Document version: 0.1
Last modification: 28/05/12
Author(s): Filip Jurcicek

UFAL Dialogue state update

The purpose of this document is to describe the principles and implementation of dialogue state update in spoken dialogue systems developed at UFAL, MFF UK, Czech Republic. As a prerequisite, the reader should be familiar with the definition of the UFAL dialogue act scheme as described in ufal-dialogue-acts.odt.

Table of Contents

Definition of a dialogue state

In a spoken dialogue system, the purpose of a dialogue state is to track the progress in a dialogue. Based on the dialogue state then the dialogue policy suggests how to continue in the dialogue

In its simplest form, the dialogue state can be composed only of variables that represents the goal of the user. A good example of this approach is a tourist information domain where users seeks information about restaurants, bars, and hotels, users can constraint their search by area, price range, stars, and the dialogue system can provide information about the address, the postcode, or the phone number for the selected venue. In this domain, the goal can be composed of the following variables:

venue_type	– defines what type of a venue the user is looking for
area	– desired location of the venue
price_range	– price range of the venue
near	– whether the requested venue should be near another venue
food_type	– type of food served at the venue if the venue type is a restaurant
stars	– number of stars of the venue if the venue type is a hotel

Sometimes these variables are called slots or concepts and their values are called attributes. For convenience, the slots representing the goal are called goal slots in this report. In addition to the goal, the dialogue state can store information about what the user requested or wants to confirm, what the system already informed about or what the system confirmed. This information is stored in additional variables / slots.

The dialogue state update

The UFAL dialogue systems uses the concept of Information State Update, where the state is updated whenever new information is available, either from a user or the system itself or any other partner in the communication. Since the update is defined on the dialogue state, the process of updating the dialogue states is called the dialogue state update. The dialogue state update is best depicted using an influence diagram.

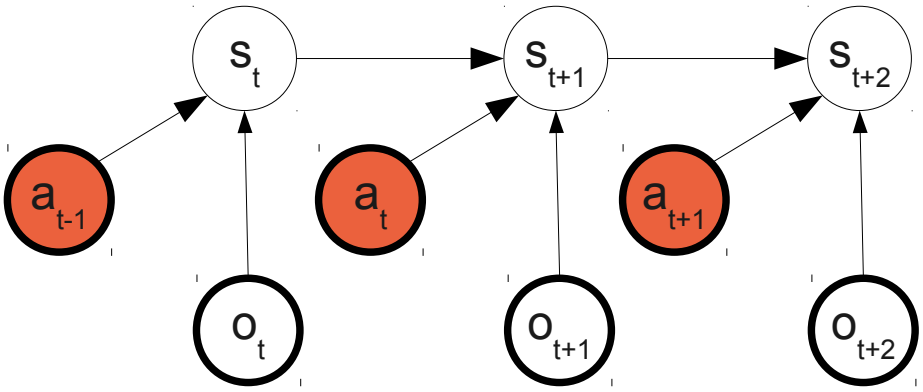


Figure 1: Dialogue state update influence diagram

Shown at Figure 1, the dialogues state $\mathbf{s_{t+1}}$ depends on the previous state $\mathbf{s_t}$, the previous system dialogue act $\mathbf{a_t}$, and the current user dialogue act $\mathbf{o_{t+1}}$. This can be formally written as

$$s_{t+1} \leftarrow s_t + a_t + o_{t+1} \quad .$$

It is assumed that the update is deterministic. Therefore, it must result in one specific dialogue state given the previous state, system dialogue act and the current user dialogue act. Following the example provided in the previous section, an example of the dialogue state update in a tourist information domain can be performed as follows:

Turn 1

Dialogue state:

```
venue_type     = None
area           = None
price_range    = None
near           = None
food_type      = None
stars          = None
```

System says: How may I help you?
System dialogue act: `hello()`

User says: Hi I am looking for a **restaurant**.
User dialogue act: `hello()&inform(venue_type='restaurant')`

Turn 2

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = None
near           = None
food_type      = None
stars          = None
```

System says: What type of food are you looking for?
System dialogue act: `request(food_type)`

User says: I want a **Chinese** restaurant in a **cheap** price range.
User dialogue act:
```
inform(venue_type='restaurant')&
inform(food_type='Chinese')&
inform(price_range='cheap')
```

Turn 3

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None
```

Note that the value “None” denotes that user did not say anything about the relevant slot and it is a default value for each slot.

Note that the values allowed in the slots in the previous example are very simple. The slots can store only one value out of many. More complex values, such as sets or logic expressions, are not currently supported. If sets are really needed then one can defined slot values as elements of a power set defined for the original slot values.

Relatively simple requirement that the system should remember what user does not want turns out to be complex to implement. The problem is that remembering only one value that user does not want is not enough. For example, storing information from a user that does not want Indian food does not

discriminate between other options such as Chines, English, Italian, French, etc. To properly handle negations, one would have to define set of binary slots which would indicate whether a slot value is wanted by the user or not. Having sometimes thousands of slot values in one slot, this can lead to overly complex dialogue state. To approximate this, the system could try to remember last N values the user does not want. At this moment, the UFAL dialogue managers do not support this type of negations.

To implement the negations at least partially, the UFAL dialogue dialogue systems support the deny dialogue act. Every time the user says that does not want something, e.g. “I do not want Chinese food.” - deny(food_type='Chinese') the slot value of the slot food_type is set None. This approach is shown in the next example:

Turn 3

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None
```

System says: Ok. You want a Chinese restaurant in a cheap price range. Is that right?
System dialogue act:
```
ack()&inform(venue_type='restaurant')&
inform(food_type='Chinese')&
inform(price_range='cheap')
```

User says: No. I do not want Chinese restaurant.
User dialogue act:
```
negate()&deny(food_type='Chinese')
```

Turn 4

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = None
stars          = None
```

If the user said in the 3rd turn, for example, “No. I do not want Chinese restaurant. I want Indian.” then the update would look as follows:

Turn 3

User says: No. I do not want Chinese restaurant. I want Indian.
User dialogue act: `negate()&deny(food_type='Chinese')& inform(food_type='Indian')`

Turn 4

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = None
stars          = None
```

Simply, the inform act overwrites the deny act since the inform act is more informative.

In general, dealing with negation tends to be very complex. For example, should be the utterance “I do not want Chinese restaurant.” interpreted as deny(food_type='Chinese') or deny(food_type='Chinese')& deny(venue_type='restaurant') ?

Recording requests, confirmations and selects by users

So far, only the user inform dialogue act was considered when performing dialogue state update. Other dialogue acts which must be considered are request, confirm, and select.

Requests

Typically a user requests a value of a particular slot if the user is offered a venue and the system did not informed about that slot. Note the distinction between a property of a venue being offered and the constrain the user specified. They do not have to be necessary always the same.

Therefore, in UFAL dialogue systems user can request some information only when some venue was already offered.

The fact that user requests details about some properties of a venue is a new information and should be stored in the dialogue state. Once the dialogue system informs about the requested slot, it should be recorded that the system already provided the requested information. Therefore for storing information about requested slots, specialised slots have to be defined called request history slots. A request history slot has three values: 1) None, 2) user-requested, and 3) system-informed. Just for convenience, all request history slots starts with the prefix 'rh_' and are flowed by the name of the relevant slot.

Assuming that a user can request information about all slot from the previous example, the dialogue state update can continue as follows:

Turn 3

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None

rh_venue_type  = None
rh_area        = None
rh_price_range = None
rh_near        = None
rh_food_type   = None
rh_stars       = None
```

System says: Char Su is a nice cheap Chinese restaurant.
System dialogue act: `inform(name='Char Su')&`
`inform(venue_type='restaurant')&`
`inform(food_type='Chinese')&`
`inform(price_range='cheap')`

User says: Where is it located?
User dialogue act: `request(area)`

Turn 5

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None

rh_venue_type  = None
rh_area        = user-requested
rh_price_range = None
rh_near        = None
rh_food_type   = None
rh_stars       = None
```

System says: Char Su is in the city centre.
System dialogue act: `inform(name='Char Su')&inform(area='centre')`

Turn 6

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None

rh_venue_type  = None
rh_area        = system-informed
rh_price_range = None
rh_near        = None
rh_food_type   = None
rh_stars       = None
```

Note that the system does not have inform immediately or about all requested slots, therefore the request information is stored and dealt with later if necessary. Once the system informs about the requested slot, it is recorded by storing storing system-informed in the request history slot. This ensures that the system will not inform about it multiple times unless it is asked again.

The setting the request slot to system-informed can be also interpreted as receiving a system dialogue act inform(rh_area='system-informed') and storing the rh_area value in the dialogue state. For example, the system response in previous example in the 6th turn would then be.

System says: Char Su is in the city centre.
System dialogue act: `inform(name='Char Su')&inform(area='centre')&inform(rh_area='system-informed')`

Confirms

When recording information about what user is trying to confirm more information must be stored. For example, if a user tries to confirm whether the venue is expensive then it does not necessary mean that user is looking for an expensive restaurant. The user just wants to make sure that the offered venue is not in the expensive price range and still keep the goal the same. Therefore, a dialogue state has to store not only what slots user wants to confirm but also what value is confirming in a new set of slots. These slots are called confirm history slots. This practically leads to duplication of all domains slots.

The confirm history slots include all values as the original slots plus the system-informed slot value to record that the system responded to the user confirmation dialogue act.

Assuming that a user can confirm information about all slot from the previous example, the dialogue state update can continue as follows:

Turn 5

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None
```

rh_venue_type = None	ch_venue_type = None
rh_area = user-requested	ch_area = None
rh_price_range = None	ch_price_range = None
rh_near = None	ch_near = None
rh_food_type = None	ch_food_type = None
rh_stars = None	ch_stars = None

System says: Char Su is in the city centre.
System dialogue act: `inform(name='Char Su')&inform(area='centre')`

User says: Sorry! Can you tell me again is it expensive?
User dialogue act: `apology()&confirm(price_range=expensive)`

Turn 6

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None
```

rh_venue_type = None	ch_venue_type = None
rh_area = system-informed	ch_area = None
rh_price_range = None	ch_price_range = expensive
rh_near = None	ch_near = None
rh_food_type = None	ch_food_type = None
rh_stars = None	ch_stars = None

System says: No, Char Su is in the cheap price range.
System dialogue act: `negate()&inform(name='Char Su')&`

```
inform(price_range='cheap')
```

An alternative answer of the system output can be:

System says: No, Char Su is not in the expensive price range.
System dialogue act: `negate()&inform(name='Char Su')&deny(price_range='expensive')`

Turn 7

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None
```

rh_venue_type	= None	ch_venue_type	= None
rh_area	= system-informed	ch_area	= None
rh_price_range	= None	ch_price_range	= system-informed
rh_near	= None	ch_near	= None
rh_food_type	= None	ch_food_type	= None
rh_stars	= None	ch_stars	= None

Note that for the system to be able to answer that the restaurant is not in the expensive price range, the system has to know that user is asking whether the restaurant is in the expensive price range.

Again, the system does not have inform immediately or about all confirmed slots, therefore the confirm information is stored and dealt with later if necessary. Once the system informs about the confirmed slot, it is recorded to prevent repetition.

Since the confirm history slots have the same number of values as the goal slots, they can significantly increase the complexity of the update. Assuming that user would be satisfied if the system informs about the true value of the slot then a simple approximation can be used - the confirms will be dealt with as if they were requests. In this case, any time a user tries to confirm some value in a slot, the system will inform always about the true value. For example, such approximation would be result in the following responses in the 5th turn:

Turn 4

User says: Sorry! Can you tell me again is it expensive?
User dialogue act: `apology()&confirm(price_range=expensive)`

Turn 5

System says: Char Su is in the cheap price range.
System dialogue act: `inform(name='Char Su')&inform(price_range='cheap')`

This answer is an approximation since the system does no explicitly that it is not in the expensive price range. From experience, some human users are confused by this behaviour and the full implementations of confirms is preferred.

Selects

At this moment, the select dialogue act is not supported in the UFAL dialogue systems. However, the implementation would follow the approach outlined in two previous sections. For each goal slot, the system have to include into the dialogue state two additional slots. The first for one value of the select dialogue act and the second for the other. These slots are called select history nodes 1 and 2 with prefixes 'sh1_' and 'sh2'.

The select history slots include all values as the original slots plus the system-informed slot value to record that the system responded to the user select dialogue act.

Assuming that a user can use the select dialogue act with all slot from the previous example, the dialogue state update can continue as follows:

Turn 6

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None
```

rh_venue_type	= None	ch_venue_type	= None
rh_area	= system-informed	ch_area	= None
rh_price_range	= None	ch_price_range	= expensive
rh_near	= None	ch_near	= None
rh_food_type	= None	ch_food_type	= None
rh_stars	= None	ch_stars	= None

sh1_venue_type	= None	sh2_venue_type	= None
sh1_area	= None	sh2_area	= None
sh1_price_range	= None	sh2_price_range	= None
sh1_near	= None	sh2_near	= None
sh1_food_type	= None	sh2_food_type	= None
sh1_stars	= None	sh2_stars	= None

System says: No, Char Su is not in the expensive price range.
System dialogue act: `negate()&inform(name='Char Su')&deny(price_range='expensive')`

User says: Also, did you say that the restaurant serves Chinese or Indian?
User dialogue act: `apology()&confirm(price_range=expensive)`

Turn 7

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None
```

rh_venue_type	= None	ch_venue_type	= None
rh_area	= system-informed	ch_area	= None
rh_price_range	= None	ch_price_range	= system-informed
rh_near	= None	ch_near	= None
rh_food_type	= None	ch_food_type	= None
rh_stars	= None	ch_stars	= None

sh1_venue_type	= None	sh2_venue_type	= None
sh1_area	= None	sh2_area	= None
sh1_price_range	= None	sh2_price_range	= None
sh1_near	= None	sh2_near	= None
sh1_food_type	= Chinese	sh2_food_type	= Indian
sh1_stars	= None	sh2_stars	= None

System says: Char Su serves Chinese food. It does not serve Indian.
System dialogue act: `inform(name='Char Su')&inform(food_type='Indian')&deny(food_type='Indian')`

Turn 8

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
```

```
food_type      = Chinese
stars          = None
```

rh_venue_type	= None	ch_venue_type	= None
rh_area	= system-informed	ch_area	= None
rh_price_range	= None	ch_price_range	= system-informed
rh_near	= None	ch_near	= None
rh_food_type	= None	ch_food_type	= None
rh_stars	= None	ch_stars	= None

sh1_venue_type	= None	sh2_venue_type	= None
sh1_area	= None	sh2_area	= None
sh1_price_range	= None	sh2_price_range	= None
sh1_near	= None	sh2_near	= None
sh1_food_type	= system-informed	sh2_food_type	= system-informed
sh1_stars	= None	sh2_stars	= None

The group of request history slots, confirm history slots, and select history slots will be refereed to as history slots.

Context

In the previous part of report, the description of the goal and history slots, and their update was provided. When considering the update the following dialogue acts were used: inform, deny, request, confirm, and select.

Still the dialogue system has to know how handle the rest of the dialogue acts such as ack, affirm, apology, bye, hangup, hello, help, negate, null, repeat, reqalts, reqmore, restart, or thankyou.

These dialogue acts can be divided into two groups: one need context to be properly interpreted, the other not.

- The context dependent dialogue acts are affirm and negate.
- The context independent dialogue acts are ack, apology, bye, hangup, hello, help, null, repeat, reqalts, reqmore, restart, thankyou.

To interpret the affirm and negate dialogue acts, one has to know the last system act. For example, user's “Yes” interpreted as affirm() does not tell what the user agrees with. Therefore the UFAL dialogue system implements the following transformation rules:

```
if the system act is confirm(x=y) and the user act is affirm() then
     affirm() → inform(x=y)
if the system act is confirm(x=y) and the user act is negate() then
     negate() → deny(x=y)
```

These transformation rules are demonstrated on the next example:

Turn 9

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None
```

System says: Do you want the restaurant to be in the city centre?
System dialogue act: `inform(venue_type='restaurant')&confirm(area='centre')`

User says: No.
User dialogue act: `negate()`
After transform: `deny(area='centre')`

An alternative user response could look like:

User says: Yes.
User dialogue act: `affirm()`
After transform: `inform(area='centre')`

Another alternative user response could look like:

User says: No. I want it be in the riverside.
User dialogue act: `negate()`
After transform: `deny(area='centre')&inform(area='riverside')`

The context has to be also used, for example, when user says “I do not care” - inform(='dontcare'). In this case we do not know what the user does not care about. Therefore the UFAL dialogue system implements the following transformation rule:

```
if the system act is request(x) and the user act is
inform(='dontcare') then
     inform(='dontcare') → inform(x='dontcare')
```

Sometimes it is useful to implement some domain context dependent transformation rules. For example, the tourist information domain could include the additional goal slots such as:

```
has_internet     = ['yes', 'no']
```

```
children_allowed = ['yes', 'no']
```

to make sure that the requested venue has WIFI internet or not, or that there are children allowed at the venue or not. When the system requests for such constraints the questions are typically formulated as “Are you looking for a venue with free internet?” or “Are you coming with children?” Then the typical answer is “Yes” – affirm() or “No” – negate() . Since the user is practically using affirm or negate dialogue acts instead of inform act for these slots, it has to handled using transformation rules.

```
if the system act is request(has_internet) and the user act is affirm() then
    affirm() → inform(has_internet='yes')
if the system act is request(has_internet) and the user act is negate() then
    affirm() → inform(has_internet='no')
if the system act is request(children_allowed) and the user act is affirm() then
    affirm() → inform(children_allowed='yes')
if the system act is request(children_allowed) and the user act is negate() then
    affirm() → inform(children_allowed='no')
```

These rules must be generated for all binary goal slots with slot values yes or no. The domain dependent transformation rules have a lot of use and can alleviate some of the problem one has when designing a new dialogue system.

The last dialogue act slot

For the store and interpret context independent dialogue acts, the UFAL dialogue systems include additional slot referred to as the last dialogue act (LDA) slot. Every time user on of the following dialogue acts ack, apology, bye, hangup, hello, help, null, repeat, reqalts, reqmore, restart, thankyou, it is recorded in LDAS. This is shown in the next example:

Turn 9

Updated dialogue state:

```
venue_type     = restaurant
area           = None
price_range    = cheap
near           = None
food_type      = Chinese
stars          = None
```

```
lda             = None
```

System says: Do you want the restaurant to be in the city centre?
System dialogue act: `inform(venue_type='restaurant')&confirm(area='centre')`

User says: Can you repeat that?
User dialogue act: `repeat()`

Turn 10

Updated dialogue state:

```
venue_type      = restaurant
area            = None
price_range     = cheap
near            = None
food_type       = Chinese
stars           = None

lda             = repeat
```

System says: Do you want the restaurant to be in the city centre?
System dialogue act: `inform(venue_type='restaurant')&confirm(area='centre')`

User says: No.
User dialogue act: `negate()`

Turn 9

Updated dialogue state:

```
venue_type      = restaurant
area            = None
price_range     = cheap
near            = None
food_type       = Chinese
stars           = None

lda             = system-informed
```

Once the dialogue system handled the last dialogue act, it sets the slot to system-informed to prevent any repetition such providing help twice or repeating the last system dialogue act multiple times.

Ontology

The structure of a dialogue state can be defined by some form of ontology. In its simplest form, the ontology can be defined as list of slots and the possible values for the slots. In general, the ontology can define what the user or the system can talk or ask about.

The ontology for a specific domain in a UFAL dialogue system defines:

- all slots in the domain
- all values for the slots in the domain
- the dependencies between the slots
- what slots can a user inform about
- what slots can a user request
- what slots can a user confirm
- on what slots can a user use the select dialogue act
- what slots can a system inform about
- what slots can a system request
- what slots can a system confirm
- on what slots can a system use the select dialogue act

In the UFAL dialogue systems, the ontology is defined as Python script and using Python syntax.

1. Slots and their values are defined as a dictionary. E.g.

```
slots = {'venue_type':  ['restaurant', 'bar', 'hotel'],
         'area':        ['adenbrooks', 'arbury', ],
         'price_range': ['free', 'cheap', 'moderate', 'expensive'],
         'near':        ['the bakers', 'trinity college'],
         'food_type':   ['chinese', 'indian', 'english'],
         'stars':       ['1', '2', '3', '4', '5']
        }
```

2. Slot attributes stores what operation a user or a system can do the slots: e.g. inform, request, confirm, or select.

```
slot_attributes =
```

```
{'venue_type': ['user_informs',
                'user_requests',
                'user_confirms',
                'user_selects',
                'system_informs',
                'system_requests',
                'system_confirms',
                'system_selects',
               ],
 ...
 'near':       ['user_informs',
                'user_requests',
                'user_confirms',
                'system_informs'
                'system_confirms',
                'system_selects'],
 ...
 }
```

In the example above, the user and the system can freely talk and ask about the slot venue_type. It is expected that you user knows what type of venue is looking for and that system can freely asks abut it.

In the case of the near slot, the situation is more complex. It is assumed that a user would not be happy if the systems explicitly asked for the nearness of the venue to some specific place. Therefore, the system is not allowed to request this slot (note that the 'system_requests' is missing). Nevertheless, the system can inform about this slot if it decides that it is appropriate or it can confirm or ask the user to select values in the slot, for example once a user started to talk about the nearness of the requested venue.

From the perspective of a dialogue system, the slot attributes define what slots will be explicitly present in the dialogue state. For example, 1) the slot with the user_informs attribute then it must be present among the goal slots, the slot with the user_requests attribute then it must be present among the request history slots, 3) the slot with the user_confirms attribute then it must be present among the confirm slots, 4) the slot with the user_selects attribute then it must be present among the selects slots.

More complex ontologies can define dependencies between the slots themselves. Some times, not all variable are applicable at all states. For example, the food_type variable is applicable only when the requested venue is a restaurant. Similarly, the variable stars is applicable only when the requested venue is a hotel. Therefore each slot variable can be stores a generic value 'N/A' (not applicable) when

the context (the rest of the state) suggest that the content of the slot is irrelevant.

When designing dependencies between slots, it is difficult to do it in a general way that would fit all possible ways of implementing dialogue state update and its approximations. There for the UFAL ontology defines only a very simple dependencies.

1. It is assumed each slot depend on the value in the slot in the previous turn. For example, if user did not say anything about the slot then the slot value are copied from previous turn to the next. This dependency is assumed to be implemented in the dialogue state update algorithm and it does not have any supporting syntax in the ontology.

2. The UFAL ontology supports applicability dependencies. For example, the food_type slot in the tourist information domain is applicable only when the requested venue is a restaurant.

The applicability rules are defined as list of tuples where each tuple represent one applicability rule. The applicability tuple has four elements: 1) parent slot name, 2) value in the parent slot, 3) child slot name and 4) applicability. The applicability element has two values i) 'A' – applicable and ii) 'N/A' - not applicable. For example, the tuple ('venue_type','restaurant','food_type', 'A') defines that slot food_type is applicable when the value of the slot venue_type is equal to restaurant. Another example, the tuple ('venue_type','restaurant','stars', 'N/A') defines that slot food_type is applicable when the value of the slot venue_type is equal to restaurant. Sometimes it is more efficient to define what slots are applicable and sometimes what slots are not applicable. It is assumed that if 'A' values are specified for applicability then the default value for not specified slots is 'N/A' and the opposite when 'N/A' values are provided. An example of definition of applicability dependencies follows:

```
applicability = [('venue_type','restaurant','food_type', 'A'),
                 ('venue_type','restaurant','stars', 'N/A'),
                 ('venue_type','bar','stars', 'N/A')
                ]
```

Note that the last two rules could be more efficiently expressed by ('venue_type','hotel','stars', 'A').

The applicability dependencies can be illustrated on the next example. In Figure 2, there is dependence diagram describing the dependencies in the tourist information domain. When user says

`inform(food_type='Chinese')&inform(area='centre')`

then it means that the slots food_type and area must be set to the provided values.

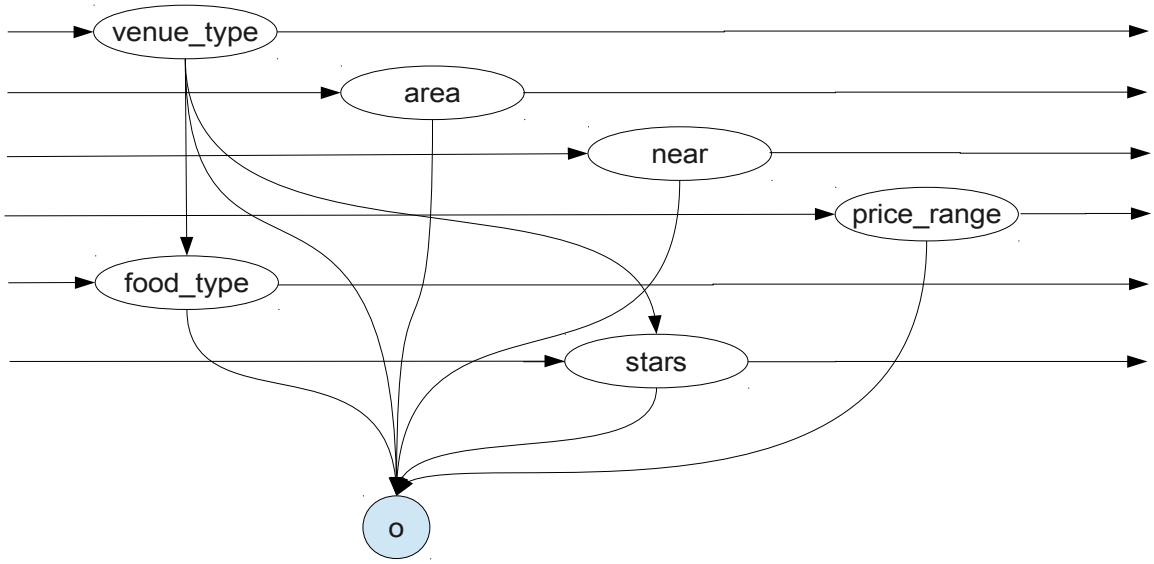
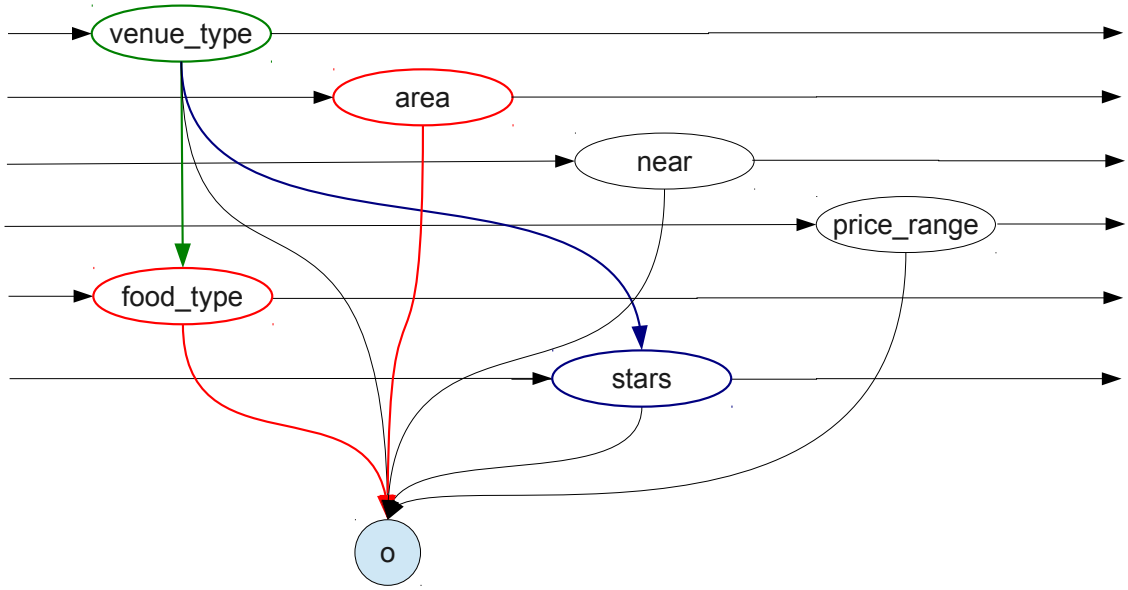


Figure 2: Applicability dependencies

Using the applicability dependencies, one can deduce that the value of the venue_type slot has to be set to restaurant since it is implied by the applicability rules. Consequently, the value in the stars has to be set to 'N/A' since the stars slot is not applicable when the value of the slot venue_type is restaurant. The used dependencies are depicted in Figure 3.



inform(food_type='Chinese')&inform(area='centre')

Figure 3: Example of used applicability dependencies

The main purpose of the applicability dependencies is to enable “forgetting of irrelevant information” during a course of a dialogue. For example, user requests a Chinese restaurant and then the user

changes its mind and wants a hotel. The applicability ensures that the information specific just to restaurant is forgotten, e.g. to make sure that the system is not assuming that the user wants a hotel serving Chinese food. Using the applicability dependencies reduces the state space and can help to make dialogue systems more robust. On the other hand, if a user really asked for a hotel serving Chinese food then the system is not able to answer that there are no hotels serving Chinese food.

Probabilistic extension of the dialogue state update

In the previous part of the report, deterministic update of a dialogue state was discussed. However, in real life this is not sufficient. The input of a dialogue system is affected by external noise, e.g. a car passing behind a user, grammatical errors in spoken speech or simply by ambiguity of the natural language. All these conditions affects the performance of the automatic speech recognition (ASR) and the spoken language understanding (SLU). Therefore, the input of a dialogues system is often erroneous.

Previously it was assumption that the system knows exactly what the user wants, e.g.

User says: I want a restaurant in a cheap price range.
User dialogue act: `inform(venue_type='restaurant')&`
`inform(price_range='cheap')`

However depending on the quality of ASR and SLU, real input can look like:

User says: I want a restaurant in a cheap price range.
User dialogue act: `inform(price_range='cheap')`

However in noisy conditions, it is better to consider a list of hypotheses about what user have said with a probability associated with each hypothesis where the probability then represents the likelihood of the hypothesis being correct. For example:

User says: I want a restaurant in a cheap price range.
User dialogue act: `0.6 : inform(venue_type='restaurant')&`
`inform(price_range='cheap')`
User dialogue act: `0.2 : inform(venue_type='restaurant')`
User dialogue act: `0.1 : inform(price_range='cheap')`
User dialogue act: `0.1 : null()`

Where the hypotheses inform(venue_type='restaurant')&inform(price_range='cheap') has likelihood being correct 0.6 and the hypothesis that user did not say anything (null()) is 0.1.

Under these conditions a deterministic update is no longer possible since different input dialogue acts lead to different updates resulting in different dialogue states each with a different probability. Therefore, a distribution over all dialogue states has to be maintained. Since there are billions of states even for the simplest dialogue systems, some approximations has to be introduced. Instead of maintaining joint distribution over all slots in the dialogue state, a factored distribution is defined and conditional independence of the factors is exploited to speed up the computation. Further approximation as Loopy belief propagation is assumed to be used to achieve real-time update

performance.

This section will describe a trivial extension of the deterministic update presented earlier in this report. Note that there are many other ways how to estimate dialogue state under such noisy conditions. The approach based on the extension of deterministic update is best described on an example. The Figure 4 depicts a Bayesian network with only one slot which depends on the previous value in the same slot and the observation.

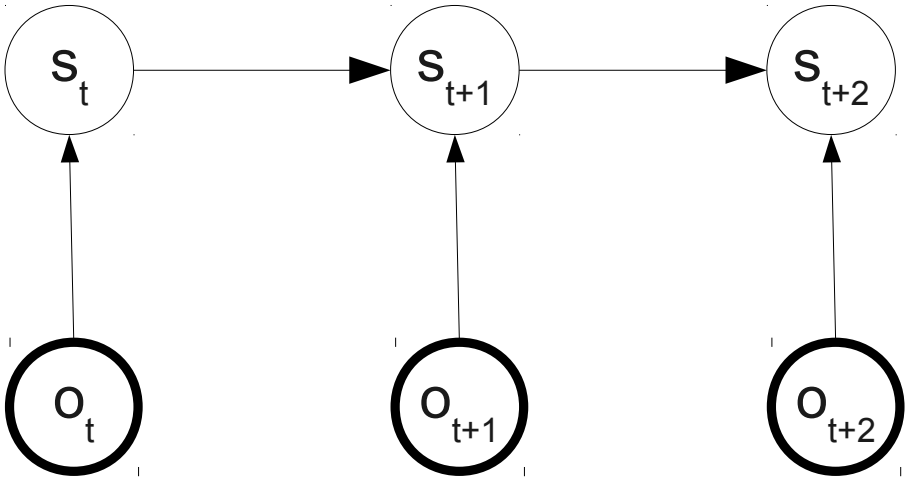


Figure 4: Example of Bayesian network for one slot, e.g. the venue_type slot.

Assuming the model above, the probability $p(s_{t+1})$ can be computed as follows:

$$\begin{aligned} p(s_{t+1}) &= \sum_{s_t, o_{t+1}} p(s_{t+1}, s_t, o_{t+1}) \\ &= \sum_{s_t, o_{t+1}} p(s_{t+1}|s_t, o_{t+1})\, p(s_t, o_{t+1}) \\ &\approx \sum_{s_t, o_{t+1}} p(s_{t+1}|s_t, o_{t+1})\, p(s_t)\, p(o_{t+1}) \end{aligned}$$

Which can be written as

$$p(s_{t+1}|p(s_t), p(o_{t+1})) \approx \sum_{s_t, o_{t+1}} p(s_{t+1}|s_t, o_{t+1})\, p(s_t)\, p(o_{t+1})$$

Probability $p(s_{t+1})$ is sometimes called belief state since it represents our beliefs about in what state the environment is and the computation of the belief state is called belief state update.

Since the probability $p(s_t)$ is obtained by the previous computation and the probability $p(o_{t+1})$ is provided by the SLU component, the most crucial part of the expression above is the transition

probability $p(s_{t+1}|s_t, o_{t+1})$. Following the ideas of the deterministic update presented earlier, the transition probability should be deterministic:

- $p(s_{t+1}|s_t, o_{t+1}) =$
  - $= \delta(s_{t+1}, o_{t+1}) \quad if\ o_{t+1} \neq \text{None} \wedge s_t \neq \text{None}$
  - $= \delta(s_{t+1}, s_t) \quad if\ o_{t+1} = \text{None} \wedge s_t \neq \text{None}$
  - $= \delta(s_{t+1}, o_{t+1}) \quad if\ o_{t+1} \neq \text{None} \wedge s_t = \text{None}$
  - $= \delta(s_{t+1}, \text{None}) \quad if\ o_{t+1} = \text{None} \wedge s_t = \text{None}$

Note that the “None” state value is the default value in the slot and the observation “None” is result of the user being silent about the slot.

This transition probability defines the following principles:

- if the observation is different to “None” then the slot should equal to the observation
  - overwrite what has been remembered before
- if the observation is equal to “None” then the slot should equal to the previous value
  - copy what was remembered in the previous turn

Assuming this very simple deterministic transition probability, a simple and therefore efficient update of the belief state can be derived

$$p(s_{t+1}|p(s_t), p(o_{t+1})) \approx \sum_{s_t, o_{t+1}} p(s_{t+1}|s_t, o_{t+1})\, p(s_t)\, p(o_{t+1})$$

$$p(s_{t+1}|p(s_t), p(o_{t+1})) \approx \sum_{s_t} \sum_{o_{t+1}} p(s_{t+1}|s_t, o_{t+1})\, p(s_t)\, p(o_{t+1})$$

$$\begin{aligned} p(s_{t+1}|p(s_t), p(o_{t+1})) \approx & \sum_{s_t \neq \text{None}} \sum_{o_{t+1} \neq \text{None}} p(s_{t+1}|s_t, o_{t+1})\, p(s_t)\, p(o_{t+1}) \\ & + \sum_{s_t \neq \text{None}} \delta(s_{t+1}, s_t)\, p(s_t)\, p(o_{t+1} = \text{None}) \\ & + \sum_{o_{t+1} \neq \text{None}} \delta(s_{t+1}, o_{t+1})\, p(s_t = \text{None})\, p(o_{t+1}) \\ & + \delta(s_{t+1}, \text{None})\, p(s_t = \text{None})\, p(o_{t+1} = \text{None}) \end{aligned}$$

$$\begin{aligned} p(s_{t+1}|p(s_t), p(o_{t+1})) \approx & (1 - \delta(s_{t+1}, \text{None}))(1 - p(s_t = \text{None}))\, p(o_{t+1} = s_{t+1}) \\ & + (1 - \delta(s_{t+1}, \text{None}))\, p(s_t = s_{t+1})\, p(o_{t+1} = \text{None}) \\ & + (1 - \delta(s_{t+1}, \text{None}))\, p(s_t = \text{None})\, p(o_{t+1} = s_{t+1}) \\ & + \delta(s_{t+1}, \text{None})\, p(s_t = \text{None})\, p(o_{t+1} = \text{None}) \end{aligned}$$

$$p(s_{t+1}|p(s_t), p(o_{t+1})) \approx (1 - \delta(s_{t+1}, \text{None}))\, p(o_{t+1} = s_{t+1}) + p(s_t = s_{t+1})\, p(o_{t+1} = \text{None})$$

The result is representation of the principles endowed to the transition probability:

- if the observation is different to “None” then the probability of the observation is used ($p(o_{t+1}=s_{t+1})$)
  - the probability information in the slot overwritten
- if the observation is equal to “None” then the probability of a value in the previous slot is used ($p(s_t=s_{t+1})$)
  - the probability information in the previous slot is copied

In the tourist domain, the previously described model can be used, for instance, to model update of the venue_type slot where the observations are only inform dialogue acts about the slot and the slot values are hotel, bar, and restaurant. The following table provides an example of the update using 4 observations where there is high uncertainty about what was observed (50% of probability mass of the observation is assigned to the value “None” which represents that the user did not say anything).

Observations		0_1	0_2	0_3	0_4
Values		0.5 - hotel	0.5 - bar	0.5 - hotel	0.5 - hotel
		0.5 - None	0.5 - None	0.5 - None	0.5 - None

Slots	s_0	s_1	s_2	s_3	s_4
Values	1.0 - None	0.5 - hotel	0.50 - bar	0.625 - hotel	0.8125 - hotel
		0.5 - None	0.25 - hotel	0.250 - bar	0.1250 - bar
			0.25 - None	0.125 - None	0.0625 - None

Initially all probability mass in the slots is assigned to the default value “None”. Then, after several iterations, the most probability mass is assigned to the value “hotel”. Note that that the probability of the value “hotel” in the slot is higher than observation the value “hotel”. This means that the model can accumulate probability mass over multiple turns.

Appendices

Appendix A: CamInforRest

Appendix B: CamInfo

Appendix C: Letiste Ptaha