CSC421 Assignment1
Da Meng V00838849

Introduction

The program is divided into four main parts: create the city graph, define five search function, visualize the map, and call the search function do the calculation.

First of all, generate a 100*100 list, which like the world_map, and then randomly select 26 pairs of x-axis and y-axis (check to make sure they are all different), then assign the city name from A to Z to the pairs. Store the location into a dictionary.

Secondly, use the location dictionary to calculate the distance between cities for each city, and then select the closest four neighbours (not include the city itself) for each city and store them in a ordered dictionary, which will be the city graph. Also check the graph to make sure that the edge is bi-directional. In my case, the neighbour cities stored in the graph are sorted based on the closest distance. Therefore, the final city graph looks like:

```
OrderedDict([('A', ['V', 'G', 'K', 'F', 'N']), ('B', ['W', 'F', 'R', 'L', 'H', '
K']), ('C', ['T', 'O', 'M', 'U', 'D', 'J']), ('D', ['J', 'Z', 'C', 'I']), ('E',
['U', 'X', 'R', 'L']), ('F', ['R', 'K', 'B', 'Q', 'A', 'G']), ('G', ['Q', 'K', '
A', 'F', 'V']), ('H', ['L', 'W', 'B', 'R']), ('I', ['Z', 'S', 'J', 'U', 'D']), (
'J', ['D', 'Z', 'I', 'C', 'S']), ('K', ['F', 'Q', 'R', 'B', 'A', 'G']), ('L', ['
H', 'W', 'B', 'R', 'E']), ('M', ['T', 'P', 'X', 'O', 'C', 'N', 'Y']), ('N', ['V'
, 'X', 'A', 'M']), ('O', ['C', 'T', 'M', 'P', 'Y']), ('P', ['Y', 'M', 'O', 'T'])
, ('Q', ['G', 'K', 'F', 'R']), ('R', ['F', 'K', 'B', 'W', 'E', 'H', 'L', 'Q']),
('S', ['I', 'Z', 'U', 'J']), ('T', ['C', 'O', 'M', 'X', 'P', 'U', 'Y']), ('U', [
'E', 'C', 'I', 'T', 'S', 'Z']), ('V', ['A', 'N', 'X', 'G']), ('W', ['B', 'L', 'H
', 'R']), ('X', ['M', 'N', 'T', 'E', 'V']), ('Y', ['P', 'M', 'O', 'T']), ('Z', [
'I', 'S', 'J', 'U', 'D'])])
```

(The closest neighbours of A will be V, then G then K then F, then N. etc. The reason why A has 5 neighbours is because N is not the top four close cities of A, but A is the top four of N. Thus, because of the bi-directional edges, N was added to A's neighbours)

Thirdly, write a for loop to call those functions (because we want to generate 100 instance). In the for loop, call the function to generate the world map, then call the function to generate the city graph. Now randomly generate two city names from A to Z (make sure they are not same) as the start state and goal state, for example, start_state = 'A', goal_state = 'C'.

Fourthly, define the search function and implement the search algorithm. Details will be introduced in the implementation overview section.

Fifthly, call the search function, given the start state, goal state and city graph as the basic parameters. For different search algorithm, the input parameter might be different. Search function will return the path, the number of explored cities (which will be the time complexity) and also the maximum nodes generated (which will be the space complexity). Calculate the total space complexity, time complexity, running time, path length and number of solved problem.

Sixthly, after the for loop, calculate the average of the above statistics.

Experimental Results

Firstly, I want to answer the questions in the assignment requirement.

Q1: The way how I formulate this problem as a search problem is as what I mentioned in the previous section. I create a city graph, and randomly select two city names, and given start city name, goal city name, and also the city graph as the parameters when I call the search function. The state space is presented by the city name. The successor-function is like the neighbours which will be visited next. This can be get through the city graph, like city_graph[city_name], and this will return all the neighbours of the current city, which will be the successors.

Q2: After generate 100 instance graph and calculate the branches, the average branches is 104. The way to count the branches is done by the visualization method. Every time draw an edge, increase the count of branches.

Q3: Examples of calling breadth first search (bfs), depth first search (dfs) and iterative deepening search (ids, consider the running time of the iterative deepening search, set the depth to 10) are as follow:

```
#run bfs
city_graph = create_graph(city_location)
bfs_start_time = time.time()
bfs_result = bfs(start_state, goal_state, city_graph)
bfs_end_time = time.time()
bfs_running_time = bfs_running_time + (bfs_end_time - bfs_start_time)
bfs_path = None
if bfs_result is not None:
    bfs_solved = bfs_solved + 1
    bfs_path = bfs_result[0]
    bfs_path_len = bfs_path_len + len(bfs_path)
    bfs_time_complexity = bfs_time_complexity + int(bfs_result[1])
    bfs_space_complexity = bfs_space_complexity + int(bfs_result[2])
print 'bfs_path:', bfs_path

#run dfs
city_graph = create_graph(city_location)
dfs_start_time = time.time()
dfs_result = dfs(start_state, goal_state, city_graph)
dfs_end_time = time.time()
dfs_running_time = dfs_running_time + (dfs_end_time - dfs_start_time)
dfs_path = None
if dfs_result is not None:
    dfs_solved = dfs_solved + 1
    dfs_path = dfs_result[0]
    dfs_path_len = dfs_path_len + len(dfs_path)
    dfs_time_complexity = dfs_time_complexity + dfs_result[1]
    dfs_space_complexity = dfs_space_complexity + dfs_result[2]
print 'dfs_path:', dfs_path

#run ids
city_graph = create_graph(city_location)
ids_start_time = time.time()
ids_result = ids(start_state, goal_state, city_graph)
ids_end_time = time.time()
ids_running_time = ids_running_time + (ids_end_time - ids_start_time)
ids_path = None
if ids_result is not None:
    ids_solved = ids_solved + 1
    ids_path = ids_result[0]
    ids_path_len = ids_path_len + len(ids_path)
    ids_time_complexity = ids_time_complexity + ids_result[1]
    ids_space_complexity = ids_space_complexity + ids_result[1]
print 'ids_path:', ids_path
```

Result looks like:

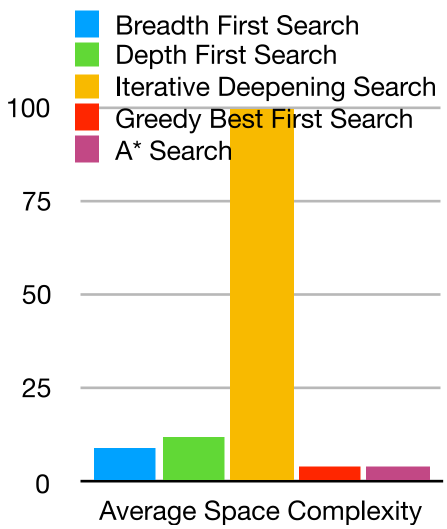
```
start_state: M
goal_state: S
bfs_path: ['M', 'N', 'S']
dfs_path: ['M', 'O', 'Y', 'S']
ids_path: ['M', 'N', 'S']
OrderedDict([('A', ['X', 'G', 'P', 'F']), ('B', ['E', 'R', 'Z', 'Q']), ('C', ['J
', 'L', 'K', 'W', 'V']), ('D', ['G', 'M', 'X', 'H', 'F']), ('E', ['R', 'B', 'P',
 'U']), ('F', ['X', 'G', 'A', 'D']), ('G', ['X', 'D', 'A', 'P', 'F']), ('H', ['M
', 'O', 'D', 'I']), ('I', ['N', 'M', 'O', 'U', 'H']), ('J', ['C', 'V', 'S', 'L',
 'W']), ('K', ['T', 'Q', 'Z', 'C', 'L', 'W']), ('L', ['W', 'C', 'J', 'K']), ('M'
, ['O', 'I', 'D', 'H', 'N']), ('N', ['I', 'O', 'S', 'M', 'Y']), ('O', ['M', 'I',
 'N', 'H', 'Y']), ('P', ['R', 'U', 'E', 'A', 'G']), ('Q', ['Z', 'T', 'K', 'U', '
B']), ('R', ['E', 'P', 'U', 'Z', 'B']), ('S', ['V', 'Y', 'J', 'N']), ('T', ['Z',
 'Q', 'K', 'U']), ('U', ['Z', 'T', 'Q', 'R', 'E', 'I', 'P']), ('V', ['S', 'Y', '
J', 'C']), ('W', ['L', 'C', 'J', 'K']), ('X', ['G', 'A', 'F', 'D']), ('Y', ['S',
 'V', 'O', 'N']), ('Z', ['Q', 'U', 'T', 'K', 'B', 'R'])])
```

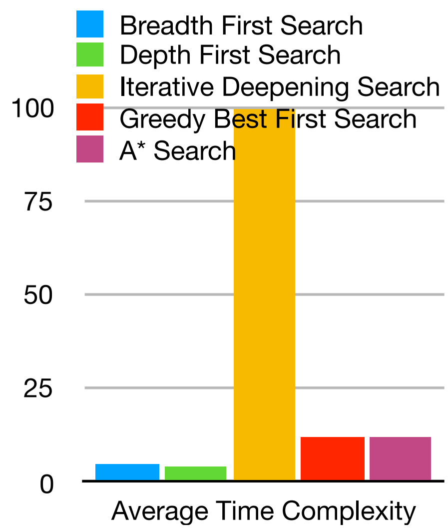
Q4, Q6: The results are based on the same 100 graph, which means every time generate a graph, run these five search algorithm and record the result.

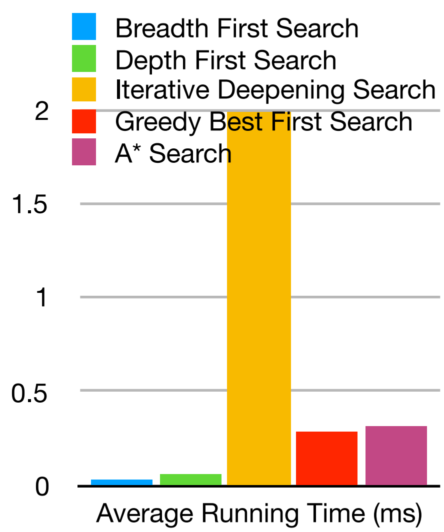
Search Alg.	Average Space Complexity	Average Time Complexity	Average Running Time (s)	Average Path Length	Problems Solved
Breadth First (bfs)	9 $O(b^{d+1})$	5 $O(b^{d+1})$	2.54E-05	3	97
Depth First (dfs)	12 $O(bm)$	4 $O(b^m)$	5.9E-05	5	97
Iterative Deepening (ids)	601 $O(bm)$	601 $O(b^m)$	0.034	3	97
Greedy Best First (bestfs)	4 $O(b^m)$	12 $O(b^m)$	0.00028	4	97
A*	4 $O(b^m)$	12 $O(b^m)$	0.00031	4	97

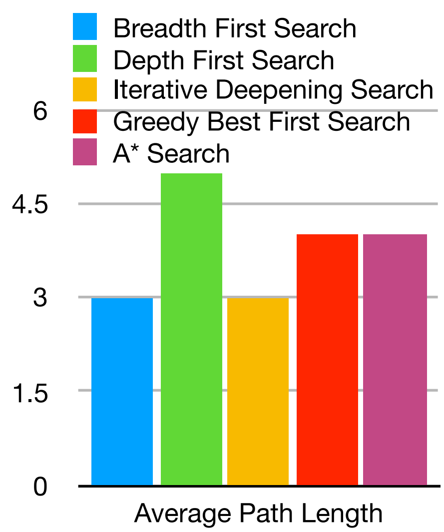
Here I will do the experimental result analysis:

Based on the average space complexity, we could see that ids has the highest amount, which is what we could imagine. Because it is doing the recursive visit according to the depth, it will use more memory than others and also cause a longer running time (especially when it has no solution). For greedy best first search and A* search, because they do a selection before select the neighbours, so they will store less useless cities and also because of the selection, it will take a little bit longer running time. The time complexity is vary depends on the situation, but overall, ids visit the most amount of nodes. For the average path length, we could see that dfs has a longer length and that because dfs will not always return a optimal path. Whereas bfs and ids will return the shortest path, while, greedy best first search (not optimal) and A* is more rely on the distance cost. Because all search algorithm are running on the same graph every time, so the solved problems is equal in this sample. The bar graph could also show the same thing: (for a better view, set a upper bound for the value, but know that ids has a large amount)







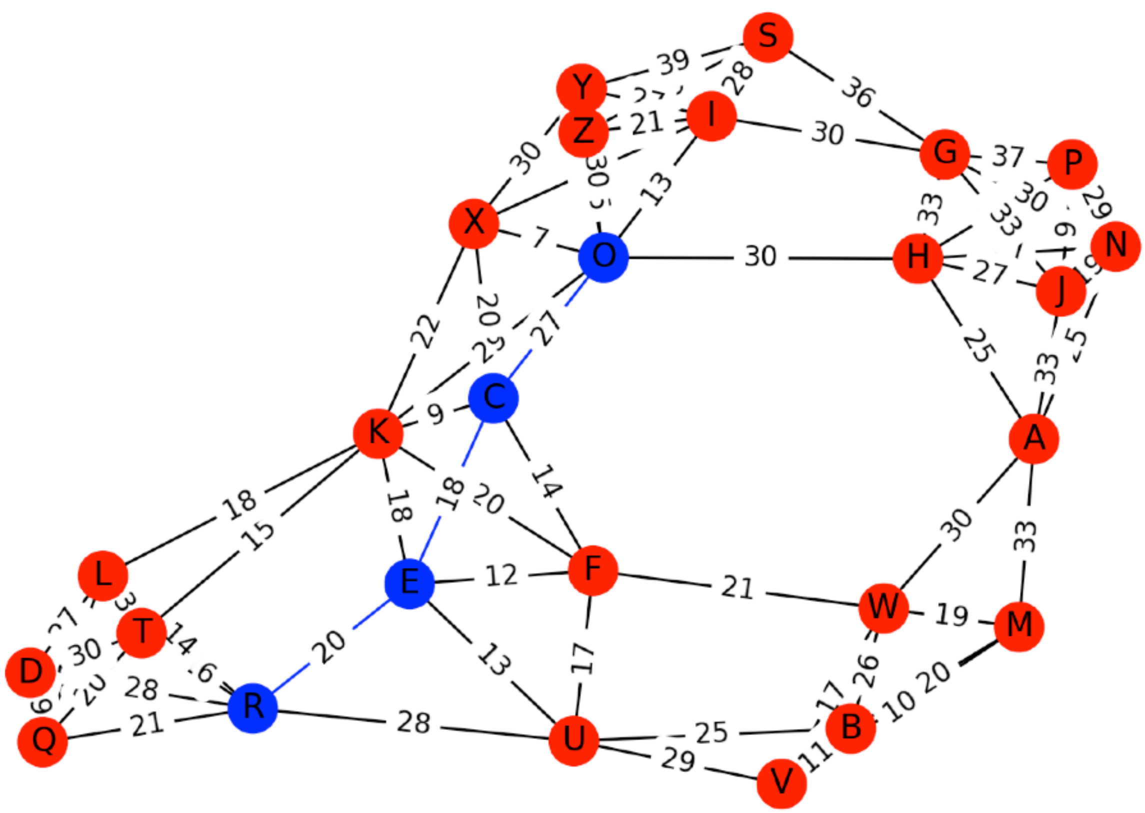


Q5: (he) straight-line Euclidean distance is the direct distance between the current state to the goal state. (h0) means that the start state is the goal state. (hm) Manhattan distance measures the distance in the two dimension coordinates and count the move steps. In the greedy best-first search, the heuristic function takes the (he) estimated cost to the goal state, which means that the next explored node should has the shortest (he) straight-line distance to the goal state. But the return path might not be optimal. While in the A* search, the heuristic takes both the real cost to the current, and the estimated cost to the goal state. For example, if currently at city C, then A* will search if there is a less cost path to get C, if there is one, then get rid of the current one, choose the less cost one. Thus, the return path is optimal. Considering the heuristics is because want to get a optimal path.

When h(m) > h(e) for all n nodes, then h(m) dominates h(e) better for search.

Q7: Visualizes the cities and path and show examples of how it would work.

After running the breadth first search, call the method draw_graph(city_graph, city_location, bfs_path) and then it will show a graph looks like this:



Where blue nodes and the blue edges is the path.

Implementation Overview

The program is implemented using python based on the sudo code from textbook and also the graph search algorithm from aima GitHub account. As mentioned before, the program is basically divided into four parts: create the city map, define the search algorithm, visualize the city map and call the search function do the calculation. Now I am going to introduce each function in my program.

For the first part, create the city map, there are three function I wrote:

create_map( ): used to generate a 100*100 list and then randomly assign 26 cities' location. Return a list of city location.

dist_cost(listone, listtwo): used to calculate the distance between two cities. Return an int distance value.

create_graph(city_location): used to create the city map. Return an ordered dictionary which is the city graph.

For the second part, define search function, I will introduce each search method and some variables I used in all method.

Variables:

frontier: list that stores the neighbours of the start state. It is a queue in the breadth first search function, but it is a stack in the depth first search function. I define it clearly in the code.

frontier_size: This is used to track the space complexity. It is the maximum size of the queue or the stack.

explored: list of all the nodes have been visited

parent: dictionary that used to track where the node comes from

Function:

get_path(parent, start, end): This is a general method of getting the path by given the parent list and the start, end state. Return a list contains the path.

bfs(start_state, goal_state, city_graph): This is the method of breadth first search algorithm. When the frontier is not empty, get the first-in one and check if it is the goal state. If not, then check each of its child (neighbour), if not been visited not in the frontier, check if it is the goal, if not then add to the frontier, else return. Return the path, total number of nodes been visited, maximum size of the queue or none.

dfs(start_state, goal_state, city_graph): This is the method of depth first search algorithm. Same idea as the previous one, instead frontier here is a stack, and always get the last-in one to check. Return the path, total number of nodes been visited, maximum size of the stack or none.

ids(start_state, goal_state, city_graph): This is the method of iterative deepening search algorithm. Set a depth (consider the running time, set it to 10), and call the depth_limited_search and return the result from there or none.

depth_limited_search(start_state, goal_state, city_graph, limit): Inside this method, define a recursive method to search based on the limit. Because it is recursive, to calculate this space complexity, I use the (node in current depth) times depth to present. Return the path, total number of nodes been visited, space complexity or none.

best_first_search(start_state, goal_state, city_graph, f): f here is the list of the straight line distance from each node to the goal state. In the method, frontier is the priority queue. The priority of each neighbour city is assigned using the integer distance to the goal state (example frontier = [[55,'A'],[27, 'B'],[60, ‘C’]]). Because the neighbour cities are already sorted based on the distance (mentioned in the very beginning, this is the way I create the graph), so the running result is almost the same as the A* search. Return the path, total number of nodes been visited, maximum size of the queue or none.

cost(goal_state, city_location): Calculate the distance from each node to the distance.

A_star_search(start_state, goal_state, city_graph, city_location, h): h plays the same role as the f in the greedy best first search. I add g which is a dictionary contains the real cost from start node to the current node. Return the path, total number of nodes been visited, maximum size of the queue or none.

Third part is to visualize the graph. I import the matplotlib.pyplot and networkx.

draw_graph(city_graph, city_location, path): Create an empty graph and add the node and edges to it. Draw it with different colour and value.

Last part is just the for loop to call the above function. Call create_map( ) and create_graph( ) to get the city graph and then randomly select two city names. Call each search function to calculate the statistics. After the for loop, calculate the average statistics for each search function.

Code Listing

```
import re
import sys
import random
import math
import collections
import Queue
import matplotlib.pyplot as plt
import networkx as nx
from datetime import datetime
import timeit
import time

#randomly create the world map with 26 cities
def create_map():
    #global world_map
    world_map = [[0 for x in range(100)] for y in range(100)] #range(100): 0-99
    city_list = collections.OrderedDict()
    for i in range(26):  #0-25
        x = random.randint (0,99)   #0-99
        y = random.randint (0,99)
        #while world_map[x][y] != 0:

        while world_map[x][y] is not 0:
            x = random.randint (0,99)
            y = random.randint (0,99)

        if world_map[x][y] is 0:
            city_name = chr(ord('A') + i)
            world_map[x][y] = city_name
            city_list[city_name] = [x,y]
            #print(city_list)

    return city_list

#calculate the closest four neighbours for each city
def create_graph(city_location):
    graph = collections.OrderedDict()
```

```
    for i in range(26):
        city_name = chr(ord('A') + i)
        all_neighbor = {}
        for j in range(26):
            cmped_city_name = chr(ord('A') + j)
            dist = dist_cost(city_location[city_name], city_location[cmped_city_name])
            if dist!= 0:
                all_neighbor[cmped_city_name] = dist
        # sort all neighbors based on distance
        sorted_neighbor = sorted(all_neighbor.items(), key = lambda x:x[1])

        close_neighbor = []   # the four closest neighbors
        for i in range(4):
            close_neighbor.append(sorted_neighbor[i][0])
        graph[city_name] = close_neighbor

    #add bi-direction neighbor, if A:[B,C,D,T], but B:[M,N,Q,E], need add A to B's neighbor list
    for i in range(26):
        city_name = chr(ord('A') + i)
        for j in range(26):
            cmped_city_name = chr(ord('A') + j)
            if city_name in graph[cmped_city_name]:
                if cmped_city_name not in graph[city_name]:
                    graph[city_name].append(cmped_city_name)

    return graph

def dist_cost(listone, listtwo):
    x_dist = listone[0]-listtwo[0]
    y_dist = listone[1]-listtwo[1]
    square_dist = math.pow(x_dist,2) + math.pow(y_dist,2)
    distance = math.sqrt(square_dist)
    new_distance = int(distance)

    return new_distance

#general method of getting the path
def get_path(parent, start, end):
    child = end
    path = [end]
#    print 'parent:', parent
    if start in parent:
        del parent[start]
    if child in parent:
        while parent[child]:
            if parent[child] in path:
                break
            path.append(parent[child])
            child = parent[child]
            if child not in parent:
                break
            # avoid the case 'K': 'I', 'I':'K', while loop will be infinite
            child_parent = parent[child]
            if child_parent in parent:
```

```
                if parent[child_parent] == child:
                    if child_parent not in path:
                        path.append(child_parent)
                    break
    if start != path[-1]:
        path.append(start)
    path.reverse()
    return path

#-----------------------BFS:Breadth First Search---------------------------------------#
def bfs(start_state, goal_state, city_graph):
    frontier = collections.deque(city_graph[start_state])
    frontier_size = len(frontier)
    explored = []
    explored.append(start_state)
    path = []
    parent = {}
    for city in city_graph[start_state]:
        parent[city] = start_state
    while frontier:
        node = frontier.popleft()
        frontier_size = frontier_size - 1
        explored.append(node)
        if node == goal_state:
            path.append(start_state)
            path.append(node)
            return (path, len(explored), frontier_size)
        child_list = city_graph[node]
        for child in child_list:
            if child not in explored and child not in frontier:
                if child == goal_state:
                    parent[child] = node
                    path = get_path(parent, start_state, goal_state)
                    return (path, len(explored), frontier_size)
                parent[child] = node
                frontier.append(child)
                frontier_size = frontier_size + 1
    return None

#-----------------------------------DFS------------------------------------#
def dfs(start_state, goal_state, city_graph):
    frontier = []
    for i in range(len(city_graph[start_state])-1,-1,-1):
        frontier.append(city_graph[start_state][i])
    frontier_size = len(city_graph[start_state])
    explored = []
    explored.append(start_state)
    parent = {}
    for city in city_graph[start_state]:
        parent[city] = start_state
    path = []
    while frontier:
        node = frontier.pop()
        frontier_size = frontier_size - 1
```

```
        explored.append(node)
        if node == goal_state:
            path = get_path(parent, start_state, goal_state)
        #    print 'dfs_explored:', explored
            return (path,len(explored), frontier_size)
        child_list = city_graph[node]  #the neighbors of the current city
        for child in child_list:
            if child not in explored and child not in frontier:
                parent[child] = node
        frontier.extend(child for child in child_list if child not in explored and child not in frontier)
        frontier_size = frontier_size + 1
    return None

#-----------------------------------IDS------------------------------------#
def depth_limited_search(start_state, goal_state, city_graph, limit):
    parent = {}
    for city in city_graph[start_state]:
        parent[city] = start_state
    explored = []
    def recursive_dls(node, goal_state, city_graph, limit, parent, explored):
        explored.append(node)
        if node == goal_state:
        #    print 'parent1:', parent
            result = get_path(parent, start_state, goal_state)
            return (result, len(explored), len(parent)*limit)
        elif limit == 0:
            return 'cutoff'
        else:
            cutoff_occurred = False
            child_list = city_graph[node]
            for child in child_list:
                parent[child] = node
                result = recursive_dls(child, goal_state,city_graph, limit - 1, parent, explored)
                if result == 'cutoff':
                    cutoff_occurred = True
                elif result is not None:
                #    print 'paren2:', parent
                #     result = get_path(parent, start_state, goal_state)
                    return result
            return 'cutoff' if cutoff_occurred else None

    return recursive_dls(start_state, goal_state, city_graph, limit, parent, explored)

def ids(start_state, goal_state, city_graph):
    for depth in xrange(10):
        result = depth_limited_search(start_state, goal_state, city_graph, depth)
        if result != 'cutoff':
            return result
    return None

#---------------------------------Greedy Best first search---------------------------------#
def cost(goal_state, city_location):
    staright_line_distance = {}
    for i in city_location:
```

```
        dist = dist_cost(city_location[i], city_location[goal_state])
        staright_line_distance[i] = dist
    distance = sorted(staright_line_distance.items(), key = lambda x:x[1])
    dict_distance = collections.OrderedDict()
    for i in range(len(distance)):
        dict_distance[distance[i][0]] = distance[i][1]
#    print "staright_line_distance:", dict_distance
    return dict_distance

def best_first_search(start_state, goal_state, city_graph, f):  #have a list with all the distance of
neighbor to the goal_state
    frontier = Queue.PriorityQueue()
    for city in city_graph[start_state]:
        frontier.put([f[city], city])   #  frontier  = [[1,'A'],[2, 'B'],[3, 'C']]
    frontier_size = frontier.qsize()
    parent = {}
    for city in city_graph[start_state]:
        parent[city] = start_state
    explored = []
    explored.append(start_state)
    while not frontier.empty():
        city = frontier.get()
        frontier_size = frontier_size - 1
        node = city[1]
        explored.append(node)
        if node == goal_state:
            path = get_path(parent, start_state, goal_state)
            return (path, len(explored), frontier_size)
        child_list = city_graph[node]
        for child in child_list:
            frontier_list = frontier.queue
            frontier_city = []
            for i in range(len(frontier_list)):
                frontier_city.append(frontier_list[i][1])
            if child not in explored and child not in frontier_list:
                parent[child] = node
                frontier.put([f[child], child])
                frontier_size = frontier_size + 1
    return None

#-----------------------------------A* Search-----------------------------------------#
def A_star_search(start_state, goal_state, city_graph, city_location, h):  ##have a list with all the
distance of neighbor to the goal_state
    frontier = Queue.PriorityQueue()
    for city in city_graph[start_state]:
        frontier.put([h[city], city])   #  frontier  = [[1,'A'],[2, 'B'],[3, 'C']]
    frontier_size = frontier.qsize()
    parent = {}
    for city in city_graph[start_state]:
        parent[city] = start_state
    explored = []
    explored.append(start_state)
    # g is real cost from start to the current
    g = {}
```

```
    for city in city_graph[start_state]:
        dist = dist_cost(city_location[city], city_location[start_state])
        g[city] = dist

    while not frontier.empty():
        city = frontier.get()
        frontier_size = frontier_size - 1
        node = city[1]
        explored.append(node)
        if node == goal_state:
            path = get_path(parent, start_state, goal_state)
            return (path, len(explored), frontier_size)
        child_list = city_graph[node]
        for child in child_list:
            dist = dist_cost(city_location[child], city_location[node])
            ## get a city list
            frontier_list = frontier.queue
            frontier_city = []
            for i in range(len(frontier_list)):
                frontier_city.append(frontier_list[i][1])
            ##
            if child not in explored and child not in frontier_list:
                parent[child] = node
                g[child] = g[node]+ dist
                frontier.put([h[child], child])
                frontier_size = frontier_size + 1
            elif child in frontier_list:
                if g[child] < g[node]+dist:
                    parent[child] = node
                    frontier.put([h[child], child])
                    frontier_size = frontier_size + 1

    return None

#---------------------------------Visualization---------------------------------#
def draw_graph(city_graph, city_location, path):
    g = nx.Graph()    #empty graph
    for node in city_graph:
        g.add_node(node)
    count = 0
    edge_list = []
    for node in city_graph:
        for neighbor in city_graph[node]:
            if node in path and neighbor in path:
                edge_list.append((node,neighbor))
            if (node,neighbor) not in g.edges():
                g.add_edge(node, neighbor, weight = dist_cost(city_location[node],
city_location[neighbor]))
                count = count + 1

    pos = nx.spring_layout(g)
    arc_weight = nx.get_edge_attributes(g,'weight')
    nx.draw_networkx(g, pos, node_color = ['r' if not node in path else 'b' for node in g.nodes()],
edge_color = ['black' if not edge in edge_list else 'b' for edge in g.edges()])
```

```
    nx.draw_networkx_edge_labels(g, pos, edge_labels=arc_weight)
    plt.axis('off')
    plt.show()
    return count

count_branches = 0

bfs_time_complexity = 0
bfs_space_complexity = 0
bfs_running_time = 0
bfs_path_len = 0
bfs_solved = 0

dfs_time_complexity = 0
dfs_space_complexity = 0
dfs_running_time = 0
dfs_path_len = 0
dfs_solved = 0

ids_time_complexity = 0
ids_space_complexity = 0
ids_running_time = 0
ids_path_len = 0
ids_solved = 0

bestfs_time_complexity = 0
bestfs_space_complexity = 0
bestfs_running_time = 0
bestfs_path_len = 0
bestfs_solved = 0

Astar_time_complexity = 0
Astar_space_complexity = 0
Astar_running_time = 0
Astar_path_len = 0
Astar_solved = 0

for i in range(1):
    city_location = create_map() #create the map
    start_state = chr(ord('A') + random.randint(0,25))
    #need to test if goal_state == start_state
    goal_state = chr(ord('A') + random.randint(0,25))
    while goal_state == start_state:
        goal_state = chr(ord('A') + random.randint(0,25))

    print 'start_state:', start_state
    print'goal_state:', goal_state

    #run bfs
    city_graph = create_graph(city_location)
    bfs_start_time = time.time()
    bfs_result = bfs(start_state, goal_state, city_graph)   #bfs_result = [[path], # of node
explored]
    bfs_end_time = time.time()
```

```
    bfs_running_time = bfs_running_time + (bfs_end_time - bfs_start_time)
    bfs_path = None
    if bfs_result is not None:
        bfs_solved = bfs_solved + 1
        bfs_path = bfs_result[0]
        bfs_path_len = bfs_path_len + len(bfs_path)
        bfs_time_complexity = bfs_time_complexity + int(bfs_result[1])
        bfs_space_complexity = bfs_space_complexity + int(bfs_result[2])
    print 'bfs_path:', bfs_path

    #run dfs
    city_graph = create_graph(city_location)
    dfs_start_time = time.time()
    dfs_result = dfs(start_state, goal_state, city_graph)
    dfs_end_time = time.time()
    dfs_running_time = dfs_running_time + (dfs_end_time - dfs_start_time)
    dfs_path = None
    if dfs_result is not None:
        dfs_solved = dfs_solved + 1
        dfs_path = dfs_result[0]
        dfs_path_len = dfs_path_len + len(dfs_path)
        dfs_time_complexity = dfs_time_complexity + dfs_result[1]
        dfs_space_complexity = dfs_space_complexity + dfs_result[2]
    print 'dfs_path:', dfs_path

    #run ids
    city_graph = create_graph(city_location)
    ids_start_time = time.time()
    ids_result = ids(start_state, goal_state, city_graph)
    ids_end_time = time.time()
    ids_running_time = ids_running_time + (ids_end_time - ids_start_time)
    ids_path = None
    if ids_result is not None:
        ids_solved = ids_solved + 1
        ids_path = ids_result[0]
        ids_path_len = ids_path_len + len(ids_path)
        ids_time_complexity = ids_time_complexity + ids_result[1]
        ids_space_complexity = ids_space_complexity + ids_result[1]
    print 'ids_path:', ids_path

    #run best_first_search
    city_graph = create_graph(city_location)
    bestfs_start_time = time.time()
    staright_distance_togoal = cost(goal_state, city_location)
    bestfs_result = best_first_search(start_state, goal_state, city_graph,
staright_distance_togoal)
    bestfs_end_time = time.time()
    bestfs_running_time = bestfs_running_time + (bestfs_end_time - bestfs_start_time)
    bestfs_path = None
    if bestfs_result is not None:
        bestfs_solved = bestfs_solved + 1
        bestfs_path = bestfs_result[0]
        bestfs_path_len = bestfs_path_len + len(bestfs_path)
        bestfs_time_complexity = bestfs_time_complexity + bestfs_result[1]
```

```
        bestfs_space_complexity = bestfs_space_complexity + bestfs_result[2]
    print 'bestfs_path', bestfs_path

    #run A*
    city_graph = create_graph(city_location)
    Astar_start_time = time.time()
    estimate_staright_distance_togoal = cost(goal_state, city_location)
    Astar_result = A_star_search(start_state, goal_state, city_graph, city_location,
estimate_staright_distance_togoal)
    Astar_end_time = time.time()
    Astar_running_time = Astar_running_time + (Astar_end_time - Astar_start_time)
    Astar_path = None
    if Astar_result is not None:
        Astar_solved = Astar_solved + 1
        Astar_path = Astar_result[0]
        Astar_path_len = Astar_path_len + len(Astar_path)
        Astar_time_complexity = Astar_time_complexity + Astar_result[1]
        Astar_space_complexity = Astar_space_complexity + Astar_result[2]
    print 'Astar_path:', Astar_path

    city_graph = create_graph(city_location)
    print city_graph

    count_branches =  count_branches + draw_graph(city_graph, city_location, bfs_path)

#-----------------------Calculate the average and print the result------------------------------#
average_branches = count_branches/100
print average_branches

average_bfs_time_complexity = bfs_time_complexity/100
average_bfs_space_complexity = bfs_space_complexity/100
average_bfs_running_time = bfs_running_time/100
average_bfs_path_len = bfs_path_len/100
print 'average_bfs_time_complexity:', average_bfs_time_complexity
print 'average_bfs_space_complexity:', average_bfs_space_complexity
print 'average_bfs_running_time:', average_bfs_running_time
print 'average_bfs_path_len:', average_bfs_path_len
print 'bfs_solved:', bfs_solved

average_dfs_time_complexity = dfs_time_complexity/100
average_dfs_space_complexity = dfs_space_complexity/100
average_dfs_running_time = dfs_running_time/100
average_dfs_path_len = dfs_path_len/100
print 'average_dfs_time_complexity:', average_dfs_time_complexity
print 'average_dfs_space_complexity:', average_dfs_space_complexity
print 'average_dfs_running_time:', average_dfs_running_time
print 'average_dfs_path_len:', average_dfs_path_len
print 'dfs_solved:', dfs_solved

average_ids_time_complexity = ids_time_complexity/100
average_ids_space_complexity = ids_space_complexity/100
average_ids_running_time = ids_running_time/100
average_ids_path_len = ids_path_len/100
print 'average_ids_time_complexity:', average_ids_time_complexity
```

```
print 'average_ids_space_complexity:', average_ids_space_complexity
print 'average_ids_running_time:', average_ids_running_time
print 'average_ids_path_len:', average_ids_path_len
print 'ids_solved:', ids_solved

average_bestfs_time_complexity = bestfs_time_complexity/100
average_bestfs_space_complexity = bestfs_space_complexity/100
average_bestfs_running_time = bestfs_running_time/100
average_bestfs_path_len = bestfs_path_len/100
print 'average_bestfs_time_complexity:', average_bestfs_time_complexity
print 'average_bestfs_space_complexity:', average_bestfs_space_complexity
print 'average_bestfs_running_time:', average_bestfs_running_time
print 'average_bestfs_path_len:', average_bestfs_path_len
print 'bestfs_solved:', bestfs_solved

average_Astar_time_complexity = Astar_time_complexity/100
average_Astar_space_complexity = Astar_space_complexity/100
average_Astar_running_time = Astar_running_time/100
average_Astar_path_len = Astar_path_len/100
print 'average_Astar_time_complexity:', average_Astar_time_complexity
print 'average_Astar_space_complexity:', average_Astar_space_complexity
print 'average_Astar_running_time:', average_Astar_running_time
print 'average_Astar_path_len:', average_Astar_path_len
print 'Astar_solved:', Astar_solved
```