



中山大學  
SUN YAT-SEN UNIVERSITY

# 中山大学计算机学院

## 人工智能

### 本科生实验报告

(2022学年春季学期)

课程名称: Artificial Intelligence

教学班级	20 级计科一班	专业 (方向)	计算机科学与技术
学号	20337056	姓名	李昊伟

修改声明: 和2022年6月22日提交的13\_20337056相比, 本版本:

1. 修改了v1中PDDL测试文件中的test3和test4, 保证与下面报告中说明的输入格式一直。前一个提交版本将v1和v2的输入文件混同了。
2. 输出的动作序列以类似函数的形式表示, 其成员以键的字典序排序。比如stack输出为stack(a b x y),而domain中声明的顺序是stack(a x b y), 只有顺序不同, 不影响正确性。

## 一. 实验题目

实现一个简单的STRIPS规划器。

## 二. 实验内容

### 1. 算法原理

## 1. 原理1: STRIPS规划器是如何把一个规划问题转化为一个搜索问题的?

在课堂上,老师强调:规划问题本质上与搜索问题是相通的。什么是规划问题?其实就是给定一个初始状态,一个目标状态,想要找到一组正确的动作序列,不断地依次改变这个初始状态,直到达到目标状态。在我复习学期初学习搜索问题的课件时,发现这个定义与搜索问题本身的定义非常的相似。所以,我们可以把变化中的状态视为搜索问题的状态,把动作的加入使得状态变化视为搜索问题中状态的扩展,这样的话,就可以用搜索,特别是启发式搜索求解一个规划问题了。

## 2. 原理2: 什么是动作?

在上面一条说了,状态和状态之间的转化依靠的是动作。那么究竟什么是一个动作呢?

- 在现实生活中,我吃面条这是一个动作。
- 这个动作首先要有变量,因为吃这个行为可以是任何人发出,作用在任何食物上面的。
- 这里吃这个行为有两个变量,一个是我,这个变量类型是人,一个变量是面条,这个变量类型是食物。
- 我做这个动作还需要有前提,前提就是我有面条。这个前提必须在之前的状态中。
- 我做这个动作改变了状态。首先,增加了一个状态:我吃饱了。
- 第二,减少了一个状态:有面条。因为现在面条被我吃完了。

所以,我把动作做了一个对象类的定义,拥有以上几个成员属性,并定义格式化输出方式。这样,在编程与思考的过程中,可以更好地对动作做整体的处理,也方便我debug。以下是我对一个动作的定义输出,这个输出可以直接`print(action)`得到:

```
=====
action_name: move
parameters: {'p': 'npc', 'l1': 'cave', 'l2': 'river'}
preconditions: [('at', 'npc', 'cave'), ('border', 'cave', 'river'), ('n_guarded', 'river')]
effect_adds: [('at', 'npc', 'river')]
effect_dels: [('at', 'npc', 'cave')]
=====
```

## 3. 原理3: 如何构建启发式函数?

这次的启发式搜索不同于十五数码,难以找到一个直观的距离估计。所以,我们只能从启发式函数的本源:寻找松弛问题做起,看看对于一个规划问题,能不能找到一个松弛问题,很快得到它的解,把这个解当作启发式函数值。

- 我们可以这样考虑:如果一个动作只增加状态,不减少状态,这样松弛,可以吗?课堂上我们发现,这样一个问题的求解,也是NP-hard的,因为难以找到覆盖增加状态的最小动作子集。而且即使是找到了这样一个动作子集,这个启发式函数也不是可采纳的。因为我们挑出的只是在这个stage覆盖增加状态的最小动作集合,对于整个规划来说忽略了stage之间的相互影响,不一定是最优的。但countAction在实验中表现是可以的,而且启发度足够高,可以快速找到一条路径,即使不最优,也不会大的离谱。所以,我们仍然使用countAction作为启发式函数。
- count\_action算法:

## 2.关键代码展示

这里只展示最基本的功能代码，我的优化和细节代码将在下一部分分析

### 1.启发式函数

```
# 启发式函数count_action
def count_action(self, goals, state_layer, cur_layer):
    # 递归出口
    if cur_layer == 0:
        return 0
    # 划分Gp和Gn
    Gp = []
    Gn = []
    for goal in goals:
        if goal in state_layer[cur_layer-1]:
            Gp.append(goal)
        else:
            Gn.append(goal)
    # 根据状态获得有效动作，也就是动作层
    usable_actions = self.find_action(state_layer[cur_layer-1])
    action_list = []
    adds_list = []
    # 获得覆盖Gn的一个动作集合
    for valid_action in usable_actions:
        adds = valid_action.effect_adds
        for add in adds:
            if add in Gn:
                action_list.append(valid_action)
                for add in adds:
                    adds_list.append(add)
                break

        if set(Gn) <= set(adds_list):
            break
    # 获得下一状态层的目标
    goal_next = Gp.copy()
    for act in action_list:
        for prec in act.preconditions:
            if prec not in goal_next:
                goal_next.append(prec)
    # 递归
    return len(action_list) + self.count_action(goal_next,
state_layer, cur_layer-1)

def h_function(self, state):
    layers = self.build_layers(state)
    h = self.count_action(self.goal, layers, len(layers)-1)
    return h
```

## 2.A\*搜索

```
def A_star_search(self):
    start_time = time.time()
    # close列表
    close = []
    # open队列
    q = queue.PriorityQueue()
    h = self.h_function(self.init)
    # 初始结点
    init_node = Node(self.init, 0, h, self.goal, [])
    # 初始结点入队
    q.put(init_node)
    pop_time = 0
    # 开始搜索
    while not q.empty():
        head_node = q.get()
        pop_time += 1
        close.append(head_node.state)

        # 搜索成功, 退出
        if self.goal_achieved(head_node.state):
            end_time = time.time()
            print("===== result:
=====")
            for i in range(len(head_node.acts)):
                action = head_node.acts[i]
                print("step" , i , ": " , end='')
                self.standard_output(action)

            print("=====
=")

            print("plan_time: " , end_time - start_time ,
"seconds.")

            print('pop_times: ' , pop_time)

            print("=====
=")

            break
        # 扩展结点
        # 获得当前状态下有效动作
        valid_actions = self.find_action(head_node.state)
        # 依次扩展每个动作后的状态
        for action in valid_actions:
            new_state = head_node.state.copy()
            acts = head_node.acts.copy()

            for add in action.effect_adds:
                if add not in new_state:
```

```

        new_state.append(add)

    for _del in action.effect_dels:
        new_state.remove(_del)

    # 如果新结点不在close列表中, 也就是没有扩展过, 扩展它
    if new_state not in close:
        acts.append(action)
        h = self.h_function(new_state)
        new_node = Node(new_state, head_node.g+1 , h ,
self.goal , acts)

        q.put(new_node)
        close.append(new_state)

```

## 3. 创新点和实验改进

### 1. 改进1: 自己编写的paser.py


因为开始动手做的时间比较早, 所以没有用给定的pddl解析器, 而是自己写了一份parser代码。该解析器基于正则表达式。虽然在代码组织上对一些网络上的代码有借鉴和参考, 但大致上是自己实现的; 并且在最后一个改进中加入一个比较重要的模块, 使得最后一个改进成为可能。由于需要使用正则表达式, 所以对测试样例文本部分括号之间插入了空格, 比如

```

(:action attack
  :parameters (?p - player ?m - monster ?l1 - location ?l2 - location)
  :precondition (and (at ?p ?l1) (at ?m ?l2) (border ?l1 ?l2) (guarded ?l2))
  :effect ((not (at ?m ?l2)) (not (guarded ?l2)))
)

```

改为:

 屏幕截图 2022-06-23 11:42:16

除此之外, 对助教给定的测试样例无任何改动。

### 2. 改进2: 建立动作库——只做一次排列组合

在规划器的逻辑中, 有很多次使用了一个函数: `find_action()`。这个函数的功能是根据给定的状态, 获得前件已经被满足的所有动作。在构建layers, 还有搜索中扩展结点的过程中, 这都是非常重要的一步。

但是, 解析器解析的动作却只有动作名称和动作参数的类型, 比如`move(p - player, l - location)`, 而我们想要的却是`move(npc, field)`这样的动作。这就需要对动作进行解析和赋值, 需要用到排列组合的方法, 是一个复杂度较高的过程。

如果在每次调用`find_action()`的时候执行这个过程, 会非常麻烦, 而且有许多重复的运算。所以, 在这里, 我采取了打表大法。

在planner初始化的时候，根据所有动作的参数类型，和该problem里面该类型的所有对象，将所有动作可能的实例化都存储在动作列表之内。每一个实例化的动作用一个动作对象来存储(见实验原理2),:

```
class action:
    def __init__(self , action_name , parameters , preconditions ,
effect_adds , effect_dels):
        self.action_name = action_name
        self.parameters = parameters
        self.preconditions = preconditions
        self.effect_adds = effect_adds
        self.effect_dels = effect_dels

    def __str__(self):
        print("=====")
        print("action_name:" , self.action_name)
        print("parameters:" , self.parameters)
        print("preconditions:" , self.preconditions)
        print("effect_adds:" , self.effect_adds)
        print("effect_dels:", self.effect_dels)
        print("=====")
        return ""
```

比如test2, move总共有 $A_3^2$ 也就是12种组合，attack有 $1 \times 2 \times A_3^2$ 共24种组合，open有 $1 \times 2 \times 4$ 共8种组合，collect-earth和collect-fire各有 $1 \times 2 \times 2 \times 4$ 共16种组合，最后build-fireball有一种组合. 所以，实例化之后动作有77个. 把这77个动作提前组合好，等调用find\_action()的时候，直接挨个把action.preconditions和state作比较就行了。

这样需要实现一些排列组合，以及赋值的函数。

```
# 输入：形式参数及其类型
# 输出：形式参数的不同实例化方式，以字典的形式表示，比如{'player':'npc' ,
'11':'field' , '12':'town'}
def fill(self , target):
    result = []
    type_variable_map = dict()
    for type_group in self.objects:
        type_variable_map[type_group[-1]] = []
        for i in range(len(type_group) - 1):
            type_variable_map[type_group[-1]].append(type_group[i])

    type_variable_map1 = dict()
    for para in target:
        vtype = para[1]
        if vtype in type_variable_map1:
            type_variable_map1[vtype].append(para[0])
        else:
            type_variable_map1[vtype] = []
            type_variable_map1[vtype].append(para[0])
```

```

        for key in type_variable_map1:
            filled = self.permutation(type_variable_map1[key] ,
type_variable_map[key])
            result.append(filled)

merged_result = []
a = itertools.product(*result)
for it in a:
    merge = it[0].copy()
    for dic in it:
        merge.update(dic)
    merged_result.append(merge)
return merged_result

# 输入：某一特定类型的形式参数及其类型名称，如('l1' , 'l2' , 'location')
# 输出：形式参数的不同实例化方式，以字典的形式表示，比如{'l1':'field' ,
'l2':'town'}
def permutation(self , list1 , list2):
    result = []
    a = itertools.permutations(list2 , len(list1))
    for fill in a:
        dic = dict()
        for i in range(len(list1)):
            dic[list1[i]] = fill[i]
        result.append(dic)
    return result

# 遍历动作的前件、影响，构建涵盖所有可能动作的动作库
def instantiate_actions(self):
    filled_actions = []
    for i in range(len(self.actions['name'])):
        action_name = self.actions["name"][i]
        parameters = self.actions["parameters"][i]
        preconditions = self.actions["prec"][i]
        adds = self.actions['adds'][i]
        dels = self.actions['dels'][i]
        variable_map = self.fill(parameters)
        for map_version in variable_map:
            filled_preconditions = []
            filled_adds = []
            filled_dels = []
            for prec in preconditions:
                filled_prec = [prec[0]]
                i = 1
                while i < len(prec):
                    filled_prec.append(map_version[prec[i]])
                    i += 1
                filled_preconditions.append(tuple(filled_prec))

```

```

for add in adds:
    filled_add = [add[0]]
    i = 1
    while i < len(add):
        filled_add.append(map_version[add[i]])
        i += 1
    filled_adds.append(tuple(filled_add))

for _del in dels:
    filled_del = [_del[0]]
    i = 1
    while i < len(_del):
        filled_del.append(map_version[_del[i]])
        i += 1
    filled_dels.append(tuple(filled_del))

    new_action = action(action_name , map_version ,
filled_preconditions , filled_adds , filled_dels)
    filled_actions.append(new_action)

return filled_actions

```

据实验，该方法可以有效减少规划所用时间，对于test2:

改进前时间	改进后时间
0.0981s	0.0259s

### 改进3：对松弛问题的再思考

在我们求解松弛问题时，我们忽略了动作的前件中所有的负原子。比如(not\_guarded\_field)这样的负原子。并且忽略了动作本身影响当中的负原子，不会把任何的状态从状态层中删去。

我想要做这样的改进，以使得启发式函数值对代价的预测更精确，更具有“启发性”：

- 在确定某一动作是否有效时，如果该动作的前件中有负原子，要求这个负原子至少在初始状态中或者之前添加的状态的影响中出现过。
- 比如某个动作的前件中有not\_guarded(field)，那么我要求要不是初始状态中有not\_guarded(field)，要不就是之前添加的动作的effect\_dels中有guarded(field)。
- 如何实现呢？我们将负原子也看成正原子，比如删除guarded(field)，就相当于在状态中删去guarded(field)，添加not\_guarded(field)。而添加guarded(field)，就相当于在状态中添加guarded(field)，删除not\_guarded(field)。这样的话，如果松弛问题中只添加，不删除，就会出现guarded(field)和not\_guarded(field)同时存在的情况。
- 而如果不做这个改进，我们只知道现在存在guarded(field), not\_guarded(field)存在过吗？不知道。



- 这样同时需要根据封闭世界法则，对初始状态init做一个增强，将所有没有出现过的子句都冠以not\_，添加到初始状态中去。
- 由于需要分析子句增强init, 所以必须把名称相同而参数类型不同的逻辑表达式视为两种类型，比如on(block , block)和on(block , table). 所以，我对输入做出较小修改，将后者改为on\_table(block , table)。当然，我也可以采取C语言编译器的方法，把函数名和参数类型组合作为真正的函数名。这里简便起见我直接修改了输入。

```
(:action stack
:parameters (?a - block ?b - block ?t1 - table)
:precondition (and (block ?a) (block ?b) (table ?t1) (clear ?a) (clear ?b) (on_table ?a ?t1) (on_table ?b ?t1) )
:effect (and (on ?a ?b) (not (on_table ?a ?t1)) (not (clear ?b)))
)
```

- 实现代码如下：

# 位于paser中，获得初始状态、前件、影响中所有的子句及其变量类型，为增强初始状态做准备

```
def make_pred_list(self):
    # preds = []
    # for i in self.preconditions:
    #     preds += i
    for i in range(len(self.preconditions)):
        preds = self.preconditions[i]
        parameters = self.parameters[i]
        for it in preds:
            # print(it)
            if it[0][0] == 'n' and it[0][1] == 'o' and it[0][2]
            == 't':
                pred = ''
                for i in range(len(it[0])):
                    if i > 3:
                        pred += it[0][i]
                else:
                    pred = it[0]
                if (pred not in self.predicate_list):
                    tmp = []
                    for i in range(len(it)):
                        if i > 0:
                            vtype = self.find_type(it[i] ,
parameters)
                            tmp.append((str(i) , vtype))
                    self.predicate_list[pred] = tmp

    for i in range(len(self.effect_adds)):
        preds = self.effect_adds[i]
        parameters = self.parameters[i]
        for it in preds:
            # print(it)
            if it[0][0] == 'n' and it[0][1] == 'o' and it[0][2]
            == 't':
                pred = ''
```

```

        for i in range(len(it[0])):
            if i > 3:
                pred += it[0][i]
            else:
                pred = it[0]
        if pred not in self.predicate_list:
            tmp = []
            for i in range(len(it)):
                if i > 0:
                    vtype = self.find_type(it[i] ,
parameters)

                    tmp.append((str(i) , vtype))
            self.predicate_list[pred] = tmp

    for i in range(len(self.effect_dels)):
        preds = self.effect_dels[i]
        parameters = self.parameters[i]
        for it in preds:
            # print(it)
            if it[0][0] == 'n' and it[0][1] == 'o' and it[0][2]
== 't':

                pred = ''
                for i in range(len(it[0])):
                    if i > 3:
                        pred += it[0][i]
                    else:
                        pred = it[0]

                if pred not in self.predicate_list:
                    tmp = []
                    for i in range(len(it)):
                        if i > 0:
                            vtype = self.find_type(it[i] ,
parameters)

                            tmp.append((str(i) , vtype))
                    self.predicate_list[pred] = tmp

```

# 位于parser, 对一个逻辑表达式取反, 比如把not\_guarded(field)变为guarded(field), 或者反过来。

```

def neg(self , pre):
    head = pre[0]
    new_pre = []
    if head[0] == 'n' and head[1] == 'o' and head[2] == 't':
        new_head = ''
        for i in range(len(head)):
            if i > 3:
                new_head += head[i]
        new_pre.append(new_head)
    for i in range(len(pre)):

```

```

        if i > 0:
            new_pre.append(pre[i])
    else:
        new_head = "not_" + head
        new_pre.append(new_head)
        for i in range(len(pre)):
            if i > 0:
                new_pre.append(pre[i])
    return tuple(new_pre)

```

```

# 位于planner,增强初始状态
def init_pro(self):
    for key in self.predicate_list:
        filled_para = self.fill(self.predicate_list[key])
        for para in filled_para:
            items = para.items()
            tmp = [key]
            for k , value in items:
                tmp.append(value)
            new_pred = tuple(tmp)
            if new_pred not in self.init:
                head = "not_" + new_pred[0]
                tmp = [head]
                for i in range(len(new_pred)):
                    if i > 0:
                        tmp.append(new_pred[i])
                self.init.append(tuple(tmp))

```

经过实验，该改进可以减少大部分搜索的扩展次数：

测例	改进前扩展次数	改进后扩展次数
test0	3	3
test1	4	4
test2	15	11
test3	8	5
test4	3	3
test5(我自己添加的上次作业的八数码问题)	13	14

但是，该改进因为加入大量新的逻辑表达式，运算上会减慢，也是一个trade\_off.

## 对A\*本身的改进

另外，我还做了一些对A\*算法本身的改进，比如将优先队列当中同等f条件下h函数值较大，深度较小的放在前面，这样就会优先扩展深度较小的结点，这一思路是来自于学期初两位同学对project1的pre。此举将test2的21次扩展减少到了15次，时间减少了三分之一。但由于这不是针对规划问题的优化，所以不再赘述。

## 4. 实验结果

实验对test0-test4进行了测试，还增加了上次作业的八数码问题作为test5. 由于上次作业的block问题有forall之类的句子不好处理，所以暂未添加入本次测例。

### v1:未应用改进3

测例	扩展次数	规划时间	是否最优
test0	3	<0.001s	是
test1	4	<0.001s	是
test2	15	0.025s	是
test3	8	0.002s	是
test4	3	0.002s	是
test5	13	0.35s	是

### v2:应用改进3

测例	扩展次数	规划时间	是否最优
test0	3	<0.001s	是
test1	4	<0.001s	是
test2	11	0.044s	是
test3	5	0.001s	是
test4	3	0.001s	是
test5	14	1.003s	是

可以看到，对于有负前提的样例，v2在扩展次数上表现较好。但v2将前件与状态比较时由于需要比较的太多，大量耗时，所以在某些样例的时间表现上不如v1.

测试图详见./result文件夹。

## 三. 实验心得

认认真真做完最后一个期末大作业，感觉它就像整个学期人工智能实验课的一个缩影：充实，有趣而又具有启发性。感谢认真的老师，感谢负责优秀的助教，这门课教给我的，我永远不会忘记。

虽然前两次大实验，特别是第一次大实验，我因为理解错了大作业和小作业的关系，没有重视实验报告，导致分数很低，但是我完成实验的过程是更加可贵的，我也确实收获不少，不管是知识上，还是代码能力上。

来到中大两年了，无论是数电实验，还是计组实验，还是这学期的人工智能和操作系统实验，我都觉得非常的有趣，我也投入了大量的时间和精力。有人说实验课学分少，事情多，而我觉得没有实验课，理论课还有什么意思呢？感谢助教耐心的引导，这门课和数电实验、计组实验、操作系统实验一起，构成了我大学两年最大的收获之一。