人工智能基础实验 1

傅申 PB20000051

1. 实验环境

本次实验生成的程序在我的笔记本上运行, 其 CPU 为 Intel i5-1035G1, 带有 16 GB 双通道 DDR4 内存.

笔记本上运行的操作系统为 GNU/Linux 6.1.30-1-MANJARO x86_64. 在本次实验中, 将使用 Clang/Clang++ 作为编译器, 其版本为 0.15.2. 编译时的选项为 `-Ofast -std=c++17`. 比如编译实验 1.1 的代码使用的命令为: `clang++ astar.cpp -Ofast -std=c++17 -o astar`.

2. A* 实验

如无特殊说明, 该部分中将会称一次同时转动相邻呈 L 字形的三个拨轮为一次操作/动作.

2.1. 问题的特性

首先, 题中的问题是一定有解的, 考虑一个 2×2 的锁盘, 不论它的初始状态如何, 我们总是能将其解锁:

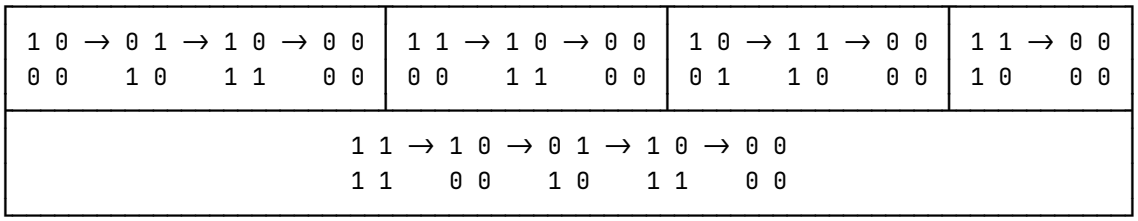

图 1: 2×2 锁盘的解法

对于任意大小的锁盘, 只需要将其分割为许多个 2×2 的锁盘, 按照上面的解法依次求解, 即可到达目标状态.

其次, 若要执行多个动作, 动作的顺序和最终状态是无关的. 因为顺序并不会影响每个拨轮被拨动的次数.

2.2. 启发式函数

状态记为 s. 因为我们需要找到操作数最少的解, 所以每步操作的代价相同, 定义为 1.

考虑一次操作前后锁盘上 1 的个数的变化, 记操作前后 1 的个数分别为 $n(s)$ 和 $n(s')$, 则只可能有四种情况:

$$n(s') = n(s) - 3 \quad n(s') = n(s) - 1 \quad n(s') = n(s) + 1 \quad n(s') = n(s) + 3$$

考虑到最终状态 $n_{\text{goal}} = 0$, 显然可以得到一个可采纳的启发式:

$$h(s) = \left\lfloor \frac{n(s)}{3} \right\rfloor + n(s) \bmod 3$$

这个启发式是可采纳的, 因为它 $n(s)$ 至少经过 $h(s)$ 次 ± 1 和 ± 3 变化才会变为 0. 实际上这个启发式函数也是一致的, 在上面的四种情况中, 分别有:

$$n(s')=n(s)-3\Rightarrow h(s')=\left\lfloor\frac{n(s')}{3}\right\rfloor+(n(s')\bmod 3)=\left\lfloor\frac{n(s)}{3}\right\rfloor-1+n(s)\bmod 3\geq h(s)-1$$

$$n(s')=n(s)-1\Rightarrow h(s')=\begin{cases}\left\lfloor\frac{n(s)}{3}\right\rfloor-1+n(s)\bmod 3+2\geq h(s)-1 & (3\mid n(s))\\ \left\lfloor\frac{n(s)}{3}\right\rfloor+n(s)\bmod 3-1\geq h(s)-1 & (3\nmid n(s))\end{cases}$$

$$n(s')=n(s)+1\Rightarrow h(s')=\begin{cases}\left\lfloor\frac{n(s)}{3}\right\rfloor+1+n(s)\bmod 3-2\geq h(s)-1 & (3\mid n(s)+1)\\ \left\lfloor\frac{n(s)}{3}\right\rfloor+n(s)\bmod 3+1\geq h(s)-1 & (3\nmid n(s)+1)\end{cases}$$

$$n(s')=n(s)+3\Rightarrow h(s')=\left\lfloor\frac{n(s')}{3}\right\rfloor+(n(s')\bmod 3)=\left\lfloor\frac{n(s)}{3}\right\rfloor+1+n(s)\bmod 3\geq h(s)-1$$

2.3. 剪枝策略

在实际实验中, 上面的启发式函数以及一些其他的启发式函数的尝试都没有收获很好的性能. 大多启发式函数要么不是可采纳的, 要么无法在输入规模较大的情况下在有限的时间和空间中运行. 为了解决这一问题, 可以尝试在求解过程中进行适当的剪枝.

前面提到, 在求解路径上, 动作是顺序无关的. 并且, 一次动作在当前锁盘上将 3 个 0 翻转为 1 看起来是不明智的, 因为它或许是最优解中的一步, 但是在求解过程中, 其他动作会将这 3 个 0 翻转为 1, 在有 0 被翻转为 1 之后在进行该动作是更合理的.

基于上面的两点, 可以让每个状态拓展的节点数减少到 12: 每次只拓展会翻转锁盘中某个特定的 1 的 12 个动作. 这个剪枝策略显然是正确的: 在到达目标状态的路径中, 一定有这 12 个动作中的某 (几) 个将这个 1 翻转为 0, 因为动作是顺序无关的, 只需要将这个动作调整到路径的最开始即可.

实际上, 如果约定翻转 1 的顺序, 拓展节点数可以进一步减少到 6. 假设我们逐行翻转 1. 在翻转第 k 行的 1 时, 第 1 行与第 $k-1$ 行之间全都是 0. 如果翻转第 k 行的 1 的同时翻转了第 $k-1$ 行的 1, 会带来额外的代价. 因此, 12 个动作中涉及上一行的 6 个动作是不必要的. 对于最后一行, 上面的讨论显然不成立了, 因为动作必然会涉及上一行. 但是, 最后一行本来就只能拓展不涉及下一行的 6 个动作. 因此, 所有节点只需要拓展 6 个节点即可. 具体的剪枝策略为:

- 从上到下, 从左到右, 逐行翻转 1.
- 当找到的 1 不在最后一行时, 执行只涉及下一行的 6 个动作并拓展. 否则, 执行只涉及上一行的 6 个动作并拓展.

在执行了剪枝策略后, 程序的性能有了极大的提升. 对于所有测试样例, 都能在 100 ms 内解决.

2.4. 程序实现

在程序的实现中, 我选择 `std::bitset` 作为存储锁盘的数据结构, 并自行实现了动作 `action` 类和节点 `node` 类. 程序的主体由 `solver` 类构成, 它将执行处理输入文件, 使用 A* 算法求解, 输出最优解, 验证解的正确性等任务. `solver` 类使得 `main` 函数变得相对简洁. 在 `main` 函数中, 程序将遍历 10 个输入文件, 生成相应的 `solver` 对象进行求解, 最后将最优解输出到文件中. 程序将记录求解各个输入的用时, 并验证其正确性, 将它们输出到标准输出流中.

```cpp
int main()
{
    for (int i = 0; i < CASE_NUM; i++) {
        ifstream input(INPUTS[i]);

        solver s = solver().from_file(input);

        auto start = now(); // auto now = std::chrono::high_resolution_clock::now;
        s.solve();
        auto end = now();

        ns duration = end - start; // using ns = std::chrono::nanoseconds;

        std::cout << "Time taken for " << INPUTS[i] << ": "
                  << duration.count() / 1e6 << " ms, "
                  << "solution verified: " << s.verify() << endl;

        ofstream output(OUTPUTS[i]);
        s.path_to(output);
    }

    return 0;
}
```

对于启发式函数, 使用 `std::bitset` 的 `count()` 函数得到锁盘中 1 的个数, 再进行处理即可:

```cpp
const size_t MAX_SIZE = 12;
using grid_t = std::bitset<MAX_SIZE * MAX_SIZE>;

unsigned heuristic(const grid_t &g)
{
    int count = g.count();
    return count / 3 + count % 3;
}
```

节点类 `node` 的定义如下, 节点将以树的形式被组织, `parent` 指针和 `from_parent` 动作是在找到最优解后回溯出路径的关键.

```cpp
class node
{
  public:
    shared_ptr<node> parent;       // parent node
    action           from_parent;  // action from parent
    unsigned         g;            // steps from the root node
    unsigned         h;            // heuristic value
    unsigned         f;            // h + g
    grid_t           grid;         // grid of the current node

    // ...
}
```

A* 算法的实现如下，其中使用了 `explored` 记录探索过的节点. 因为所有动作的代价是一致的，所以 `explored` 不会影响解的正确性. `get_chidrens` 会执行上述提到的剪枝策略，每次至多拓展 6 个节点.

```
class solver {
  public:

    // ...

    void solve()
    {
        auto root = make_shared<node>(grid, heuristic(grid));

        using node_p = shared_ptr<node>;

        priority_queue<node_p, vector<node_p>, greater<node_p>> frontier;
        frontier.push(root);

        unordered_set<grid_t> explored;

        while (not frontier.empty()) {
            auto current = frontier.top();
            frontier.pop();

            if (current->h == 0) {
                set_solution(current);
                return;
            }

            explored.insert(current->grid);

            for (const auto &[g, a] : get_childrens(current->grid)) {
                auto child = make_shared<node>(current, a, g, heuristic(g));

                if (explored.find(g) == explored.end()) {  // not explored
                    frontier.push(child);
                }
            }
        }

        // This should never be reached
        return;
    }

    // ...
}
```

2.5. 与 Dijkstra 算法的比较

执行上面的程序，在输出文件夹中，会生成 10 个输出文件，每个文件中有最优解的路径和其长度. 这 10 个最优解的长度依次为: 5, 4, 5, 7, 7, 7, 11, 14, 16, 23. 同时，在标准输出流中，会输出如下的结果:

```
$ ./astar
Time taken for ../input/input0.txt: 0.020454 ms, solution verified: 1
Time taken for ../input/input1.txt: 0.009553 ms, solution verified: 1
Time taken for ../input/input2.txt: 0.008058 ms, solution verified: 1
Time taken for ../input/input3.txt: 0.014212 ms, solution verified: 1
Time taken for ../input/input4.txt: 0.038196 ms, solution verified: 1
Time taken for ../input/input5.txt: 0.137338 ms, solution verified: 1
Time taken for ../input/input6.txt: 0.234596 ms, solution verified: 1
Time taken for ../input/input7.txt: 0.582743 ms, solution verified: 1
Time taken for ../input/input8.txt: 0.219054 ms, solution verified: 1
Time taken for ../input/input9.txt: 67.6683 ms, solution verified: 1
```

可以看到, 程序的运行时间较快, 前 9 个输入的求解时间都小于 1 ms.

将启发式函数的值修改为 1, 并修改 `solve()` 中对目标状态的判断为 `current->grid.count() == 0`, 就得到了 Dijkstra 算法. 执行修改后的程序, 程序在只有有限资源的情况下, 只能计算出前 9 个输入的最优解, 这前 9 个输入的最优解长度与 A* 算法的长度没有区别, 但是执行时间大大变长了:

```
$ ./astar
Time taken for ../input/input0.txt: 1.22082 ms, solution verified: 1
Time taken for ../input/input1.txt: 0.121491 ms, solution verified: 1
Time taken for ../input/input2.txt: 0.67177 ms, solution verified: 1
Time taken for ../input/input3.txt: 6.44446 ms, solution verified: 1
Time taken for ../input/input4.txt: 7.65852 ms, solution verified: 1
Time taken for ../input/input5.txt: 3.75263 ms, solution verified: 1
Time taken for ../input/input6.txt: 571.216 ms, solution verified: 1
Time taken for ../input/input7.txt: 4619.66 ms, solution verified: 1
Time taken for ../input/input8.txt: 18009.1 ms, solution verified: 1
zsh: killed     ./astar
```

3. CSP 实验

3.1. 问题的三个成分

实验问题作为一个 CSP 问题, 包含 3 个成分:

变量集合 $\boldsymbol{X}$ $D \times S$ 个班次 $\{X_1, \cdots, X_{DS}\}$.

值域集合 $\boldsymbol{D}$ 每个班次可能的取值都是 N 个宿管阿姨, 用编号表示为 $D_i = \{1, \cdots, N\}$.

约束集合 $\boldsymbol{C}$ 首先, 每个宿管阿姨不能工作连续两个班次: $\langle (X_i, X_{i+1}), X_i \neq X_{i+1} \rangle$; 其次, 为了公平性, 每个宿管阿姨至少被分配到 $\left\lfloor \frac{DS}{N} \right\rfloor$ 次值班, 记为 $\mathrm{Fair}(X_1, \cdots, X_{DS})$.

3.2. 算法

算法将使用回溯法, 从第一个班次开始, 依次为每个班次分配一个宿管阿姨. 算法会优先选择有请求的宿管阿姨, 并且为保证公平性, 会优先考虑分配班次最少的宿管阿姨. 如果没有请求, 则先跳过该班次, 最后再分配.

算法的具体实现如下, 其中 `patch()` 会尽可能公平地补全未被分配的班次, 并在最后检查是否满足公平性约束, 如果不满足, 则重新回溯, 直到找到一个解:

```
void solve() { backtrack(0); }


bool backtrack(int shift)
{
    if (shift == total_shift_number) {
        return patch(0);
    }

    auto cmp = [this, shift](const int &a, const int &b) {
        if (requests[a][shift] != requests[b][shift]) {
            return requests[a][shift] == true;
        }
        if (staff_assigned[a] != staff_assigned[b]) {
            return staff_assigned[a] < staff_assigned[b];
        }
        return staff_request[a] < staff_request[b];
    };
    auto staffs = staffs_;
    sort(staffs.begin(), staffs.end(), cmp);

    int best_staff = staffs.front();
    if (not requests[best_staff][shift]) {
        return backtrack(shift + 1);
    }

    for (auto &staff : staffs) {
        if (not requests[staff][shift]) {
            return false;
        }
        if (not valid(staff, shift)) {
            continue;
        }
        assign(staff, shift);
        request_satisfied += requests[staff][shift];
        if (backtrack(shift + 1)) {
            return true;
        }
        request_satisfied -= requests[staff][shift];
        cancel(staff, shift);
    }

    return false;
}
```

3.3. 优化方法

算法主要采用了 MRV 的思想. 为了最大化满足的请求数, 算法会主动选择有请求的宿管阿姨, 并且尽可能选择被分配班次少且请求数量少的宿管阿姨. 使用此优化方法, 算法能满足绝大多数的请求.

3.4. 结果

运行程序, 会在输出文件夹中写入相应的输出文件, 并在标准输出流中输出:

```
$ ./csp
../input/input0.txt: total shift number 21, request satisfied 20
../input/input1.txt: total shift number 60, request satisfied 60
../input/input2.txt: total shift number 33, request satisfied 33
../input/input3.txt: total shift number 114, request satisfied 114
../input/input4.txt: total shift number 69, request satisfied 69
../input/input5.txt: total shift number 576, request satisfied 576
../input/input6.txt: total shift number 1008, request satisfied 1008
../input/input7.txt: total shift number 378, request satisfied 378
../input/input8.txt: total shift number 2160, request satisfied 2160
../input/input9.txt: total shift number 720, request satisfied 720
```

程序会对安排的班次进行检查, 以使得最终的结果满足要求.

特别地, 对于 `input0.txt`, 程序在 `output0.txt` 中输出了:

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

对应的排班表如下, 其中灰色的为没有满足请求的排班:

宿管阿姨	第 1 天			第 2 天			第 3 天			第 4 天			第 5 天			第 6 天			第 7 天		
	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
1	✓					✓			✓			✓			✓			✓		✓	
2		✓		✓				✓			✓			✓			✓		✓		
3			✓		✓		✓			✓			✓			✓					✓