Trabalho prático individual nº 3

Inteligência Artificial / Introdução à Inteligência Artificial
Ano Lectivo de 2016/2017

8 de Novembro de 2016

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 "tpi3.py"e inclua o seu nome e número mecanográfico; não deve modificar nenhum dos módulos fornecidos em anexo a este enunciado. Casos de teste, instruções de impressão e código não relevante devem ser comentados ou removidos.

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 em conta: correcção e completude; estilo; e originalidade / evidência de trabalho independente. 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 `bayes_net`, `tree_search` e `constraintsearch`. Estes módulos são similares aos que usou nas aulas práticas, mas com alterações. Deverá resolver os exercícios exclusivamente no módulo `tpi3`, cujo esqueleto está já disponibilizado em anexo, deixando intactos os módulos dados. No módulo `tpi3_tests` existem alguns testes para as funcionalidades pedidas.

1. Com algumas adaptações, a estratégia de pesquisa em profundidade pode ter desempenho interessante em problemas complexos. A estratégia de pesquisa banbou segue uma ordem de expansão de nós parecida com a da pesquisa em profundidade, mas ignora alguns nós tendo em conta a função de avaliação habitual na pesquisa A*. As funcionalidades pedidas em seguida destinam-se a implementar a pesquisa banbou. Deverá implementar os métodos pedidos na classe MyTree.

   Nota: Foram incluídos alguns parâmetros extra no construtor da classe SearchNode do módulo tree_search, os quais são armazenados sem processamento. Poderá usar estes parâmetros para armazenar dados adicionais nos nós da árvore, caso precise.

   a) Crie um método de pesquisa search2() semelhante ao método original search() da classe SearchTree, e acrescente código de forma a atribuir valores aos seguintes campos do self:
      - self.solution_cost - Custo da solução encontrada.
      - self.tree_size - Número total de nós da árvore de pesquisa gerada.

      Exemplo:

```
>>> p = SearchProblem(cidades_portugal,'Braga','Faro')
>>> t = MyTree(p,'depth')
>>> t.search2()
['Braga', 'Porto', 'Agueda', 'Aveiro', 'Coimbra', 'Leiria',
  'Castelo Branco', 'Santarem', 'Lisboa', 'Evora', 'Beja', 'Faro']
>>> t.solution_cost
1079
>>> t.tree_size
26
```

   b) No que diz respeito à ordem de expansão dos nós, segue-se a política de expandir os nós mais recentes, tal como acontece na pesquisa em profundidade. No entanto, dentro dos nós resultantes de uma dada expansão (filhos do mesmo nó pai), a expansão será por ordem crescente da função de avaliação, isto é, custo acumulado + heurística. Implemente na classe MyTree o método banbou_add_to_open(), que é usado para acrescentar novos nós à fila. [Nota: Este método já está a ser chamado no método add_to_open da classe SearchTree.]

   c) Modifique agora o método search2() de forma a suportar a estratégia banbou de acordo com as seguintes regras:
      - A pesquisa vai encontrar várias soluções e só termina quando a fila estiver vazia;
      - Mantém-se registo da melhor solução encontrada até ao momento;
      - Quando um nó visitado tem função de avaliação superior ao custo da melhor solução encontrada até ao momento, esse nó é descartado;
      - Quando se encontra uma solução com custo inferior ao da melhor solução anteriormente encontrada, regista-se a nova solução

      Exemplo:

```
>>> s = MyTree(p,'banbou')
>>> s.search2()
['Braga', 'Guimaraes', 'Porto', 'Agueda', 'Coimbra',
  'Leiria', 'Santarem', 'Evora', 'Beja', 'Faro']
>>> s.solution_cost
693
>>> s.tree_size
149
```

2. As redes de Bayes têm diversas propriedades matemáticas. Uma dessas propriedades é a cobertura de Markov (*Markov blanket*).

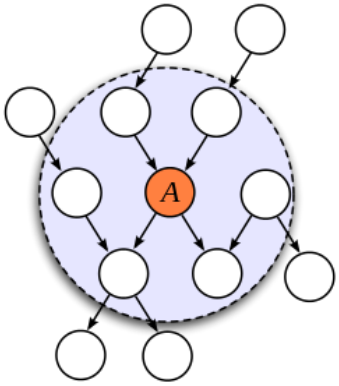

Figura 1: Cobertura de Markov da variável A.

A cobertura de Markov contém todas as variáveis que protegem uma dada variável do resto da rede. Isto significa que todo conhecimento sobre essa variável pode ser extraído do conhecimento da sua cobertura de Markov. A cobertura de Markov de uma variável A numa rede de Bayes é o conjunto dos nós ∂A composto pelos pais de A, os seus filhos, e os demais pais dos seus filhos.

Implemente na classe `MyBN` um método `markov_blanket(self, var)` que retorne uma lista com as variáveis que compõem a cobertura de Markov da variável `var`.

Para fins de teste, a rede representada na Figura 2 está incluída no módulo `tpi3_tests`.

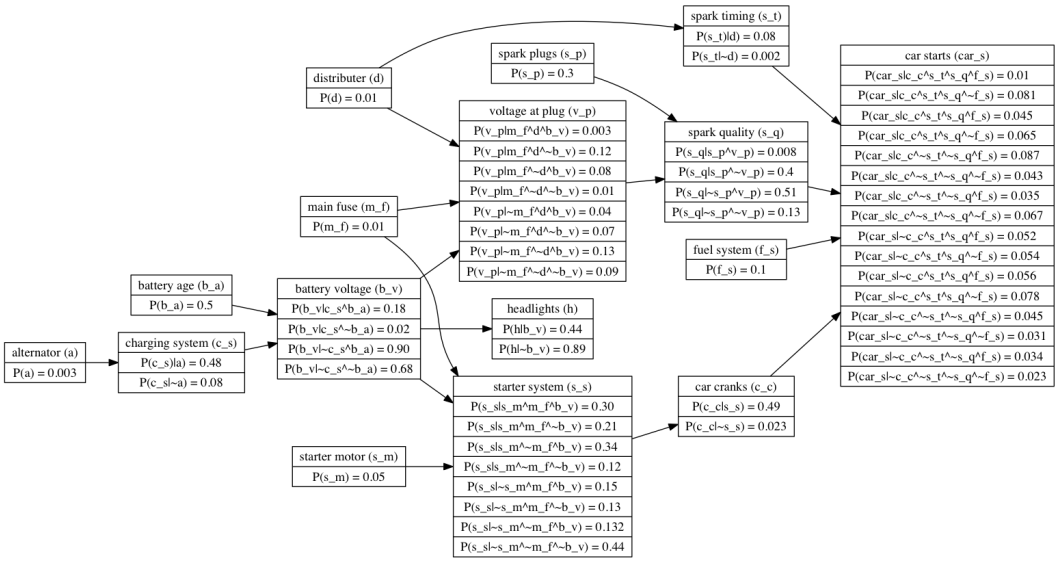

Figura 2: Rede de bayes para diagnóstico de um automóvel que não arranca.

Exemplos:

```
>>> mb = markovBlanket(s_t)
['d', 'car_s', 'f_s', 's_q', 'c_c']
>>> mb = markovBlanket(c_s)
['a', 'b_v', 'b_a']
```

3. A resolução do problema cripto-arimético `TWO + TWO = FOUR`, bastante trabalhada nas aulas teóricas, está agora completa no módulo `tpi3_tests`. Estando codificada a resolução do problema, queremos naturalmente saber quantas soluções existem, e quais são elas.

   O módulo `constraintsearch`, fornecido em anexo, tem já algumas melhorias, incluindo a poderosa propagação de restrições.

   Há no entanto, dois pequenos problemas para os quais se pede a sua ajuda:

   a) O método `search()` da classe `ConstraintSearch` só devolve uma solução. Desenvolva na classe derivada `MyCS`, um método similar `search_all()` modificado para devolver uma lista com todas as soluções.

   b) Devido ao número de variáveis originais (6) e ao número de restrições de ordem superior (4), este problema apresenta uma combinatória explosiva. A implementação naïve disponibilizada para o método `search()` piora ainda mais as coisas uma vez que, ao considerar todas as ordens possíveis de fixação das variáveis do problema, vai encontrar cada solução múltiplas vezes. Modifique o seu método `search_all` de forma a garantir que cada solução é encontrada apenas uma vez.

   Teste:

```
>>> cs = MyCS(...
>>> cs.search()
{ 'T': 1, 'O': 2, 'R': 4, 'FORTUW': (0, 2, 4, 1, 6, 3),
  'WX1UX2': (3, 0, 6, 0), 'ORX1': (2, 4, 0), 'X2': 0, 'U': 6,
  'TX2OF': (1, 0, 2, 0), 'W': 3, 'X1': 0, 'F': 0 }
>>> print(len(cs.search_all()))
19
```

III Esclarecimento de dúvidas

O acompanhamento do trabalho será feito via `http://detiuaveiro.slack.com`. O esclarecimento das principais dúvidas será também colocado aqui. Bom trabalho!

1. Professor, no módulo `tree_search` na class `SearchNode`, podemos dar outros nomes aos `arg3` e `arg4` de forma a ficar mais perceptível para mim?

   Resposta: Não pode ser, porque cada um de vocês vai escolher nomes diferentes, e depois o teste automático não funciona!

2. Na alinea c) do grupo 1, apenas modificamos a search2() para aquela nova estratégia, deixando o antigo funcionamento para as outras estratégias, correcto?

   Resposta: Sim, acrescentam nova funcionalidade, preservando a antiga!

3. No exercício 1.c), a solução é suposto ter `"Guimaraes"`?

Resposta: Com o domínio tal como está, é suposto dar o que está no enunciado. No entanto, há uma distancia no domínio (entre `Évora` e `Beja`) que é inferior à linha recta, fazendo com que a heuristica seja não admissível. Se corrigirem de 80 para 105, a heuristica fica admissível, e `Guimaraes` desaparece da solução que está no exemplo. Com essa alteração, passaria a funcionar assim:

```
>>> s = MyTree(p,'banbou')
>>> s.search2()
['Braga', 'Porto', 'Agueda', 'Coimbra', 'Leiria', 'Santarem',
  'Evora', 'Beja', 'Faro']
>>> s.solution_cost
706
>>> s.tree_size
183
```

Notem que, ser admissível ou não ser, para os exercícios em questão, não é relevante. Tem é que dar o resultado certo para o domínio que for.

4. Eu estou a tentar fazer o exercício 3.b), e não estou a entender a parte do enunciado em que se fala da ordem de fixação das variáveis.

   Resposta: A questão é que o algorimo de pesquisa original (dado) tanto fixa a variável A primeiro e depois a B como vice-versa. Explora os dois caminhos. Por isso, pode chegar por vários caminhos diferentes à mesma solução.

5. Poderia disponibilizar mais resultados do 1.c) ?

   Resposta: Aqui está:

```
q = SearchProblem(cidades_portugal,'Guarda','Lisboa')
>>> r = MyTree(q,'banbou')
>>> r.search2()
['Guarda', 'Covilha', 'Castelo Branco', 'Santarem', 'Lisboa']
>>> r.solution_cost
350
>>> r.tree_size
24
```

6. Entretanto, para termo de comparação, foi feita uma análise mais detalhada dos tempos da 3.b) com a solução do docente, em dois computadores diferentes, 200 testes em cada:

   - *mediana*=0.59 seg., *média*=1.13 seg., *max*=12 seg., *desviop*=2.00, número de vezes superior a 2.0 seg. = 13
   - *mediana*=1.24 seg., *média*=2.43 seg., *max*=16.55 seg., *desviop*=3.81, número de vezes superior a 2.0 seg. = 31

   Está no repositório um programa e um *script* para tirar os tempos. Os tempos piores estão relacionados com situações em que a primeira chave do dicionário `domains` é a variável `FORTUW`.