Trabalho prático individual nº 2
Representação do conhecimento e resolução automática de problemas

Inteligência Artificial / Introdução à Inteligência Artificial
Ano Lectivo de 2014/2015

4 de Dezembro de 2014

I Observações importantes

1. Este trabalho deverá ser entregue no prazo de 48 horas após a publicação deste enunciado. Os trabalhos poderão ser entregues para além das 48 horas, mas serão penalizados em 5% por cada hora adicional.
2. Submeta as classes e funções pedidas num único ficheiro com o nome "tpi2.py"e inclua o seu nome e número mecanográfico; não deve modificar nenhum dos módulos fornecidos em anexo a este enunciado.
3. Pode discutir o enunciado com colegas, mas não pode copiar programas, ou partes de programas, qualquer que seja a sua origem.
4. Se discutir o trabalho com colegas, inclua um comentário com o nome e número mecanográfico desses colegas. Se recorrer a outras fontes, identifique essas fontes também.
5. Todo o código submetido deverá ser original; embora confiando que a maioria dos alunos fará isso, serão usadas ferramentas de detecção de copianço. Alunos que participem em casos de copianço terão os seus trabalhos anulados.
6. Os programas serão avaliados tendo a conta: correcção e completude (70%); estilo (10%); e originalidade / evidência de trabalho independente (20%). A correcção e completude serão normalmente avaliadas através de teste automático. Se necessário, os módulos submetidos serão analisados pelos docentes para dar o devido crédito ao esforço feito.

II Exercícios

Em anexo a este enunciado, pode encontrar os módulos semantic_network e tree_search , os quais não deve alterar. Estes módulos são similares aos que usou nas aulas práticas, mas com pequenas alterações.

Em anexo a este enunciado, pode ainda encontrar o módulo automoveis, que representa algum conhecimento de suporte ao diagnostico de avarias em automóveis, na forma de uma rede semântica, e o módulo cidades, similar ao usado nas aulas. Deverá resolver os exercícios exclusivamente num novo módulo com o nome tpi2, deixando intactos os módulos dados.

No módulo tpi2, deve criar uma classe MySN, derivada de SemanticNetwork, e uma classe MyTree, derivada de SearchTree. Para efeitos de teste, os módulos automoveis e cidades importam o seu módulo tpi2, que por sua vez importa os módulos semantic_network e tree_search. No fim do trabalho, deverá submeter apenas o módulo tpi2.

1. Em aplicações que envolvem relações de dependência causal, como é o caso dos sistemas de diagnóstico, faz sentido dar um tratamento especial a essas dependências. Assim, foi acrescentada uma classe Depends, derivada de Relation, ao módulo semantic_network, que já conhece das aulas. É essa nova versão a que se disponibiliza em anexo.

   Para exemplo, e para suporte aos seus testes, disponibiliza-se também em anexo o módulo automoveis, no qual pode encontrar uma rede semântica para suporte a inferências sobre avarias em automóveis.

   Esta rede está parcialmente ilustrada na figura 1. Por simplicidade, omite-se na figura as associações debug_time, as quais poderá consultar no módulo.

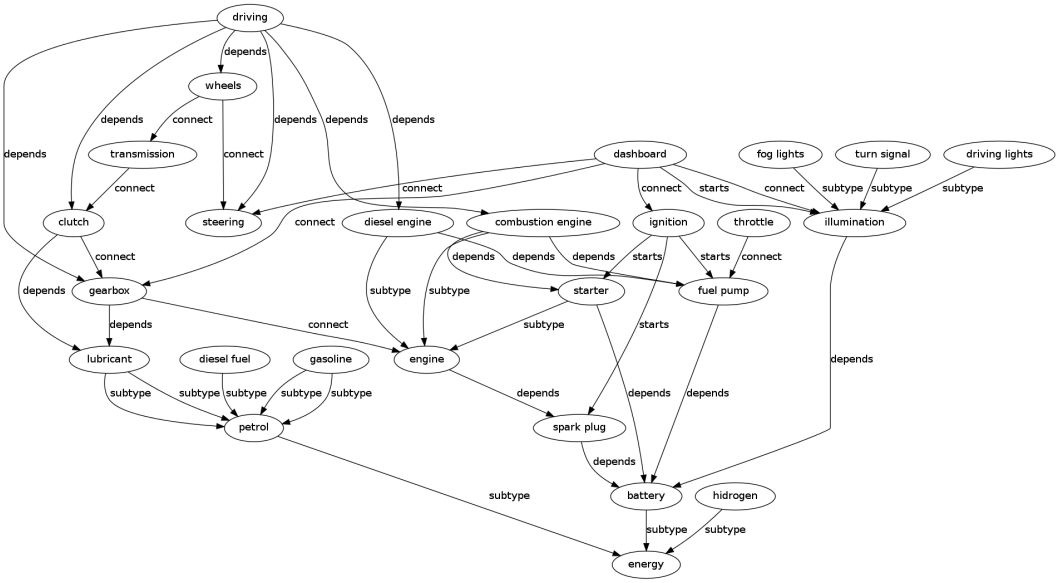

Figura 1: Relações principais existentes na rede semântica do módulo automoveis

a) Desenvolva um método query_dependents() na classe MySN que, dada uma entidade (tipicamente um tipo de componente automóvel), E, devolve a lista de todas as entidades cuja funcionamento depende do funcionamento de E. Essas dependências são dadas pela relação Depends e podem resultar de várias relações desse tipo encadeadas. Assim, engine depende de battery através de spark plug. Por outro lado, essas dependências podem ser herdadas. Por exemplo, driving lights é um subtipo de illumination e por isso depende de battery. Na lista que vai ser produzida pelo método pedido, não devem ser incluidas entidades que tenham subtipos. Ou seja, por exemplo, nos dependentes de battery, inclui-se driving lights, mas não illumination.

Exemplo:

```
>>> print z.query_dependents('battery')
['turn signal', 'fuel pump', 'driving', 'fog lights',
'driving lights', 'diesel engine', 'combustion engine',
'spark plug', 'starter']
```

b) Neste tipo de aplicações, é especialmente importante identificar as potenciais causas de uma avaria. Desenvolva um método query_causes() na classe MySN que, dada uma entidade (componente, funcionalidade), E, devolve a lista de todas as entidades cuja avaria ou mau funcionamento pode provocar avaria ou mau funcionamento de E. Enquanto a função anterior percorre a cadeia de dependências de causas para efeitos, a função query_causes() faz o inverso. Mais uma vez, não devem ser incluidos tipos que tenham subtipos registados na rede.

Exemplo:

```
>>> z.query_causes('driving')
['fuel pump', 'battery', 'clutch', 'diesel engine', 'combustion engine',
'spark plug', 'lubricant', 'wheels', 'gearbox', 'steering', 'starter']
```

c) Se o carro avariar, é preciso verificar as causas do problema. Cada componente leva um certo tempo a ser analisado, pelo que interessa ordenar as potenciais causas por ordem crescente do tempo de análise necessário. Desenvolva um método query_causes_sorted() na classe MySN que, dada uma entidade (componente, funcionalidade), E, devolve uma lista de tuplos (X, T), em que X é uma entidade cuja avaria ou mau funcionamento pode provocar avaria ou mau funcionamento de E (X é uma potencial causa do problema observado em E), e T é o tempo necessario para analisar X. A função retorna estes tuplos para todas as potenciais causas numa lista ordenada por ordem crescente de tempo. Para a identificação das causas pode usar a função da alínea anterior. Quanto ao tempo, como cada mecanico leva um tempo diferente, baseie-se na média dos tempos (ver informação no módulo automoveis).

Exemplo:

```
>>> z.query_causes_sorted('driving')
[('wheels', 1), ('battery', 10), ('lubricant', 16), ('fuel pump', 60),
('steering', 65), ('starter', 65), ('spark plug', 136), ('clutch', 190),
('gearbox', 203), ('combustion engine', 245), ('diesel engine', 270)]
```

2. Nas restantes alíneas, deverá fazer duas extensões ao módulo tree_search fornecido em anexo. Este módulo é similar ao das aulas, como pequenas alterações: i) O método search() da classe SearchTree, antes de retornar, guarda o resultado em self.result; ii) O método search() já faz prevenção de ciclos; iii) Estão incluidos alguns parâmetros adicionais no construtor da classe SearchNode, os quais são armazenados sem processamento, podendo dar-lhes o uso que entender. As extensões devem ser feitas na classe MyTree do módulo tpi2. Note também que o módulo cidades fornece uma definição da classe Cidades mais completa do que a inicialmente fornecida para as aulas práticas.

a) Como sabe, algumas técnicas de pesquisa não produzem soluções óptimas para a maior parte dos problemas. Uma forma de optimizar soluções produzidas por pesquisa em árvore é realizar sucessivas passagens pela solução, substituindo partes da solução por "atalhos".

Desenvolva um método optimize() na classe MyTree que, partindo de um caminho $[S_1, S_2, .., S_n]$ previamente guardado em self.result, tenta produzir um caminho melhor (ainda que nao necessariamente óptimo) entre os estados S_1 e S_n. Em cada iteração, a função percorre o caminho da esquerda para a direita procurando detectar estados S_i e S_j, em que $j - i > 1$, para os quais exista uma transição directa de S_i para S_j. Por exemplo, no caminho

```
['Porto','Aveiro','Figueira','Coimbra','CasteloBranco']
```

verifica-se que existe ligação directa entre o 2º estado (Aveiro) e o 4º estado (Coimbra). Assim, substituindo o sub-caminho

```
['Aveiro','Figueira','Coimbra']
```

pela ligação directa, obtém-se o caminho

```
['Porto','Aveiro','Coimbra','CasteloBranco']
```

Repete-se o procedimento que detecta sub-caminhos substituíveis por ligações directas até que nenhum sub-caminho seja detectado nessas condições. Finalmente, a função retorna o caminho optimizado.

A função deve também registar as optimizações feitas, na forma de uma lista de tuplos (S_i, S_j), em self.optimizations.

Exemplo:

```
>>> p = SearchProblem(cidades_portugal,'Lisboa','Faro')
>>> t = MyTree(p,'depth')
>>> t.search()
['Lisboa', 'Santarem', 'Evora', 'Beja', 'Faro']
>>> t.optimize()
['Lisboa', 'Beja', 'Faro']
>>> t.optimizations
[('Lisboa', 'Evora'), ('Lisboa', 'Beja')]
>>>
```

b) Implemente a estratégia de pesquisa astar_limited (A* com limite), que funciona normalmente como a pesquisa A*, mas não expande nós cuja função de avaliação ultrapassa um limite dado no construtor da classe SearchTree. Para este efeito, deverá implementar na classe MyTree um método search2(). Pode copiar o método original search() da classe SearchTree modificando-o para responder a esta alínea. Deverá igualmente implementar na classe MyTree um método astar_add_to_open(), cuja chamada está já incluída no módulo tree_search.

No fim do proceso de pesquisa, o método search2() deve atribuir valores aos seguintes atributos:

- self.solution_cost - Custo da solução encontrada, ou None, caso não tenha sido encontrada solução.
- self.solution_length - Comprimento (número de transições) da solução encontrada, ou None, caso não tenha sido encontrada solução.
- self.tree_size - Número total de nós da árvore de pesquisa gerada.

Exemplo:

```
>>> p = SearchProblem(cidades_portugal,'Lisboa','Faro')
>>> t = MyTree(p,'astar_limited',5000)
```

```
>>> t.search2()
['Lisboa', 'Setubal', 'Beja', 'Faro']
>>> t.solution_cost, t.solution_length, t.tree_size
309, 3, 13
>>>
>>> t = MyTree(p,'astar_limited',290)
>>> t.search2()
>>> t.solution_cost, t.solution_length, t.tree_size
None, None, 5
```