

Inteligência Artificial - Guião 2

Resolução Automática de Problemas

©Luís Seabra Lopes

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

I Objectivos

O presente guião centra-se no tema da resolução automática de problemas através de diferentes técnicas de pesquisa de soluções. Em particular, explora-se a utilização de técnicas de pesquisa em árvore.

Este guião é usado nas disciplinas de *Inteligência Artificial*, da *Licenciatura em Engenharia Informática*, e *Introdução à Inteligência Artificial*, do *Mestrado Integrado em Engenharia de Computadores e Telemática*.

O guião será realizado em 4 a 5 aulas práticas. ***Para um bom aproveitamento das aulas, os exercícios que estejam no âmbito temático de uma dada aula devem ser completados antes da aula seguinte.***

II Pesquisa em árvore

1 Apresentação do módulo inicial

Uma implementação completa do algoritmo básico de pesquisa em árvore é fornecida em anexo a este guião, no módulo `tree_search`.

O módulo contém as seguintes classes:

- Classe `SearchDomain()` – classe abstracta que formata a estrutura de um domínio de aplicação
- Classe `SearchProblem(domain, initial, goal)` – classe para especificação de problemas concretos a resolver
- Classe `SearchNode(state, parent)` – classe dos nós da árvore de pesquisa
- Classe `SearchTree(problem)` – classe das árvores de pesquisa, contendo métodos para a geração de uma árvore para um dado problema

Como se pode inferir da estrutura de dados adoptada, cada instância da classe `SearchTree` tem acesso aos seguintes atributos e métodos:

- `self.problem` - O problema a resolver (uma instância de `SearchProblem`)

- `self.problem.domain` - O domínio (uma instância de `SearchDomain`) em que se enquadra o problema
- `self.problem.domain.actions(state)` - Devolve uma lista com as acções aplicáveis em `state`
- `self.problem.domain.result(state, action)` - Devolve o resultado de `action` em `state`
- `self.problem.domain.cost(state, action)` - Devolve o custo de `action` em `state`
- `self.problem.domain.heuristic(state1, state2)` - Devolve uma estimativa do custo de ir de `state1` para `state2`
- `self.problem.domain.satisfies(state, goal)` - Verifica se um dado estado (`state`) satisfaz um dado objectivo (`goal`)
- `self.problem.initial` - O estado inicial
- `self.problem.goal` - O estado objectivo
- `self.problem.goal_test(state)` - Verifica se `state` é o objectivo
- `self.strategy` - A estratégia de pesquisa usada
- `self.open_nodes` - A fila dos nós abertos (folhas da árvore, a expandir), em que cada nó é uma instância de `SearchNode`
- `self.search()` - O método principal de pesquisa

O método principal da classe `SearchTree` implementa um procedimento genérico de pesquisa, baseado em fila de nós abertos:

```
def search(self):
    while self.open_nodes != []:
        node = self.open_nodes[0]
        if self.problem.goal_test(node.state):
            return self.get_path(node)
        self.open_nodes[0:1] = []
        lnewnodes = []
        for a in self.problem.domain.actions(node.state):
            newstate = self.problem.domain.result(node.state, a)
            lnewnodes += [SearchNode(newstate, node)]
        self.add_to_open(lnewnodes)
    return None
```

Em anexo, encontra ainda o módulo `ciudades`, com um domínio de aplicação concreto, que pode usar para testes.

2 Exercícios

Resolva em seguida as seguintes alíneas:

1. A implementação fornecida não previne ciclos. Isso leva a desperdício de espaço de memória na pesquisa em largura e a ciclos infinitos na pesquisa em profundidade. Assim, altere e/ou acrescente o código necessário por forma a prevenir a criação de ramos com ciclos. Teste o programa com a estratégia de *pesquisa em profundidade*.
2. Na estrutura de dados usada para representar os nós no módulo de pesquisa, acrescente um atributo para registar a profundidade do nó. Considera-se que a raiz da árvore de pesquisa está na profundidade 0.
3. Inclua na árvore de pesquisa uma @property `length` que devolva o comprimento da solução encontrada, dado pelo número de transições de estado desde o estado inicial até ao estado que satisfaz o objectivo.
4. Faça as alterações necessárias ao módulo `tree_search`, por forma a suportar a pesquisa em profundidade com limite.
5. Acrescente código ao método `search()` da classe `SearchTree` por forma a calcular o número total de nós terminais (folhas) e não terminais (nós já expandidos, mesmo que sem filhos) existentes na árvore após a conclusão da pesquisa. Essa informação deverá ficar armazenada em atributos do `self`.
6. O factor de ramificação média é dado pelo ratio entre o número de nós filhos (ou seja, todos os nós com excepção da raiz da árvore) e o número de nós pais (nós expandidos ou não terminais). Inclua na árvore de pesquisa uma @property `avg_branching` que devolva o respectivo factor de ramificação média.
7. Na classe `Cidades` do módulo `ciudades`, acrescente uma implementação do método `cost()`, o qual, dado um estado e uma acção, devolve o respectivo custo de executar essa acção nesse estado. Neste caso, para uma acção $(C1, C2)$, correspondente a uma deslocação da cidade $C1$ para a cidade $C2$, o custo deverá ser a distância entre essas cidades.
8. Na estrutura de dados usada para representar os nós no módulo `tree_search`, acrescente um atributo para o custo acumulado desde a raiz até cada nó. Modifique o algoritmo de pesquisa por forma a registar o custo acumulado em cada nó introduzido na árvore.
9. Acrescente na árvore de pesquisa (uma instância de `SearchTree`), uma @property `cost` que devolva o custo da solução encontrada, dado pelo custo acumulado do nó solução.
10. Faça as alterações necessárias ao código deste módulo por forma a suportar a *pesquisa de custo uniforme*.
11. Identifique uma heurística (função que estima o custo de chegar a um estado solução a partir de outro estado dado) adequada para a classe de domínios de pesquisa definida no módulo `ciudades` (classe `Cidades`) e implemente o método `heuristic()` dessa classe.
12. Na estrutura de dados usada para representar os nós no módulo de pesquisa, acrescente um atributo para registar o respectivo valor da heurística.
13. Faça as alterações necessárias na classe `SearchTree` por forma a suportar a pesquisa gulosa.

14. Faça as alterações necessárias na classe `SearchTree` por forma a suportar a pesquisa A*. Compare os resultados das diferentes técnicas de pesquisa.
15. Acrescente código ao método `search()` da classe `SearchTree` por forma a determinar o nó ou nós com maior custo acumulado. Esta informação deve ser armazenada na forma de uma lista num atributo do `self`.
16. Acrescente código ao método `search()` da classe `SearchTree` por forma a determinar a profundidade média dos respectivos nós. Esta informação deve ser armazenada num atributo do `self`.

III Pesquisa com operadores STRIPS

Em anexo a este guião, pode encontrar o módulo `strips`, com um novo domínio de pesquisa baseado em operadores STRIPS. Por sua vez, o módulo `blocksworld` implementa predicados e operadores STRIPS para o bem conhecido "mundo dos blocos".

Um operador, representado por uma classe derivada de `Operator`, especifica as pré-condições, efeitos negativos e efeitos positivos de uma classe de acções:

```
class Stack(Operator):
    # operador: empilhar
    args = ['X', 'Y']          # argumentos
    pc    = [Holds('X'), Free('Y')] # pre-condicoes
    neg   = [Holds('X'), Free('Y')] # efeitos negativos
    pos   = [On('X', 'Y'), HandFree(), Free('X')] # efeitos positivos
```

Podemos instanciar um operador como no seguinte exemplo:

```
>>> op = Stack.instantiate(['a', 'b'])
>>> op
Stack(a,b)
>>> print(op)
Stack([a,b], [Holds(a), Free(b)], [Holds(a), Free(b)],
[On(a,b), HandFree(), Free(a)])
```

Este método é usado para implementar o método `actions()` na classe de domínios de pesquisa STRIPS.

```
>>> initial_state
{ Floor(a), Floor(b), Floor(d), Holds(e), On(c,d), Free(a), Free(b), Free(c) }
>>> bwdomain = STRIPS()
>>> bwdomain.actions(initial_state)
[ Stack(e,a), Stack(e,b), Stack(e,c), Putdown(e) ]
```

1 Exercícios

Embora a maior parte do trabalho já esteja feita, falta finalizar alguns detalhes:

1. Implemente os métodos `result()` e `satisfies()` na classe STRIPS.
2. Na estrutura de dados usada para representar os nós no módulo `tree_search`, acrescente um atributo para registar a acção que conduziu a esse nó. Na árvore de pesquisa, acrescente

um atributo `plan` para registar a sequência de acções que constitui a solução encontrada. Modifique o algoritmo de pesquisa de forma a atribuir os valores correctos a estes atributos.

Pode agora experimentar, começando pelo exemplo criado no módulo `blocksworld`:¹

```
>>> goal_state
[ Floor(c), On(d,c), On(e,d), On(a,e), Floor(b) ]
>>> p = SearchProblem(bwdomain, initial_state, goal_state)
>>> t = SearchTree(p)
>>> t.search()
>>> t.plan
[ Stack(e,a), Unstack(c,d), Putdown(c), Pickup(d), Stack(d,c),
  Unstack(e,a), Stack(e,d), Pickup(a), Stack(a,e) ]
```

IV Pesquisa para problemas de atribuição com restrições

Em anexo a este guião, pode encontrar o módulo `constraintsearch`, similar ao desenvolvido nas aulas teóricas. O módulo disponibiliza uma classe `ConstraintSearch` que permite resolver problemas de atribuição com restrições. Por sua vez, e a título de exemplo, o módulo `rainhas` cria uma instância de `ConstraintSearch` para resolver o problema das 4 rainhas.

1 Exercícios

1. Resolva os exercícios IV.4 e IV.5 do guião teórico-prático usando o módulo `constraintsearch`.
2. O método `search()` da classe `ConstraintSearch` não faz propagação de restrições. Acrescente um método para fazer propagação de restrições e utilize-o no método `search()`.

¹Note que, dada a utilização de dicionários no módulo `strips`, o comportamento da pesquisa é não determinístico e o tempo que demora para o mesmo problema é variável.