STATEFUL KNOWLEDGE SESSION IMPLEMENTATION WITH RETE ALGORITHM

Daniela GOTSEVA
Technical University of Sofia, Bulgaria

Abstract. This paper discusses some concepts related to the Rete Algorithm, which consist of two parts: compilation and runtime execution. After a brief explanation of Rete Algorithm and its combination with the latest technology the stateful knowledge session is discussed and the cross product is presented.

Keywords: Rete algorithm, artificial intelligence

1. Introduction into Rete algorithm

The Rete algorithm was invented by Dr. Charles Forgy in 1978-79 [1, 2, 6]. It can be broken into 2 parts: rule compilation and runtime execution.

The compilation algorithm describes how the Rules in the Production Memory are processed to generate an efficient discrimination network. The nodes at the top of the network would have many matches, and as we go down the network, there would be fewer matches. At the very bottom of the network are the terminal nodes. In Dr. Forgy's 1982 paper, he described four basic nodes: root, 1-input, 2-input and terminal (Figure 1).

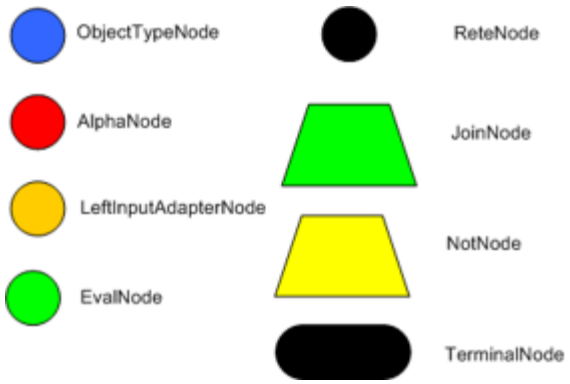

Figure 1. Rete Nodes

The root node is where all objects enter the network. From there, it immediately goes to the ObjectTypeNode (Figure 2). The purpose of the ObjectTypeNode is to make sure the engine doesn't do more work than it needs to. For example, say we have two objects: Account and Order. If the rule engine tried to evaluate every single node against every object, it would waste a lot of cycles. To make things efficient, the engine should only pass the object to the nodes that match the object type. The easiest way to do this is to create an ObjectTypeNode and have all 1-input and 2-input nodes descended from it. This way, if an application asserts a new Account, it won't propagate to the nodes for the Order object. In Drools when an object is asserted it retrieves a list of valid ObjectTypesNodes via a lookup in a HashMap from the object's Class; if this list doesn't exist it scans all the ObjectTypeNodes finding valid matches which it caches in the list. This enables Drools to match against any Class type that matches with an instance of check.

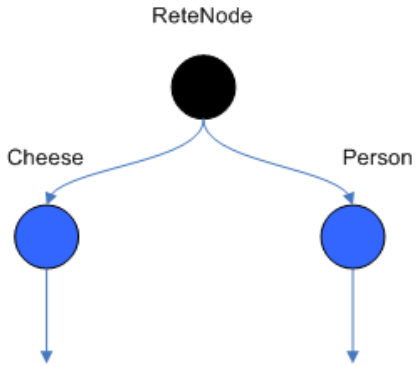

Figure 2. ObjectTypeNodes

Drools extend Rete by optimizing the propagation from ObjectTypeNode to AlphaNode using hashing. Each time an AlphaNode (Figure 3) is added to an ObjectTypeNode it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectTypeNode, rather than propagating to each AlphaNode, it can instead retrieve the correct AlphaNode from the HashMap, thereby avoiding unnecessary literal checks.

In this paper the combination between Rete Algorithm and the latest technologies is described, and how all fuss together to provide the DSS (Decision Support System).

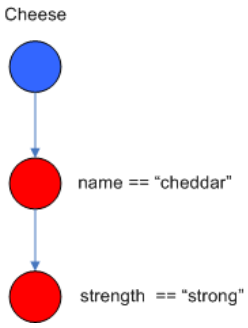

Figure 3. Alpha Nodes

2. Knowledge sessions

2.1. Stateless knowledge session

Stateless session, not utilising inference, forms the simplest use case. A stateless session can be called like a function passing it some data and then receiving some results back [3÷5]. Some common use cases for stateless sessions are, but not limited to:

- Validation
- Calculation
- Routing and Filtering

An explanation on how to use Stateless Knowledge Session can be found in [7].

2.2. Stateful knowledge session

Stateful Sessions [1] are longer lived and allow iterative changes over time. Some common use cases for Stateful Sessions are, but not limited to:

- Monitoring
- Stock market monitoring and analysis for semi-automatic buying.
- Diagnostics
- Fault finding, medical diagnostics
- Logistics
- Parcel tracking and delivery provisioning
- Compliance
- Validation of legality for market trades.

In contrast to a Stateless Session, the dispose() method must be called afterwards to ensure there are no memory leaks, as the Knowledge Base contains references to Stateful Knowledge Sessions when they are created. StatefulKnowledgeSession also supports the BatchExecutor interface, likeStatelessKnowledgeSession, the only difference being that the FireAllRules command is not automatically called at the end for a Stateful Session.

The monitoring use case with an example for raising a fire alarm is illustrated. Using just four classes, we can represent rooms in a house, each of which has one sprinkler. If a fire starts in a room, this can be represented that with a single Fire instance (Listing 1).

```
public class Room {
private String name
   // getter and setter methods here
}

public class Sprinkler {
private Room room;
privateboolean on;
   // getter and setter methods here
}

public class Fire {
private Room room;
   // getter and setter methods here
}

public class Alarm {
}
```

Listing 1. Java Classes for Fire Alarm Example

Let's introduce the concepts of inserting and matching against data. This example assumed that only a single instance of each object type was ever inserted and thus only used literal constraints. However, a house has many rooms, so rules must express relationships between objects, such as a sprinkler being in a certain room. This is best done by using a binding variable as a constraint in a pattern. This "join" process results in what is called cross products, which are covered in the next section.

When a fire occurs, an instance of the Fire class is created, for that room, and inserted into the session. The rule uses a binding on the room field of the Fire object to constrain matching to the sprinkler for that room, which is currently off. When this rule fires and the consequence is executed the sprinkler is turned on (Listing 2).

```
rule "When there is a fire turn on the sprinkler"
when
Fire($room : room)
   $sprinkler : Sprinkler( room == $room, on == false )
then
   modify( $sprinkler ) { setOn( true ) };
System.out.println( "Turn on the sprinkler for room " + $room.getName() );
end
```

Listing 2. Rule for Fire Alarm Example

Whereas the Stateless Session uses standard Java syntax to modify a field [7], in the above rule the modify statement, which acts as a sort of "with" statement is used. It may contain a series of comma separated Java expressions, i.e., calls to setters of the object selected by the modify statement's control

expression. This modifies the data, and makes the engine aware of those changes so it can reason over them once more. This process is called inference, and it's essential for the working of a Stateful Session. Stateless Sessions typically do not use inference, so the engine does not need to be aware of changes to data. Inference can also be turned off explicitly by using the sequential mode.

So far we have rules that tell us when matching data exists, but what about when it does *not* exist? How do determine that a fire has been extinguished, i.e., that there isn't a Fire object anymore? Previously the constraints have been sentences according to Propositional Logic, where the engine is constraining against individual instances. A pattern under the keyword not matches when something does not exist. The rule given in Listing 3 turns the sprinkler off as soon as the fire in that room has disappeared.

```
rule "When the fire is gone turn off the sprinkler"
when
   $room : Room( )
   $sprinkler : Sprinkler( room == $room, on == true )
not Fire( room == $room )
then
   modify( $sprinkler ) { setOn( false ) };
System.out.println( "Turn off the sprinkler for room " + $room.getName() );
end
```

Listing 3. Rule for turning off the sprinkler

While there is one sprinkler per room, there is just a single alarm for the building. An Alarm object is created when a fire occurs, but only one Alarm is needed for the entire building, no matter how many fires occur. Previously not was introduced to match the absence of a fact; now it is used its complement exists which matches for one or more instances of some category (Listing 4).

```
rule "Raise the alarm when we have one or more fires"
when
exists Fire()
then
insert( new Alarm() );
System.out.println( "Raise the alarm" );
end
```

Listing 4. Rule for raising alarm for >= 1 fires

Likewise, when there are no fires we want to remove the alarm, so the not keyword can be used again (Listing 5).

```
rule "Cancel the alarm when all the fires have gone"
when
not Fire()
   $alarm : Alarm()
then
retract( $alarm );
System.out.println( "Cancel the alarm" );
end
```

Listing 5. Rule for cancel alarm

Finally there is a general health status message that is printed when the application first starts and after the alarm is removed and all sprinklers have been turned off (Listing 6).

```
rule "Status output when things are ok"
when
not Alarm()
not Sprinkler( on == true )
then
System.out.println( "Everything is ok" );
end
```

Listing 6. Rule for status oputput

The above rules should be placed in a single DRL file and saved to some directory on the classpath and using the file name fireAlarm.drl, as in the Stateless Session example. Then it is building a Knowledge Base, as before, just using the new name fireAlarm.drl. The difference is that this time we create a Stateful Session from the Knowledge Base, whereas before we created a Stateless Session (Listing 7).

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource( "fireAlarm.drl", getClass() ),
              ResourceType.DRL );
if ( kbuilder.hasErrors() ) {
   System.err.println( kbuilder.getErrors().toString() );
}
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
```

Listing 7. fireAlarm file

With the session created it is now possible to iteratively work with it over time. Four Room objects are created and inserted, as well as one Sprinkler object for each room. At this point the engine has done all of its matching, but no rules have fired yet. Calling ksession.fireAllRules() allows the matched rules to fire, but without a fire that will just produce the health message (Listing 8).

```
String[] names = new String[]{"kitchen", "bedroom
", "office", "livingroom"};
Map<String,Room> name2room = new HashMap<
String,Room>();
for( String name: names ){
   Room room = new Room( name );
   name2room.put( name, room );
   ksession.insert( room );
   Sprinkler sprinkler = new Sprinkler( room );
   ksession.insert( sprinkler );
}

ksession.fireAllRules();
> Everything is ok
```

Listing 8. Main program

Then it is created two fires and insert them; this time a reference is kept for the returned FactHandle. A Fact Handle is an internal engine reference to the inserted instance and allows instances to be retracted or modified at a later point in time. With the fires now in the engine, once fireAllRules() is called, the alarm is raised and the respective sprinklers are turned on (Listing 9).

```
Fire kitchenFire = new Fire( name2room.get( "kitch
en" ) );
Fire officeFire = new Fire( name2room.get( "office"
 ) );

FactHandle kitchenFireHandle = ksession.insert( kit
chenFire );
FactHandle officeFireHandle = ksession.insert( offic
eFire );

ksession.fireAllRules();
> Raise the alarm
> Turn on the sprinkler for room kitchen
> Turn on the sprinkler for room office
```

Listing 9. FactHandle engine

After a while the fires will be put out and the Fire instances are retracted. This results in the sprinklers being turned off, the alarm being cancelled, and eventually the health message is printed again. The testing of the system is presented in Listing 10.

2.3. Methods versus Rules

People often confuse methods (Listing 11) and rules (Listing 12), and new rule users regular ask, "How do I call a rule?" After the last section the answer to that is obvious, but let's summarize the differences nonetheless.

- Methods are called directly.
- Specific instances are passed.
- One call results in a single execution.
- Rules execute by matching against any data as long it is inserted into the engine.
- Rules can never be called directly.
- Specific instances cannot be passed to a rule.
- Depending on the matches, a rule may fire once or several times, or not at all.

```
ksession.retract( kitchenFireHandle );
ksession.retract( officeFireHandle );

ksession.fireAllRules();
> Cancel the alarm
> Turn off the sprinkler for room office
> Turn off the sprinkler for room kitchen
> Everything is ok
```

Listing 10. System Testing

```
public void helloWorld(Person person) {
if ( person.getName().equals( "Chuck" ) ) {
System.out.println( "Hello Chuck" );
   }
}
```

Listing 11. Method sample

```
rule "Hello World"
when
Person( name == "Chuck" )
then
System.out.println( "Hello Chuck" );
end
```

Listing 12. Rule sample

3. Cross products

Earlier the term "cross product" was mentioned, which is the result of a join. Imagine for a moment that the data from the fire alarm example were used in combination with the following rule where there are no field constraints (Listing 13).

```
rule
when
   $room : Room()
   $sprinkler : Sprinkler()
then
System.out.println( "room:" + $room.getName()
 +" sprinkler:" +
 $sprinkler.getRoom().getName() );
end
```

Listing 13. Cross product

In SQL terms this would be like doing **select * from Room**, Sprinkler and every row in the Room table would be joined with every row in the Sprinkler table resulting in the output shown in Listing 14.

```
room:officesprinkler:office
room:officesprinkler:kitchen
room:officesprinkler:livingroom
room:officesprinkler:bedroom
room:kitchensprinkler:office
room:kitchensprinkler:kitchen
room:kitchensprinkler:livingroom
room:kitchensprinkler:bedroom
room:livingroomsprinkler:office
room:livingroomsprinkler:kitchen
room:livingroomsprinkler:livingroom
room:livingroomsprinkler:bedroom
room:bedroomsprinkler:office
room:bedroomsprinkler:kitchen
room:bedroomsprinkler:livingroom
room:bedroomsprinkler:bedroom
```

Listing 14. Output from cross product

These cross products can obviously become huge, and they may very well contain spurious data. The size of cross products is often the source of performance problems for new rule authors. From this it can be seen that it's always desirable to constrain the cross products, which is done with the variable constraint (Listing 15).

```
rule
when
   $room : Room()
   $sprinkler : Sprinkler( room == $room )
then
System.out.println( "room:" + $room.getName() +
" sprinkler:" + $sprinkler.getRoom().getName() );
end
```

Listing 15. Variable constraint

```
room:officesprinkler:office
room:kitchensprinkler:kitchen
room:livingroomsprinkler:livingroom
room:bedroomsprinkler:bedroom
```

Listing 16. Correct cross product

This results in just four rows of data, with the correct Sprinkler for each Room. In SQL (actually HQL) the corresponding query would be **select * from Room, Sprinkler where Room == Sprinkler.room**, which is shown in Listing 16.

4. Conclusion

In the following paper the Rete Algorithm is described and the latest technologies that can be combined with it. All specific codding patterns are shown here to express the way in which Rete can be realized. It's a good idea to create a template for statefull knowledge session, which allows iterative changes at long period of time in the future.

References

1. Selvamony, R. (2010) *Introduction to the Rete algorithm*. SAP Community Network. Available at: www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/10dea1d3-fbef-2d10-0e89-a7447f95bc0e?overridelayout=true&49868865442396
2. *** *Rete Algorithm*. Available at: http://en.wikipedia.org/wiki/Rete_algorithm
3. Winston, P.H. (1992) *Artificial Intelligence*. Third Edition, Pearson Education, ISBN-13: 978-0201533774
4. Patterson, D.W. (1990) *Introduction to Artificial Intelligence and Expert Systems*. Prentice Hall India, ISBN: 978-81-203-0777-3
5. Russel, St., Norvig, P. (2009) *Artificial Intelligence- A Modern Approach*. 3rd Edition, Prentice Hall, ISBN 978-0136042594
6. Madden, N. (2003) *Optimizing Rete for Low-Memory, Multi-Agent Systems*. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.6328&rep=rep1&type=pdf
7. Dimakopoulos, I. (2013) *Implementing Rete Algorithm with Stateless Knowledge Session*. Proceedings of International Conference on Challenges in Higher Education and Research in the 21st Century, Sozopol, Bulgaria

Received in August 2013