# Production System Architecture

MD Asif Hasan

10/10/2017

**Overview**

The goal of this assignment is to develop a production system architecture that can be used for spatial reasoning. Production system uses rules and conditions to reason. Rules are applied on conditions and conditions are applied on variables. For example, *if a is on left of b then b is on right of a*.The architecture (PD) developed uses a long term memory (LTM) and a short term memory (STM). LTM is responsible to memorize all the rules whereas the STM keeps all the conditions known to the system. Furthermore, PD supports a query function that can be used to reason about spatial relationship between two variables. The following sections describes various aspects of the system.

**Development Platform:** Python

**Inputs**

There are basically two types of inputs, Conditions and Query. Format for each are given below:

Conditions:

Format: “condition_name, variable list”

Example: "left of", ["fork","plate"]

Conditions are added via instantiating a class called *Condition* and then calling *addCondition* function of short term memory data structure.

Example: pd.stm.addCondition(Condition("left of", ["fork","plate"]))

Query:

Current architecture supports querying spatial relationship between two variables. Production system provides a function for that.

Example:

pd.query("fork", "knife")

**Outputs**

When the production system is queried on spatial relationship between two variables, it prints the conditions (relationships) that holds true between the variables on console as well as return the list of conditions as output of the query function.

Example:

Input:

Query('plate', 'cat')

Output:

Spatial relation between plate and cat:

right of: ['plate', 'cat']

**Rules for Spatial Reasoning**

To solve reasoning problems of the particular kind concerned in this assignment, the following types of rules were used:

- If a is at *relation_name* of b then b is at *complementary_relation_name* of a
- If a is at *relation_name* of b and b is at *relation_name* of c then a is at *relation_name* of c

The first one deals with complementary relationships. For example,

- If a is at *right* of b then b is at *left* of a

The second one deals with chaining the rules. For example,

- If a is at *left* of b and b is at *left* of c then a is at *left* of c

A separate class called RuleCreator is used to create the rules. It provides a function to do that that takes in two parameters: *relation_name* and *complementary_relation_name*. Upon calling it creates four rules. For example:

Call: createRules(*relation_name* , *complementary_relation_name*)

Rules Created:

- If a is at *relation_name* of b then b is at *complementary_relation_name* of a
- If a is at *relation_name* of b and b is at *relation_name* of c then a is at *relation_name* of c
- If a is at *complementary_relation_name* of b then b is at *relation_name* of a
- If a is at *complementary_relation_name* of b and b is at *complementary_relation_name* of c then a is at *complementary_relation_name* of c

Here, a, b, c are variables that are assigned as part of rule processing procedure during runtime and depends on which variables exist in the system at that point.

**Query Algorithm**

Function: Query (a, b)

Step 1:

Find all the conditions related to variable a and b

Step 2:

Prepare a list of only those rules that deal with any of the set of conditions selected in previous step

Step 3:

Execute each of the rules selected from the previous step one by one by the order they were added in the list

Step 4:

Loop to step 1 if no new conditions were added in the last step

Step 5:

List the conditions in short term memory that are between a and b. If no such condition exists, respond, “I don’t know!”, otherwise, print the conditions.

**Executing the Program**

The program requires python 3.6 to run. To run the program the following command is to be executed from the command prompt.

python ProductionSystem.py <test_no>
<test_no> may be any number from 1 to 6.

**Results**

For the queries listed in the assignment instruction, the results are shown below along with the query.

Q1: The fork is to the left of the plate. The plate is to the left of the knife. Where is the fork, in relation to the knife?

A: left of: ['fork', 'knife'], which means fork is to the left of knife.

Q2: The fork is to the left of the plate. The plate is above the napkin. Where is the fork, in relation to the napkin?

A: I don't know!

Q3: The fork is to the left of the plate. The spoon is to the left of the plate. Where is the fork, in relation to the spoon?

A: I don't know!

Q4: The fork is to the left of the plate. The spoon is to the left of the fork. The knife is to the left
of the spoon. The pizza is to the left of the knife. The cat is to the left of the pizza. Where is the plate, in relation to the cat?

A: right of: ['plate', 'cat'], which means the plate is to the right of cat.

Query 5 is set up for experimenting the system's behavior when there are rules in the system that are not related to answer the concerned query. The query is answered correctly without processing rules that are not related. For this experiment, the rules are created using the following where the rules other than the first one is not necessary to answer the query.

```
rc.createRules("left of", "right of", pd.ltm)
rc.createRules("above of", "below of", pd.ltm)
rc.createRules("west", "east", pd.ltm)
rc.createRules("south", "nort", pd.ltm)
```

Conditions:

```
pd.stm.addCondition(Condition("left of", ["fork","plate"]))
pd.stm.addCondition(Condition("left of", ["plate", "knife"]))
```

Query:

```
query("fork", "knife")
```

Query 6 is extended from Query 5's set up for experimenting the system's behavior when there are conditions in the system that are not related to answer the concerned query. The query is answered correctly without processing rules that concerns the conditions not related. For this experiment following rules, conditions and query are used.

Rules are created by:

```
rc.createRules("left of", "right of", pd.ltm)
rc.createRules("above of", "below of", pd.ltm)
```

```
rc.createRules("west", "east", pd.ltm)
rc.createRules("south", "nort", pd.ltm)
```

Conditions:

```
pd.stm.addCondition(Condition("left of", ["fork","plate"]))
pd.stm.addCondition(Condition("left of", ["plate", "knife"]))
pd.stm.addCondition(Condition("left of", ["plate1", "knife1"]))
pd.stm.addCondition(Condition("left of", ["plate1", "knife1"]))
pd.stm.addCondition(Condition("left of", ["plate2", "knife2"]))
pd.stm.addCondition(Condition("left of", ["plate4", "knife3"]))
```

Query:

```
pd.queryFast("fork", "plate")
```

Query 5 and 6 iterates through same number of rules.

**Discussion**

While implementing the production system architecture for spatial reasoning, one point became evidently clear that the system requires accessing data in a really flexible way. For example, finding rules that can are applicable to a specific condition, For chaining rule, finding variables that the rule can be applied on, etc. A relational database can be of big help for such flexible querying of various relationships among variables, conditions and rules.

For the query, the first version was implemented in the following way.

- Step 1: Process all the rules in the system
- Step 2: Loop to step 1 until no new conditions are added in the system

The problem with this approach is computational complexity. If the system has large number of rules, this algorithm will go through all of those even though the query may need only a tiny fraction of these rules to be processed. Furthermore, if the system has conditions which are not related to the query, the rules would continue to process until all the conditions are resolved to a point that no new condition can be reasoned. It may be the case that the goal of the query is achieved far earlier than the algorithm actually finishes. So, the new algorithm as described in “Query Algorithm” section is devised, which processes only those rules and takes into account only those conditions that are related to the query.

To test if the goal is reached for the query, the algorithm uses the condition that, no new conditions were added in the last iteration of processing the related rules. This is used as the goal test because, it is assumed that between two variables, there may be multiple spatial relationships. Furthermore, a condition which is valid after an iteration, due to an effect of another rule in a future iteration, may become invalid. Although for the queries experimented with in this assignment, these scenarios can not be reached, the decision of this kind of goal test was taken in order to keep the query algorithms generic enough.

Talking about generic use of the production system, support for setting a condition to be false at runtime is not added. This will be a trivial implementation in the current architecture which requires all the references of the conditions kept in different data structures need to be removed. Similarly, no support is added for removing rules from the system in runtime.

Lastly, the sequence of query variables is important. Query(a, b) is not same as Query(b, a).

I believe the algorithm can be further optimized by introducing better goal testing, possibly including some of the domain knowledge too. Having to use basic data structure such as array and dictionary is not sufficient at all. If a database were used, exploring how that would impact the power of the system (such as creating rules, query algorithm, etc.) would be interesting. The query algorithm currently supports querying relations between only two parameters. But the way the system and the algorithm is setup it can handle querying of relationship between arbitrary number of variables. Exploring that would be also interesting.