robot game tutorial

Rowan Hargreaves

2014-03-12

Contents

Introduction

This is a rather brief tutorial to robot game see robotgame.net

The aim is to quickly build some basic bots to test using `rgkit` testing kit introducing the parts of the robot game `rg` library as we go.

We will create three bots of increasing complexity:

- `guard_bot.py`
- `random_walker_bot.py`
- `seeker_bot.py`

We will develop them in stages and you can copy and paste the code snippets as we go, but you may have to think about where exactly to put the code.

Then finally you can work on your own or with others to improve your bots and then pitch them in battle with each other.

Creating a guard bot

Now we will write our simplest bot `guard_bot.py`. So if you haven't already done this when testing `rgkit` create a file called `guard_bot.py` and enter the following code:

```
"""
guard_bot.py

"""

class Robot:
    def act(self, game):
        return ['guard']
```

Now run `rgrun guard_bot.  guard_bot.py`.

What happens?

Attack and random move

OK so not a lot happened there. Let's make a bot that moves around randomly and attacks any enemies in range. So create the file `random_walker_bot.py` and put the following code in it:

```
"""

random_walker_bot_v2.0.py

This bot attacks any enemies that are close by or otherwise moves
around randomly.

"""

import random

import rg

class Robot:

    def act(self, game):
        # If there are enemies around, attack them.
        for loc, bot in game.robots.iteritems():
            if bot.player_id != self.player_id:
                if rg.dist(loc, self.location) <= 1:
```

```
                return ['attack', loc]

        # Otherwise find possible moves around the bot and then
        # randomly pick one.
        moves = rg.locs_around(self.location,
                               filter_out=('invalid', 'obstacle'))

        if len(moves) > 0:
            return ['move', random.choice(moves)]

        # If no moves available just guard
        return ['guard']
```

So this code makes use of some of the functions provided by the `rg` library. For more information look here: http://robotgame.net/rgdocs

Now let's see how this bot does against the guard bot. Run: `rgrun guard_bot.py random_walker.py`

How does it do?

Well the bots are at least attacking each other, so that's good.

But their motion is not so random right?

Not quite sure how the code is being called by `rgrun` but we can make it behave more randomly by making a call to `random.seed()` somewhere in the `act` method.

Now call it again. Any better? What else is wrong?

(BTW You can speed up the game replay speed using the controls on the visualiser.)

Stop blocking our own team

So if you watch the motion of the bots when they come close to each other sometimes they don't appear to move when perhaps they could. This is because the `filter_out` keyword argument in `rg.locs_around()` is not filtering for our own bots.

So we need to find out where all our own bots are. Using this list comprehension, we create a list `own_locs`:

```
own_locs = [loc for loc, bot in game.robots.iteritems()
                    if bot.player_id == self.player_id]
```

And then we can filter our possible moves in `moves` using another list comprehension:

```
moves = [loc for loc in moves if loc not in own_locs]
```

Now try running `random_walker_bot.py` again.

Are the bots avoiding their buddies?

Well, no, not exactly. If their team mates don't move during that game turn then they avoid them but if they move into a location that might cause a problem we don't know that they are there as the locations stored in the `game` dictionary are only updated when all the bots have declared their moves for the turn.

Remember that during a turn `act()` is called for each bot alive in the game.

So how are we going to know where our bots are going to be?

We can create a class variable to store where we have asked our bots to move to.

So outside of `act()`, but within the `Robot` class, create a class variable called `new_locs` and set it to the empty list:

```
new_locs = []
```

And then in `act()`, just before we `return ['attack', loc']` append the bot that is attacking's location by

```
self.new_locs.append(self.location)
```

And now we no longer need to create the `own_locs` list and when we create the `moves` list we can replace it with `self.new_locs` like so

```
moves = [loc for loc in moves if loc not in self.new_locs]
```

We are not quite done as we need to append our choice of move to `self.new_locs` and if no moves are available and our bot guards we need to append its location.

So change this code:

```
if len(moves) > 0:
  return ['move', random.choice(moves)]

# If no moves available just guard
return ['guard']
```

to

```
if len(moves) > 0:
  choice = random.choice(moves)
  self.new_locs.append(choice)
  return ['move', choice]

self.new_locs.append(self.location)
return ['guard']
```

Phew! That was a fair bit of work. Now run it!

What happens?

Some of the bots seem to be getting stuck and not moving anywhere. Why is this?

It's because we appending more and more locations to `self.new_locs` and so there are less and less places for them to move to!

So we need to clear out `self.new_locs` every turn.

We can do this easily by creating another class variable to store the last turn number:

```
last_turn = 0
```

And then in `act()` we need to figure out if a new turn has just happened so we add at the start of this method:

```
# When a new turn is started need to clear the new_locs list.
if self.last_turn != game.turn:
  self.new_locs = []
  self.last_turn = game.turn
```

NB the turn number is stored in `game.turn`.

So now try and run it again, all well?

So here is our final `random_walker_bot.py`:

```
"""

random_walker_bot_v2.3.1.py

This bot randomly walks around and attacks any enemies that are close
by.

Avoids collisions with own bots by not moving into space where the bots will move to.
```

```
"""

import random

import rg

class Robot:
  new_locs = []
  last_turn = 0

  def act(self, game):
    # When a new turn is started need to clear the new_locs list.
    if self.last_turn != game.turn:
      self.new_locs = []
      self.last_turn = game.turn

    # If there are enemies around, attack them.
    for loc, bot in game.robots.iteritems():
      if bot.player_id != self.player_id:
        if rg.dist(loc, self.location) <= 1:
          self.new_locs.append(self.location)
          return ['attack', loc]


    random.seed()
    # Otherwise move around pick a direction that can randomly
    # move to and move there.
    moves = rg.locs_around(self.location, filter_out=('invalid', 'obstacle'))
    moves = [loc for loc in moves if loc not in self.new_locs]

    if len(moves) > 0:
      choice = random.choice(moves)
      self.new_locs.append(choice)
      return ['move', choice]

    self.new_locs.append(self.location)
    return ['guard']
```

Seek and destroy

Now we are going to create a new bot - `seeker_bot.py` - that moves towards the closest enemy and when close enough attacks it.

We will use `random_walker_bot.py` as a starting point.

The first thing we need to do is to find where are our adversary's bots are and

how far they are from our bot.

Luckily the `rg` library has some useful functions for us to use.

So first we create a list of tuples containing each enemy bot's distance from our bot and their location, again using a list comprehension:

```
# Create a list of distances to the enemy bots and their
# locations.
enemy_dist_and_locs = [(rg.dist(loc, self.location), loc)
                                for loc, bot in game.robots.iteritems() if
                                bot.player_id != self.player_id ]
```

And then we sort them so that the closest are first and create variables to store the closest distance and location:

```
# Now sort by distance.
enemy_dist_and_locs = sorted(enemy_dist_and_locs)

closest_dist, closest_loc = enemy_dist_and_locs[0]
```

Now we need to use `closest_dist` and `closest_loc` in the rest of the code to get our bot to move towards our closest enemy and attack it.

The attack part of the code should be changed to

```
if closest_dist <= 1 :
  self.new_locs.append(self.location)
  return ['attack', closest_loc]
```

And now instead of moving randomly we can try to move towards `closest_loc`, luckily `rg` has a nice method for us:

```
move_towards = rg.toward(self.location, closest_loc)
self.new_locs.append(move_towards)
return ['move', move_towards]
```

We can leave the part about moving randomly and guarding in, provided they are at after the new code above in `act()`.

So run it what happens? Do your bots seek and destroy?

The only problem is that in their hurry to engage the enemy the bots don't move around their team mates.

Here's our final `seeker_bot.py`

```
"""

seeker_bot_v3.1.py

This bot seeks out closest enemy and tries to move towards it, and when close enough attacks

Avoids collisions with own bots by not moving into space where the bots will move to.

"""

import random

import rg

class Robot:
    new_locs = []
    last_turn = 0

    def act(self, game):

        # When a new turn is started need to clear the new_locs list.
        if self.last_turn != game.turn:
            self.new_locs = []
            self.last_turn = game.turn


        # Create a list of distances to the enemy bots and their
        # locations.
        enemy_dist_and_locs = [(rg.dist(loc, self.location), loc)
                               for loc, bot in game.robots.iteritems() if
                               bot.player_id != self.player_id ]

        # Now sort by distance.
        enemy_dist_and_locs = sorted(enemy_dist_and_locs)

        closest_dist, closest_loc = enemy_dist_and_locs[0]

        if closest_dist <= 1 :
            self.new_locs.append(self.location)
            return ['attack', closest_loc]

        move_towards = rg.toward(self.location, closest_loc)
        self.new_locs.append(move_towards)
        return ['move', move_towards]

        random.seed()
```

```
        # Otherwise move around pick a direction that can randomly
        # move to and move there.
        moves = rg.locs_around(self.location, filter_out=('invalid', 'obstacle'))
        moves = [loc for loc in moves if loc not in self.new_locs]

        if len(moves) > 0:
            choice = random.choice(moves)
            self.new_locs.append(choice)
            return ['move', choice]

        self.new_locs.append(self.location)
        return ['guard']
```

Room for improvement - dojo challenge

As you can see from our really basic bots there is plenty of room for improvement!

So the dojo challenge is to make the best bot you can!

Here are some ideas:

- Be able to move around obstacles better (path finding)
- Move off the spawn squares
- Suicide
- Cooperative attacks

For more info, look at:

- library documentation
- For time limits restrictions on what is allowed on the real game server see here

---

Special thanks to Paweł Widera for his bottle app code for incrementally revealing the tutorial and his help with it.
