人工智能Lab1实验报告

PB19030888 张舒恒

问题一

A*搜索

每次从优先队列`fringe`中调用`pop()`方法取出首个节点，判断该节点是否是目标节点，若是则返回，否则调用`state_manage::get_succeed_states()`方法获取合法后继节点，将其插入`closeset`无序集合进行去重，插入`nodes`节点数组方便统计搜索节点总数，最后插入`fringe`优先队列。

```
Node *AStar::graph_search() {
    Node *n = this->nodes[0];
    int h;
    if (this->h_func == "h1")
        h = this->state_manager->h_function(n->state);
    else
        h = this->state_manager->h_function2(n->state);
    int g = n->depth;
    int f = g + h;
    this->fringe.push(std::make_pair(std::make_pair(f, g), n));
    while (!fringe.empty()) {
        auto t = fringe.top();
        fringe.pop();
        f = t.first.first;
        g = t.first.second;
        n = t.second;
        if (AStar::check(n->state, this->dest_state)) {
            //cout << this->fringe.size();
            return n;
        }
        auto succeeds = state_manage::get_succeed_states(n->state);
        for (auto &s: succeeds) {
            if (this->insert_closeset(s.second)) {
                Node *n_temp = new Node;
                n_temp->state = s.second;
                n_temp->from_parent_movement = s.first;
                n_temp->depth = n->depth + 1;
                n_temp->parent = n;
                int n_temp_g = n_temp->depth;
                int n_temp_h;
                if (this->h_func == "h1")
                    n_temp_h = this->state_manager->h_function(n_temp->state);
                else
                    n_temp_h = this->state_manager->h_function2(n_temp->state);
                int n_temp_f = n_temp_g + n_temp_h;
                fringe.push(std::make_pair(std::make_pair(n_temp_f, n_temp_g),
n_temp));
                this->nodes.emplace_back(n_temp);
            }
        }
    }
```

```
    return nullptr;
}
```

IDA*搜索

设置一个深度搜索门槛`doorsill`，它通过每轮深度搜索门槛外的节点评价函数的最小值来更新。如果当前搜索节点的评价函数超过门槛则直接返回，如果当前搜索节点是目标节点则返回成功，否则调用`state_manage::get_succeed_states()`方法获取合法后继节点。对每个后继节点，先判断是否访问过。如果从未访问过则将其`Info.in_stack`标记为`true`以记录该节点正在搜索中，再对该节点进行递归DFS，退出后将其`Info.in_stack`标记为`false`以记录该节点不在当前搜索。如果曾经访问过，则判断`Info.in_stack`是否为`true`，若为`true`则表明当前搜索出现环路，若为`false`则表明是之前轮次的搜索访问过该节点，此时若路径更优则进行更新。

```
int IDAStar::DFS(Node *n, int doorsill) {
    nodes_count++;
    int f = n->depth + n->h;
    if (f > doorsill) {
        return f;
    }
    if (this->check(n->state, this->dest_state)) {
        print_movement(n, this->resultFile);//TODO
        return -1;
    }
    int min = this->INFINITY;
    auto succeeds = state_manage::get_succeed_states(n->state);
    for (auto &s: succeeds) {
        auto n_find = this->visited.find(s.second);
        if (n_find == this->visited.end()) { //如果从未访问过
            Node *n_temp = new Node;
            n_temp->state = s.second;
            n_temp->movement = s.first;
            n_temp->depth = n->depth + 1;
            n_temp->parent = n;
            if(this->h_func == "h1")
                n_temp->h = this->state_manager->h_function(n_temp->state);
            else
                n_temp->h = this->state_manager->h_function2(n_temp->state);

            this->nodes.emplace_back(n_temp);
            this->visited.emplace(n_temp->state, INFO{true, n_temp});

            int temp = DFS(n_temp, doorsill);
            this->visited[n_temp->state].check_stack_in = false;
            if (temp == -1) {
                return -1;
            }
            if (temp < min) {
                min = temp;
            }
        } else {
            if (!n_find->second.check_stack_in) {  //访问过不在栈中
                if (n->depth + 1 <= n_find->second.n_his->depth) {  //找到更优的则
更新
                    Node *n_temp = n_find->second.n_his;
                    n_temp->depth = n->depth + 1;
                    n_temp->parent = n;
```

```
                    n_temp->movement = s.first;
                    n_find->second.check_stack_in = true;
                    int temp = DFS(n_temp, doorsill);
                    n_find->second.check_stack_in = false;
                    if (temp == -1) {
                        return -1;
                    }
                    if (temp < min) {
                        min = temp;
                    }
                } else {
                    delete[] s.second; //释放空间
                }
            } else {
                delete[] s.second;
            }
        }
    }
    return min;
}
```

启发函数2

采用改进的曼哈顿距离，即对于一个状态的任何当前星球位置和目标星球位置，当前飞船位置和目标飞船位置之间的曼哈顿距离**最多仅可使用一次隧道优化**。因为任意星球要达到目标位置最优的路径最多使用一次隧道，也即改进后的曼哈顿距离考虑的是两个星球之间最少的代价评估，每次移动飞船的最多只能使改进后的曼哈顿距离-2，所以这种改进后的曼哈顿距离的评价不会超过节点移动到目标状态的真实代价，从而是**可采纳**的。

```
int state_manage::h_function2(const char *state) const {
    int miss_count = 0;
    for (int i = 0; i < DIM * DIM; i++) {
        for (int j = 0; j < DIM * DIM; j++) {
            if (state[i] == this->dest_state[j]) {
                if (i == j)
                    break;
                else {
                    int xi = i % DIM, xj = j % DIM, yi = i / DIM, yj = j / DIM;
                    int x_distance = abs(xi - xj), y_distance = abs(yi - yj);
                    if (x_distance >= 3)
                        miss_count += (5 - x_distance + y_distance);
                    else if (y_distance >= 3)
                        miss_count += (5 - y_distance + x_distance);
                    else
                        miss_count += (y_distance + x_distance);
                    break;
                }
            }
        }
    }
    return miss_count;
}
```

算法设计亮点与优化

采用**面向对象**的结构设计，即使每个对象的构造都需要额外的时间，但带来的收益是参数直接可以在当前对象中读取，从而减少了参数传递的开销。

采用**字符编码**而非直接使用题目所给的数字类型，由于`char`只占一个字节，可以减少内存开销和读取，构造，拷贝，运算等等的时间开销，而且还可以调用常见的C++内置字符串操作方法。

```
for (int i = 0; i < DIM * DIM; ++i) {
    initFile >> buffer;
    char num = (char) std::stoi(buffer);
    if (num == 0) {
        init_state[i] = ZERO;
    } else if (num < 0) {
        init_state[i] = MINUS;
    } else {
        init_state[i] = num;
    }
}
```

采用C++内置的**多级优先队列**，不仅速度更快，还可支持用`pair`作为排序依据从而在评价函数相同时再以路径深度作为依据进行二次排序。

```
priority_queue<pair<pair<int, int>, Node *>, vector<pair<pair<int, int>, Node *>
>, greater<> > fringe;
```

采用C++内置的**无序集合**来进行去除重复状态，避免重复搜索，同时只需通过是否插入成功来判断重复，无需进行查找，在节点较多的11样例中速度提升明显。

```
std::unordered_set<std::string> close_set;
```

采用`emplace_back()`方法代替传统的`push_back()`，由于`push_back()`需要**先构造再拷贝对象**，而`emplace_back()`**原地构造**，所以后者性能更佳，尤其是该实验的时间瓶颈之一是出入队列。

```
result.emplace_back(make_pair(zero_location, 'l'), exchange_state(state,
zero_location, zero_location - 1));
```

IDA*搜索中通过搜索栈信息的记录来避免出现**环路搜索**以及在出现更优路径时及时对节点进行更新。

运行时间统计

运行时间会受到CPU硬件配置的影响，这里采用的是**Intel-i7-11600H CPU**。

A_h1

样例编号	移动序列	时间	移动步数
00	ddrur	0.001s	5
01	ulluuldd	0s	8
02	ddluullurr	0s	10
03	dldrrurrruuurr	0s	14
04	luuurullurddrdr	0.000993s	17
05	lluurrruurddddluurdd	0.002997s	20
06	drdllululuuurdrurdrdrrr	0.004999s	23
07	urrrrdllllldrrrrdllllldrrrr	0.001002s	25
08	dllldruuuuldrrrrulddddrdlurd	0.018994s	27
09	rdrdluuuurrdrddruuulldrulurr	1.028s	28
10	ddrruuuullulluullllluurrddddrr	0.088056s	30
11	druurdrrdruuldluldldrdldrurdrurd	6.13201s	32

A_h2

样例编号	移动序列	时间	移动步数
00	ddrur	0s	5
01	ulluuldd	0s	8
02	ddluullurr	0s	10
03	dldrrurrruuurr	0s	14
04	luuurullurddrdr	0.001003s	17
05	lluurrruurddddluurdd	0s	20
06	drdllululuuurdrurdrdrrr	0.001998s	23
07	urrrrdllllldrrrrdllllldrrrr	0.002s	25
08	dllldruuuuldrrrrulddddrdlurd	0.003s	27
09	rdrdluuuurrdrddruuulldrulurr	0.074001s	28
10	ddrruuuullulluullllluurrddddrr	0.003999s	30
11	druurdrrdruuldluldldrdldrurdrurd	0.1704s	32

IDA_h1

样例编号	移动序列	时间	移动步数
00	ddrur	0s	5
01	ulluuldd	0s	8
02	ddluullurr	0s	10
03	dldrrurrruuurr	0s	14
04	luuurullurddrdr	0.001003s	17
05	lluurrruurddddluurdd	0.003009s	20
06	drdllululuuurdrurdrdrrr	0.004001s	23
07	urrrrdllllldrrrrdllllldrrrr	0.001s	25
08	dllldruuuuldrrrrulddddrdlurd	0.020002s	27
09	rdrdluuuurrdrddruuulldrulurr	0.625002s	28
10	ddrruuuullulluullllluurrddddrr	0.127014s	30
11	druurdrrdruuldluldldrdldrurdrurd	7.46001s	32

IDA_h2

样例编号	移动序列	时间	移动步数
00	ddrur	0s	5
01	ulluuldd	0.001004s	8
02	ddluullurr	0s	10
03	dldrrurrruuurr	0s	14
04	luuurullurddrdr	0.001002s	17
05	lluurrruurddddluurdd	0s	20
06	drdllululuuurdrurdrdrrr	0.001004s	23
07	urrrrdllllldrrrrdllllldrrrr	0.001002s	25
08	dllldruuuuldrrrrulddddrdlurd	0.001997s	27
09	rdrdluuuurrdrddruuulldrulurr	0.048929s	28
10	ddrruuuullulluullllluurrddddrr	0.004001s	30
11	druurdrrdruuldluldldrdldrurdrurd	0.231336s	32

可以看出启发函数h2对于搜索效率提升明显，尤其是较为复杂的样例09和11搜索时间减少一个数量级。

问题二

变量集合共有7*工人数个，即每个工人每天是否工作是一个基本变量。值域集合是{0,1}，其中0代表休息，1代表工作。约束集合考虑如下约束：

1.每行至少两个0

2.每行不可出现连续3个0

3.每列至少m个1

4.每列至少一个senior

5.部分行之间同列不可同时为1

以设计表2为例，算法主要思路是先考虑每行至少两个0且不可出现连续3个0筛选出每行只有73种取值。再考虑最严的约束5，即第一行和第五行，第二行和第六行，第八行和第十行同列不可同时为1，每两行的联合取值只有138种。由于8，10都是senior且出现在之前的约束5所以容易在上一步基础上使8，10每天有一个在工作从而满足约束4。最后只剩约束3每列至少5个1，可以安排约束5产生的6行每列至少3个1，余下4行至少2个1。找到符合问题的一个解即可。

采用**约束传播**的优化思路，即将约束1和2产生的取值集合进一步传播到接下来的约束搜索；在8，10不可同时为1的约束前提下为了满足约束4从而考虑让8，10每天有一个在工作。此外，对于最难的约束3，采用**约束分割**的方法，即让其中6行每列至少3个1，余下4行至少2个1。这些优化对于搜索效率提升是可见的，没有优化的搜索时间在0.1s数量级，优化后的搜索时间在0.01s数量级。

模拟退火伪代码

```
while(current_state does not satisfy the constraint){
    Generate next_state;
    if(next_state satisfies more constraints than current_state)
        current_state <- next_state;
    else
        Executes{
            current_state <- next_state;
        }with probabilities p;
    delete next_state from SET;
}
```