PROMETHEUS AI
Phase 1

Sean Stappas

260639512

ECSE-498: Honours Thesis I

Supervised by: Prof. Joseph Vybihal

April 11, 2017

Abstract

Prometheus AI is a model of the human brain with the goal of controlling multiple robots in a swarm environment. This could be useful in environments hazardous for humans, such as the aftermath of a nuclear power disaster, or in outer space. The model consists of four layers: the Neural Network (NN), the Knowledge Node Network (KNN), the Expert System (ES), and the Meta Reasoner (META). The NN classifies the signals coming from the robots' sensors and sends formatted tags to the KNN. The KNN represents memory and can initiate cascaded activation of memories in the form of tags, which are passed on to the ES. The ES is a simple logic reasoner and provides a recommendation for an action to the Meta Reasoner. The META represents high-level thinking and makes an intelligent decision for what the robots should do. The assigned task this semester was to implement prototypes of the KNN and ES layers in Java. This was achieved using specific design criteria and extensive feedback from the project supervisor, Prof. Vybihal. Personal design criteria included using object-oriented programming principles, optimizing the system for speed and space efficiency, and maximizing code readability. Tests were created in TestNG and extensive documentation was written in Javadoc[1].

Acknowledgments

---

[1] The Javadoc can be found here: `http://cs.mcgill.ca/~sstapp/prometheus/index.html`.

Contents

List of Figures

List of Tables

Listings

Abbreviations

NN	Neural Network
KNN	Knowledge Node Network
ES	Expert System
META	Meta Reasoner
OOP	Object-Oriented Programming

1 Introduction

The goal of this project is to create an artificial intelligence system to control multiple robots. Applications for this type of system include robots in hazardous environments, such as in outer space (Mars, Moon, etc.), in nuclear plants after a nuclear disaster, and in military zones.

The structure of the system is inspired from the functionality of the human brain, and is composed of the following four layers (in order of increasing abstraction): the Neural Network (NN), the Knowledge Node Network (KNN), the Expert System (ES), and the Meta Reasoner (META), as can be seen in Figure 1. These will be described in Section 2.
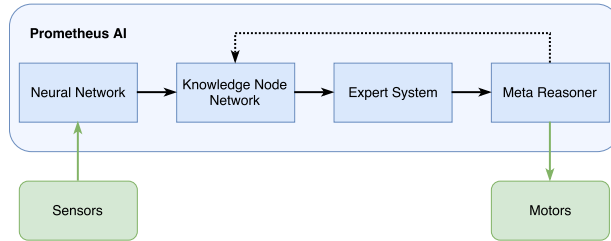
Figure 1: Prometheus AI model.

2 Background

2.1 Neural Network

The NN layer consists of a network of neurons with a similar structure to neurons in the human brain. In the context of this project, it is the interface between the robots' sensors and the rest of the AI system. Informational tags are generated by this layer based on observations that the robots make in their environment. These tags are passed onto the KNN.

The NN layer is based around the perceptron. A simple model of the perceptron can be seen in Figure 2.
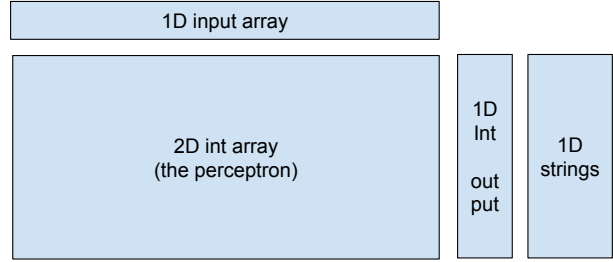
Figure 2: High-level model of the perceptron of the NN [1].

2.2 Knowledge Node Network

The KNN layer represents memory in the human brain. It takes in the tags provided by the NN and outputs tags based on its knowledge of the environment. These output tags can be simple facts, such as "I see a wall", or can be recommendations for future actions such "turn left". These tags are passed on to the next layer, the Expert System (ES).

The KNN is based around the Knowledge Node, which is an abstract structure representing a memory and its connections to other memories. A simple model of a Knowledge Node can be seen in Figure 3.
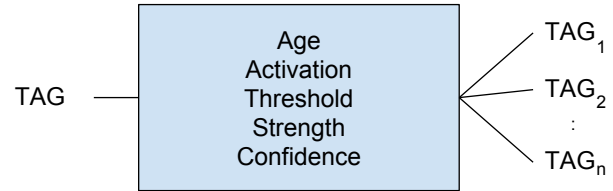
Figure 3: High-level model of the Knowledge Node of the KNN [2].

In its simplest form, the Knowledge Node can be excited by an activated input tag, which increments its activation parameter, which initially

starts at 0. If the activation parameter is greater than the node's threshold, the Knowledge Node fires, causing the activation of the output tags. This description corresponds to forwards thinking with linear activation. The activation can also be with a sigmoid function, which is more representative of neurons in the brain.

Other ways of thinking are also possible. Indeed, the KNN has three main ways of thinking: forwards, backwards, and lambda. The version of thinking to be done by the KNN is chosen by the META.

Forwards thinking is the simplest and was described earlier. It is depicted in Figure 4. Firing a Knowledge Node can cause forward activation of more Knowledge Nodes, hence the "forwards" naming.

Figure 4: Thinking forwards in the KNN.

Thinking backwards starts at the output tags of Knowledge Nodes, and works backwards, as can seen in Figure 5. In its simplest f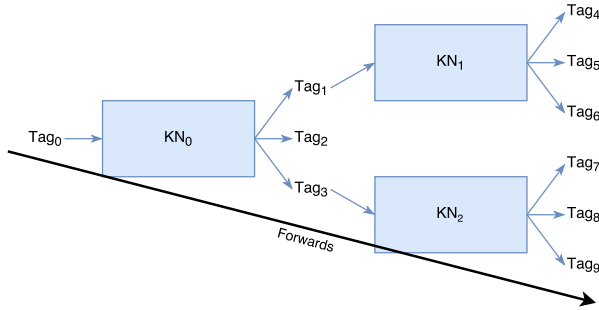orm, it checks output tags of Knowledge Nodes and, if all of them are active, the input tag must be active as well, so that Knowledge Node is activated.

This type of thinking can be a bit more difficult to implement than forwards thinking, since one must decide what percentage of active output tags of a Knowledge Node corresponds to an active input tag. For instance, should a Knowledge Node activate when all its output nodes are active (100%), or when 75% are active? This relates to the confidence with which the KNN believes the tag associated with that node to be true. As a concrete example, if you observe an object that has four wheels, seats, and a steering wheel, how confident are you that that object is

a car? Realistically, humans with often classify what they observe with a certain uncertainty and this is what can be represented with this type of backwards thinking. The memories of this observation are then even more uncertain.

This type of thinking occurs constantly in the background in humans [2], and this is something to keep in mind during implementation.
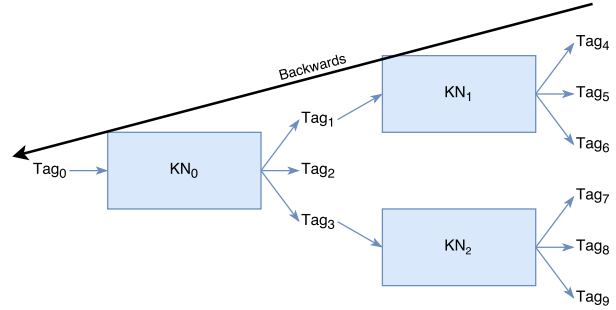
Figure 5: Thinking backwards in the KNN.

Lambda thinking uses a combination of forwards and backwards, and can be seen in Figure 6. It is called "lambda" thinking because the shape of the thinking trajectory matches a Greek uppercase lambda (Λ). It first looks at output tags of Knowledge Nodes and propagates activation backwards, similar to backwards thinking. After a certain amount of nodes are fired, it will then start activating forwards, similar to thinking forwards. An interesting question is how far backwards should lambda thinking go before starting to cascade forwards? This relates to how general one wants to explore before searching for a more specific value.

This type of thinking occurs in humans when using analogical reasoning to solve problems [3]. In essence, when a person wants to find a solution to a problem, they will explore related problems, move their way "backwards" to concepts related to the desired problem, and think "forwards" to focus in on the desired problem.

All forms of thinking can continue until there are no more Knowledge Nodes to activate, which corresponds to natural quiescence. There can also be a fixed number of cycles through the list of active tags. This fixed cycle number represents how much effort is being put into thinking.

Figure 6: Lambda thinking in the KNN.

Indeed, in animals, most thinking is never done at 100% effort, and it would be interesting to model this.

2.3 Expert System

The ES layer is a basic logic reasoner. It is not aware of its current reality, or any context. It takes in the tags provided by the KNN and interprets them as either facts, recommendations or rules.

Facts are simple calculus predicates showing that something is true. For example, the fact (A) represents that A itself is true or active, $(A = 1)$ means that A is equal to 1, $(A > 1)$ repres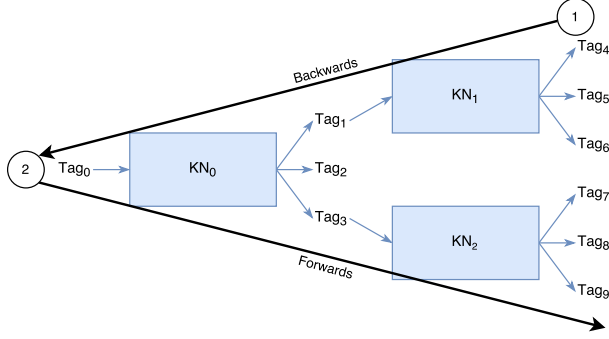ents that A is greater than 1, etc. As a more concrete example, a fact can represent a certain measurement, like $(distance = 5)$ representing the robot's distance from a wall as measured by one of its sensors.

Recommendations represent suggestions for actions to be taken by a robot. For example, $(\#turn\_left)$ is a recommendation for a robot to turn left, if it sees a wall directly in front of it and must avoid it, for example. These are recommendations and not commands because the META can decide whether or not to actually take that action.

Rules are a many-to-many structure with facts or recommendations as inputs and outputs. When all the input tags become active, the output tags become active and the rule itself is also said to be active. In this way, a rule can represent a logical AND of all its input tags.

The runtime of the ES consists of the follow-ing general steps [4]:

1. Reset

2. Add facts and rules

3. Think

4. Send recommendations to META

The most important part of the previous process is the thinking stage, which consists of first iterating through all the rules in the ES and checking if they are active by inspecting the lists of facts and recommendations. Rules may then become active and cause cascading activation of more rules. This can continue until there are no more rules to activate, which corresponds to natural quiescence. There can also be a fixed threshold to the number of possible cycles through the lists of facts and recommendations. In this sense, this limit represents the effort put into thinking, similarly to the KNN.

These recommendations are passed on to the final layer, the Meta Reasoner (META).

2.4 Meta Reasoner

The META layer represents high-level reasoning in human brains. It is aware of its environment and context, and makes decisions based on what it believes to be right. It is paranoid, and constantly checks whether the tags reported by the rest of the AI system make sense based on its expected view of the world. If it decides to make a decision, it sends a command to the actuators of the robots to decide how to move. If it is not happy with the recommendation(s) from the ES, it may initiate another think cycle in the KNN to generate new recommendations(s).

2.5 Summary

With this full description of the Prometheus AI model, the system with labeled input and output can be seen in Figure 7.
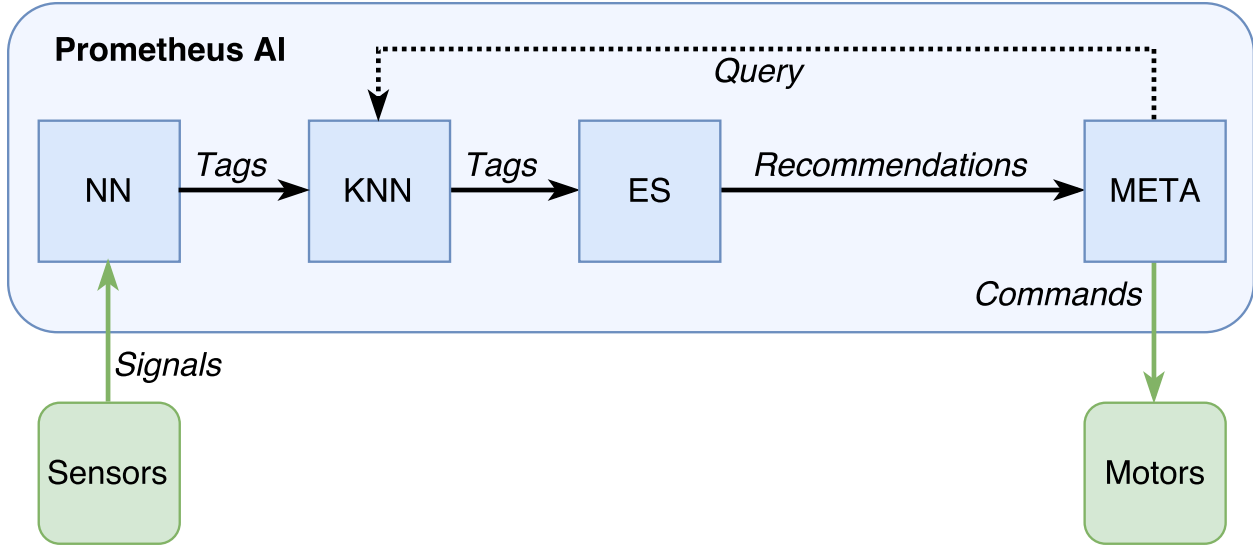
6

Figure 7: Prometheus AI model with labeled input and output.

3 Problem

The assigned task was to construct two out of the four layers listed in Section 2: the Expert System (ES), and the Knowledge Node Network (KNN). The other two layers are to be completed by another Honours Thesis student.

As a deliverable for the end of this first semester, it was required to complete a prototype of these two layers, with basic versions of the functionality described in Section 2. This was to be done entirely in Java.

4 Design Criteria

4.1 Efficiency

A very important consideration when designing the system is speed. Since the robots may have to react very quickly to stimulus in the environment, the reasoning in the AI must be as fast as possible. This is especially true in the hazardous environments for which this system could be useful for, as specified in Section 1. An example of a design choice that was made to improve speed is the use of Java Sets for most of the collections in the ES and KNN layers. The original specifications mentioned using ArrayLists, but, since there is no specific iteration order necessary for most operations in the ES and KNN,

these collections were changed to Sets.

When designing every algorithm in the system, the first criterion in mind is speed. Indeed, for the `think()` methods of the ES and KNN, it was important to think about the right way to leverage all the available data structures to maximize speed.

4.2 Object Oriented Design

Another important choice is to leverage object-oriented design as much as possible. Object-oriented programming (OOP) allows extensive planning before even beginning to write code, which can identify any flaws in the initial design. It also allows the code to be very clean and reusable. Since Java is the programming language chosen for the project, OOP is also the natural way to proceed. To follow OOP, the ES and KNN will be designed around Java classes and the methods associated with those classes. OOP principles such as polymorphism and encapsulation will be followed closely.

4.2.1 Abstraction

One important design aspect is that the system should be as abstract as possible, while still performing its desired task. For instance, the system should be general enough to perform un-

7

der simulations, as well as in real-life environments. It should also ideally be able to perform in vastly different environments, with different tasks.

One example of making use of abstraction is the choice to create a `PrometheusLayer` interface for both the ES and KNN, since they share some functionality, like the ability to think.

4.2.2 Encapsulation

Each layer of the system has unique, localized functionality that does not need to be visible from the rest of the system. For instance, the intricacies of the `think()` method in the ES and KNN layers do not need to be known to the rest of the system.

4.2.3 Polymorphism

Many methods in the system can have different functionality depending on context. For instance, the `think()` method in the KNN and ES layers can take an optional number of think cycles to specify a threshold for quiescence.

4.2.4 Inheritance

Careful thought was put into how objects may inherit functionality from each other. This can most clearly be seen with the implementation of the tags, which will we described in Section 5.

5 Implementation

The details of how the KNN and ES were implemented in Java will now be discussed.

5.1 Tags

The entire system revolves around tags passed from layer to layer. For this reason, a lot of thought was put into the proper design of these tags. To leverage OOP principles and readability, it was decided to implement these tags with a `Tag` class in Java, with `Fact`, `Rule` and `Recommendation` subclasses.

The tags described in Section 2 need to be as general as possible. A natural choice for this structure would be a Java `String`, which would be relatively simple to pass around the system. However, these tags represent various concepts; each tag can either be a fact, a recommendation, or a rule. If implemented as `String`s, the tags would have to be encoded on creation to represent each concept and decoded on use to retrieve the important information. This seems like a bad use of the OOP principles of Java. For this reason, the tags are instead implemented using a `Tag` Java class, with `Recommendation`, `Fact`, and `Rule` subclasses. This should also make manipulating the Tags faster, while incurring a slight memory overhead. To store these Tags in a database, they can be converted to JSON format. On read from the database, they can be easily decoded.

The `Tag` class itself was made abstract. This means that an object may not be directly instantiated as a `Tag`, but must be instantiated as one of its subclasses. This makes sense, since a tag *must* be one of the three types: rule, recommendation or fact.

All of these classes were placed inside the `tags` package. A UML diagram of the `tags` package can be seen in Figure 9 of the Appendix.

5.2 Knowledge Node Network

All code relating directly to the KNN was placed in the `knn` package of the project. A UML diagram of the `knn` package can be seen in Figure 10 of the Appendix.

The KNN layer is based around the `KnowledgeNodeNetwork` Java class. This class has the following fields:

mapKN   Map of input `Tags` to associated `KnowledgeNodes`

activeTags   Set of active `Tags`, corresponding to input `Tags` of fired `KnowledgeNodes`

The `KnowledgeNode` class implements the functionality of a Knowledge Node, which has

the following important fields implemented functionality described in Section 2:

inputTag   input `Tag`.

outputTags `Array` of output `Tags`.

activation `int` starting at 0, incrementing when the `KnowledgeNode` is excited.

threshold   `int` threshold such that `activation` ≥ `threshold` causes firing of the `KnowledgeNode`.

strength   `int` that biases the activation of a `KnowledgeNode`, causing early firing. Simple implementation: if `activation * strength` ≥ `threshold` then fire the KN.

confidence `int` representing the belief that `inputTag` is true (0 to 100%).

age   `int` representing the age of the `KnowledgeNode`.

The most important method in the `KnowledgeNodeNetwork` is `think()`, which chooses either `thinkForwards()`, `thinkBackwards()`, or `thinkLambda()`. These methods implement the functionality described in Section 2 and return the `Tags` activated as a result of thinking.

Without a parameter, the `think()` method runs to natural quiescence. There is also an overloaded version of `think()` that takes an `int numberOfCycles` as a parameter. This parameter represents the thinking effort described earlier. Similarly, `thinkForwards()`, `thinkBackwards()`, and `thinkLambda()` all have overloaded versions with `numberOfCycles` as a parameter.

Currently, only the `thinkForwards()` method is fully completed. The version of the method that runs until quiescence can be seen in Listing 1.

Listing 1: Thinking forwards until quiescence in the KNN.

```java
private Set<Tag> thinkForwards() {
    Set<Tag> totalActivatedTags = new HashSet<>();
    Set<Tag> activatedTags;
    do {
        activatedTags = forwardThinkCycle();
        totalActivatedTags.addAll(activatedTags);
    } while (!activatedTags.isEmpty());
    return totalActivatedTags;
}
```

The version of `thinkForwards()` with a fixed number of thinking cycles can be seen in Listing 2.

Listing 2: Thinking forwards for a fixed number of cycles in the KNN.

```java
private Set<Tag> thinkForwards(int numberOfCycles) {
    Set<Tag> totalActivatedTags = new HashSet<>();
    for (int i = 0; i < numberOfCycles; i++) {
        Set<Tag> activatedTags = forwardThinkCycle();
        if (activatedTags.isEmpty()) {
            break;
        }
        totalActivatedTags.addAll(activatedTags);
    }
    return totalActivatedTags;
}
```

One can see that the `thinkForwards()` methods repeatedly call `forwardThinkCycle()`, which can be seen in Listing 3. This method is where `Tags` can become active, being added into the `activeTags` Set.

Listing 3: Single cycle of thinking forwards in the KNN.

```java
private Set<Tag> forwardThinkCycle() {
    Set<Tag> allPendingActiveTags = new HashSet<>();
    for (Tag t : activeTags) {
        if (mapKN.containsKey(t)) {
            Set<Tag> pendingActiveTags =
                excite(mapKN.get(t));
            if (!pendingActiveTags.isEmpty()) {
                for (Tag tag : pendingActiveTags) {
                    if (!activeTags.contains(tag))
                        allPendingActiveTags.add(tag);
                }
            }
        }
    }
    activeTags.addAll(allPendingActiveTags);
    return allPendingActiveTags;
}
```

One can see that the `forwardThinkCycle()` calls an `excite()` method to excite a `KnowledgeNode`. This method returns the `Tags` activated as a result of excitation. The code can be seen in Listing 4.

Listing 4: Method to excite a Knowledge Node.

```
private Set<Tag> excite(KnowledgeNode knowledgeNode) {
    Set<Tag> pendingActiveTags = new HashSet<>();
    knowledgeNode.activation++;
    if (knowledgeNode.activation >=
        knowledgeNode.threshold) {
        pendingActiveTags = fire(knowledgeNode);
    }
    return pendingActiveTags;
}
```

This method, in turn, may call a `fire()` method to fire the `KnowledgeNode` and activate its output tags. It returns all the `Tags` activated as a result of firing that `KnowledgeNode`. The code can be seen in Listing 5.

Listing 5: Method to fire a Knowledge Node, activating its output tags.

```
private Set<Tag> fire(KnowledgeNode knowledgeNode) {
    Set<Tag> pendingActiveTags = new HashSet<>();
    for (Tag outputTag : knowledgeNode.outputTags) {
        if (!activeTags.contains(outputTag)) {
            pendingActiveTags.add(outputTag);
        }
    }
    return pendingActiveTags;
}
```

5.3 Expert System

All code relating directly to the ES was placed in the `es` package of the project. A UML diagram of the `es` package can be seen in Figure 11 of the Appendix.

The ES layer is based around the `ExpertSystem` Java class. This class has the following fields:

readyRules   `Set` of `Rules` that have not been activated yet.

activeRules   `Set` of active `Rules`.

facts   `Set` of active `Facts`.

recommendations `Set` of active `Recommendations`.

The most important method in the `ExpertSystem` is `think()`, which chooses either `thinkForwards()`, `thinkBackwards()`, or `thinkLambda()`. These methods implement the functionality described in Section 2.

Without a parameter, the `think()` method runs to natural quiescence. This can be seen in Listing 6.

Listing 6: Thinking until quiescence in the ES.

```
public Set<Tag> think() {
    Set<Tag> allActivatedTags = new HashSet<>();
    Set<Tag> activatedTags;
    do {
        activatedTags = thinkCycle();
        allActivatedTags.addAll(activatedTags);
    } while (!activatedTags.isEmpty());
    Set<Tag> activatedRecommendations = new
        HashSet<>();
    for (Tag tag : allActivatedTags) {
        if (tag.isRecommendation())
            activatedRecommendations.add(tag);
    }
    return activatedRecommendations;
}
```

There is also an overloaded version of `think()` that functions similarly to that of the `KnowledgeNodeNetwork`. This can be seen in Listing 7

Listing 7: Thinking with a fixed number of cycles in the ES.

```
public Set<Tag> think(int numberOfCycles) {
    Set<Tag> allActivatedTags = new HashSet<>();
    for (int i = 0; i < numberOfCycles; i++) {
        Set<Tag> activatedTags = thinkCycle();
        if (activatedTags.isEmpty())
            break;
        allActivatedTags.addAll(activatedTags);
    }
    Set<Tag> activatedRecommendations = new
        HashSet<>();
    for (Tag tag : allActivatedTags) {
        if (tag.isRecommendation())
            activatedRecommendations.add(tag);
    }
    return activatedRecommendations;
}
```

Both versions of `think()` call a method named `thinkCycle()` representing a single cycle of thinking. The code for this method can be seen in Listing 8

Listing 8: A single cycle of thinking in the ES.

```
private Set<Tag> thinkCycle() {
    Set<Tag> activatedTags = new HashSet<>();
    Set<Rule> pendingActivatedRules = new HashSet<>();
    for (Rule rule : readyRules) {
        for (Tag outputTag : rule.outputTags) {
            if ((!facts.contains(outputTag) ||
                !recommendations.contains(outputTag)))
                {
                boolean shouldActivate = true;
```

```
            for (Tag condition : rule.inputTags) {
                if (!facts.contains(condition) &&
                    !recommendations.contains(condition))
                    {
                    shouldActivate = false;
                    break;
                }
            }
            if (shouldActivate)
                pendingActivatedRules.add(rule);
        }
    }
}
for (Rule rule : pendingActivatedRules) {
    readyRules.remove(rule);
    activeRules.add(rule);
    for (Tag outputTag : rule.outputTags) {
        activatedTags.add(outputTag);
        if (outputTag.isRecommendation())
            recommendations.add((Recommendation)
            outputTag);
        else if (outputTag.isFact())
            facts.add((Fact) outputTag);
    }
}
return activatedTags;
}
```

5.4   Testing

All tests on the system were conducted using
the TestNG framework in Java, which provides
a simple and intuitive way to create assertions
in tests. These tests were placed in the `test`
package of the project. A UML diagram of the
`test` package can be seen in Figure 12 of the
Appendix.

5.4.1   Unit Tests

Unit tests were generated for the KNN and
ES to test every single method on its own.

5.4.2   Integration Tests

Integration tests were made to test the com-
bined functionality of the KNN and ES.

The test setup for the `testKNN()` method can
be seen in Figure 8.

The test setup for the `testES()` method can
be seen in Table 1.

The `testKNNandES()` method has the same
setup as `testKNN()` in Figure 8, except the out-
put active Tags are passed on to the ES. The
resulting setup and activation can be seen in Ta-
ble 2.

5.5   Documentation

Proper documentation of the source code is
very important. This is to ensure that anyone
wanting to work with the code which was de-
signed has an easy time doing so. Also, for
anyone wanting to understand how the system
works, documentation is essential. This doc-
umentation was achieved through Javadoc and
UML diagrams.

5.5.1   Javadoc

Extensive Javadoc was generated for the
entire code base. This can be found here:
`http://cs.mcgill.ca/~sstapp/prometheus/`
`index.html`. The functionality of every single
class and method was described in detail.

5.5.2   UML

UML diagrams of every package in the code
base were created.

6   Plan for Next Semester

The plan for next semester is to finalize the
two layers that were started this semester, and
to test them on the robots available in Prof. Vy-
bihal's lab.

6.1   Finalization

6.1.1   Knowledge Node Network

Many complex features of the KNN layer are
still to be implemented, such as backwards and
lambda thinking, confidence, and aging.

Backwards thinking can be implemented us-
ing a background thread in Java, constantly run-
ning in the background at some rate.

Some thought still needs to be put into the
specifics on implementation of lambda think-
ing. Most importantly, the use cases of lambda
thinking will have to be well understood and de-
scribed.

Confidence values will probably be associated
with every individual `Tag`, as well as in an abso-
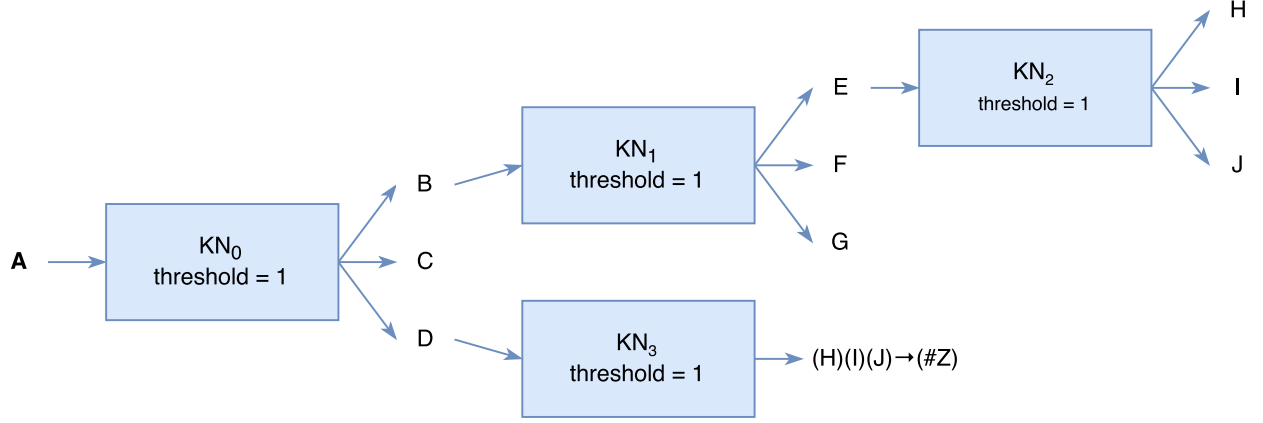lute sense in the KNN. Activation through the

Figure 8: Setup for the `testKNN()` method. Every node has a `threshold` value of 1 to simplify the activation.

Ready Rules	Active Rules	Active Facts	Active Recommendations
$(A)(B) \rightarrow (D)$ $(D)(B) \rightarrow (E)$ $(D)(E) \rightarrow (F)$ $(G)(A) \rightarrow (H)$ $(\#X)(\#Y) \rightarrow (\#Z)$		$(A), (B)$	$(\#X), (\#Y)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$(G)(A) \rightarrow (H)$	$(A)(B) \rightarrow (D)$ $(D)(B) \rightarrow (E)$ $(D)(E) \rightarrow (F)$ $(\#X)(\#Y) \rightarrow (\#Z)$	$(A), (B),$ $(D), (E)$ (F)	$(\#X), (\#Y), (\#Z)$

Table 1: Test setup for `testES()`. Middle activation steps omitted.

Ready Rules	Active Rules	Active Facts	Active Recommendations
$(H)(I)(J) \rightarrow (\#Z)$		$(B), (C), (D)$ $(E), (F), (G)$ $(H), (I), (J)$	
	$(H)(I)(J) \rightarrow (\#Z)$	$(B), (C), (D)$ $(E), (F), (G)$ $(H), (I), (J)$	$(\#Z)$

Table 2: Test setup and activation for the ES portion of `testKNNandES()`.

layers of the KNN can theoretically then reduce the absolute confidence that the output `Tags` are true, since at every step the confidence should be multiplied with the previous value.

The intuitive implementation of aging is to constantly increment some age value associated with every Knowledge Node, and discard nodes whose age is greater than some threshold. The age would be reset when the node is excited. This would however be unnecessarily computationally intensive, requiring some kind of constant updating of every single node. A more efficient way of doing it would be to save a timestamp for every Knowledge Node when it is excited, and if the KNN attempts to excite a Knowledge Node whose previous timestamp value is too far into the past, that node is instead deleted.

Activation of Knowledge Nodes with sigmoid functions can also be looked into. This can be easily implemented using a cache of known sigmoid values.

Also, more thought will have to be put into the design of the KNN to allow cyclic graphs, representing recursive memories in the human brain.

6.1.2 Expert System

Some features of the ES layer are still to be implemented, such as more complex Fact checking.

6.2 Integration

One very important task left to be done is to integrate the two layers described in this report (ES and KNN) with the other layers developed separately (NN and META). Ideally, the layers should be able to work together, but there will surely be some conflicts at the interface of the layers. These will have to be resolved when the time comes.

6.3 Testing

6.3.1 Simulation

Once all the layers are functioning together, they can be tested. The first and easiest way to test would be in a simulated environment. One simulator that may be used is Simbad, which is a Java 3D robot simulator [5]. This can allow for some early debugging and fixes.

6.3.2 Physical

One the simulation testing is completed and working properly, the system can be tested in the lab. Prof. Vybihal's lab has multiple robots with ultrasonic sensors, and these will be the test subjects of this phase.

7 Impact on Society and the Environment

7.1 Use of Non-renewable Resources

As a purely software-oriented project, there are no physical materials needed to construct this system.

7.2 Environmental Benefits

The system could be used as a tool to control robots in dangerous environments such as nuclear plants after a radioactive disaster. Indeed, the system could coordinate robots to help contain the damage faster than humans could, thus limiting the risk on the environment.

7.3 Safety and Risk

It is critical that, once this system is completed, it is used in an ethical way, and for the right purposes.

One example of use that may cause ethical concern is in a military setting, where an AI system like the one described here could be used in a battlefield in place of soldiers.

In the very long term, there are concerns with the possibility that an AI system might achieve intelligence and awareness close to a human. If

such an AI were to obtain "consciousness" in its own way, should that entity be entitled to its own rights, like humans do?

7.4 Benefits to Society

Going back to the example use case in a radioactive disaster, the system could be used to send robots in an area that would otherwise be very dangerous for humans. This would therefore help prevent the unnecessary loss of life in cleaning up these leaks.

8 Conclusion

This semester, prototypes of the Expert System (ES) and Knowledge Node Network (KNN) layers of the AI model were completed in Java. These prototypes were tested individually and together, with positive results.

The main goal for next semester is to finalize the entire system, implementing more complex features that were omitted for this prototype stage.
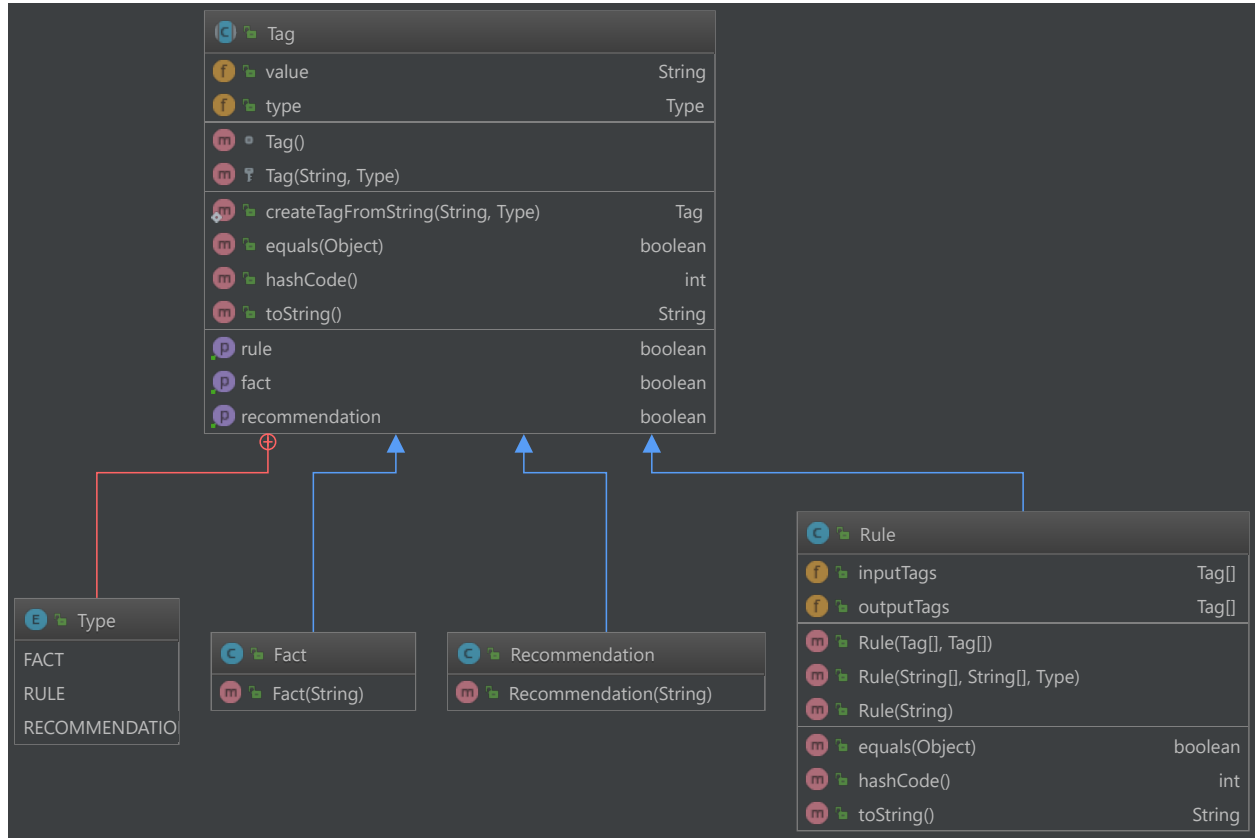
A   UML Diagrams
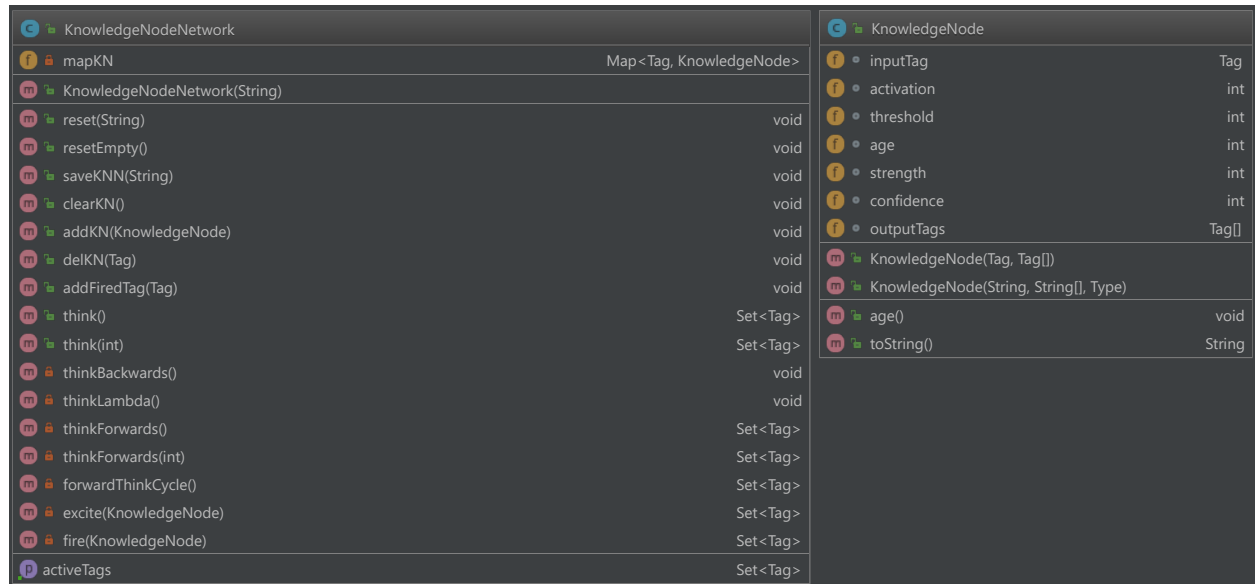

Figure 9: UML diagram of the `tags` *package.*
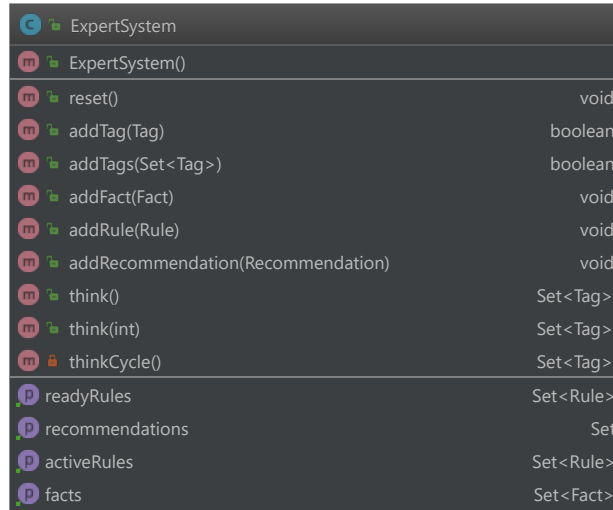

Figure 10: UML diagram of the `knn` *package.*

Figure 11: UML diagram of the es package.

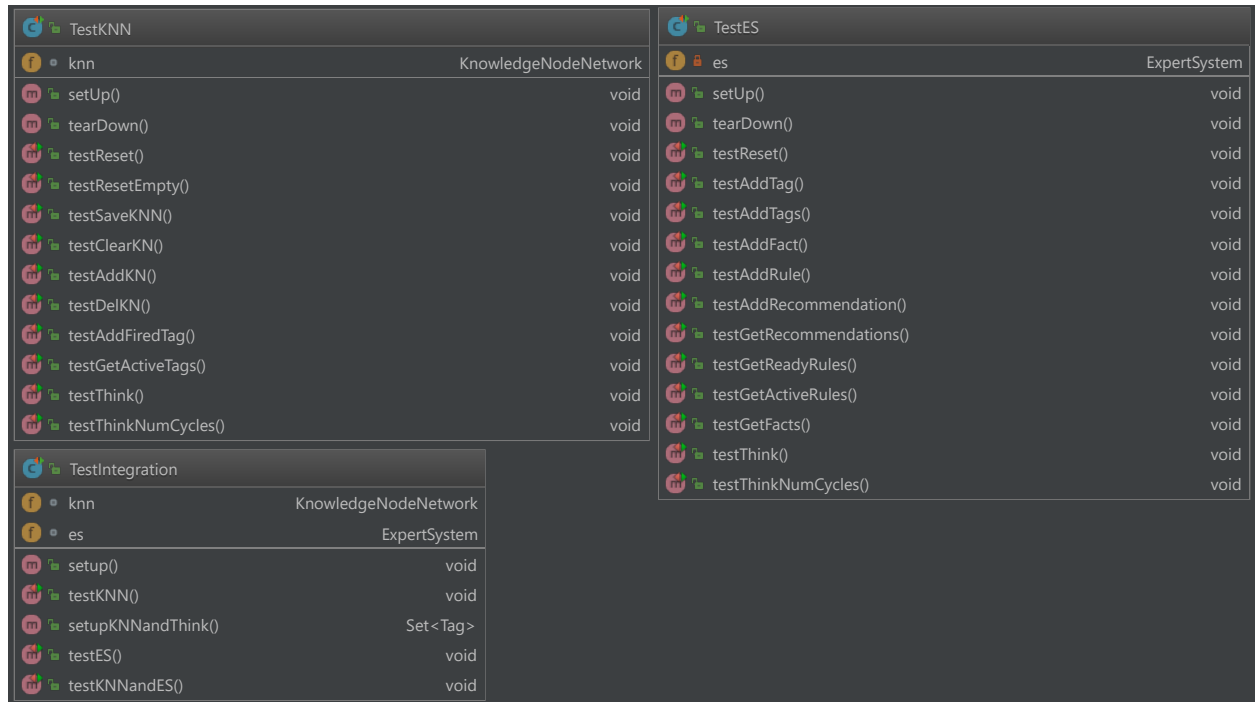Figure 12: UML diagram of the test package.

References

[1] J. Vybihal, "Perceptron learning," 2016.

[2] ——, "Knowledge nodes," 2017.

[3] J. Vybihal and T. R. Shultz, "Search in analogical reasoning," 1990.

[4] J. Vybihal, "Expert systems," 2016.

[5] "Simbad 3d robot simulator," http://simbad.sourceforge.net/index.php, (Accessed on 03/27/2017).