8. Goal Stack Planning

17/10/2019

INTRODUCTION

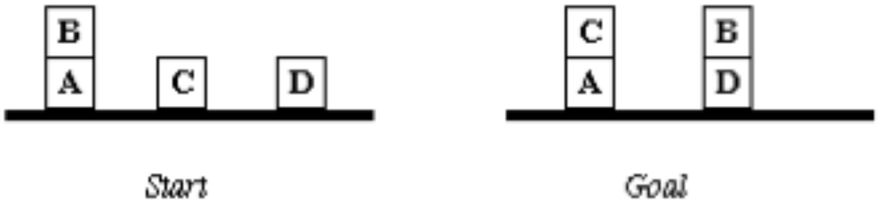
One of the earliest techniques is planning using goal stack. Problem solver uses single stack that contains

- sub goals and operators bots
- sub goals are solved linearly and then finally the conjoined sub goal is solved

Plans generated by this method will contain complete sequence of operations for solving one goal followed by complete sequence of operations for the next etc.

Problem solver also relies on

- A database that describes the current situation
- Set of operators with precondition, add and delete lists



IMPLEMENTATION THROUGH DATA STRUCTURE

Stack data structure is used for implementing goal stack planning algorithm.

I. Start by pushing the original goal on the stack. Repeat this until the stack becomes empty. If stack top is a compound goal, then push its unsatisfied subgoals on the stack.

II. If stack top is a single unsatisfied goal then, replace it by an action and push the action's precondition on the stack to satisfy the condition.

III. If stack top is an action, pop it from the stack, execute it and change the knowledge base by the effects of the action.

IV. If stack top is a satisfied goal, pop it from the stack.

- We can see that the first goal is achieved block C is on the table.
- The second goal is also achieved block C is clear.
- Remember that HOLDING(B) is still true which means that the arm is not empty. This can be achieved by placing B on the table or planting it on block D if it is clear.
- Lookahead could be used here to compare the ADD lists of the competing operators with the goals in the goal stack and there is a match with ON(B,D) which is satisfied by STACK (B,D). This also binds some block to block D.
- Applying STACK (B,D) generates extra goals CLEAR(D) and HOLDING(B).

The new goal stack becomes;

CLEAR(D)
HOLDING(B)
CLEAR(D) $\wedge$ HOLDING(B)
STACK (B, D)
ONTABLE(C) $\wedge$ CLEAR(C) $\wedge$ ARMEMPTY
PICKUP(C)

- At this point the top goal is true and the next and thus the combined goal leading to the application of STACK (B,D), which means that the world model becomes
  ONTABLE(A) ONTABLE(C) ONTABLE(D) ON(B,D) ARMEMPTY

- This means that we can perform PICKUP(C) and then STACK (C, A).
- Now coming to the goal ON(B, D) we realise that this has already been achieved and checking the final goal we derive the following plan

1. UNSTACK(B, A)
2. STACK(B, D)
3. PICKUP(C)
4. STACK(C, A)

PYTHON IMPLEMENTATION

```
import re

class block(object):
    def __init__(self, name, prop={}):
        self.name = name
        self.props = prop
    def add_prop(self, name, value=False):
        self.props[name] = value
    def check_prop(self, name):
        if name in self.props:
            return self.props[name]
        else:
            return False
    def set_prop(self, name, value):
        self.props[name] = value

class robotarm(object):
    def __init__(self):
        self.empty = True
        self.holding = '#'
    def pickup(self, block_name):
        self.holding = block_name
        self.empty = False
```

```python
    def put_down(self, block_name):
        self.holding = '#'
        self.empty = True
    def is_empty(self):
        return self.empty

PREDICATES = ['ON', 'ONTABLE', 'HOLDING', 'CLEAR', 'ARMEMPTY']

def check_p(to_check, claw, current_state=[]):
    tst = to_check[to_check.index('(')+1:to_check.index(')')]
    block_names = re.split(',', tst)
    iterar = True
    cont = 1
    if to_check.startswith('ONTABLE'):
        pass
    elif to_check.startswith('ON'):
        cont = 2
    elif to_check.startswith('CLEAR'):
        cont = 3
    elif to_check.startswith('HOLDING'):
        iterar = False
        cont = 4
    elif to_check.startswith('ARMEMPTY'):
        iterar = False
        cont = 5

    if iterar:
        if cont == 1:
            for torre in current_state:
                if torre[0] == block_names[0]:
                    return True
        elif cont == 2:
            for torre in current_state:
                if block_names[0] in torre:
                    ind = torre.index(block_names[0])
                    if ind > 0:
                        under = torre[ind-1]
                        if block_names[1] == under:
                            return True
        elif cont == 3:
            for torre in current_state:
                if block_names[0] in torre:
                    ind = torre.index(block_names[0])
                    if ind == (len(torre)-1):
                        return True
        return False
    else:
        if cont == 4:
```

```python
            if claw.is_empty() is False:
                if claw.holding == block_names[0]:
                    return True
        elif cont == 5:
            return claw.is_empty()
        return False

def get_relevant_actions(to_check, current_state, claw):
    tst = to_check[to_check.index('(')+1:to_check.index(')')]
    block_names = re.split(',', tst)
    to_execute = []
    b1_name = block_names[0]
    if to_check.startswith('ONTABLE'):
        to_execute.append('PUTDOWN({})'.format(b1_name))
        to_execute.append('HOLDING({})'.format(b1_name))
        for torre in current_state:
            if b1_name in torre:
                ind = torre.index(b1_name)
                to_execute.append('UNSTACK({},{})'.format(b1_name, 
torre[ind-1]))
                to_execute.append('ARMEMPTY(@)')
                to_execute.append('CLEAR({})'.format(b1_name))
                to_execute.append('ON({},{})'.format(b1_name, torre[ind-1]))
    elif to_check.startswith('ON'):
        to_execute.append('STACK({},{})'.format(b1_name, block_names[1]))
        to_execute.append('HOLDING({})'.format(b1_name))
        to_execute.append('CLEAR({})'.format(block_names[1]))
    elif to_check.startswith('CLEAR'):
        for torre in current_state:
            if b1_name in torre:
                ind = torre.index(b1_name)
                to_execute.append('UNSTACK({},{})'.format(torre[ind+1], 
b1_name))
                to_execute.append('ARMEMPTY(@)')
                to_execute.append('CLEAR({})'.format(torre[ind+1]))
                to_execute.append('ON({},{})'.format(torre[ind+1], b1_name))
    elif to_check.startswith('HOLDING'):
        to_execute.append('PICKUP({})'.format(b1_name))
        to_execute.append('ARMEMPTY(@)')
        to_execute.append('ONTABLE({})'.format(b1_name))
        to_execute.append('CLEAR({})'.format(b1_name))
    elif to_check.startswith('ARMEMPTY'):
        to_execute.append('PUTDOWN({})'.format(claw.holding))
        to_execute.append('HOLDING({})'.format(claw.holding))
    return to_execute
```

```python
def apply_action(to_apply, current_state, claw):
    tst = to_apply[to_apply.index('(')+1:to_apply.index(')')]
    block_names = re.split(',', tst)
    b1_name = block_names[0]
    if to_apply.startswith('STACK'):
        for torre in current_state:
            if b1_name in torre:
                torre.remove(b1_name)
            if block_names[1] in torre:
                torre.append(b1_name)
        claw.put_down(claw.holding)
    elif to_apply.startswith('UNSTACK'):
        for torre in current_state:
            if b1_name in torre:
                torre.remove(b1_name)
        claw.pickup(b1_name)
    elif to_apply.startswith('PICKUP'):
        claw.pickup(b1_name)
        for torre in current_state:
            if b1_name in torre:
                torre.remove(b1_name)
    elif to_apply.startswith('PUTDOWN'):
        claw.put_down(b1_name)
        nt = [b1_name]
        current_state.append(nt)

    return current_state

INPUT = open('initial.txt', 'r')
STATE1 = []
for linea in INPUT:
    tmp_list = re.split(r'\W+', linea)
    if '' in tmp_list:
        tmp_list.remove("")
    STATE1.append(tmp_list)
INPUT.close()

INPUT2 = open('final.txt', 'r')
STATE2 = []
for linea in INPUT2:
    tmp_list = re.split(r'\W+', linea)
    if '' in tmp_list:
        tmp_list.remove("")
    STATE2.append(tmp_list)
INPUT2.close()
```

```python
def solve():
    my_stack = []
    plan = []
    current_state = STATE1
    blocks = []
    for line in STATE1:
        mi_len = len(line)
        for i in range(mi_len):
            on_table = False
            clear = True
            ntmp = line[i]
            if i+1 < mi_len:
                clear = False

            if i < 1:
                on_table = True
            else:
                pass
            props = {'name': ntmp, 'onTable': on_table, 'clear':clear}
            btmp = block(ntmp, props)
            blocks.append(btmp)

    complex_goal = ''
    for line in STATE2:
        mi_len = len(line)
        state = ''
        for i in range(mi_len):
            ntmp = line[i]
            if i-1 >= 0:
                state += 'ON({},{})^'.format(ntmp, line[i-1])
            if i < 1:
                state += 'ONTABLE({})^'.format(ntmp)
        complex_goal += state
    complex_goal = complex_goal[:-1]
    sub_goals = re.split(r'\^', complex_goal)
    my_stack.append(complex_goal)

    for goal in sub_goals:
        my_stack.append(goal)

    my_claw = robotarm()
    while my_stack:
        for torre in current_state:
            if len(torre) < 1:
                current_state.remove(torre)
        accion_actual = my_stack.pop()
        ind = accion_actual.index('(')
        pred = '@'
```

```
        if ind > 0:
            pred = accion_actual[:ind]
        print accion_actual
        if pred in PREDICATES:
            if '^' in accion_actual:
                tmp_gls = re.split(r'\^', accion_actual)
                cumplido = True
                for meta in tmp_gls:
                    cumplido = cumplido and check_p(meta, my_claw,
current_state)
            else:
                cumplido = check_p(accion_actual, my_claw, current_state)
                if cumplido == False:
                    rest = get_relevant_actions(accion_actual, current_state,
my_claw)
                    my_stack.extend(rest)
        else:
            current_state = apply_action(accion_actual, current_state, my_claw)
            plan.append(accion_actual)
    print('**********************')
    print(plan)

solve()
```

ADVANTAGES AND DISADVANTAGES OF LINEAR PLANNING

- Reduced search space, since goals are solved one at a time.
- Advantageous if goals are (mainly) independent.
- Linear planning is sound.
- Linear planning may produce suboptimal solutions (based on the number of operators in the plan).
- Linear planning is incomplete.

ADVANTAGES AND DISADVANTAGES OF NON-LINEAR PLANNING

- This planning is used to set a goal stack and is included in the search space of all possible subgoal orderings. It handles the goal interactions by interleaving method.
- Non-linear planning may be an optimal solution with respect to plan length (depending on search strategy used).
- It takes larger search space, since all possible goal orderings are taken into consideration.
- Complex algorithm to understand.