人工智能基础 实验一

PB19071501 李平治

I. 实验环境

- 2GHz 4vIntel Core i5
- g++-11 (Homebrew GCC 11.2.0_3) 11.2.0

II. A^*算法路径规划

i. 主要算法

1. A^*算法

类似代价一致搜索，A^*算法也采用耗散函数来计算每一步不同走法的代价，以此来选择路径扩展方向。区别在于A^*结合了从开始节点的路径代价和到目标节点的最小代价估计，即$f(n)=g(n)+h(n)$。关键在于启发函数的选择，要求低于实际代价，对于网格地图，通常使用曼哈顿距离或八角距离。

迭代A^*算法是A^*算法和迭代加深算法的结合，基本思路是用一个迭代增大的耗散值来限制每一个迭代的搜索范围。

2. 启发函数

- $h_1 = number\ of\ misplaced\ stars$
- $h_2 = min(MD(n, target), [MD(n, tunnel_{in})MD(tunnel_{out}, target)])$
  - MD为曼哈顿距离，$tunnel_{in}$和$tunnel_{out}$为隧道的入口和出口
  - Proof: h_2的实际含义是完全不考虑其他节点的情况下，该节点移动到目标的最小移动次数，因此一定严格不大于实际代价

3. 数据结构与变量

```cpp
int prev_step[MAX_LENGTH];
// 记录当前编号状态的上一步状态
int prev_direct[MAX_LENGTH];
// 记录当前编号状态的上一步移动方向
vector< vector<int> > input_pos(N, vector<int>(N, 0));
// 输入形势
vector< vector<int> > target_pos(N, vector<int>(N, 0));
// 目标形势
unordered_map<int, position> planet_to_target_pos;
// 行星编号到目标位置的映射
```

ii. 用例测试

1. A_h1

样例编号	运行时间/s	移动序列	总步数
00	1.4e-4	DDRUR	5
01	9.5e-5	ULLUULDD	8
02	4.8e-5	DDLUULLURR	10
03	2.22e-4	DLDRRURRRUUURR	14
04	5.45e-4	LUUURULLURDDRDR	15
05	1.36e-3	LLUURRRUURDDDDLUURDD	20
06	2.83e-3	DRDLLULULUUURDRURDRDRRR	23
07	5.94e-4	URRRRDLLLLDRRRRDLLLLDRRRR	25
08	9.23e-3	DLLLDRUUUULDRRRRULDDDRDLURD	27
09	0.13	RDRDLUUUURRUUURDRUUULDLDDDRR	28
10	6.12e-3	DDRRUUUULLULLUULLLLLUURRDDDDRR	30
11	0.679	DRUURDRRDRUULDLULDLDRDLDRURDRURD	32

2. A_h2

样例编号	运行时间/s	移动序列	总步数
00	4.8e-5	DDRUR	5
01	9e-5	ULLUULDD	8
02	5.2e-5	DDLUULLURR	10
03	2.33e-4	DLDRRURRRUUURR	14
04	1.43e-4	LUUURULLURDDRDR	15
05	1.54e-4	LLUURRRUURDDDDLUURDD	20
06	8.07e-4	DRDLLULULUUURDRURDRDRRR	23
07	2.89e-4	URRRRDLLLLDRRRRDLLLLDRRRR	25
08	4.28e-4	DDRULLLLDRUUUULDRRRRULDDDDR	27
09	3.72e-3	RDRDLUUUURRUUURDRUUULDLDDDRR	28
10	2.59e-4	DDRRUUUULLULLUULLLLLUURRDDDDRR	30
11	7.67e-3	DRUURDRRDRUULDLULDLDRDLDRURDRURD	32

3. IDA_h1

样例编号	运行时间/s	移动序列	总步数
00	4.1e-5	DDRUR	5
01	5e-5	ULLUULDD	8
02	4.3e-5	DDLUULLURR	10
03	4.14e-4	DLDRRURRRUUURR	14
04	1.12e-3	LUUURULLURDDRDR	15
05	2.15e-3	LLUURRRUURDDDDLUURDD	20
06	4.96e-3	DRDLLULULUUURDRURDRDRRR	23
07	6.05e-4	URRRRDLLLLDRRRRDLLLLDRRRR	25
08	2.30e-2	DLLLDRUUUULDRRRRULDDDRDLURD	27
09	0.3266	RDRDLUUUURRUUURDRUUULDLDDDRR	28
10	0.019	DDRRUUUULLULLUULLLLLUURRDDDDRR	30
11	1.41	DRUURDRRDRUULDLULDLDRDLDRURDRURD	32

4. IDA_h2

样例编号	运行时间/s	移动序列	总步数
00	4.6e-5	DDRUR	5
01	6.2e-5	ULLUULDD	8
02	5.1e-5	DDLUULLURR	10
03	2.52e-4	DLDRRURRRUUURR	14
04	1.56e-4	LUUURULLURDDRDR	15
05	1.31e-4	LLUURRRUURDDDDLUURDD	20
06	9.36e-4	DRDLLULULUUURDRURDRDRRR	23
07	2.99e-4	URRRRDLLLLDRRRRDLLLLDRRRR	25
08	4.62e-4	DDRULLLLDRUUUULDRRRRULDDDDR	27
09	6.86e-3	RDRDLUUUURRUUURDRUUULDLDDDRR	28
10	2.56e-4	DDRRUUUULLULLUULLLLLUURRDDDDRR	30
11	0.013	DRUURDRRDRUULDLULDLDRDLDRURDRURD	32

iii. 主要优化

- 使用stl有序容器（priority_queue、set）存储状态集合，加速最小耗散值状态选取
- 预处理建立行星到其目标位置的字典，加速启发函数h_2计算
- 使用重复检测技术减小搜索空间

III. 作业调度问题

i. 数据结构与变量

1. 分配结构

```
typedef struct Assignment{
    bool is_assigned = false;
    bool value = false;
} Assignment;
```

- 其中is_assigned表示该节点已被分配，从而value`值有效

2. 约束集合

```
class Constraint {
public:
    int num_workers;
    int num_seniors;
    int least_daily_num_workers;
    int least_rest_num_days;
    int most_continuous_rest_days;
    int worker_levels[MAX_NUM_WORKERS];
    std::vector< std::pair<int, int> > conflicts;

    bool back_track(Assignment assigns[][DAY_PER_WEEK]);
    bool is_complete(const Assignment assigns[][DAY_PER_WEEK]);
    bool validate_with_forward_checking(Assignment assigns[][DAY_PER_WEEK]);
    void select_unassigned_variable(const Assignment assigns[][DAY_PER_WEEK], int &worker, int&day);
    void duplicate_assignments(Assignment source_assigns[][DAY_PER_WEEK], Assignment dest_assigns[][DAY_PER_WEEK]);
};
Constraint constraints[NUM_CASES];
```

ii. 主要算法

- `Constraint::select_unassigned_variable()`

  返回一个未分配变量

- `Constraint::back_track()`

  进行回溯算法：运用`select_unassigned_variable()`选取一个未分配变量，并分别判断它的`value`取true和false时是否可用。若可用则继续进行迭代`back_track()`，否则进行回溯。

- `Constraint::validate_with_forward_checking()`

  判断当前分配集合是否满足约束，并进行前向检验

iii. 优化方法: 前向检验

1. 前向检验

在`Constraint::validate_with_forward_checking()`函数中，附带了前向检验代码，极大减小了搜索过程中的搜索空间：

- 检验是否含有工人冲突：

```
    for (int j=0; j<DAY_PER_WEEK; j++){
        /* handle worker conflicts */
        for (auto &conf:conflicts){
            int worker_first = conf.first;
            int worker_second = conf.second;
            if (assigns[worker_first][j].is_assigned && assigns[worker_first][j].value && assigns[worker_second][j].is_assigned && assigns[worker_second][j].value){
                // they both work today
                return false;
            } else if (!assigns[worker_first][j].is_assigned && assigns[worker_second][j].is_assigned && assigns[worker_second][j].value) {
                assigns[worker_first][j].is_assigned = true;
                assigns[worker_first][j].value = false;
                return validate_with_forward_checking(assigns);
            } else if (!assigns[worker_second][j].is_assigned && assigns[worker_first][j].is_assigned && assigns[worker_first][j].value) {
                assigns[worker_second][j].is_assigned = true;
                assigns[worker_second][j].value = false;
                return validate_with_forward_checking(assigns);
            }
        }
    }
```

当检测到冲突一方在当天被分配为工作时，立即将另一方分配为当天不工作

- 检测连续休息

```
    for (int i=0; i<num_workers; i++){
        /* most continuous rest days */
        int rest_num_days = 0;
        for (int j=0; j<DAY_PER_WEEK; j++){
            if (assigns[i][j].is_assigned && assigns[i][j].value){
                rest_num_days = 0;
            } else if (assigns[i][j].is_assigned && !assigns[i][j].value) {
                rest_num_days ++;
                if (rest_num_days > most_continuous_rest_days)
                    return false;
            } else if (!assigns[i][j].is_assigned) {
                if (rest_num_days == most_continuous_rest_days ){
                    assigns[i][j].is_assigned = true;
                    assigns[i][j].value = false;
                    return validate_with_forward_checking(assigns);
                }
                rest_num_days = 0;
            }
        }
    }
```

当检测到有工人已经连续休息了最长天数时，立即将其当天分配为工作

2. 优化效果

- 运行带有前向检验的算法时，两个测试用例运行结果为

```
Case 1 Time: 0.000279
2 4 6 7
1 3 5 7
2 4 6 7
1 3 5 7
2 4 5 6
1 3 5 7
2 4 5 6
Case 2 Time: 0.000932
5 6 7 9 10
1 2 3 4 8
5 6 7 9 10
1 2 3 4 8
5 6 7 9 10
1 2 3 4 8
5 6 7 9 10
```

- 运行不带前向检验的算法时，两个测试用例运行结果为

```
Case 1 Time: 0.002343
2 5 6 7
2 4 6 7
1 3 5 7
2 4 6 7
2 4 5 6
1 3 5 7
2 4 5 6
Case 2 Time: 0.852975
5 6 7 9 10
5 6 7 9 10
1 2 3 4 8
5 6 7 9 10
5 6 7 9 10
1 2 3 4 8
5 6 7 9 10
```

可以看出，时间优化效果非常显著。

iv. 模拟退火

伪代码如下，其中Constraint::check()为判断满足约束条件个数的函数

```
bool Constraint::back_track(Assignment assigns){
  if (is_satisfied(assigns)){
    return true;
  } else {
    auto assigns_cpy = assigns;
    for (auto &var: var_set){
      assigns.insert(var);
      auto new_val = assigns;
      if (check(new_val) > check(assigns_cpy) or check(new_val) > random(0,
1)*check(assigns_cpy)) {
        return back_track(new_val);
      } else {
        return false;
      }
    }
  }
}
```

总结

- 通过本次实验对两种搜索算法的实现，对算法思想和过程有了更加深刻的认识
- 通过本次实验，提高了C++编码能力和对stl的使用
- 本次实验用时：
  - A^* ： 5h
  - CSP：3.5h
  - 实验报告：1h