Πανεπιστήμιο ΠειραιώςΤμήμα Πληροφορικής ΠΜΣ “Πληροφορική”
Ακαδημαϊκό έτος 2021-22(εαρινό εξάμηνο)

<<Εργασία στο μάθημα Τεχνητή Νοημοσύνη-Έμπειρα Συστήματα>>

Επιβλέποντας καθηγητές: Θέμης Παναγιωτόπουλος
Φοιτητής: Φώτιος Τσιούμας (ΜΠΠΛ21079)

A* Path Finding Algorithm

Αθήνα
Σεπτέμβριος 2022

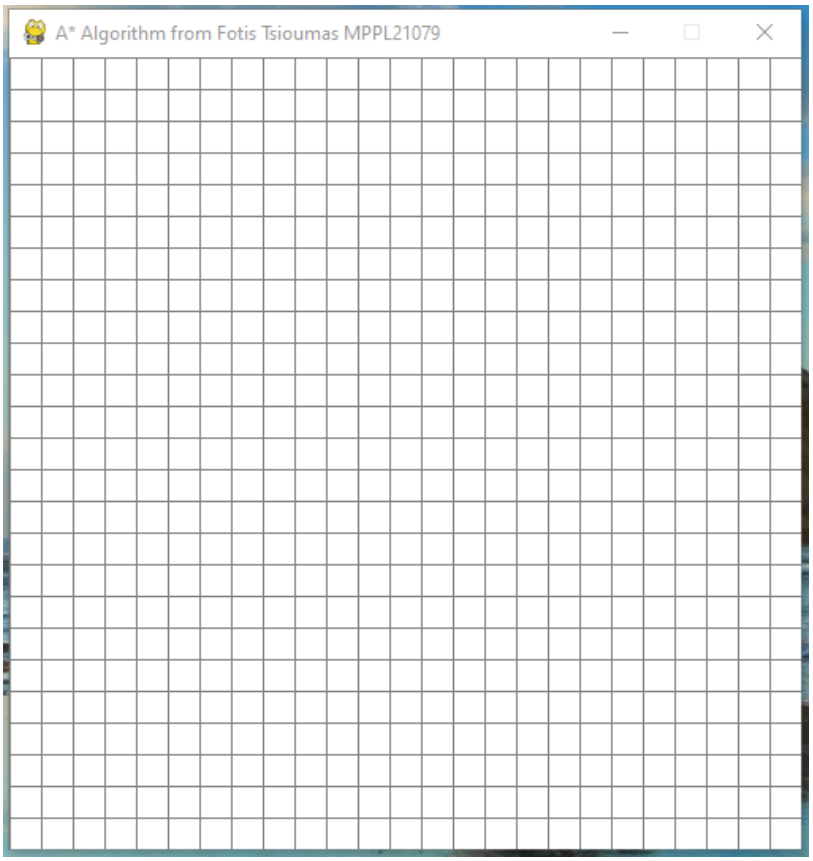
Περιεχόμενα

Παρουσίαση του παιχνιδιού

Το παιχνίδι παίζεται σε ένα grid 500x500 px και από άποψη nodes σε 25x25.

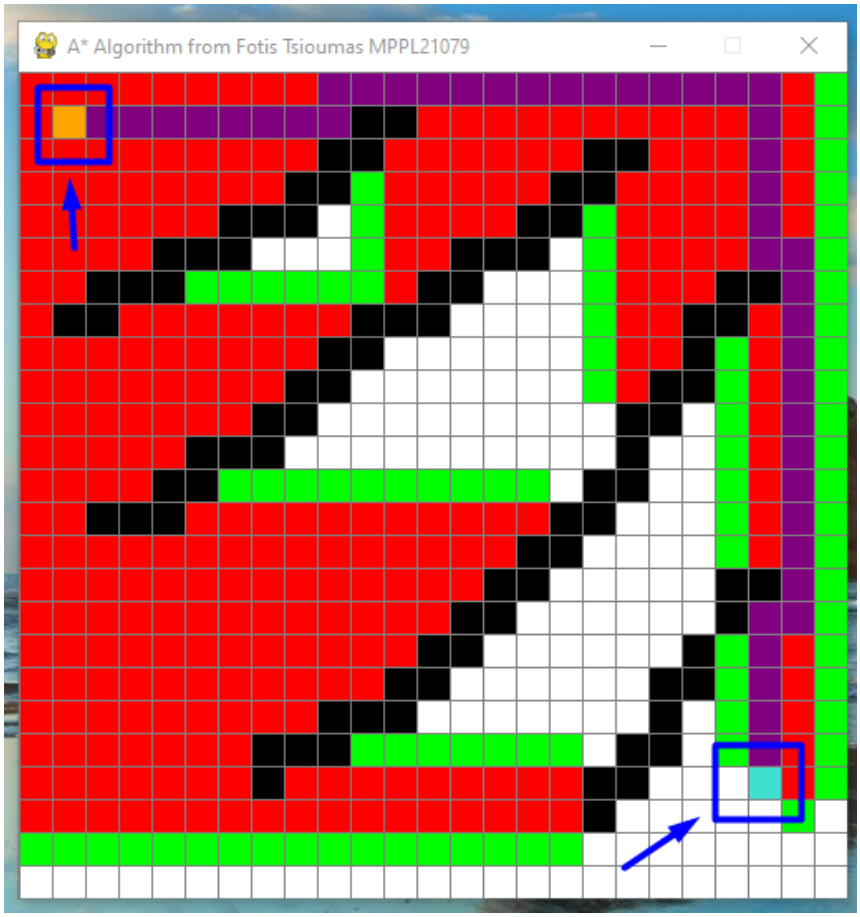
Πλήκτρα παιχνιδιού:

- Αριστερό κλικ με το ποντίκι: Ζωγραφίζεις.
- Δεξί κλικ με το ποντίκι: Σβήνεις.
- Πλήκτρο «C»: Επαναφέρεις το grid στην αρχική του κατάσταση.
- Πλήκτρο «Space»: Ξεκινάς το παιχνίδι αφού υπάρχει αρχικό και τελικό node.
- Κουμπί «X» παραθύρου: Τερματίζεις το παιχνίδι.



Χρώματα παιχνιδιού:

- Πορτοκαλί: Αρχή.
- Τιρκουάζ: Τέλος.
- Μαύρο: Εμπόδια.
- Κόκκινο: Nodes που έχουν εξεταστεί.
- Πράσινο: Νέοι γείτονες που μπορούν να εξεταστούν.
- Μοβ: Best path.

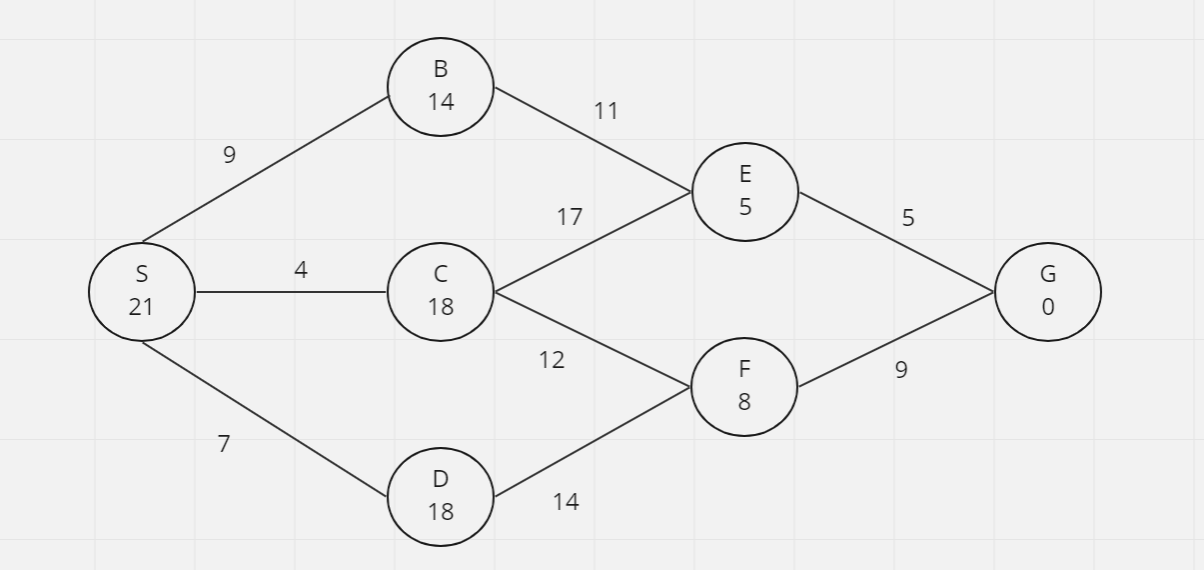


Επεξήγηση A* αλγορίθμου σε θεωρητικό επίπεδο

Ο αλγόριθμος A* ανήκει σε μια ευρύτερη κατηγορία αλγορίθμων που ονομάζονται best first, δηλαδή σε κάθε του κίνηση επιλέγει να πάει στον κόμβο που θεωρεί καλύτερο.

Ο A* είναι πλήρης αλγόριθμος, δηλαδή εγγυάται πάντα ότι θα βρει μονοπάτι από την αρχή στον στόχο. Εκτός από πλήρης όμως είναι και βέλτιστος, δηλαδή δεν θα βρει απλά μονοπάτι αλλά θα βρει το καλύτερο δυνατό μονοπάτι για να πας από την αρχή στον στόχο. Ένα όμως βασικό του μειονέκτημα είναι ότι η πολυπλοκότητα του είναι εκθετική.

Για να τον κατανοήσουμε καλύτερα, θα τον εξηγήσουμε μέσω του παρακάτω γραφήματος:



Ο στόχος μας είναι να πάμε από τον κόμβο S στον κόμβο G χρησιμοποιώντας τον αλγόριθμο A*. Οι αριθμοί που είναι πάνω στους δεσμούς μεταξύ των κόμβων είναι τα πραγματικά κόστη που θέλουμε για να πάμε από τον έναν κόμβο στον άλλο. Οι αριθμοί που είναι μέσα στους κόμβους είναι τα ευριστικά κόστη, τα οποία υπολογίζονται από μια ευριστική συνάρτηση και αντιπροσωπεύουν το κόστος που χρειαζόμαστε για να πάμε από έναν κόμβο στον στόχο.

Μια συνάρτηση που θα χρησιμοποιήσουμε κατά την διάρκεια της εξήγησης του αλγορίθμου είναι η παρακάτω και υποδηλώνει ότι το κόστος f ενός κόμβου n ισούται με το πραγματικό κόστος g του κόμβου n, συν το ευριστικό κόστος h του κόμβου n:

$$\underline{f(n) = g(n) + h(n)}$$

g(n): είναι το πραγματικό κόστος και αντιπροσωπεύει την απόσταση ενός κόμβου n από τον αρχικό κόμβο n και αποτελείται από το άθροισμα των αριθμών που βρίσκονται στους δεσμούς μεταξύ των κόμβων.

h(n): είναι το ευριστικό κόστος που προκύπτει από την ευριστική συνάρτηση h και δηλώνεται μέσα στο κύκλο των κόμβων.

Ο αλγόριθμος A* επιλέγει σε κάθε βήμα τον κόμβο που έχει το μικρότερο f κόστος.

Στο παράδειγμα μας θα χρησιμοποιηθούν δύο λίστες:

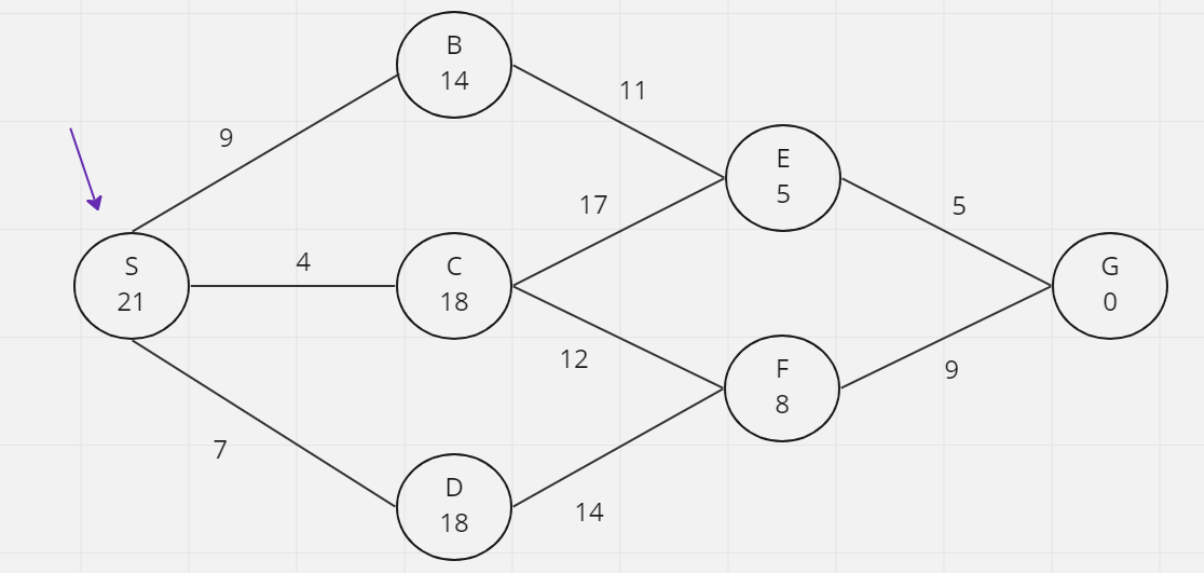
- **Ανοικτοί {γείτονας, f(γείτονα), ήρθε_από}**: Θα βρίσκονται όλοι οι κόμβοι που δεν έχουν ακόμα επισκεφτεί.
- **Κλειστοί**: Θα βρίσκονται οι κόμβοι που έχουν επισκεφτεί.

Σημείωση:

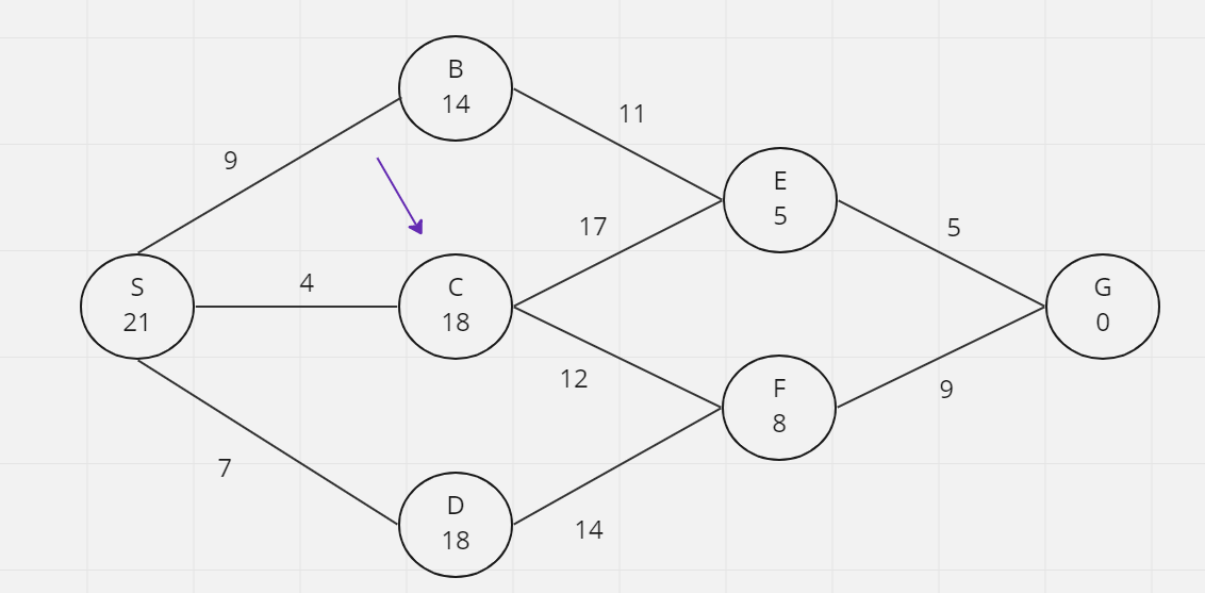
- **Έχω επισκεφτεί** έναν κόμβο σημαίνει έχω πατήσει πάνω του και έχω εξερευνήσει όλους τους γείτονες.
- **Γείτονες** είναι όλοι οι κόμβοι που συνδέονται απ' ευθείας με έναν κόμβο.
- Στην μεταβλητή **ήρθε_από** αποθηκεύονται οι γονείς των γειτόνων, δηλαδή οι κόμβοι από τους οποίους έχουν προέλθει οι γείτονες.
- Όταν επιλέγουμε να εξερευνήσουμε έναν κόμβο, ελέγχουμε και τις δύο λίστες αν υπάρχει ήδη. Αν δεν υπάρχει τον προσθέτουμε στην λίστα «**Ανοικτοί**». Αν υπάρχει βλέπουμε το "**f**" σκορ που είχε την προηγούμενη φορά που τον είχαμε εξερευνήσει και αν κρατάμε το μικρότερο.

Επίσης στο παράδειγμά μας θα χρησιμοποιήσουμε ένα **κόμβο current** που θα είναι ο κόμβος που εξερευνούμε κάθε φορά και θα τον **συμβολίζουμε με ένα βελάκι**.

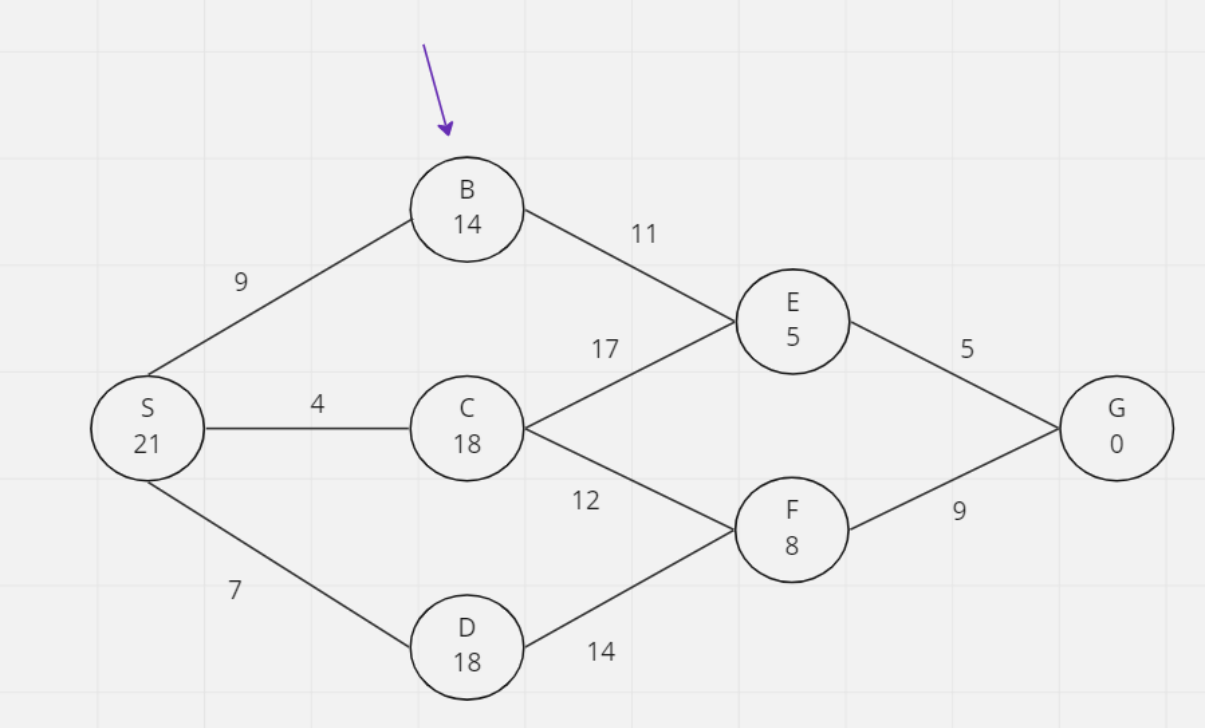
Ξεκινάμε να τρέχουμε τον αλγόριθμο:



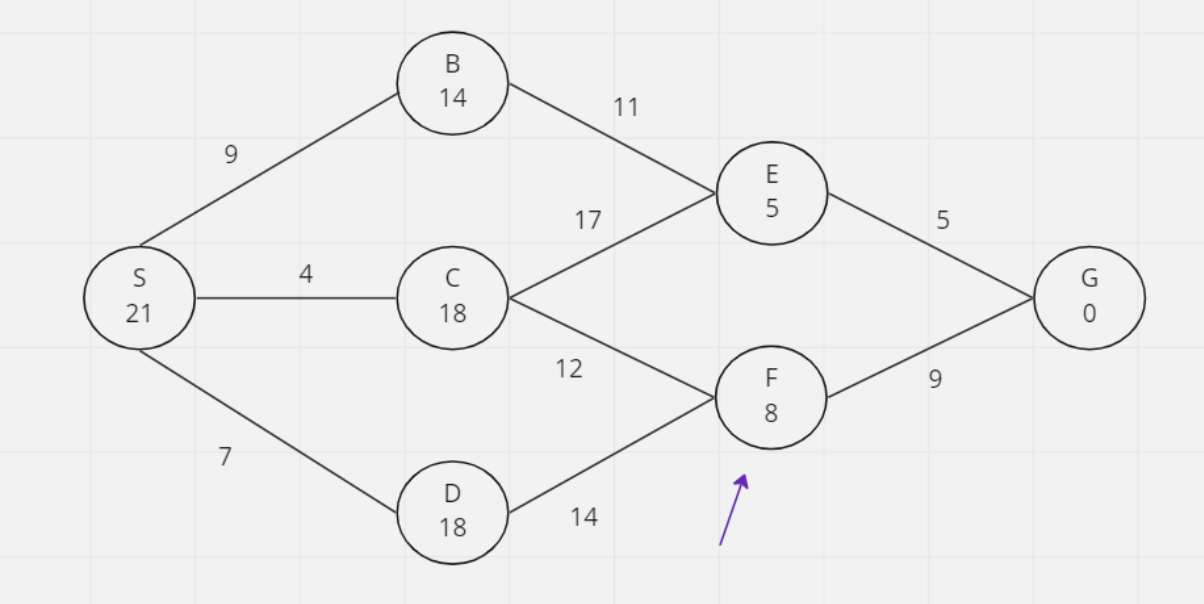
Ανοικτοί {γείτονας, f(γείτονα), ήρθε_από**}**	**Κλειστοί**
~~**{**S, 21, null**}**~~	**{**S, 21, null**}**
{B, 23, S**}**	
{C, 22, S**}**	
{D, 25, S**}**	



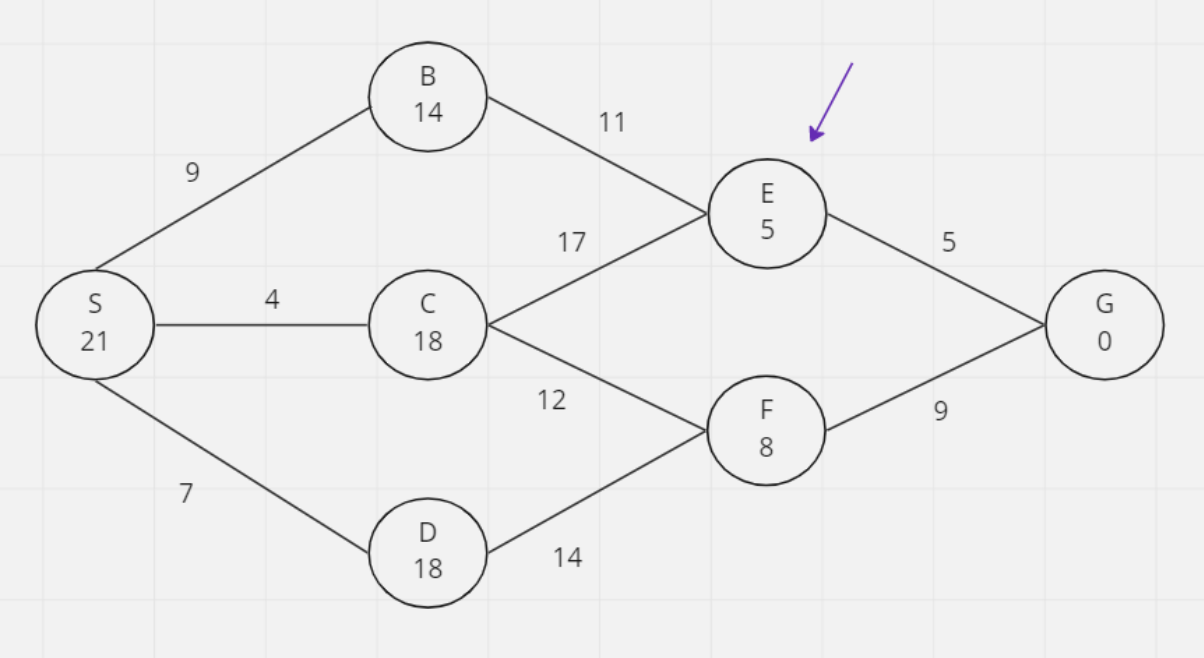
Ανοικτοί {γείτονας, f(γείτονα), ήρθε_από}	**Κλειστοί**
~~{S, 21, null}~~	{S, 21, null}
{B, 23, S}	{C, 22, S}
~~{C, 22, S}~~	
{D, 25, S}	
{E, 26, C}	
{F, 24, C}	



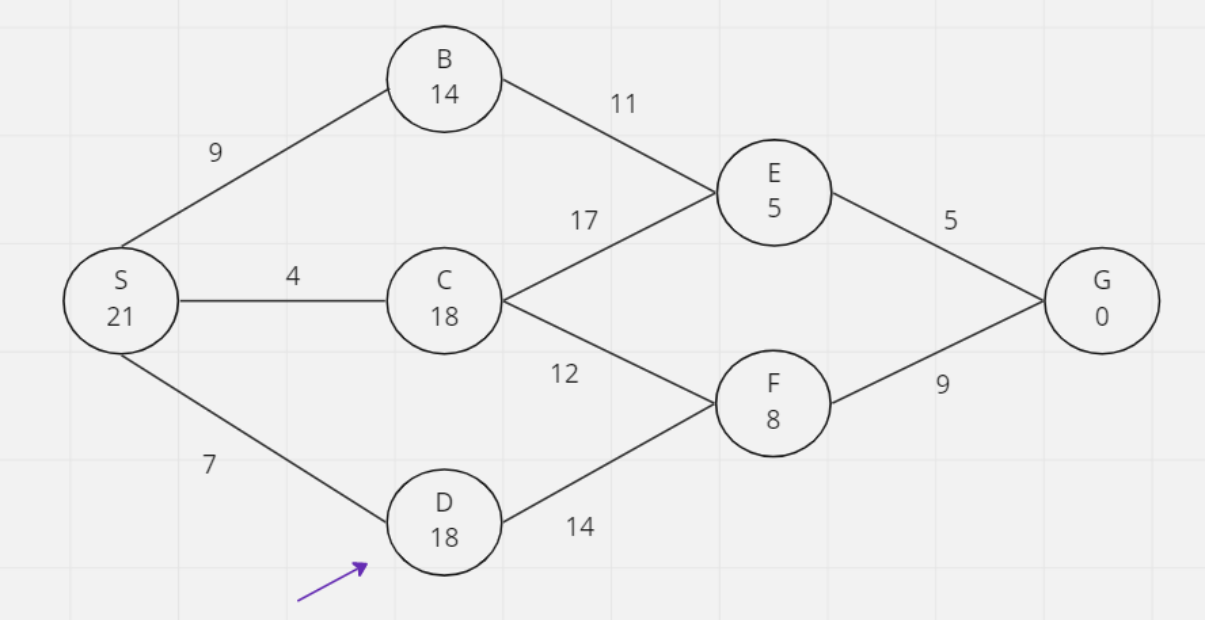
Ανοικτοί {γείτονας, f(γείτονα), ήρθε_από}	**Κλειστοί**
~~{S, 21, null}~~	{S, 21, null}
~~{B, 23, S}~~	{C, 22, S}
~~{C, 22, S}~~	{B, 23, S}
{D, 25, S}	
~~{E, 26, C}~~ {E, 25, B}	
{F, 24, C}	



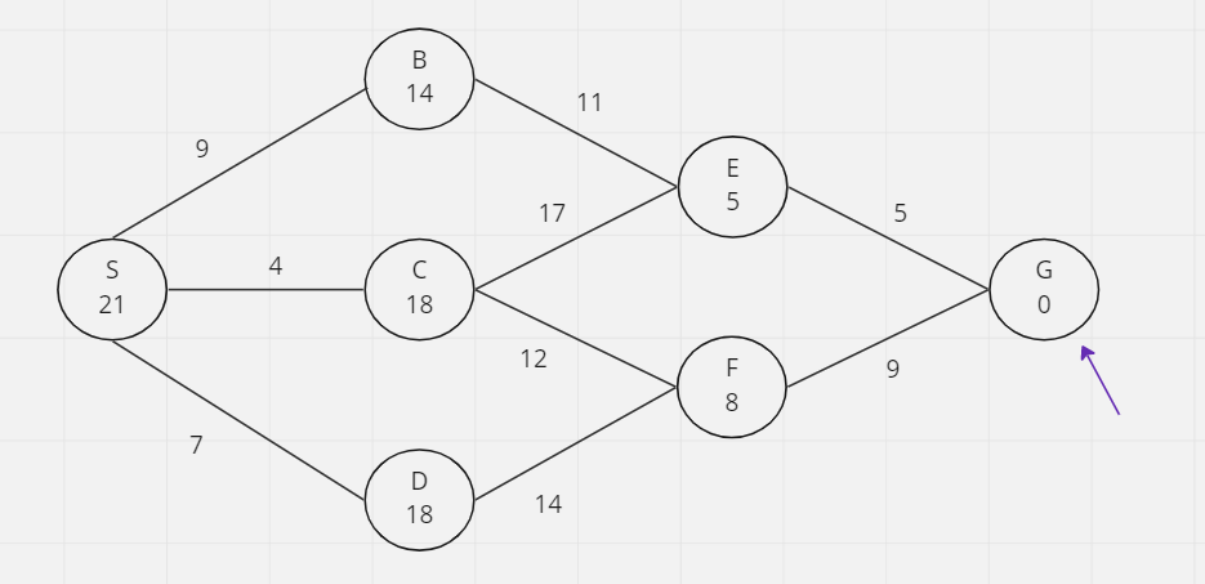
Ανοικτοί {γείτονας, f(γείτονα), ήρθε_από}	**Κλειστοί**
~~{S, 21, null}~~	{S, 21, null}
~~{B, 23, S}~~	{C, 22, S}
~~{C, 22, S}~~	{B, 23, S}
{D, 25, S}	{F, 24, C}
~~{E, 26, C}~~ {E, 25, B}	
~~{F, 24, C}~~	
{G, 25, F}	



Ανοικτοί {γείτονας, f(γείτονα), ήρθε_από}	**Κλειστοί**
~~{S, 21, null}~~	{S, 21, null}
~~{B, 23, S}~~	{C, 22, S}
~~{C, 22, S}~~	{B, 23, S}
{D, 25, S}	{F, 24, C}
~~{E, 26, C}~~ ~~{E, 25, B}~~	{E, 25, B}
~~{F, 24, C}~~	
{G, 25, F}	



Ανοικτοί {γείτονας, f(γείτονα), ήρθε_από}	**Κλειστοί**
~~{S, 21, null}~~	{S, 21, null}
~~{B, 23, S}~~	{C, 22, S}
~~{C, 22, S}~~	{B, 23, S}
~~{D, 25, S}~~	{F, 24, C}
~~{E, 26, C}~~ ~~{E, 25, B}~~	{E, 25, B}
~~{F, 24, C}~~	{D, 25, S}
{G, 25, F}	



Ανοικτοί {γείτονας, f(γείτονα), ήρθε_από}	**Κλειστοί**
~~{S, 21, null}~~	{S, 21, null}
~~{B, 23, S}~~	{C, 22, S}
~~{C, 22, S}~~	{B, 23, S}
~~{D, 25, S}~~	{F, 24, C}
~~{E, 26, C}~~ ~~{E, 25, B}~~	{E, 25, B}
~~{F, 24, C}~~	{D, 25, S}
{G, 25, F}	

Σε αυτή την φάση ο αλγόριθμος τερματίζει καθώς βρήκαμε τον στόχο που είναι ο G κόμβος. Για να βρούμε το best path ξεκινάμε από το τέλος και πηγαίνουμε προς την αρχή ακολουθώντας το «**ήρθε_από**». Μετά αντιστρέφουμε το path για να έχουμε πρώτο τον αρχικό κόμβο και τελευταίο τον τελικό κόμβο και έτσι βρίσκουμε τον best path, που είναι το παρακάτω:

S -> C -> F -> G

Υλοποίηση A* σε Python – Επεξήγηση κώδικα

Ακολουθούν screenshots με τον κώδικα και comments για την επεξήγηση του:

```python
# Import τα libraries που θα χρησιμοποιήσουμε
import pygame
from queue import PriorityQueue

# Ορισμός σταθερών
WIDTH = 500
WIN = pygame.display.set_mode((WIDTH, WIDTH))
pygame.display.set_caption("A* Algorithm from Fotis Tsioumas MPPL21079")

# Ορισμός χρωμάτων
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 255, 0)
YELLOW = (255, 255, 0)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
PURPLE = (128, 0, 128)
ORANGE = (255, 165 ,0)
GREY = (128, 128, 128)
TURQUOISE = (64, 224, 208)
```

```python

# Κλάση spot του grid
class Spot:
    # Constarctor κλάσης
    def __init__(self, row, col, width, total_rows):
        self.col: Any
        self.col = col
        self.x = row * width
        self.y = col * width
        self.color = WHITE
        self.neighbors = []
        self.width = width
        self.total_rows = total_rows

    # Μέθοδος που επιστρέφει το position
    def get_pos(self):
        return self.row, self.col

```

```
    # Μέθοδοι χρωματισμού
    def is_closed(self):
        return self.color == RED
    def is_open(self):
        return self.color == GREEN
    def is_barrier(self):
        return self.color == BLACK
    def is_start(self):
        return self.color == ORANGE
    def is_end(self):
        return self.color == TURQUOISE
    def reset(self):
        self.color = WHITE
    def make_start(self):
        self.color = ORANGE
    def make_closed(self):
        self.color = RED
    def make_open(self):
        self.color = GREEN
    def make_barrier(self):
        self.color = BLACK
    def make_end(self):
        self.color = TURQUOISE
    def make_path(self):
        self.color = PURPLE

```

```
    # Μέθοδος ζωγραφίσματος spot
    def draw(self, win):
        pygame.draw.rect(win, self.color, (self.x, self.y, self.width, self.width))

    # Μέθοδος προσθήκης γειτώνων
    def update_neighbors(self, grid):
        self.neighbors = []
        # Down movement
        # Αν το row είναι μικρότερο από το τελευταίο row και το επόμενο row δεν είναι barrier τοτε μπορεί το επόμενο row να προστεθεί στους γείτονες
        if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_barrier():
            self.neighbors.append(grid[self.row + 1][self.col])
        # Up movement
        # Αν το row είναι μεγαλύτερο απο το πρώτο row, δηλαδή το 0 και το προηγούμενο row δεν είναι barrier τοτε μπορεί το προηγούμενο row να προστεθεί στους γείτονες
        if self.row > 0 and not grid[self.row - 1][self.col].is_barrier():
            self.neighbors.append(grid[self.row - 1][self.col])
        # Right movement
        # Αν το column είναι μικρότερο από το τελευταίο column και το επόμενο column δεν είναι barrier τοτε μπορεί το επόμενο column να προστεθεί στους γείτονες
        if self.col < self.total_rows - 1 and not grid[self.row][self.col + 1].is_barrier():
            self.neighbors.append(grid[self.row][self.col + 1])
        # Left movement
        # Αν το column είναι μεγαλύτερο απο το πρώτο column, δηλαδή το 0 και το προηγούμενο column δεν είναι barrier τοτε μπορεί το προηγούμενο column να προστεθεί στους γείτονες
        if self.col > 0 and not grid[self.row][self.col - 1].is_barrier():
            self.neighbors.append(grid[self.row][self.col - 1])

    # Χρήση lt magic method για να μπορέσω να συγκρίνω δύο object της κλάσης Spot μεταξύ τους
    def __lt__(self, other):
        return False

```

```python
# "h" function (ευριστική συνάρτηση)
def h(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(x1 - x2) + abs(y1 - y2)

# Σχηματίζει το best path όταν αυτό βρεθεί
def reconstruct_path(came_from, current, draw):
    while current in came_from:
        current = came_from[current]
        current.make_path()
        draw()

# algorithm function
def algorithm(draw, grid, start, end):
    count = 0

    # Ουρά προτεραιότητας (πρώτα η χαμηλότερη). Επιστρέφει τη μικρότερη τιμή στη λίστα
    open_set = PriorityQueue()

    # Προσθήκη spot στην ουρά
    open_set.put((0, count, start))

    # Dictionary βέλτιστου μονοπατιού
    came_from = {}

    # Υπολογισμός "g" score
    # Το "g" score  του start spot είναι 0
    # float("inf") = Λειτουργεί ως μια απεριόριστη ανώτερη τιμή για σύγκριση. Αυτό είναι χρήσιμο για την εύρεση χαμηλότερων τιμών, κατά τον υπολογισμός του κόστους διαδρομής
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0

    # Υπολογισμός "f" score
    # Το "f" score  του start spot είναι η τιμή της "h" function
    f_score = {spot: float("inf") for row in grid for spot in row}
    f_score[start] = h(start.get_pos(), end.get_pos())

    # Το open_set_hash βάζει μια τιμή στην ουρά προτεραιότητας. Το κάνουμε αυτό για να μπορέσουμε να εκτελέσουμε τον βρόχο while στην επόμενη γραμμή και επίσης μπορούμε να χρησιμοποιήσουμε αυτήν την ουρά προτεραιότητας για να κάνουμε πίσω στο σημείο που έχουμε ταξιδέψει.
    open_set_hash = {start}

    # Όσο η ουρά προτεραιότητας δεν είναι άδεια, εκτελείται ο βρόχος while
    while not open_set.empty():

        # Εξασφαλίζει ότι αν κλείσει το παράθυρο θα τερματίσει το παιχνίδι
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

        # Χρησιμοποιούμε το "current" για να τραβήξουμε την καλύτερη τιμή της ουράς προτεραιότητας και στη συνέχεια την αφαιρούμε από την λίστα "open_set_hash" καθώς πρόκειται να την αντικαταστήσουμε με το επόμενο node
        current = open_set.get()[2]
        open_set_hash.remove(current)

        # Ελέγχουμε πρώτα αν ο "cuurent" κόμβος είναι ο "end" κόμβος. Αν είναι αυτός, έχουμε βρει το best path και τρέχουμε την reconstruct_path() για να σχηματίσει με μοβ χρώμα το best path. Μετα χρωματίζουμε ξανα τον "start" και "end" κόμβο και τερματίζουμε με true τον βρόχο while.
        if current == end:
            reconstruct_path(came_from, end, draw)
            start.make_start()
            end.make_end()
            return True

        # Εαν δεν ισχύει η παραπάνω συνθήκη αρχίζουμε να εξετάζουμε τους γείτονες του "current" κόμβου. Πρώτα θέλουμε να παρακολουθήσουμε σε ποιον κόμβο βρισκόμαστε και το κάνουμε αυτό αυξάνοντας το "g_score" του κόμβου που ήταν πριν. Στη συνέχεια, ελέγχουμε αν το "g_score" του ελιναι καλύτερο απο του προηγούμενου. Αν είναι καλύτερο τότε θα τον κάνουμε τον επόμενο κόμβο προς χρήση. Υπολογίζουμε το "f_score" και "το g_score" και αν δεν είναι μεσα στη λίστα προτεραιότητας τον προσθέτουμε. Στη συνέχεια κάνουμε τους γείτονες με πράσινο, που σημαίνει ότι θα ελεγχθούν στην επόμενη επανάληψη.
        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1
            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((f_score[neighbor], count, neighbor))
                    open_set_hash.add(neighbor)
                    neighbor.make_open()

        # Χρωματίζουμε ξανά το grid
        draw()

        # Ορίζουμε τα όρια των τιμών που έχουμε ελέγξει με κόκκινο καθώς θέλουμε να διατηρήσουμε το χρώμα του αρχικού κόμβου διακριτό.
        if current != start:
            current.make_closed()

    # Εαν η ουρά προτεραιότητας είναι κενή δεν τρέχει ποτέ το while loop και επιστρέφει ο αλγόριθμος false καθώς δεν εκτελέστηκε.
    return False
```

```
# Κατασκευή grid
# gap = λευκο κουτι ανάμεσα απο γκρι γραμμαμές
def make_grid(rows, width):
    grid = []
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            spot = Spot(i, j, gap, rows)
            grid[i].append(spot)
    return grid

# Χρωματισμός γκρι γραμμών grid
def draw_grid(win, rows, width):
    gap = width // rows
    for i in range(rows):
        pygame.draw.line(win, GREY, (0, i * gap), (width, i * gap))
        for j in range(rows):
            pygame.draw.line(win, GREY, (j * gap, 0), (j * gap, width))

# Χρωματισμός όλων των spot του grid
def draw(win, grid, rows, width):
    win.fill(WHITE)
    for row in grid:
        for spot in row:
            spot.draw(win)
    draw_grid(win, rows, width)
    pygame.display.update()

```

```
# Επιστρέφει το τετράγωνο που πάτησα κλικ
def get_clicked_pos(pos, rows, width):
    gap = width // rows
    y, x = pos
    row = y // gap
    col = x // gap
    return row, col


# main function
def main(win, width):

    # Ορισμός rows και σχηματισμός πλέγματος
    ROWS = 25
    grid = make_grid(ROWS, width)

    # Ορισμός μεταβλητών start, end, run
    start = None
    end = None
    run = True

```

```
224
225     # Loop δημιουργίας grid και έναρξης παιχνιδιού
226     while run:
227
228         # Ζωγράφισμα grid
229         draw(win, grid, ROWS, width)
230
231         # Loop για οποιοδήποτε pygame.event συμβει
232         for event in pygame.event.get():
233
234             # Εξασφαλίζει ότι αν κλείσει το παράθυρο θα τερματίσει το παιχνίδι
235             if event.type == pygame.QUIT:
236                 run = False
237
238             # Actions με left mouse click (γράψιμο)
239             if pygame.mouse.get_pressed()[0]:
240                 pos = pygame.mouse.get_pos()
241                 row, col = get_clicked_pos(pos, ROWS, width)
242                 spot = grid[row][col]
243                 # Χρωματισμός start spot (δεν μπορεί να είναι το end spot)
244                 if not start and spot != end:
245                     start = spot
246                     start.make_start()
247                 # Χρωματισμός end spot (δεν μπορεί να είναι το start spot)
248                 elif not end and spot != start:
249                     end = spot
250                     end.make_end()
251                 # Χρωματισμός barrier spot (δεν μπορεί να είναι ούτε το start spot ούτε το end spot)
252                 elif spot != end and spot != start:
253                     spot.make_barrier()
254
```

```
254
255             # Actions με right mouse click (σβήσιμο)
256             elif pygame.mouse.get_pressed()[2]:
257                 # Βρίσκω το spot που έγινε το κλικ και το κάνω reset, δηλαδή λευκό
258                 pos = pygame.mouse.get_pos()
259                 row, col = get_clicked_pos(pos, ROWS, width)
260                 spot = grid[row][col]
261                 spot.reset()
262                 # Αν είναι start ή end spot τότε επιπλέον κάνω και τις μεταβλητες start και stop, None
263                 if spot == start:
264                     start = None
265                 elif spot == end:
266                     end = None
267
268             # Actions με keyboard
269             if event.type == pygame.KEYDOWN:
270                 # Με button "space" και αν υπάρχει ορισμένο start και end spot, ξεκινάει η εύρεση γειτώνων και ο έναρξη του αλγορίθμου
271                 if event.key == pygame.K_SPACE and start and end:
272                     # Εύρεση γειτώνω
273                     for row in grid:
274                         for spot in row:
275                             spot.update_neighbors(grid)
276                     # Έναρξη αλγορίθμου
277                     algorithm(lambda: draw(win, grid, ROWS, width), grid, start, end)
278                 # Με button "C" επαναφέρω το grid στην αρχική του μορφή
279                 if event.key == pygame.K_c:
280                     start = None
281                     end = None
282                     grid = make_grid(ROWS, width)
283
284     # Τερματισμός main function
285     pygame.quit()
286
287 # call main function
288 main(WIN, WIDTH)
```