**Report:**My code are able to do every with in complition bit. I did use a star search to find the fastest route of two nodes, I found a way to specify start and goal nodes and used appropriate heuristic and used right cost measure which g = g\* + edge.weight and f = g + h(neigh).

It Has a priority queue of appropriate elements and Correctly selects shortest paths.All roads are printed out correctly with no duplicates, the length of them and the total length are also printed. I highlighted shortest path as well. Also I tried bit of challenge part , I did add GUI of distance and time but it did not work .

two appropriate heuristic are:1. The estimated cost to goal is never greater than the true cost (admissible heuristic)

2. For each node, the first expand always has the minimal cost from start (consistent/monotonic heuristic)

cost measure are:

g = g\* + edge.weigh

f = g + h(neigh);

Minimum:

**A\* search pseudocode:**

input : Node start ,Node goal

for n :graph.nodes:

set n unvisited

make n’s previous node null

create new Fringe object first= (node,prev,cost so far,estimated distance to end)//the first element

create new priorityQueue fringe

fringe.offer(first)

while (fringe is not empty){

poll the first element out from fringe

if (node is unvisted){

Set node\* as visited, and set node\*.prev = prev\*;

}

if (node\* is the target node) break;

for (edge = (node\*, neigh) outgoing from node\*) {

if (neigh is unvisited) {

g = g\* + edge.weight;

f = g + h(neigh);

add a new element into the fringe;

}

}

create new list route to store nodes for shortest route

create a temp node equal to goal

while(temp != start){

route.add temp

temp = temp.previousnode

}

add start to route as well

create new list route\_seg to store segments for shortest route

for each nodes n in route{

route\_seg.add(n.findSegment);

}

create a hashset road to store roads for shortest route

for each segment s in route\_seg{

road.add(s.road)

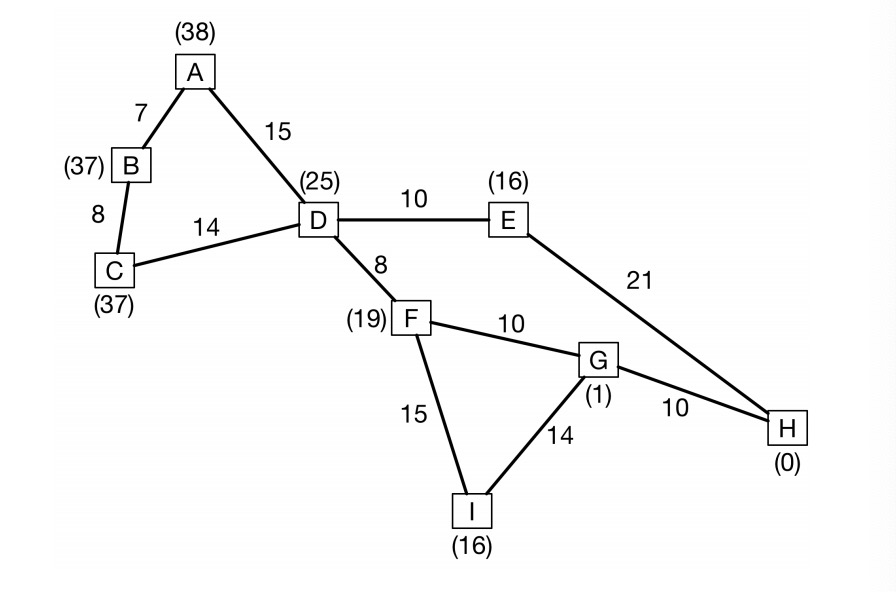
}

print road.name in hashset road out

}

Core:

**Question:**



1,A\* Search

(1)

Step 0: Fringe elements: {{D, null, 0, 25} Element to visit next:{ D, null, 0, 25}

Step 1:Fringe elements：{{E,D,10,26},{F,D,8,27},{C,D,14,51},{A,D,15,53} Element to visit next:{E,D,10,26}

Step 2:Fringe elements：{{F,D,8,27},{H,E,31,31},{C,D,14,51},{A,D,15,53} Element to visit next:{F,D,8,27}

Step 3:Fringe elements：{{G,F,18,19},{H,E,31,31},{I,F,23,39},{C,D,14,51},{A,D,15,53} Element to visit next:{G,F,18,19}

Step 4:Fringe elements：{{H,G,28,28},{H,E,31,31},{I,F,23,39},{C,D,14,51},{A,D,15,53} Element to visit next:{H,G,28,28}(reached the goal , break)

(2) the final shortest path as a sequence of nodes by A\* search: D-------F-------G------H

2: 1-to-1 Dijkstra’s algorithm

(1)

Step 0: Fringe elements: {D, null, 0} Element to visit next: {D, null, 0}

Step 1: Fringe elements: {F,D,8},{E,D, 10},{C,D,14},{A,D,15} Element to visit next: {F,D,8}

Step 2:Fringe elements: {E,D, 10},{C,D,14},{A,D,15} ,{G,F,18},{I,F,23} Element to visit next:{E,D, 10}

Step 3:Fringe elements: {C,D,14},{A,D,15} ,{G,F,18},{I,F,23},{H,E,31} Element to visit next: {C,D,14}

Step 4:Fringe elements:{A,D,15},{G,F,18},{B**,C**,22},{I,F,23},{H,E,31} Element to visit next: {A,D,15}

Step 5:Fringe elements:{G,F,18},{B**,**C,22},{A,C,22},{I,F,23},{H,E,31} Element to visit next: {G,F,18}

Step 6:Fringe elements:{B**,**C,22},{B,A,22},{I,F,23},{H,G,28},{H,E,31} Element to visit next: {B**,**C,22}

Step 7:Fringe elements:{I,F,23},{H,G,28},{H,E,31} Element to visit next: {I,F,23}(node B has already visited so we will not visit node B and element{B,A,22} should be poll out of the fringe)

Step 7:Fringe elements:{H,G,28},{H,E,31},{G,I,37} Element to visit next: {H,G,28}

(reached the goal node , return)

(element next has already visited ,we searched every elements , a \* search finished)

(2) the final shortest path as a sequence of nodes: D-----F----G----H

3:The reason A\* search takes fewer steps than 1-to-1 Dijkstra’s algorithm is because A\* search uses best-first-search to speed things up. A\* search expand based on f(node) which is adding cost so far and estimated cost to the end but Dijkstra only expands the minimum cost so far. so the A\* is basically an advanced variation of Dijkstra that why A\* search is faster than Dijkstra’s algorithm.