ANSWERS

# A4.8 Project: plan the route

Trees and graphs · A level · OCR H446 1.4.2, AQA 7517 4.2.4.1, Eduqas A500QS 1.1 · about 35 min

## What this lesson is about

Model the mat as a graph, find the shortest route with breadth-first search, and drive it.

- The brief
- The plan
- Step 1: from grid to graph
- Step 2: search, then read the route
- Step 3: drive it
- Testing

## Questions

**5 marks in all**

**1.** The robot must reach a square in the fewest moves, and every move costs the same. Why use breadth-first rather than depth-first search? [1 mark]

- ☐ **A** Breadth-first finds a shortest path in an unweighted graph; depth-first may not
- ☐ **B** Depth-first cannot find any route at all
- ☐ **C** Breadth-first uses less memory
- ☐ **D** Depth-first only works on trees

> **Answer:** A. Depth-first finds a route, but it can be much longer than necessary.

**2.** Why is an adjacency list a good choice for a grid of squares? [1 mark]

- ☐ **A** Each square has at most four neighbours, so the graph is sparse
- ☐ **B** A grid is a dense graph
- ☐ **C** An adjacency matrix cannot store a grid
- ☐ **D** Adjacency lists answer 'are these two squares joined?' in one look

> **Answer:** A. Most of an adjacency matrix for a grid would be empty cells.

**3.** A 5 by 5 grid of squares has no blocked squares. Each square is joined to the squares directly above, below, left and right of it. How many edges does the graph have? [1 mark]

> **Answer:** 40. Each of the 5 rows has 4 edges across, and each of the 5 columns has 4 edges up and down: 20 + 20.

**4.** What does this program print? [1 mark]

```
from collections import deque
GRAPH = {"S": ["A", "B"], "A": ["S", "C"], "B": ["S", "D"], "C": ["A", "G"], "D": ["B"],
"G": ["C"]}
parent = {"S": None}
queue = deque(["S"])
while queue:
    v = queue.popleft()
    for n in GRAPH[v]:
        if n not in parent:
            parent[n] = v
            queue.append(n)
route = []
v = "G"
while v is not None:
    route.append(v)
    v = parent[v]
print(" ".join(reversed(route)))
```

**Answer:**

```
S A C G
```

The parents lead back from G to C, A and S; reversing gives the route from the start.

**5.** After a breadth-first search records each vertex's parent, what shape do the parent links form? [1 mark]

- ☐ **A** A tree rooted at the start vertex
- ☐ **B** A cycle through every vertex
- ☐ **C** A complete graph
- ☐ **D** A queue

**Answer:** A. Every vertex except the start has exactly one parent, and there are no cycles, so there is one path from the root to each vertex.

## The task: plan the route

`GRID` is the mat from the brief, with row 0 at the far end of the mat and column 0 on the left, and `NAMES` gives the letters in reading order. The robot starts in the centre of square A facing up the mat, and square centres are 25 cm apart. Build an adjacency list for the free squares. Print `vertices:` and the number of free squares, then `edges:` and the number of edges. Find the route from A to O with the fewest moves, using a breadth-first search that takes squares from the front of a queue with `popleft()` or `pop(0)`. Print `route:` followed by the letters of the squares from A to O separated by single spaces, and then `moves:` followed by the number of moves. Finally drive the route square by square, without touching a blocked square, finishing in O.

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

from collections import deque

GRID = ["....",
        ".##.",
        "..#.",
        "#..."]
NAMES = "ABCDEFGHIJKLMNOP"

graph = {}
```

**The hint students can ask for:** Build the adjacency list first: each free square is joined to the free squares directly above, below, left and right of it. Breadth-first search from A, remembering for each zone the zone it was discovered from. Then walk those links back from O to A and reverse them to get the route.

## A solution

```
from bugbot import *
connect()
from collections import deque

GRID = ["....",
        ".##.",
        "..#.",
        "#..."]
NAMES = "ABCDEFGHIJKLMNOP"

def name(r, c):
    return NAMES[r * 4 + c]

graph = {}
for r in range(4):
    for c in range(4):
        if GRID[r][c] == ".":
            graph[name(r, c)] = []
            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nr, nc = r + dr, c + dc
                if 0 <= nr < 4 and 0 <= nc < 4 and GRID[nr][nc] == ".":
                    graph[name(r, c)].append(name(nr, nc))
print("vertices:", len(graph))
print("edges:", sum(len(v) for v in graph.values()) // 2)
```

```
def shortest_route(graph, start, goal):
    came_from = {start: None}
    queue = deque([start])
    while queue:
        zone = queue.popleft()
        if zone == goal:
            break
        for nxt in graph[zone]:
            if nxt not in came_from:
                came_from[nxt] = zone
                queue.append(nxt)
    route = []
    zone = goal
    while zone is not None:
        route.append(zone)
        zone = came_from[zone]
    route.reverse()
    return route

route = shortest_route(graph, "A", "O")
print("route:", " ".join(route))
print("moves:", len(route) - 1)

def centre(zone):
    i = NAMES.index(zone)
    return (i % 4) * 25, -(i // 4) * 25

for zone in route[1:]:
    x, y = position()
    tx, ty = centre(zone)
    if tx > x + 2:
        right(60, distance=tx - x)
    elif tx < x - 2:
        left(60, distance=x - tx)
    if ty > y + 2:
        forward(60, distance=ty - y)
    elif ty < y - 2:
        backward(60, distance=y - ty)
```

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