Trabalho prático individual nº 2
Representação do conhecimento

Inteligência Artificial / Introdução à Inteligência Artificial
Ano Lectivo de 2019/2020

29-30 de Novembro de 2019

I Important remarks

1. This assignment is to be completed individually by each student, within 27 hours after this description is published. Assignments can be submitted beyond the 27 hours, with a penalty of 5% per each additional hour.
2. Submit the requested classes and/or functions in a single module named ”tpi2.py”and include your name and number; don’t submit other modules.
3. You can discuss the assignment with your colleagues, but you cannot copy the requested code from any source.
4. If you discuss the assignment with other colleagues, identify them in the modules with name and number. If you use any other sources, identify them as well.
5. All submitted code should be original; although trusting that most students will submit their own work, code plagiarism will be checked with appropriate tools. Students caught cheating will automatically fail the assignment.
6. Programs will be evaluated based on: correctness and completeness (70%); code style (10%); and originality / evidence of independent work (20%). Correctness and completeness will be primarily based on automated testing, but if your work is not fully functioning, it will be analyzed by the course staff to give due credit.

II Exercices

Attached to this description, you can find the semantic_network and bayes_net modules, which you are not allowed to change. These modules are similar to those used in practical classes, with small changes. You are requested to solve the proposed exercises exclusively in the module `tpi2`. Also attached to this description, you can find the module tpi2_tests , which contains several tests for the requested functions. If needed, you can add other test code in this module.

1. In applications involving causal dependency relations, as is the case of diagnosis applications, it makes sense to give special treatment to these dependencies. Thus, the Depends class, derived from Relation, was added to the semantic_network module. As an example, and to support testing, you can find in the module tpi2_tests a semantic network to support inferences about malfunctions in cars. This network is partially shown in Figure 1 (for simplicity, debug_time associations are omitted in the figure, but you can consult them in the module).

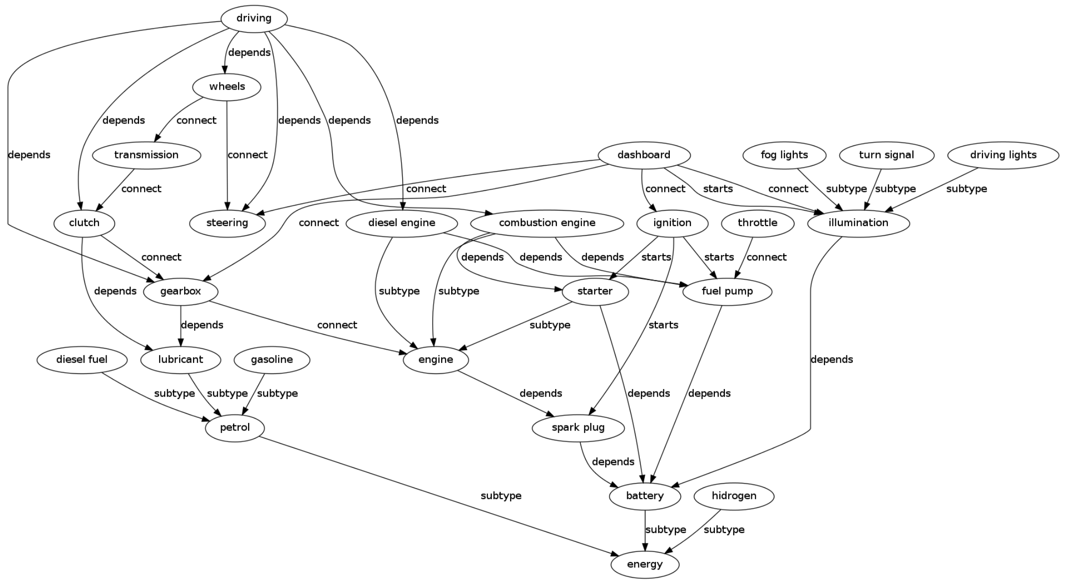


Figura 1: Main relations in the car domain.

a) Develop a method query_dependents() in the class MySN that, given an entity (typically a type of car component), E, returns the list of all entities whose operation depends on the correct operation of E. These dependencies are given by the Depends relation and can also result from chaining several such relations. This way, 'engine' depends on 'battery' through 'spark plug'. Moreover, these dependencies are inherited by subtypes. For example, 'driving lights' is a subtype of 'illumination', therefore it depends on 'battery'. The list that will be produced by the requested method should not include supertypes (i.e. entities with subtypes). For example, one of the dependents of battery is 'driving lights', because it is a subtype of 'illumination', so the latter is not included.

Examples:

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

b) In such applications, it is especially important to identify the potential causes of a malfunction. Develop a method query_causes() in the class MySN that, given an entity (component, functionality), E, returns the list of all entities whose failure or malfunction can cause failure or malfunction in E. The previous function traverses the chain of dependencies from causes to effects, whereas the function query_causes() does the opposite. Again, types having subtypes should not be included.

   Example:

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

c) If the car breaks down, the causes of the problem must be checked. Each component takes some time to be analyzed, so it is useful to sort potential causes by increasing order of the required analysis time. Develop a method query_causes_sorted() in the class MySN, which, given an entity (component, functionality), E, returns a list of tuples (X, T), where X is an entity whose failure or malfunction can cause damage or malfunction of E (X is a potential cause of the problem observed in E) and T is the time required to analyze X. The function returns these tuples for all potential causes as a list sorted in ascending order of time. For the identification of the causes, you can use the function from the previous exercice. As each technician takes a different amount of time, the sorting should be based on average time (see information on module tpi2_tests).

   Example:

```
>>> 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. Bayesian networks have several mathematical properties. One such property is the *Markov blanket*. It is defined as a set containing all the variables that protect a given variable from the rest of the network. All knowledge about this variable can be extracted from the knowledge of its Markov blanket. The Markov blanket of a variable A in a Bayesian network is the set of nodes ∂A consisting of the parents of A , their children, and the other parents of their children (see Fig. 2).

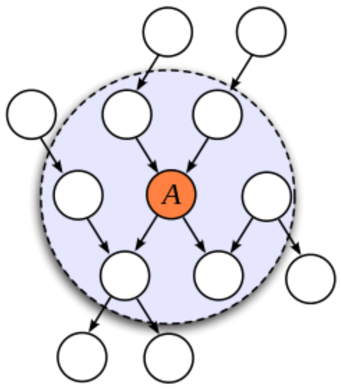
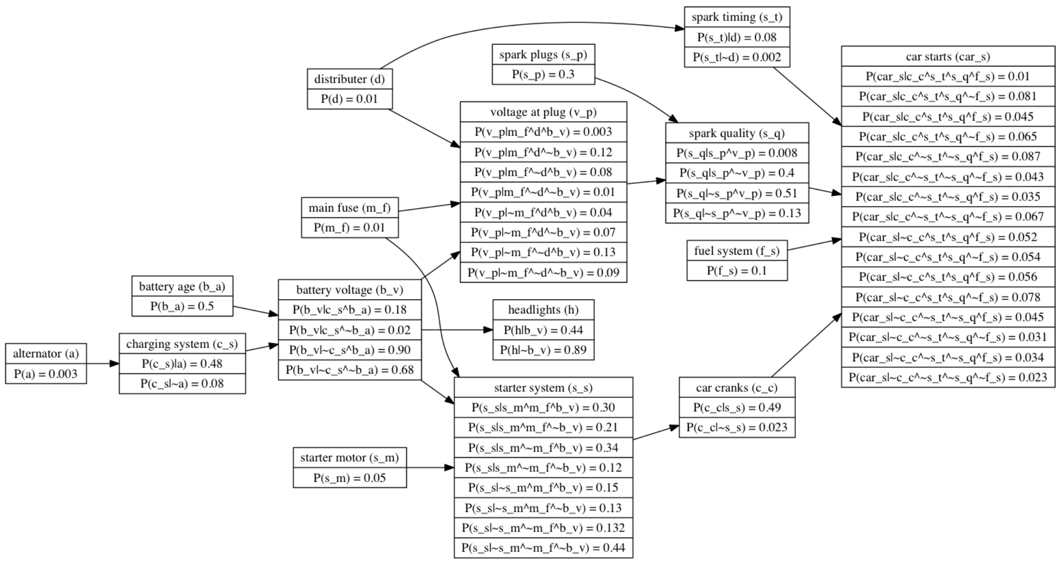


Figura 2: Markov blanket of a variable A.

Implement in the `MyBN` class a `markov_blanket()` method that, given a variable of the network, `var`, returns a list of the variables that make up the Markov blanket of that variable.

For testing purposes, the network represented in Figure 3 is included in the `tpi2_tests` module.



spark timing (s_t)
P(s_t)|d) = 0.08
P(s_t|~d) = 0.002
spark plugs (s_p)
P(s_p) = 0.3
distributer (d)
P(d) = 0.01
voltage at plug (v_p)
P(v_p|m_f^d^b_v) = 0.003
P(v_p|m_f^d^~b_v) = 0.12
P(v_p|m_f^~d^b_v) = 0.08
P(v_p|m_f^~d^~b_v) = 0.01
P(v_p|~m_f^d^b_v) = 0.04
P(v_p|~m_f^d^~b_v) = 0.07
P(v_p|~m_f^~d^b_v) = 0.13
P(v_p|~m_f^~d^~b_v) = 0.09
spark quality (s_q)
P(s_q|s_p^v_p) = 0.008
P(s_q|s_p^~v_p) = 0.4
P(s_q|~s_p^v_p) = 0.51
P(s_q|~s_p^~v_p) = 0.13
car starts (car_s)
P(car_s|c_c^s_t^s_q^f_s) = 0.01
P(car_s|c_c^s_t^s_q^~f_s) = 0.081
P(car_s|c_c^s_t^s_q^f_s) = 0.045
P(car_s|c_c^s_t^s_q^~f_s) = 0.065
P(car_s|c_c^~s_t^~s_q^f_s) = 0.087
P(car_s|c_c^~s_t^~s_q^~f_s) = 0.043
P(car_s|c_c^~s_t^~s_q^f_s) = 0.035
P(car_s|c_c^~s_t^~s_q^~f_s) = 0.067
P(car_s|~c_c^s_t^s_q^f_s) = 0.052
P(car_s|~c_c^s_t^s_q^~f_s) = 0.054
P(car_s|~c_c^s_t^s_q^f_s) = 0.056
P(car_s|~c_c^s_t^s_q^~f_s) = 0.078
P(car_s|~c_c^~s_t^~s_q^f_s) = 0.045
P(car_s|~c_c^~s_t^~s_q^~f_s) = 0.031
P(car_s|~c_c^~s_t^~s_q^f_s) = 0.034
P(car_s|~c_c^~s_t^~s_q^~f_s) = 0.023
main fuse (m_f)
P(m_f) = 0.01
fuel system (f_s)
P(f_s) = 0.1
battery age (b_a)
P(b_a) = 0.5
battery voltage (b_v)
P(b_v|c_s^b_a) = 0.18
P(b_v|c_s^~b_a) = 0.02
P(b_v|~c_s^b_a) = 0.90
P(b_v|~c_s^~b_a) = 0.68
headlights (h)
P(h|b_v) = 0.44
P(h|~b_v) = 0.89
alternator (a)
P(a) = 0.003
charging system (c_s)
P(c_s)|a) = 0.48
P(c_s|~a) = 0.08
starter system (s_s)
P(s_s|s_m^m_f^b_v) = 0.30
P(s_s|s_m^m_f^~b_v) = 0.21
P(s_s|s_m^~m_f^b_v) = 0.34
P(s_s|s_m^~m_f^~b_v) = 0.12
P(s_s|~s_m^m_f^b_v) = 0.15
P(s_s|~s_m^m_f^~b_v) = 0.13
P(s_s|~s_m^~m_f^b_v) = 0.132
P(s_s|~s_m^~m_f^~b_v) = 0.44
car cranks (c_c)
P(c_c|s_s) = 0.49
P(c_c|~s_s) = 0.023
starter motor (s_m)
P(s_m) = 0.05

Figura 3: Bayesian network for a car that does not start

Examples:

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