Τεχνητή Νοημοσύνη: Εργαστηριακή Άσκηση 1

Ο στόχος της εργασίας είναι η εφαρμογή αλγορίθμμων αναζήτησης και εύρεσης καλύτερου μονοπατιού σε ένα τετραγωνικό grid διαστάσεων $N \times N$

Αρχικά, ο στόχος μας είναι η κατασκευή προβλημάτων αναζήτησης σε τετραγωνικές πίστες, όπου υπάρχει τουλάχιστον ένα μονοπάτι για τη μετάβαση από ένα αρχικό σημείο σε ένα στόχο.

Εκφώνηση

Στην παρούσα εργασία σας ζητείται να μελετήσετε και υλοποιήσετε τα παρακάτω:

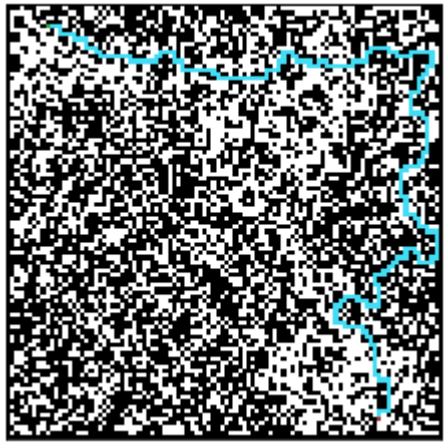
Μέρος 1

Στο πρώτο μέρος της εργασίας καλείστε να κατασκευάσετε τους τετραγωνικούς χάρτες με τους οποίους θα εργαστείτε. Ο χάρτης πρακτικά θα είναι ένας πίνακας μεγέθους $N \times N$ όπου σε κάθε θέση του θα υπάρχει 1 ή 0, όπου το 1 σημαίνει ότι στην αντίστοιχη θέση υπάρχει εμπόδιο, ενώ το 0 σημαίνει ότι είναι ελεύθερη. Τα σύνορα του χάρτη θα περιέχουν 1, δηλαδή ο χάρτης θα είναι οριοθετημένος. Επίσης για κάθε χάρτη θα γνωρίζουμε την αρχή (start) και τον στόχο (goal). Η κατανομή των εμποδίων στον χώρο θα γίνεται τυχαία με μια δοσμένη πιθανότητα p ενώ η πυκνότητα εμποδίων θα μπορεί να προσαρμόζεται από τον χρήστη. Αυτό πρακτικά σημαίνει ότι μπορούμε να κατασκευάσουμε πίστες με διαφορετικές πυκνότητες εμποδίων με σκοπό να ελέγξουμε την συμπεριφορά των αλγόριθμών με βάση αυτή την αλλαγή. Παρόλα αυτά κάθε πίστα πρέπει να περιέχει τουλάχιστον ένα μονοπάτι από τον κόμβο εκκίνησης προς τον κόμβο στόχο. Η βασική δομή της κλάσης που υλοποιεί τα παραπάνω είναι δοσμένη. Σε αυτήν περιέχονται οι συναρτήσεις adjacent(node) η όποια επιστρέφει μια λίστα με τους γείτονες κάθε κόμβου, draw_map(Start, Goal) η όποια δέχεται έναν κόμβο εκκίνησης και έναν κόμβο στόχο και εμφανίζει τον χάρτη που έχει κατασκευαστεί ζωγραφίζοντας παράλληλα με διαφορετικό χρώμα τις παραπάνω εισόδους, και τέλος έχει υλοποιηθεί και ένα μέρος την συνάρτησης init. Την συνάρτηση αυτή καλείστε να συμπληρώσετε τον αλγόριθμό που θα εγγυάται την ύπαρξη τουλάχιστον ενός μονοπατιού από το start στο goal, έτσι ώστε να έχει νόημα η αναζήτηση του βέλτιστου μονοπατιού σε αυτόν. Τονίζεται ότι δεν πρέπει το μονοπάτι που υπάρχει εγγυημένα να είναι το βέλτιστο, απλά ένα τυχαίο μονοπάτι από το σημείο εκκίνησης στο σημείο τερματισμού.

Παρακάτω φαίνονται ενδεικτικά χάρτες με μεγέθη $N = 10, 25, 50$.

Μέρος 2

Στο δεύτερο μέρος της εργασίας καλείστε να κατασκευάσετε διάφορους αλγόριθμους εύρεσης συντομότερων μονοπατιών μεταξύ δυο κόμβων για τους παραπάνω χάρτες. Παράλληλα καλείστε να κατασκευάσετε και διάφορες συναρτήσεις τόσο για την μέτρηση του πραγματικού κόστους όσο και για την εκτίμηση των αποστάσεων από έναν κόμβο στον κόμβο στόχο (heuristic). Επίσης μπορείτε να πειραματιστείτε με οποιαδήποτε από τις παραμέτρους κάθε αλγορίθμου με σκοπό να μελετήσετε το πώς οι αλλαγές αυτές επηρεάζουν την πολυπλοκότητα των αλγορίθμων καθώς και το βέλτιστο μονοπάτι.
Ένα παράδειγμα βέλτιστου μονοπατιού σε έναν χάρτη με χαρακτηριστικά $(N = 100, p = 0.6)$ φαίνεται παρακάτω



Η γενική μορφή της συνάρτησης κόστους στον αλγόριθμο A* είναι:

$$f(n) = g(n) + h(n)$$

Παραπάνω, η συνάρτηση $g(n)$ δίνει την πραγματική απόσταση από το σημείο εκκίνησης μέχρι τον κόμβο n, και η συνάρτηση $h(n)$ αποτελεί μια ευριστική της απόστασης από τον κόμβο n μέχρι τον στόχο. Σας ζητείται να πειραματισθείτε με τις εξής επιλογές για τις δύο συναρτήσεις:

- $g(n) = 0$ και $h(n) = \{\text{manhattan}(n), \text{euclidean}(n)\}$. Ποιος αλγόριθμος αναζήτησης προκύπτει; Μπορεί να βρει πάντα το βέλτιστο μονοπάτι;
- $g(n) = 1$ και $h(n) = 0$. Ποιος αλγόριθμος αναζήτησης προκύπτει; Μπορεί να βρει πάντα το βέλτιστο μονοπάτι;
- $g(n) = 1$ και $h(n) = \{\text{manhattan}(n), \text{euclidean}(n)\}$. Ποιος αλγόριθμος αναζήτησης προκύπτει; Μπορεί να βρει πάντα το βέλτιστο μονοπάτι;

Μπορείτε να προτείνετε και άλλες ευριστικές συναρτήσεις εκτός από τις αποστάσεις manhattan και euclidean;

Μέρος 3

Στο τρίτο και τελευταίο μέρος καλείστε να υλοποιήσετε μια συγκριτική μελέτη των αλγορίθμων που κατασκευάσατε στο Μέρος 2 με σκοπό να καταλήξετε σε ορισμένα συμπεράσματα. Οι αλγόριθμοι θα συγκρίνονται με βάση διάφορα χαρακτηριστικά, όπως την πολυπλοκότητά τους και το κατά πόσο μπορούν να βρούν το βέλτιστο μονοπάτι.

Για να γίνει αυτή η πειραματική μελέτη, παραμετροποιούμε τις εισόδους των αλγορίθμων ως εξής:

- Για να κρίνουμε τον τρόπο με τον οποίο το μέγεθος του χάρτη επηρεάζει τους αλγορίθμους, κατασκευάζουμε χάρτες με σταθερή πιθανότητα ύπαρξης εμποδίου $p = \frac{1}{2}$ και διάσταση $N = [10, 20, \ldots, 100]$. Για κάθε παραμετροποίηση προτείνεται να κατασκευάζετε 100 διαφορετικούς χάρτες, στους οποίους θα εκτελείτε τους παραπάνω αλγορίθμους.
- Για να κρίνουμε τον τρόπο με τον οποίο η πυκνότητα εμποδίων επηρεάζει τους αλγορίθμους, κατασκευάζουμε χάρτες με σταθερό μέγεθος $N = 50$ και πιθανότητα $p = [0, 0.1, \ldots, 1]$. Για κάθε παραμετροποίηση προτείνεται να κατασκευάζετε 100 διαφορετικούς χάρτες, στους οποίους θα εκτελείτε τους παραπάνω αλγορίθμους.

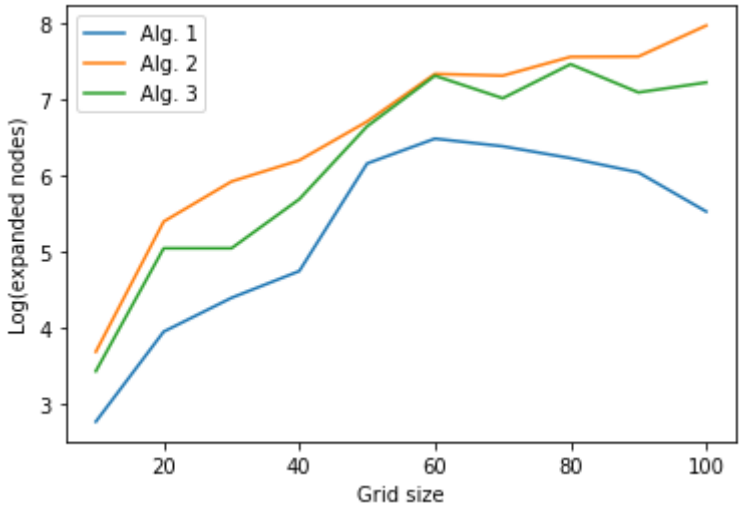
  Για καθεμία από τις παραπάνω παραμετροποιήσεις, ζητείται να συλλέξετε τα εξής χαρακτηριστικά:

- το μήκος του ελάχιστου μονοπατιού που εξάγει ο εκάστοτε αλγόριθμος.
- το πλήθος των επεκτεταμένων κόμβων (expanded nodes) του κάθε αλγορίθμου, που αποτελεί μέτρο της πολυπλοκότητάς του.

  Για να σας είναι εύκολο να εξάγετε συμπεράσματα από τις παραπάνω παραμετροποιήσεις, ζητείται να κατασκευάσετε 4 γραφικές παραστάσεις:

- το μήκος του ελάχιστου μονοπατιού συναρτήσει του μεγέθους του χάρτη
- το πλήθος των expanded nodes συναρτήσει του μεγέθους του χάρτη
- το μήκος του ελάχιστου μονοπατιού συναρτήσει της πιθανότητας p
- το πλήθος των expanded nodes συναρτήσει της πιθανότητας p

  Ένα παράδειγμα γραφικής παράστασης είναι:



Σας ζητείται σχολιάσετε τις παραπάνω γραφικές παραστάσεις, και συγκεκριμένα το πώς μεταβάλλεται η συμπεριφορά των αλγορίθμων συναρτήσει των N και p.

Σημειώνουμε τα παρακάτω:

- Σε όλα τα παραπάνω πειράματα μπορείτε να λάβετε σαν σημείο εκκίνησης το $(1, 1)$ και σαν σημείο τερματισμού $(N - 2, N - 2)$. Εναλλακτικά, μπορείτε να πειραματισθείτε με τυχαία σημεία εκκινησης και τερματισμού.
- Αν και οι χάρτες παράγονται τυχαία, η εκτέλεση αλγορίθμων σε διαφορετικούς χάρτες παράγει μη συγκρίσιμα αποτελέσματα.
- Για λόγους ευκολίας σύγκρισης, προτείνεται να τοποθετήσετε πολλαπλές γραφικές παραστάσεις στο ίδιο σύστημα αξόνων, όπου αυτό είναι εφικτό.

Visualization: Παράλληλα με τα παραπάνω σας δίνεται έτοιμη και μια κλάση η όποια κατασκευάζει ένα animation της αναζήτησης το όποιο υλοποιεί κάθε αλγόριθμος. Για την χρήση της κλάσης αυτής πρέπει να κάνετε τα εξής 4 βήματα:

1. Δημιουργία ενός instance της κλάσης εκτελώντας την παρακάτω εντολή: `visualization(Start, Goal)`. Η εντολή αυτή δημιουργεί ένα αντικείμενο τύπου visualization.
2. Το βίντεο που παράγεται κάθε φορά ουσιαστικά αποτελείται από stacked frames. Συνεπώς σε κάθε βήμα εκτέλεσης όπου δηλαδή θέλουμε να προσθέσουμε ένα frame στο βίντεο πρέπει να καλέσουμε την μέθοδο της κλάσης: `draw_step(grid, frontier, expanded_nodes)` όπου το πρώτο όρισμα είναι ένας χάρτης (τύπου Grid) το δεύτερο μια λίστα με το μέτωπο της αναζήτηση ενώ το τρίτο μια λίστα με τους κόμβους οι όποιοι έχουν ήδη επεκταθεί από τον αλγόριθμο.

3. (Προαιρετικό) Αν θέλουμε να προσθέσουμε στο animation και το βέλτιστο μονοπάτι που βρήκε ο αλγόριθμός μας μπορούμε να καλέσουμε την μέθοδο add_path(path) η οποία δέχεται σαν όρισμα μια λίστα με όλους τους κόμβους που ανήκουν στο βέλτιστο μονοπάτι (συμπεριλαμβανομένων και των κόμβων αρχής και τέλους).
4. Τέλος καλούμε την συνάρτηση `show_gif()` η όποια εμφανίζει το animation. Επίσης μπορούμε και να αποθηκεύσουμε το gif καλώντας την συνάρτηση `save_gif (filename)` (το αρχείο πρέπει να έχει κατάληξη .gif) καθώς επίσης μπορούμε να εμφανίσουμε μόνο το τελευταίο frame καλώντας την συνάρτηση `show_last_frame()` στο όποιο (αν έχουν γίνει όλα όπως παραπάνω) θα φαίνονται όλοι οι κόμβοι οι οποίοι έχουν επεκταθεί από τον αλγόριθμο, το τελευταίο μέτωπο και (προαιρετικά) και το βέλτιστο μονοπάτι.

Μπορείτε να καταλήξετε στα ίδια συμπεράσματα όσον αφορά την πολυπλοκότητα των αλγορίθμων παρατηρώντας το visualization;

```
from random import shuffle, randrange

def make_maze(w = 16, h = 8):
    vis = [[0] * w + [1] for _ in range(h)] + [[1] * (w + 1)]
    ver = [["|  "] * w + ['|'] for _ in range(h)] + [[]]
    hor = [["+--"] * w + ['+'] for _ in range(h + 1)]

    def walk(x, y):
        vis[y][x] = 1

        d = [(x - 1, y), (x, y + 1), (x + 1, y), (x, y - 1)]
        shuffle(d)
        for (xx, yy) in d:
            if vis[yy][xx]: continue
            if xx == x: hor[max(y, yy)][x] = "+  "
            if yy == y: ver[y][max(x, xx)] = "   "
            walk(xx, yy)

    walk(randrange(w), randrange(h))

    s = ""
    for (a, b) in zip(hor, ver):
        s += ''.join(a + ['\n'] + b + ['\n'])
    return s

```

```
if __name__ == '__main__':
    print(make_maze())
```

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        |                    |        |        |
+  +  +  +  +--+--+--+--+--+  +--+  +  +--+  +  +
|  |  |  |  |  |           |     |  |     |  |  |
+  +  +  +  +  +  +  +--+--+--+  +  +--+  +  +  +
|  |  |  |  |  |  |              |  |     |  |  |
+  +  +  +  +  +  +--+--+--+--+--+  +  +--+--+  +
|  |  |  |  |     |  |              |           |
+  +  +--+  +  +--+  +  +--+--+--+  +--+--+--+  +
|  |  |  |  |     |              |  |  |     |  |
+--+  +  +  +--+  +--+--+--+--+  +  +  +  +  +  +
|     |     |  |  |     |        |  |     |  |  |
+  +  +--+--+  +  +  +  +  +--+--+  +  +--+  +  +
|  |     |     |  |  |  |  |        |  |  |  |  |
+  +--+--+  +  +  +  +  +--+  +--+--+  +  +  +  +
|           |        |        |           |     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

```
# install latest matplotlib version
!pip install --upgrade matplotlib
```

```
Requirement already up-to-date: matplotlib in /usr/local/lib/python3.6/dist-packages (3.1.2)
Requirement already satisfied, skipping upgrade: python-dateutil>=2.1 in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied, skipping upgrade: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.6
Requirement already satisfied, skipping upgrade: kiwisolver>=1.0.1 in /usr/local/lib/python3.6/dist-packages (from ma
Requirement already satisfied, skipping upgrade: numpy>=1.11 in /usr/local/lib/python3.6/dist-packages (from matplotl
Requirement already satisfied, skipping upgrade: cycler>=0.10 in /usr/local/lib/python3.6/dist-packages (from matplot
Requirement already satisfied, skipping upgrade: six>=1.5 in /usr/local/lib/python3.6/dist-packages (from python-date
Requirement already satisfied, skipping upgrade: setuptools in /usr/local/lib/python3.6/dist-packages (from kiwisolve
```

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter
from IPython.display import HTML
```

```
 6
 7 class visualization:
 8     def __init__(self, S, F):
 9         '''
10           Η μέθοδος αυτή αρχικοποιεί ένα αντικείμενο τύπου visualization.
11           Είσοδος:
12           -> S: το σημείο εκκίνησης της αναζήτησης
13           -> F: το σημείο τερματισμού
14         '''
15         self.S = S
16         self.F = F
17         self.images = []
18
19     def draw_step(self, grid, frontier, expanded_nodes):
20         '''
21           Η συνάρτηση αυτή καλείται για να σχεδιαστεί ένα frame στο animation (πρακτικά έπειτα από την επέκταση κάθε κ
22           Είσοδος:
23           -> grid: Ένα χάρτης τύπου grid
24           -> frontier: Μια λίστα με τους κόμβους που ανήκουν στο μέτωπο της αναζήτησης
25           -> expanded_nodes: Μια λίστα με τους κόμβους που έχουν ήδη επεκταθεί
26           Επιστρέφει: None
27           Η συνάρτηση αυτή πρέπει να καλεστεί τουλάχιστον μια φορά για να μπορέσει να σχεδιαστεί ένα animation (πρεπεί
28         '''
29         image = np.zeros((grid.N, grid.N, 3), dtype=int)
30         image[grid.grid == 0] = [255, 255, 255]
31         image[grid.grid == 1] = [0, 0, 0]
32
33         for node in expanded_nodes:
34             image[node] = [0, 0, 128]
35
36         for node in frontier:
37             image[node] = [0, 225, 0]
38
39         image[self.S] = [50, 168, 64]
40         image[self.F] = [168, 50, 50]
41         self.images.append(image)
42
43     def add_path(self, path):
```

```
    def add_path(self, path):
        '''
          Η συνάρτηση αυτή προσθέτει στο τελευταίο frame το βέλτιστο μονοπάτι.
          Είσοδος:
          -> path: Μια λίστα η όποια περιέχει το βέλτιστο μονοπάτι (η οποία πρέπει να περιέχει και τον κόμβο αρχή και
          Έξοδος: None
        '''
        for n in path[1:-1]:
            image = np.copy(self.images[-1])
            image[n] = [66, 221, 245]
            self.images.append(image)

    def create_gif(self, fps = 30, repeat_delay = 2000):
        if len(self.images) == 0:
            raise EmptyStackOfImages("Error! You have to call 'draw_step' at  first.")
        fig = plt.figure()
        plt.axis('off')
        ims = []
        for img in self.images:
            img = plt.imshow(img)
            ims.append([img])
        ani = animation.ArtistAnimation(fig, ims, interval=1000//fps, blit=True, repeat_delay= repeat_delay)
        plt.close(fig)
        return ani

    def save_gif(self, filename, fps = 30):
        '''
            Η συνάρτηση αυτή ξαναδημιουργεί και αποθηκεύει το animation σε ένα αρχείο.
            Είσοδος:
            -> Το όνομα του αρχείου με κατάληξη .gif
            Έξοδος: (None)
        '''
        ani = self.create_gif(fps)
        writer = PillowWriter(fps= fps)
        ani.save(filename, writer=writer)

    def show_gif(self, fps= 30, repeat_delay = 2000):
        '''
```

```
            Η συνάρτηση αυτή εμφανίζει inline το animation.
            Είσοδος:
            -> fps: τα frames per second
            Έξοδος: Το αντικείμενο που παίζει το animation
            Exceptions: EmptyStackOfImages αν το animation δεν έχει ούτε ένα frame, δηλαδή αν η draw_step δεν έχει καλε
        '''
        ani = self.create_gif(fps, repeat_delay)
        # return HTML(ani.to_html5_video())
        return HTML(ani.to_jshtml())

    def show_last_frame(self):
        '''
            Η μέθοδος αυτή εμφανίζει inline το τελευταίο frame που έχει δημιουργηθεί.
            Είσοδος:
            Έξοδος: Το αντικείμενο που εμφανίζει την εικόνα.
            Exceptions: EmptyStackOfImages αν το animation δεν έχει ούτε ένα frame, δηλαδή αν η draw_step δεν έχει καλε
        '''
        if len(self.images) == 0:
            raise EmptyStackOfImages("Error! You have to call 'draw_step' at  first.")
        else:
            plt.imshow(self.images[-1])


class EmptyStackOfImages(Exception):
    pass
```

Μέρος Α: Κατασκευή χάρτη με διαφορετική πυκνότητα εμποδίων

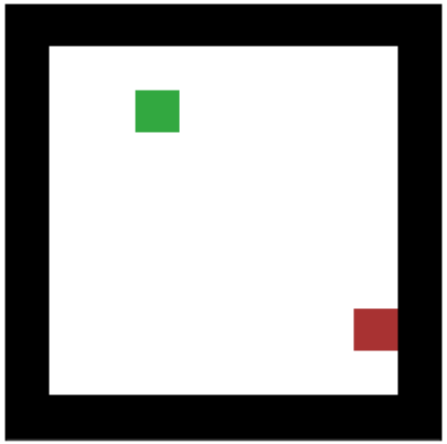
```
%matplotlib inline
import numpy as np
from queue import LifoQueue
from random import shuffle, uniform
import matplotlib.pyplot as plt


class grid:
```

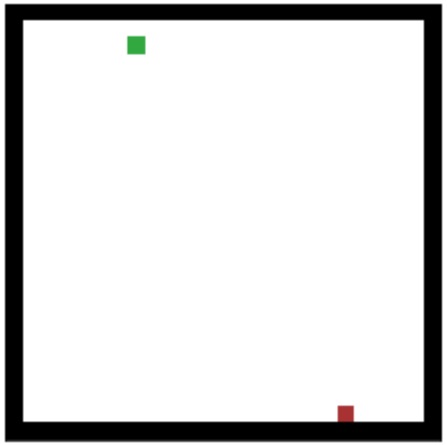
```
class grid:
    def __init__(self, N, S, F, p):

        ## Make sure start and end are within the grid
        assert N > 2
        assert S[0] < N
        assert S[1] < N
        assert F[0] < N
        assert F[1] < N

        assert S[0] > 0
        assert S[1] > 0
        assert F[0] > 0
        assert F[1] > 0

        self.N = N

        self.grid = np.zeros((N, N), dtype=np.int32)

        ## Surround the grid with obstacles
        self.grid[0, :] = 1
        self.grid[N - 1, :] = 1
        self.grid[:, 0] = 1
        self.grid[:, N - 1] = 1

        obstacle_free_points = {S, F}

        ### Fill the grid with obstacles.
        ### An obstacle at position (x,y) -> grid[x,y]=1
        ### Ensure there is a path from S to F
        ### Your code here ###


    def adjacent(self, node):
        adjacent_nodes = []
        for n in (node[0] - 1, node[1]), (node[0] + 1, node[1]), (node[0], node[1] - 1), (node[0], node[1] + 1):
            if self.grid[n] == 0:
                adjacent_nodes.append(n)
```

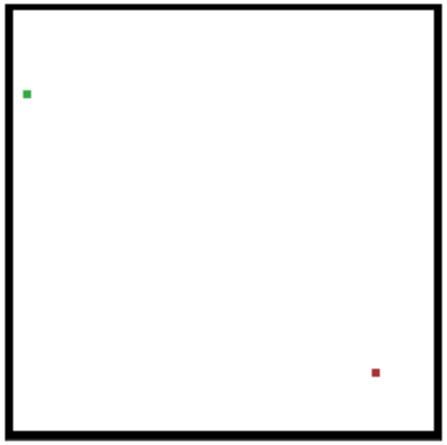
```
        return adjacent_nodes
```

```
    def draw_map(self, S=None, F=None, path=None):

        image = np.zeros((self.N, self.N, 3), dtype=int)

        image[self.grid == 0] = [255, 255, 255]
        image[self.grid == 1] = [0, 0, 0]
        if S:
            image[S] = [50, 168, 64]
        if F:
            image[F] = [168, 50, 50]
        if path:
            for n in path[1:-1]:
                image[n] = [66, 221, 245]

        plt.imshow(image)
        plt.xticks([])
        plt.yticks([])
        plt.show()
```

```
# show different grid sizes with different obstacle densities

for N, S, F, p in (10, (2, 3), (7, 8), .3), (25, (2, 7), (23, 19), .5), (50, (10, 2), (42, 42), .8):
    map = grid(N, S, F, p)
    map.draw_map(S, F)
```



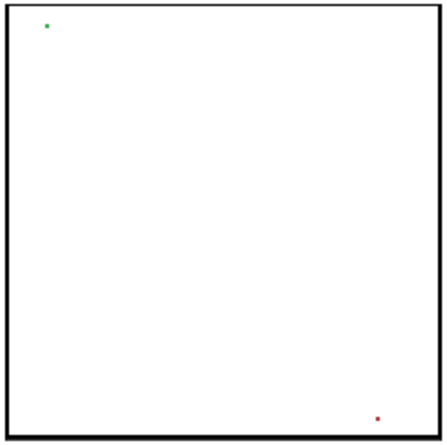




Μέρος Β: μελέτη επίδοσης αλγορίθμων pathfinding σε διαφορετικά είδη grids

```
class pathfinder:
    def __init__(self, S, F, grid, c, h):
        self.S = S  ## S is the starting point - a tuple (x,y)
        self.F = F  ## F is the goal
        self.grid = grid ## A grid object (from A)
        self.vis = visualization(S, F)
        self.path = []
        self.cost = c
        self.heuristic = h



    ## Καλέστε την self.vis.draw_step()
    ### Fill the path list with the coordinates of each point in the path from S to F
    ### Your code here


    def get_path(self):
        return self.path

```

```
N = 100
S = (5, 9)
F = (95, 85)
p = .6

map = grid(N, S, F, p)
pf = pathfinder(S, F, map, lambda x, y: 1, lambda x, y: 0)
map.draw_map(S, F, pf.get_path())
```



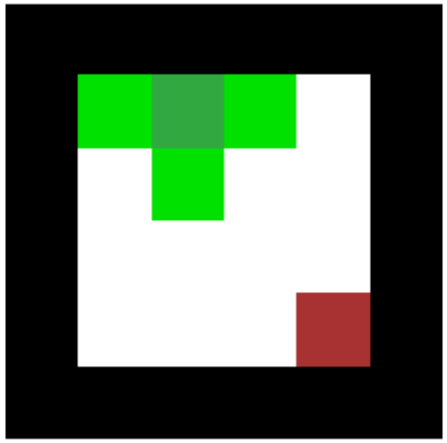
Μέρος Γ: σύγκριση αλγορίθμων αναζήτησης

```
# your code goes here
```

Παράδειγμα χρήσης της κλάσης Οπτικοποίησης

```
#Ορίζουμε τα χαρακτηριστικά του χάρτη
N= 6
S = (1, 2)
F = (4, 4)
p = 0
map = grid(N, S, F, .3) #αρχικοποίούμε τον χάρτη
#ορίζουμε το αντικείμενο που θα βρεί την βέλτιστη διαδρομή
#σε αυτό ορίζεται και ένα instance της κλάσης visualization με το όνομα vis
pf = pathfinder(S, F, map, lambda x, y: 1, lambda x, y: 0)


#σε κάθε βήμα του αλγορίθμου pathfinding θα εκτελούνται τα παρακάτω
frontier = [(1, 3), (2, 2), (1, 1)] # θα βρίσκουμε το μέτωπο αναζήτησης
```

```
frontier =  [(1, 3), (2, 2), (1, 1)] # θα βρίσκουμε το μετωπο αναζητησης
expanded_nodes =  [(1, 2)] # θα βρίσκουμε τους κόμβους που έχουν ήδη επεκταθεί
#καλούμε την παρακάτω μέθοδο για να εισάγουμε αυτό το frame στο animation
pf.vis.draw_step(map, frontier, expanded_nodes) #σχεδίαση ένος frame

#κάνουμε το παραπάνω για κάθε βήμα της αναζήτησης
frontier =  [(1, 4), (2, 3), (2, 2), (1, 1)]
expanded_nodes =  [(1, 2), (1, 3)]
pf.vis.draw_step(map, frontier, expanded_nodes) #σχεδίαση επόμενου frame

frontier =  [(2, 4), (2, 3), (2, 2), (1, 1)]
expanded_nodes =  [(1, 2), (1, 3), (1, 4)]
pf.vis.draw_step(map, frontier, expanded_nodes) #σχεδίαση επόμενου frame

frontier =  [(3, 4), (2, 3), (2, 2), (1, 1)]
expanded_nodes =  [(1, 2), (1, 3), (2, 4), (1, 4)]
pf.vis.draw_step(map, frontier, expanded_nodes) #σχεδίαση επόμενου frame

frontier =  [(4, 4), (2, 3), (2, 2), (3, 3), (1, 1)]
expanded_nodes =  [(1, 2), (1, 3), (1, 4), (3, 4), (2, 4)]
pf.vis.draw_step(map, frontier, expanded_nodes) #σχεδίαση επόμενου frame

#εισάγουμε το Path στο animation
path = [(1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4)]
pf.vis. add_path(path)

#καλούμε την μέθοδο για να παρουσιάσουμε το animation στο Notebook
pf.vis.show_gif(fps = 1)

#αν θέλουμε μπορούμε και να το αποθηκεύσουμε τοπικά
# pf.vis.save_gif("mygif.gif")

#ή μπορούμε να παρουσιάσουμε μόνο το τελευταίο frame, όπου φαίνονται όλοι οι κόμβοι που έχουν επεκταθεί μαζί με το τελ
#και το βέλτιστο μονοπάτι
# pf.vis.show_last_frame()
```



Once Loop Reflect