

**Test Preview****TestSummary.txt: 1/1****Asia Belfiore - ab6124:a5**

```
1: Test Preview: Summary for ab6124 of a5
2: -----
3:
4:   Public Tests:
5:     Question 0:  1.0 / 1.0
6:     Question 1:  4.0 / 4.0
7:     Question 2:  5.0 / 5.0
8:     Question 3:  5.0 / 5.0
9:
10: Git Repo: git@gitlab.doc.ic.ac.uk:lab2425_autumn/Coursework1_ab6124.git
11: Commit ID: a6205
```

## Test Preview

coursework1.py: 1/14

Asia Belfiore - ab6124:a5

```

1: #!/usr/bin/env python
2: # coding: utf-8
3:
4: # ## **70028 - Reinforcement Learning: Coursework 1**
5: # ### Belfiore Asia, 02129867
6: # ---
7:
8: # In[1]:
9:
10:
11: import numpy as np
12: import random
13: import matplotlib.pyplot as plt # Graphical library
14:
15:
16: # In[2]:
17:
18: # # Coursework 1 :
19: # See pdf for instructions.
20:
21: # In[3]:
22:
23:
24: # WARNING: fill in these two functions that will be used by the auto-marking
script
25: # [Action required]
26:
27: def get_CID():
28:     return "02129867" # Return your CID (add 0 at the beginning to ensure it is 8
digits long)
29:
30: def get_login():
31:     return "ab6124" # Return your short imperial login
32:
33:
34: # ## Helper class
35:
36: # In[4]:
37:
38:
39: # This class is used ONLY for graphics
40: # YOU DO NOT NEED to understand it to work on this coursework
41:
42: class GraphicsMaze(object):
43:
44:     def __init__(self, shape, locations, default_reward, obstacle_locs,
absorbing_locs, absorbing_rewards, absorbing):
45:
46:         self.shape = shape
47:         self.locations = locations
48:         self.absorbing = absorbing
49:
50:         # Walls
51:         self.walls = np.zeros(self.shape)
52:         for ob in obstacle_locs:
53:             self.walls[ob] = 20
54:
55:         # Rewards
56:         self.rewarders = np.ones(self.shape) * default_reward
57:         for i, rew in enumerate(absorbing_locs):
58:             self.rewarders[i] = 10 if absorbing_rewards[i] > 0 else -10
59:
60:         # Print the map to show it
61:         self.paint_maps()
62:
63:     def paint_maps(self):

```

## Test Preview

coursework1.py: 2/14

Asia Belfiore - ab6124:a5

```

64: """
65: Print the Maze topology (obstacles, absorbing states and rewards)
66: input: /
67: output: /
68: """
69: plt.figure(figsize=(15,10))
70: plt.imshow(self.walls + self.rewarders)
71: plt.show()
72:
73: def paint_state(self, state):
74:     """
75:     Print one state on the Maze topology (obstacles, absorbing states and
rewards)
76:     input: /
77:     output: /
78:     """
79:     states = np.zeros(self.shape)
80:     states[state] = 30
81:     plt.figure(figsize=(15,10))
82:     plt.imshow(self.walls + self.rewarders + states)
83:     plt.show()
84:
85: def draw_deterministic_policy(self, Policy):
86:     """
87:     Draw a deterministic policy
88:     input: Policy {np.array} -- policy to draw (should be an array of values
between 0 and 3 (actions))
89:     output: /
90:     """
91:     plt.figure(figsize=(15,10))
92:     plt.imshow(self.walls + self.rewarders) # Create the graph of the Maze
93:     for state, action in enumerate(Policy):
94:         if(self.absorbing[0,state]): # If it is an absorbing state, don't plot any
action
95:             continue
96:         arrows = [r"$\uparrow$",r"$\rightarrow$", r"$\downarrow$", r"$\leftarrow$"]
] # List of arrows corresponding to each possible action
97:         action_arrow = arrows[action] # Take the corresponding action
98:         location = self.locations[state] # Compute its location on graph
99:         plt.text(location[1], location[0], action_arrow, ha='center', va='center')
# Place it on graph
100:         plt.show()
101:
102: def draw_policy(self, Policy):
103:     """
104:     Draw a policy (draw an arrow in the most probable direction)
105:     input: Policy {np.array} -- policy to draw as probability
106:     output: /
107:     """
108:     deterministic_policy = np.array([np.argmax(Policy[row,:]) for row in
range(Policy.shape[0])])
109:     self.draw_deterministic_policy(deterministic_policy)
110:
111: def draw_value(self, Value):
112:     """
113:     Draw a policy value
114:     input: Value {np.array} -- policy values to draw
115:     output: /
116:     """
117:     plt.figure(figsize=(15,10))
118:     plt.imshow(self.walls + self.rewarders) # Create the graph of the Maze
119:     for state, value in enumerate(Value):
120:         if(self.absorbing[0, state]): # If it is an absorbing state, don't plot
any value
121:             continue
122:         location = self.locations[state] # Compute the value location on graph

```

```

Test Preview      coursework1.py: 3/14      Asia Belfiore - ab6124:a5
123:     plt.text(location[1], location[0], round(value,2), ha='center', va='center')
) # Place it on graph
124:     plt.show()
125:
126:     def draw_deterministic_policy_grid(self, Policies, title, n_columns, n_lines):
127:         """
128:         Draw a grid representing multiple deterministic policies
129:         input: Policies {np.array of np.array} -- array of policies to draw (each
should be an array of values between 0 and 3 (actions))
130:         output: /
131:         """
132:         plt.figure(figsize=(20,8))
133:         for subplot in range (len(Policies)): # Go through all policies
134:             ax = plt.subplot(n_columns, n_lines, subplot+1) # Create a subplot for
each policy
135:             ax.imshow(self.walls+self.rewarders) # Create the graph of the Maze
136:             for state, action in enumerate(Policies[subplot]):
137:                 if(self.absorbing[0,state]): # If it is an absorbing state, don't plot
any action
138:                     continue
139:                 arrows = [r"$\uparrow$",r"$\rightarrow$", r"$\downarrow$", r"$\leftarrow$"] # List of arrows corresponding to each possible action
140:                 action_arrow = arrows[action] # Take the corresponding action
141:                 location = self.locations[state] # Compute its location on graph
142:                 plt.text(location[1], location[0], action_arrow, ha='center', va='center')
) # Place it on graph
143:                 ax.title.set_text(title[subplot]) # Set the title for the graph given as
argument
144:                 plt.show()
145:
146:     def draw_policy_grid(self, Policies, title, n_columns, n_lines):
147:         """
148:         Draw a grid representing multiple policies (draw an arrow in the most
probable direction)
149:         input: Policy {np.array} -- array of policies to draw as probability
150:         output: /
151:         """
152:         deterministic_policies = np.array([[np.argmax(Policy[row,:]) for row in
range(Policy.shape[0])] for Policy in Policies])
153:         self.draw_deterministic_policy_grid(deterministic_policies, title,
n_columns, n_lines)
154:
155:     def draw_value_grid(self, Values, title, n_columns, n_lines):
156:         """
157:         Draw a grid representing multiple policy values
158:         input: Values {np.array of np.array} -- array of policy values to draw
159:         output: /
160:         """
161:         plt.figure(figsize=(20,8))
162:         for subplot in range (len(Values)): # Go through all values
163:             ax = plt.subplot(n_columns, n_lines, subplot+1) # Create a subplot for
each value
164:             ax.imshow(self.walls+self.rewarders) # Create the graph of the Maze
165:             for state, value in enumerate(Values[subplot]):
166:                 if(self.absorbing[0,state]): # If it is an absorbing state, don't plot
any value
167:                     continue
168:                 location = self.locations[state] # Compute the value location on graph
169:                 plt.text(location[1], location[0], round(value,1), ha='center', va=
'center') # Place it on graph
170:                 ax.title.set_text(title[subplot]) # Set the title for the graoh given as
argument
171:                 plt.show()
172:
173:
174: # ## Maze class

```

```

Test Preview      coursework1.py: 4/14      Asia Belfiore - ab6124:a5
175:
176: # In[5]:
177:
178:
179: # This class define the Maze environment
180:
181: class Maze(object):
182:
183:     # [Action required]
184:     def __init__(self):
185:         """
186:         Maze initialisation.
187:         input: /
188:         output: /
189:         """
190:
191:         # Q0 [Action required]
192:         # Properties set from the CID
193:         self._prob_success = 0.8 + (0.02 * (9.0 - float(get_CID()[6]))) # float
194:         self._gamma = 0.8 + (0.02 * float(get_CID()[6])) # float
195:         self._goal = int(get_CID()[7]) % 4 # integer (0 for R0, 1 for R1, 2 for R2,
3 for R3)
196:
197:         # Build the maze
198:         self._build_maze()
199:
200:
201:         # Functions used to build the Maze environment
202:         # You DO NOT NEED to modify them
203:         def _build_maze(self):
204:             """
205:             Maze initialisation.
206:             input: /
207:             output: /
208:             """
209:
210:             # Properties of the maze
211:             self._shape = (13, 10)
212:             self._obstacle_locs = [
213:                 (1,0), (1,1), (1,2), (1,3), (1,4), (1,7), (1,8),
(1,9), \
214:                 (2,1), (2,2), (2,3), (2,7), \
215:                 (3,1), (3,2), (3,3), (3,7), \
216:                 (4,1), (4,7), \
217:                 (5,1), (5,7), \
218:                 (6,5), (6,6), (6,7), \
219:                 (8,0), \
220:                 (9,0), (9,1), (9,2), (9,6), (9,7), (9,8), (9,9), \
221:                 (10,0)
] # Location of obstacles
222:             self._absorbing_locs = [(2,0), (2,9), (10,1), (12,9)] # Location of
absorbing states
223:             self._absorbing_rewards = [ (500 if (i == self._goal) else -50) for i in
range (4) ]
224:             self._starting_locs = [(0,0), (0,1), (0,2), (0,3), (0,4), (0,5), (0,6),
(0,7), (0,8), (0,9)] #Reward of absorbing states
225:             self._default_reward = -1 # Reward for each action performs in the
environment
226:             self._max_t = 500 # Max number of steps in the environment
227:
228:             # Actions
229:             self._action_size = 4
230:             self._direction_names = ['N','E','S','W'] # Direction 0 is 'N', 1 is 'E' and
so on
231:
232:
233:             # States

```

```

234:     self._locations = []
235:     for i in range (self._shape[0]):
236:         for j in range (self._shape[1]):
237:             loc = (i,j)
238:             # Adding the state to locations if it is no obstacle
239:             if self._is_location(loc):
240:                 self._locations.append(loc)
241:     self._state_size = len(self._locations)
242:
243:     # Neighbours - each line is a state, ranked by state-number, each column is ✓
a direction (N, E, S, W)
244:     self._neighbours = np.zeros((self._state_size, 4))
245:
246:     for state in range(self._state_size):
247:         loc = self._get_loc_from_state(state)
248:
249:         # North
250:         neighbour = (loc[0]-1, loc[1]) # North neighbours location
251:         if self._is_location(neighbour):
252:             self._neighbours[state][self._direction_names.index('N')] = ✓
self._get_state_from_loc(neighbour)
253:         else: # If there is no neighbour in this direction, coming back to current ✓
state
254:             self._neighbours[state][self._direction_names.index('N')] = state
255:
256:         # East
257:         neighbour = (loc[0], loc[1]+1) # East neighbours location
258:         if self._is_location(neighbour):
259:             self._neighbours[state][self._direction_names.index('E')] = ✓
self._get_state_from_loc(neighbour)
260:         else: # If there is no neighbour in this direction, coming back to current ✓
state
261:             self._neighbours[state][self._direction_names.index('E')] = state
262:
263:         # South
264:         neighbour = (loc[0]+1, loc[1]) # South neighbours location
265:         if self._is_location(neighbour):
266:             self._neighbours[state][self._direction_names.index('S')] = ✓
self._get_state_from_loc(neighbour)
267:         else: # If there is no neighbour in this direction, coming back to current ✓
state
268:             self._neighbours[state][self._direction_names.index('S')] = state
269:
270:         # West
271:         neighbour = (loc[0], loc[1]-1) # West neighbours location
272:         if self._is_location(neighbour):
273:             self._neighbours[state][self._direction_names.index('W')] = ✓
self._get_state_from_loc(neighbour)
274:         else: # If there is no neighbour in this direction, coming back to current ✓
state
275:             self._neighbours[state][self._direction_names.index('W')] = state
276:
277:         # Absorbing
278:         self._absorbing = np.zeros((1, self._state_size))
279:         for a in self._absorbing_locs:
280:             absorbing_state = self._get_state_from_loc(a)
281:             self._absorbing[0, absorbing_state] = 1
282:
283:         # Transition matrix
284:         self._T = np.zeros((self._state_size, self._state_size, self._action_size)) ✓
# Empty matrix of domension S*S*A
285:         for action in range(self._action_size):
286:             for outcome in range(4): # For each direction (N, E, S, W)
287:                 # The agent has prob_success probability to go in the correct direction
288:                 if action == outcome:
289:                     prob = 1 - 3.0 * ((1.0 - self._prob_success) / 3.0) # (theoritically ✓

```

```

equal to self.prob_success but avoid rounding error and garanty a sum of 1)
290:         # Equal probability to go into one of the other directions
291:         else:
292:             prob = (1.0 - self._prob_success) / 3.0
293:
294:         # Write this probability in the transition matrix
295:         for prior_state in range(self._state_size):
296:             # If absorbing state, probability of 0 to go to any other states
297:             if not self._absorbing[0, prior_state]:
298:                 post_state = self._neighbours[prior_state, outcome] # Post state ✓
number
299:                 post_state = int(post_state) # Transform in integer to avoid error
300:                 self._T[prior_state, post_state, action] += prob
301:
302:         # Reward matrix
303:         self._R = np.ones((self._state_size, self._state_size, self._action_size)) # ✓
Matrix filled with 1
304:         self._R = self._default_reward * self._R # Set default_reward everywhere
305:         for i in range(len(self._absorbing_rewards)): # Set absorbing states rewards
306:             post_state = self._get_state_from_loc(self._absorbing_locs[i])
307:             self._R[:,post_state,:] = self._absorbing_rewards[i]
308:
309:         # Creating the graphical Maze world
310:         self._graphics = GraphicsMaze(self._shape, self._locations, ✓
self._default_reward, self._obstacle_locs, self._absorbing_locs, ✓
self._absorbing_rewards, self._absorbing)
311:
312:         # Reset the environment
313:         self.reset()
314:
315:
316:     def _is_location(self, loc):
317:         """
318:         Is the location a valid state (not out of Maze and not an obstacle)
319:         input: loc {tuple} -- location of the state
320:         output: _ {bool} -- is the location a valid state
321:         """
322:         if (loc[0] < 0 or loc[1] < 0 or loc[0] > self._shape[0]-1 or loc[1] > ✓
self._shape[1]-1):
323:             return False
324:         elif (loc in self._obstacle_locs):
325:             return False
326:         else:
327:             return True
328:
329:
330:     def _get_state_from_loc(self, loc):
331:         """
332:         Get the state number corresponding to a given location
333:         input: loc {tuple} -- location of the state
334:         output: index {int} -- corresponding state number
335:         """
336:         return self._locations.index(tuple(loc))
337:
338:
339:     def _get_loc_from_state(self, state):
340:         """
341:         Get the state number corresponding to a given location
342:         input: index {int} -- state number
343:         output: loc {tuple} -- corresponding location
344:         """
345:         return self._locations[state]
346:
347:         # Getter functions used only for DP agents
348:         # You DO NOT NEED to modify them
349:     def get_T(self):

```

```

350:     return self._T
351:
352: def get_R(self):
353:     return self._R
354:
355: def get_absorbing(self):
356:     return self._absorbing
357:
358: # Getter functions used for DP, MC and TD agents
359: # You DO NOT NEED to modify them
360: def get_graphics(self):
361:     return self._graphics
362:
363: def get_action_size(self):
364:     return self._action_size
365:
366: def get_state_size(self):
367:     return self._state_size
368:
369: def get_gamma(self):
370:     return self._gamma
371:
372: # Functions used to perform episodes in the Maze environment
373: def reset(self):
374:     """
375:     Reset the environment state to one of the possible starting states
376:     input: /
377:     output:
378:         - t {int} -- current timestep
379:         - state {int} -- current state of the environment
380:         - reward {int} -- current reward
381:         - done {bool} -- True if reach a terminal state / 0 otherwise
382:     """
383:     self._t = 0
384:     self._state = /
385: self._get_state_from_loc(self._starting_locs[random.randrange(len(self._starting_locs))])
386:     self._reward = 0
387:     self._done = False
388:     return self._t, self._state, self._reward, self._done
389:
390: def step(self, action):
391:     """
392:     Perform an action in the environment
393:     input: action {int} -- action to perform
394:     output:
395:         - t {int} -- current timestep
396:         - state {int} -- current state of the environment
397:         - reward {int} -- current reward
398:         - done {bool} -- True if reach a terminal state / 0 otherwise
399:     """
400:     # If environment already finished, print an error
401:     if self._done or self._absorbing[0, self._state]:
402:         print("Please reset the environment")
403:         return self._t, self._state, self._reward, self._done
404:
405:     # Drawing a random number used for probability of next state
406:     probability_success = random.uniform(0,1)
407:
408:     # Look for the first possible next states (so get a reachable state even if /
409:     probability_success = 0)
410:     new_state = 0
411:     while self._T[self._state, new_state, action] == 0:
412:         new_state += 1
413:     assert self._T[self._state, new_state, action] != 0, "Selected initial state /
should be probability 0, something might be wrong in the environment."

```

```

413:
414:     # Find the first state for which probability of occurrence matches the random /
415: value
416:     total_probability = self._T[self._state, new_state, action]
417:     while (total_probability < probability_success) and (new_state < /
self._state_size-1):
418:         new_state += 1
419:         total_probability += self._T[self._state, new_state, action]
420:     assert self._T[self._state, new_state, action] != 0, "Selected state should /
be probability 0, something might be wrong in the environment."
421:
422:     # Setting new t, state, reward and done
423:     self._t += 1
424:     self._reward = self._R[self._state, new_state, action]
425:     self._done = self._absorbing[0, new_state] or self._t > self._max_t
426:     self._state = new_state
427:     return self._t, self._state, self._reward, self._done
428:
429: # ## DP Agent
430:
431: # In[6]:
432:
433:
434: # This class define the Dynamic Programing agent
435: class DP_agent(object):
436:
437: def policy_eval(self, env, gamma, policy, threshold = 0.0001):
438:     """
439:     Policy Evaluation Step
440:     input:
441:         - env {Maze object} -- Maze to solve
442:         - gamma {np.array} -- Discount Factor
443:         - policy {float} -- Current Policy
444:         - threshold {float} -- Default 0.0001
445:     output:
446:         - V {np.array} -- Updated Value function
447:     """
448:     # Environment Description
449:     state_size = env.get_state_size()
450:     action_size = env.get_action_size()
451:     T = env.get_T()
452:     R = env.get_R()
453:     absorbing = env.get_absorbing()
454:
455:     # Initialisation
456:     V = np.zeros(env.get_state_size()) # Initialise value function to 0
457:     delta = threshold*2 # random initialization of delta > threshold
458:     V_eval = np.copy(V)
459:
460:     while (delta > threshold):
461:
462:         for state in range(state_size): # loop through every state in the maze
463:             if not absorbing[0, state]: # only evaluate policy for non-terminal /
(current) states
464:                 v = 0
465:
466:                 for action in range(action_size):
467:                     state_action = 0
468:                     for next_state in range(state_size):
469:                         state_action += T[state, next_state, action] * (R[state, /
next_state, action] + (gamma * V[next_state]))
470:                     v += (policy[state, action] * state_action)
471:
472:                 V_eval[state] = v
473:

```

Test Preview	coursework1.py: 9/14	Asia Belfiore - ab6124:a5	Test Preview	coursework1.py: 10/14	Asia Belfiore - ab6124:a5
	<pre>474:     delta = max(abs(V_eval - V)) 475:     V = np.copy(V_eval) 476: 477:     return V 478: 479: 480: def policy_iteration(self, env, gamma): 481:     """ 482:     Policy Improvement Step 483:     input: 484:         - env {Maze object} -- Maze to solve 485:         - gamma {np.array} -- Discount Factor 486:     output: 487:         - policy {np.array} -- Optimal Policy ze 488:         - V {np.array} -- Optimal State Values 489:     """ 490:     # Environment Description 491:     state_size = env.get_state_size() 492:     action_size = env.get_action_size() 493:     T = env.get_T() 494:     R = env.get_R() 495:     absorbing = env.get_absorbing() 496: 497:     # Policy and Value Initialisation 498:     policy = np.zeros((state_size, action_size)) 499:     policy[:, 1] = 1 # start action = 1 500:     # V = np.zeros(state_size) 501: 502:     policy_stable = False 503:     while not policy_stable: 504:         # 1) Policy Evaluation 505:         V = self.policy_eval(env, gamma, policy) 506:         # 2) Policy Improvement 507:         policy_stable = True 508:         for state in range(state_size): 509:             if not absorbing[0, state]: # only for non-terminal states 510:                 old_action = np.argmax(policy[state, :]) 511:                 state_optimal = np.zeros(action_size) 512: 513:                 for next_state in range(state_size): 514:                     state_optimal += T[state, next_state, :] * (R[state, next_state, :] + (gamma * V[next_state])) 515:                 new_state_policy = np.zeros(action_size) 516:                 new_state_policy[np.argmax(state_optimal)] = 1 # set value of best 517:                 # action to 1, others to 0 518:                 policy[state, :] = new_state_policy 519: 520:                 if old_action != np.argmax(policy[state, :]): 521:                     policy_stable = False 522: 523:         return policy, V 524: 525: 526: # Q1 [Action required] 527: # WARNING: make sure this function can be called by the auto-marking script 528: def solve(self, env): 529:     """ 530:     Solve a given Maze environment using Dynamic Programming 531:     input: env {Maze object} -- Maze to solve 532:     output: 533:         - policy {np.array} -- Optimal policy found to solve the given Maze environment 534:         - V {np.array} -- Corresponding Value function 535:     """ 536:</pre>			<pre>537:     """ 538:     # Add your code here 539:     # WARNING: for this agent only, you are allowed to access env.get_T(), env.get_R() and env.get_absorbing() 540:     """ 541:     gamma = env.get_gamma() # Discount Factor 542:     # gamma = 2.0 # Q1c) 543: 544:     policy, V = self.policy_iteration(env, gamma) 545: 546:     return policy, V 547: 548: 549:     # ## MC agent 550: 551:     # In[7]: 552: 553: 554:     # This class define the Monte-Carlo agent 555: 556:     class MC_agent(object): 557: 558:     def improve_greedy_policy(self, state_size, action_size, Q, t, epsilon = 0.4, decay_rate = 0.9995): 559:         """ 560:         Update the Optimal Policy following the epsilon-greedy approach 561:         input: 562:             - state_size {int} -- number of states in the maze 563:             - action_size {int} -- number of possible actions (4) per state 564:             - Q {np.array} -- Optimal policy found to solve the given Maze environment 565:             - t {int} -- current episode (for epsilon decay) 566:             - epsilon {float} -- parameter for (epsilon) ̵-greedy action 567:             (defaults to 0.4) 568:             - decay_rate {float} -- decaying rate of epsilon (defaults to 0.9995) 569:         output: 570:             - greedy_policy {np.array} -- Updated greedy policy 571:         """ 572:         greedy_policy = np.zeros((state_size, action_size)) 573: 574:         epsilon = epsilon * pow(decay_rate, t) 575: 576:         for state in range(state_size): # check optimal action for every state 577:             for action in range(action_size): 578:                 if action == np.argmax(Q[state, :]): # if current action is optimal 579:                     greedy_policy[state, action] = (1 - epsilon) + (epsilon / action_size) 580:                 else: 581:                     greedy_policy[state, action] = epsilon / action_size 582: 583:         return greedy_policy 584: 585:     def estimate_Q(self, env, Q, episode_steps, times_seen_state_action): 586:         """ 587:         Update Q(s,a) for each tuple (state, action) seen in an episode. 588:         input: 589:             - env {Maze object} -- Maze to solve 590:             - Q {np.array} -- Current Q Function 591:             - episode_steps {int} -- List of tuples (state,action,reward) representing 592:                 each (time) step of the episode 593:             - episode_steps {list} -- List of tuples (state,action,reward) representing 594:                 each (time) step of the episode 595:         output: 596:             - updated_Q {np.array} -- Updated Q function (off-policy) 597:             - times_seen_state_action {np.array} -- Track how many times each 598:</pre>	
git@gitlab.doc.ic.ac.uk:lab2425_autumn/Coursework1_ab6124.git	a6205	git@gitlab.doc.ic.ac.uk:lab2425_autumn/Coursework1_ab6124.git	a6205		

```

Test Preview      coursework1.py: 11/14      Asia Belfiore - ab6124:a5

(state,action)
598:                                     tuple has been seen throughout ✓
every episode
599:                                     generated so far
600:     - total_episode_return {float} -- Non-discounted Sum of Rewards collected ✓
so far
601:     """
602:     updated_Q = np.copy(Q)
603:     gamma = env.get_gamma()
604:     total_episode_return = 0 # keep running sum of all rÃ¢-collected rewards in ✓
episode
605:     visited_state_action_pairs = [] # keep track of seen (state,action) pairs
606:
607:     for t,(state, action, reward) in enumerate(episode_steps): # get ✓
state-action (and observed reward) at each time step
608:         total_episode_return += reward
609:
610:         if (state,action) not in visited_state_action_pairs:
611:             G = 0
612:             for i, (_,_,reward) in enumerate(episode_steps[t:]):
613:                 G += (reward * pow(gamma, i)) # discounted reward
614:
615:             times_seen_state_action[state,action] += 1
616:             # only account for new gain (G-Q(s,a))
617:             updated_Q[state, action] = Q[state, action] + ((G - Q[state, ✓
action])/times_seen_state_action[state,action])
618:             visited_state_action_pairs.append((state,action))
619:
620:     return updated_Q, times_seen_state_action, total_episode_return
621:
622:
623: def generate_episode(self, env, policy, action_size):
624:     """
625:     Generate an episode (trace) keeping track of the states visited,
626:     actions taken and associated rewards collected
627:     input:
628:         - env {Maze object} -- Maze to solve
629:         - policy {np.array} -- Current Policy
630:         - action_size {int} -- number of possible actions (4) for each state
631:     output:
632:         - episode_steps {list} -- List of tuples (state, action, next_reward) of ✓
visited states,
633:                                     actions taken and reward collected during each ✓
step of the episode (in order)
634:     """
635:     episode_steps = []
636:     t, state, _, done = env.reset() # start episode
637:
638:     while (not done) and (t<500):
639:         # randomly choose action from current state based on current (Îµp-soft) ✓
policy
640:         action = np.random.choice(range(action_size), p=policy[state, :])
641:         next_t, next_state, next_reward, done = env.step(action) # move in maze ✓
with chosen action
642:         episode_steps.append((state, action, next_reward)) # store (state, action, ✓
reward) tuples for each time step
643:         t, state = next_t, next_state
644:
645:     return episode_steps
646:
647:
648: # [Action required]
649: # WARNING: make sure this function can be called by the auto-marking script
650: def solve(self, env):
651:     """
652:     Solve a given Maze environment using Monte Carlo learning

```

```

Test Preview      coursework1.py: 12/14      Asia Belfiore - ab6124:a5

653:     input: env {Maze object} -- Maze to solve
654:     output:
655:         - policy {np.array} -- Optimal policy found to solve the given Maze ✓
environment
656:         - values {list of np.array} -- List of successive value functions for each ✓
episode
657:         - total_returns {list of float} -- Corresponding list of successive total ✓
non-discounted sum of reward for each episode
658:     """
659:     state_size = env.get_state_size()
660:     action_size = env.get_action_size()
661:
662:     # Initialisation (can be edited)
663:     Q = np.random.rand(state_size, action_size)
664:     policy = self.improve_greedy_policy(state_size, action_size, Q, t=0)
665:     total_returns = [] # keep track of reward changes
666:
667:     V = np.zeros(state_size)
668:     values = [V] # keep track of the state value changes
669:     times_seen_state_action = np.zeros((state_size, action_size))
670:
671:     #####
672:     # Add your code here
673:     # WARNING: this agent only has access to env.reset() and env.step() (and ✓
env.get_gamma())
674:     # You should not use env.get_T(), env.get_R() or env.get_absorbing() to ✓
compute any value
675:     #####
676:
677:     for t in range(1000): # for each episode
678:         # 1) generate an episode using current policy
679:         episode_steps = self.generate_episode(env, policy, action_size)
680:         # 2) update Q function based on the sequence of (state,action,reward) seen ✓
in the last episode
681:         Q, times_seen_state_action, sum_collected_rewards = self.estimate_Q(env, ✓
Q, episode_steps, times_seen_state_action)
682:         # 3) update (Îµp-greedy) policy
683:         policy = self.improve_greedy_policy(state_size, action_size, Q, t=t)
684:
685:         values.append(np.max(Q, axis=1)) # approximate V as the best state-action ✓
value
686:         total_returns.append(sum_collected_rewards)
687:
688:     return policy, values, total_returns
689:
690:
691: # ## TD agent
692:
693: # In[27]:
694:
695:
696: # This class define the Temporal-Difference agent
697:
698: class TD_agent(object):
699:
700:     def improve_greedy_policy(self, state_size, action_size, Q, t, epsilon = 0.2, ✓
decay_rate = 0.9995):
701:         """
702:         Update the Optimal Policy following the epsilon-greedy approach
703:         input:
704:             - state_size {int} -- number of states in the maze
705:             - action_size {int} -- number of possible actions (4)
706:             - Q {np.array} -- Optimal policy found to solve the given Maze environment
707:             - t {int} -- current time-step (for epsilon decay)
708:             - epsilon (Îµp) {float} -- parameter for (epsilon) Îµp-greedy action ✓
(defaults to 0.2)

```



```

Test Preview      coursework1.py: 13/14      Asia Belfiore - ab6124:a5

709:     - decay_rate {float} -- decaying rate of epsilon (defaults to 0.9995)
710:     output:
711:     - greedy_policy {np.array} -- Updated greedy policy
712:     """
713:     greedy_policy = np.zeros((state_size, action_size))
714:
715:     epsilon = epsilon * pow(decay_rate, t)
716:
717:     for state in range(state_size): # check optimal action for every state
718:         for action in range(action_size):
719:             if action == np.argmax(Q[state, :]): # if current action is optimal ✓
720:                 greedy_policy[state, action] = (1 - epsilon) + (epsilon / action_size)
721:             else:
722:                 greedy_policy[state, action] = epsilon / action_size
723:
724:     return greedy_policy
725:
726: def estimate_Q(self, env, Q, state, next_state, action, reward, alpha):
727:     """
728:     Update Q(s,a) for each (state, action) seen in an episode.
729:     input:
730:     - env {Maze object} -- Maze to solve
731:     - Q {np.array} -- Current Q Function
732:     - state {float} -- Current State
733:     - next_state {float} -- Next State reached from Current state with chosen ✓
734:     action
735:     - reward {float} -- Reward collected upon reaching Next State
736:     - apha {float} -- Agent learning rate
737:     output:
738:     - updated_Q {np.array} -- Updated Q function (off-policy)
739:     """
740:     gamma = env.get_gamma()
741:
742:     updated_Q = np.copy(Q)
743:     updated_Q[state,action] = Q[state,action] + (alpha * (reward + ((gamma * ✓
744:     Q[next_state, np.argmax(Q[next_state,:]))]) - Q[state, action]))
745:
746:     return updated_Q
747:
748: def q_learning(self, env, state_size, action_size, policy, Q, episode, alpha, ✓
749: epsilon = 0.2):
750:     """
751:     TD (Q-Learning) Process.
752:     input:
753:     - env {Maze object} -- Maze to solve
754:     - state_size {int} -- number of states in the maze
755:     - action_size {int} -- number of possible actions (4) per state
756:     - policy {np.array} -- Current policy
757:     - Q {np.array} -- Current Q Function
758:     - episode {int} -- current episode number
759:     - apha {float} -- Agent learning rate
760:     - epsilon {float} -- parameter for (epsilon) ̂p-greedy action ✓
761:     (defaults to 0.2)
762:     output:
763:     - total_episode_rewards {list} -- Non-discounted sum of collected reards ✓
764:     so far
765:     - Q {np.array} -- Updated Q function (off-policy)
766:     - policy {np.array} -- Updated target policy
767:     """
768:     total_episode_rewards = 0
769:
770:     policy = self.improve_greedy_policy(state_size, action_size, Q, episode, ✓
771: epsilon)

```

```

Test Preview      coursework1.py: 14/14      Asia Belfiore - ab6124:a5

768:     t, state, _, done = env.reset() # start episode
769:
770:     while (not done) and (t<500):
771:         # randomly choose action from current state based on current (̂p-soft) ✓
772:         policy
773:         action = np.random.choice(range(action_size), p=policy[state, :])
774:
775:         t, next_state, reward, done = env.step(action) # move in maze with chosen ✓
776:         action
777:         total_episode_rewards += reward
778:
779:         Q = self.estimate_Q(env, Q, state, next_state, action, reward, alpha)
780:         policy = self.improve_greedy_policy(state_size, action_size, Q, episode, ✓
781: epsilon)
782:         state = next_state
783:
784:     return total_episode_rewards, Q, policy
785:
786: # [Action required]
787: # WARNING: make sure this function can be called by the auto-marking script
788: def solve(self, env):
789:     """
790:     Solve a given Maze environment using Temporal Difference learning
791:     input: env {Maze object} -- Maze to solve
792:     output:
793:     - policy {np.array} -- Optimal policy found to solve the given Maze ✓
794:     environment
795:     - values {list of np.array} -- List of successive value functions for each ✓
796:     episode
797:     - total_rewards {list of float} -- Corresponding list of successive total ✓
798:     non-discounted sum of reward for each episode
799:     """
800:     state_size = env.get_state_size()
801:     action_size = env.get_action_size()
802:
803:     # Initialisation (can be edited)
804:     Q = np.random.rand(state_size, action_size)
805:     # Q = np.zeros((state_size, action_size))
806:     V = np.zeros(state_size)
807:     policy = self.improve_greedy_policy(state_size, action_size, Q, t=0)
808:     values = [V]
809:     total_rewards = []
810:
811:     #####
812:     # Add your code here
813:     # WARNING: this agent only has access to env.reset() and env.step()
814:     # You should not use env.get_T(), env.get_R() or env.get_absorbing() to ✓
815:     compute any value
816:     """
817:     alpha = 0.2
818:
819:     for episode in range(1000):
820:         total_episode_rewards, Q, policy = self.q_learning(env, state_size, ✓
821: action_size, policy, Q, episode, alpha=alpha)
822:         values.append(np.max(Q, axis=1))
823:         total_rewards.append(total_episode_rewards)
824:
825:     return policy, values, total_rewards

```



## Test Preview

testResults.txt: 1/1

Asia Belfiore - ab6124:a5

```
1: ----- Test Output -----
2:
3: -----
4: Automarking outputs:
5:
6: Question 0: Login and CID
7: Well done.
8:
9: Question 1: Dynamic Programming
10: Value compared to ground truth solution: DP RMSE: 5.969906226695359e-06
11: Well done.
12:
13: Question 2: MC learning
14: Well done.
15:
16: Question 3: TD learning
17: Well done.
18:
19: ----- Test Errors -----
20:
```