Robot motion planning workshop 2020: two-player turn based game

You are given a team of n blue robots competing against a team of n red robots, where $1 \leq n \leq 3$. All the disk robots have the same radius s. The robots operate in a workspace cluttered with polygonal obstacles.

The workspace also includes m randomly placed triangular coupons whose values are $c_1, c_2, \ldots, c_m \geq 0$. The triangles are of variable sizes. These triangles are not obstacles and robots can freely move through them.

Goal: The goal of each team is to reach the (unlabeled) target positions from the start positions as soon as possible, while collecting coupons whose overall value is at least $0.5 \sum_{i=1}^{m} c_i$.

When a robot touches a coupon it collects the value c_i, of the coupon. The sum of values collected by the robot is written on the robot itself.

You are encouraged to plan a winning strategy. Note that you may also want to consider blocking the opponent.

A turn: At each turn the player is given the following parameters: a distance value $d \geq 0$, number of seconds $t \geq 0$, and the current positions of the robots of the other team.
The player is allowed to move her team of robots, such that the sum of distances travelled by all robots in this turn is at most d.
She will have to complete **the computation** of her turn in at most t seconds.

Robots of the team can move simultaneously. Consider two points in the C-space (two configurations) such that the path corresponding to the linear interpolation between them is valid. This path corresponds to a fixed time step. During this time step each robot moves in a fixed velocity, but the velocities of different robots may vary.

Notice that the actual velocity of the robots in the team is not an important factor in the game. What matters only is the relative velocity of robots in the same team that move simultaneously, and this is described above.

When the program that runs the game checks your move for the current turn, the validity of every step (transition between two configurations) is tested, as well as the sum of distances traveled by the team of robots. If the step is invalid or if the the sum of distances due to this step exceeds the specified max distance d, the turn will terminate and the robots will stay in the last valid configuration.

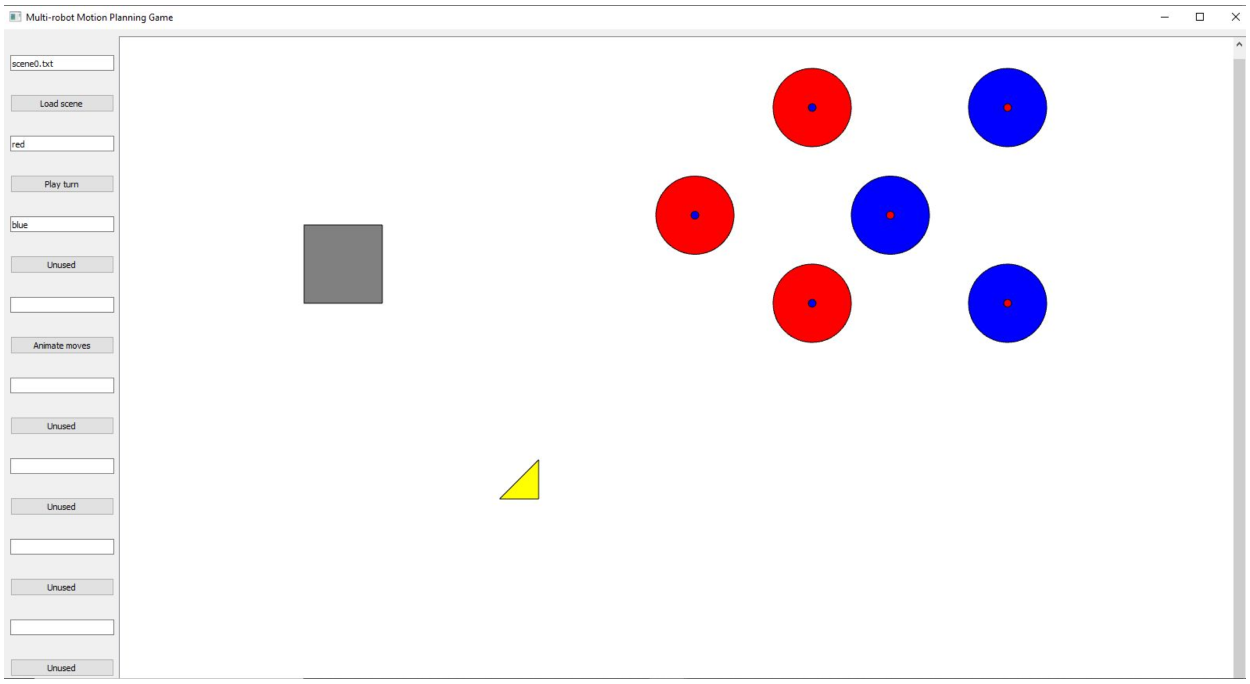
Breaking a deadlock: If no progress is made after a number of turns, the game is stopped. The winner is determined as follows: (i) the sum of shortest paths of the robots to the goal region, and (ii) the sum of collected coupon values.

The program will be developed in python, using the provided library and GUI.
Collision detection with the obstacles is already implemented. The following are provided:

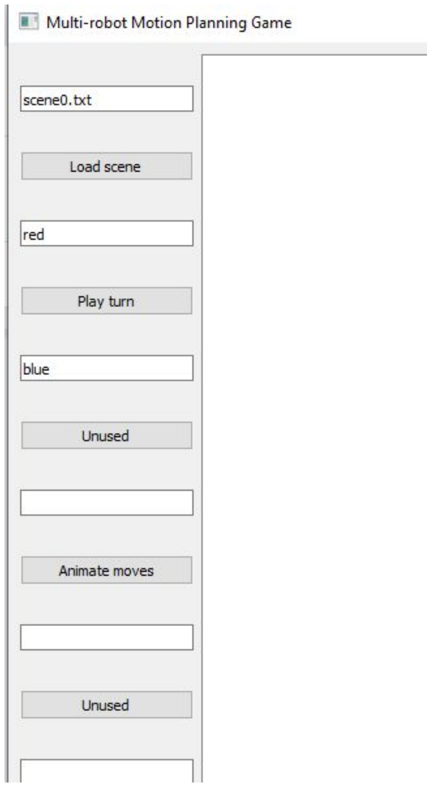
- A procedure testing whether a certain configuration is valid.
- A verifier outputting whether a certain path, represented as an ordered list of configurations (points in the C-space), is valid. Note that the transition between two consecutive configurations in this list corresponds to a linear interpolation between them.

An example of the GUI with a very simple scene:

- The robots are at their start positions. The target positions of the robot centers are marked as blue/red dots. In this example the teams need to switch positions.
- Coupons appear as yellow triangles.
- Obstacles appear as gray polygons.



Explaining the GUI:



- Enter the name of the scene file, e.g., scene0.txt, in the upper text box. (The format of a scene file is explained next.)
- Enter the name of the py file containing the initialize & play_turn functions (explained next) for the red team (this example uses red.py)
- Enter the name of the py file containing the initialize & play_turn functions (explained next) for the blue team (this example uses blue.py)
- Click the “Load scene” button to load the scene and to call the initialize functions of the two teams
- Click the “Play turn” button to play the turn of the current team
- Clicking the “Animated moves” button displays the animation of all turns played since the last time this button was clicked (or from the beginning of the game, if the button was never clicked before)

Structure of a scene file:

1. The first line---beginning with “N”---specifies the number *n* of robots in a team.
   For example “N 3” for 3 robots in each team.
2. The next line---beginning with “OBSTACLES”---specifies the number of obstacles.
   For example “OBSTACLES 1” for a scene with a single obstacle
3. The next line---beginning with “BONUSES”---specifies the number of coupons.
   For example “BONUSES 1” for a scene with a single coupon.
4. The next line---beginning with “RADIUS”---specifies the radius of each robot as a rational number
   For example “RADIUS 1/1” for robots with radius 1.
5. The next line---beginning with “TIME”---specifies the time for initialization.
   Note that: (i) currently the same value is used as the time budget for every turn. However, in the future we may pass different time values when calling your play_turn functions (ii) Currently we do not verify that you complete your computation within the given time budget. This will be changed in the future.
6. The next line---beginning with “D”---specifies the maximal distance d for travelling allowed in one turn. The sum of distances travelled by all robots in the team in one turn should be at most d.
7. The next *n* lines specify the start positions of the red team. Each line is composed of two rational numbers for the x and y coordinates of the center of a specific robot.
   For example: 10/1 49/4
8. The next *n* lines specify the start positions of the blue team. Each line is composed of two rational numbers for the x and y coordinates of the center of a specific robot.
9. The next *n* lines specify the goal positions of the red team. Each line is composed of two rational numbers for the x and y coordinates of the center of a specific robot.
10. The next *n* lines specify the goal positions of the blue team. Each line is composed of two rational numbers for the x and y coordinates of the center of a specific robot.
11. The next lines---beginning with “O”---specify the polygonal obstacles. Each obstacle is defined in a separate line as follows:
    - The first integral value represents the number *k* of vertices of the obstacle
    - It is followed by *k* pairs of coordinates, separated by spaces, representing the vertices of the polygonal obstacle in a counter-clockwise order.
12. The next lines---beginning with “O”---specify the polygonal obstacles. Each obstacle is defined in a separate line as follows:
    - The first integral value represents the number *k* of vertices of the obstacle
    - It is followed by *k* pairs of coordinates, separated by spaces, representing the vertices of the polygonal obstacle in a counter-clockwise order.
13. The next lines---beginning with “B”---specify the triangular coupons. Each coupon is defined in a separate line as follows:
    - The first value (3) indicates the coupon is a triangle.
    - It is followed by *3* pairs of coordinates, separated by spaces, representing the vertices of the triangle in a counter-clockwise order.

The functions that you should implement:
In the beginning you'll have time for preprocessing.
To do so you should implement a function named ***initialize*** that gets a python list as input.
The list contains the following items:

1. t_preprocess (int), specifies the number of seconds you have for preprocessing
2. robot_radius (FT), specifies the radius of a robot
3. team_start_positions (list of Point_2 objects), specifies the current positions of the playing team robot centers. The i-th item represents the start position of the i-th robot as Point_2.
4. opponent_start_positions (list of Point_2 objects), specifies the current positions of the opponent team robot centers. The i-th item represents the start position of the i-th opponent robot as Point_2.
5. team_goals (list of Point_2 objects), specifies the goal positions of the centers of the playing team.
6. opponent_goals (list of Point_2 objects), specifies the goal positions of the centers of the opponent team.
7. obstacles (list of Polygon_2 objects), specifies the list of polygonal obstacles
8. coupons (list of tuples, each tuple is a pair of Polygon_2 and FT), specifies the list of coupons. Each coupon is represented as a pair, whose first element is a Polygon_2 object representing the triangle, and the second element is the value of the coupon.
9. data (list): This list is initially empty. During the preprocessing stage you may populate this list with objects of your choice that you'll receive again at every turn.

The initialize function is called when loading a scene using the GUI (after pressing the button "Load scene").

You should implement a function named ***play_turn*** that is called at every turn. This function gets a python list as its input.
The list contains the following items:

1. path (list of lists of Point_2), the function play_turn should populate this list with items representing configurations. Each configuration is a list of size *n* composed of Point_2 objects, where the i-th object describes the configuration of robot i. The overall sequence of configurations represents a path for the robot team in this turn.
2. t (int), specifies the number of seconds you have for this turn.
3. d (float), specifies the sum of distances the team's robots are allowed to move in this turn.
4. team_status (list of tuples, each tuple is a pair of Point_2 and FT), specifies the list of current positions of the playing team centers as well as the sum of coupon values collected by each robot. The i-th tuple is composed of a (i) a Point_2 object for the position of the i-th robot and (ii) a value of the sum of coupons collected by the i-th robot.
5. opponent_status (list of tuples, each tuple is a pair of Point_2 and FT), specifies the list of current positions of the opponent team centers as well as the sum of coupon values collected by each robot. The i-th tuple is composed of a (i) a Point_2 object for the

position of the i-th opponent robot and (ii) a value of the sum of coupons collected by the i-th robot.

6. coupons (list of tuples, each tuple is a pair of Polygon_2 and FT), specifies the list of the coupons. Each coupon is represented as a pair, whose first element is a Polygon_2 object representing the triangle, and the second element is the value of the coupon. Important: the value of a coupon that has already been collected is 0.
7. data (list): The list of objects that were created during the preprocessing stage.

Useful information about the CGAL Python bindings:

The Python bindings currently include (most of):

- CGAL Arrangements [link]
- Kernel [link]
- Minkowski sums [link]
- 2D Regularized Boolean Set-Operations [link]
- Triangulations [link]
- dD Spatial Searching [link]

Important:

a) The bindings are currently compiled with circle_segment_traits and with an exact kernel (exact predicates + exact constructions).

b) The CGAL kd-tree (part of the dD Spatial Searching) is compiled for 6D points (this is fixed, due to compilation issues). In order to represent a six-dimensional point we will use CGAL Point_d. Suppose that the number of robots in each team is two, then every point in the kdtree should be represented as a six-dimensional point (Point_d), where the last two coordinates are 0.

Iterators and circulators in CGAL are implemented as iterators in the Python bindings.
For example, given an arrangement *arr* one can iterate over its vertices using a for loop.
In C++ one should iterate from the begin-iterator *arr.vertices_begin()* until reaching the past-the-end iterator *arr.vertices_end().* In Python the following code should be used instead:

```
for v in arr.vertices():
    #v is of type Vertex
```

Instead of circulators we use iterators. In the following example we demonstrate how to iterate over the halfedges around a vertex *s :*

```
#s is of type Vertex
for e in s.incident_halfedges():
    #e is of type Halfedge
```