ANSWERS

U9.8 Roadmaps

Planning · University · about 35 min

What this lesson is about

Sample the free space once, link it into a graph, and answer every later question with a graph search. Probabilistic roadmaps, narrow passages, and when a tree is better.

- Two phases
- Asking it questions
- How many neighbours?
- The narrow passage
- Roadmap or tree?

The task: one roadmap, three trips

Build a roadmap once, then drive three queries on it without touching a wall: from the start to the green corner, from there to the bottom right bay, and from there to a third goal that nobody knows until the program runs. - **The build.** 150 free samples, each linked to its K = 8 nearest neighbours that `clear()` says it can see. Only samples: the start and the goals are not in it. Straight after the build, print `nodes:` and how many nodes the roadmap has, which is 150. - **`add_node(p)`.** Takes a point `p = (x, y)` in mat centimetres, adds it to the roadmap linked to its K nearest nodes that it can see, and returns its node number. - **`query(a, b)`.** Takes two points, adds both with `add_node`, and runs Dijkstra between them. Returns `(route, length)`, where `route` is the list of `(x, y)` points from `a` to `b` and `length` is its length in centimetres, or `None` if the search never reaches `b`. When a query comes back `None`, throw the samples away, build a fresh roadmap of 150 and ask again. - **The third goal.** The simulator picks it afresh every run, somewhere in the free space to the left of the first wall, and the program reads it with `input()`: two lines, x then y, in mat centimetres. The starter reads it into `GOAL3` and marks it on the mat with a red square, because the mat does not show it on its own. It is not in the build either; it joins the roadmap like any other query point. - **The trips.** Visit the corner, then the bay, then finish within 8 cm of `GOAL3`, one query each. Start each query from `here()`, where the robot actually is. Before driving each one, print `query 1 length:`, `query 2 length:` and `query 3 length:` with the route's length. `free()`, `clear()`, `here()`, `GOAL3` and `follow(route)`, which drives a list of waypoints the way U9.7 did, are written for you.

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

DT = 0.1
INFLATE = 10.0
N_SAMPLES, K = 150, 8
V_MAX, V_LAT = 20.0, 15.0
START = (30.0, 30.0)
GOALS = [(170.0, 170.0), (170.0, 30.0)]        # the corner, then the bay
WALLS = [(70.0, 0.0, 8.0, 115.0), (125.0, 85.0, 8.0, 115.0)]

def free(x, y):
    if x < INFLATE or y < INFLATE or x > 200 - INFLATE or y > 200 - INFLATE:
        return False
    return not any(ox - INFLATE <= x <= ox + ow + INFLATE and oy - INFLATE <= y <= oy + oh
+ INFLATE
                   for ox, oy, ow, oh in WALLS)

def clear(a, b):
    d = math.hypot(b[0] - a[0], b[1] - a[1])
    n = max(2, int(d / 1.5))
    return all(free(a[0] + (b[0] - a[0]) * k / n, a[1] + (b[1] - a[1]) * k / n) for k in
range(n + 1))

GOAL3 = (float(input()), float(input()))       # the third goal: the simulator picks it
every run, x then y in mat cm
draw("goal 3", [GOAL3], "red", "squares")      # the mat does not show it, so mark it

def here():
    px, py = position()
    return START[0] + px, START[1] + py

def follow(route):
    """Drive a list of (x, y) waypoints in order, holding the heading at zero."""
```

```
    for wx, wy in route[1:]:
        for tick in range(400):
            x, y = here()
            dx, dy = wx - x, wy - y
            gap = math.hypot(dx, dy)
            if gap < 4.0:
                break
            speed = min(13.0, 3.0 + 0.5 * gap)
            vx, vy = speed * dx / gap, speed * dy / gap
            h = math.radians(heading())
            spin = (heading() + 180) % 360 - 180
            drive(100 * (vx * math.sin(h) + vy * math.cos(h)) / V_MAX, 100 * (vx *
math.cos(h) - vy * math.sin(h)) / V_LAT,
                  max(-30.0, min(30.0, -0.8 * spin)))
            wait(DT)
    stop()

# build the roadmap once from samples alone, then query it three times
```

The hint students can ask for: Build from the samples alone and print the node count. Then work out what add_node(p) has to do so that a point nobody knew at build time can join the graph, and call it for both ends inside each query. Decide what query() should hand back when the search never reaches the goal, and what the program does then. The third goal is already read into GOAL3 and marked on the mat: it is just one more query, from wherever the robot is, on the roadmap you already have.

A solution

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

DT = 0.1
INFLATE = 10.0
N_SAMPLES, K = 150, 8
V_MAX, V_LAT = 20.0, 15.0
START = (30.0, 30.0)
GOALS = [(170.0, 170.0), (170.0, 30.0)]        # the corner, then the bay
WALLS = [(70.0, 0.0, 8.0, 115.0), (125.0, 85.0, 8.0, 115.0)]

def free(x, y):
    if x < INFLATE or y < INFLATE or x > 200 - INFLATE or y > 200 - INFLATE:
        return False
    return not any(ox - INFLATE <= x <= ox + ow + INFLATE and oy - INFLATE <= y <= oy + oh
+ INFLATE
                   for ox, oy, ow, oh in WALLS)

def clear(a, b):
    d = math.hypot(b[0] - a[0], b[1] - a[1])
    n = max(2, int(d / 1.5))
    return all(free(a[0] + (b[0] - a[0]) * k / n, a[1] + (b[1] - a[1]) * k / n) for k in
range(n + 1))

GOAL3 = (float(input()), float(input()))       # the third goal: the simulator picks it
every run, x then y in mat cm
draw("goal 3", [GOAL3], "red", "squares")      # the mat does not show it, so mark it

def here():
```

```
    px, py = position()
    return START[0] + px, START[1] + py

def follow(route):
    """Drive a list of (x, y) waypoints in order, holding the heading at zero."""
    for wx, wy in route[1:]:
        for tick in range(400):
            x, y = here()
            dx, dy = wx - x, wy - y
            gap = math.hypot(dx, dy)
            if gap < 4.0:
                break
            speed = min(13.0, 3.0 + 0.5 * gap)
            vx, vy = speed * dx / gap, speed * dy / gap
            h = math.radians(heading())
            spin = (heading() + 180) % 360 - 180
            drive(100 * (vx * math.sin(h) + vy * math.cos(h)) / V_MAX, 100 * (vx *
math.cos(h) - vy * math.sin(h)) / V_LAT,
                  max(-30.0, min(30.0, -0.8 * spin)))
            wait(DT)
    stop()

def nearest(p, count):
    return sorted(range(len(nodes)), key=lambda j: math.hypot(nodes[j][0] - p[0], nodes[j]
[1] - p[1]))[:count]

def build():
    """Sample the free space and link each sample to its K nearest that it can see. Samples
only."""
    global nodes, adj
    nodes = []
    while len(nodes) < N_SAMPLES:
        p = (random.uniform(0, 200), random.uniform(0, 200))
        if free(*p):
            nodes.append(p)
    adj = {i: {} for i in range(len(nodes))}
    for i, a in enumerate(nodes):
        for j in nearest(a, K + 1)[1:]:
            if j not in adj[i] and clear(a, nodes[j]):
                adj[i][j] = adj[j][i] = math.hypot(nodes[j][0] - a[0], nodes[j][1] - a[1])

def add_node(p):
    near = nearest(p, K)
    i = len(nodes)
    nodes.append(p)
    adj[i] = {}
    for j in near:
        if clear(p, nodes[j]):
            adj[i][j] = adj[j][i] = math.hypot(nodes[j][0] - p[0], nodes[j][1] - p[1])
    return i

def query(a, b):
    s, g = add_node(a), add_node(b)
    dist, came, q = {s: 0.0}, {s: None}, [(0.0, s)]
    while q:
        d, u = heapq.heappop(q)
        if u == g:
            break
        if d > dist[u]:
```

```
            continue
        for v, w in adj[u].items():
            if d + w < dist.get(v, 1e18):
                dist[v], came[v] = d + w, u
                heapq.heappush(q, (d + w, v))
    if g not in dist:
        return None
    path, u = [], g
    while u is not None:
        path.append(nodes[u])
        u = came[u]
    return path[::-1], dist[g]

def plan(a, b):
    """A query, and if the roadmap has no route, a fresh set of samples and another try."""
    answer = query(a, b)
    while answer is None:
        print("no route: resampling")
        build()
        answer = query(a, b)
    return answer

build()
print("nodes:", len(nodes))

for k, goal in enumerate(GOALS + [GOAL3]):
    route, length = plan(here(), goal)
    print("query", k + 1, "length:", round(length, 1))
    follow(route)
```

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