人工智能基础实验 1 实验报告

PB20000180 刘良宇

AStar

启发函数

> 八连通区域或八邻域，是指对应像素位置的上、下、左、右，左上、右上、左下、右下 共 8 个紧邻的位置。
> 八连通分量，是指在八连通区域中，由相邻像素组成的连通区域。

下面给出我们启发函数的描述。

- 当前状态 s
- 锁盘上有 n 个八连通分量
- 每个八连通分量内「锁定」的拨轮的个数为 c_i

则启发函数为：

$$h_{temp}(s) = \sum_{i=1}^{n} \left\lceil \frac{c_i}{3} \right\rceil$$

$$h(s) = h_{temp}(s) + ((\sum_{i=1}^{n} c_i - h_{temp}(s)) \ \% \ 2)$$

首先分别估算每个八连通分量内需要「锁定」的拨轮个数，然后将这些个数相加，得到 $h_{temp}(s)$。

注意到最后的总操作次数应该和锁盘上总锁定的拨轮个数同奇同偶，因此我们将 $h_{temp}(s)$ 与锁盘上总锁定的拨轮个数的奇偶性相匹配，得到 $h(s)$。

这个函数是可采纳的，且是一致的。

可采纳性

- 对于每个八连通分量，我们需要至少 $\lceil c_i/3 \rceil$ 次操作才能将其完全解锁。
- 拨动操作不会跨越两个已有的八连通分量，因此每个拨动操作都可以被归入某个八连通分量内。
- 奇偶性的约束是显然的。两个约束同时满足，最终还是可采纳的。

一致性

因为每步操作的代价均为 1，所以只要证明一步操作不会使启发函数值下降超过 1，也就是 $h(s) - h(s') < 2$ 即可。

注意到每一步操作都会改变奇偶性，因此一定有 $h(s) - h(s') \neq 2k$。因此可以放宽到证明 $h(s) - h(s') < 3$。

每次解锁操作最多改变 3 个格子的状态；如果用于解锁格子，那么就不会减少连通片数，如果用于锁定格子，就会增加连通片内锁定的拨轮个数。

之前提到一个解锁操作不能跨越两个八连通分量，因此用于解锁格子最多减少一个操作，因此 $h(s) - h(s') < 3$。

主要思路

A* 启发函数

```
int search_eight_connected(int i, int j, bool marked[16][16]) const {
    if (marked[i][j] || !state.get(i, j))
        return 0;
    marked[i][j] = true;
    auto n = state.size();
    auto count = 0;
    static auto frontier = deque<pair<int, int>>{};
    frontier.clear();
    frontier.emplace_back(i, j);
    while (!frontier.empty()) {
        auto [i, j] = frontier.front();
        frontier.pop_front();
        count++;
        for (auto op = 0; op < 9; op++) {
            auto new_i = i - 1 + op / 3;
            auto new_j = j - 1 + op % 3;
            if (new_i < 0 || new_i ⩾ n || new_j < 0 || new_j ⩾ n)
                continue;
            if (marked[new_i][new_j] || !state.get(new_i, new_j))
                continue;
            marked[new_i][new_j] = true;
            frontier.emplace_back(new_i, new_j);
        }
    }
    return count;
}

[[nodiscard]] int heuristic() {
    auto n = state.size();
    auto cost = 0;
    auto total_count = 0;
    bool marked[16][16];
    for (auto i = 0U; i < n; i++)
        for (auto j = 0U; j < n; j++)
            marked[i][j] = false;
    for (auto i = 0U; i < n; i++) {
        for (auto j = 0U; j < n; j++) {
            auto count = search_eight_connected(i, j, marked);
            total_count += count;
            cost += (count + 2) / 3;
        }
    }
    if (cost % 2 ≠ total_count % 2)
        cost++;
    return cost;
}
```

使用 BFS 队列搜寻八连通分量并记录每个分量内部锁定的拨轮个数，然后计算启发函数。

这里的 `static` 声明限制了 `Solver` 类只能单线程使用，但是可以避免每次调用启发函数时都重新分配内存。

A* 搜索函数

因为使用了可采纳的启发函数，所以 A* 算法无需重复入队也一定能够找到最优解。

```
Actions solve() {
    auto explored = unordered_set<State, StateHash>{};
    auto pq = priority_queue<Node>({}, {start});
    auto next_nodes = vector<Node>{};

    while (!pq.empty()) {
        auto top = pq.top();
        pq.pop();
        if (top.get_h() == 0) [[unlikely]]
            return top.get_actions();
        if (explored.contains(top.get_state()))
            continue;
        explored.emplace(top.get_state());
        for (const auto &next : top.next_nodes(next_nodes))
            pq.push(next);
    }
    return {};
}
```

后继状态

这里也是值得重点优化的地方。事实上，对于每个状态，我们只需要给出一个能确其中存在最优解的状态的状态集合即可。这个集合的大小越小，搜索的速度就越快。

```
[[nodiscard]] pair<int, int> first_one_index() const {
    auto n = state.size();
    for (auto i = 0U; i < n; i++)
        for (auto j = 0U; j < n; j++)
            if (state.get(i, j))
                return {i, j};
    return {-1, -1};
}

[[nodiscard]] const vector<Action> &good_actions() const {
    auto n = state.size();
    auto [first_one_i, first_one_j] = first_one_index();
    static auto good_actions = vector<Action>{};
    good_actions.clear();
    auto i = first_one_i == n - 1 ? first_one_i - 1 : first_one_i;
    for (auto j : {first_one_j - 1, first_one_j}) {
        if (j < 0 || j >= n - 1)
            continue;
        auto dx = first_one_i - i;
        auto dy = first_one_j - j;
        for (auto not_op = 0; not_op < 4; not_op++) {
            if (dx * 2 + dy == not_op)
                continue;
            good_actions.emplace_back(i, j, not_op);
        }
    }
    return good_actions;
}
```

由于最优解必定会覆盖第一个（从上到下从左到右）找到的 1，所以我们只从这个 1 拓展所有可能的操作（共计 12 个），也一定能找到最优解。

而从上到下从左到右找到的第一个 1 还有一个性质，那就是它上面的行一定全为 0。因此 12 个操作中，涉及到上面行的 6 个也是可以不用展开的。因为假设某个最优解涉及到了上面行，由于上面行最终还是全为 0，因此一定有其他操作与它抵消。那么，将这些涉及到上面行的操作全部翻转到下面行，也一定能找到最优解（最后一行除外，最后一行没有下一行）。

优化与比较

使用启发函数后：

```
~/AI_lab/PB20000180_刘良宇_exp1/astar ./astar
input0: solution g=5, done in 0.028ms
input1: solution g=4, done in 0.013ms
input2: solution g=5, done in 0.025ms
input3: solution g=7, done in 0.02ms
input4: solution g=7, done in 0.064ms
input5: solution g=7, done in 0.164ms
input6: solution g=11, done in 0.331ms
input7: solution g=14, done in 0.172ms
input8: solution g=16, done in 0.114ms
input9: solution g=23, done in 8.502ms
```

设置 $h = 0$:

```
~/AI_lab/PB20000180_刘良宇_exp1/astar ./astar
input0: solution g=5, done in 0.092ms
input1: solution g=4, done in 0.02ms
input2: solution g=5, done in 0.099ms
input3: solution g=7, done in 0.621ms
input4: solution g=7, done in 0.474ms
input5: solution g=7, done in 0.709ms
input6: solution g=11, done in 84.021ms
input7: solution g=14, done in 141.432ms
input8: solution g=16, done in 649.972ms
input9: solution g=23, done in 7001.48ms
```

可以发现启发函数极大改善了性能，尤其是规模较大时，快了接近一千倍。

> 注：相比启发函数，减少后继结点数量带来的改进也是可观的。否则会造成极其严重的状态膨胀。

CSP

问题形式化

- 变量集合：$d \times s$ 个排班。
- 值域集合：n 个职员。
- 约束集合：
  - （硬约束）：每个职员至少分配到 $\lfloor d \times s / n \rfloor$ 个班次；
  - （硬约束）：任意连续两个班次职员不能相同；
  - （软约束）：最大化满足的排版数。

算法思路

因为数据特点（请求很多），我们可以使用贪心算法来求解。

```
int solve() {
    for (auto shift = 0; shift < d * s; shift++)
        assign_shift(shift / s, shift % s);

    while (!is_fair())
        adjust();

    auto sat_count = 0;
    for (int i = 0; i < d; ++i)
        for (int j = 0; j < s; ++j)
            if (requests[schedule[i * s + j] - 1][i][j])
                sat_count++;

    return sat_count;
}
```

首先最小剩余值分配班次，此时采用贪心算法尽可能满足排班请求，这会导致一些职员分配到的班次数不足均值。然后，我们调整这些职员的班次，使得他们的班次数达到均值。

优化方法

优先满足请求：

```
void assign_shift(int i, int j) {
    auto min_shift_staff_shifts = MAX;
    auto min_shift_staff = 0;
    for (auto staff : match_requests(i, j)) {
        if (not_consistent(i, j, staff) &&
            staff_shifts_count[staff - 1] < min_shift_staff_shifts) {
            min_shift_staff_shifts = staff_shifts_count[staff - 1];
            min_shift_staff = staff;
        }
    }

    if (min_shift_staff != 0) {
        schedule[i * s + j] = min_shift_staff;
        staff_shifts_count[min_shift_staff - 1]++;
```

```
    }
}
```

分配时首先考虑是否对应班次有请求。

```
[[nodiscard]] vector<int> match_requests(int i, int j) const {
    vector<int> request_staffs;
    for (int k = 0; k < n; ++k)
        if (requests[k][i][j])
            request_staffs.push_back(k + 1);

    if (request_staffs.empty())
        for (int k = 0; k < n; ++k)
            request_staffs.push_back(k + 1);

    return request_staffs;
}
```

运行时间：

```
$ time ./csp
./csp  0.03s user 0.00s system 99% cpu 0.035 total
```

input 0

如下所示：

```
1,2,3
1,3,2
3,2,1
3,1,2
1,2,1
3,2,3
1,2,3
20
```