Δημήτριος Κυριακίδης el17077

Χαράλαμπος Δάλπης el17067

Τεχνητή Νοημοσύνη: Εργαστηριακή Άσκηση 1

Ο στόχος της εργασίας είναι η εφαρμογή αλγορίθμων αναζήτησης και εύρεσης καλύτερου μονοπατιού σε λαβύρινθο διαστάσεων $N \times N$

Εκφώνηση

Μέρος 1

Ο στόχος του πρώτου μέρους είναι η κατασκευή **λαβυρίνθων**. Σε αυτό το άρθρο της Wikipedia μπορείτε να βρείτε διάφορους αλγορίθμους για αυτό το σκοπό. Προσέξτε ότι οι αλγόριθμοι του παραπάνω άρθρου κατασκευάζουν ένα μόνο μονοπάτι από κάθε σημείο του λαβυρίνθου προς οποιοδήποτε άλλο (η δομή του λαβυρίνθου έχει μορφή συνεκτικού δέντρου). Καλείστε να τροποποιήσετε τον αλγόριθμο της επιλογής σας (ενδεχομένως και πέραν αυτών που υπάρχουν στο προαναφερθέν άρθρο) ώστε οι λαβύρινθοι που κατασκευάζει να έχουν τουλάχιστον δύο (2) μονοπάτια μεταξύ ενός σημείου αφετηρίας (S) και ενός σημείου τερματισμού (F). Η τροποποίηση του αλγορίθμου μπορεί να γίνει και με πιθανοτικό τρόπο, αλλά σε αυτή την περίπτωση θα πρέπει να βεβαιωθείτε πως οι λαβύρινθοι που θα χρησιμοποιήσετε στα επόμενα ερωτήματα έχουν τουλάχιστον δύο μονοπάτια μεταξύ των S και F. Αναφέρετε ποιον αλγόριθμο επιλέξατε, περιγράψτε τον συνοπτικά καθώς και τον τρόπο που αντιμετωπίσατε το παραπάνω πρόβλημα, και υλοποιήστε τον στο κελί κώδικα του Μέρους 1.

Σε όλα τα ερωτήματα φροντίστε να χρησιμοποιήσετε **δομημένο** κώδικα, με **σχόλια** που επισημαίνουν τη λογική του.

Παραδείγματα λαβυρίνθων

Μέρος 2

Στο δεύτερο μέρος της εργασίας θα υλοποιήσετε τον αλγόριθμο A* για την εύρεση συντομότερων μονοπατιών μεταξύ δυο κόμβων για τους λαβυρίνθους που παράγονται από τον κώδικα του πρώτου μέρους. Θα κατασκευάσετε και διάφορες συναρτήσεις κόστους τόσο για την μέτρηση των πραγματικών αποστάσεων όσο και για την εκτίμηση των αποστάσεων από έναν κόμβο στον κόμβο στόχο (heuristic).

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 πειραματιστείτε 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 μέρος καλείστε να συγκρίνετε την απόδοση των αλγορίθμων που υλοποιήσατε στο πρώτο μέρος, με βάση την πολυπλοκότητά τους και το κατά πόσο μπορούν να βρούν το βέλτιστο μονοπάτι.

Αρχικά, θα πρέπει να κατασκευάσετε πλήθος λαβυρίνθων με μέγεθος $N \in [10, 11, \ldots, 50]$ (τουλάχιστον 10 λαβυρίνθους για κάθε N).

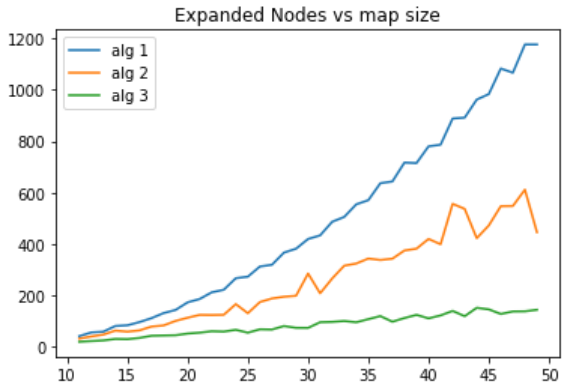
Έπειτα, για τους τρεις αλγορίθμους που υλοποιήσατε στο δεύτερο μέρος, καθώς και για ό,τι επιπλέον υλοποιήσατε (πχ επιπλέον ευριστικές), ζητείται να συλλέξετε τα εξής χαρακτηριστικά:

- 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 δύο γραφικές παραστάσεις:

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

Ένα παράδειγμα φαίνεται παρακάτω:



Σας ζητείται σχολιάσετε τιςγραφικές παραστάσεις που θα φτιάξετε, και συγκεκριμένα το πώς μεταβάλλεται η συμπεριφορά των αλγορίθμων συναρτήσει του N.

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 πειραματιστείτε με τυχαία σημεία εκκινησης και τερματισμού.
- Αν και οι χάρτες παράγονται τυχαία, η εκτέλεση αλγορίθμων σε διαφορετικούς χάρτες παράγει μη συγκρίσιμα αποτελέσματα, οπότε οι αλγόριθμοι πρέπει να τρέξουν στους ίδιους χάρτες.
- Για λόγους ευκολίας σύγκρισης, προτείνεται να τοποθετήσετε πολλαπλές γραφικές παραστάσεις στο ίδιο σύστημα αξόνων, όπου αυτό είναι εφικτό.

Για το πως κατασκευάζουμε γραφικές παραστάσεις μέσω python, μπορείτε να βρείτε παραδείγματα στον κώδικα που δίνεται, καθώς και (για το ζητούμενο της άσκησης) εδώ .

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 Maze) 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;

Μέρος 4

Στο τέταρτο και τελευταίο μέρος της εργασίας σας δίνεται η δυνατότητα να κατασκευάσετε λαβυρίνθους με το χέρι.

Καλείστε λοιπόν να **κατασκευάσετε** και να **σχολιάσετε** τους παρακάτω 5 λαβυρίνθους:

- **Λαβύρινθος 1**: Ο Hill-climbing δεν βρίσκει μονοπάτι από την αφετηρία στο στόχο.
- **Λαβύρινθος 2**: Ο A *αργεί όσο το δυνατόν περισσότερο να βρει το βέλτιστο μονοπάτι ανεξαρτήτως της μετρικής απόστασης που χρησιμοποιείται ως ευριστική. Σε τι είδους μονοπάτι η επιλογή διαφορετικής μετρικής απόστασης (Ευκλίδεια/Manhattan) ως ευριστικής οδηγεί τον A* να βρίσκει το μονοπάτι πιο γρήγορα;
- **Λαβύρινθος 3**: Ο αλγόριθμος άπληστης αναζήτησης πρώτα στο καλύτερο (greedy best first) αργεί σημαντικά περισσότερο από τον A* για να καταλήξει στο στόχο. Σε ποια περίπτωση οι δύο αλγόριθμοι θα συμπεριφέρονταν με τον ίδιο τρόπο;
- **Λαβύρινθος 4**: Ο A *καταφέρνει να βρει το μονοπάτι προς το στόχο σημαντικά πιο γρήγορα από τον Dijkstra. Ο χώρος πρέπει αναγκαστικά να περιέχει εμπόδια. Πώς επηρεάζει το μέγεθος/είδος του εμποδίου τον αριθμό βημάτων που χρειάζεται ο Dijsktra και πώς τον αριθμό βημάτων του A*;
- **Λαβύρινθος 5**: Ο A *και ο Dijkstra φτάνουν στο στόχο πραγματοποιώντας τον ίδιο ακριβώς αριθμό βημάτων για την προσέγγιση της βέλτιστης διαδρομής. Σημείωση: **αγνοήστε** την τετριμμένη περίπτωση όπου ο A* συμπεριφέρεται όπως ο Dijkstra (h(n)=0).

Ο A* σε όλες τις περιπτώσεις πραγματοποιεί τη βέλτιστη δυνατή αναζήτηση.

Για κάθε λαβύρινθο που σχηματίζετε, δώστε μια σύντομη εξήγηση για την επιλογή σας.

ΠΡΟΣΟΧΗ: το output των κελιών που σχηματίζετε διατηρείται για περιορισμένο χρονικό διάστημα στο notebook, γι αυτό καλείστε να αποθηκεύσετε τη φωτογραφία (το πραγματοποιεί η συνάρτηση draw()) και να τη μεταφέρετε στο ανάλογο markdown ώστε να μπορέσουμε να την αξιολογήσουμε.

Στη συνέχεια, καλείστε να τρέξετε τους αλγορίθμους pathfinding πάνω στους χάρτες που σχηματίσατε. Για το σκοπό αυτό, μετά την εκτέλεση της draw() καλείτε τη συνάρτηση draw_grid() που μετατρέπει την εικόνα που σχηματίσατε σε χάρτη πάνω στον οποίο μπορούν να τρέξουν οι αλγόριθμοι με χρήση της συνάρτησης pathfinder(). Χρειάζεται να επιλέξετε κατάλληλα τα ορίσματα της pathfinder ώστε να καλέσετε το σωστό αλγόριθμο αναζήτησης σε κάθε ερώτημα. Εκτελέστε τα αντίστοιχα κελιά με τον κώδικα που σας δίνεται και παρουσιάστε την έξοδο αυτών. Φυσικά, εάν ένα ερώτημα περιλαμβάνει την εκτέλεση περισσότερων του ενός αλγορίθμου, αντιγράφετε τα αντίστοιχα κελιά για κάθε αλγόριθμο και τοποθετείτε τα σωστά ορίσματα. Τέλος, η show_gif() σας παρουσιάζει το animation, όπως και στο Μέρος 3. **Παρατήρηση:** εφόσον δε σας δίνεται/δεν έχετε υλοποιήσει τον Hill climbing, δε σας ζητείται να τον τρέξετε στο χάρτη που θα σχηματίσετε.

Κώδικας visualization

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter
from IPython.display import HTML

class visualization:
    def __init__(self, S, F):
```

```
        '''
          Η μέθοδος αυτή αρχικοποιεί ένα αντικείμενο τύπου visualization.
          Είσοδος:
          -> S: το σημείο εκκίνσης της αναζήτησης
          -> F: το σημείο τερματισμού
        '''
        self.S = S
        self.F = F
        self.images = []

    def draw_step(self, grid, frontier, expanded_nodes):
        '''
          Η συνάρτηση αυτή καλείται για να σχεδιαστεί ένα frame στο animation (πρακτικά έπειτα από την επέκταση κάθε κόμ
          Είσοδος:
          -> grid: Ένα χάρτης τύπου grid
          -> frontier: Μια λίστα με τους κόμβους που ανήκουν στο μέτωπο της αναζήτησης
          -> expanded_nodes: Μια λίστα με τους κόμβους που έχουν ήδη επεκταθεί
          Επιστρέφει: None
          Η συνάρτηση αυτή πρέπει να καλεστεί τουλάχιστον μια φορά για να μπορέσει να σχεδιαστει ένα animation (πρεπεί τ
        '''
        image = np.zeros((grid.N, grid.N, 3), dtype=int)
        image[~grid.grid] = [0, 0, 0]
        image[grid.grid] = [255, 255, 255]
        # Use this to treat 1/True as obstacles
        # image[grid.grid] = [0, 0, 0]
        # image[~grid.grid] = [255, 255, 255]

        for node in expanded_nodes:
            image[node] = [0, 0, 128]

        for node in frontier:
            image[node] = [0, 225, 0]

        image[self.S] = [50, 168, 64]
        image[self.F] = [168, 50, 50]
        self.images.append(image)

    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 το
          Έξοδος: None
        '''
        for n in path[1:-1]:
            image = np.copy(self.images[-1])
            image[n] = [66, 221, 245]
            self.images.append(image)
        for _ in range(100):
            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 καλεσ
        '''
        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 δημιουργήθει.
            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 καλεσ
        '''
        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
```

Λύση

Μέρος 1

Στο παρακάτω κελί κώδικα σας δίνεται ο σκελετός της κλάσης `Maze`, όπου και θα υλοποιήσετε τον αλγόριθμο κατασκευής λαβυρίνθων. Για να λειτουργεί σωστά η οπτικοποίηση, ο λαβύριθνος αναπαριστάται ως ένας $N \times N$ boolean πίνακας στο attribute `grid` της κλάσης. Η τιμή `False` σε ένα κελί του πίνακα αναπαριστά την ύπαρξη εμποδίου στο σημείο αυτό, ενώ η τιμή `True` ελεύθερο κελί.

Μπορείτε να τροποποιήσετε τον δοσμένο κώδικα όπως θέλετε.

In [ ]:
```
%matplotlib inline
import numpy as np
from queue import LifoQueue, PriorityQueue
import random
import matplotlib.pyplot as plt

class Maze:
    def __init__(self, N, S, F):
        """
        N: integer that indicates the size of the NxN grid of the maze
        S: pair of integers that indicates the coordinates of the starting point (S)
        F: pair of integers that indicates the coordinates of the finish point (F)
        You can add any other parameters you want to customize maze creation (e.g. variables that
        control the creation of additional paths)
        """

        assert N > 2

        ## Make sure start and end are within the grid

        assert S[0] <= N-1
        assert S[1] <= N-1
        assert F[0] <= N-1
        assert F[1] <= N-1

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

        # Add here any additional constraints your implementation may have

        self.N = N
        self.S = S
        self.F = F

        # Grid initialized with obstacles (array of 1/True)
        # 0/False indicates available cells
        self.grid = np.ones((N, N), dtype = bool)
```

```
        ## Start and end position have no obstacles
        self.grid[S] = 0

        ## YOUR CODE HERE
        self.createMaze()

    def createMaze(self):
        self.obList = []
        self.addWalls(self.S)
        while(self.obList):
            (currentX, currentY) = random.choice(self.obList)
            C = (currentX, currentY)
            self.obList.remove(C)
            if(not self.grid[C]): continue
            R = (currentX, currentY + 1) # Right
            L = (currentX, currentY - 1) # Left
            U = (currentX + 1, currentY) # Up
            D = (currentX - 1, currentY) # Down

            rlFlag = udFlag = False ## rlFlag means it's possible to expand maze in the right-left direction
            if(self.isInGrid(R) and self.isInGrid(L)):
                if(self.grid[L] != self.grid[R]): rlFlag = True
            if(self.isInGrid(U) and self.isInGrid(D)):
                if(self.grid[U] != self.grid[D]): udFlag = True
            if(rlFlag and udFlag):
                ## choose randomly
                direction = random.choice(["RL", "UD"])
                self.addToMaze(C, direction)
            elif(rlFlag): self.addToMaze(C, "RL")
            elif(udFlag): self.addToMaze(C, "UD")

        self.grid[self.F] = 0
        # θέτοντας το grid[F] ίσο με 0 στο τέλος της διαδικασίας εξασφαλίζεται η ύπαρξη διαδρομής S->F
        # και η εγκυρότητα του λαβυρίνθου
        self.carveSecondPath()

    def isInGrid(self, point):
        (x, y) = point
        return x >= 0 and x < self.N and y >=0 and y < self.N


    ## add all the valid surrounding walls of pair to the list
    def addWalls(self, pair):
        (x, y) = pair
        if(self.isInGrid((x + 1, y)) and self.grid[(x + 1, y)]): self.obList.append((x + 1, y))
        if(self.isInGrid((x - 1, y)) and self.grid[(x - 1, y)]): self.obList.append((x - 1, y))
        if(self.isInGrid((x, y + 1)) and self.grid[(x, y + 1)]): self.obList.append((x, y + 1))
        if(self.isInGrid((x, y - 1)) and self.grid[(x, y - 1)]): self.obList.append((x, y - 1))


    def addToMaze(self, wall, direction):
        (wx, wy) = wall
        if(direction == "RL"):
            wr = (wx, wy + 1)
            wl = (wx, wy - 1)
            self.grid[wl] = self.grid[wall] = self.grid[wr] = 0
            self.addWalls(wl)
            self.addWalls(wr)
        else:
            wu = (wx + 1, wy)
            wd = (wx - 1, wy)
            self.grid[wu] = self.grid[wall] = self.grid[wd] = 0
            self.addWalls(wu)
            self.addWalls(wd)


    def carveSecondPath(self):
        ## find all reachable walls from s with bfs
        ## same for f
        ## then eligible walls: not edge and surrounded by wall rl or ud
        ## break one, favoring rl first
        reachableFromS = self._getAllReachableWalls(self.S)
        reachableFromF = self._getAllReachableWalls(self.F)
        for W in reachableFromS & reachableFromF:
            (Wx, Wy) = W
            if(Wx != 0 and Wx != self.N - 1 and Wy != 0 and Wy != self.N - 1):
                R = (Wx, Wy + 1)
                L = (Wx, Wy - 1)
                U = (Wx + 1, Wy)
                D = (Wx - 1, Wy)
```

```
                if(self.isInGrid(R) and self.isInGrid(L) and self.grid[R] == 1 and self.grid[L] == 1):
                    self.grid[W] = 0
                    break
                elif(self.isInGrid(U) and self.isInGrid(D) and self.grid[U] == 1 and self.grid[D] == 1):
                    self.grid[W] = 0
                    break


    def _getAllReachableWalls(self, V):
        Q = PriorityQueue()
        reachableWalls = set()
        visited = np.zeros((self.N, self.N), dtype = bool)
        Q.put(V)
        visited[V] = True
        while(not Q.empty()):
            C = Q.get()
            for NB in self._getValidNeighbors(C):
                if(visited[NB]): continue
                if(self.grid[NB] == 1): reachableWalls.add(NB)
                else: Q.put(NB)
                visited[NB] = True
        return reachableWalls


    def _getValidNeighbors(self, V):
        ans = []
        (x, y) = V
        if(self.isInGrid((x + 1, y))): ans.append((x + 1, y))
        if(self.isInGrid((x - 1, y))): ans.append((x - 1, y))
        if(self.isInGrid((x, y + 1))): ans.append((x, y + 1))
        if(self.isInGrid((x, y - 1))): ans.append((x, y - 1))
        return ans


    def draw_maze(self, path = None):
        """
        Draws the maze as an image. Considers grid values of 0/False to represent obstacles and
        values of 1/True to represent empty cells, but this can be customized. Obstacles are painted
        black and empty cells are painted white. Starting point is painted green and finish point red.
        Optionally accepts as a parameter a path within the maze which is painted blue.
        """
        image = np.zeros((self.N, self.N, 3), dtype = int)
        image[~self.grid] = [0, 0, 0]
        image[self.grid] = [255, 255, 255]

        # Uncomment the next 2 lines of code to treat 1/True as obstacles (and 0/False as free maze cells)
        image[self.grid] = [0, 0, 0]
        image[~self.grid] = [255, 255, 255]

        image[self.S] = [50, 168, 64]
        image[self.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()
```

Τρέξτε το παρακάτω κελί κώδικα πριν παραδώσετε την άσκηση για να φαίνονται παραδείγματα λαβυρίνθων. Μπορείτε να το τροποποιήσετε.

In [ ]:
```
for N, S, F in (11, (1, 3), (7, 9)), (25, (3, 7), (23, 19)), (51, (9, 3), (41, 41)):
    maze = Maze(N, S, F)
    maze.draw_maze()
```

Σχολιασμός

Αρχικά, έχουμε κάνει την σύμβαση πως στον πίνακα τα ελεύθερα κελλιά αναπαρίστανται από το 0, και τα εμπόδια από το 1.

Για την κατασκευή του λαβυρίνθου επελέγη ο αλγόριθμος Prim για Minimum Spanning Trees:

1. Start with a grid full of walls.
2. Pick a cell, mark it as part of the maze. Add the walls of the cell to the wall list.
3. While there are walls in the list:
    A. Pick a random wall from the list. If only one of the cells that the wall divides is visited, then:
        a. Make the wall a passage and mark the unvisited cell as part of the maze.
        b. Add the neighboring walls of the cell to the wall list.
    B. Remove the wall from the list.

("Randomized Prim")

Αυτή η μέθοδος παράγει εγγυημένα λαβυρίνθους με μια τουλάχιστον λύση. Για την εξασφάλιση της δεύτερης, ακολουθείται η εξής διαδικασία:

1. Με αναζήτηση κατά πλάτος εντοπίζονται όλα τα κελλιά - τοίχοι προσβάσιμα και από την S και από την F
2. Εξ αυτών επιλέγεται ένα, η αφαίρεση του οποίου δεν θα οδηγούσε σε μη έγκυρο λαβύρινθο με κατάλληλες συνθήκες, και μετατρέπεται σε ελεύθερο κελλί.

Μέρος 2

Παρακάτω σας δίνετε ο σκελετός της κλάσης Pathfinder η οποία αρχικοποιείται με ένα αντικείμενο `Maze` (από το πρώτο μέρος), τη συνάρτηση πραγματικού κόστους `c` και την ευριστική `h`. Σε αυτήν θα υλοποιήσετε τον αλγόριθμο αναζήτησης A*.

In [ ]:
```
from queue import PriorityQueue

class Pathfinder:
    def __init__(self, maze, c, h):
        """
        maze : Αντικείμενο τύπου Maze (από Μέρος 1)
        c : Συνάρτηση που υπολογίζει την πραγματική απόσταση μεταξύ δύο σημείων
        h : Συνάρτηση που υπολογίζει την ευριστική μεταξύ δύο σημείων
        """
        self.maze = maze
        self.S = maze.S
        self.F = maze.F
        self.vis = visualization(maze.S, maze.F)
        self.path = []
        self.cost = c
        self.heuristic = h


        ### Fill the path list with the coordinates of each point in the path from maze.S to maze.F
        ### Your code here

        ## ADDITIONAL FIELDS
        ## Η κλάση Priority Queue του module queue υλοποιεί αυτομάτως την ταξινόμηση βάσει τους πρώτου στοιχείου του tup
        ## εν προκειμένω της τιμής ευρετικής
        ## https://docs.python.org/3/library/queue.html#queue.PriorityQueue
        self.Q = PriorityQueue()
        self.visited = np.zeros((self.maze.N, self.maze.N), dtype = bool)
        self.parent = {} # key is a node, value is its parent
        self.temporaryValue = np.zeros((self.maze.N, self.maze.N), dtype = int) # used by A* to assess whether a
        # node that has already been seen must be updated with a lower value

    def find_path(self):
        ## YOUR CODE HERE
        (sx, sy) = self.S
        f = self.heuristic(self.S, self.F) ## F(n) = g(n) + h(n), g = 0 for starting node
        self.Q.put((f, (sx, sy)))
        while(not self.Q.empty()):
            pair = self.Q.get() ## get() both returns and pops
            # print(pair)
            (value, currentNode) = pair ## currentNode stores the coordinates
            if(currentNode == self.F):
                # create path based on parent
                head = currentNode
                while(head != self.S):
                    self.path.append(head)
                    head = self.parent[head]
                self.path.append(self.S)
                return

            self.pushEligibleNeighbors(pair)
            self.visited[currentNode] = True


    def pushEligibleNeighbors(self, cell):
        (value, (x, y)) = cell
        if(self.maze.isInGrid((x + 1, y)) and self.maze.grid[(x + 1, y)] == 0): self.util(cell, x + 1, y)
        if(self.maze.isInGrid((x - 1, y)) and self.maze.grid[(x - 1, y)] == 0): self.util(cell, x - 1, y)
        if(self.maze.isInGrid((x, y + 1)) and self.maze.grid[(x, y + 1)] == 0): self.util(cell, x, y + 1)
        if(self.maze.isInGrid((x, y - 1)) and self.maze.grid[(x, y - 1)] == 0): self.util(cell, x, y - 1)


    def util(self, cell, neighborX, neighborY):
        (value, parent) = cell
        neighbor = (neighborX, neighborY)
        f = value - self.heuristic(parent, self.F) + self.cost(*neighbor) + self.heuristic(neighbor, self.F)
        if(not self.visited[neighbor] or f < self.temporaryValue[neighbor]):
            self.Q.put((f, neighbor))
            self.parent.update({neighbor : parent})
            self.temporaryValue[neighbor] = f

    def get_path(self):
        self.find_path()
        return self.path

    def getPathLength(self):
        '''
```

```
        Only call after executing find_path
        '''
        return len(self.path)

    def getExpandedNodes(self):
        '''
        Only call after executing find_path
        '''
        return np.count_nonzero(self.visited) ## effectively counts 1s
```

Για να δείτε το μονοπάτι που κατασκευάσατε, μπορείτε να καλείτε τη μέθοδο `draw_map` του αντικειμένου `maze` οπως φαίνεται στο παρακάτω κελί.

In [ ]:
```
## heuristic functions
from math import sqrt

def manhattanDistance(A: (int, int), B: (int, int)) -> int:
    (Ax, Ay) = A
    (Bx, By) = B
    return abs(Ax - Bx) + abs(Ay - By)

def euclideanDistance(A: (int, int), B: (int, int)) -> float:
    (Ax, Ay) = A
    (Bx, By) = B
    return sqrt((Ax - Bx) ** 2 + (Ay - By) ** 2)
```

Στην οριακή περίπτωση όπου $g(n) = 0$, ο A λειτουργεί ως Best-First αναζήτηση βάσει μόνο της ευρετικής. Αντιστοίχως, αν $h(n) = 0$, εκφυλίζεται σε uniform cost αναζήτηση (Dijkstra).*

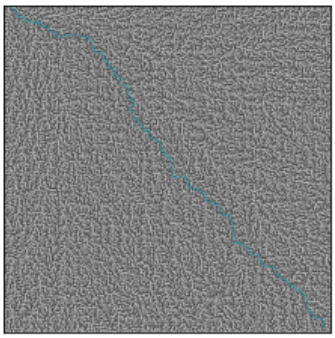
In [ ]:
```
## Create a 41x41 maze
N = 41
S = (5, 9)
F = (37, 37)
maze = Maze(N, S, F)

## Find and visualize the path

# g(n) = 1, h(n) = 0 (Dijkstra)
pf = Pathfinder(maze = maze, c = lambda x, y: 1, h = lambda x, y: 0)
maze.draw_maze(pf.get_path())

#g(n) = 1, h(n) = manhattan (A*)
pf = Pathfinder(maze = maze, c = lambda x, y: 1, h = lambda x, y: manhattanDistance(x, y))
maze.draw_maze(pf.get_path())

#g(n) = 1, h(n) = euclidean (A*)
pf = Pathfinder(maze = maze, c = lambda x, y: 1, h = lambda x, y: euclideanDistance(x, y))
maze.draw_maze(pf.get_path())

#g(n) = 0, h(n) = manhattan (Best First)
pf = Pathfinder(maze = maze, c = lambda x, y: 0, h = lambda x, y: manhattanDistance(x, y))
maze.draw_maze(pf.get_path())

#g(n) = 0, h(n) = euclidean (Best First)
pf = Pathfinder(maze = maze, c = lambda x, y: 0, h = lambda x, y: euclideanDistance(x, y))
maze.draw_maze(pf.get_path())
```

Παρατηρούμε πως στην τελευταία εκτέλεση ο Best First βρίσκει διαφορετικό, μη βέλτιστο μονοπάτι.

Για να χρονομετρήσετε το τρέξιμο ενός κελιού μπορείτε να χρησιμοποιήσετε το μαγικό `%%time`

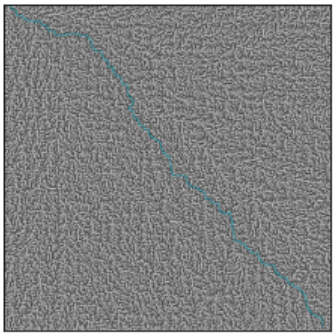
In [ ]:
```
## Create a 500x500 maze
N = 500
S = (5, 9)
F = (495, 485)
maze = Maze(N, S, F)
```

In [ ]:
```
%%time
# g(n) = 1, h(n) = 0
pf = Pathfinder(maze = maze, c = lambda x, y: 1, h = lambda x, y: 0)
maze.draw_map(pf.get_path())
```



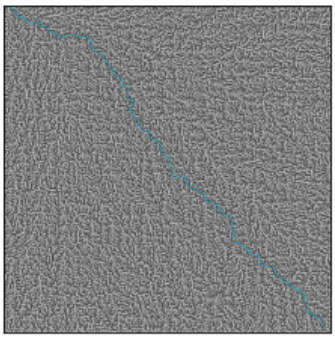
```
Wall time: 1.6 s
```

In [ ]:
```
%%time
#g(n) = 1, h(n) = manhattan
pf = Pathfinder(maze = maze, c = lambda x, y: 1, h = lambda x, y: manhattanDistance(x, y))
maze.draw_map(pf.get_path())
```



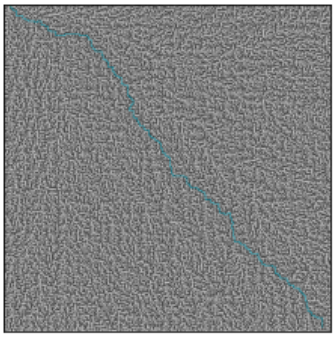
```
Wall time: 547 ms
```

In [ ]:
```
%%time
#g(n) = 1, h(n) = euclidean
pf = Pathfinder(maze = maze, c = lambda x, y: 1, h = lambda x, y: euclideanDistance(x, y))
maze.draw_map(pf.get_path())
```



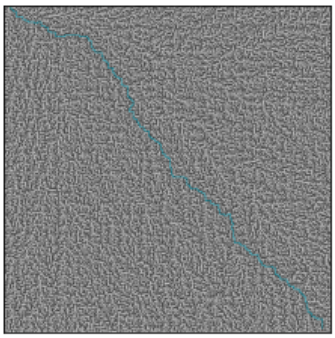
```
Wall time: 1.6 s
```

In [ ]:
```
%%time
#g(n) = 0, h(n) = manhattan
pf = Pathfinder(maze = maze, c = lambda x, y: 0, h = lambda x, y: manhattanDistance(x, y))
maze.draw_map(pf.get_path())
```



```
Wall time: 398 ms
```

In [ ]:
```
%%time
#g(n) = 0, h(n) = euclidean
pf = Pathfinder(maze = maze, c = lambda x, y: 0, h = lambda x, y: euclideanDistance(x, y))
maze.draw_map(pf.get_path())
```



```
Wall time: 881 ms
```

Μέρος 3

Στο παρακάτω κελί σας δίνεται ένα παράδειγμα για την κατασκευή γραφικών παραστάσεων. Θα πρέπει να κατασκευάσετε τις δύο γραφικές παραστάσεις σύμφωνα με τα ζητούμενα της άσκησης.

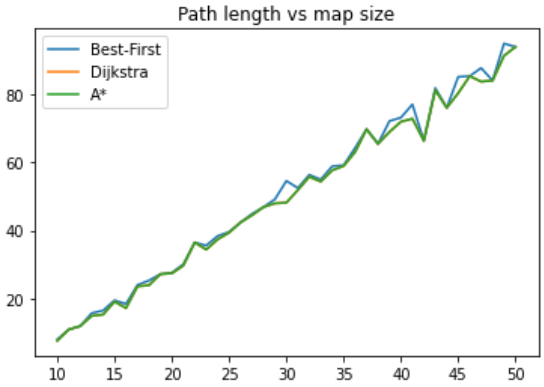
In [ ]:
```
algorithms = ["Best-First", "Dijkstra", "A*"]
```

In [ ]:
```
def getStatistics(maze, c, h):
    pf = Pathfinder(maze = maze, c = c, h = h)
    pf.find_path()
    return (pf.getPathLength(), pf.getExpandedNodes())
```

In [ ]:
```
def generateData(Nmin = 10, Nmax = 50, step = 1, M = 10):
    data = {} ## keys: algorithm names, values: a dictionary with key1: pathLengths, value1 corresponding list, key2: ex
    for alg in algorithms:
        data.update({alg : {"pathLengths" : {}, "expNodes" : {}}})

    pathLengths = {} ## key: maze dimension, value: list of lengths for the 10 mazes of that dimension
    expNodes = {}
    for n in range(Nmin, Nmax + 1, step): ## maze dimension
        avgLength = 0
        avgNodes = 0
        data["Best-First"]["pathLengths"].update({n : 0})
        data["Best-First"]["expNodes"].update({n : 0})
        data["Dijkstra"]["pathLengths"].update({n : 0})
        data["Dijkstra"]["expNodes"].update({n : 0})
        data["A*"]["pathLengths"].update({n : 0})
        data["A*"]["expNodes"].update({n : 0})
        for _ in range(M):
            S = (random.randint(1, 5), random.randint(1, 5)) ## S in [1, 5]x[1, 5]
            F = (random.randint(n - 6, n - 2), random.randint(n - 6, n - 2))
            maze = Maze(n, S, F)

            pl, en = getStatistics(maze, c = lambda x, y: 0, h = lambda x, y: manhattanDistance(x, y))
            data["Best-First"]["pathLengths"][n] += pl
            data["Best-First"]["expNodes"][n] += en

            pl, en = getStatistics(maze, c = lambda x, y: 1, h = lambda x, y: 0)
            data["Dijkstra"]["pathLengths"][n] += pl
            data["Dijkstra"]["expNodes"][n] += en

            pl, en = getStatistics(maze, c = lambda x, y: 1, h = lambda x, y: manhattanDistance(x, y))
            data["A*"]["pathLengths"][n] += pl
            data["A*"]["expNodes"][n] += en
        ## divisions to get average
        data["Best-First"]["pathLengths"][n] /= M
        data["Best-First"]["expNodes"][n] /= M
        data["Dijkstra"]["pathLengths"][n] /= M
        data["Dijkstra"]["expNodes"][n] /= M
        data["A*"]["pathLengths"][n] /= M
        data["A*"]["expNodes"][n] /= M

    return data
```

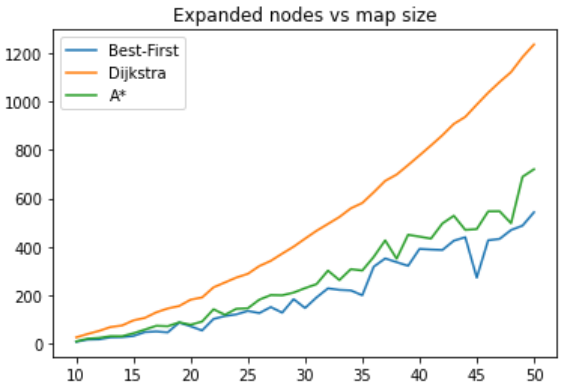
In [ ]:
```
data = generateData()
```

In [ ]:
```
plt.plot(*zip(*data["Best-First"]["pathLengths"].items()))
plt.plot(*zip(*data["Dijkstra"]["pathLengths"].items()))
plt.plot(*zip(*data["A*"]["pathLengths"].items()))
plt.legend(algorithms)
plt.title('Path length vs map size')
plt.show()
```



Ο Best-First δεν βρίσκει πάντα την βέλτιστη διαδρομή. Επειδή η ευρετική που έχουμε χρησιμοποιήσει πληροί την προϋπόθεση του A, βρίσκει πάντα την βέλτιστη διαδρομή και η καμπύλη του ταυτίζεται με αυτήν του Dijkstra.*

In [ ]:
```
plt.plot(*zip(*data["Best-First"]["expNodes"].items()))
plt.plot(*zip(*data["Dijkstra"]["expNodes"].items()))
plt.plot(*zip(*data["A*"]["expNodes"].items()))
plt.legend(algorithms)
plt.title('Expanded nodes vs map size')
plt.show()
```



Παρατηρούμε πως η καμπύλη για τον αλγόριθμο Dijkstra έχει μορφή εκθετικής, όπως ήταν αναμενόμενο, και πως ο Best-First επεκτείνει τους λιγότερους κόμβους, με αντίτιμο την μη-πληρότητα.

Μέρος 4

Σας δίνεται η συνάρτηση `draw(filename='filename.png')` η οποία παράγει ένα interactive grid ώστε να σχηματίσετε το λαβύρινθο. Θεωρήστε ως μαύρο το χρώμα του εμποδίου, πράσινο την αφετηρία και κόκκινο το στόχο, και πατήστε finish για να ολοκληρωθεί η εκτέλεση του τρέχοντος κελιού. Σε περίπτωση που θέλετε να παράξετε νέο λαβύρινθο, απλά ξανατρέξτε το κελί, επιλέξτε τα τετράγωνα και πατήστε finish. Κάθε εκτέλεση αποθηκεύει την εικόνα σε μορφή png σε όνομα που μπορείτε να καθορίσετε μέσω του filename (τοποθετώντας και την κατάληξη .png) κατά την κλήση της συνάρτησης.

Μπορείτε να το δοκιμάσετε τρέχοντας το επόμενο κελί κώδικα.

In [ ]:
```
from IPython.display import HTML, Image
from google.colab.output import eval_js
from base64 import b64decode
import PIL


canvas_html = """
<canvas width=301 height=301></canvas>
```

```
<br>

<button id = "start" style="background-color: #008000">Start</button>
<button id = "end" style="background-color: #800000">End</button>
<button id = "finish">Finish</button>
<script>

var canvas = document.querySelector('canvas')
var ctx = canvas.getContext('2d')

for (var x = 0.5; x < 301; x += 20) {
  ctx.moveTo(x, 0);
  ctx.lineTo(x, 300);
}

for (var y = 0.5; y < 301; y += 20) {
  ctx.moveTo(0, y);
  ctx.lineTo(300, y);
}

ctx.strokeStyle = "#000";
ctx.stroke();

var button = document.getElementById('finish')
var start_button = document.getElementById('start')
var end_button = document.getElementById('end')


start_button.onclick = ()=>{
    ctx.fillStyle="#008000";
  }

end_button.onclick = ()=>{
    ctx.fillStyle="#800000";
  }


var mouse = {x: 0, y: 0}
canvas.addEventListener('mousemove', function(e) {
  mouse.x = e.pageX - this.offsetLeft
  mouse.y = e.pageY - this.offsetTop
})

function getMousePos(canvas, evt) {
        var rect = canvas.getBoundingClientRect();
        return {
          x: evt.x - rect.left,
          y: evt.y - rect.top
        };
      }

function getNearestSquare(position) {
    var x = position.x;
    var y = position.y;

    if (x < 0 || y < 0) return null;
    x = (Math.floor(x / 20) * 20) + 0.5
    y = (Math.floor(y / 20) * 20) + 0.5
    return {x: x, y: y};
}

function containsObject(obj, list) {
    var i;
    for (i = 0; i < list.length; i++) {
        if (list[i].x === obj.x && list[i].y == obj.y) {
            return i;
        }
    }
    return -1;
}

borders = []

canvas.onmousedown = ()=>{
    var pos = getNearestSquare(getMousePos(canvas, mouse));
    if (pos != null) {
        index = containsObject(pos, borders)
        if (index != -1){
            borders.splice(index, 1)
            ctx.clearRect(pos.x + 1, pos.y, 19, 19)
```

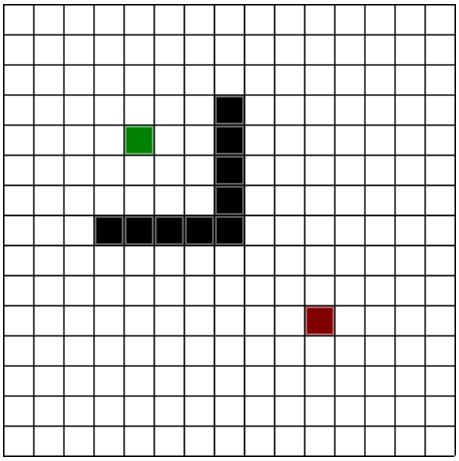
```
        }
        else{
            ctx.fillRect(pos.x+1,pos.y+1,18,18);
            borders.push(pos)
        }
        ctx.fillStyle="#000000";
    }
}
var data = new Promise(resolve=>{
  button.onclick = ()=>{
    resolve(canvas.toDataURL('image/png'))
  }
})
</script>
"""

def draw(filename='drawing.png'):
    display(HTML(canvas_html))
    data = eval_js("data")
    print (data)
    binary = b64decode(data.split(',')[1])
    with open(filename, 'wb') as f:
        f.write(binary)
    return len(binary)


draw()
```

Οι συναρτήσεις στο παρακάτω κελί κώδικα είναι βοηθητικές. Μετατρέπουν την εικόνα που αποθηκεύεται από την `draw` του προηγούμενου κελιου ξανά σε αντικείμενο τύπου Maze για να μπορέσετε να τρέξετε τους αλγορίθμους που έχετε υλοποιήσει καθώς και τον κώδικα visualization που σας δίνεται.

In [ ]:
```
def find_points(image_array, pixels):
    points = []
    for grid_i, i in enumerate(range (10, image_array.shape[0]-10, 20)):
        for grid_j, j in enumerate(range (10, image_array.shape[1]-10, 20)):
            if np.array_equal(image_array[i][j], pixels):
                points.append([grid_i+1, grid_j+1])
    return points

def load_maze(fname='/content/drawing.png'):
    N = 16
    image = PIL.Image.open(fname)
    image_array = np.round(np.array(image)/255)

    start_x, start_y = find_points(image_array, [0,1,0,1])[0]
    end_x, end_y = find_points(image_array, [1, 0, 0, 1])[0]
    walls = find_points(image_array, [0, 0, 0, 1])
    new_grid = Maze(N, (start_x, start_y), (end_x, end_y))

    for i in range(N):
        for j in range(N):
            if [i, j] in walls:
                new_grid.grid[i,j] = 1 ## ΑΛΛΑΓΗ ΛΟΓΩ ΠΡΟΑΝΑΦΕΡΘΕΙΣΑΣ ΠΑΡΑΔΟΧΗΣ
            else:
                new_grid.grid[i, j] = 0

    g = new_grid
    g.draw_maze()
    return g
```

Υπενθυμίζεται ότι το output των κελιών που σχηματίζετε διατηρείται για περιορισμένο χρονικό διάστημα στο notebook, γι αυτό καλείστε να αποθηκεύσετε τη φωτογραφία (το πραγματοποιεί η συνάρτηση draw()) και να τη μεταφέρετε στο ανάλογο markdown.

4.1.

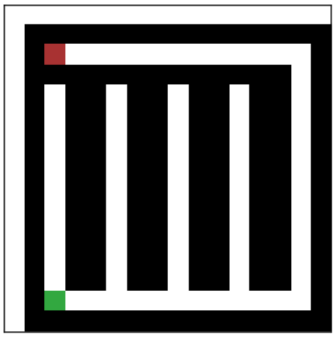


Επειδή ο Hill Climbing επιλέγει τις καταστάσεις - κελλιά με την χαμηλότερη τιμή ευρετικής και δεν διατηρεί τις υπόλοιπες στο μέτωπο αναζήτησης, κατευθυνόμενος προς αυτές με την χαμηλότερη απόσταση (Ευκλείδια ή Manhattan) θα φθάσει στο αδιέξοδο και θα τερματίσει χωρίς να βρει λύση.

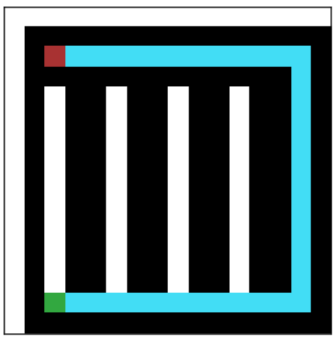
4.2.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')

g = load_maze('/content/drive/My Drive/Colab Notebooks/AI/4_2.png')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remoun
t=True).
```



In [ ]:
```
pf = Pathfinder(maze = g, c = lambda x, y: 1, h = lambda x, y: manhattanDistance(x, y))   # change arguments to choose t
g.draw_maze(pf.get_path())
```



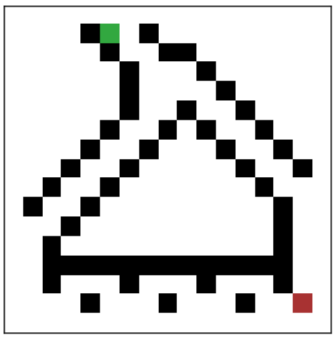
Σε αυτόν τον λαβύρινθο ο A αναγκάζεται να αναζητήσει όλα τα κελλιά πριν βρει την μοναδική σωστή διαδρομή, γιατί αυτή έχει τις μεγαλύτερες δυνατές τιμές f(n).*

4.3.

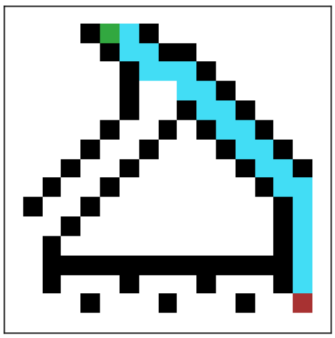
In [ ]:
```
def badHeuristic(A: (int, int), B: (int, int)) -> int:
    (Ax, Ay) = A
```

```
    (Bx, By) = B
    return abs(Ax - Bx)
```

In [ ]:
```
g = load_maze('4_3.png')
```

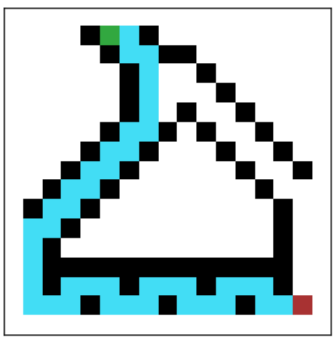


In [ ]:
```
%%time
pf = Pathfinder(maze = g, c = lambda x, y: 1, h = lambda x, y: badHeuristic(x, y))
# change arguments to choose the appropriate pathfinding algorithm and cost/heuristic (if applicable)
g.draw_maze(pf.get_path())
```



```
Wall time: 154 ms
```

In [ ]:
```
%%time
pf = Pathfinder(maze = g, c = lambda x, y: 0, h = lambda x, y: badHeuristic(x, y))
# change arguments to choose the appropriate pathfinding algorithm and cost/heuristic (if applicable)
g.draw_maze(pf.get_path())
```

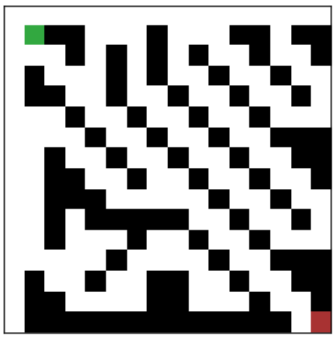


```
Wall time: 37 ms
```

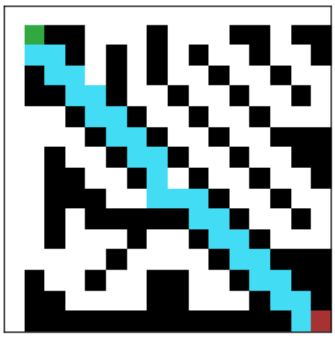
Στον παραπάνω λαβύρινθο φαίνεται ένα παράδειγμα όπου ο Best First δεν βρίσκει την βέλτιστη λύση, επειδή καθοδηγείται από ανεπαρκή ευρετική, ενώ ο A, επειδή λαμβάνει υπ' όψιν και την συνάρτηση κόστους, επιλέγει την σωστή κατεύθυνση. Εναλλακτικά, σε έναν λαβύρινθο όπου το βέλτιστο μονοπάτι απαιτεί προσωρινή μετακίνηση προς υψηλότερη ευρετική, ο Best-First δεν θα το επέλεγε και θα χάραζε διαδρομή μεγαλύτερου μήκους, ενώ ο A θα το εντόπιζε

4.4.

In [ ]:
```
g = load_maze('/content/drive/My Drive/Colab Notebooks/AI/4_4.png')
```

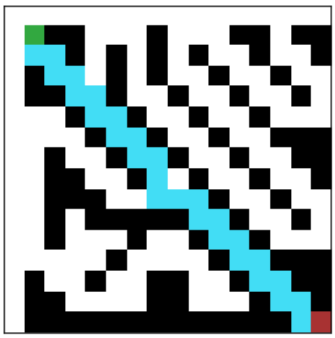


In [ ]:
```
%%time
pf = Pathfinder(maze = g, c = lambda x, y: 1, h = lambda x, y: manhattanDistance(x, y))
g.draw_maze(pf.get_path())
```



```
CPU times: user 46.6 ms, sys: 1.93 ms, total: 48.5 ms
Wall time: 51.2 ms
```

In [ ]:
```
%%time
pf = Pathfinder(maze = g, c = lambda x, y: 1, h = lambda x, y: 0)
g.draw_maze(pf.get_path())
```

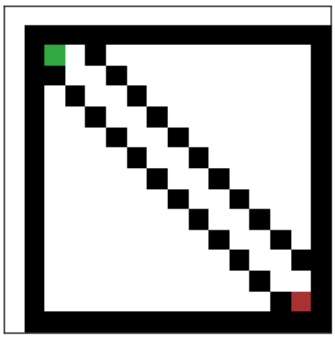


```
CPU times: user 80.8 ms, sys: 794 µs, total: 81.6 ms
Wall time: 88 ms
```

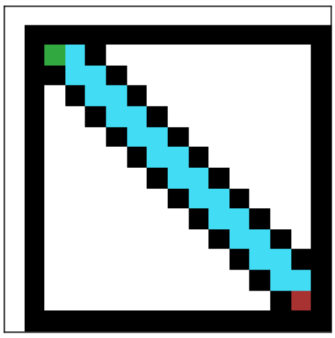
Παρατηρούμε σημαντική διαφορά στον χρόνο εύρεσης λύσης της τάξης του 60%, η οποία οφείλεται στο γεγονός πως η χρήση της ευρετικής καθοδηγεί τον A πρώτα προς την σωστή κατεύθυνση, ενώ ο Dijkstra εξερευνά όλες τις διακλαδώσεις.*

4.5.

In [ ]:
```
g = load_maze('/content/drive/My Drive/Colab Notebooks/AI/4_5.png')
```

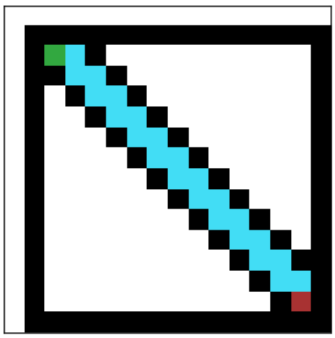


In [ ]:
```
%%time
pf = Pathfinder(maze = g, c = lambda x, y: 1, h = lambda x, y: manhattanDistance(x, y))
g.draw_maze(pf.get_path())
```



```
CPU times: user 48.7 ms, sys: 0 ns, total: 48.7 ms
Wall time: 51.2 ms
```

In [ ]:
```
%%time
pf = Pathfinder(maze = g, c = lambda x, y: 1, h = lambda x, y: 0)
g.draw_maze(pf.get_path())
```



```
CPU times: user 48.6 ms, sys: 3.05 ms, total: 51.7 ms
Wall time: 55.5 ms
```

Οι δύο αλγόριθμοι συμπεριφέρονται με τον ίδιο τρόπο γιατί από κάθε κατάσταση υπάρχει μοναδική μετάβαση.