

exp1报告

Astar搜索

问题描述

二进制迷锁具有一个大小为 $N \times N$ 的拨轮锁盘, 每一个格点上具有一个可转动的拨轮, 上面刻着 数字 0和1 (表示非锁定) 和 (锁定)。由于拨轮之间相互链接的关系, 拨轮切换锁定的规则如下: 只能同时转动相邻呈“L”字形 (四个方向的朝向均可) 的三个拨轮, 将它们同时由各自的锁定切换为非锁定状态, 或从非锁定切换为锁定状态。

1. 第一种, 同时转动 $(1,1)$, $(1,2)$, $(0,1)$:

0	1	0		0	0	0
0	1	1	-->>	0	0	0 (成功解锁)
0	0	0		0	0	0

1. 第二种, 同时转动 $(1,1)$, $(0,1)$, $(1,0)$:

0	1	0		0	0	0
0	1	1	-->>	1	0	1
0	0	0		0	0	0

3. 第三种, 同时转动 $(1,1)$, $(1,0)$, $(2,1)$:

0	1	0		0	1	0
0	1	1	-->>	1	0	1
0	0	0		0	1	0

4. 第四种, 同时转动 $(1,1)$, $(2,1)$, $(1,2)$:

0	1	0		0	1	0
0	1	1	-->>	0	0	0
0	0	0		0	1	0

- 为这个问题设计一个合适的启发式函数, 并证明它是 admissible 的, 并论证其是否满足 consistent 性质。
- 根据上述启发式函数, 开发对应的 A* 算法找到一个解法, 将它恢复为全 状态以解开这个迷锁。
- 设置启发式函数为 0, 此时 A* 退化为 Dijkstra 算法, 比较并分析使用 A* 方法带来的优化效果。

实验过程

1. 启发式函数设计

首先是把单步cost设置为3

一共设置了4种启发式

- 可采纳的启发式 $h(lock) = n$, 其中n是1的个数。

可采纳性证明:

任意一个1, 至少都需要一步解锁, 即cost为3去复原, 而一次解锁最多复原3个1, 故无论如何, 解锁的代价都会大于等于估计的代价。因此 $h(n) \leq h'(n)$, 因此这个启发式是可采纳的

一致性证明:

$$\text{即证 } h(n) \leq 3 + h(n')$$

因为每一步至多减少3个1, 所以 $h(n') \geq h(n) - 3$, 证明显然成立

至于把每一步cost设置为3, 是因为这样对距离的刻画粒度更细, 可以更好的区分结点间的好坏

- 类似于贪心算法, $h(n) = 6 * \text{number_of_one}(n)$

这是非可采纳的启发式

设计这个启发式的思路是问题规模变大后, 会导致储存空间指数级上升, 因此, 减少深度和扩展的结点是极为重要的

这个启发式把1的个数乘以6, 在下一步扩展时就会优先扩展那些消去更多1的结点, 从而节约了空间

- 退化为Dijkstra算法, 直接令 $h(n) = 0$

- 依据结构的启发式

L形状的四种旋转赋予1的权重, 不构成L形状的, 但是两个连起来的1赋予2的权重, 单独的1赋予3的权重

思路:

L形状只需要1次翻转就能变成全0状态。

而两个连起来的1需要2次翻转(如[[1,1],[0,0]]型, 先对(1,0)做一次1型翻转, 得到[[0,1],[1,1]], 再对(1,1)做一次2型翻转, 得到复原态)

单独的1复原需要3次翻转(如[[1,0],[0,0]]型, 先对(1,0)做一次1型翻转, 得到[[0,0],[1,1]], 此后转换为2次的问题)

因此统计每种结构, 并赋予相应权值, 可以更细的划分启发式

但是设计一种没有误差的统计方式相当困难, 因此我采取了近似算法, 虽然可能有一些误差, 但是在可忍受范围之内。

这个函数是non-admissible的

2. 算法设计

- 结点设计

```
struct operation{
    short _i, _j, _k; //k = 1, L 型, 此后逆时针旋转
};
struct node{
    short _n; //n represent the size of lock max represent 64*64
    short _cost{};
    short _heuristic{};
    std::vector<std::vector<bool>>>_blocks{};
    std::vector<operation>_path{};
    node():_n(0){}
    node(short
n, vector<vector<bool>>>blocks, vector<operation>path):_n(n), _blocks(bloc
ks), _path(path){

    }
    bool operator<(const node &j)const{
        return _cost+_heuristic>j._cost+j._heuristic;
    }
};
```

这里用了一个2维的vector储存锁的格局, 同时, 每一个结点保存了之前的路径(带来约2倍的空间开销)

同时, 重载了node间的比较, 以放入优先队列中

- open list和close list的设计

```

long long getHashval(node &n){
    long long hashvalue = 0;
    long long base = 1;
    for (int i = 0; i < n._n; i++) {
        for (int j = 0; j < n._n; j++) {
            hashvalue = (hashvalue + base * n._blocks[i][j]) % MOD;
            base = (base * P) % MOD;
        }
    }
    return hashvalue;
}
priority_queue<node> open_list{};
unordered_set<long long>visited{};

```

`openlist` 用一个优先队列存放，以扩展最优的结点

`closeList` 用一个哈希表存放，以节约空间

o A star核心算法

参考ppt上算法

A* search {

closed list = []

open list = [start node]

```

do {
    if open list is empty then {
        return no solution
    }
    n = heuristic best node
    if n == final node then {
        return path from start to goal node
    }
    foreach direct available node do{
        if current node not in open and not in closed list do {
            add current node to open list and calculate heuristic
            set n as his parent node
        }
        else{
            check if path from star node to current node is
            better;
            if it is better calculate heuristics and transfer
            current node from closed list to open list
            set n as his parrent node
        }
        delete n from open list
        add n to closed list
    } while (open list is not empty)
}

```

}

`open_list.push(init);`

```

while(!open_list.empty()){
    node n = open_list.top();
    open_list.pop();
    if(is_goal(n)){
        flag = true;
        out<<n._path.size()<<'\n';
        for(auto &i:n._path){
            out<<i._i<<','<<i._j<<','<<i._k<<'\n';
        }
        out.close();
        break;
    }
    long long hash = getHashVal(n);
    visited.emplace(hash);
    for(int i=0;i<n._n;i++){
        for(int j=0;j<n._n;j++){
            for(int k=1;k<5;k++){
                if(is_valid_op(n,i,j,k)){
                    node newState = turn(n,i,j,k);
                    if(n._path.size()>35){
                        continue;//防止规模过大
                    }
                    long long val = getHashVal(newState);
                    if(visited.find(val)==visited.end()){
                        open_list.push(newState);
                    }
                }
            }
        }
    }
}
if(!flag){
    out<<"No valid solution.";
}

```

首先取出顶上结点，判断是否解锁，是，则终止搜索，并打印解

反之则将顶上结点做哈希放入closeList，并且以当前结点为基础，扩展出后面的结点

并且提前做出剪枝和限制搜索深度

- 剪枝算法

主要是判断了此次操作是否有意义

```

inline bool is_valid_op(node &n,short i,short j,short k){
    //判断这次操作是否合理
    if(i+dy[k]>=0&&i+dy[k]<n._n&&j+dx[k]>=0&&j+dx[k]<n._n){// in bound
        if(n._blocks[i][j]!=0 || n._blocks[i+dy[k]][j]!=0 ||
        n._blocks[i][j+dx[k]]!=0){
            return true;//没有1的旋转无意义
        }
    }
    return false;
}

```

判断是否越界。以及对3个0的反转无意义

- 启发式算法

一共3种

```

short h1(node &n){

```

```

//直接使用1的个数
short h{};
for(int i=0;i<n._n;i++){
    for(int j=0;j<n._n;j++){
        h += n._blocks[i][j];
    }
}
if(n._n<7){
    return h;
}
return h*6;
}
short h2(node&n){
    return 0;
}
short h3(node &n){
    short cnt = 0;
    set<pair<int,int>>visited;
    //先统计4种L形状的
    for(int i = 0;i<n._n;i++){
        for(int j = 0;j<n._n;j++){
            //当前块已被访问，则跳过
            if(!n._blocks[i][j]){
                continue;
            }
            if(visited.find({i,j})!=visited.end()){
                continue;
            }
            for(int k = 1;k<5;k++){
                if(is_valid_op(n,i,j,k)){
                    bool isDyvalid = n._blocks[i+dy[k]][j] &&
(visited.find({i+dy[k],j})==visited.end());
                    bool isDxvalid = n._blocks[i][j+dx[k]] &&
(visited.find({i,j+dx[k]})==visited.end());
                    if(isDxvalid && isDyvalid){
                        visited.insert({i+dy[k],j});
                        visited.insert({i,j+dx[k]});
                        visited.insert({i,j});
                        cnt += 1;
                        break;
                    }
                }
            }
        }
    }
}
//再统计两个连着的
for(int i = 0;i<n._n;i++){
    for(int j = 0;j<n._n;j++){
        //当前块已被访问，则跳过
        if(!n._blocks[i][j]){
            continue;
        }
        if(visited.find({i,j})!=visited.end()){
            continue;
        }
        for(int k = 1;k<5;k++){
            if(is_valid_op(n,i,j,k)){
                bool isDyvalid = n._blocks[i+dy[k]][j] &&
(visited.find({i+dy[k],j})==visited.end());
                if(isDyvalid){
                    visited.insert({i+dy[k],j});

```

```

        visited.insert({i,j});
        cnt += 2;
        break;
    }
    bool isDxValid = n._blocks[i][j+dx[k]] &&
    (visited.find({i,j+dx[k]})==visited.end());
    if(isDxValid){
        visited.insert({i,j+dx[k]});
        visited.insert({i,j});
        cnt += 2;
        break;
    }
    }
    }
}
//统计单独的
for(int i = 0;i<n._n;i++){
    for(int j = 0;j<n._n;j++){
        //当前块已被访问，则跳过
        if(n._blocks[i][j] && visited.find({i,j})==visited.end()){
            cnt += 3;
        }
    }
}
return cnt;
}

```

实验结果

1. 使用A*启发式算法1

在输入规模为7*7及以下时，采用可采纳的启发式，即 n ，在得到以下结果

```

D:\ai2023\USTC-AI2023\lab\exp1\astar\src> .\a.exe
Now Processing input0
time used:0.377293 second(s)
Max node count:128629
Now Processing input1
time used:0.004987 second(s)
Max node count:1957
Now Processing input2
time used:0.104047 second(s)
Max node count:41759
Now Processing input3
time used:5.25061 second(s)
Max node count:2032496
Now Processing input4
time used:11.1568 second(s)
Max node count:4000202
Now Processing input5
time used:5.5467 second(s)
Max node count:2056150

```

在input 4时扩展出了400万个结点，此时消耗内存为1.2GB

在输入规模更大时，使用可采纳的启发式会导致内存的不足，16GB的内存完全无法承受。

因此我只能选择求解出次优解，使用了6*n做启发式
得到input9有33步解
下为input0的解

```
5
0,1,4
0,1,3
0,2,4
1,1,3
2,3,2
```

2. 不使用启发式函数

名称	状态	30% CPU	94% 内存	0% 磁盘	0% 网络	3% GPU
a.exe		12.1%	9,218.8 MB	0 MB/秒	0 Mbps	0%

此图求解input0的内存消耗，由于规模过大，无法求解
可以对规模进行估算，input1是5*5的，每一个结点可以扩展出100个结点
因此扩展n层就是 100^n ，最优解是5步，也就是说可能扩展出100亿级别的结点，因此根本无法求解

3. 使用结构化的启发式函数，单步代价为1

```
Now Processing input0
time used:0.003989 second(s)
Max node count:1484
Now Processing input1
time used:0.001995 second(s)
Max node count:395
Now Processing input2
time used:0.009975 second(s)
Max node count:4444
Now Processing input3
time used:0.008975 second(s)
Max node count:3702
Now Processing input4
time used:0.008976 second(s)
Max node count:3674
Now Processing input5
time used:0.013962 second(s)
Max node count:6248
Now Processing input6
time used:0.08976 second(s)
Max node count:15958
Now Processing input7
time used:0.192485 second(s)
Max node count:31093
Now Processing input8
time used:0.233996 second(s)
Max node count:32369
Now Processing input9
time used:2.21124 second(s)
Max node count:256594
```

最多的仅仅扩展了25万个结点
比原有的朴素解法小了10倍
然而，如果将单步消耗改为如下的形式

```
#define COST (n._n<7)?3:1
```

则规模最大的输入仅仅扩展了4千个结点

```
Now Processing input0
time used:0.005984 second(s)
Max node count:2374
Now Processing input1
time used:0.000996 second(s)
Max node count:374
Now Processing input2
time used:0.002977 second(s)
Max node count:1353
Now Processing input3
time used:0.004987 second(s)
Max node count:2445
Now Processing input4
time used:0.006983 second(s)
Max node count:2786
Now Processing input5
time used:0.005983 second(s)
Max node count:2977
Now Processing input6
time used:0.004986 second(s)
Max node count:864
Now Processing input7
time used:0.009973 second(s)
Max node count:1460
Now Processing input8
time used:0.019948 second(s)
Max node count:2343
Now Processing input9
time used:0.045877 second(s)
Max node count:4415
```

不足之处是，输入较小时，得到的解比较复杂，因此最终采用了COST为1，启发式函数为启发式4的方案

csp问题

背景

学校新招募了一批宿管阿姨，不巧的是负责排班的管理人员生病请假了。你的任务是开发一个 CSP 算法，为学校的这批宿管阿姨安排一个值班表，以满足给定的约束条件，并尽可能满足阿姨们的轮班请求。

问题描述

你将获得以下信息：

- 宿管阿姨数量 (staff_num, N)
- 值班天数 (days_num, D)
- 每日轮班次数 (shifts_num, S)
- 轮班请求 $\text{Requests} \subset \{0, 1\}^{N \times D \times S}$

课程表必须满足以下约束条件：

1. 每天分为轮班次数个值班班次；
2. 每个班次都分给一个宿管阿姨，同一个宿管阿姨不能工作连续两个班次；
3. 公平起见，每个宿管阿姨在整个排班周期中，应至少被分配到 $\lfloor \frac{D \cdot S}{N} \rfloor$ 次值班。

你的目标是：

- 构造一个排班表 $\text{Shifts} \subset \{0, 1\}^{N \times D \times S}$
- 在满足上述约束的条件下，尽可能最大化满足的请求数，即
$$\max_{\text{Shifts}} \sum_{n \in N} \sum_{d \in D} \sum_{s \in S} \text{Requests}_{n,d,s} \times \text{Shifts}_{n,d,s}$$
- 请尽量最大化即可，但最终得分将考虑满足的请求数量

实验过程

1. 问题刻画

- 变量集合：总人数 N ，总天数 D ，一天班数 S ，阿姨 $A[N]$ ，排班表 $T[D \times S]$ ，排班请求 $R[N \times D \times S]$
- 值域集合：
$$R[i, j, k] \in \{0, 1\}$$
 代表阿姨 i 在 j 天 k 班是否想排到这一班
$$T[i, j] \in [1, N]$$
，代表第 i 天，第 j 班所排的阿姨
- 约束集合：
$$\forall i \in [0, D \times S], T[i] \neq T[i + 1] \quad (1)$$
 代表排班不能连续排一个阿姨
$$\forall i \in [1, N], \sum_{D, S} (T[D, S] = i) > \lfloor \frac{D \times S}{N} \rfloor \quad (2)$$
，代表最少排班要求的满足

2. 算法主要思路

先赋予排班表一个初始格局，之后通过local search找到最优。

1. 初始化

直接轮流分配每一个阿姨，这样直接就使得约束(2)成立

```
auto x = initList.begin();
for (int i = 0; i < D*S; i++)
{
    table.push_back(*x);
    if(++x == initList.end()){
        x = initList.begin();
    }
}
```

2. local search提高满足的排班数

不同于直接赋值修改结点的思路，而是采取结点交换的思路去优化排班

这是因为直接赋值修改结点的消耗过大，最大的输入下，扩展一层就会出现2160个新格局，同时，可以做出的选择也更多了，这使得计算机根本无法处理这种规模的问题。

而如果采取排班交换的方式，只需要每一次都达到一个更优的状态，直至无法再优化即可
伪代码如下

```
def localsearch()
    randomly choose x
    iter all schedule
    if improve satisfied number:
        exchange(x,i)
        return True
    return False
def modify()
    maxAttempt = 0
    while maxAttempt < 100
        if localsearch():
            maxAttempt = 0
        else:
            maxAttempt ++
```

3. 优化方法

1. 最少约束值优化

算法在读入数据时，会先做出一次排序，将选择了更多班次的阿姨排在前面，选择了更少班次的阿姨放在后面，然后在初始化排班时，先对更多班次的阿姨进行排班，再对少班次的阿姨进行排班。

这是因为，多班次的阿姨更可能得到满足，因此先将她们排班，就有了更多可满足的可能

```
sort(initList.begin(),initList.end(),[this](int aunt1,int aunt2)
{return aunt_count[aunt1]>aunt_count[aunt2]});
```

2. 前向检查

在运行local search时，对交换后的表就行约束检查，如果不符合约束则直接回退

```
if(!CheckSamework()){
    //破坏约束
    exchangeJob(failedIndex,i);
    continue;
}
```

核心代码

仅贴出local search部分

```
void cspSolver::localSearch(){
    int maxAttempt = 100;//最大失败尝试搜索次数
    int attempt = 0;

    random_device seed;
    std::mt19937 engine{seed()};
    while (attempt<maxAttempt)
    {
        if(notBestNum.empty()){
            //已经达到最优
            break;
        }
        //采用随机交换结点的方式！
```

```

//如果采用修改结点的方式,规模将会是无法承受的
uniform_int_distribution<> distrubution(0,notBestNum.size()-1);

//随机选一个不能满足的班次
int failedIndex = notBestNum[distrubution(engine)];
if(!job_aunt[failedIndex].size()){
    attempt+=1;
}
//遍历所有排班,尝试交换,是否有更优的分配
for(int i=0;i<D*S;i++){
    if(i==failedIndex || table[i]==table[failedIndex]){
        //不能自己换自己,也不能别人
        continue;
    }else{
        exchangeJob(failedIndex,i);
        if(!CheckSamework()){
            //破坏约束
            exchangeJob(failedIndex,i);
            continue;
        }else{
            //不破坏约束,则判断是否会使得当前状态更优,是则保留
            //现在i是原来的failed,failed是原来的i

            //表明原来的第i个是满足的
            int checkAuntISatification = static_cast<int>
(find(job_aunt[i].begin(),job_aunt[i].end(),table[failedIndex])!=job_aunt[i].
end());

            //检查现在的满足情况
            //原来的failed情况
            int checkFailed0 = static_cast<int>
(find(job_aunt[failedIndex].begin(),job_aunt[failedIndex].end(),table[failedI
ndex])!=job_aunt[failedIndex].end());
            //选取的i满足情况
            int checkI = static_cast<int>
(find(job_aunt[i].begin(),job_aunt[i].end(),table[i])!=job_aunt[i].end());

            if(checkI+checkFailed0>checkAuntISatification){
                //达到了一种更优的情况
                exchangeTimes++;
                if(checkFailed0){
                    auto it =
find(notBestNum.begin(),notBestNum.end(),failedIndex);
                    if(it == notBestNum.end()){
                        cerr<<(failedIndex)<<"error1\n";
                        for(auto i:notBestNum){
                            cerr<<i<<"\t";
                        }
                    }
                    notBestNum.erase(it);
                }
                if(checkI && !checkAuntISatification){
                    auto it =
find(notBestNum.begin(),notBestNum.end(),i);
                    if(it == notBestNum.end()){
                        cerr<<"error2";
                    }
                    notBestNum.erase(it);
                }
                attempt=0;
                break;
            }
        }
    }
}

```

```

        }else{
            //搜不到就回退
            exchangeJob(failedIndex,i);
        }
    }
}

}
attempt++;
}
}
}

```

实验结果

```

//前面的时间为0
Now processing input5
time used:0.001996 second(s)
Total exchange times:276
Now processing input6
time used:0.003026 second(s)
Total exchange times:468
Now processing input7
time used:0.000997 second(s)
Total exchange times:174
Now processing input8
time used:0.008975 second(s)
Total exchange times:1069
Now processing input9
time used:0.002993 second(s)
Total exchange times:362
All inputs used:0.110705 second(s)

```

可以看出，在大规模的输入时，仅仅交换了1000次即达到最优
下为output0和output8

```

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

```

```

//前略，但是为全满足
65,14,9,99,67,128
80,10,137,37,22,117
54,100,53,68,135,77
39,87,28,78,62,111
63,106,15,50,60,21
2160

```

已经均达到了最优的状况

测试脚本

```
import os
dx = [0,1,-1,-1,1]
dy = [0,-1,-1,1,1]
def checkAstar() :
    def change(x):
        if(x=='0'):
            return False
        return True
    def checkValidTurn(scale,i,j,k):
        if(i+dy[k]>=0 and i+dy[k]<scale and j+dx[k]>=0 and j+dx[k]<scale):
            return True
        return False
    def turn(configuration:list,i,j,k):
        assert(checkValidTurn(len(configuration),i,j,k)==True)
        configuration[i][j] = not configuration[i][j]
        configuration[i+dy[k]][j] = not configuration[i+dy[k]][j]
        configuration[i][j+dx[k]] = not configuration[i][j+dx[k]]
        return configuration

    for i in range(0,10):
        inputPath = './astar/input/input'+str(i)+'.txt'
        outputPath = './astar/output/output'+str(i)+'.txt'

        #get problem
        file = open(inputPath)
        problemScale = int(file.readline().replace('\n',''))
        configuration = []
        for line in file:
            configuration.append(list(map(change,line.split())))
        file.close()
        #get solution
        file = open(outputPath)
        totalStep = int(file.readline().strip())
        cnt = 0
        for line in file:
            cnt += 1
            op=list(map(int,line.split(',')))
            configuration = turn(configuration,op[0],op[1],op[2])
        file.close()
        assert(totalStep == cnt)
        for x in range(problemScale):
            for y in range(problemScale):
                assert(configuration[x][y] == False)
        print("astar-test" + str(i) + ' : passed')

def checkCsp():
    def sol(x):
        return int(x)-1
    def change(x):
        if x=='0':
            return False
        return True
    def checkSamework(table):
        for i in range(len(table)-1):
```

```

        if(table[i]==table[i+1]):
            return False
    return True
def checkMinwork(table:list,cnt,minReq):
    for i in range(cnt):
        if(table.count(i)<minReq):
            return False
    return True
def checkReqCnt(table:list,cnt,Req:list,totalJobs,expect):
    totalSAT = 0
    for i in range(totalJobs):
        if Req[i].count(table[i]) > 0:
            totalSAT += 1
    if(totalSAT == expect):
        return True
    return False

for i in range(0,10):

    inputPath = './csp/input/input'+str(i)+'.txt'
    outputPath = './csp/output/output'+str(i)+'.txt'

    #get problem
    file = open(inputPath)
    problemScale = list(map(int,file.readline().strip().split(',')))
    request = [[] for _ in range(problemScale[1]*problemScale[2])]
    workerCnt = 0
    dayCnt = 0
    for line in file:
        req = list(map(change,line.split(',')))
        if(workerCnt>=problemScale[0]):
            break
        #print(req)
        for schedule in range(problemScale[2]):
            if req[schedule] == True:

request[dayCnt*problemScale[2]+schedule].append(workerCnt)
            dayCnt += 1
            if dayCnt == problemScale[1]:
                workerCnt += 1
                dayCnt = 0
    file.close()
    #print(request)

    #get solution
    table = []
    file = open(outputPath)
    cnt = 0
    SAT = 0
    for line in file:
        if(cnt == problemScale[1]*problemScale[2]):
            SAT = int(line.strip())
            break
        cnt += problemScale[2]
        op=list(map(sol,line.split(',')))
        table.extend(op)
    file.close()
    assert(checkSamework(table) == True)

    assert(checkMinwork(table,problemScale[0],problemScale[1]*problemScale[2]//p
problemScale[0]) == True)

```

```
    assert(checkReqCnt(table,problemScale[0],request,problemScale[1]*problemScale[2],SAT) == True)
        print("csp-test" + str(i) + ' : passed')
#checkAstar()
print(os.getcwd())
checkAstar()
checkCsp()
```