HACETTEPE UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING BBM405

Name : Mehmet Taha

Surname : USTA

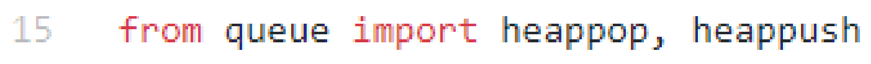
Number : 21527472

Subject : Compare the Search Algorithms

Data Due: 17.05.2020 23.59

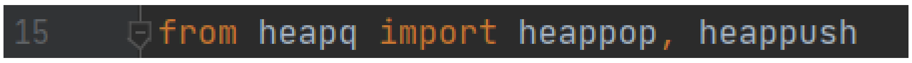
1. İmplementation and small changes

I used the https://github.com/chitholian/AI-Search-Algorithms site for search algorithms. I made some minor changes for some of the algorithms to work. I changed the queue in line 15 to heapq.



```
15    from queue import heappop, heappush
```

(before)



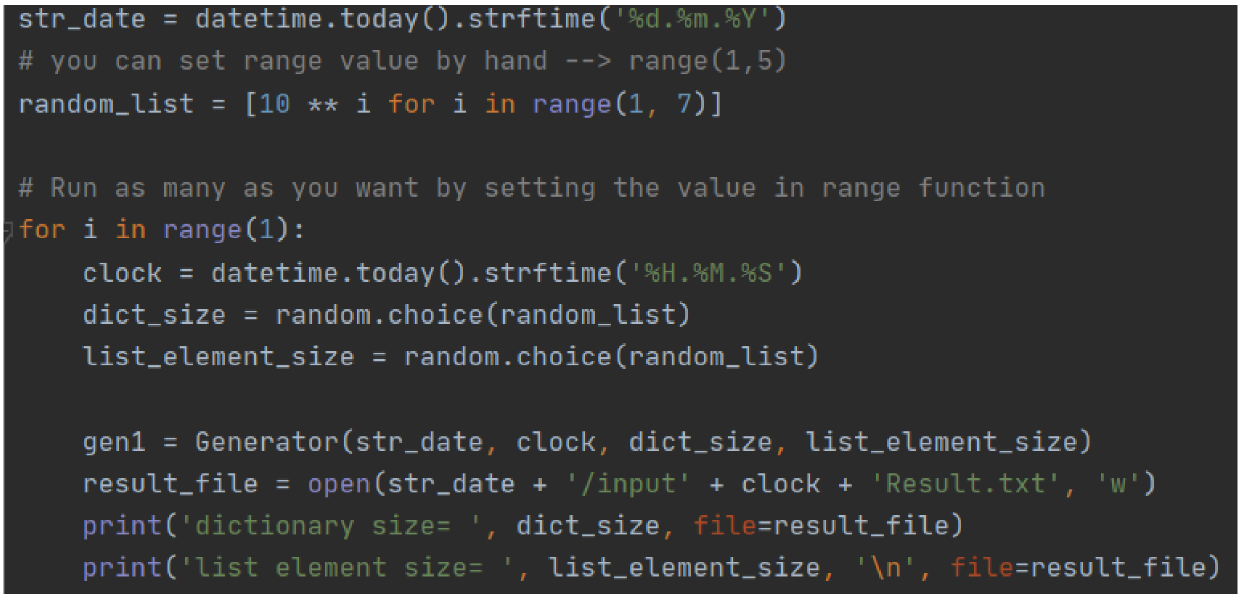
```
15    from heapq import heappop, heappush
```

(after)

2. Main.py and Generator.py

The reason I have created these two python files is to compare the results by trying the inputs that I generated via Generator.py in the search algorithms I created in Main.py.

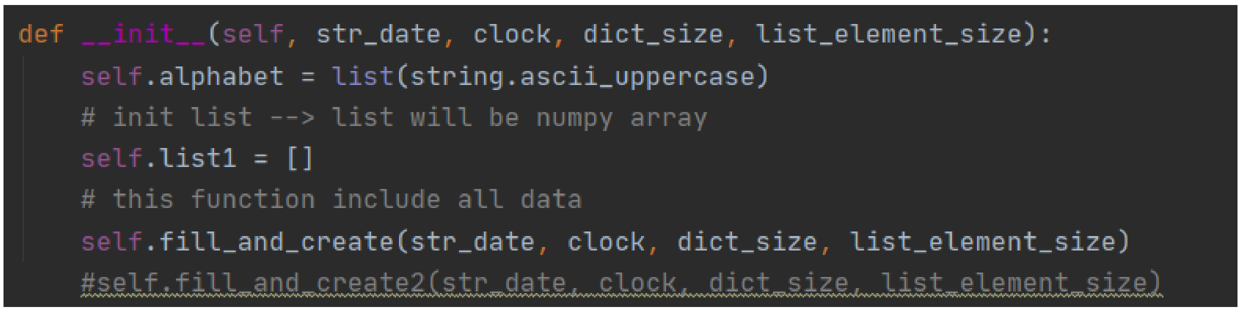
First, the date of the str_date variable is kept to create the file path. The random_list variable is defined to determine the width of the path costs.



```
str_date = datetime.today().strftime('%d.%m.%Y')
# you can set range value by hand --> range(1,5)
random_list = [10 ** i for i in range(1, 7)]

# Run as many as you want by setting the value in range function
for i in range(1):
    clock = datetime.today().strftime('%H.%M.%S')
    dict_size = random.choice(random_list)
    list_element_size = random.choice(random_list)

    gen1 = Generator(str_date, clock, dict_size, list_element_size)
    result_file = open(str_date + '/input' + clock + 'Result.txt', 'w')
    print('dictionary size= ', dict_size, file=result_file)
    print('list element size= ', list_element_size, '\n', file=result_file)
```

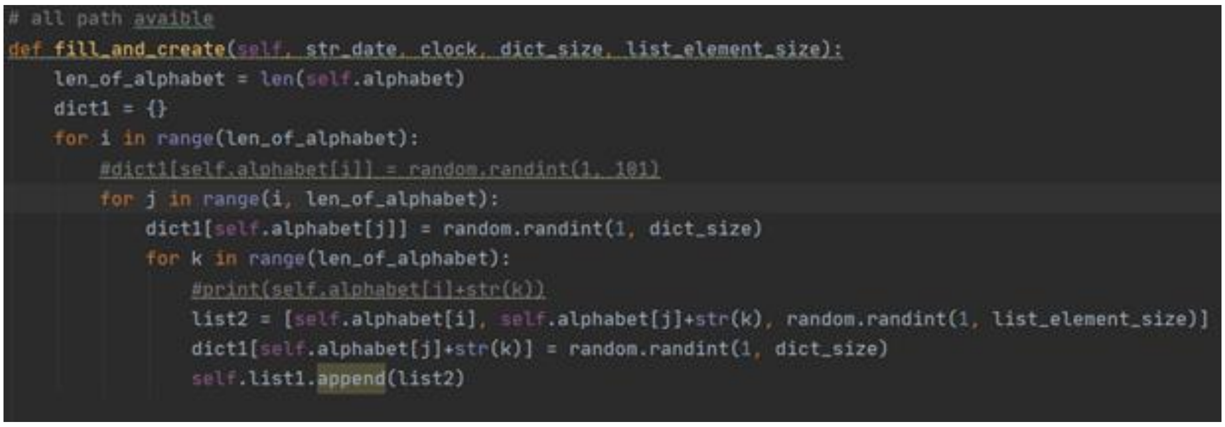
The Clock variable keeps the current time. The dict_size variable determines the width of the heuristic values in the A * Search algorithm and is randomly selected from random_list.

The list_element_size variable determines the costs between the edges and is randomly selected from random_list. After the widths are determined, the gen1 object is created and the constructer and input edges are created and stored in the list1 variable.

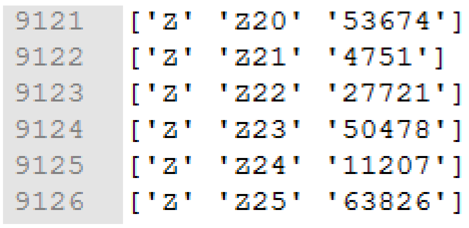


```python
def __init__(self, str_date, clock, dict_size, list_element_size):
    self.alphabet = list(string.ascii_uppercase)
    # init list --> list will be numpy array
    self.list1 = []
    # this function include all data
    self.fill_and_create(str_date, clock, dict_size, list_element_size)
    #self.fill_and_create2(str_date, clock, dict_size, list_element_size)
```

Defined to identify names in self.alphabet edges. The string library for the alphabet was imported. The fill_and_create () function in the Constructer generates edges and stores them in list1. The fill_and_create2 () function in the Constructer produces edges, but it depends on the probability that each edge is stored in list1.

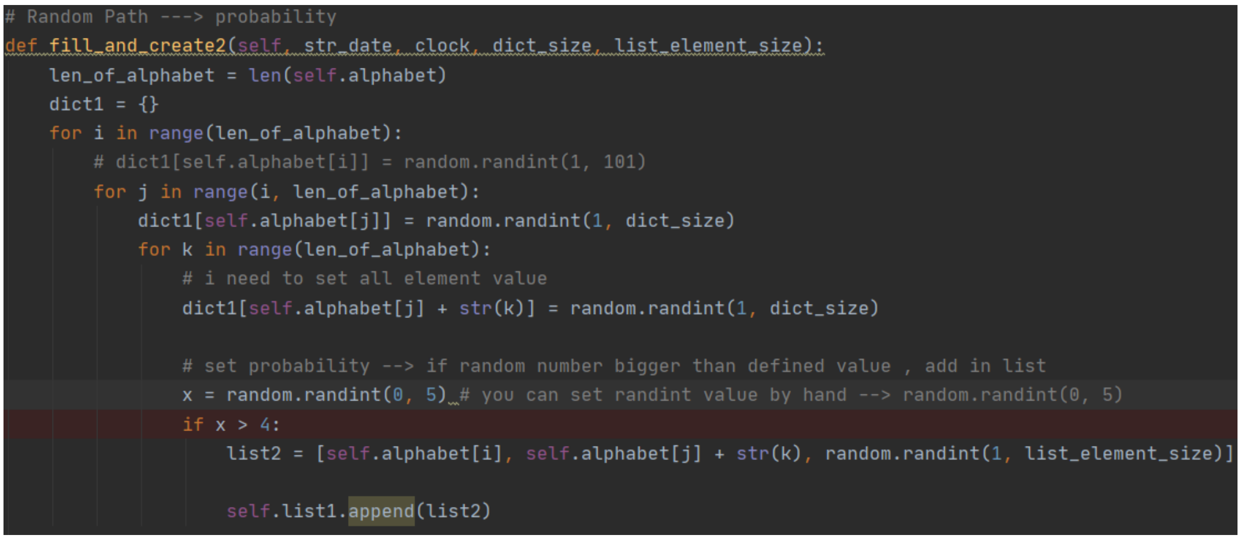


```python
# all path avaible
def fill_and_create(self, str_date, clock, dict_size, list_element_size):
    len_of_alphabet = len(self.alphabet)
    dict1 = {}
    for i in range(len_of_alphabet):
        #dict1[self.alphabet[i]] = random.randint(1, 101)
        for j in range(i, len_of_alphabet):
            dict1[self.alphabet[j]] = random.randint(1, dict_size)
            for k in range(len_of_alphabet):
                #print(self.alphabet[i]+str(k))
                list2 = [self.alphabet[i], self.alphabet[j]+str(k), random.randint(1, list_element_size)]
                dict1[self.alphabet[j]+str(k)] = random.randint(1, dict_size)
                self.list1.append(list2)
```



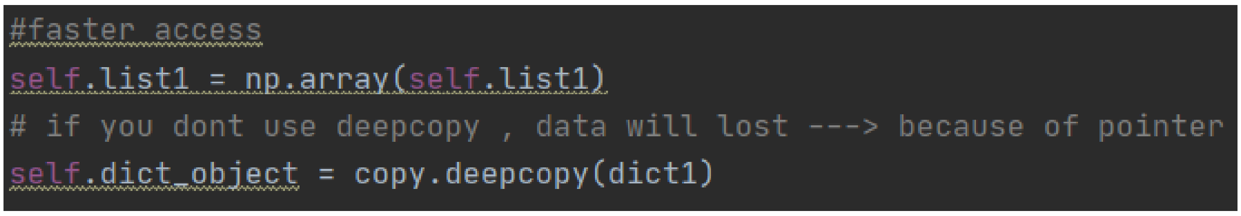
```
9121  ['Z'  'Z20'  '53674']
9122  ['Z'  'Z21'  '4751']
9123  ['Z'  'Z22'  '27721']
9124  ['Z'  'Z23'  '50478']
9125  ['Z'  'Z24'  '11207']
9126  ['Z'  'Z25'  '63826']
```

The fill_and_create () function produces 9126 edges.

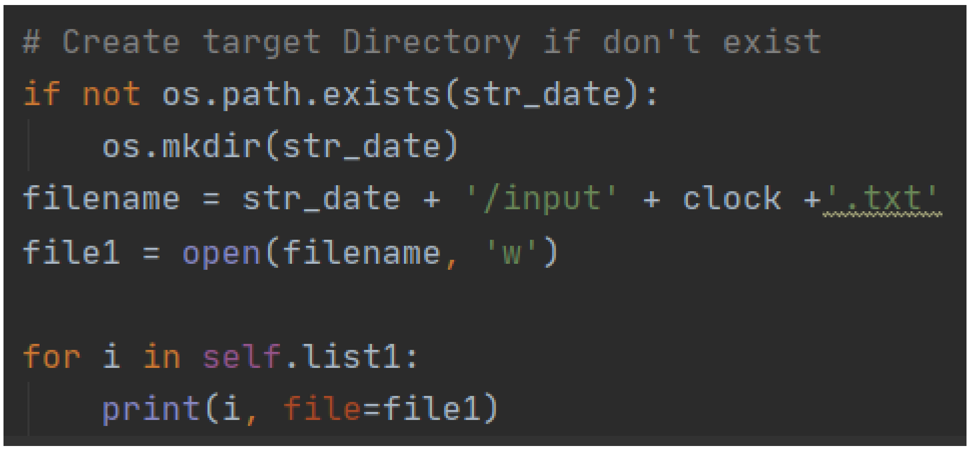


```python
# Random Path ---> probability
def fill_and_create2(self, str_date, clock, dict_size, list_element_size):
    len_of_alphabet = len(self.alphabet)
    dict1 = {}
    for i in range(len_of_alphabet):
        # dict1[self.alphabet[i]] = random.randint(1, 101)
        for j in range(i, len_of_alphabet):
            dict1[self.alphabet[j]] = random.randint(1, dict_size)
            for k in range(len_of_alphabet):
                # i need to set all element value
                dict1[self.alphabet[j] + str(k)] = random.randint(1, dict_size)

                # set probability --> if random number bigger than defined value , add in list
                x = random.randint(0, 5) # you can set randint value by hand --> random.randint(0, 5)
                if x > 4:
                    list2 = [self.alphabet[i], self.alphabet[j] + str(k), random.randint(1, list_element_size)]

                    self.list1.append(list2)
```

The function fill_and_create2 () can produce up to 9126 edges.



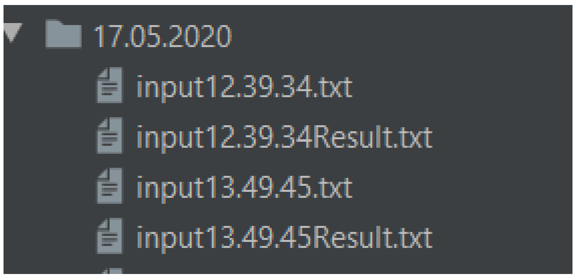
```python
#faster access
self.list1 = np.array(self.list1)
# if you dont use deepcopy , data will lost ---> because of pointer
self.dict_object = copy.deepcopy(dict1)
```



```python
# Create target Directory if don't exist
if not os.path.exists(str_date):
    os.mkdir(str_date)
filename = str_date + '/input' + clock +'.txt'
file1 = open(filename, 'w')

for i in self.list1:
    print(i, file=file1)
```

Since the nodes are created randomly, they are stored in the file with the .txt extension on the computer so that they do not disappear when the process is finished.

After the gen1 object is created, the file is created to store all search results. The results are recorded in this.

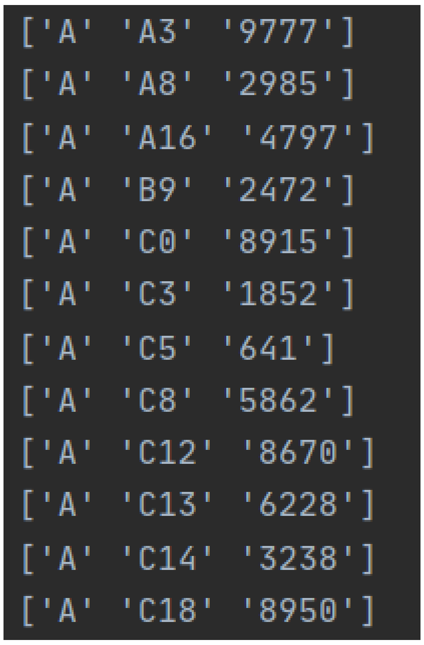


```
17.05.2020
    input12.39.34.txt
    input12.39.34Result.txt
    input13.49.45.txt
    input13.49.45Result.txt
```

17.05.2020 → klasör → It is created by Generator.py.

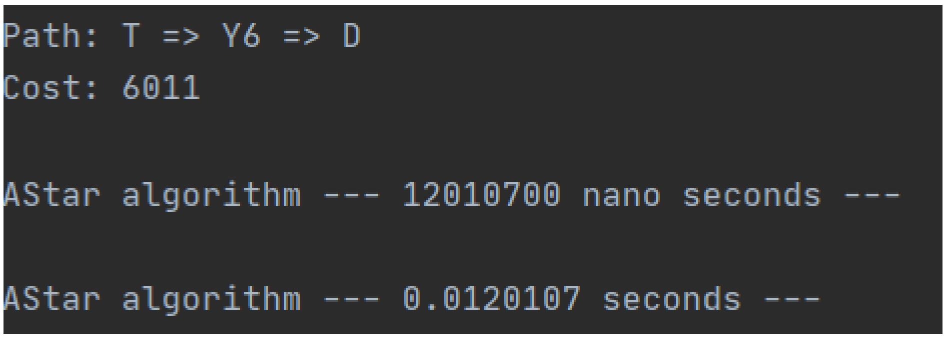
Input12.39.34.txt → The edges in the path are stored here.

Input12.39.34Result.txt → The path and results found are stored.



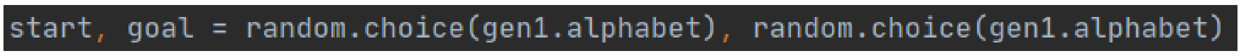
```
['A' 'A3' '9777']
['A' 'A8' '2985']
['A' 'A16' '4797']
['A' 'B9' '2472']
['A' 'C0' '8915']
['A' 'C3' '1852']
['A' 'C5' '641']
['A' 'C8' '5862']
['A' 'C12' '8670']
['A' 'C13' '6228']
['A' 'C14' '3238']
['A' 'C18' '8950']
```

Sample of Input12.39.34.txt



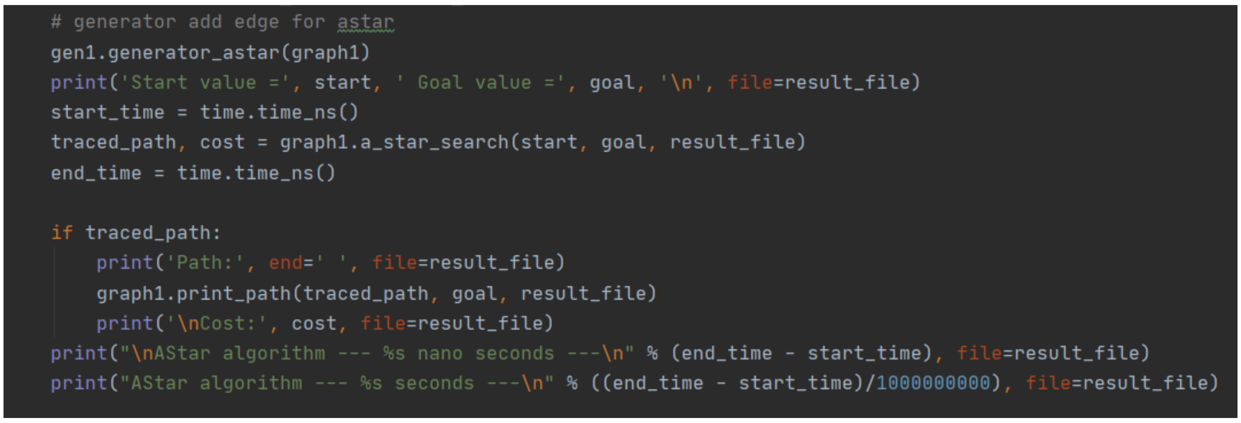
```
Path: T => Y6 => D
Cost: 6011

AStar algorithm --- 12010700 nano seconds ---

AStar algorithm --- 0.0120107 seconds ---
```

Sample of Input12.39.34Result.txt



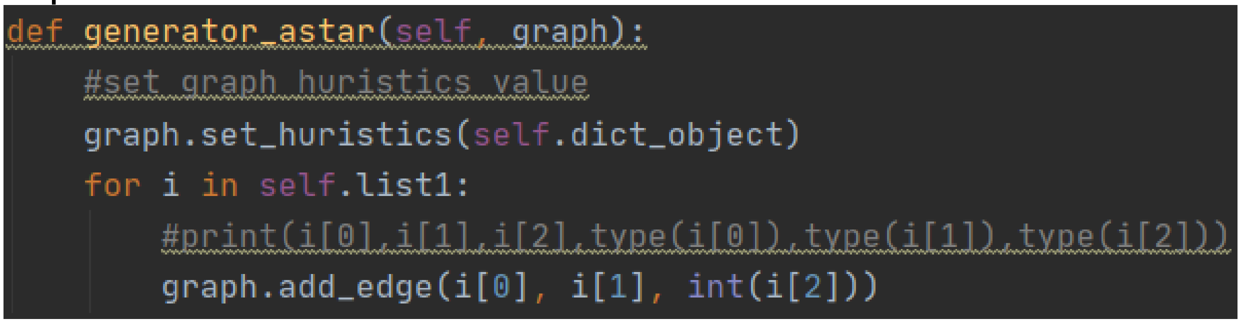
```
start, goal = random.choice(gen1.alphabet), random.choice(gen1.alphabet)
```

Start and goal are randomly selected.



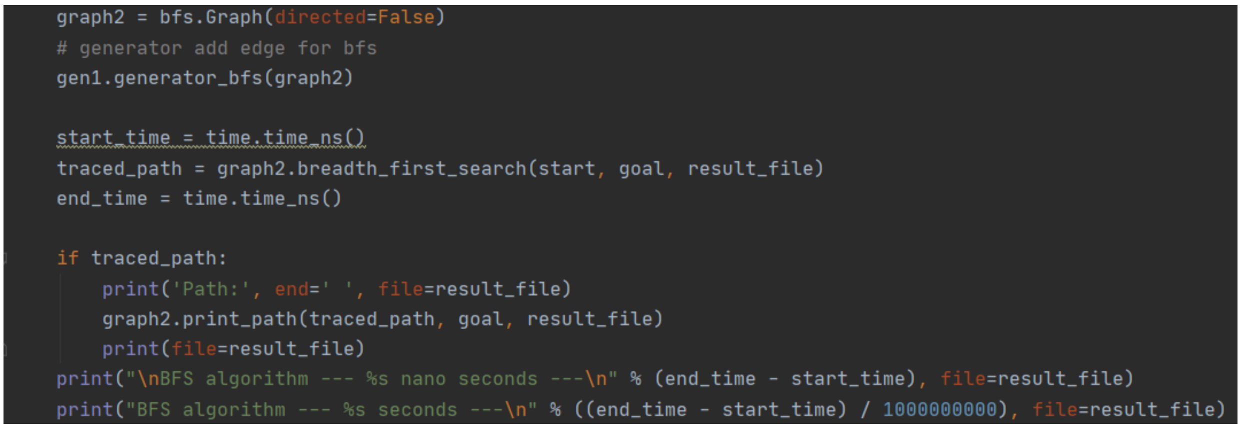
```
# generator add edge for astar
gen1.generator_astar(graph1)
print('Start value =', start, ' Goal value =', goal, '\n', file=result_file)
start_time = time.time_ns()
traced_path, cost = graph1.a_star_search(start, goal, result_file)
end_time = time.time_ns()

if traced_path:
    print('Path:', end=' ', file=result_file)
    graph1.print_path(traced_path, goal, result_file)
    print('\nCost:', cost, file=result_file)
print("\nAStar algorithm --- %s nano seconds ---\n" % (end_time - start_time), file=result_file)
print("AStar algorithm --- %s seconds ---\n" % ((end_time - start_time)/1000000000), file=result_file)
```

The picture above is the lines where the A * Search algorithm is implemented.



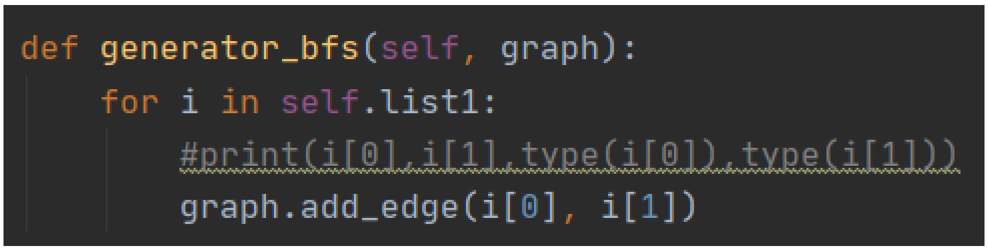
```python
def generator_astar(self, graph):
    #set graph huristics value
    graph.set_huristics(self.dict_object)
    for i in self.list1:
        #print(i[0],i[1],i[2],type(i[0]),type(i[1]),type(i[2]))
        graph.add_edge(i[0], i[1], int(i[2]))
```

Add edges to graph and set huristics values.



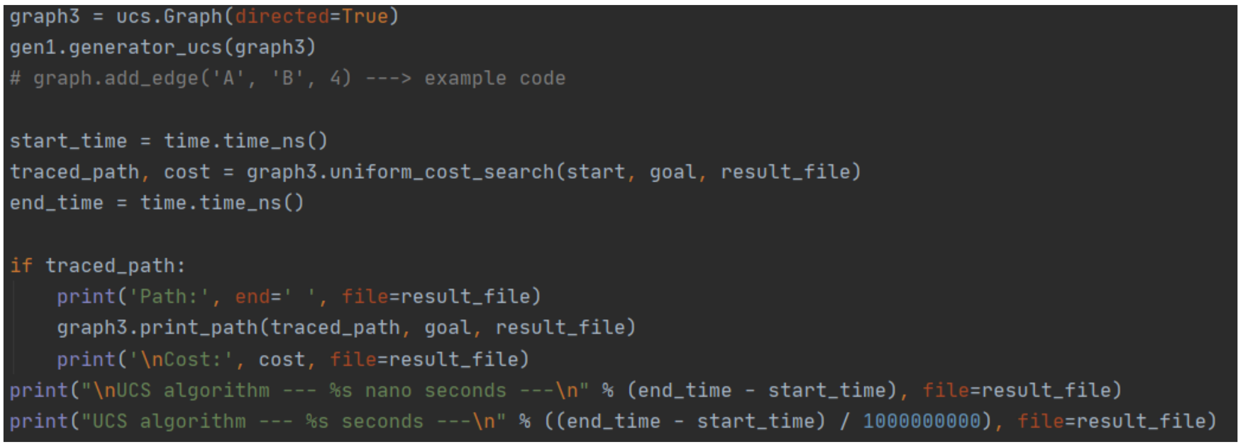
```python
    graph2 = bfs.Graph(directed=False)
    # generator add edge for bfs
    gen1.generator_bfs(graph2)

    start_time = time.time_ns()
    traced_path = graph2.breadth_first_search(start, goal, result_file)
    end_time = time.time_ns()

    if traced_path:
        print('Path:', end=' ', file=result_file)
        graph2.print_path(traced_path, goal, result_file)
        print(file=result_file)
    print("\nBFS algorithm --- %s nano seconds ---\n" % (end_time - start_time), file=result_file)
    print("BFS algorithm --- %s seconds ---\n" % ((end_time - start_time) / 1000000000), file=result_file)
```

The picture above is the lines where the BFS algorithm is implemented .



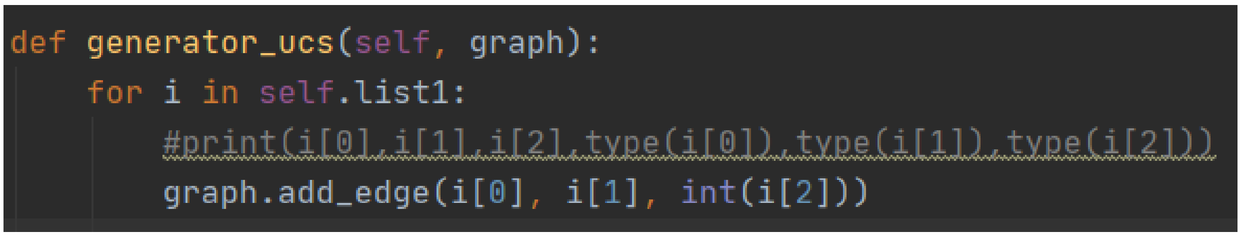
```python
def generator_bfs(self, graph):
    for i in self.list1:
        #print(i[0],i[1],type(i[0]),type(i[1]))
        graph.add_edge(i[0], i[1])
```

Add edges to graph



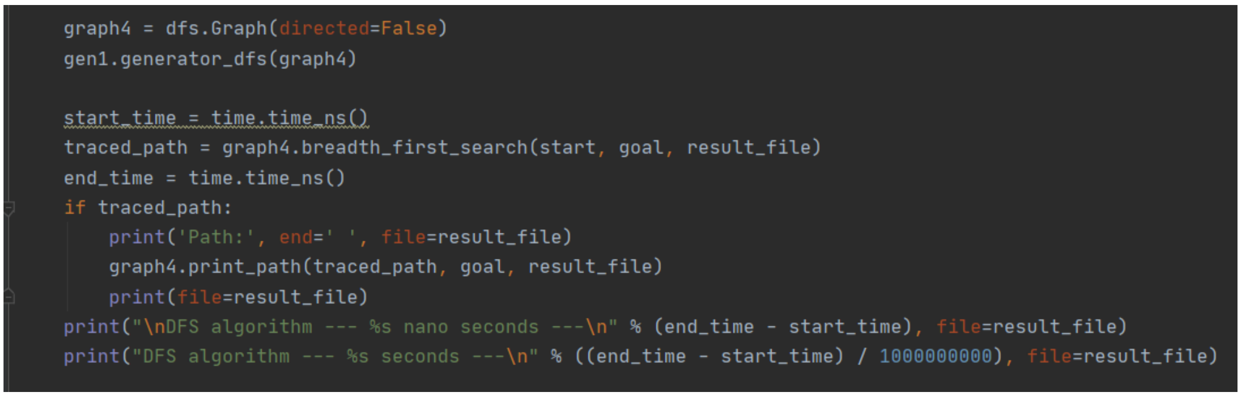
```python
graph3 = ucs.Graph(directed=True)
gen1.generator_ucs(graph3)
# graph.add_edge('A', 'B', 4) ---> example code

start_time = time.time_ns()
traced_path, cost = graph3.uniform_cost_search(start, goal, result_file)
end_time = time.time_ns()

if traced_path:
    print('Path:', end=' ', file=result_file)
    graph3.print_path(traced_path, goal, result_file)
    print('\nCost:', cost, file=result_file)
print("\nUCS algorithm --- %s nano seconds ---\n" % (end_time - start_time), file=result_file)
print("UCS algorithm --- %s seconds ---\n" % ((end_time - start_time) / 1000000000), file=result_file)
```

The picture above is the lines where the UCS algorithm is implemented.



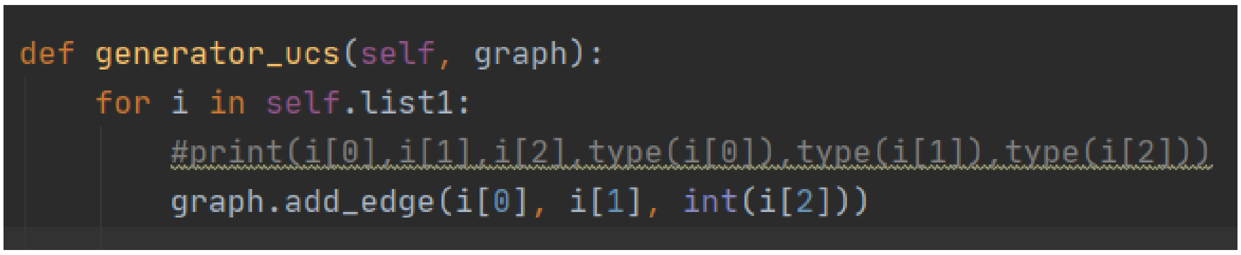
```python
def generator_ucs(self, graph):
    for i in self.list1:
        #print(i[0],i[1],i[2],type(i[0]),type(i[1]),type(i[2]))
        graph.add_edge(i[0], i[1], int(i[2]))
```

Add edges to graph



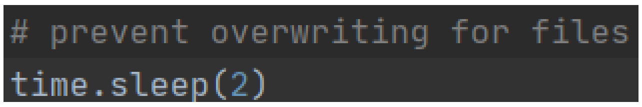
```python
    graph4 = dfs.Graph(directed=False)
    gen1.generator_dfs(graph4)

    start_time = time.time_ns()
    traced_path = graph4.breadth_first_search(start, goal, result_file)
    end_time = time.time_ns()
    if traced_path:
        print('Path:', end=' ', file=result_file)
        graph4.print_path(traced_path, goal, result_file)
        print(file=result_file)
    print("\nDFS algorithm --- %s nano seconds ---\n" % (end_time - start_time), file=result_file)
    print("DFS algorithm --- %s seconds ---\n" % ((end_time - start_time) / 1000000000), file=result_file)
```

The picture above is the lines where the DFS algorithm is implemented.



```python
def generator_ucs(self, graph):
    for i in self.list1:
        #print(i[0],i[1],i[2],type(i[0]),type(i[1]),type(i[2]))
        graph.add_edge(i[0], i[1], int(i[2]))
```

Add edges to graph



```python
# prevent overwriting for files
time.sleep(2)
```

2 second delay

3. Ten Result With fill_and_create() function

Test1	Start value = Q Goal value = D	dictionary size= 100000 list element size=100000	NanoSecond Second
A*	Path: Q => T9 => F => P21 => D	Cost: 13319	612566100 nano seconds 0.6125661 seconds
BFS	Path: Q => V12 => D		0 nano seconds 0 seconds
UCS	No path from Q to D		31270000 nano seconds 0.03127 seconds
DFS	Path: Q => X22 => D		15624300 nano seconds 0.0156243 seconds

Test2	Start value = V Goal value = X	dictionary size= 10000 list element size=100000	NanoSecond Second
A*	Path: V => Y3 => A => Y22 => X	Cost: 8179	1570220200 nano seconds 1.5702202 seconds
BFS	Path: V => Y15 => X		9026100 nano seconds 0.0090261 seconds
UCS	No path from V to X		10104600 nano seconds 0.0101046 seconds
DFS	Path: V => X22 => X		31250300 nano seconds 0.0312503 seconds

Test3	Start value = V Goal value = B	dictionary size= 10 list element size= 1000000	NanoSecond Second
A*	Path: V => Z3 => F => U9 => B	Cost: 81085	257182200 nano seconds 0.2571822 seconds
BFS	Path: V => V12 => B		5003100 nano seconds 0.0050031 seconds
UCS	No path from V to B		10007500 nano seconds 0.0100075 seconds
DFS	Path: V => X22 => B		17009900 nano seconds 0.0170099 seconds

Test4	Start value = O Goal value = F	dictionary size= 100000 list element size=10	NanoSecond Second
A*	Path: O => T4 => F	Cost: 2	254251100 nano seconds 0.2542511 seconds
BFS	Path: O => P9 => F		9006400 nano seconds 0.0090064 seconds
UCS	No path from O to F		55037300 nano seconds 0.0550373 seconds
DFS	Path: O => X22 => F		17011600 nano seconds 0.0170116 seconds

Test5	Start value = I Goal value = B	dictionary size= 100000 list element size=100	NanoSecond Second
A*	Path: I => O13 => B	Cost: 6	1955174700 nano seconds 1.9551747 seconds
BFS	Path: I => K0 => B		15608000 nano seconds 0.015608 seconds
UCS	No path from I to B		125000100 nano seconds 0.1250001 seconds
DFS	Path: I => X22 => B		15645700 nano seconds 0.0156457 seconds

Test6	Start value = E Goal value = M	dictionary size= 10000 list element size= 1000000	NanoSecond Second
A*	Path: E => Q2 => M	Cost: 64890	268190000 nano seconds 0.26819 seconds
BFS	Path: E => P9 => M		15646600 nano seconds 0.0156466 seconds
UCS	No path from E to M		218752000 nano seconds 0.218752 seconds
DFS	Path: E => V7 => M		15625200 nano seconds 0.0156252 seconds

Test7	Start value = S Goal value = U	dictionary size= 10 list element size=100	NanoSecond Second
A*	Path: S => T17 => N => U14 => U	Cost: 11	774296900 nano seconds 0.7742969 seconds
BFS	Path: S => V12 => U		7004600 nano seconds 0.0070046 seconds
UCS	No path from S to U		22017700 nano seconds 0.0220177 seconds
DFS	Path: S => X22 => U		14008600 nano seconds 0.0140086 seconds

Test8	Start value = F Goal value = D	dictionary size= 100000 list element size=10000	NanoSecond Second
A*	F => T12 => D	Cost: 189	3077541300 nano seconds 3.0775413 seconds
BFS	Path: F => K0 => D		31250700 nano seconds 0.0312507 seconds
UCS	No path from F to D		235075600 nano seconds 0.2350756 seconds
DFS	Path: F => H3 => D		10983500 nano seconds 0.0109835 seconds

Test9	Start value = E Goal value = R	dictionary size= 100000 list element size=100	NanoSecond Second
A*	Path: E => M14 => M => V11 => R	Cost: 22	586619100 nano seconds 0.5866191 seconds
BFS	Path: E => U4 => R		25015600 nano seconds 0.0250156 seconds
UCS	No path from E to R		187132500 nano seconds 0.1871325 seconds
DFS	Path: E => V7 => R		10006200 nano seconds 0.0100062 seconds

Test10	Start value = S Goal value = D	dictionary size= 1000 list element size=10	NanoSecond Second
A*	Path: S => S19 => D	Cost: 2	543386700 nano seconds 0.5433867 seconds
BFS	Path: S => V12 => D		11007800 nano seconds 0.0110078 seconds
UCS	No path from S to D		23018400 nano seconds 0.0230184 seconds
DFS	Path: S => X22 => D		0 nano seconds 0 seconds

4. Result With fill_and_create2() function

4.1 %80 valid probability

Test1	Start value = H Goal value = E	dictionary size= 1000000 list element size=10000	NanoSecond Second
A*	Path: H => Z0 => A => G4 => B => G8 => E	Cost: 1038	1438962700 nano seconds 1.4389627 seconds
BFS	Path: H => O2 => E		11008600 nano seconds 0.0110086 seconds
UCS	No path from H to E		78054800 nano seconds 0.0780548 seconds
DFS	Path: H => S24 => E		14009900 nano seconds 0.0140099 seconds

Test2	Start value = G Goal value = V	dictionary size= 1000000 list element size=10	NanoSecond Second
A*	Path: G => Z6 => V	Cost: 3	518369400 nano seconds 0.5183694 seconds
BFS	Path: G => Z6 => V		13009200 nano seconds 0.0130092 seconds
UCS	No path from G to V		81057800 nano seconds 0.0810578 seconds
DFS	Path: G => S24 => C => C17 => A => T12 => K => Y5 => V		4004100 nano seconds 0040041 seconds

Test3	Start value = G Goal value = V	dictionary size= 10 list element size=1000	NanoSecond Second
A*	Path: G => U2 => A => Y9 => V	Cost: 57	283198500 nano seconds 0.2831985 seconds
BFS	Path: G => Z6 => V		16030200 nano seconds 0.0160302 seconds
UCS	No path from G to V		95067300 nano seconds 0.0950673 seconds
DFS	Path: G => S24 => K => V8 => V		11008700 nano seconds 0.0110087 seconds

4.2 %60 valid probability

Test1	Start value = F Goal value = U	dictionary size= 10 list element size=100000	NanoSecond Second
A*	Path: F => K23 => E => V22 => U	Cost: 3915	46030900 nano seconds 0.0460309 seconds
BFS	Path: F => Y2 => U		9006900 nano seconds 0.0090069 seconds
UCS	No path from F to U		76056000 nano seconds 0.076056 seconds
DFS	Path: F => P14 => B => H20 => C => J25 => J => Q3 => P => X6 => U		3002200 nano seconds 0.0030022 seconds

Test2	Start value = F Goal value = R	dictionary size= 10000 list element size= 1000	NanoSecond Second
A*	Path: F => X5 => R	Cost: 118	4336804400 nano seconds 4.3368044 seconds
BFS	Path: F => R7 => R		0 nano seconds 0 seconds
UCS	No path from F to R		89148200 nano seconds 0.0891482 seconds
DFS	Path: F => H20 => B => V6 => R		10989400 nano seconds 0.0109894 seconds

Test3	Start value = K Goal value = A	dictionary size= 10 list element size= 1000	NanoSecond Second
A*	Path: K => N12 => A	Cost: 79	79056500 nano seconds 0.0790565 seconds
BFS	Path: K => N8 => A		8026600 nano seconds 0.0080266 seconds
UCS	No path from K to A		41029100 nano seconds 0.0410291 seconds
DFS	Path: K => X2 => A		26013700 nano seconds 0.0260137 seconds

4.3 %40 valid probability

Test1	Start value = D Goal value = G	dictionary size= 100 list element size=100	NanoSecond Second
A*	Path: D => T6 => G	Cost: 21	0 nano seconds 0.0 seconds
BFS	Path: D => J12 => G		15623400 nano seconds 0.0156234 seconds
UCS	No path from D to G		31270100 nano seconds 0.0312701 seconds
DFS	Path: D => E8 => B => H19 => C => W20 => J => S1 => N => N10 => L => U2 => O => Q9 => G		0 nano seconds 0.0 seconds

Test2	Start value = X Goal value = T	dictionary size= 100 list element size=10000	NanoSecond Second
A*	Path: X => Y16 => N => U16 => T	Cost: 2150	268191300 nano seconds 0.2681913 seconds
BFS	Path: X => Y11 => T		1001100 nano seconds 0.0010011 seconds
UCS	No path from X to T		1019100 nano seconds 0.0010191 seconds
DFS	Path: X => Y16 => J => U15 => T		0 nano seconds 0 seconds

Test3	Start value = Y Goal value = R	dictionary size= 100 list element size=1000	NanoSecond Second
A*	Path: Y => Z4 => F => S17 => R	Cost: 280	62503200 nano seconds 0.0625032 seconds
BFS	Path: Y => Y1 => R		15622700 nano seconds 0.0156227 seconds
UCS	No path from Y to R		0 nano seconds 0.0 seconds
DFS	Path: Y => Z4 => X => Z16 => J => K8 => E => E8 => B => U15 => O => V12 => R		0 nano seconds 0 seconds

4.4 %20 valid probability

Test1	Start value = J Goal value = T	dictionary size= 1000000 list element size=100000	NanoSecond Second
A*	Path: J => T24 => P => T23 => T	Cost: 25505	1137564500 nano seconds 1.1375645 seconds
BFS	Path: J => Y8 => T		0 nano seconds 0.0 seconds
UCS	No path from J to T		0 nano seconds 0.0 seconds
DFS	Path: J => V10 => T		15624300 nano seconds 0.0156243 seconds

Test2	Start value = A Goal value = L	dictionary size= 1000 list element size=100000	NanoSecond Second
A*	Path: A => R13 => L	Cost: 16920	81756200 nano seconds 0.0817562 seconds
BFS	Path: A => L22 => L		7005800 nano seconds 0.0070058 seconds
UCS	No path from A to L		15010200 nano seconds 0.0150102 seconds
DFS	Path: A => L25 => I => T8 => B => N13 => J => R9 => H => P8 => N => X2 => L		10006200 nano seconds 0.0100062 seconds

Test3	Start value = I Goal value = B	dictionary size= 100 list element size=1000	NanoSecond Second
A*	Path: I => V18 => B	Cost: 136	2004000 nano seconds 0.002004 seconds
BFS	Path: I => Q11 => B		2001900 nano seconds 0.0020019 seconds
UCS	No path from I to B		5003600 nano seconds 0.0050036 seconds
DFS	Path: I => L25 => C => V21 => N => Q15 => P => T6 => Q => Z19 => T => Y11 => E => M16 => H => W21 => B		1001500 nano seconds 0.0010015 seconds

4.5 %10 valid probability

Test1	Start value = G Goal value = Z	dictionary size= 100000 list element size=1000	NanoSecond Second
A*	Path: G => X19 => M => X12 => F => Z11 => Z	Cost: 1110	280197300 nano seconds 0.2801973 seconds
BFS	Path: G => I20 => F => Z11 => Z		11007800 nano seconds 0.0110078 seconds
UCS	No path from G to Z		2001500 nano seconds 0.0020015 seconds
DFS	Path: G => X16 => T => U7 => N => R23 => A => Z11 => Z		14010300 nano seconds 0.0140103 seconds

Test2	Start value = D Goal value = H	dictionary size= 1000000 list element size= 1000000	NanoSecond Second
A*	Path: D => L12 => H	Cost: 234057	11008200 nano seconds 0.0110082 seconds
BFS	Path: D => H6 => H		1000300 nano seconds 0.0010003 seconds
UCS	No path from D to H		3003800 nano seconds 0.0030038 seconds
DFS	Path: D => Q2 => J => Z24 => R => S17 => M => U4 => K => X19 => S => Y20 => W => Y0 => C => C22 => B => R23 => N => P24 => G => Q10 => A => W22 => T => U12 => I => Y22 => V => W12 => H		2001500 nano seconds 0.0020015 seconds

Test3	Start value = Z Goal value = W	dictionary size= 100000 list element size= 1000	NanoSecond Second
A*	Path: Z => Z11 => X => Y16 => L => P13 => N => X2 => W	Cost: 1999	523438400 nano seconds 0.5234384 seconds
BFS	Path: Z => Z11 => C => Z18 => W		2998300 nano seconds 0.0029983 seconds
UCS	No path from Z to W		0 nano seconds 0.0 seconds
DFS	Path: Z => Z11 => C => H18 => B => J24 => I => W8 => P => P19 => N => Q19 => Q => U9 => J => N11 => H => X16 => W		7005400 nano seconds 0.0070054 seconds

4.6 %2 valid probability

Test1	Start value = E Goal value = W	dictionary size= 1000 list element size= 10	NanoSecond Second
A*	No path from E to W		0 nano seconds 0.0 seconds
BFS	No path from E to W		0 nano seconds 0.0 seconds
UCS	No path from E to W		0 nano seconds 0.0 seconds
DFS	No path from E to W		0 nano seconds 0.0 seconds

Test2	Start value = O Goal value = K	dictionary size= 10000 list element size=1000	NanoSecond Second
A*	Path: O => Z0 => J => Z9 => U => U23 => T => Z22 => C => P0 => A => R18 => K	Cost: 7095	0 nano seconds 0.0 seconds
BFS	Path: O => Z0 => J => Z9 => U => U23 => T => Z22 => C => P0 => A => R18 => K		0 nano seconds 0.0 seconds
UCS	No path from O to K		0 nano seconds 0.0 seconds
DFS	Path: O => Z0 => J => Z9 => U => U23 => T => Z22 => C => P0 => A => R18 => K		0 nano seconds 0 seconds

Test3	Start value = P Goal value = V	dictionary size= 100000 list element size=100	NanoSecond Second
A*	No path from P to V		15644000 nano seconds 0.015644 seconds
BFS	No path from P to V		0 nano seconds 0.0 seconds
UCS	No path from P to V		0 nano seconds 0.0 seconds
DFS	No path from P to V		0 nano seconds 0 seconds

4.7 %1 valid probability

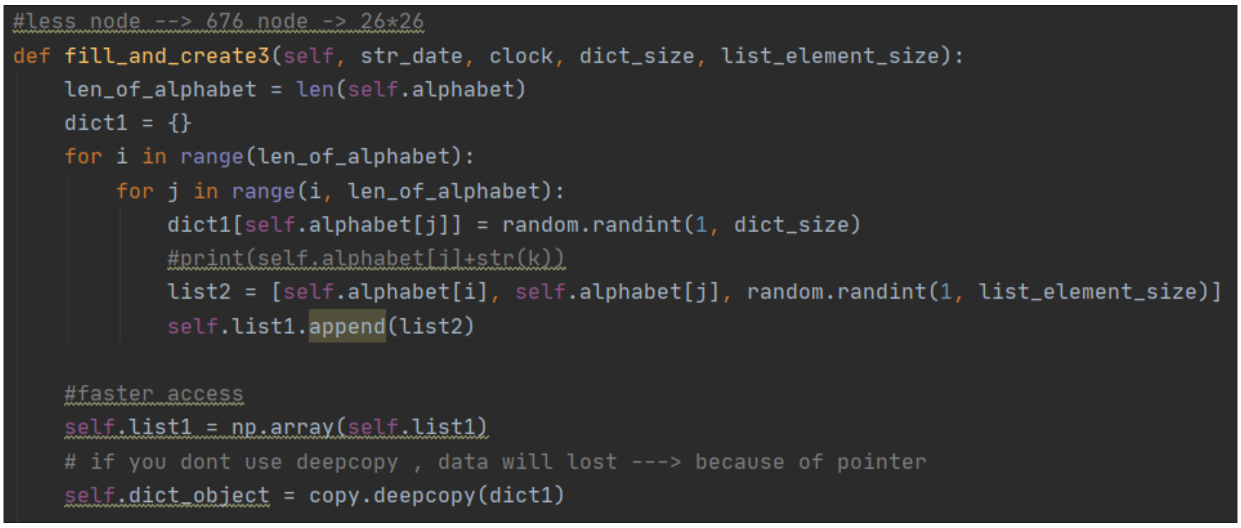
There is no path in all search algorithms in the graph, which has 1% chance.

5. UCS Failure and new 2 function

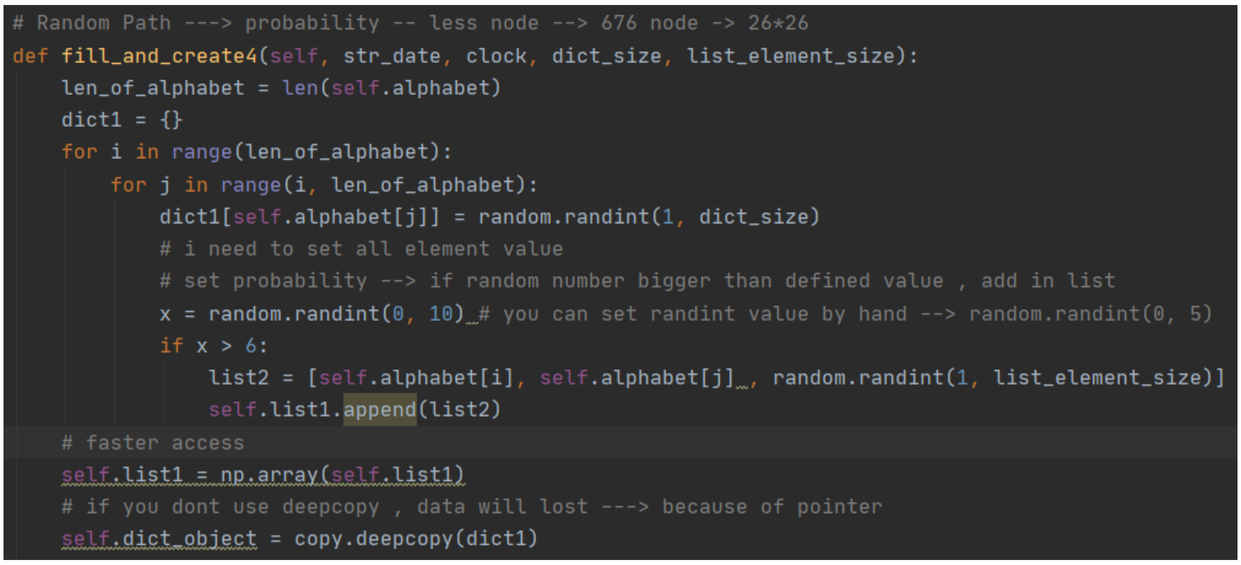
Generally other algorithms(except UCS) found a path. Unfortunately, Uniform Cost Search failed on multi-edge graphs.

Unfortunately, UCS cannot find the path when all edges are linked. This algorithm(UCS) may be stuck in an infinite loop because sometime process time not equals to 0.

If the number of edges decreases, it is successful in finding results. I have defined a new 2 function for UCS to find results.



```python
#less node --> 676 node -> 26*26
def fill_and_create3(self, str_date, clock, dict_size, list_element_size):
    len_of_alphabet = len(self.alphabet)
    dict1 = {}
    for i in range(len_of_alphabet):
        for j in range(i, len_of_alphabet):
            dict1[self.alphabet[j]] = random.randint(1, dict_size)
            #print(self.alphabet[j]+str(k))
            list2 = [self.alphabet[i], self.alphabet[j], random.randint(1, list_element_size)]
            self.list1.append(list2)

    #faster access
    self.list1 = np.array(self.list1)
    # if you dont use deepcopy , data will lost ---> because of pointer
    self.dict_object = copy.deepcopy(dict1)
```



```
# Random Path ---> probability -- less node --> 676 node -> 26*26
def fill_and_create4(self, str_date, clock, dict_size, list_element_size):
    len_of_alphabet = len(self.alphabet)
    dict1 = {}
    for i in range(len_of_alphabet):
        for j in range(i, len_of_alphabet):
            dict1[self.alphabet[j]] = random.randint(1, dict_size)
            # i need to set all element value
            # set probability --> if random number bigger than defined value , add in list
            x = random.randint(0, 10) # you can set randint value by hand --> random.randint(0, 5)
            if x > 6:
                list2 = [self.alphabet[i], self.alphabet[j] , random.randint(1, list_element_size)]
                self.list1.append(list2)
    # faster access
    self.list1 = np.array(self.list1)
    # if you dont use deepcopy , data will lost ---> because of pointer
    self.dict_object = copy.deepcopy(dict1)
```

Maximum number of edges that functions can create 676

5.1 Fill_and_create3()

Test1	Start value = M Goal value = A	dictionary size= 10000 list element size= 1000000	NanoSecond Second
A*	Path: M => E => J => G => P => A	Cost: 170675	2001800 nano seconds 0.0020018 seconds
BFS	Path: M => A		0 nano seconds 0.0 seconds
UCS	No path from M to A		998500 nano seconds 0.0009985 seconds
DFS	Path: M => A		0 nano seconds 0.0 seconds

Test2	Start value = L Goal value = K	dictionary size= 10000 list element size=100000	NanoSecond Second
A*	Path: L => K	Cost: 8276	999400 nano seconds 0.0009994 seconds
BFS	Path: L => K		0 nano seconds 0.0 seconds
UCS	No path from L to K		1001600 nano seconds 0.0010016 seconds
DFS	Path: L => K		1004600 nano seconds 0.0010046 seconds

Test3	Start value = P Goal value = W	dictionary size= 10000 list element size= 10000	NanoSecond Second
A*	Path: P => T => W	Cost: 3584	5001800 nano seconds 0.0050018 seconds
BFS	Path: P => W		1004100 nano seconds 0.0010041 seconds
UCS	Path: P => T => W	Cost: 3584	0 nano seconds 0.0 seconds
DFS	Path: P => W		1000300 nano seconds 0.0010003 seconds

5.2 Fill_and_create4()

Test1	Start value = S Goal value = W	dictionary size= 1000000 list element size= 1000000	NanoSecond Second
A*	Path: S => B => G => W	Cost: 530694	0 nano seconds 0.0 seconds
BFS	Path: S => W		1001200 nano seconds 0.0010012 seconds
UCS	Path: S => W	Cost: 920689	0 nano seconds 0.0 seconds
DFS	Path: S => W		1000700 nano seconds 0.0010007 seconds

Test2	Start value = B Goal value = Z	dictionary size= 10000 list element size=100000	NanoSecond Second
A*	Path: B => Z	Cost: 2821	0 nano seconds 0.0 seconds
BFS	Path: B => Z		1001600 nano seconds 0.0010016 seconds
UCS	Path: B => Z	Cost: 2821	0 nano seconds 0.0 seconds
DFS	Path: B => Z		998600 nano seconds 0.0009986 seconds

Test3	Start value = C Goal value = S	dictionary size= 100000 list element size=1000000	NanoSecond Second
A*	Path: C => X	Cost: 98599	0 nano seconds 0.0 seconds
BFS	Path: C => X		0 nano seconds 0.0 seconds
UCS	Path: C => X	Cost: 98599	0 nano seconds 0.0 seconds
DFS	Path: C => X		0 nano seconds 0.0 seconds

For UCS, reducing the number of edges worked, but no distinctive differences occurred.

6. Comments Algorithms

6.1 BFS

Advantages:

BFS will provide a solution if any solution exists.

If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

It requires lots of memory since each level of the tree must be saved into memory to expand the next level.

BFS needs lots of time if the solution is far away from the root node.

Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

T (b) = 1+b**2+b**3+.......+ b**d= O (b**d)

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is O(b**d).

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

6.2 DFS

Advantages:

DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.

It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantages:

There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.

DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

T(n)= 1+ n**2+ n**3 +.........+ n**m=O(n**m)

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

6.3 UCS

Advantages:

Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Completeness: Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity: Let C* is Cost of the optimal solution, and ε is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is O(b1 + [C*/ε])/.

Space Complexity: The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is O(b1 + [C*/ε]).

Optimal: Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

6.4 A* Search

Advantages:

A* search algorithm is the best algorithm than other search algorithms.

A* search algorithm is optimal and complete.

This algorithm can solve very complex problems.

Disadvantages:

It does not always produce the shortest path as it mostly based on heuristics and approximation.

A* search algorithm has some complexity issues.

The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems

Complete: A* algorithm is complete as long as:

Branching factor is finite.

Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

Admissible: the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.

Consistency: Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is O(b^d)