Welcome to Py-EcoSystem's documentation!

Main program in here.

app.main.**create_sample_fence**()
Creates random maze (labyrinth/fence).
app.main.**create_start_menu**()
Creates and opens start menu.
app.main.**main**()
Main program.
app.main.**start_simulation**()
Performs simulation.
app.main.**terminate_animal_threads**()
Terminate and join all animal threads.

Classes, functions and import mutual for several .py project files.

class app.common.**Animals**(*value*)
An enumeration.
class app.common.**Colour**(*value*)
An enumeration.
class app.common.**Directions**(*value*)
An enumeration.
app.common.**check_terrain_boundaries**(*x*, *y*)
Checks whether we are within terrain (map) boundaries.
app.common.**set_stats**(*animal_species*, *value*)
Safe multi-threading memory sharing.
app.common.**set_terrain_value**(*x*, *y*, *value*)
Safe multi-threading memory sharing.

Fence (maze) generation engine and animal movement limitations due to fence placement.

app.fence.**add_vertex**(*node_idx_1*, *node_idx_2*)
Adds node_idx_2 to node_idx_1's fence-graph neighbours
app.fence.**build_vertex**(*start_node*, *end_node*)
Builds a fence wall, first checking whether: - given nodes are placed in proper proximity (they have to be direct neighbours for the vertex to exist), - there's no redundancy (end_node will be placed in start_node's neighbourhood set and start_node will not be placed in end_node's neighbourhood set, particular node will be added to the neighbourhood set just once)
app.fence.**can_make_move**(*current_row*, *current_column*, *delta_x*, *delta_y*)
Decides whether an animal can make a certain move or not, considering fence placement and map boundaries.

Parameters:
- **current_row** – current x position of an animal
- **current_column** – current y position of an animal
- **delta_x** – x coordinate of animal's movement vector
- **delta_y** – y coordinate of animal's movement vector

Returns: boolean

app.fence.**check_if_wall_exists**(*node_idx_1*, *node_idx_2*)
Checks whether a vertex exists between given two nodes.
app.fence.**delete_all_walls**()

Fence clean-up.

app.fence.**delete_vertex**(*node_idx_1, node_idx_2*)

Deletes node_idx_2 from node_idx_1's fence-graph neighbours

app.fence.**delete_wall**(*start_node, end_node*)

Deletes vertex between the two given nodes, checks whether they are direct neighbours and the vertex exists.

app.fence.**dfs_build**()

Generuje n elementów labiryntu (płotu): 1. Wyznacza maksymalną głębokość przeszukiwania w DFS (maksymalna liczba postawionych ścian płotu w jednym cyklu wywołań rekurencyjnych DFS) - na podstawie wielkości mapy terenu (N). 2. Wyznacza losowe początki n poszczególnych elementów ('wysp') labiryntu, czyli źródła n pierwotnych wywołań DFS. 3. Wywołuje n razy procedurę dfs_visit, tworząc ściany n 'wysp' labiryntu. 4. Wyznacza po 2 losowe punkty w obrębie każdej 'wyspy' (każdego drzewa przeszukiwania wgłąb), pomiędzy którymi tworzone będzie 'rozwiązanie' labiryntu (patrz procedura get_maze_solution).

app.fence.**dfs_visit**(*current_node, wall_no, walls_already_built, maze_element_id*)

Buduje jedną pełną 'wyspę' (element) labiryntu. Generowane są zbiory łamanych otwartych, których boki mogą być względem siebie jedynie prostopadłe lub równoległe (wybór losowy). Zbiór łamanych reprezentuje ściany labiryntu, przestrzenie wewnątrz łamanych reprezentują korytarze labiryntu.

Działanie: Kolorujemy odwiedzany wierzchołek na szaro. Sprawdzamy, czy nie przekroczyliśmy limitu ścian stawianych w jednej serii wywołań rekurencyjnych. Losujemy kierunek, w którym postawimy ścianę (góra/prawo/dół/lewo). Sprawdzamy warunki postawienia ściany w tym kierunku - standardowe warunki DFS z jedną modyfikacją: za wierzchołek czarny (przetworzony) uznajemy taki, z którego wychodzą 3 ściany. Gwarantuje to otwarty charakter łamanych. Zatem wierzchołki, z których wychodzą 1 lub 2 ściany są szare, wierzchołki, z których wychodzą 3 ściany są czarne, a wierzchołki poza labiryntem są białe.

W wyniku działania tej procedury powstaje zbiór korytarzy labiryntu, bardziej lub mniej 'rozgałęzionych'. Są to jednak łamane otwarte, zatem do każdej z nich da się wejść z zewnątrz (spoza labiryntu), ale nie istnieją przejścia pomiędzy nimi (inne niż wyjście z korytarzy w obrębie danej łamanej poza labirynt i wejście do sąsiadującej łamanej). Przejścia pomiędzy łamanymi tworzone są w procedurze get_maze_solution.

app.fence.**fence_border**(*node_idx*)

Returns directions in which the given node is adjacent to the map's border, if they exist.

app.fence.**get_fence_node_dirs**(*node_idx*)

Returns x and y coordinates of the given node.

app.fence.**get_fence_node_idx**(*x, y*)

Returns the index of the node given by its' x and y coordinates.

app.fence.**get_first_common_parent**(*node_idx_1, node_idx_2, start_node_idx*)

Takes two nodes and processes their DFS tree paths (parents[]). Searches for the first node that is mutual for those two paths (any node, including the DFS source, if the two nodes are on different DFS tree branches).

Parameters:
- **node_idx_1** – First node.
- **node_idx_2** – Second node.
- **start_node_idx** – DFS source node. Along with the 'parents' array, it constitutes the DFS tree.

Returns: A dictionary of {node with a shorter 'DFS parent' path to the DFS source; first mutual node; length of the shorter 'DFS parent' path}

app.fence.**get_joined_nodes_path**(*node_with_shorter_path, first_common_node_idx, node_with_longer_path, length*)

Returns a path between any two given points within the same DFS tree (the same maze 'element'). Merges two paths: - the longer one (non-reversed), starting in node_with_longer_path and passing

through its' following DFS-parent nodes (parents[]), - the shorter one (reversed), starting in node_with_shorter_path, which is a reverse of the DFS-parent-nodes path (children[]). Returns a concatenation of these two paths, hence a path between: node_with_longer_path and node_with_shorter_path.

Parameters:
- **node_with_shorter_path** – Beginning node of the path to reverse.
- **first_common_node_idx** – The first node that is mutual for both paths. It can be any node, including the DFS

source node. :param node_with_longer_path: Beginning node of the the non-reversed path. :param length: The length of the path to be created. :return: An array in which the i-th element represents the i-th node on the path.

app.fence.**get_maze_solution**(*start_node, end_node, start_node_idx*)

Wyznacza 'rozwiązanie' labiryntu, tj. przejście pomiędzy zadanymi punktami (wierzchołkami).

Parameters:
- **start_node** – Start of the labirynth path.
- **end_node** – End of the labirynth path.
- **start_node_idx** – DFS source for technical purposes.

Założenia: Oba punkty są wierzchołkami tego samego drzewa przeszukiwania wgłąb, choć mogą (a nawet powinny, dla efektów wizualnych) leżeć na różnych jego gałęziach (po różnych 'stronach' wierzchołka-źródła DFS).

Działanie: Korzystając z gotowego już drzewa przeszukiwania wgłąb (parents[]), wyznaczać będziemy ścieżkę pomiędzy zadanymi wierchołkami.

1. Rozpatrujemy ścieżki od danych wierzchołków do wierzchołka-źródła DFS i znajdujemy pierwszy (idąc od liści) wierzchołek wspólny dla obu tych ścieżek (może być nim wierzchołek-źródło DFS, jeśli zadane wierzchołki znajdują się na innych gałęziach głównych drzewa). Wierzchołek ten wyznacza ścieżkę pomiędzy zadanymi wierzchołkami - należy wziąć ścieżkę od jednego zadanego wierzchołka do wierzchołka wspólnego i połączyć ją ze ścieżką od drugiego zadanego wierzchołka do wierchołka wspólnego, ale 'odwróconą' (zmiana kierunku krawędzi na przeciwny). Odwracamy krótszą z ww. ścieżek, następnie ścieżki 'łączymy'. W otrzymanej końcowej ścieżce wierzchołkiem początkowym jest ten, który jest bardziej oddalony od punktu wspólnego, a końcowym ten, który znajduje się bliżej punktu wspólnego.

2. W wygenerowanym w procedurze dfs_visit zbiorze ścian kluczowe są wierchołki czarne. Każdy wierchołek czarny to styk 3 ścian, co oznacza, że wyznacza on 2 lub 3 sąsiadujące ze sobą łamane otwarte (korytarze). Interesować nas będzie ciąg czarnych wierchołków obecnych na otrzymanej w pkt. 1. ścieżce. Każde 2 sąsiednie czarne wierchołki w tym ciągu, które jednocześnie nie sąsiadują ze sobą na ścieżce, wyznaczają korytarz, którym da się przejść (wzdłuż, zawartych pomiędzy, wierzchołków szarych). 'Przejście' przez labirynt polegać będzie na przejściu 'korytarzami' ww. ścieżki, usuwając po drodze pewne krawędzie. Najpierw wybieramy wierzchołek początkowy (pierwszy czarny) oraz umownie 'stronę' ściany labiryntu, wzdłuż której będziemy się poruszać (umownie lewa/prawa). Następnie, dopóki nie przejdziemy całej ścieżki:

a. wybieramy następny czarny wierzchołek

b) oceniamy, czy da się z niego przejść wzdłuż 'naszej' strony ścian do następnego wierzchołka na ścieżce - poprzez ocenę naszego położenia względem krawędzi sąsiadów wierzchołka z pkt. a). Wychodzą z niego dwie krawędzie inne niż ta, z której przyszliśmy, w tym jedna jest następną krawędzią naszej ścieżki. Jeśli jest to krawędź 'lewa', a znajdujemy się po stronie 'lewej', to możemy bez problemu poruszać się dalej po tej samej stronie. Jeśli jednak jest to krawędź 'prawa', a znajdujemy się po stronie lewej, usuwamy poprzednią krawędź (tę, z której przyszliśmy) i zmieniamy stronę na przeciwną, co gwarantuje nam dalsze podążanie ścieżką. W proc. dfs_visit powstał zbiór łamanych otwartych, czyli w zasadzie 'pseudolabirynt', zawsze można więc przejść do

sąsiedniego korytarza cofając się do wyjścia i wchodząc wejściem do sąsiedniego, zatem teoretycznie nie byłoby potrzeby usuwania krawędzi. Byłoby to jednak wybitnie nielabiryntowe, dlatego stosujemy przejście ścieżką z pkt. 1. z usuwaniem krawędzi w zależności od 'zakrętów' drzewa przeszukiwania wgłąb.

app.fence.**get_move_direction**(*delta_x, delta_y*)

Returns a movement direction given by the x and y vectors.

app.fence.**get_next_black_node**(*current_node, previous_node, nodes_path, i*)

Returns a pair of the next black (DFS tree colouring) and its' predecessor on a given node path.

Parameters:
- **current_node** – The current node. We're searching for its' next black neighbour on the given path.
- **previous_node** – Predecessor of the current node. Needed for the purposes of the get_maze_solution function.
- **nodes_path** – The graph path to follow.
- **i** – Position on the graph path to follow.

Returns: Pair of (predecessor, next black node).

app.fence.**get_node_colour**(*node_idx*)

Test for DFS-based function.

app.fence.**get_node_neighbour**(*neighbour_direction, row, column*)

Returns a neighbour of the node given by its' coordinates (row and column) and the direction to follow, if exists.

app.fence.**get_opposite_wall_side**(*current_side*)

If 'left' return 'right', if 'right' return 'left'

app.fence.**get_path_twist_direction**(*current_black_node, nodes_path, i*)

Wyznacza kierunek zakrętu 'czyhającego' za najbliższym czarnym wierzchołkiem na przemierzanej ścieżce nodes_path.

Parameters:
- **current_black_node** – Rozpatrywany czarny wierzchołek.
- **nodes_path** – Rozpatrywana ścieżka, którą podążamy.
- **i** – Wskaźnik na miejsce na ścieżce, w którym aktualnie się znajdujemy (nr indeksu current_black_node w nodes_path).

app.fence.**get_random_corner_fence_location**(*randint_1, randint_2*)

Returns a randomised fence-generating starting (x, y) position.

app.fence.**get_surrounding_nodes**(*direction, x_coord, y_coord*)

Returns all the nodes that border node given by args: x_coord, y_coord, provided that those nodes exist.

app.fence.**neighbours_relations**(*node, neighbour*)

Returns direction in which arg node borders with arg neighbour, if exists.

app.fence.**paint_fence_white**()

DFS tree maintenance.

app.fence.**reset_fence**()

Fence clean-up.

app.fence.**reset_node_colours**()

Fence clean-up.

app.fence.**reset_parents_and_children**()

Fence clean-up.

app.fence.**reverse_path**(*node_beginning, node_end*)

Takes two nodes on one DFS tree branch, processes the DFS tree path between them and reverses this path.

Animal maintenance - movement, increasing/decreasing energy, procreation, checking for overpopulation eating other species.

class app.animals.**Animal**(*x*, *y*, *identity*)

class app.animals.**Rabbit**(*x*, *y*)

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class app.animals.**Wolf**(*x*, *y*)

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

GUI homemade elements: sliders, buttons, text lines.

Program visualisation. Starting menu, general settings menu, map settings menu, simulation window.

Indices and tables

- Index
- Module Index
- Search Page