

Guião 3

Resolução Automática de Problemas através de Pesquisa

Ano Lectivo de 2016/2017

©Luís Seabra Lopes

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Última actualização: 2014-12-16

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.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 classe `Cidades` do módulo `cidades`, 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.
3. Na estrutura de dados usada para representar os nós no módulo `tree_search`, acrescente um campo 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.
4. Faça as alterações necessárias ao código deste módulo por forma a suportar a *pesquisa de custo uniforme*.
5. Modifique o algoritmo de pesquisa de maneira a registar, na árvore de pesquisa (uma instância de `TreeSearch`), o custo total da solução encontrada, dado pela soma dos custos das sucessivas transições.
6. Na estrutura de dados usada para representar os nós no módulo de pesquisa, acrescente um campo para registar a profundidade do nó (sendo que a raiz da árvore de pesquisa está na profundidade 0).
7. Modifique o algoritmo de pesquisa de maneira a registar, na árvore de pesquisa (uma instância de `SearchTree`), o comprimento da solução encontrada, dado pelo número de transições de estado desde o estado inicial até ao estado que satisfaz o objectivo.
8. Faça as alterações necessárias ao módulo `tree_search`, por forma a suportar a pesquisa em profundidade com limite.
9. Na estrutura de dados usada para representar os nós no módulo de pesquisa, acrescente um campo para registar uma estimativa (heurística) do custo de chegar a uma solução a partir do estado desse nó.
10. Identifique uma heurística adequada para o classe de domínios de problemas definida no módulo `cidades` (classe `Cidades`) e implemente o método `heuristic()` dessa classe.
11. Faça as alterações necessárias ao módulo `tree_search` por forma a suportar a pesquisa gulosa.
12. Faça as alterações necessárias ao módulo `tree_search` por forma a suportar a pesquisa A^* .
13. Acrescente código ao método `search()` da classe `SearchTree` por forma a calcular o número total de nós terminais e não terminais existentes na árvore após a conclusão da pesquisa. Essa informação deverá ficar armazenada em campos do `self`. Considere que um nó expandido, mas sem filhos, conta como nó não terminal.

14. Acrescente código ao método `search()` da classe `SearchTree` por forma a calcular o respectivo factor de ramificação média, armazenando-o num campo do `self`. Relembra-se que 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 não terminais).
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 campo 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 campo do `self`.

III 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()`.