ANSWERS

A5.9 Project: plan the route, then drive it

Algorithms and complexity · A level · OCR H446 2.3.1, AQA 7517 4.3.6.1, Eduqas A500QS 1.3 · about 35 min

What this lesson is about

Choose and justify a route planner, build the mat's graph, find the shortest route with Dijkstra's algorithm and drive it.

- The map as data
- Choosing the algorithm
- Driving a route accurately
- Putting it together

Questions

5 marks in all

1. On the project's map, why would breadth-first search choose the wrong route from S to G? [1 mark]

- ☐ **A** It finds the route with the fewest roads, which is not the shortest distance when roads have different lengths
- ☐ **B** It cannot search undirected graphs
- ☐ **C** It always walks into dead ends
- ☐ **D** It needs a heuristic

> **Answer:** A. S, A, B, G has 3 roads but is 190 cm; the 5-road route is 130 cm.

2. Why can a greedy rule, always driving to the nearest next waypoint, fail? [1 mark]

- ☐ **A** It only looks one road ahead, so it can take a short road into a dead end
- ☐ **B** It is too slow on small maps
- ☐ **C** It always takes the longest road
- ☐ **D** It needs the roads sorted first

> **Answer:** A. From E the nearest waypoint is the dead end J.

3. What is the length in cm of the shortest route from S to G on the project's map? [1 mark]

> **Answer:** 130. S, C, D, E, F, G: 25 + 25 + 30 + 25 + 25 = 130.

4. What does this print? It builds part of the project's graph. [1 mark]

```
import math
WAYPOINTS = {"S": (10, 10), "C": (35, 10), "D": (35, 35)}
ROADS = [("S", "C"), ("C", "D")]
graph = {w: {} for w in WAYPOINTS}
for a, b in ROADS:
    graph[a][b] = graph[b][a] = math.dist(WAYPOINTS[a], WAYPOINTS[b])
print(graph["C"])
```

> **Answer:**
>
> ```
> {'S': 25.0, 'D': 25.0}
> ```
>
> C is joined to both S and D, each 25 cm away, because every road is added to both of its ends.

5. Trying every possible order of visiting n waypoints grows as n!. Which statement is right? [1 mark]

- ☐ **A** It quickly becomes impractical, while Dijkstra's algorithm stays polynomial
- ☐ **B** It is $O(n^2)$, like bubble sort
- ☐ **C** It is O(log n)
- ☐ **D** It is faster than Dijkstra's algorithm for large n

> **Answer:** A. n! grows faster than any exponential of a fixed base; Dijkstra's algorithm is O((V + E) log V) with a priority queue.

The task: plan the route, then drive it

The starter has the map and the `go_to` helper. Write the rest: - Build an undirected weighted graph from `WAYPOINTS` and `ROADS`, where each road's weight is its length in cm (use `math.dist`; do not type the lengths). - Write `dijkstra(graph, start)` and use it to find the shortest route from `S` to `G`. The program must find the route: do not type it. - Print `route: <waypoints joined with ->`, such as `route: S-X-Y-G`, and `length: <n> cm`, with n a whole number. - Drive the route by calling `go_to` for each waypoint after `S`, in order. Stay on the roads: do not go near waypoints that are not on your route, and do not touch a wall. - When the robot reaches `G`, turn the LED green.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

import math
import heapq

WAYPOINTS = {"S": (10, 10), "A": (10, 90), "B": (90, 90), "C": (35, 10), "D": (35, 35),
             "E": (65, 35), "F": (65, 60), "G": (90, 60), "J": (80, 35)}
ROADS = [("S", "A"), ("A", "B"), ("B", "G"), ("S", "C"), ("C", "D"),
         ("D", "E"), ("E", "F"), ("F", "G"), ("E", "J")]
START = WAYPOINTS["S"]              # the robot starts at S

def go_to(x, y):
    """Drive to the mat point (x, y): along y, then along x, twice so the second pass
corrects any drift."""
    for attempt in range(2):
        px, py = position()
        dy = y - (START[1] + py)
        if dy > 1:
            forward(60, distance=dy)
        elif dy < -1:
            backward(60, distance=-dy)
        px, py = position()
        dx = x - (START[0] + px)
        if dx > 1:
            right(60, distance=dx)
        elif dx < -1:
            left(60, distance=-dx)
```

The hint students can ask for: Make an empty neighbour dictionary for every waypoint, then add each road to both of its ends with its length. Run Dijkstra's algorithm from S, follow the previous waypoints back from G and reverse them. Print the route, then go_to each waypoint after S in that order, and light the LED at the end.

A solution

```
from bugbot import *
connect()

import math
import heapq

WAYPOINTS = {"S": (10, 10), "A": (10, 90), "B": (90, 90), "C": (35, 10), "D": (35, 35),
             "E": (65, 35), "F": (65, 60), "G": (90, 60), "J": (80, 35)}
ROADS = [("S", "A"), ("A", "B"), ("B", "G"), ("S", "C"), ("C", "D"),
         ("D", "E"), ("E", "F"), ("F", "G"), ("E", "J")]
```

```
START = WAYPOINTS["S"]

def go_to(x, y):
    for attempt in range(2):
        px, py = position()
        dy = y - (START[1] + py)
        if dy > 1:
            forward(60, distance=dy)
        elif dy < -1:
            backward(60, distance=-dy)
        px, py = position()
        dx = x - (START[0] + px)
        if dx > 1:
            right(60, distance=dx)
        elif dx < -1:
            left(60, distance=-dx)

graph = {name: {} for name in WAYPOINTS}
for a, b in ROADS:
    length = math.dist(WAYPOINTS[a], WAYPOINTS[b])
    graph[a][b] = length
    graph[b][a] = length

def dijkstra(graph, start):
    distance = {v: float("inf") for v in graph}
    previous = {v: None for v in graph}
    distance[start] = 0
    visited = set()
    queue = [(0, start)]
    while queue:
        d, u = heapq.heappop(queue)
        if u in visited:
            continue
        visited.add(u)
        for v, w in graph[u].items():
            if v not in visited and d + w < distance[v]:
                distance[v] = d + w
                previous[v] = u
                heapq.heappush(queue, (d + w, v))
    return distance, previous

distance, previous = dijkstra(graph, "S")
route, v = [], "G"
while v is not None:
    route.append(v)
    v = previous[v]
route.reverse()
print("route: " + "-".join(route))
print("length:", round(distance["G"]), "cm")
for name in route[1:]:
    go_to(*WAYPOINTS[name])
led("green")
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.