ANSWERS

A5.7 Dijkstra's shortest path algorithm

Algorithms and complexity · A level · OCR H446 2.3.1, AQA 7517 4.3.6.1, Eduqas A500QS 1.3 · about 30 min

What this lesson is about

Tracing Dijkstra's algorithm in a table, why negative weights break it, a priority queue version, its efficiency and applications.

- The algorithm
- A trace
- Why it works, and when it does not
- In Python
- Efficiency
- Where it is used

Questions

5 marks in all

1. In Dijkstra's algorithm, which vertex is visited next? [1 mark]

- ☐ **A** The unvisited vertex with the smallest distance so far
- ☐ **B** The neighbour joined by the lightest edge
- ☐ **C** The next vertex in alphabetical order
- ☐ **D** The vertex with the most edges

> **Answer:** A. Taking the closest unvisited vertex is what makes its distance final.

2. Why can Dijkstra's algorithm give a wrong answer if a graph has a negative edge weight? [1 mark]

- ☐ **A** A vertex's distance is fixed when it is visited, but a later negative edge could make a shorter route to it
- ☐ **B** It cannot store negative numbers
- ☐ **C** It loops forever
- ☐ **D** It only works on directed graphs

> **Answer:** A. The algorithm assumes going through a further vertex can only add distance, which a negative weight breaks.

3. In the lesson's graph (A-B 4, A-C 2, B-C 1, B-D 5, C-D 8, C-E 10, D-E 2, D-F 6, E-F 3), what is the shortest distance from A to E? [1 mark]

> **Answer:** 10. A, C, B, D, E: 2 + 1 + 5 + 2 = 10.

4. What does this print? It reads a route back from a table of previous vertices. [1 mark]

```
previous = {"A": None, "B": "C", "C": "A", "D": "B", "E": "D"}
path = []
v = "E"
while v is not None:
    path.append(v)
    v = previous[v]
print("-".join(reversed(path)))
```

Answer:

```
A-C-B-D-E
```

E came from D, D from B, B from C, C from A; reversed that is A-C-B-D-E.

5. Which are applications of shortest path algorithms such as Dijkstra's? [1 mark]

Tick every answer that is true.

- ☐ **A** Satellite navigation route planning
- ☐ **B** Routing packets between routers (for example OSPF)
- ☐ **C** Planning a robot's route between waypoints
- ☐ **D** Compressing an image

Answer: A, B, C. All three find a lowest-cost route through a weighted graph. Compression is a different problem.

The task: the shortest route table

`roads` in the starter is the graph above as a dictionary of dictionaries (an adjacency list with weights). Write `dijkstra(graph, start)` that returns two dictionaries, the shortest distance to each vertex and the previous vertex on that route (`None` for the start). Run it from `A` and print one line per vertex, in alphabetical order, in exactly this form, `<vertex>: <distance> from <previous vertex>`, for example: `X: 12 from Y` with `A: 0 from -` for the start. Then follow the previous vertices back from `G` and print the route and its length: `route A to G: <vertices joined with ->, cost <n>` such as `route A to G: A-X-Y-G, cost 20`. The program must find the route: do not type it.

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

roads = {
    "A": {"B": 7, "C": 3},
    "B": {"A": 7, "C": 2, "D": 4},
    "C": {"A": 3, "B": 2, "D": 8, "E": 6},
    "D": {"B": 4, "C": 8, "E": 1, "F": 5},
    "E": {"C": 6, "D": 1, "F": 9, "G": 12},
    "F": {"D": 5, "E": 9, "G": 2},
    "G": {"E": 12, "F": 2},
}
```

The hint students can ask for: Start every distance at infinity and the start at 0. Repeatedly take the unvisited vertex with the smallest distance, and for each unvisited neighbour see whether going through it is shorter; if it is, record the new distance and where it came from. To get the route, start at G and keep following the previous vertex until there is none, then reverse.

A solution

```
from bugbot import *
connect()

import heapq

roads = {
    "A": {"B": 7, "C": 3},
    "B": {"A": 7, "C": 2, "D": 4},
    "C": {"A": 3, "B": 2, "D": 8, "E": 6},
    "D": {"B": 4, "C": 8, "E": 1, "F": 5},
    "E": {"C": 6, "D": 1, "F": 9, "G": 12},
    "F": {"D": 5, "E": 9, "G": 2},
    "G": {"E": 12, "F": 2},
}

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(roads, "A")
for v in sorted(roads):
    print(v + ":", distance[v], "from", previous[v] or "-")
path, v = [], "G"
while v is not None:
    path.append(v)
    v = previous[v]
path.reverse()
print("route A to G: " + "-".join(path) + ", cost", distance["G"])
```

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