Trabalho
de Inteligência1 Artificial

AAAI Press
Pontifícia Universidade Católica do Rio Grande do Sul
Ipiranga, 6681 - Parteno
Porto Alegre - RS, 90619-900

Introdução

Em Ciência da Computação, normalmente existem duas propriedades principais na criação e elaboração de algoritmos em IA:

- Fazer o algoritmo ter um tempo de execução sempre aceitável.
- Ser a solução ótima ou provavelmente boa para o problema em todos os casos.

No entanto, um algoritmo heurístico não cumpre uma dessas propriedades, podendo ser ou um algoritmo que encontra boas soluções a maioria das vezes, mas não tem garantias de que sempre encontrará ou um algoritmo que tem processamento rápido, mas não tem provas de que será rápido para todas as situações.

Esse trabalho tem como objetivo a implementação de 3 funções:

- Heuristic function.
- Validator.
- Heuristic search.

As funções heurística são funções que classificam as alternativas em algoritmos de pesquisa em cada etapa de ramificação, com base nas informações disponíveis para decidir qual ramificação seguir.

Sendo a função Heuristic uma rotina capaz de calcular e retornar o valor da heurística de um estado especifico, Validator uma função que verifica se o plano em questão atinge os objetivos e Heuristic Search a qual retorna o melhor plano para atingir o seu objetivo.

Heuristic

Como explicado anteriormente a função Heuristic(h) é uma função que verifica o custo da heurística de cada estado retornando assim um valor N entre 0 e ∞de acordo com a formulá:

$$h^{max}(s,\ g) = \max_{g_i \in g} \begin{cases} 0, & \text{if } g_i \in s \\ \min\{cost(a) + h^{max}(s,\ pre(a)) | a \in A \text{and} g_i \in \text{eff}(a)\} & \text{otherwise} \end{cases}$$

Dada a formula o desenvolvimento foi simples, basta iterar até que o objetivo seja alcançado ou até não se ter mais estados disponíveis, dentro dessa iteração deve-se observar o seguinte, sera adicionado no array Factlevel a Factlevel atual $\cup$ a (se alguma ação a $\subset$ factlevel[i]) recebendo assim os efeitos dos possíveis estados e retornando o numero te iterações feitas pelo algoritmo.

Algorithm 1 Pseudo codigo Heuristic H

```
1:  while Objetivo_Não_Alcançado do
2:      if Estado_Atual ⊂ for Ações.positive_preconditions
    then
3:          Visitados ← Estado_Atual.
4:      end if
5:      Estado_Atual ∪ for Ações.addEfects.
6:      i++
7:      if Proximo_Estado for vazio then
8:          Return ∞
9:      end if
10: end while
11: Return i
```

```python
def h(self, actions, state, goals):
    i = 0
    factlevel = [state]
    actionlevel = []
    positive_goals = goals[0]


    while not positive_goals.issubset(factlevel[i]):
        # actionlevel[i]
        actionlevel.append(
            [a for a in actions
             if a.positive_preconditions.issubset(factlevel[i])])

        # factlevel[i+1]
        factlevel.append(factlevel[i].union(
            [pre for a in actionlevel[i]
             for pre in a.add_effects]))
        if factlevel[i+1] == factlevel[i]:
            return float("inf")
        i += 1
    return i
```

Validate

A função Validate é basicamente uma função que recebe um planejamento e verificar se o mesmo leva para o objetivo.

Tendo em vista isso o desenvolvimento foi fácil, bastou percorrer as ações, encontrando as mesmas do planejamento e completando os itens faltantes no planejamento. Em seguida iteramos sobre planejamento com as funções applicable e aply fazendo assim nosso estado chegar ao estado

final do planejamento, Apos isso checamos se o estado atual esta contido na lista de objetivos.

Algorithm 2 Pseudo codigo Heuristic H

```
for a in actions do
    for p in plan do
        if p.parameters == a.parameters then
            p ← a.
        end if
    end for
end for
for p in plan do
    if applicable(p) then
        actual_state ← aply(p).
    end if
end for
for g in goals do
    if g(p) ⊂ actual_state then
        Return True
    end if Else then Return False
end for
```

```python
def validate(self, actions, initial_state, goals, plan):
    actual_state = initial_state
    for a in actions:
        for p in plan:
            if p.name == a.name:
                if p.parameters == a.parameters:
                    p.positive_preconditions = a.positive_preconditions
                    p.negative_preconditions = a.negative_preconditions
                    p.add_effects = a.add_effects
                    p.del_effects = a.del_effects

    for p in plan:
        applicable_action = applicable(
            actual_state, (p.positive_preconditions, p.negative_preconditions))
        if applicable_action == True:
            actual_state = apply(
                actual_state, (p.add_effects, p.del_effects))

    for g in goals:
        if g.issubset(actual_state):
            return True
        else:
            return False
```

Heuristic search

A* é uma travessia de gráfico e algoritmo de busca de caminho, que é frequentemente usado devido à sua completude, otimização e eficiência ideal. Uma grande desvantagem prática é sua complexidade de espaço $O(b^n)$, uma vez que armazena todos os nós gerados na memória. Assim sendo facilmente superado por algoritmos que podem pré-processar o gráfico para obter melhor desempenho.

Inicialmente Começamos o algoritmo criando uma fila de prioridade, a qual ira receber os nossos nós de acordo com o peso deles, em seguida percorremos o vetor de ação e salvamos as aplicáveis com o calculo da Heurística de cada uma e o custo até o nó em questão. Tendo assim na nossa fila de prioridade criada com cada ordenada respectivamente pelo seu peso.

```python
def solve(self, actions, state, goals):
    # YOUR CODE HERE
    #raise NotImplementedError()
    Tree = []
    plan = []
    less = Action('take', ['k1', 'cc', 'cb', 'p1', 'l1'], [], [], [], [])
    closed = []
    i = 0
    frontier = queue.PriorityQueue()
    frontier.put((0, state, 0))

    while not (frontier.empty() == True):
        n = frontier.get()
        plan.append(n[1])
        actual_state = n[1]
        closed.append(actual_state)
        if applicable(actual_state, goals):
            return plan

        for a in actions:
            applicable_action = applicable(
                actual_state, (a.positive_preconditions, a.negative_preconditions))
            if applicable_action == True:
                resulting_state = apply(
                    actual_state, (a.add_effects, a.del_effects))
                if resulting_state not in closed:
                    v = n[2] + h.h(actions, resulting_state, goals)
                    frontier.put((v, resulting_state, n[2]+1))

    return plan  # No plan was found
```