พีรณัฐ กิตติวิทยากุล 6330374121

Homework 5: เขียนข้อหุ่นยนต์ดูดฝุ่น

จากที่คุยในห้องเรียน

ให้ state คือ

```
class state {
  int score;
  int row,col;
  vector<vector<int>> floor;  // floor[r][c] ระบุความสกปรกของช่องนั้น ถ้า -1 แปลว่าเดินไปไม่ได้
};
```

ให้ทำ Least-Cost Search จากโครงที่ระบุไว้ให้

Global Variable

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <queue>

using namespace std;

int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // up, down, left, right
```

ระบุ directions array ขนาด 4 x 2 เป็นค่าแกน x และ y หรือ row , column ที่เคลื่อนที่ได้ทั้ง 4 แบบเนื่องจากจะถูกเรียกใช้งานในฟังก์ชันอื่น ๆ อีก 2 ฟังก์ชันจึงประกาศเป็น global เพื่อความสะดวก

State Class

```cpp
class state
{
public:
    int score;
    int row, col;
    vector< vector<int> > floor;

    int heuristic_value;

    state(int s, int r, int c, vector< vector<int> > f) : score(s), row(r), col(c), floor(f) {} //

    // For std::set check for position in row, col and floor matrix
    bool operator<(const state &other) const {
        return !(this->row == other.row && this->col == other.col && this->floor == other.floor);
    }
};
```

เพิ่มตัวแปร int heuristic_value ไว้เก็บค่า heuristic value ของ state ที่กำลังจะเดินไปสำรวจเพื่อเอามาเทียบเรียงใน priority queue, เพิ่ม constructor function และเขียนทับ operator < เพื่อบอกว่าถ้า state ที่เครื่องอยู่ row, column เดียวกันและความสะอาดทุกช่องเหมือนกันคือตัวเดียวกัน เรื่องลำดับไม่ใช่ประเด็นเพราะเราสนแค่ว่าเป็นตารางเดียวกันไหมจึงใช้ ! เลย

Helper Function อื่น ๆ ที่เพิ่มมา

```
27    class heuristic_comparer {
28    public:
29        bool operator()(const state &s1, const state &s2) const {
30            return s1.heuristic_value > s2.heuristic_value;
31        }
32    };
```

Heuristic comparer class เขียนเตรียมไว้ใส่ใน priority queue บอกว่า state ไหนจะมาก่อนในคิว โดยต้องการอันที่ heuristic value น้อยกว่ามาก่อน จึงใช้ >

```
34  // Build bool matrix of floor[][] size where true is reachable cell and false is blocked cell
35  vector< vector<bool> > get_reachable_matrix(state &x)
36  {
37      int rows = x.floor.size();
38      int cols = x.floor[0].size();
39      vector< vector<bool> > visited(rows, vector<bool>(cols, false));
40      visited[x.row][x.col] = true;
41      queue< pair<int, int> > q;
42      q.push(make_pair(x.row, x.col));
43
44      while (!q.empty())
45      {
46          pair<int,int> pair = q.front();
47          q.pop();
48          for (int i = 0; i < 4; i++)
49          {
50              int newRow = pair.first + directions[i][0];
51              int newCol = pair.second + directions[i][1];
52              if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols) {
53                  if (x.floor[newRow][newCol] != -1 && !visited[newRow][newCol]) {
54                      q.push(make_pair(newRow, newCol));
55                      visited[newRow][newCol] = true;
56                  }
57              }
58          }
59      }
60
61      return visited;
62  }
```

get_reachable_matrix รับค่า state แล้ว return vector< vector<bool> > ที่มีขนาดเท่า floor แต่เก็บ boolean แทนค่า int โดยช่อง true คือช่องที่ถ้าเริ่มจาก state ที่รับไปจะสามารถเดินไปถึง แต่ถ้าเดินไปไม่ได้

เพราะโดนช่อง -1 ขวางไว้และไม่มีทางไปถึงได้จะเป็น false วิธีการสร้างจะใช้ breadth-first search โดยใช้ queue ร่วมกับ visited[r][c] เป็นตัว track ว่าช่องนี้มาร์คไว้แล้ว และส่งเป็นตารางคำตอบ

1. isClean(state x)

```
81  bool isClean(state &x)
82  {
83      vector< vector<bool> > reachable_matrix = get_reachable_matrix(x);
84      int rows = reachable_matrix.size();
85      int cols = reachable_matrix[0].size();
86      for (int r = 0; r < rows; ++r) {
87          for (int c = 0; c < cols; ++c) {
88              if (reachable_matrix[r][c] && x.floor[r][c] > 0) {
89                  return false;
90              }
91          }
92      }
93      return true;
94  }
```

ตรวจสอบว่า ณ state x ตาราง floor ทุก cell ที่เดินทางไปได้นั้นสะอาดหรือยัง นั่นคือมีค่าเป็น 0 ทุก cell โดยเราจะสร้างตาราง bool ที่บอกขอบเขตที่เดินไปได้ทั้งหมดด้วยฟังก์ชัน get_reachable_matrix(x) มาช่วยเทียบ เช็คทีละช่องว่าช่องที่เป็น true มีค่ามากกว่า 0 หรือไม่ ถ้าใช่ return false คือยังไม่ clean

2. children_of(state x)

```
96  vector<state> children_of(state &x)
97  {
98      vector<state> children;
99      int rows = x.floor.size();
100     int cols = x.floor[0].size();
101
102     for (int i = 0; i < 4; i++) {
103         int newRow = x.row + directions[i][0];
104         int newCol = x.col + directions[i][1];
105         if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && x.floor[newRow][newCol] != -1)
106         {
107             state child = state(x.score + 1, newRow, newCol, x.floor);
108             child.heuristic_value = heuristic(child);
109             children.push_back(child);
110         }
111     }
112     return children;
113 }
```

รับ state x แล้ว return เป็น vector ของ state ทั้งหมดที่เดินไปได้จาก state x ที่ส่งไป โดยใช้ global variable directions ไล่บวก ใน line 105 เช็คว่า index ใหม่ row กับ column ยังอยู่ในขอบเขตมั้ย และ floor[][] ช่องนั้นไม่ -1 นั่นคือเดินไปได้ จะสร้าง state ใหม่โดยใช้ค่าใหม่ที่คำนวณมาและเพิ่ม score อีก 1 จากการเดิน รวมถึง assign ค่า heuristic_value ในขั้นตอนนี้ก่อนที่จะ push_back ใส่ vector เพราะในขั้นนี้ข้อมูลพร้อมคำนวณ heuristic value แล้ว โดยจะกล่าวถึง heuristic(state x) ในขั้นตอนต่อไป

3. heuristic(state x)

```
64  int heuristic(state &x) {
65      vector< vector<bool> > reachable_matrix = get_reachable_matrix(x);
66      int rows = reachable_matrix.size();
67      int cols = reachable_matrix[0].size();
68      int sum = 0;
69      for (int i = 0; i < rows; i++) {
70          for (int j = 0; j < cols; j++) {
71              if (reachable_matrix[i][j] && x.floor[i][j] > 0) {
72                  int row_diff = abs(i - x.row);
73                  int col_diff = abs(j - x.col);
74                  sum += row_diff + col_diff + (3 * x.floor[i][j] - 2); /
75              }
76          }
77      }
78      return x.score + sum;
79  }
```

heuristic function คำนวณค่าประมาณจาก 2 อย่างรวมกัน คือ ในทุก cell ที่ยังสกปรก (floor[i][j] > 0) ค่าใช้จ่ายอย่างน้อยที่สุดที่หุ่นยนต์จะต้องทำงานแน่ ๆ ในการทำความสะอาด cell หนึ่ง ๆ คือการเดินทางไปที่ cell นั้นเพื่อทำความสะอาด 1 ครั้งแล้วเดินไป cell อื่นที่อยู่ติดกัน 1 cell เพื่อเดินกลับมาที่ cell เดิมแล้วทำความสะอาดวนไปเรื่อย ๆ จนกว่า floor[i][j] ของช่องนั้น ๆ จะเท่ากับ 0 ดังนั้น เราจะคิดค่า heuristic โดย

1. ระยะจากตำแหน่งเครื่องปัจจุบันจนถึง cell สกปรก L-1 distance => row difference + column difference
2. เมื่อถึงช่องนั้น ใช้ 1 หน่วย ทำความสะอาดและใช้อีกทุก 2 หน่วย ในการเดินไปช่องข้าง ๆ และกลับมาใช้ 1 หน่วยเพื่อหักค่า floor[i][j] ไป 1 ทำซ้ำ ๆ จนเป็น 0 => 1 + 3*(floor[i][j]-1) => 3*(floor[i][j]) -2
3. เอาค่ารวมของ 1. และ 2. มารวมกันจะได้ค่าประมาณอย่างต่ำของ cell นั้น ๆ เมื่อวน loop รวมค่าของแต่ละ cell ทุกช่องใน floor ที่ยังสกปรกและเดินไปได้โดยใช้ get_reachable_matrix(x) มาช่วย รวมกับ score ที่รวมแล้วจนถึง state ปัจจุบัน จะได้ค่า heuristic value ทั้งหมด return กลับออกไป

3	2	-1	1
0	1	-1	5
2	1	-1	-1
2	-1	10	3

จากตาราง 4 x 4 ข้างต้น ให้ state ปัจจุบันคืออยู่ที่ (0,0) ที่มุมบนซ้ายสุด บริเวณสีส้มคือไม่สามารถเดินไปได้แน่ ๆ เพราะติด block -1 ขวางไว้ heuristic value อย่างต่ำที่ต้องใช้ทำความสะอาดช่องล่างซ้าย (3,0) ให้ floor[3][0] เป็น 0 คือ (3-0) + (0-0) = 3 หน่วยที่ต้องเดินทางไป, 1 หน่วยทำความสะอาด 1 ครั้งเหลือ 2-1 = 1 แล้วเดินไปช่อง (2,0) แล้วกลับมา (3,0) เพื่อทำความสะอาดอีกครั้ง คือ 3*(1) = 3 หน่วย แล้วคิดแบบนี้กับทุก cell ที่เดินไปได้ จะได้สมการ heuristic function ของ state

พีรณัฐ กิตติวิทยากุล 6330374121

ภาพตัวอย่างการคำนวณ heuristic function ของ cell 1 cell (ค่าใช้จ่ายขั้นต่ำ) ยกตัวอย่างเครื่องอยู่ที่ (0,0) คำนวณของ cell (3,0) โดยให้วงกลมสีเขียวคือตำแหน่งหุ่นยนต์

3	2	-1	1
0	1	-1	5
2	1	-1	-1
2	-1	10	3

หุ่นยนต์อยู่ที่ (0,0) จะไป (3,0)

3	2	-1	1
0	1	-1	5
2	1	-1	-1
2	-1	10	3

เดินทางใช้ค่าใช้จ่ายเป็นระยะ L-1 distance

3	2	-1	1
0	1	-1	5
2	1	-1	-1
1	-1	10	3

clean() 1 ครั้งเพิ่ม score 1 หน่วย

3	2	-1	1
0	1	-1	5
2	1	-1	-1
1	-1	10	3

clean เสร็จต้องหาช่องเดินที่เป็นช่องข้างเคียงที่ไปได้เพิ่ม score 1 หน่วย

3	2	-1	1
0	1	-1	5
2	1	-1	-1
1	-1	10	3

เดินกลับไปช่องเดิมที่พิจารณาใช้ score 1 หน่วย

3	2	-1	1
0	1	-1	5
2	1	-1	-1
0	-1	10	3

ทำความสะอาด 1 หน่วย วนซ้ำ 3 รูปหลังจน floor ของ cell นั้นเป็น 0

Helper Function ใน Least-cost search

```
115  void clean(state &x)
116  {
117      if (x.floor[x.row][x.col] > 0) {
118          x.floor[x.row][x.col] -= 1; // Decrease the dirtiness by 1
119          x.score += 1;               // Increase the score by 1 for cleaning
120      }
121  }
122
123  // Display currect state to output log
124  void print_state(state &x, int best_so_far, int state_number) {
125      vector< vector<int> > v = x.floor;
126      cout << "State Number: " << state_number << endl;
127      cout << "At position row: " << x.row << " column: " << x.col << endl;
128      cout << "Score: " << x.score << " isClean = " << isClean(x) << endl;
129      cout << "Heuristic score: " << x.heuristic_value << endl;
130      for (size_t i = 0; i < v.size(); i++)
131      {
132          for (size_t j = 0; j < v[0].size(); j++)
133          {
134              cout << v[i][j] << " ";
135          }
136          cout << endl;
137      }
138      cout << "Best min value so far: " << best_so_far << endl;
139      cout << "--------------------------------" << endl;
140  }
```

- void clean(state x)

function สั่งทำความสะอาดช่องที่อยู่ตอนนั้น โดยถ้าสะอาดแล้วไม่ทำ ถ้ายังสกปรกก็ -1 ค่า floor แล้ว +score

- print_state(x)

function print ข้อมูลของ state ปัจจุบันเพื่อให้เห็นภาพการทำงานมากขึ้นว่าตอนนี้ตารางเป็นอย่างไร และบอกว่าตัว least-cost search ทำงานแล้วกี่ state

Least-Coast Search

```
142  // Least cost search main function
143  state cleaning_robot(state &start)
144  {
145      int best = INT_MAX;
146      state best_state = start;
147      int state_num = 0;
148      if (isClean(start)) return start;
149
150      set<state> state_collection;
151      priority_queue<state, vector<state>, heuristic_comparer> pq;
152
153      state_collection.insert(start);
154      pq.push(start);
155      while (!pq.empty())
156      {
157          state current = pq.top();
158          pq.pop();
159          state_num++;
160          print_state(current, best, state_num);
161
162          clean(current);
163          if (isClean(current))
164          {
165              if (current.score < best)
166              {
167                  best = current.score;
168                  best_state = current;
169              }
170          }
171          else {
172              vector<state> children = children_of(current);
173              for (state &child : children)
174              {
175                  if (child.heuristic_value < best) {
176                      if (state_collection.find(child) == state_collection.end()) {
177                          state_collection.insert(child);
178                          pq.push(child);
179                      }
180                  }
181              }
182          }
183      }
184      return best_state;
185  }
```

- intialiaze ค่า best และ best_state เป็น MAX และ start ตามลำดับ
- check state start ว่าสะอาดหรือยัง ถ้าสะอาดแล้วก็ไม่ต้องทำงาน return เลย

- สร้าง set state_collection เก็บ state ที่เคยมาแล้วเพื่อลดการทำซ้ำซ้อน
- สร้าง priority queue ของ state โดยใช้ comparator เป็น heuristic_comparer ที่เขียนเตรียมไว้
- push start state เข้าคิวแล้วเข้าทำงานใน while loop จน priority queue ทำงานจนหมด
- pop ตัวหน้าสุดมาทำงาน โดย line 159,160 เป็นการคำนวณและ print เพื่อ debug ไม่เกี่ยวกับ search โดยตรง
- เรียก clean(current) ทำความสะอาดช่องปัจจุบัน แล้วเช็คว่า floor สะอาดหรือยัง ถ้าสะอาดแล้ว ตรวจสอบว่าจำนวน score ที่มาถึง state นี้ที่ทำงานสำเร็จน้อยกว่าค่า best ที่เก็บไว้ค่าก่อนหน้าหรือไม่ ถ้าใช่ ให้ set state ที่ดีที่สุดเป็น state ปัจจุบัน ถ้า isClean(current) ไม่สะอาด เข้า else block
- ใน else block เรียก children_of(current) ดูช่องต่อไปที่เป็นไปได้ทั้งหมด แล้ววนทำงานทีละตัว ถ้าค่า heuristic_value ที่คำนวณจาก heuristic function มีค่าน้อยกว่าค่า best แปลว่ามีโอกาสได้ทางที่ดีกว่าให้ใส่เข้า set และ priority queue ไปทำงานต่อถ้าใน set ยังไม่เคยมี
- เมื่อทำจนครบแล้ว return state ที่ดีที่สุดที่เก็บไว้ในตัวแปร best_state

Main Function

```
187  int main(int argc, char const *argv[])
188  {
189      vector< vector<int> > floor(4, vector<int> (4));
190      floor[0][0] = 3;
191      floor[0][1] = 2;
192      floor[0][2] = -1;
193      floor[0][3] = 1;
194      floor[1][0] = 0;
195      floor[1][1] = 1;
196      floor[1][2] = -1;
197      floor[1][3] = 5;
198      floor[2][0] = 2;
199      floor[2][1] = 1;
200      floor[2][2] = -1;
201      floor[2][3] = 3;
202      floor[3][0] = 2;
203      floor[3][1] = -1;
204      floor[3][2] = -1;
205      floor[3][3] = 48;
206      /* start at (0,0)
207      3 2 -1 1
208      0 1 -1 5
209      2 1 -1 3
210      2 -1 -1 48
211      finish
212      0 0 -1 1
213      0 0 -1 5
214      0 0 -1 31
215      0 -1 -1 48
216      */
217      state start(0, 0, 0, floor);
218      state result = cleaning_robot(start);
219
220      //print_state(result, result.score, -1);
221      cout << "Total Score: " << result.score << endl;
222      return 0;
223  }
```

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