Introduction to Artificial Intelligence HW2

0711239 李勝維

Part 1: Breadth-first Search

```
edge_filename = "edges.csv"
def bfs(start, end):
    # generate graph
    graph = defaultdict(list)
    with open(edge_filename, newline="") as file:
        data = list(csv.reader(file))[1:]
        for row in data:
            addEdge(graph, int(row[0]), int(row[1]), float(row[2]))
```

First, I read the data from “edges.csv” and represent the graph using adjacency list data structure.

Details of adjacency list:

Given node u, graph[u] is a list contains tuples (v, w), where v represents neighbor of u and w represents the weight of the edge connecting them.

Details of utility “addEdge”:

```
    def addEdge(graph, u, v, w):
        graph[u].append((v, w))
```

Given directed edge (u, v) of weight w, add tuple (v,w) to list graph[u]

After reading all the data we need, we start the bfs search algorithm:

```
    # bfs
    visited = {start}  # using set
    queue = [start]
    parent = {}
    visited_num = 0
```

We first declare some variables that will be used in bfs:

visited: a set which stores all the nodes that have been added into “queue”

queue: a list that contains nodes that will be visited by bfs

parent: a dictionary that stores the parent of node on the bfs tree, this is for tracing the path found by bfs after the search

visited_num: a integer that stores the amount of nodes that have been visited

```
    while len(queue) > 0:
        u = queue.pop(0)
        visited_num += 1
        if u == end:
            break
        for v, _ in graph[u]:
            if v not in visited:
                visited.add(v)
                queue.append(v)
                parent[v] = u
    path, dist = trace(graph, start, end)
    return path, dist, visited_num
```

This is the main loop of bfs search algorithm.
In each iteration, we pop the first element in “queue” , u, and check if “u” is the goal(end). If not, we add all the neighbors of “u” that have not been visited into the queue and stores their parent.

After bfs terminates (we assume that there must be a path between start and end), we use function “trace” to trace the path and calculates the length of the path.
Here’s the details of function “trace”:

```
def trace(graph, start, end):
    path = [end]
    dist = 0.0
    while path[-1] != start:
        path.append(parent[path[-1]])  # add parent into list
        for v, w in graph[path[-1]]:  # add the weight of the edge
            if v == path[-2]:
                dist += w
    path.reverse()
    return path, dist
```

It simply adds the parent of the ending node into the list until the path reaches start node, and sums the weight of the edges.

Finally, bfs returns the path, the distance of the path and number of nodes that have been visited by the algorithm.

Part 2: Depth-first Search

Since the implementation of dfs will be almost identical to bfs, I will not give unnecessary details.

```
def dfs(start, end):
    # generate graph
    graph = defaultdict(list)
    with open(edge_filename, newline="") as file:
        data = list(csv.reader(file))[1:]
        for row in data:
            addEdge(graph, int(row[0]), int(row[1]), float(row[2]))
```

The first part of dfs does the exact same thing as bfs does: reads the data and represents the graph using adjacency list.

```
    # dfs
    visited = {start}  # using set
    stack = [start]
    parent = {}
    visited_num = 0
    while len(stack) > 0:
        u = stack.pop(-1)
        visited_num += 1
        if u == end:
            break
        for v, _ in graph[u]:
            if v not in visited:
                visited.add(v)
                stack.append(v)
                parent[v] = u
    path, dist = trace(graph, start, end)
    return path, dist, visited_num
```

This is the main loop of dfs search algorithm, the only difference between dfs and bfs is that dfs uses stack instead of queue to store nodes that will be visited later.

Finally, dfs returns the path, the distance of the path and number of nodes that have been visited by the algorithm.

Part 3: Uniform Cost Search

```
def ucs(start, end):
    # generate graph
    graph = defaultdict(list)
    with open(edge_filename, newline="") as file:
        data = list(csv.reader(file))[1:]
        for row in data:
            addEdge(graph, int(row[0]), int(row[1]), float(row[2]))
```

First, we read the data and represents the graph using adjacency list.

Note that the implementation of function “addEdge” is different from bfs and dfs:

```
    # position of w and v inversed due to PQ
    def addEdge(graph, u, v, w):
        graph[u].append((w, v))
```

The tuple stores (w, v) instead of (v,w) in bfs and dfs since we need to order the tuple by their weight in ucs algorithm, by swapping the order of v and w makes the code more consistent.

```
    # ucs
    visited = set()
    pq = PriorityQueue()
    pq.put((0, start))
    weighted_parent = {}
    visited_num = 0
```

Instead of queue or stack, we use priority queue in ucs.

(Note: “PriorityQueue” is from module “queue” in the python standard library)

The “weighted_parent” stores the parent of the node on the ucs tree, but unlike bfs or dfs, the parent of the node must be the one with the least cost, we will see this in the main loop.

Here is the main loop of ucs search algorithm:

```
    while not pq.empty():
        cost, u = pq.get()
        visited_num += 1
        if u == end:
            break
        if u not in visited:
            visited.add(u)
            for w, v in graph[u]:
                if v not in visited:
                    tmp_cost = cost + w
                    pq.put((tmp_cost, v))
                    if v not in weighted_parent:
                        weighted_parent[v] = (u, tmp_cost)
                    elif weighted_parent[v][1] > tmp_cost:
                        weighted_parent[v] = (u, tmp_cost)
    path, dist = trace(graph, start, end)
    return path, dist, visited_num
```

In each iteration, we first pop the element with least cost in priority queue, u, then we check if “u” is the goal. If not, we calculate the cost of its neighbors by adding the weight of the edge between them to the cost of “u”. Then we update the parent of v if the cost of v is smaller then the current one stored in “weighted_parent”. After all, we push v into the priority queue.

After ucs terminates, we trace the path and calculates the length of the path same as we do in bfs and dfs.

Finally, ucs returns the path, the distance of the path and number of nodes that have been visited by the algorithm.

Part 4: A* Search

```
def astar(start, end):
    # generate graph
    graph = defaultdict(list)
    with open(edge_filename, newline="") as file:
        data = list(csv.reader(file))[1:]
        for row in data:
            addEdge(graph, int(row[0]), int(row[1]), float(row[2]))
```

First, we read the data and represents the graph using adjacency list.

```
    # generate heuristic
    heuristic = {}
    with open(heuristic_filename, newline="") as file:
        data = list(csv.reader(file))
        n = None
        for idx, node in enumerate(data[0]):
            # check which column to read
            if node.isdigit() and end == int(node):
                n = idx
                break
        if not n:  # end node ID is not in first row of heuristic.csv
            raise BaseException("End node heuristic not found")
        for row in data[1:]:
            heuristic[int(row[0])] = float(row[n])
```

Second, we read the provided heuristic data, the program can read the correct column corresponding to different end node IDs.

```
    # a*
    visited = {start}
    pq = PriorityQueue()
    pq.put((0, start))
    parent = {}
    cost_G = {}
    cost_G[start] = 0
    visited_num = 0
```

Different from ucs, we use a python dictionary, “cost_G”, to store the cost of the node instead of getting the cost directly from the priority queue. The reason of this is the “cost” stored in the priority queue is the sum of cost and heuristic(f = g() + h()) in A* search, but we need the actual cost of the node(the “g()”).

Here is the main loop of A* search algorithm:

```
    while not pq.empty():
        _, u = pq.get()
        visited_num += 1
        if u == end:
            break
        for w, v in graph[u]:
            new_cost = cost_G[u] + w
            if v not in visited or new_cost < cost_G[v]:
                visited.add(v)
                cost_G[v] = new_cost
                tmp_cost = new_cost + heuristic[v]
                pq.put((tmp_cost, v))
                parent[v] = u

    path, dist = trace(graph, start, end)
    return path, dist, visited_num
```

The most part of main loop is identical to ucs, but there are three differences:

1. We get the cost of the node by “cost_G” instead of priority queue
2. Although the neighbor node, “v”, has been visited before, but if the calculated new cost is smaller then its current cost, we add it into the priority queue again. (This is similar to the part of ucs where we update the parent)
3. We add the heuristic term to the cost of neighbor nodes “v”, by doing so, we give the algorithm an “idea” of which node is “better” considering the goal.

Finally, A* returns the path, the distance of the path and number of nodes that have been visited by the algorithm.

Part 5: Test your implementation

ID1: From NYCU to Big City:

BFS:

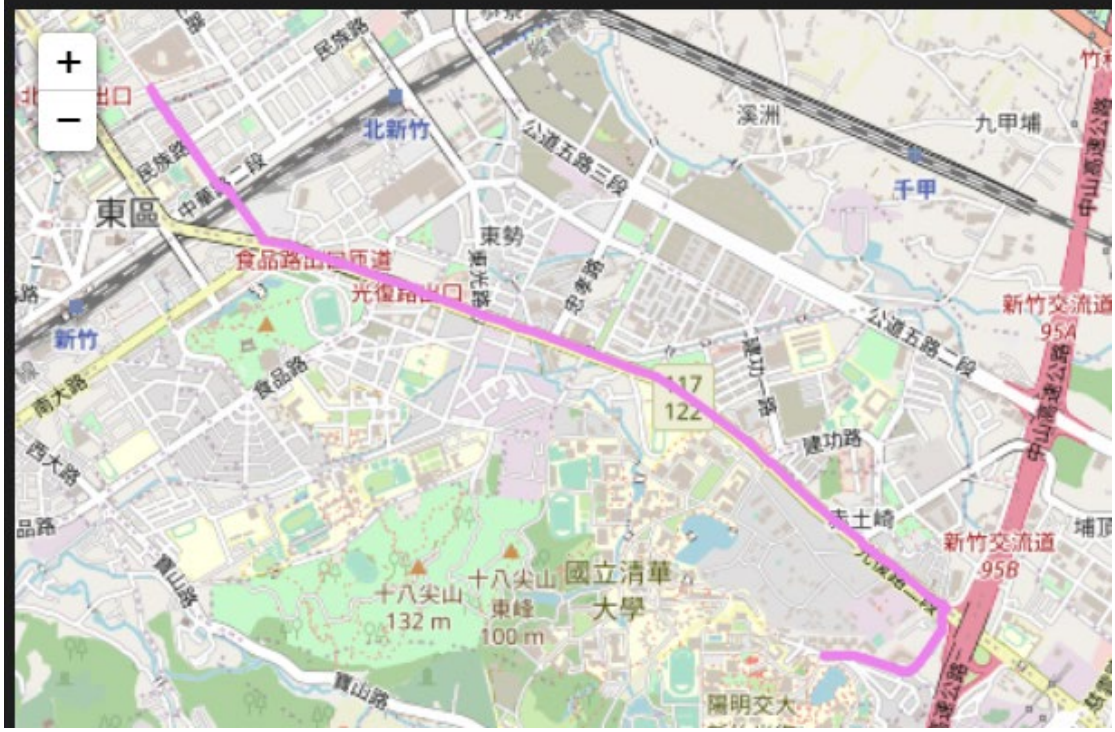
```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.881999999998 m
The number of visited nodes in BFS: 4274
```

DFS:

```
The number of nodes in the path found by DFS: 1718
Total distance of path found by DFS: 75504.3150000001 m
The number of visited nodes in DFS: 4712
```

UCS:

```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.8809999999985 m
The number of visited nodes in UCS: 5483
```



A*:

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.8809999999985 m
The number of visited nodes in A* search: 262
```

ID2: From Hsinchu Zoo to COSTCO:

BFS:

```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521000000001 m
The number of visited nodes in BFS: 4607
```

DFS:

```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.307999999895 m
The number of visited nodes in DFS: 9366
```

UCS:

```
The number of nodes in the path found by UCS: 63
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7854
```

A*:

```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 1245
```

ID3: From National Experimental High School to Nanliao Fishing Port

BFS:

```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11242
```

DFS:

```
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.993000000024 m
The number of visited nodes in DFS: 2248
```

UCS:

```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.413 m
The number of visited nodes in UCS: 12933
```

A*:

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413 m
The number of visited nodes in A* search: 7571
```

Discussions:

- Why DFS always performs the worst?
  DFS always visits the deepest unvisited node, which doesn't consider the depth nor the cost, thus it finds the "leftmost" path (depends on implementations). In other word, DFS isn't optimal.

- Why the path found by BFS has the least number of nodes?
  Although BFS doesn't consider the cost of the node, it always visits the shallowest unvisited node. Thus, it finds the path with least number of steps. In other word, BFS is only optimal on unweighted graph.

- UCS VS BFS:
  UCS is a generalization of BFS. Instead of visiting the shallowest unvisited node, UCS visits the least-cost unvisited node, which means that UCS cares about the total cost of the path instead of the number of steps. Thus, it is optimal on weighted graph as long as all the weights are non-negative.

- Why A* visits significantly less numbers of nodes compared to others?
  Since A* is an informed search algorithm, it considers not only the total cost of the path, it also considers the estimation of how close the goal is, which is the heuristic term. By adding the heuristic term, it will guide the direction of the search closer to the goal, which means that A* will not explore unnecessary nodes that are far from the goal, thus significantly reduces the number of visited nodes.
  A* also considers the total cost of the path, hence if the estimated heuristic is an admissible Heuristic, i.e. an underestimation of true value, A* is optimal.

Part 6: Search with a different objective

Before explaining my implementation, I want to explain my design on the heuristic function first.

I want to achieve two goals on the heuristic function:

The first goal is that A* still be optimal, in other word, the heuristic function must be an underestimation of the true heuristic.

The second goal is that I want to keep A* benefit from the heuristic, which means that the order of magnitude must be the same for the total cost and the heuristic, e.g., if the heuristic always equals to zero, although A* will still be an optimal algorithm, but will not benefit from the heuristic term and thus degrade to a UCS algorithm.

Considering all the above, my heuristic function looks like this:

$$h(x) = \frac{straight - line\ distance(x)}{maximum\ speed\ limit\ in\ the\ map}$$

Since I divide the straight-line distance by the maximum speed limit in the map, this will never overestimate the true heuristic. Plus, the order of magnitude will be roughly the same on the heuristic term and the total cost of the path.

Here is my implementation on finding the fastest path using A*:

```
def astar_time(start, end):
    # generate graph
    graph = defaultdict(list)
    max_speed_limit = 0.0
    with open(edge_filename, newline="") as file:
        data = list(csv.reader(file))[1:]
        for row in data:
            addEdge(graph, int(row[0]), int(row[1]),
                    float(row[2]), float(row[3]))
            max_speed_limit = max(max_speed_limit, float(row[3]))

    straight_line_dist = {}
    with open(heuristic_filename, newline="") as file:
        data = list(csv.reader(file))
        n = None
        for idx, node in enumerate(data[0]):
            if node.isdigit() and end == int(node):
                n = idx
                break
        if not n:
            raise BaseException("End node heuristic not found")
        for row in data[1:]:
            straight_line_dist[int(row[0])] = float(row[n])
```

Same as before, I read the data for the graph and the straight-line distance. Also, I save the maximum speed limit in the map for calculating heuristic.

```
    # a*
    visited = {start}
    pq = PriorityQueue()
    pq.put((0, start))
    parent = {}
    cost_G = {}
    cost_G[start] = 0
    visited_num = 0

    while not pq.empty():
        _, u = pq.get()
        visited_num += 1
```

```
        if u == end:
            break
        for s, w, v in graph[u]:
            new_cost = cost_G[u] + w/s
            if v not in visited or new_cost < cost_G[v]:
                visited.add(v)
                cost_G[v] = new_cost
                tmp_cost = new_cost + get_heuristic(v)
                pq.put((tmp_cost, v))
                parent[v] = u

    path, dist = trace(graph, start, end)
    return path, dist, visited_num
```

This is similar to the above, the differences are I calculate the cost by dividing the weight by the speed limit, and I've introduced a new heuristic function:

```
    def get_heuristic(id):
        return straight_line_dist[id] / max_speed_limit
```

I calculate the heuristic by formula:

$$h(x) = \frac{straight - line\ distance(x)}{maximum\ speed\ limit\ in\ the\ map}$$

Finally, A* returns the path, the travel time and number of nodes that have been visited by the algorithm.

Results:

ID1: From NYCU to Big City:

```
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083164 s
The number of visited nodes in A* search: 2016
```

ID2: From Hsinchu Zoo to COSTCO:

```
The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.44366343603014 s
The number of visited nodes in A* search: 2955
```

ID3: From National Experimental High School to Nanliao Fishing Port

When going from National Experimental High School to Nanliao Fishing Port, we can see that the fastest path is different from the shortest path.

Validation:

Does my heuristic function work?

At the start of part 6, I said that keeping the order of magnitude same for the heuristic and the total cost will keep A* benefit from it, I will validate this through different heuristic functions.

I will test these three different heuristic functions for following experiments:

$$h_{ori}(x) = \frac{straight - line\ distance(x)}{maximum\ speed\ limit\ in\ the\ map}$$

$$h_{over}(x) = straight - line\ distance(x)$$

$$h_{zero}(x) = 0$$

Which represents original, overestimate and zero respectively

ID1	h_{ori}	h_{zero}	h_{over}
Find the fastest path?	True	True	False
Number of visited nodes	2016	5263	142

ID2	h_{ori}	h_{zero}	h_{over}
Find the fastest path?	True	True	False
Number of visited nodes	2955	7732	84

ID3	h_{ori}	h_{zero}	h_{over}
Find the fastest path?	True	True	False
Number of visited nodes	8727	11920	355

We can see that since overestimated heuristic cannot keep A* optimal, h_{over} fails to find the fastest path in every case.
Also, h_{zero} tends to visit more nodes then h_{ori} , this tells us that my design of heuristic did benefit the algorithm.

In conclusion, I think my design of heuristic function did work!

Problems I meet and how I solve them:

- Problem: UCS always returns a wrong path, but the cost is correct.
  Solution: I didn't update the parent base on the cost.
- Problem: A* doesn't find the optimal solution.
  Solution: Even neighbor node "v" has been visited, I need to update it again if there is another path with less cost reaches it.