delta)$
    $s'.move(action)$ // Dynamics
    **if** $s'$ is collision-free **and** sliding-free **then**
      $c \leftarrow (s', action)$ // Pair of child node and action
      $C \leftarrow (C, c)$
    **end if**
  **end for**
**end for**
**return** $C$
**Output:** List of pairs of child and action $C$

---

## 2.2 Secondary Search

When Best-First cannot find a path after the iteration limit, the agent will switch to RRT (continue with the last state of Best-First). Due to computational cost of high dimensionality, the sampling is applied to no more than 2 moving boxes ($2 \times 2$ degrees of freedom=4 dimensions (4D), other movable objects are regarded as static obstacles) for the box subagent and the robot (3 degrees of freedom=3 dimensions (3D)). The four main components are:

- Sampling strategy: uniform and random.
- Configuration validation: collision check (line-segment-based) given. For box subagent, check between current box and all other boxes plus static obstacles; For robot subagent, check between robot and all other objects.
- Connection strategy: Euclidean distance $d$ less than $d_{step}$. For box subagent, $d = \sqrt{\Delta x_{mb}^2 + \Delta y_{mb}^2}$ (initially 2-dimensional (2D), only the box of interest) or

$d = \sqrt{\Delta x_{mb}^2 + \Delta y_{mb}^2 + \Delta x_o^2 + \Delta y_o^2}$ (the box of interest plus another movable object if 2D RRT fails);

For robot subagent, $d = \sqrt{\Delta x_{bot}^2 + \Delta y_{bot}^2 + k\Delta\alpha_{bot}^2}$, where $k$ is a small constant factor to approximate (flatten) 3D problems to 2D

- Line segment in C-space validation: divide the edge into 10 segments, try walking through the edge via these segments one by one, if any segment contains collision then return and discard the edge.

The algorithm is illustrated below, where specialized sample node classes are implemented for states and a list $G$ is used for storing connected nodes:

---

**Algorithm 3: Sampling-based Search (Denoted as $rrt()$)**

---

**Input:** Box or robot state node $s$, object (box/robot) to move $o$, optional secondary movable object $b$

$G \leftarrow ()$ // Empty list to put vertices linked
$s' \leftarrow s.deepCopy()$
**while** $s' \neq s_{goal}$ **do**
  **if** $|G| >$ GRAPH_SIZE_LIMIT **then**
    **return** $\emptyset$ // No route is found within the limit
  **end if**
  **while** $s'$ **is** collision-free **do**
    **for all the** $a \in A$ **do**
      $\Delta \leftarrow random()$
      $action \leftarrow (o, a, \Delta)$
      $s'.move(action)$
      **if** $b \neq \emptyset$ **then** // b is not null, 4D problem
        $\Delta \leftarrow random()$
        $action \leftarrow (b, a, \Delta)$
        $s'.move(action)$
      **end if**
    **end for**
  **end while** // End of random sampling
  **if** $s' \notin G$ **then**
    $s'' \leftarrow G.getDistanceClosest(s')$
    **if** $d(s', s'') > d_{step}$ **then** // Samples still too far
      **while** $s'$ **is** collision-free **do**
        $\Delta \leftarrow d(s', s'')/10$ // Same as line segment checking
        $a \leftarrow s'' \overrightarrow{s'}$ // Direction vector from $s''$ to $s'$
        $action \leftarrow (o, a, \Delta)$
        $s' \leftarrow s''.move(action)$
        **if** $b \neq \emptyset$ **then** // b is not null, 4D problem
          $action \leftarrow (b, a, \Delta)$
          $s' \leftarrow s''.move(action)$
        **end if**
      **end while**
    **end if**
    **if** $validLineSegment(s', s'')$ **then**
      $s'.parent \leftarrow s''$ // Connect the vertices (as an edge)

$G \leftarrow (G, s')$
**if** $d(s', s_{goal}) < d_{step}$ **then** // One step away from goal
$s_{goal}.parent \leftarrow s'$
**end if**
**end if**
**end if**
**end while**
**return** $G$ // Sampling always has a solution but may take too long
**Output:** List of box or robot states

---

## 2.3 Overall Algorithm

The box subagent and the robot subagent work in sequence. There are, as mentioned before, maximum expansion or sampling limits to prevent time-out. For each subagent, primary Best-First search is applied and if the solution is not returned within the limit, use RRT and apply another Best-First on the RRT result. The box subagent can do 4D RRT if 2D RRT does not return an answer within the limit. If any intermediate search step fails (either Best-First or RRT), the algorithm will jump to file output which is the best it can solve.

---

**Algorithm 4: Overall Problem Solving**

---

**Input:** Problem Specification $ps$
// Empty list to put states of the subagents
$L_{bot} \leftarrow ()$
$L_{box} \leftarrow ()$

// First get the initial state for the box subagent
$s_{box} \leftarrow (ps.movingBoxes, ps.movingObstacles, ps.staticObstacles, ps.goals)$
$s_{boxGoal} \leftarrow s_{box}.getGoalState()$
$w \leftarrow ps.width$

// Solve box path with Best-First
$s_{box} \leftarrow boxAgent.aStar(s_{box}, w)$
**if** $s_{box} \neq s_{boxGoal}$ **then** // Switch to RRT engine
**for all the** $o \in s_{box}.getUnresolvedBox()$ **do**
$L_{tempBox} \leftarrow boxAgent.rrt(s_{box}, o)$ // 2D RRT
**if** $L_{tempBox} = \emptyset$ **then** // Object is probably stuck around another particular object
$b \leftarrow o.getObjectWithMostCollision()$
$L_{tempBox} \leftarrow boxAgent.rrt(s_{box}, o, b)$ // 4D RRT
**if** $L_{tempBox} = \emptyset$ **then**
**continue** // Still cannot solve, skip this and go to next stage
**end if**
**end if**
**for all the** $s_{tempBox} \in L_{tempBox}$ **do**
$s_{box}.setGoal(s_{tempBox})$
$s_{box} \leftarrow boxAgent.aStar(s_{box}, 0.001)$
**if** $s_{box} = s_{tempBox}$ **then**
$s_{box}.removeUnresolvedBox(o)$

$L_{box} \leftarrow (L_{box}, boxAgent.constructPath(s_{box}, s_{tempBox}))$
**else**
**break**
**end if**
**end for**
**end for**
**else** // Search completed with Best-First only
$L_{box} \leftarrow boxAgent.constructRoute(s_{box}, s_{boxGoal})$ // Using the state graph M
**end if**

**for all the** $boxAction \in L_{box}$ **do**
$s_{robot} = boxAction.getRobotStateBefore()$
$s_{robotGoal} = boxAction.getRobotStateAfter()$
$s_{robot} \leftarrow robotAgent.aStar(s_{robot}, 0.001)$
**if** $s_{robot} \neq s_{robotGoal}$ **then** // Switch to RRT
$L_{tempRobot} \leftarrow robotAgent.rrt(s_{robot}, s_{robot}.robot)$
**if** $L_{tempRobot} = \emptyset$ **then**
**break** // Still cannot solve, give up and go to next stage
**else**
**for all the** $s_{tempRobot} \in L_{tempRobot}$ **do**
$s_{robot}.setGoal(s_{tempRobot})$
$s_{robot} \leftarrow robotAgent.aStar(s_{robot}, 0.001)$
**if** $s_{robot} \neq s_{tempRobot}$ **then**
**break** // Still cannot solve, give up and go to next stage
**else**
$L_{step} \leftarrow$
$boxAgent.constructRoute(s_{robot}, s_{tempRobot})$
$L_{bot} \leftarrow (L_{bot}, L_{step})$
**end if**
**end for**
**end if**
**else** // Path found with primary Best-First
$L_{bot} \leftarrow (L_{bot}, robotAgent.constructPath())$
**end if**
**end for**
**if** $|L_{box}| = 0$ **or** $|L_{bot}| = 0$ **then**
**return** // Either subagent totally fails, will not write to file, may happen if no solutions present
**end if**
$writeToFile(L_{bot})$
**Output:** file containing overall robot path (or null if agent totally fails)

---

# 3. Well-performed Scenarios

The agent is mostly capable of routing any case with at least one solution, with a specified error of 0.0001. The speed is determined by the amount of the objects (e.g. time complexity in iterations) and the complexity of the layout (heuristic estimations). Other than a smaller number of objects (dimensionality), the following cases are considered quicker for the agent to solve.

## 3.1 Sparse distribution of / generous gap between objects

The agent can handle the case when the gaps between objects are large (generally larger than 1.5 times the width of the largest object). That is because there are more solutions existing in a sparse distributed map and when the box subagent is trying to conduct Best-First search, it is less likely to encounter a collision. The agent can solve the problem quicker if the distribution of boxes is sparser. For example, in Figure 2, the right map is considered less sparse than the left map thus it can be solved in a less amount of time.


**Figure 1 – Sparse maps**

Figure 3 shows the solving process for each of the map from Figure 2. The left map can be solved almost 20 seconds faster than the right map as right map triggers RRT search one more time, which generally takes much more time than Best-First search.

```
Solving...
Solve stage 1..
Stage solved to most close solution with 176 steps.
Solve stage 2..
Stage solved to most close solution with 186 steps.
Solve stage 3..
Stage solved to most close solution with 268 steps.
Solve stage 4..
Stage solved to most close solution with 252 steps.
Solve stage 5..
Stage solved to most close solution with 297 steps.
Solve stage 6..
Stage solved to most close solution with 295 steps.
Box Stage Done.
1 out of 7 goals not reached, Perform further search one by one.
Further search for 1 start..
RRT start..
RRT solved with 8023 samples.
RRT finishes.
RRT works, calculate actions for robot, total of 120 size
120 / 120
This RRT is done
Start to calculate robot routes..
90/1189
RRT start
RRT solved with 810 samples.
RRT finish
1189/1189
Done.
Time: 45.30 s
```

```
Solving...
Solve stage 1..
Stage solved to most close solution with 174 steps.
Solve stage 2..
Stage solved to most close solution with 193 steps.
Solve stage 3..
Stage solved to most close solution with 278 steps.
Solve stage 4..
Stage solved to most close solution with 214 steps.
Solve stage 5..
Stage solved to most close solution with 268 steps.
Solve stage 6..
Stage solved to most close solution with 264 steps.
Box Stage Done.
2 out of 7 goals not reached, Perform further search one by one.
Further search for 1 start..
RRT start..
RRT solved with 5816 samples.
RRT finishes.
RRT works, calculate actions for robot, total of 62 size
62 / 62
This RRT is done
Further search for 2 start..
RRT start..
RRT solved with 3260 samples.
RRT finishes.
RRT works, calculate actions for robot, total of 60 size
60 / 60
This RRT is done
Start to calculate robot routes..
74/1720
RRT start
RRT solved with 1434 samples.
RRT finish
1720/1720
Done.
Time: 63.65 s
```

**Figure 2 – Solving process for sparse maps**

## 3.2 Best-First failure scenario

Because the heuristic of Best-First search is based on the direct distance from box position to goal position, it is likely that Best-First will fail when the box must move away from the goal (increase heuristic) for a certain number of steps before it can reach the goal. The example in Figure 4 indicates this problem.

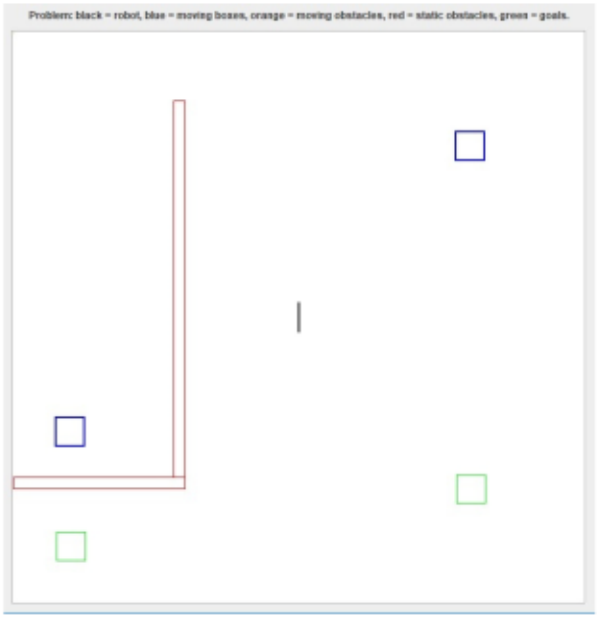


**Figure 3 – Best-First failure scenario of box subagent**

Best-First search will expand and search the state near the obstacle first before moving upwards. It will take a significant large amount of time before Best-First starts to search the “exit” on top of the map. However, the agent will switch to RRT search if Best-First fails to find a solution within a certain amount of computation time, which enables the agent to solve problems that Best-First

cannot solve. The solution of the agent to the problem is shown in Figure 4. The box goes around the static obstacle and reaches the goal.

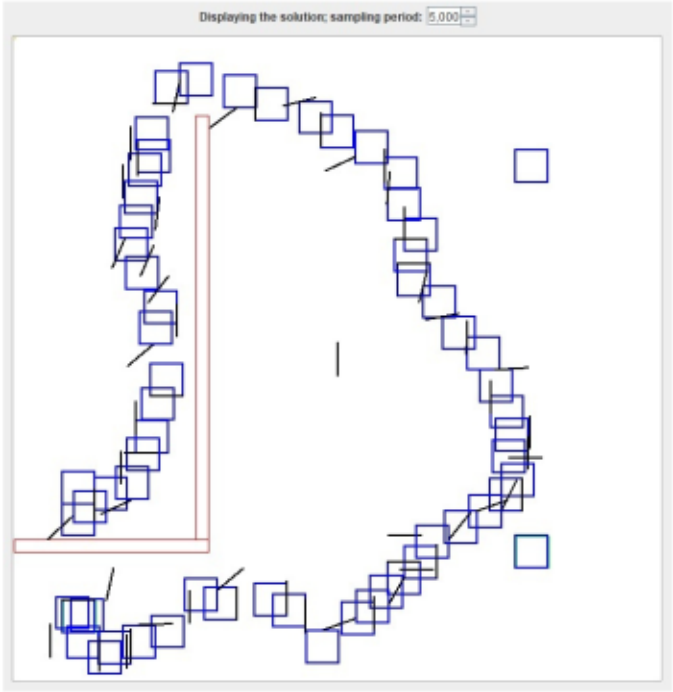


**Figure 4 – Solution for Best-First failure scenario of box subagent**

The same principle also applies to robot subagent. In Figure 6, it is easy for the box subagent to find a route, which is simply pushing the box towards the right.

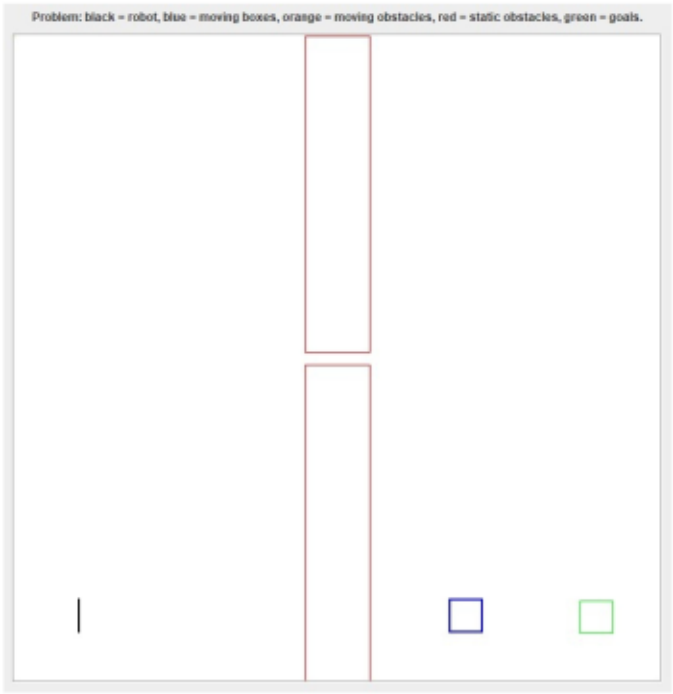


**Figure 5 – Best-First failure scenario of robot subagent**

However, to push the box, the robot must move upwards and sequence through the small gap between two static obstacles, which could take a lot of time if searching in Best-First as the heuristic is based on the direct distance between two position. Therefore, RRT will be used if Best-

First fails to reach the goal within a certain amount of computation time. The solution is shown in Figure 6 after using RRT search.

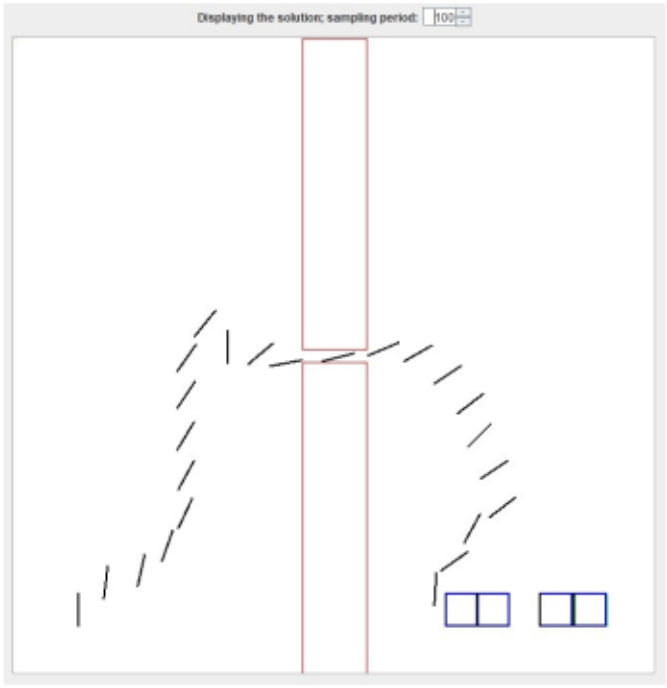


**Figure 6 - Solution for Best-First failure scenario of robot subagent**

## 3.3 Two-Dimension RRT failure scenario

Generally, RRT can always find a solution as long as there are some valid solutions existing in the C-space. However, there is a scenario where the route of the box is blocked by another movable object. For instance, in Figure 7, performing RRT on the box alone (2D RRT) cannot find a solution.

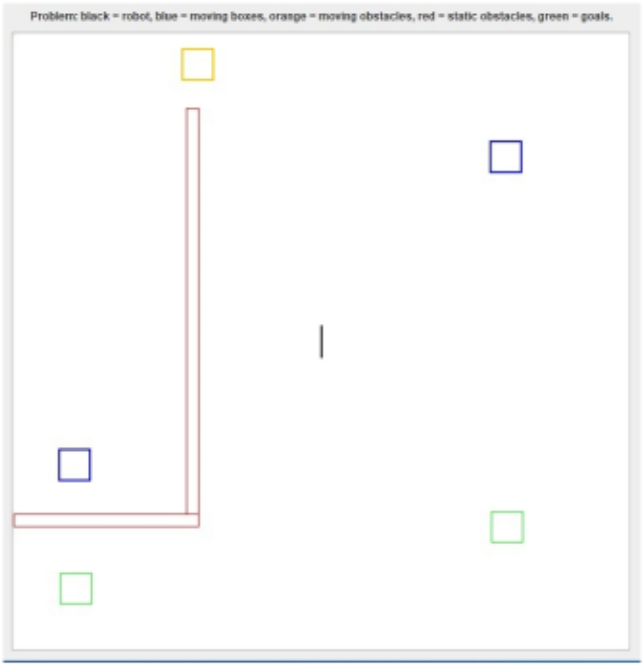

**Figure 7 – 2D RRT failure scenario**

In this case, after RRT fails to find a route for a single box within a certain number of samples, it will give a feedback containing a movable object that causes the most collision in C-space (Problem box) during sampling. Then, the agent will perform RRT search on two objects (Original box and the problem box) at the same time, trying to find a solution. The solution is shown in Figure 8, where the robot moves the two objects at the same time and find a route for the box to reach the goal.


**Figure 8 – Solution for 2D RRT failure scenario**

# 4. Inapplicable Scenarios

## 4.1 No solution

There are some trivial circumstances where solutions are absent, such as: initial robot (without any boxes and goals around) is surrounded by static obstacles; path from any box to its goal is completely blocked by static obstacles.

## 4.2 "Trap" cases

Consider the scenario in Figure 9, the box subagent will find a solution instantly which is pushing the box up by a width to reach the goal. However, this action can never be done by the robot, as the robot can not slide between objects. The correct logic should be pushing one of the movable objects first before the robot can push the box, but this is not checked by the agent and cannot be solved.

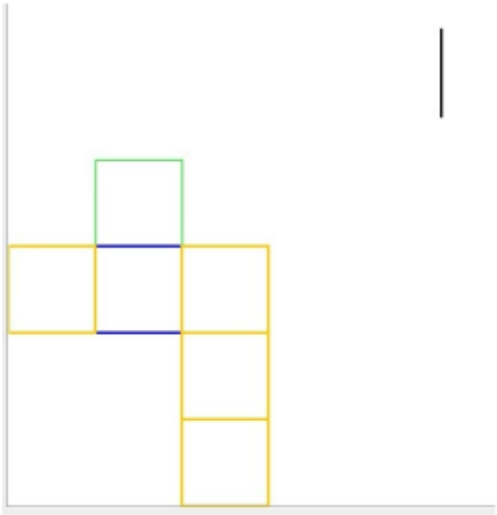

**Figure 9 – "Trap" Case**

## 4.3 RRT limitation cases

RRT algorithm, in short, builds up a linked tree in the free space, and as the tree grows, eventually the tree will approach the goal. The way RRT grows is by extending towards a random sample from the closest node in the tree. However, the characteristic of RRT can be its limitation as well. In Figure 10, where the robot is trying to move from configuration A (In black) to configuration B (In red):

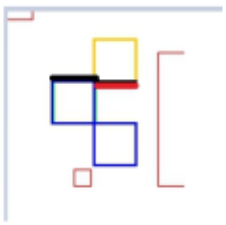

**Figure 10 – RRT limitation**

The two configurations have the same y position and same angle. The only difference between A and B is the x position. As the C-space for moving robot is in 3D: x position, y position and angle (z position), the distance between the start configuration and the goal configuration is extremely small. To let the tree grows towards to goal configuration, a random sample must be sampled near the goal configuration in C-space. However, a sample nears goal configuration in most cases are closer to the start configuration, resulting in the tree grows from the start configuration towards to goal configuration, which will cause collision at all time. Figure 11 below shows the plot of C-space after 100,000 samples in 3-dimension, where the red dot represents the goal configuration, the yellow dot represents the start configuration and blue dots represent samples in the tree. When being viewed from z axis (x-y view), it matches the current environment fairly well, as shown in Figure 12.

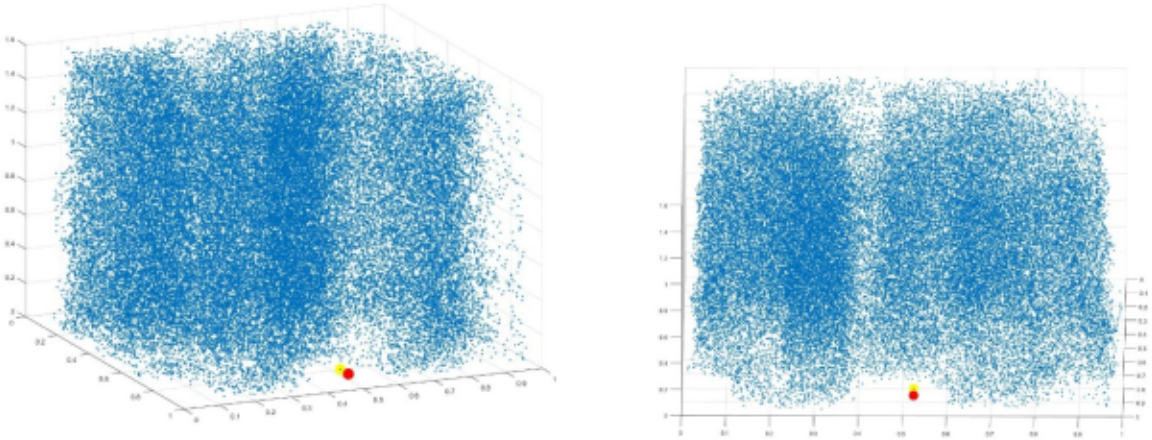

**Figure 11 – 100,000 samples in c-space (Plot in MATLAB)**

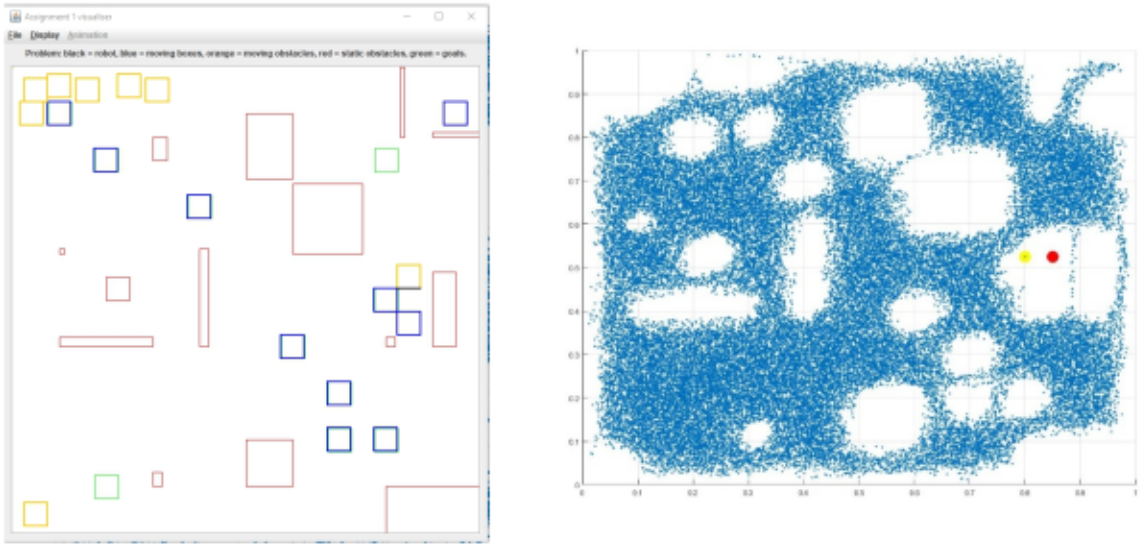

**Figure 12 – C-space in z-axis**

There are a few samples to the right of the goal configuration. However, they are only close to the goal without considering the z axis value. In 3D C-space, the goal configuration is closer to the start configuration which makes the tree extremely hard to grow towards to goal configuration. Only a very small number of samples can grow the tree closer towards to goal, which means it will need a significant large number of samples to solve this problem. Though in the agent, the z-axis has been compressed by 10 times ($k = 0.1$), it will still take very long to solve this kind of scenario.

## 4.4 Higher dimension RRT

In Figure 13, A 4-Dimension RRT can not move the box to the goal as there're two moving obstacles blocking the "exit". Higher dimension RRT takes much more time than 4D RRT search and therefore is blocked by the agent. The agent will give up a goal if 4D RRT search fails to find a solution.

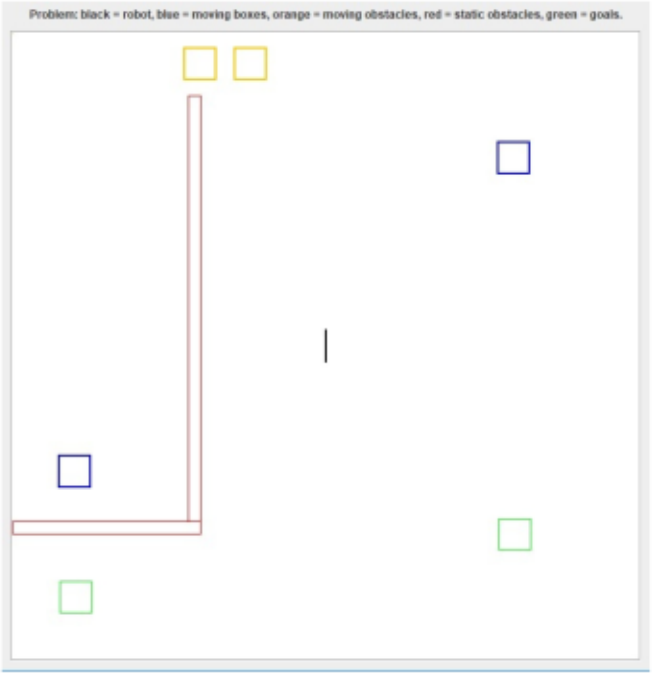


**Figure 13 – Higher Dimension RRT search problem**

## 4.5 Other timed-out cases – too many boxes to move

The high dimensionality leads to more branches and greater depth of the tree, which indicate longer iterations in some of the data structure operations (whose complexity is not $O(1)$, in particular) involved. The whole task may be done eventually but depending on the hardware, the duration may exceed the required time.