MineNavigation

Release 1.0

Simone Azeglio

Jun 19, 2019

CONTENTS:

1 maze.py 1

2 cli.py 3

3 heuristics.py 5

4 mission.py 7

5 algorithm.py 9

6 genetic.py 11

7 hillclimbing.py 13

8 Indices and tables 15

Python Module Index 17

Index 19

CHAPTER

ONE

MAZE.PY

Main file - it runs the mission

Basically, after loading the world (*.xml* file) it starts with two concatenated while statements (formally the mission is a concatenated while statement). The second one represents each run. In the end it calculates the score and it saves a *.csv* file (log file) by setting the selected algorithm's score. We used *time.sleep(2)* in order to avoid issues related to items counting (stochastics fluctuations in time are significant and most of the times the set_score function wouldn't work because of that, compromising the learning process)

CHAPTER

TWO

CLI.PY

A file used for everything related to world, algorithms and files selection from terminal:

cli.**valid_algorithms**()
: Shows valid algorithms (*genetic* and *hillclimbing* so far) and description

cli.**get_algorithm**(*alg_selected*)
: Returns a strategy based on the selected algorithm (alg_selected has to be one of the valid algorithms)

cli.**build_maze_filepath**(*maze*)
: It gets the filepath of the choosen *.xml* file

cli.**algorithms_list**()
: Create list of algorithms for *parse_args()* method

cli.**parse_args**()
: Returns a triad : world (maze), algorithm and output file. In this way it specifies the defined configuration. The output file is used in order to let the agent build a memory and to plot the fitness function

CHAPTER

THREE

HEURISTICS.PY

A file used for useful function:

heuristics.**distance**(*lhs*, *rhs*)
: Calculate the distance between left-hand-side (*lhs*) and right-hand-side (*rhs*). E.g : lhs and rhs could be the agent and the diamond

heuristics.**location**(*entity*)
: Define location as *x,y,z* triad

heuristics.**dot**(*lhs*, *rhs*)
: Dot product between left-hand-side (*lhs*) and right-hand-side (*rhs*)

heuristics.**diff**(*lhs*, *rhs*)
: Coordinate difference between left-hand-side (*lhs*) and right-hand-side (*rhs*)

heuristics.**magnitude**(*vec*)
: Magnitude of a vector (*vec*)

heuristics.**normalize**(*vec*)
: Normalize a vector (*vec*)

heuristics.**get_player_location**(*el*)
: Returns Agent's location

heuristics.**get_closest_entity**(*el*, *entity_name*)
: Returns the closest entity and the coordinate difference between agent and closest entity

heuristics.**closest_cardinals**(*dir*, *obs*)
: Returns closest cardinal, in order to plan the path (dir = direction, obs = observation)

heuristics.**opposite_direction**(*dir*)
: Returns opposite direction given a direction (dir = direction)

heuristics.**random_direction**(*obs*)
: Returns a random direction in order to implement a random strategy (useful in order to don't get stuck) (obs = observation)

heuristics.**towards_item**(*obs*)
: Returns the direction to follow in order to get to the item (obs = observation)

CHAPTER

FOUR

MISSION.PY

class mission.**mission**

A class used for everything related to a mission (a run of the environment):

load(*mission_file*)

Load the world from *.xml* file and create default **Malmo** objects

start()

Set client pool, client info , reset the world for each new mission, set dimensions of the video-window and agent's viewpoint. Attempt to start a mission for 10 times if there's any issue occurring (some stochastics errors - take a look at Malmo's official documentation: https://github.com/microsoft/malmo), when the mission starts it keeps counting the time.

is_running()

Check if the mission is currently running: returns True if mission is running

get_observation()

Loads floor grid, edge distances, current player position, and entity (e.g. diamond) position in order to let the agent know the distance to the diamond (odor-like representation). It also counts collected items and exports the world view in a *.json* file. Agent's observation is a *3x3 grid*: he's in the center and he knows only 1 block in every possible direction

send_command()

Send command input to the Agent in order to perform the next move. It even counts if the current cell is a newly explored one or not (this will be useful in implementing the score function)

stop_clock()

It stops the time at the end of the mission (useful for score function based on time)

check_errors()

Check whether there are errors in the mission (check *world_state* Malmo's method). Those errors are typically related to Malmo Client

block_score()

The agent has to explore new blocks in order to get a better *fitness*, he gets a penalty by being in the same block (no points for fitness)

time_score()

The agent gets one fitness point for each seconds he spends alive

item_score()

The agent gets some more points for each item he picks up from the ground (1 diamond = 50 fitness points)

score()

Sums up block_score , time_score and item_score to get the Fitness

class mission.**observation**(*set_grid*, *set_edge_distances*, *set_cell*, *set_entity_locations*)

A class used to represent an observation of the world:

set_grid

Extracted from the *.json* World File by *mission.get_observation()* The grid is the part of the world seen by the agent

Type str

set_edge_distances

Extracted from the *.json* World File by *mission.get_observation()* Distances from the edges of the world (The world is limited)

Type str

set_cell

Extracted from the *.json* World File by *mission.get_observation()* Cell is the agent location in the world (The agent has a GPS)

Type str

set_entity_locations

Extracted from the .json World File by mission.get_observation() Locations of entities (e.g. Diamonds, Zombies)

Type str

at_junction()

States whether a move is plausible or not : The agent can only walk over glowstone. This is useful in order to limit the world with another material (e.g. Lava)

CHAPTER

FIVE

ALGORITHM.PY

class algorithms.algorithm.**algorithm**(*set_actions*)

A class used to define an algorithm and its actions:

Attributes

process_score(*score*)

@*abc.abstractmethod* After specifying the algorithm, while running the program, it processes the score by following the current algorithm rule (*genetic, hillclimb*)

set_score(*score*)

When each run of a mission is ended it sets the score and it saves the score in a *.csv* file (useful for plotting the fitness function)

get_action(*obs*)

@*abc.abstractmethod* It gets the action that the agent has to perform from the specific algorithm (*genetic, hillclimb*)

CHAPTER

SIX

GENETIC.PY

Pseudocode for the creation of a new population:

fittest = four top scoring strings within the population

for i in range(population_size):

parent1 = random.choice(fittest)

parent2 = random.choice(fittest)

offspring = parent1[crossover:] + parent2[:crossover]

for heuristic in offspring:

5% chance to mutate heuristic to another one

population.append(offspring)

class algorithms.genetic.**genetic**(*set_actions*, *set_gen_size=8*, *set_str_len=5*, *set_sel_frac=0.5*, *set_mut_prob=0.05*)

A class used to define the genetic algorithm:

set_actions(*str*)
list of possible actions the agent can take

set_gen_size(*int*)
Size of the generation, each generation is a list of strategies (t = towards the entity, the diamond ; r = random move). Each generation has 8 list strategies in this model

set_str_len(*int*)
Length of the list, it starts with an average (gaussian distribution) of length 5

set_sel_frac(*double*)
Sets the selected fraction of the most 4 top high-scoring strings in order to generate strings in the next iteration

set_mut_prob(*double*)
Mutation probability of the genetic algorithm (p = 0.05)

next_generation()
Creates next generation of strategies. It finds the best scores in the population, it selects the top 4 high-scoring strings in order to generate the next generation of strings.

process_score(*score*)
Keeps track of how the score changes and improves at each iteration

get_action(*obs*)
Returns an action based on observations and on a policy which considers the previous methods

CHAPTER

SEVEN

HILLCLIMBING.PY

After scoring the first string, the algorithm runs a mission for each string adjacent to the first string in the search space. An adjacent string is a string which differs by only one addition of a heuristic from the string, removal of a heuristic, or change of a heuristic. After scoring every adjacent string, the algorithm chooses the string with the best score. It then explores the adjacent strings to that string, choosing the best one of those, and so on. These incremental improvements allow the algorithm to find heuristic strings that produce higher and higher scores.

Added *Simulated Annealing* feature! The purpose of this probabilistic behavior is to maximize the space that the hill-climbing algorithm explores. Rather than sticking with whatever seems locally optimal, the hill-climbing algorithm may find even better strings in areas of the search space that, at first glance, seemed sub-optimal.

Pseudocode for hillclimbing:

while True:

for string adjacent to current_string:

if score(string) > score(current_string): *best_string = string*

current_string = best_string

Pseudocode for Simulated Annealing:

while True: *prob = probability that we choose a suboptimal choice*

eps = random.random()

cooling_rate = 0.5

neighbors = every string adjacent to current_string

if eps < p: *string = random.choice(neighbors)*

else:

for string adjacent to current_string:

if score(string) > score(current_string): *best_string = string*

current_string = best_string

*prob *= cooling_rate*

class algorithms.hillclimbing.**climber**(*set_actions*, *init_eps=0.0*, *set_cooling=1.0*)

A class used to define the hillclimbing algorithm:

set_actions(*str*)

List of possible actions the agent can take

init_eps (*double*)
Minimum reduction in the function before termination (Simulated Annealing - https://www.aero.iitb.ac.in/~rkpant/webpages/DefaultWebApp/salect.pdf)

set_cooling (*double*)
Cooling rate (Simulated Annealing - https://www.aero.iitb.ac.in/~rkpant/webpages/DefaultWebApp/salect.pdf)

generate_local_space ()
Starting from a string it performs the three possible operations on heuristics creating a local search space. After that it starts the Simulated Annealing algorithm

pick_next_string ()
It selects the next string, starting from the best score string and looking around that one in the search space

process_score (*score*)
Keeps track of how the score changes for ending in the same block. It returns a combination of score and current string heuristics

get_action (*obs*)
Returns action based on heuristics string (which is based on observations)

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

algorithms.algorithm, 8
algorithms.genetic, 9
algorithms.hillclimbing, 12

c

cli, 1

h

heuristics, 3

m

maze, **??**
mission, 5

INDEX

A

algorithm (*class in algorithms.algorithm*), 9
algorithms.algorithm (*module*), 8
algorithms.genetic (*module*), 9
algorithms.hillclimbing (*module*), 12
algorithms_list() (*in module cli*), 3
at_junction() (*mission.observation method*), 8

B

block_score() (*mission.mission method*), 7
build_maze_filepath() (*in module cli*), 3

C

check_errors() (*mission.mission method*), 7
cli (*module*), 1
climber (*class in algorithms.hillclimbing*), 13
closest_cardinals() (*in module heuristics*), 5

D

diff() (*in module heuristics*), 5
distance() (*in module heuristics*), 5
dot() (*in module heuristics*), 5

G

generate_local_space() (*algorithms.hillclimbing.climber method*), 14
genetic (*class in algorithms.genetic*), 11
get_action() (*algorithms.algorithm.algorithm method*), 9
get_action() (*algorithms.genetic.genetic method*), 12
get_action() (*algorithms.hillclimbing.climber method*), 14
get_algorithm() (*in module cli*), 3
get_closest_entity() (*in module heuristics*), 5
get_observation() (*mission.mission method*), 7
get_player_location() (*in module heuristics*), 5

H

heuristics (*module*), 3

I

init_eps (*algorithms.hillclimbing.climber attribute*), 13
is_running() (*mission.mission method*), 7
item_score() (*mission.mission method*), 7

L

load() (*mission.mission method*), 7
location() (*in module heuristics*), 5

M

magnitude() (*in module heuristics*), 5
maze (*module*), 1
mission (*class in mission*), 7
mission (*module*), 5

N

next_generation() (*algorithms.genetic.genetic method*), 12
normalize() (*in module heuristics*), 5

O

observation (*class in mission*), 7
opposite_direction() (*in module heuristics*), 5

P

parse_args() (*in module cli*), 3
pick_next_string() (*algorithms.hillclimbing.climber method*), 14
process_score() (*algorithms.algorithm.algorithm method*), 9
process_score() (*algorithms.genetic.genetic method*), 12
process_score() (*algorithms.hillclimbing.climber method*), 14

R

random_direction() (*in module heuristics*), 5

S

score() (*mission.mission method*), 7

send_command() (*mission.mission method*), 7
set_actions (*algorithms.genetic.genetic attribute*), 11
set_actions (*algorithms.hillclimbing.climber attribute*), 13
set_cell (*mission.observation attribute*), 8
set_cooling (*algorithms.hillclimbing.climber attribute*), 14
set_edge_distances (*mission.observation attribute*), 8
set_entity_locations (*mission.observation attribute*), 8
set_gen_size (*algorithms.genetic.genetic attribute*), 11
set_grid (*mission.observation attribute*), 8
set_mut_prob (*algorithms.genetic.genetic attribute*), 11
set_score() (*algorithms.algorithm.algorithm method*), 9
set_sel_frac (*algorithms.genetic.genetic attribute*), 11
set_str_len (*algorithms.genetic.genetic attribute*), 11
start() (*mission.mission method*), 7
stop_clock() (*mission.mission method*), 7

T

time_score() (*mission.mission method*), 7
towards_item() (*in module heuristics*), 5

V

valid_algorithms() (*in module cli*), 3