

รายงานวิชา 2110327 Algorithm Design

ภาคการศึกษา 2023/3

Shortest Path in Hex Map

a66_q3a_hex_map

พีรณัฐ กิตติวิทยากุล

6330374121

แนวทางการแก้ปัญหา

ข้อนี้จะใช้วิธีไล่ search adjacent node ไปเรื่อย ๆ โดยจะเอาอัลกอริทึม find path ของวิธี breadth first search มาประยุกต์ใช้ โดยจะมองช่องแต่ละช่องเป็น node หรือปมของกราฟที่มีเส้นเชื่อมจากปมนั้น ๆ ออกได้ 6 เส้นซึ่งก็คือ 6 ช่องที่เดินไปได้

Code Summary

Global Variable

```
8  const int inf = numeric_limits<int>::max();
9  int value[1000][1000];
10 int cost[1000][1000];
11 int r, c;
12
13 int dx_odd[] = {0, 0, -1, -1, 1, 1};
14 int dy_odd[] = {-1, 1, -1, 0, -1, 0};
15
16 int dx_even[] = {0, 0, -1, -1, 1, 1};
17 int dy_even[] = {-1, 1, 0, 1, 0, 1};
```

- value[][] เป็น 2D array ที่ใช้เก็บค่าใช้จ่ายที่ต้องใช้ในการเดินไปแต่ละช่องซึ่งค่าใน value[][] จะไม่ได้ถูก modified
- cost[][] เป็น 2D array ที่ใช้เก็บค่าใช้จ่ายทั้งหมดที่ต้องจ่ายในการเดินมาถึงช่องนั้น ๆ นับจากจุดเริ่มต้นโดย cost[i][j] จะเก็บค่าใช้จ่ายที่น้อยที่สุดในทุกทางเมื่อเทียบกันแล้วในการเดินมาที่ช่องที่ (i,j)
- inf เก็บค่าคงที่ infinity ไว้ initialize ไล่ array cost[][] เพื่อให้มีค่าเริ่มต้นเป็นค่าสูงที่สุด
- r และ c เก็บค่า row และ column
- dx_odd และ dy_odd เก็บวิธีการเดินทั้ง 6 วิธีที่เดินได้หาก x เป็นเลขคี่โดยแยกเก็บค่า x และ y ที่เปลี่ยนไป
- dx_even และ dy_even เก็บเช่นเดียวกับ odd แต่เป็นกรณีที่ x เป็นเลขคู่

Data Class: location

```
19 class location {
20 public:
21     int x;
22     int y;
23     int cost;
24
25     location(int a, int b, int c) : x(a), y(b), cost(c) {}
26
27     bool operator==(const location &other) const {
28         return (x == other.x) && (y == other.y);
29     }
30 };
```

ในข้อนี้เนื่องจากในแต่ละช่องระบุด้วยค่าแกน x และ y รวมถึงมีค่าใช้จ่ายจึงเหมาะที่จะสร้าง class ใหม่ที่รวมข้อมูลทั้ง 3 ไว้ใน class เดียว พร้อมทั้ง operator == เช็คว่าเป็นจุดเดียวกัน

Summary on main()

```
79 int main(int argc, char const *argv[]) {
80     ios_base::sync_with_stdio(false);
81     cin.tie(0);
82
83     int a1, b1, a2, b2;
84     cin >> r >> c;
85     cin >> a1 >> b1 >> a2 >> b2;
86     a1--; b1--; a2--; b2--; // Convert to 0-based indexing
87
88     for (int i = 0; i < r; i++) {
89         for (int j = 0; j < c; j++) {
90             cin >> value[i][j];
91             cost[i][j] = inf;
92         }
93     }
94
95     location start(a1, b1, value[b1][a1]);
96     location stop(a2, b2, inf);
97
98     cout << shortest_path_hex_map(start, stop) << endl;
99     return 0;
100 }
```

ในการรับค่าจุดเริ่มต้นและสิ้นสุด เนื่องจากในโจทย์เป็นระบบ index เริ่มต้นที่ 0 จึงต้องปรับค่า index ที่รับมาให้เข้ากับระบบ 0-based indexing ใน line 86 ส่วนใน line 91 คือการ set cost[][] ให้เป็นค่า max ใน loop ที่รับค่าใช้จ่ายที่เดียว จุดสังเกตคือตอนนำค่าพิกัดในโจทย์ไปเรียก 2D array ทั้ง value และ cost ต้องสลับตำแหน่งกันเพราะ x ในค่าตารางโจทย์แทนตำแหน่ง column ส่วน y จะแทนตำแหน่ง row จึงต้องใช้สลับกันต้อง indexing

Version 1

Points	30.0/100
Comment	PPPPP-P-----

`int shortest_path_hex_map(location start, location stop)` ของ submit นี้จะรับตำแหน่งเริ่มต้นและตำแหน่งจุดสิ้นสุดแล้ว return ค่าใช้จ่ายที่น้อยที่สุดกลับ version นี้จะเอาอัลกอริทึม find path ของวิธี breadth first search มาประยุกต์ใช้ดังที่กล่าวไปข้างต้น โดยจะใช้ queue เป็น structure เก็บ node ซึ่งให้คำตอบที่ถูกต้องแค่ 30% จึงเป็นอัลกอริทึมที่ถูกแค่ในบาง testcases ซึ่ง testcases ที่ถูกทั้งหมดมีแค่กราฟขนาดเล็ก เกือบทั้งหมด R, C น้อยกว่า 10 แปลว่าถ้ากราฟขนาดใหญ่จะให้คำตอบที่ไม่ได้มีค่าน้อยที่สุด

แนวคิดของ `shortest_path_hex_map_v1()`

```
36  int shortest_path_hex_map(location start, location stop) {
37      queue<location> q;
38      q.push(start);
39      cost[start.y][start.x] = start.cost;
40
41      while (!q.empty()) {
42          location present = q.front();
43          q.pop();
44
45          // Went here at lower cost
46          if (cost[present.y][present.x] < present.cost) {
47              continue;
48          }
49          // Reach stop point
50          if (present == stop) {
51              break;
52          }
```

แนวคิดคือใช้ queue เหมือน breath first search เริ่มที่ location start แล้ว update `cost[y][x]` ของจุดเริ่มต้นเท่ากับค่าใช้จ่ายตัวเองเพราะเริ่มจุดแรกที่ตรงนี้

ในทุก while loop หลังจาก pop มา ที่ line 46 ให้เอาค่า `cost`(attribute ใน class location) ของ node ที่ทำงานอยู่มาเทียบกับ cost array ว่าเคยมาที่ node นี้หรือยัง ถ้ายังไม่เคยมาค่าด้านซ้ายจะเป็น inf ก็จะลงไปทำงานต่อ ถ้าเคยมาแล้วจะเทียบค่า ถ้าหากทางที่เคยใช้มาเพื่อมาจุดนี้ก่อนหน้านี้ใช้

ค่าใช้จ่ายน้อยกว่าตอนปัจจุบันหมายความว่าเราจะเลือกใช้ทางเก่าดีกว่า ทางปัจจุบันไม่จำเป็นต้องทำต่อแล้ว

Line 50 จะเป็นเงื่อนไขในการออกจาก while loop ถ้า node ปัจจุบันที่ทำงานคือจุดสิ้นสุด โดยฟังก์ชันการเทียบ == ถูกเขียนทับในการประกาศ class แล้ว

```
53
54     if (present.x % 2 != 0) {
55         for (int i = 0; i < 6; i++) {
56             int x = present.x + dx_even[i];
57             int y = present.y + dy_even[i];
58             if (x < 0 || y < 0 || x >= c || y >= r) {
59                 continue;
60             }
61             if (cost[y][x] > present.cost + value[y][x]) {
62                 cost[y][x] = present.cost + value[y][x];
63                 q.push(location(x, y, cost[y][x]));
64             }
65         }
66     } else {
67         for (int i = 0; i < 6; i++) {
68             int x = present.x + dx_odd[i];
69             int y = present.y + dy_odd[i];
70             if (x < 0 || y < 0 || x >= c || y >= r) {
71                 continue;
72             }
73             if (cost[y][x] > present.cost + value[y][x]) {
74                 cost[y][x] = present.cost + value[y][x];
75                 q.push(location(x, y, cost[y][x]));
76             }
77         }
78     }
79 }
80 return cost[stop.y][stop.x];
81 }
```

หลังจากกรอง condition ที่จะออกจาก loop ออก ส่วนที่ทำงานจริง ๆ คือนำค่า x มาดูว่าเป็นเลขคี่หรือคู่ แต่เนื่องจากค่า x ในโจทย์เป็นระบบเริ่มต้นที่ 1 แต่ในโปรแกรมนี้เป็นระบบเริ่มจาก 0 ซึ่งปรับค่า x มาแล้ว เราจึงใช้ค่า x ที่เป็นเลขคี่เข้า condition block ที่เดินของ dx, dy เลขคู่ เริ่มจาก line 55 คือ loop การเดินทั้ง 6 แบบที่เป็นไปได้และกรองเอาด้านที่เดินไปชนขอบออก

ถ้าหาก node ที่เดินไปแต่ละเค้เดินมาแล้วด้วยค่า $cost[y][x]$ ที่น้อยกว่า จะไม่ทำงานต่อ ไปดูทางต่อไป (line 61) แต่ถ้ายังไม่เคยเดินมาหรือเคยมาด้วยค่าที่มากกว่า ให้บันทึกค่าที่เดินมาจากทางปัจจุบัน + ค่าใช้จ่ายช่องนั้นแล้ว push node ใหม่เข้า queue ไปทำงานต่อ ใน block else ทำงานเหมือนกัน แต่เป็นการเดินแบบ x ที่เป็นเลขคู่

หลังจากหลุดออกจาก while loop เพราะเดินถึงจุด stop แล้วค่า $cost[][]$ ที่พิกัดจุด stop จะเก็บค่าใช้จ่ายที่น้อยที่สุดที่เดินมาถึงจุดนี้ return ค่านี้เป็นคำตอบ

shortest_path_hex_map_v1() Time Complexity

อัลกอริทึมนี้ใช้ breadth-first search เราจะวิเคราะห์ time complexity ของ version นี้ได้ดังนี้

- การ initialization ของ global variable และ array ต่าง ๆ $\Rightarrow O(1)$
- ววนแถวและคอลัมน์ใส่ค่าใน value และ set cost เป็น inf $\Rightarrow O(R \times C)$
- ใน BFS loop แต่ละ cell จะถูก push เข้าไปทำงานได้แค่ครั้งเดียว ใน worst case คือทุก cell ถูก push เข้าไปจำนวนทั้งหมดจำนวน $R \times C$ cells
 - ใน while loop การ pop() front() ของ queue และกรอง condition $\Rightarrow O(1)$ ทั้งหมด
 - For loop cell ละ 6 รอบเพื่อ check การทำงานต่าง ๆ ละ queue.push() ทั้งหมดเป็น operation ที่ใช้เวลา $\Rightarrow O(1)$ ทั้งหมด

จะได้ว่าทั้ง BFS loop ใช้เวลาทั้งหมด $\Rightarrow O(R \times C)$

เมื่อนำค่ามารวมกันเพราะทำต่อกันจะได้ overall time complexity = $O(R \times C)$

ปัญหาของ shortest_path_hex_map_v1()

หากพิจารณา testcase ที่ถูกต้อง ปรากฏว่าเป็น testcases ที่มีแค่กราฟขนาดเล็ก เกือบทั้งหมด R, C น้อยกว่า 10 แปลว่าถ้ากราฟขนาดใหญ่จะให้คำตอบที่ไม่ได้มีค่าน้อยที่สุดหรืออัลกอริทึมยังให้คำตอบที่ผิด ไม่ถูกต้อง จึงต้องมองหาวิธีอื่น

เนื่องจากแต่ละช่องมีค่าใช้จ่าย หากเรามองค่าใช้จ่ายแต่ละช่องเป็นระยะทางบน edge ที่เข้าสู่ node นั้น แล้วมองปัญหาเป็นการหา shortest path แล้วใช้ Dijkstra's algorithm มาประยุกต์ใช้ได้

Version 2

Points	100.0/100
Comment	PPPPPPPPPPPPPPPPPPPP

Version 2 นี้ concept หลาย ๆ อย่างคล้าย version แรกแต่เปลี่ยนจากใช้ queue มาเป็น priority queue เนื่องจากเราเปลี่ยนการมองปัญหาเป็นจากการจ่ายค่าเดินในช่องเป็นมองว่าเป็นระยะทางใน edge ขานี้ เราสามารถมองปัญหาเป็น shortest path problem ได้ เราจึงใช้ Dijkstra's algorithm เพื่อไปสำรวจ node ที่มีค่าใช้จ่ายน้อยที่สุดก่อน นั่นหมายความว่าเราจะต้องทำงานไล่จาก node ที่มีค่าใช้จ่ายน้อยสุด จึงต้องใช้ priority queue แทน queue

Modified Code

```
19  class location {
20  public:
21      int x;
22      int y;
23      int cost;
24
25      location(int a, int b, int c) : x(a), y(b), cost(c) {}
26
27      bool operator<(const location &other) const {
28          return cost > other.cost;
29      }
30
31      bool operator==(const location &other) const {
32          return (x == other.x) && (y == other.y);
33      }
34  };
```

เนื่องจากเราต้องการเก็บ node location ใน priority queue โดยให้เรียงจากค่าใช้จ่ายน้อยไปมาก จึงต้องเขียน override ทับตัว operator < ให้ตัวน้อยกว่ามาก่อน

code ใน main() เหมือนเดิม ส่วนในฟังก์ชันหลักก็เกือบจะเหมือนเดิมทุกอย่างยกเว้นเปลี่ยนจากใช้ queue เป็น priority queue แทน และเปลี่ยน front() เป็น pop()

Full code `shortest_path_hex_map_v2()`

```
36  int shortest_path_hex_map(location start, location stop) {
37      priority_queue<location> pq;
38      pq.push(start);
39      cost[start.y][start.x] = start.cost;
40
41      while (!pq.empty()) {
42          location present = pq.top();
43          pq.pop();
44
45          // Went here at lower cost
46          if (cost[present.y][present.x] < present.cost) {
47              continue;
48          }
49          // Reach stop point
50          if (present == stop) {
51              break;
52          }
53
54          if (present.x % 2 != 0) {
55              for (int i = 0; i < 6; i++) {
56                  int x = present.x + dx_even[i];
57                  int y = present.y + dy_even[i];
58                  if (x < 0 || y < 0 || x >= c || y >= r) {
59                      continue;
60                  }
61                  if (cost[y][x] > present.cost + value[y][x]) {
62                      cost[y][x] = present.cost + value[y][x];
63                      pq.push(location(x, y, cost[y][x]));
64                  }
65              }
66          } else {
67              for (int i = 0; i < 6; i++) {
68                  int x = present.x + dx_odd[i];
69                  int y = present.y + dy_odd[i];
70                  if (x < 0 || y < 0 || x >= c || y >= r) {
71                      continue;
72                  }
73                  if (cost[y][x] > present.cost + value[y][x]) {
74                      cost[y][x] = present.cost + value[y][x];
75                      pq.push(location(x, y, cost[y][x]));
76                  }
77              }
78          }
79      }
80      return cost[stop.y][stop.x];
81  }
```


shortest_path_hex_map_v2() Time Complexity

วิเคราะห์ในลักษณะเดียวกับ v1

- การ initialization และวนไล่ค่าใน value และ cost $\Rightarrow O(R \times C)$
- ใน while loop แต่ละ cell จะถูก push เข้าไปทำงานได้แค่ครั้งเดียว ใน worst case คือทุก cell ถูก push เข้าไปคำนวณทั้งหมดจำนวน $R \times C$ cells

- จุดที่แตกต่างจาก queue คือ ในแต่ละ loop การทำงาน operation `pq.push()` และ `pq.pop()` ของ data structure binary heap หลังจากใส่หรือเอาออก จะต้องมีการ fix-up fix-down ซึ่งมี time complexity เป็น $O(\log(N))$ โดย N คือจำนวนของใน binary heap ซึ่ง worst case คือทุก cell และแต่ละ loop ต้องวนเดิน 6 รอบ for loop

จะได้ time complexity ของ while loop ทั้งหมดคือ $O((R \times C) \times 6 \times \log(R \times C)) = O((R \times C) \times \log(R \times C))$

เมื่อนำค่ามารวมกับ $O(R \times C)$ ของการ set ค่าแล้วจะได้ $O((R \times C) \times \log(R \times C))$ เพราะพจน์ $O((R \times C) \times \log(R \times C))$ dominate $O(R \times C)$

Finalized code: https://github.com/Peeranut-Kit/algorithm-design-coding/blob/main/problem/hex_map/hex_map.cpp