Report for Lab02 - CSP

AC1 与 Revise函数分析

首先分析revise，直观来讲其作用为：

对一条 arc Xi->Xj 的首尾进行约束验证，检查Xi节点的每一个可行取值，只要有其中一个取值是要被舍弃的,就会返回True

函数中一些细节的理解通过注释标注在下方代码段中：

```
def revise(csp, Xi, Xj, removals, checks=0):
    """Return true if we remove a value."""
    revised = False
    for x in csp.curr_domains[Xi]:
        # If Xi=x conflicts with Xj=y for every possible y, eliminate Xi=x
        # if all(not csp.constraints(Xi, x, Xj, y) for y in
csp.curr_domains[Xj]):
        conflict = True
        for y in csp.curr_domains[Xj]:
            if csp.constraints(Xi, x, Xj, y): # 当节点Xi/Xj分别赋值x/y时，若符合约束则
此函数返回True
                conflict = False
            checks += 1
            # 只要Xj节点的取值中有一个是符合约束的，当前验证的Xi中的取值就是可以保留的，可以跳
到Xi的下一个取值进行验证
            if not conflict:
                break
        if conflict: # 如果Xj节点的每一个取值都违背约束，则Xi中的取值x要舍弃掉
            csp.prune(Xi, x, removals)
            revised = True
    return revised, checks
```

然后分析AC1函数，这里先按照指南中所说，假设传入ac1函数的queue里面存储了图中所有的arc，那么直观来讲其作用为：

循环遍历queue中所有的arc，用revise函数验证每一个arc，直到某次遍历没有舍弃任何节点的任何可行取值时返回True

如果在达到上述条件前某个节点的可行取值域被删空了，就返回False

为什么AC1在理论上比AC3低效?

直观来讲，AC3算法避免了重复验证"无需再次验证"的arc——因为当出现了revise时，并不是所有的arc都可能受到影响需要再验证一次，只有指向可行取值域发生变动的节点的arc需要重新验证，而AC1采取的策略就是前者，只要出现revise就再循环遍历一次所有arc，这必然会导致时间的浪费

下面从理论分析时间复杂度，设节点数为 n，边数为 e，arc数为 $2e$，最大的节点可行取值域有 d 个取值

显然revise函数的最坏时间复杂度为 $O(d^2)$，因为该函数中使用了2个循环；那么遍历一次queue的最坏时间复杂度就是 $O(2ed^2)$；而最坏情况下每次循环只能删掉一个可行取值，则一共要进行 nd 次遍历，因此忽略常数后**AC1算法的最坏时间复杂度为 $O(ned^3)$**

同理，AC3和AC1使用相同的revise函数，最坏时间复杂度均为 $O(d^2)$；在AC3中我们考虑最坏情况其实就是考虑一个arc最多能进入几次queue，而arc重新enqueue是由于其尾节点删除可行取值，一个节点最多删除 $d-1$ 个取值，因此算上初始时每条arc都最多在queue中出现 d 次；最坏情况下每次revise都只触发一条arc进栈，因此要执行 $2ed$ 次revise函数，因此忽略常数后**AC3算法的时间复杂度为 $O(ed^3)$**

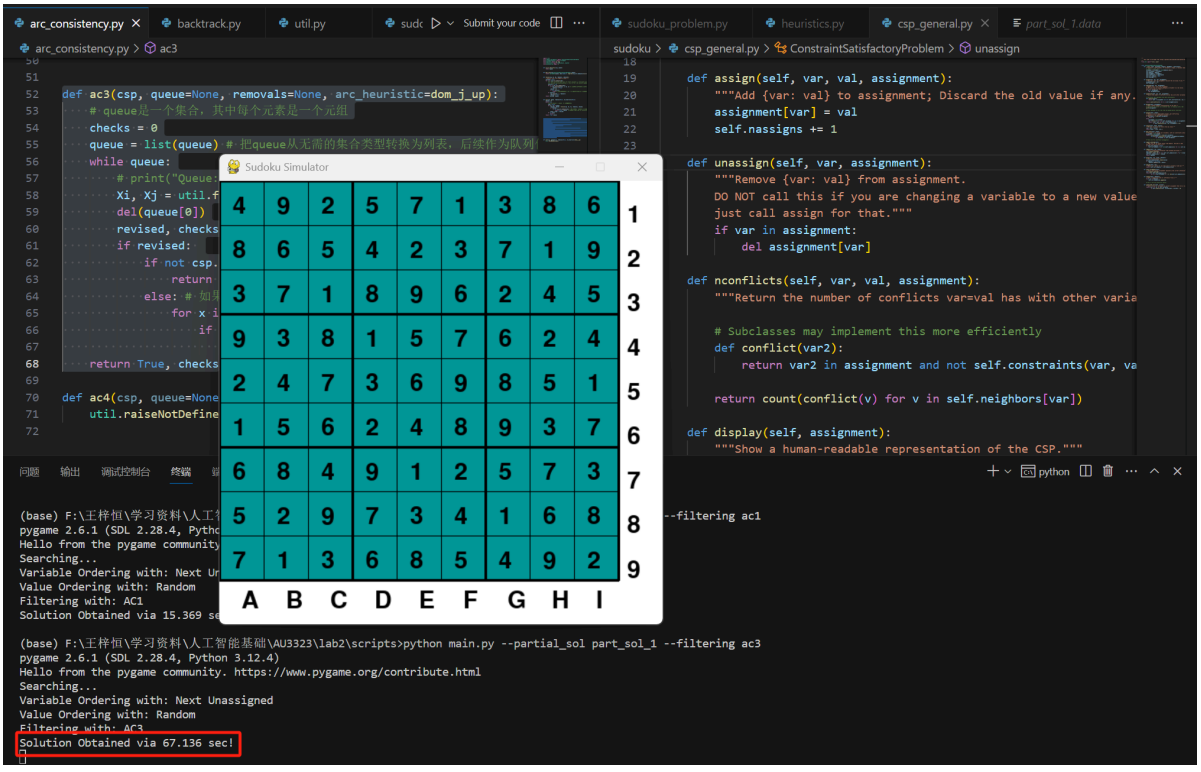
综上，笔者从直观认知和时间复杂度两个角度分析证明了AC1在理论上比AC3低效

AC3实现及实验结果

```
def ac3(csp, queue=None, removals=None, arc_heuristic=dom_j_up):
    # queue是一个集合，其中每个元素是一个元组
    checks = 0
    queue = list(queue) # 把queue从无需的集合类型转换为列表，后续作为队列使用
    while queue:
        # print("Queue:", queue)  # 打印队列状态
        Xi, Xj = util.first(queue)
        del(queue[0])
        revised, checks = revise(csp, Xi, Xj, removals, checks)
        if revised:
            if not csp.curr_domains[Xi]: # 如果节点Xi所有取值都违背约束，返回False触发回溯
                return False, checks
            else: # 如果Xi中有取值被删掉了，就要在queue中添加以Xi为tail的所有arc
                for x in csp.neighbors[Xi]:
                    if (x, Xi) not in queue: # 避免添加队列中已经存在的arc
                        queue.append((x, Xi))
    return True, checks
```

需要解释的重点都已在代码块中通过注释说明

实验结果如下图



启发式函数实现及实验结果

MRV

使用MRV方法之后，回溯搜索中每次选取讨论的节点时**优先选择可行取值域最小的节点**

需要解释的重点在代码块中通过注释说明

```
def mrv(assignment, csp):
    """Minimum-remaining-values heuristic."""
    # 在backtracking_search函数调用mrv函数前，csp.curr_domains还是None，所以此处要先初始化一下
    csp.support_pruning()
    min_domain = float("inf")
    # 遍历找出可行取值域最小的节点
    for var in csp.variables:
        if var not in assignment:
            if len(csp.curr_domains[var]) < min_domain:
                min_domain = len(csp.curr_domains[var])
                target_var = var
    return target_var
```

LCV

使用LCV方法之后，回溯搜索中每次选择assign什么值时**优先选择会造成冲突最少的可行取值**

```
def lcv(var, assignment, csp):
    """Least-constraining-values heuristic."""
    csp.support_pruning()
    dict = {} # 构建一个字典，键为var的可行取值，值为给var分配键对应的值时需要prune的次数
    for val in csp.curr_domains[var]:
        tmp_prune_cnt = 0
        csp.assign(var, val, assignment)
        for var_neighbor in csp.neighbors[var]:
            for val_neighbor in csp.curr_domains[var_neighbor]:
                tmp_prune_cnt += csp.nconflicts(var_neighbor, val_neighbor, assignment)
        dict[val] = tmp_prune_cnt
    # 按照dict中值的大小排序，返回键的升序队列
    sorted_keys = sorted(dict, key=lambda k: dict[k])
    return sorted_keys
```

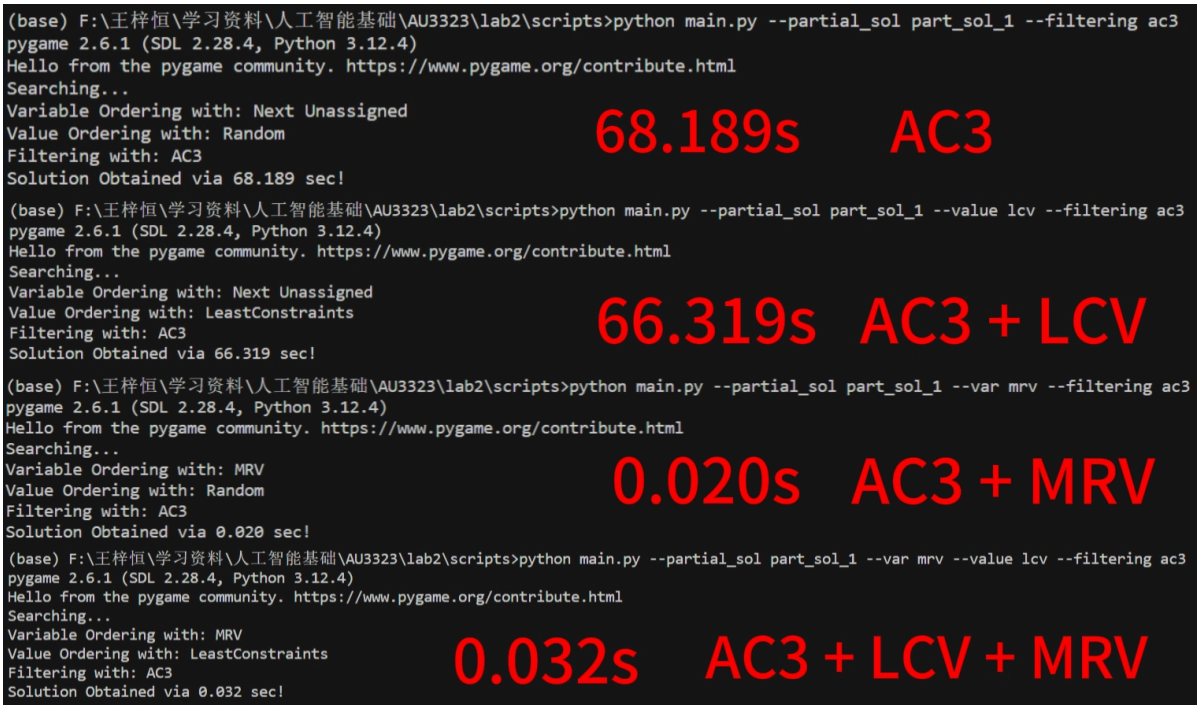
上述代码段中通过注释说明了部分重点，但下面要对nconflicts函数的使用进行补充说明：

nconflit函数检测当var_neighbor取val_neighbor时与【已经assign了值】的节点冲突的个数

而在这里只有var是assign了值的，所以相当于【检查var_neighbor取val_neighbor是否与var取value冲突】

若冲突tmp_prune_cnt值+1，若不冲突tmp_prune_cnt值不变

实验结果



对比求解运行时间发现：

1. 在AC3的基础上使用LCV或MRV都可以减小运行时间
2. 但使用**LCV只能小幅度提升效率**，而使用**MRV却能极大程度地提升效率**
3. 如果既使用LCV又使用MRV，反而比只使用MRV时运行时间长

这样的结果符合预期，因为课件上对LCV的描述为“Notably, this requires additional computation, but can still yield speed gains **depending on usage**”

Bonus

AC4实现及实验结果

```python
def ac4(csp, queue, removals=None, arc_heuristic=None):
    supports = defaultdict(set)        # 记录每个值的支持值
    counter = defaultdict(int)         # 记录每个值的有效约束计数
    processing_queue = deque()         # 用队列存储待处理的arc
    checks = 0                         # 约束检查次数

    for (Xi, Xj) in queue:
        for vi in csp.curr_domains.get(Xi, []):
            # 初始化计数器为0（假设当前值无支持）
            counter_key = (Xi, vi, Xj)
            counter[counter_key] = 0
            for vj in csp.curr_domains.get(Xj, []):
                checks += 1  # 统计约束检查次数
                if csp.constraints(Xi, vi, Xj, vj):
                    supports[(Xj, vj)].add((Xi, vi))
                    counter[counter_key] += 1
            # 如果当前值无支持，直接剪枝
            if counter[counter_key] == 0:
                csp.prune(Xi, vi, removals)
                processing_queue.append((Xi, vi))
```

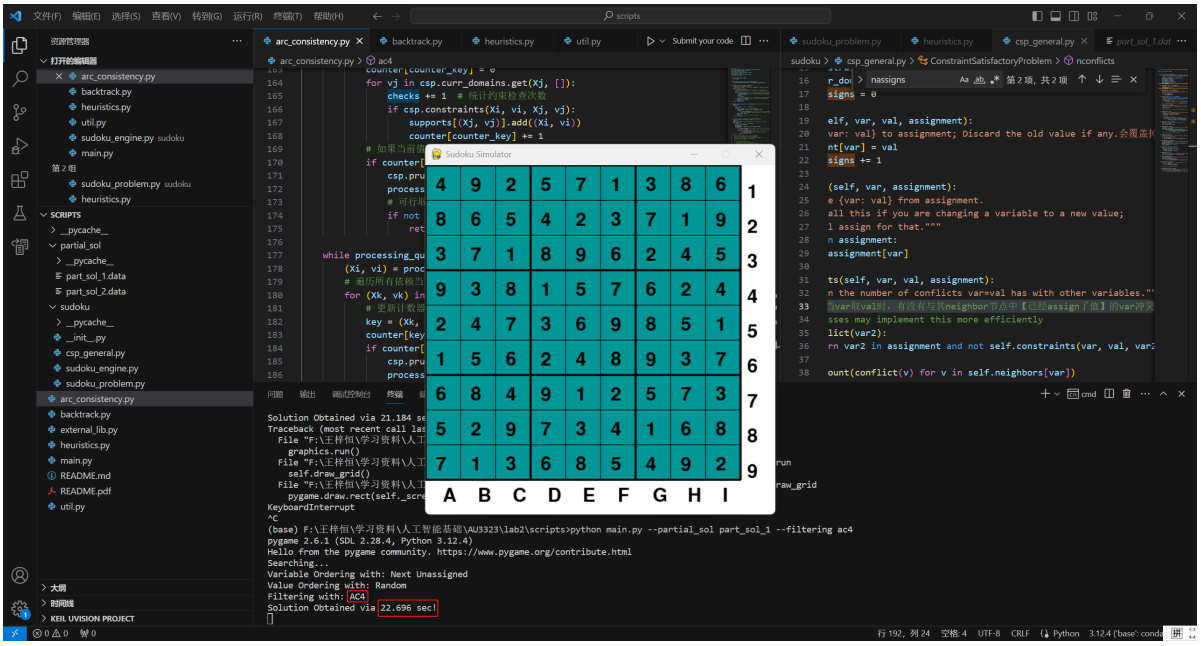
```
                # 可行取值域为空时返回失败
                if not csp.curr_domains.get(Xi):
                    return False, checks

    while processing_queue:
        (Xi, vi) = processing_queue.popleft()
        # 遍历所有依赖当前值的支持值
        for (Xk, vk) in supports.get((Xi, vi), set()):
            # 更新计数器并检查是否失效
            key = (Xk, vk, Xi)
            counter[key] -= 1
            if counter[key] == 0:
                csp.prune(Xk, vk, removals)
                processing_queue.append((Xk, vk))
                # 可行取值域为空时返回失败
                if not csp.curr_domains.get(Xk):
                    return False, checks
        checks += 1

    return True, checks
```

需要解释的重点都已在代码块中通过注释说明

实验结果如下图



如果在AC-4的基础上再加上MRV，实验发现与相比AC-3 + MRV效率更高

```
(base) F:\王梓恒\学习资料\人工智能基础\AU3323\lab2\scripts>python main.py --partial_sol part_sol_1 --var mrv --filtering ac4
pygame 2.6.1 (SDL 2.28.4, Python 3.12.4)
Hello from the pygame community. https://www.pygame.org/contribute.html
Searching...
Variable Ordering with: MRV
Value Ordering with: Random
Filtering with: AC4
Solution Obtained via 0.005 sec!
```