ANSWERS

U9.3 Dijkstra and the cost of a step

Planning · University · about 35 min

What this lesson is about

When steps cost different amounts, the cheapest route is not the shortest one, and a queue sorted by cost finds it.

- The algorithm
- A cost map with clearance in it
- Uniform cost search, and the name

Questions

6 marks in all

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

```
import heapq
EDGES = {"S": [("A", 1), ("B", 2)], "A": [("G", 5)], "B": [("C", 2)], "C": [("G", 1)], "G":
[]}
best, came, done = {"S": 0}, {"S": None}, set()
queue = [(0, "S")]
while queue:
    cost, cur = heapq.heappop(queue)
    if cur in done:
        continue
    done.add(cur)
    if cur == "G":
        break
    for nxt, w in EDGES[cur]:
        price = cost + w
        if price < best.get(nxt, 1e9):
            best[nxt] = price
            came[nxt] = cur
            heapq.heappush(queue, (price, nxt))
path, n = [], "G"
while n is not None:
    path.append(n)
    n = came[n]
print(best["G"], "".join(reversed(path)))
```

Answer:

```
5 SBCG
```

G is first pushed via A at cost 6, then improved via B and C to 5, and popped at 5. Breadth first would return S A G, the fewest edges, at cost 6.

2. In Dijkstra's algorithm, when is a node's cost final? [1 mark]

- ☐ **A** The first time it is popped off the priority queue
- ☐ **B** The first time it is pushed onto the priority queue
- ☐ **C** When all of its neighbours have been pushed
- ☐ **D** When the goal is reached

> **Answer:** A. A node can be pushed several times with different costs, and only the smallest is correct. With non-negative edges nothing popped later can be cheaper.

3. Why does Dijkstra's algorithm give wrong answers on a graph with a negative edge cost? [1 mark]

- ☐ **A** A node popped as final could later be reached more cheaply through the negative edge
- ☐ **B** The priority queue cannot store negative numbers
- ☐ **C** Negative costs make the graph disconnected
- ☐ **D** The lazy deletion trick removes the negative edges

> **Answer:** A. Correctness rests on nothing later in the queue being cheaper than what has been popped. A negative edge breaks exactly that.

4. With the U9.3 cost map, a robot takes a diagonal step on the 5 cm grid into a cell whose centre is 12 cm from a wall. What does the step cost, to two decimal places? [1 mark]

> **Answer:** 21.21 (accept within 0.01). The step is 5 x sqrt(2) = 7.07 cm, and the cell is closer than 20 cm to a wall, so it costs three times that: 21.21.

5. A cell is already on the queue and a cheaper route to it is found. What do most implementations do? [1 mark]

- ☐ **A** Push a second entry with the new cost, and skip the old one when it is popped because the node is already done
- ☐ **B** Search the heap for the old entry and change its cost
- ☐ **C** Ignore the cheaper route, since the node was pushed first
- ☐ **D** Restart the search from the start

> **Answer:** A. This is lazy deletion. It costs a little memory and avoids searching the heap.

6. Adding a clearance penalty to the cost map changed the route to run down the middle of the corridor. What changed in the algorithm? [1 mark]

- ☐ **A** Nothing: only the cost function changed, and the cheapest path under the new costs is a different route
- ☐ **B** Dijkstra switched to a greedy search
- ☐ **C** The heuristic became admissible
- ☐ **D** The graph gained extra edges down the middle

> **Answer:** A. The cost function is the interface. The extra distance is now cheaper than the penalty for hugging the wall.

The task: the cheapest way across

Same grid as U9.2, now with eight neighbours and the cost map above. Print `cost:`, the total cost of the cheapest path, and `min gap:`, the smallest distance from any cell on that path to a wall.

```
from bugbot import *
import heapq
import math
connect()

CELL = 5.0
N = 40
INFLATE = 8.0
NEAR = 20.0
WALLS = [(70.0, 0.0, 8.0, 115.0), (125.0, 85.0, 8.0, 115.0)]
```

The hint students can ask for: Same grid, eight neighbours. Entering a cell costs the length of the step in centimetres, times three if that cell's centre is closer than 20 cm to a wall. A priority queue ordered by cost so far, and a cell is finished the first time it comes off. The gap from a point to a rectangle is zero inside it and otherwise hypot of how far outside it is in x and in y.

A solution

```
from bugbot import *
import heapq
import math
connect()

CELL = 5.0
N = 40
INFLATE = 8.0
NEAR = 20.0
WALLS = [(70.0, 0.0, 8.0, 115.0), (125.0, 85.0, 8.0, 115.0)]

def centre(c):
    return (c[0] * CELL + CELL / 2, c[1] * CELL + CELL / 2)

def gap(c):
    x, y = centre(c)
    best = 1e9
    for ox, oy, ow, oh in WALLS:
        dx = max(ox - x, 0.0, x - (ox + ow))
        dy = max(oy - y, 0.0, y - (oy + oh))
        best = min(best, math.hypot(dx, dy))
    return best

def blocked(i, j):
    x, y = centre((i, j))
    if x < INFLATE or y < INFLATE or x > 200 - INFLATE or y > 200 - INFLATE:
        return True
    for ox, oy, ow, oh in WALLS:
        if ox - INFLATE <= x <= ox + ow + INFLATE and oy - INFLATE <= y <= oy + oh +
INFLATE:
            return True
    return False

grid = [[blocked(i, j) for j in range(N)] for i in range(N)]
```

```
start, goal = (6, 6), (34, 34)
STEPS = [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (1, -1), (-1, 1), (-1, -1)]

best = {start: 0.0}
came = {start: None}
queue = [(0.0, start)]
done = set()
while queue:
    cost, cur = heapq.heappop(queue)
    if cur in done:
        continue
    done.add(cur)
    if cur == goal:
        break
    for dx, dy in STEPS:
        nxt = (cur[0] + dx, cur[1] + dy)
        if not (0 <= nxt[0] < N and 0 <= nxt[1] < N) or grid[nxt[0]][nxt[1]]:
            continue
        step = CELL * math.hypot(dx, dy)
        price = step * (3.0 if gap(nxt) < NEAR else 1.0)
        if cost + price < best.get(nxt, 1e18):
            best[nxt] = cost + price
            came[nxt] = cur
            heapq.heappush(queue, (cost + price, nxt))

path, cur = [], goal
while cur is not None:
    path.append(cur)
    cur = came[cur]
print("cost:", round(best[goal], 1))
print("min gap:", round(min(gap(c) for c in path), 1))
```

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