Trabalho prático individual nº 2

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

3 de Novembro de 2016

I Observações importantes

1. Este trabalho deverá ser entregue no prazo de 60 horas após a publicação deste enunciado. Os trabalhos poderão ser entregues para além das 60 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. 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 `semnet` e `tree_search`. Estes módulos são similares aos que usou nas aulas práticas, mas com pequenas alterações. Deverá resolver os exercícios exclusivamente num novo módulo com o nome `tpi2`, deixando intactos os módulos dados. No módulo `tpi2_tests` existem alguns testes para as funcionalidades pedidas.

1. O pequeno módulo de redes semânticas usado nas aulas práticas foi concebido para facilitar a entrada dos alunos no tema. Por essa razão, foram deixados de fora muitos aspectos que, num sistema mais profissional, teriam que ser considerados. O módulo `semnet` que se encontra em anexo, foi concebido com base no das aulas, mas tem algumas diferenças. Por exemplo, no módulo das aulas não é possível saber se uma associação está estabelecida entre dois objectos ou entre dois tipos. Também não existe maneira de definir se uma associação admite apenas um valor ou vários. O construtor da classe `Association` foi então modificado, passando a ter os seguintes argumentos:

   - `entity1` - Primeiro argumento da associação.
   - `name` - Nome da associação.
   - `entity2` - Segundo argumento (ou valor) da associação.
   - `cardin` - Cardinalidade da associação, que pode ser:
     - `None` - A usar em associações entre objectos, ou seja, `entity1` e `entity2` são necessariamente nomes de objectos.
     - `"one"` - A usar em associações entre tipos. Especifica que, quando a associação for usada entre objectos, ela admitirá apenas um valor. Exemplo: uma pessoa tem apenas um pai.
     - `"many"` - A usar em associações entre tipos. Especifica que cada objecto pode ter essa associação com vários. Exemplo: uma pessoa pode ter vários filhos.
   - `default` - A usar em associações entre tipos quando `cardin=="one"`. Especifica um valor por defeito para a associação. Exemplo: por defeito, a altura de um homem é 1.75.
   - `fluent` - A usar em associações entre tipos quando `cardin=="one"`. Se `fluent==True`, a aplicação da associação a objectos será verdadeira num intervalo de tempo.

   Alguns exemplos de associações:

   - `Association('mamifero','altura','number','one',1.2)` - Define a associação `altura` como sendo uma associação entre objectos do tipo `mamifero` e objectos do tipo `number`. A associação admite apenas um valor que, por defeito, é `1.2`.
   - `Association('socrates','altura',1.85)` - Utilização da associação anterior para registar a altura de Sócrates.
   - `Association('homem','progenitor','homem','many')` - Associação segundo a qual cada homem pode ser progenitor de vários homens.
   - `Association('sofronisco','progenitor','socrates')` - Utilização da associação anterior para registar o progrenitor de Sócrates.
   - `Association('agent','at','cell','one',(0,0),True)` - Associação segundo a qual os agentes podem estar em células. Na aplicação da associação, admite-se apenas um valor, que por defeito é `(0,0)`. O valor pode variar ao logo do tempo.
   - `Association('snake','at',(1,2))` - Utilização da associação anterior para indicar a posição actual de `snake`.

   Desenvolva então as seguintes funcionalidades:

   a) Um método `getObjects(user)` na classe `MySemNet` que devolve uma lista com todos os objectos cuja existência é possível inferir da rede.
   Exemplo:

```
>>> z.getObjects()
[1.2, 1.85, (0, 0), 'matematica', 'sofronisco', 80, 'aristoteles',
'socrates', 'platao', 'filosofia']
```

b) Como continuamos a ter vários interlocutores/utilizadores (fontes de conhecimento), que podem fornecer informações ou vocabulario divergentes, convém ter funcionalidades que ajudem a lidar com essa divergências. Desenvolva um método `getAssocTypes(assocname)` na classe `MySemNet` que, dado o nome de uma associação, devolve uma lista de tuplos `(t1,t2,freq)`, em que `t1` é o tipo da primeira entidade, `t2` é o tipo da segunda entidade e `freq` é a frequência relativa com que ocorre nas declarações disponíveis. (Nota: As associações entre objectos não são relevantes para este método.)

Exemplos:

```
>>> z.getAssocTypes('altura')
[('mamifero', 'number', 1.0)]

>>> z.getAssocTypes('pulsacao')
[('homem', 'numero', 0.5), ('homem', 'number', 0.5)]
```

c) Na mesma linha, desenvolva agora um método `getObjectTypes(obj)` na classe `MySemNet` que, dado um objecto, devolve uma lista de tuplos `(t,f)`, em que `t` é um dos tipos atribuídos ao objecto e `f` é a frequência relativa com que ocorre nas declarações disponíveis. Para este efeito, deverá procurar o tipo do objecto, não só em relações de `member`, mas também em associações cujos tipos dos argumentos sejam conhecidos. Por exemplo, `snake` é um `agent` porque tem uma associação `at`, e esta associação está declarada para o tipo `agent`.

Exemplos:

```
>>> z.getObjectTypes(1.2)
[('number', 1.0)]

>>> z.getObjectTypes('matematica')
[]

>>> z.getObjectTypes('socrates')
[('mamifero', 0.25), ('filosofo', 0.25), ('homem', 0.5)]
```

d) Uma relação (ou predicado) fluente é uma relação que será verdadeira apenas temporariamente. Isso poderá acontecer apenas com associações, uma vez que as relações de `member` e `subtype` são permanentes. Quando uma associação entre tipos é declarada com `fluent=True` (ver acima), então a utilização dessa associação entre objectos será acompanhada de informação temporal, na forma de um tuplo `(i1,i2)`, em que `i1` e `i2>i1` são instantes de tempo. No classe `Association` existe já um atributo `time` que por omissão terá o valor `None`. No entanto, caso a associação tenha sido declarada com fluente, esse atributo deverá ser preenchido com o intervalo de tempo em que a associação foi verdadeira. Desenvolva então um método `insert2(user,rel)` na classe `MySemNet` que, dado um utilizador e uma relação, regista a relação na rede. Caso a relação seja uma associação fluente, o intervalo temporal poderá ter que ser actualizado.

Exemplos:

```
>>> z.insert2('tracker',Association('agent','at','cell','one',(0,0),True))
>>> for i in range(10):
```

```
...       cell = (1,2) if i<7 else (2,3)
...       z.insert2('tracker',Association('snake','at',cell))
...
>>> z.query_local(rel='at')
[ decl(tracker, at(agent,cell[=(0, 0)])),
  decl(tracker, at(snake,(1, 2),(162, 174))),
  decl(tracker, at(snake,(2, 3),(176, 180))) ]
```

2. 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.solution, 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','Leiria']
   ```

   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','Leiria']
   ```

   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')]
   >>>
   ```

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!