ANSWERS

# A4.4 Breadth-first traversal

Trees and graphs · A level · OCR H446 2.3.1, AQA 7517 4.3.1.1 · about 30 min

## What this lesson is about

Level by level with a queue, shortest paths in unweighted graphs, and a breadth-first visit of the mat's zones.

- The idea: a queue
- Shortest paths in an unweighted graph
- What breadth-first traversal is for
- Depth-first or breadth-first?
- The mat as a graph

## Questions

**5 marks in all**

**1.** Which data structure does breadth-first traversal use? [1 mark]

- ☐ **A** A queue
- ☐ **B** A stack
- ☐ **C** A binary search tree
- ☐ **D** A two-dimensional array

> **Answer:** A. Vertices are visited in the order they were discovered: first in, first out.

**2.** In which kind of graph is breadth-first traversal guaranteed to find a shortest path? [1 mark]

- ☐ **A** An unweighted graph
- ☐ **B** Any weighted graph
- ☐ **C** Only a tree
- ☐ **D** Only a directed graph

> **Answer:** A. It counts edges. With weights, a route with more edges can cost less, which needs Dijkstra's algorithm.

**3.** An undirected graph has the edges A-B, A-C, B-D and C-E. Put the vertices in the order a breadth-first traversal from A visits them, taking neighbours in alphabetical order. [1 mark]

Number the lines 1 to 5 to put them in the right order.

___ `C`
___ `E`
___ `B`
___ `D`
___ `A`

**Answer:**

```
A
B
C
D
E
```

B and C are one edge from A, so both come before D and E, which are two edges away.

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

```
from collections import deque
GRAPH = {"A": ["B", "C"], "B": ["A", "D"], "C": ["A", "D"], "D": ["B", "C", "E"], "E":
["D"]}
dist = {"A": 0}
queue = deque(["A"])
while queue:
    v = queue.popleft()
    for n in GRAPH[v]:
        if n not in dist:
            dist[n] = dist[v] + 1
            queue.append(n)
print(dist["D"], dist["E"])
```

**Answer:**

```
2 3
```

D is two edges from A (through B) and E is one more. The first time a vertex is discovered gives its shortest distance.

**5.** Which statements about breadth-first and depth-first traversal are true? [1 mark]

Tick every answer that is true.

- ☐ **A** Breadth-first visits all vertices one edge from the start before any two edges away
- ☐ **B** Depth-first traversal can be written recursively
- ☐ **C** Both visit every vertex reachable from the start exactly once
- ☐ **D** Depth-first traversal always finds the shortest path

**Answer:** A, B, C. Depth-first finds a path, but not necessarily the shortest one.

## The task: a breadth-first visit

The mat's zones and tracks are in `GRAPH`, an adjacency list with each zone's neighbours in alphabetical order. `CENTRE` and `go_to(zone)` are written for you. Write a breadth-first traversal from zone A that uses a queue, taking zones from the front with `popleft()` or `pop(0)`. Print one line, `BFS order:` followed by the zones in the order they come off the queue, separated by single spaces. Then drive the robot to each zone in that order with `go_to`. The robot starts in zone A.

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

from collections import deque

# where each zone's centre is, in cm from the start (the robot starts in the middle of zone
A)
CENTRE = {"A": (0, 0), "B": (30, 0), "C": (60, 0), "D": (0, -30), "E": (30, -30), "F": (60,
-30),
          "G": (0, -60), "H": (30, -60), "I": (60, -60)}

GRAPH = {
    "A": ["B", "D"],
    "B": ["A", "C", "E"],
    "C": ["B"],
    "D": ["A", "G"],
    "E": ["B", "F", "H"],
    "F": ["E", "I"],
    "G": ["D", "H"],
    "H": ["E", "G"],
    "I": ["F"],
}

def go_to(zone):
    """Drive sideways then forwards or backwards to the centre of a zone."""
    x, y = position()
    tx, ty = CENTRE[zone]
    if tx > x + 1:
        right(80, distance=tx - x)
    elif tx < x - 1:
        left(80, distance=x - tx)
    if ty > y + 1:
        forward(80, distance=ty - y)
    elif ty < y - 1:
        backward(80, distance=y - ty)
```

**The hint students can ask for:** Put A in a queue and mark it discovered. Repeatedly take the zone at the front, visit it, and add each neighbour not yet discovered to the back. The queue decides the order; the robot just drives to each zone as it comes off the front.

### A solution

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

# where each zone's centre is, in cm from the start (the robot starts in the middle of zone
A)
```

```
CENTRE = {"A": (0, 0), "B": (30, 0), "C": (60, 0), "D": (0, -30), "E": (30, -30), "F": (60,
-30),
          "G": (0, -60), "H": (30, -60), "I": (60, -60)}

GRAPH = {
    "A": ["B", "D"],
    "B": ["A", "C", "E"],
    "C": ["B"],
    "D": ["A", "G"],
    "E": ["B", "F", "H"],
    "F": ["E", "I"],
    "G": ["D", "H"],
    "H": ["E", "G"],
    "I": ["F"],
}

def go_to(zone):
    """Drive sideways then forwards or backwards to the centre of a zone."""
    x, y = position()
    tx, ty = CENTRE[zone]
    if tx > x + 1:
        right(80, distance=tx - x)
    elif tx < x - 1:
        left(80, distance=x - tx)
    if ty > y + 1:
        forward(80, distance=ty - y)
    elif ty < y - 1:
        backward(80, distance=y - ty)

def bfs(graph, start):
    order = []
    discovered = [start]
    queue = deque([start])
    while queue:
        zone = queue.popleft()
        order.append(zone)
        for nxt in graph[zone]:
            if nxt not in discovered:
                discovered.append(nxt)
                queue.append(nxt)
    return order

order = bfs(GRAPH, "A")
print("BFS order:", " ".join(order))
for zone in order:
    go_to(zone)
```

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